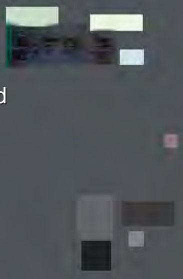
Mini-Computers

Yaniv Frishman

Origin of the Name

- Mini-minor
- Mini-skirt
- DEC Europe sales person

Comparison with Main-Frames



Mainframes:
- Leased (rented)
- HW modifications not allowed
- HW & SW sold together

Minicomputers:
- Sold, not rented
- Much cheaper
- Open spec, modifications encouraged
- OEM model

Defining Facets

1. Architecture
2. Packaging
3. Role of third-party in apps development
4. Price
5. Financing

Word Length

- Typical 1960s IBM Mainframes used 36-bits
  - 10 decimal digits
- Manufacturers assumed less would not be enough
- Shorter instructions reduce addressing abilities

Short Word Workarounds

- More complex instructions
  - Address stored in a different register
- Using double precision math (add with carry)

DEC PDP-1

- Included many of the TX-0 features
- Designed from the ground using transistors
- Capable of 100,000 additions per second
- Core memory of 4K words
- About 50 machines sold

PDP-1 I/O - DMA

- I/O flows directly from the device to memory
- Multiple interrupts
- HW support for correct handling (priority)
- Cheap: a single IBM I/O channel cost more than a complete PDP-1 ($120K)

PDP-1 Space Invaders

- Developed by MIT students
- Using 1024x1204 CRT
- Planet Map
- Simulated Gravity

PDP-1 Space Invaders

Applet demo – see

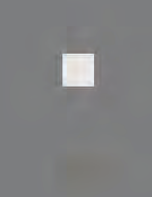
http://spacewar.oversigma.com/

Business Model / Interaction with Customers

- IBM rented its computers
  - Modifications need IBM approval
- DEC model:
  - PDP-1 was sold, not leased

DEC openness

- Encouraged modification by the customers
- Published detailed specs on cheap paper
- Didn't develop specialized HW and SW



DEC PDP-8

- 12 bit word length
- 50,000 computers installed
- Successful - performance, storage, packaging, price
- Improvements in logic and core memory reduced the cycle time to 1.6 microsec.

PDP-8 Addressing

- 7bits were used for the address field (small)
- Memory: 32 blocks of 128 words (4KW)
- Access across a block achieved by setting bits in the opcode

PDP8 packaging

- Constructed from a series of compact modules
  - Each performed a specific function
- Modules plugged into a chassis
- Wire wrap connection
- Small: embedded in other equipment
- 8 cubic feet (volume), 250 pounds

PDP8 pricing

- Very cheap: $18K
- Price dropped to $10K after a few years
- Price shocked the industry, many orders
- Once again estimates of the computer market size were proved incorrect

PDP8 programming

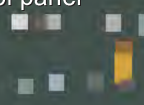
- Limited memory prevented high-level programming
- Simple, easy to understand computer
- Gave rise to OEMs

OEMs

- OEM: a separate company that bought minicomoputers, added specialized HW & SW and sold them under their own label
- Relieved DEC of developing specialized SW
- Ranged across all segment of society:
  - Medical instrumentation
  - Small business records
  - Industrial controllers

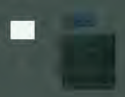
OEM example

- LS-8 used to operate theatrical stage lighting
- Cited as a key element in the success of the Broadway hit “A Chorus Line”
- Contained a PDP-8A, introduced 1975
- Application specific control panel



DEC PDP-10

- One of the most influential computers
- The machine that made time-sharing common
  - The basis of the ARPANET
  - The platform upon which many applications were first developed:
    - EMACS
    - TeX
    - ISPELL
    - Kermit

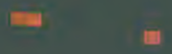


PDP-10 Architecture

- An improved HW implementation of the PDP-6
- Shared the same 36-bit word length
- Slightly extended the instruction set

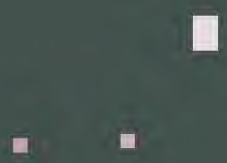
KA-10

- The original PDP-10 processor was the KA10



Wire Wrap Backplane

- Backplanes wire wrapped, semi-automated manufacturing process



PDP-10 Memory Management

- KA10: maximum main memory of 256 Kwords
- Management consisted of 2 sets of protection & relocation registers -"base and bounds" registers
- This allowed separate read-only shareable code segment and read-write data/stack segment

PDP-11

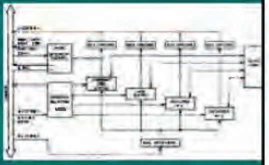
- A successor to the PDP-8
- Was easier to program than its predecessors
- the world's most successful family of minicomputers
- Was replaced by VAX-11

PDP-11 Instruction Set

- A highly-orthogonal instruction set:
  Operation ; operand access mode
- Any addressing mode would work with any operation

PDP-11 I/O

- New architecture: no dedicated I/O bus
- It had only a memory bus, the Unibus
- I/O devices are memory mapped
  - No need for special I/O instructions
- Four levels of interrupts
- Interrupting device puts its address on the bus



DEC VAX Minicomputers

VAX

- An extension of the PDP-11, with mainframe performance
- Design began 1974
- VAX = **V**irtual **A**ddress e**X**tension (of PDP-11)
- VAX was able to execute PDP-11 instructions in a 32 instead of 16-bit address space
- PDP-11 compatibility bit that was later dropped

VAX - a virtual memory computer

- Making small but fast main memory seem to be bigger by swapping data from a slower disk
- Overall performance not seriously degraded
- User is not aware that swapping is done
- The VAX provided a 32-bit virtual address
- Memory divided into pages

VAX Instruction Set

- Sixteen general purpose 32-bit registers
- Rich set of 250 instructions
- Two and three operand formats
- Register or memory operand in most instructions
- The quintessential CISC processing architecture

VAX Commercial Aspects

- Successful: 100,000 units in 10 years
- General purpose computer that came with standard languages and SW
- Biggest impact in engineering and science
- Prices started at $120K
- Cheap enough to serve a division in the aerospace, automotive, chemical firms

1977

MIPS

- The performance of VAX 11/780 became known as MIPS (million instructions per second)
- Later used as a benchmark of performance

A Brief VAX Timeline

1975

VAX Kickoff

- The VAX Architecture Committee began work on a computer with 32-bit architecture
- Image: inside of VAX-11/780

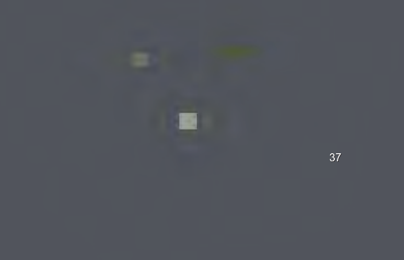
1977

First VAX

- VAX-11/780 introduced

VAX OS

- V1.0 of the VMS operating system ships
- FORTRAN IV and DECnet, a 64 megabyte memory limit, an event driven priority scheduler, process swapper, process deletion/creation/control



LSI VAX

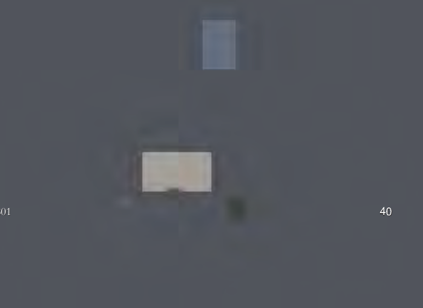
- Introduction of the VAX-11/750
- The industry's first Large Scale Integration (LSI) 32-bit minicomputer

ECL VAX

- VAX 8600: the first VAX implementation in ECL technology

“Personal “ VAX

- VAX station I.
- A powerful, single-user computing system supporting the professional user



VLSI VAX

- MicroVAX: VAX on a VLSI chip

VAX Station Best-Seller

- The VAXStation 2000 is introduced.
- DIGITAL's first workstation with a cost of less than $5,000
- Became the highest volume workstation in the industry

Last VAX

- VAX 7000 series, DIGITAL's most powerful VAX system, field-upgradeable to the Alpha 64-bit processor

Summary

- Computer class between big-iron mainframes and personal computers
- The basis for nowadays servers (VAX)
- Continuing trend of bringing the computer to more people
- Introduced important architectural aspects

References

- WikiPedia
- A History of MTS - http://www.clock.org/~jss/work/mts/30years.html
- Batch VS TS – Bruce Lakin, NJ EDU Computer Network 1980.
- *A History of Modern Computing by Paul E Ceruzzi.*
- The McGraw-Hill Computer Handbook
- IEEE Annals of the History of Computing
- 234120 - Operating Systems – CS Technion

References

- House of VAX: http://www.mcmanis.com/chuck/computers/vaxen/
- VAX timeline: http://research.microsoft.com/~gbell/digital/timeline/32-bit.htm
- PDP-10: http://www.columbia.edu/acis/history/pdp10.html
- http://webcourse.cs.technion.ac.il/234118/
- www.computerhistory.org

References

- http://history.sandiego.edu/GEN/recording/computer1.html
- http://www.old-computers.com/history/detail.asp?n=58&t=4
- http://www.poetproductions.com
- http://www.ibiblio.org/pub/academic/computer-science/history/pictures/pdp1_1.jpg
- http://spacewar.oversigma.com/

Architectural Evolution in DEC’s 18b Computers

Bob Supnik, 26-Jul-2003

Abstract

DEC built five 18b computer systems: the PDP-1, PDP-4, PDP-7, PDP-9, and PDP-15. This paper documents the architectural changes that occurred over the lifetime of the 18b systems and analyses the benefits and tradeoffs of the changes made.

Introduction

From 1961 to 1975, Digital Equipment Corporation (DEC) built five 18b computer systems: the PDP-1, PDP-4, PDP-7, PDP-9, and PDP-15 (see table below). Each system differed from its predecessors, sometimes in major ways representing significant architectural breaks, and sometimes in minor ways representing new features or incompatibilities. The architectural evolution of these systems demonstrates how DEC’s ideas about architectural versus implementation complexity, I/O structures, and system features evolved over the period of a decade.

	PDP-1	PDP-4	PDP-7	PDP-9	PDP-15
First ship	Nov 1960	Jul 1962	Dec 1964	Aug 1966	May 1970
Number built	50	45	120	445	790
Memory cycle	5usec	8usec	1.75usec	1usec	0.8usec
Base price	$120K	$65.5K	$45K	$25K	$19.8K

Reproduced from Computer Engineering: A DEC View Of Hardware Systems Design

The PDP-1

The PDP-1 was DEC’s first computer system. Introduced in 1960, the PDP-1 reflected ideas from Lincoln Labs’ TX-2 project as well as the existing capabilities of DEC’s module logic family. It was implemented in 5Mhz logic.

Arithmetic System

The PDP-1 was a 1’s complement arithmetic machine. In 1’s complement arithmetic, negative numbers are represented by the bit-for-bit inversion of their positive counterparts:

```
+1    =         000001
-1    =         777776

+4    =         000004
-4    =         777773
```

One’s complement arithmetic has two problems. First, zero has two representations, +0 and -0:

```
+0    =         000000
-0    =         777777
```

Second, addition of negative numbers requires an “end around carry” from the high order position to the low order position:

```
      -1    =             777776
      -1    =             777776
      --                ---------
      sum               1 777774
                        |----->1
      -2    =             777775
```

The PDP-1 tried to solve the zero-representation problem by guaranteeing that arithmetic operations never produced –0. To do this, it performed an extra logic step during addition, checking the result for –0 and converting it to 0. However, the PDP-1 performed subtraction by complementing the AC, adding the memory operand, and recomplementing the result. The recomplementation step occurred in the same time slot as the –0 detect during add. As a result, subtract had one special case: -0 – (+0) yielded –0.

Character Sets

The PDP-1’s first console typewriter was a Friden Flexowriter. (Production units used an IBM Soroban B typewriter.) The console’s six bit character set was called FIODEC, which stood for **F**riden **I**nput **O**utput for **D**igital **E**quipment **C**orporation. This code included both upper and lower case letters, using shift characters to move between sets. The PDP-1’s line printer used Hollerith (BCD) coding. FIODEC and Hollerith had common encodings for letters but not for symbols, requiring character conversions throughout the software.

Instruction Set Architecture

The PDP-1’s visible state included the following registers and capabilities:

AC<0:17>	accumulator
IO<0:17>	I/O register
OV	overflow flag
PC<0:11>	program counter
EPC<0:3>	extended program counter (if memory > 4K)
EXTM	extend mode
PF<1:6>	program flags
SS<1:6>	sense switches
TW<0:17>	test word (front panel switches)
IOSTA<0:17>	I/O status

In addition, the PDP-1 had non-observable state in the I/O system for I/O timing (see below).

The PDP-1 had 32 opcodes and implemented six instruction formats: memory reference, skip, shift, operate, I/O, and load immediate. The memory reference format was:

```
  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|      op      |in|              address              | mem reference
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

<0:4> <5>   mnemonic    action
```

```
 00
 02          AND         AC = AC & M[MA]
 04          IOR         AC = AC | M[MA]
 06          XOR         AC = AC ^ M[MA]
 10          XCT         M[MA] is executed as an instruction
 12          JFD         change fields, PC = MA
 14
 16    0     CAL         M[100] = AC, AC = PC, PC = 101
 16    1     JDA         M[MA] = AC, AC = PC, PC = MA + 1
 20          LAC         AC = M[MA]
 22          LIO         IO = M[MA]
 24          DAC         M[MA] = AC
 26          DAP         M[MA]<6:17> = AC<6:17>
 30          DIP         M[MA]<0:5> = AC<0:5>
 32          DIO         M[MA] = IO
 34          DZM         M[MA] = 0
 36
 40          ADD         AC = AC + M[MA]
 42          SUB         AC = AC - M[MA]
 44          IDX         AC = M[MA] = M[MA] + 1
 46          ISP         AC = M[MA] = M[MA] + 1, skip if AC >= 0
 50          SAD         skip if AC != M[MA]
 52          SAS         skip if AC == M[MA]
 54          MUL         AC'IO = AC * M[MA]
 56          DIV         AC, IO = AC'IO / M[MA]
 60          JMP         PC = MA
 62          JSP         AC = PC, PC = MA
```

The skip format was:

```
  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| 1  1  0  1  0|  |  |  |  |  |  |  |  |  |  |  |  |  | skip
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
                 |     |  |  |  |  | \______/ \______/
                 |     |  |  |  |  |     |        |
                 |     |  |  |  |  |     |        +---- program flags
                 |     |  |  |  |  |     +------------- sense switches
                 |     |  |  |  |  +------------------- AC == 0
                 |     |  |  |  +---------------------- AC >= 0
                 |     |  |  +------------------------- AC < 0
                 |     |  +---------------------------- OV == 0
                 |     +------------------------------- IO >= 0
                 +------------------------------------- invert skip
```

The shift format was:

```
  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| 1  1  0  1  1| subopcode |      encoded count       | shift
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
                 |  | \___/
                 |  |   |
                 |  |   +------------------------------ 1=AC,2=IO,
                 |  |                                   3=both
                 |  +---------------------------------- rotate/shift
                 +------------------------------------- right/left
```

The shift count was the number of 1's in bits <9:17>.

The load immediate format was:

```
  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| 1  1  1  0  0| S|           immediate               | LAW
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
                 |
                 +----- if S = 0, AC = IR<6:17>
                        else AC = ~IR<6:17>
```

The I/O transfer format was:

```
  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| 1  1  1  0  1| W| C|   subopcode  |      device     | I/O transfer
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

The operate format was:

```
  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| 1  1  1  1  1|  |  |  |  |  |  |  |  |  |  |  |  |  | operate
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
                    |  |  |  |  |  |        | \______/
                    |  |  |  |  |  |        |     |
                    |  |  |  |  |  |        |     +---- PF select
                    |  |  |  |  |  |        +---------- clear/set PF
                    |  |  |  |  |  | \_________/
                    |  |  |  |  |  |      |
                    |  |  |  |  |  |      +------------ change field
                    |  |  |  |  |  +------------------- or PC
                    |  |  |  |  +---------------------- clear AC
                    |  |  |  +------------------------- halt
                    |  |  +---------------------------- CMA
                    |  +------------------------------- or TW
                    +---------------------------------- clear IO
```

There are significant discrepancies in the extent PDP-1 documentation about memory expansion options. The original 1960 User Handbook (F15) didn't mention any. The 1961 Handbook (F15B) described two, the Type 13 and Type 14. The 1962 and 1963 Handbooks (F15C and F15D, respectively), and the Maintenance Manual, described only one, the Type 15. This option expanded memory to 64K words. The address space was divided into sixteen 4K word fields. An instruction could directly address, via its 12b address, the entire current field. If extend mode was off, indirect addresses accessed the current field, and multi-level indirect addressing was enabled; if on, indirect addresses could access all 64K, and indirect addressing was single level. The state of extend mode was captured by subroutine calls and sequence breaks, and extend mode was cleared at the start of a sequence break.

BBN built a custom memory manager for its PDP-1 timesharing system.

I/O System

The PDP-1's I/O system offered multiple modes for I/O instructions, including synchronous waiting, timed waiting, asynchronous, and sequence break (interrupt) driven. This multiplicity made the I/O system complex and redundant.

I/O operations were initiated by a single instruction, Input/Output Transfer (IOT). Bits<12:17> addressed a particular device; bits <7:11> provided additional control or opcode bits. Bits<5:6> specified the mode for the I/O transfer:

<5:6>	mode
00	asynchronous - no wait, no device completion pulse
01	timed wait - no wait, device completion pulse
10	synchronous - wait for completion
11	not used - wait, no completion pulse (hung the system if <12:17> != 0)

In synchronous wait, the CPU effectively stalled until the I/O operation completed. If synchronous wait was not specified, three different mechanisms were available for I/O completion:

- Timed wait. Execution proceeded. Eventually, the CPU issued a wait instruction. The CPU then stalled until the I/O operation completed and the device issued a completion pulse.
- Polled wait. Execution proceeded. The CPU monitored the device's flag in the I/O status word until the I/O operation completed.
- Sequence break driven. Execution proceeded. When the I/O operation completed, a sequence break (interrupt) occurred, signaling I/O done.

The IOT wait mechanism was implemented by clearing the I/O command flag (which allowed I/O instructions to execute), decrementing the PC, and re-executing the IOT that specified the wait. To allow IOT's in interrupt routines, the CPU had to remember that a wait was in progress, clear the wait for the interrupt level IOT, and restore the wait afterwards. IOT's in interrupt routines could not specify waiting.

The sequence break mechanism recorded break requests in a single pulse sensitive flip flop. Thus, like the PDP-11 but unlike the other 18b systems, break requests were independent of the device completion flags. If the sequence break system was enabled, and a break request occurred, the CPU automatically stored the state of the machine and initiated a new program by:

- storing AC in location 0
- storing EPC and PC, plus overflow and extend mode, in location 1
- storing IO in location 2
- clearing overflow and extend mode
- setting the PC to 3
- setting the sequence break in progress flag

The sequence break in progress flag blocked further breaks.

The end of the break was recognized when the CPU decoded a JMP I 1 (from field 0 in a multi-field system) while the sequence break system was enabled. At that point, the CPU automatically restored the state of the system by:

- temporarily turning on extend mode
- obtaining the new PC from location 1
- restoring the original values of overflow and extend mode
- clearing sequence-break-in-progress

A CPU option expanded the standard sequence break system from one channel to sixteen. Each channel was a unique priority level and had a dedicated four location memory block (0 – 3 for the highest priority channel, 4 – 7 for the next, etc.). The first three locations of the block were used to store AC, PC, and IO when a break occurred; the PC was then set to point to the fourth location.

Software

The PDP-1 featured some notable software offerings, including an interactive editor (called Expensive Typewriter), a macro assembler, a Lisp interpreter, and the world's first computer video game, Spacewar. (Sources to Lisp and Spacewar are still available on the Internet.)

The PDP-4

The PDP-4 was intended to be substantially lower cost than the PDP-1. Part of the cost reduction was achieved by using slower and less expensive logic (500Khz instead of 5Mhz), but part was achieved by simplifying the system and reducing the number of gates. Thus, the PDP-4 (and its closely related successors, the PDP-7 and PDP-9) simplified the architecture of the PDP-1 along multiple dimensions.

Arithmetic Systems

The PDP-4 introduced two's complement arithmetic in parallel with the PDP-1's one's complement arithmetic. Two's complement arithmetic eliminated the need for -0 detection and made implementation of multi-precision arithmetic much easier. However, 1's complement capability was not dropped; indeed, it remained the predominant arithmetic system, as reflected in future architectural extensions such as the EAE. Thus, the PDP-4 still needed end around carry propagation, as well as 1's complement overflow detection. The result was greater, rather than lesser complexity, in the hardware, and loss of valuable opcode space in the architecture. Gordon Bell commented that the retention of 1's complement arithmetic was, simply, "a mistake". By the PDP-5, it had vanished from DEC's architectures.

Character Sets

The PDP-4's console typewriter was an ASR-28 Teletype. Its five bit character code was called Baudot. It supported only upper case letters and required shift characters to get from letters to figures and back again. The line printer was unchanged and continued to use Hollerith coding.

Instruction Set Architecture

The PDP-4 and its follows-ons reduced the amount of visible state in the CPU. Specifically,

register	PDP-1	PDP-4,-7,-9
AC	arithmetic register	same, plus I/O register

IO	I/O register	removed (MQ with EAE option)
OV	overflow indicator	replaced by Link register
PF	program flags	removed
SS	sense switches	removed
TW	test word	front panel switches
EXTM	extend mode	same
IOSTA	IO flags	same

The register changes simplified the logic implementation. The L was essentially the 19th bit of the AC, rather than a special flag. The AC no longer implemented -0 detection. I/O now used the existing access paths to the AC rather than separate paths to an IO register. The elimination of the program flags, and the sense switches, was pure gain.

The PDP-4 halved the number of instructions, from 32 to 16, and reduced the number of instruction formats from 6 to 4. The memory reference format was:

```
  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|     op    |in|                address               | mem reference
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

The I/O transfer format was:

```
  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| 1  1  1  0  0  0|      device     | sdv |cl|  pulse | I/O transfer
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

The operate format was:

```
  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| 1  1  1  1  0|  |  |  |  |  |  |  |  |  |  |  |  |  | operate
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
                 |  |  |  |  |  |  |  |  |  |  |  |  |
                 |  |  |  |  |  |  |  |  |  |  |  |  +- CMA (3)
                 |  |  |  |  |  |  |  |  |  |  |  +---- CML (3)
                 |  |  |  |  |  |  |  |  |  |  +------- OAS (3)
                 |  |  |  |  |  |  |  |  |  +---------- RAL (3)
                 |  |  |  |  |  |  |  |  +------------- RAR (3)
                 |  |  |  |  |  |  |  +---------------- HLT (4)
                 |  |  |  |  |  |  +------------------- SMA (1)
                 |  |  |  |  |  +---------------------- SZA (1)
                 |  |  |  |  +------------------------- SNL (1)
                 |  |  |  +---------------------------- inv skip (1)
                 |  |  +------------------------------- rotate two (2)
                 |  +---------------------------------- CLL (2)
                 +------------------------------------- CLA (2)
```

The immediate format was:

```
  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| 1  1  1  1  1|            immediate                 | LAW
```

```
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

The following table shows the reduction in instruction count between the PDP-1 and the PDP-4:

PDP-1 instruction	PDP-4 instruction
AND	AND
IOR	removed
XOR	XOR
LAC	LAC
DAC	DAC
DZM	DZM
DIP	removed
DAP	removed
LIO	removed
DIO	removed
ADD	ADD; L used in place of overflow
SUB	removed
MUL	(EAE option)
DIV	(EAE option)
not present	TAD (2's complement add)
IDX	removed
ISP	ISZ
XCT	XCT
SAD	SAD
SAS	removed
CAL	CAL
JDA	JMS
JSP	removed
JMP	JMP
skips	OPR skips
operate	OPR operates
shifts	(EAE option)
LAW	LAW
IOT	IOT

Beyond the reduction in instruction count, the PDP-4's instruction set required less logic to implement.

- Instructions were encoded to minimize logic. For example, all instructions with IR<0:1> = 00 (CAL, DAC, JMS, DZM) did not read a memory operand. All instructions with IR<0:1> = 11 (JMP, EAE, IOT, OPR/LAW) were single cycle.
- ISZ (replacing IDX and ISP) did not modify the AC. By using 2's complement arithmetic, it did not need to detect -0.
- JMS (replacing JDA and JSP) did not modify the AC. This eliminated the transfer path from the PC to the AC. JMS (and interrupts) saved PC and L, and in later systems, the memory extend and memory protection flags.
- LAW did not mask or modify the address but instead copied the entire instruction to AC.
- OPR no longer guaranteed conflict-free execution of any combination of bits.

Finally, indirect addressing was simplified by the elimination of multi-level indirection.

The PDP-4 replaced the PDP-1's multiply, divide, and multi-bit shifts with an option, the Extended Arithmetic Element (EAE). The EAE added a second 18b arithmetic register, the MQ, and a shift/multiply/divide instruction. The EAE instruction was microprogrammed and could implement a wide variety of unsigned and signed (one's complement) operations:

```
  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| 1  1  0  1|  |  |  |  |  |  |  |  |  |  |  |  |  |  | EAE
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
              |  |  |  |  |  |  |  |  |  |  |  |  |  |
              |  |  |  |  |  |  |  |  |  |  |  |  |  +- or SC (3)
              |  |  |  |  |  |  |  |  |  |  |  |  +---- or MQ (3)
              |  |  |  |  |  |  |  |  |  |  |  +------- compl MQ (3)
              |  |  |  |  |  |  |  |  \______________/
              |  |  |  |  |  |  |  |         |
              |  |  |  |  |  \_____/         +--------- shift count
              |  |  |  |  |     |
              |  |  |  |  |     +---------------------- EAE cmd (3)
              |  |  |  |  +---------------------------- clear AC (2)
              |  |  |  +------------------------------- or AC (2)
              |  |  +---------------------------------- load sign (1)
              |  +------------------------------------- clear MQ (1)
              +---------------------------------------- load link (1)
```

The EAE architecture remained unchanged in the PDP-7, PDP-9 and PDP-15.

The PDP-4 included an extended addressing option (Type 16); two of the surviving PDP-4's in the 1972 census have more than 8KW of memory. No documentation has yet been found on this option, but it's reasonable to assume that it was the same as the PDP-7's. If that is true, the PDP-4's extended memory model was essentially the same as the PDP-1's, with 13 direct address bits instead of 12. Addressable memory was divided into four 32K word banks. Direct addresses always referenced the current memory bank; indirect addresses accessed either the current memory bank or all of memory, depending on the extend mode flag. .) As on the PDP-1, subroutine calls and interrupts saved the state of extend mode automatically.

In all, the architectural tradeoffs in the PDP-4 substantially reduced control logic at the cost of complete software incompatibility with the PDP-1. There were also a few oversights; in particular, the lack of a "complement and increment" operate (present in the PDP-5) made two's complement subtract an instruction longer. The PDP-15 finally corrected this oversight.

The PDP-4 (and the PDP-5) introduced a new feature, the concept of "auto-index" memory locations, that is, locations which, when used as indirect addresses, incremented before use. This feature allowed efficient traversal of linear data structures and made the IDX and DAP instructions unnecessary.

I/O System

The I/O system was pruned even more dramatically than the CPU. Synchronous waits and timed waits were dropped. Instead, only two mechanisms were supported: polled waits and interrupts. Further, the two mechanisms were integrated by having the device flag for polling be the triggering mechanism for device interrupts. Finally, polled waiting was implemented more efficiently by allowing devices increment the PC (skip) in response to an IO instruction. The PDP-5 also used this I/O paradigm, and it was retained throughout the life of the 12b and 18b families.

In the PDP-4, an ideal I/O device had one flag representing the state of an I/O operation. This flag was cleared when the device initiated I/O; it was set when the device completed I/O. For example, in the paper tape reader, the reader flag was cleared by a request to read a character or by explicit command, and set when the character was in the I/O buffer.

Interrupts (as sequence breaks were now called) were simplified, and control was made explicit rather than implicit.

Function	PDP-1	PDP-4
interrupt request	request flip-flop	logical or of device flags
interrupt block	request in progress flop	interrupts turned off
interrupt action	save AC	--
	save PC + flags	save PC + flags
	save IO	--
	clear OV	--
	clear extend mode	{clear extend mode}
	set break in progress	turn off interrupts
	set PC = 3	set PC = 1
interrupt complete	monitor for JMP I 1	turn on interrupts, one cycle delay to allow for JMP I 0

The PDP-4 offered a multi-level interrupt option. As in the PDP-1, each interrupt vectored to a unique memory block. Unlike the PDP-1, the memory block was a single location, which was executed. If the location contained a JMS, control transferred to an interrupt service routine. If the location contained any other instruction, the instruction was executed, but control returned to the main line program. The multi-level interrupt option replaced the real-time clock, an undesirable tradeoff in a real-time system.

Software

Because the PDP-4 was not compatible with the PDP-1, it required new software. DEC provided an editor, an assembler, and, most notably, a Fortran II compiler, all paper-tape based. While the Fortran compiler was a significant advance, the assembler was actually a step backward: the PDP-1's assembler had supported macros, the PDP-4's did not. But it offered some consolation by being a one pass assembler, obviating the need to read the source paper tape twice. The assembler assumed that unresolved references would in fact be resolved and punched unresolved binary code as it processed the source, with a resolution dictionary at the end of the output tape. The resulting tape was then read, upside down and backward, by the loader, which used the resolution dictionary to “fix up” the broken references in the binary.

The PDP-4's programs later became the basis for the PDP-7's software offerings, which accounts for lingering use of Baudot code on the PDP-7. However, the presence of FIODEC on the PDP-4 (and thus on the PDP-7) is a mystery, since the PDP-1 software base was not carried forward.

Early Mass Storage

The PDP-1 and PDP-4 started out as paper tape based systems. The development software was paper tape based; magnetic tape, if used at all, was used strictly for data. This situation was clearly unsatisfactory, and by 1963 DEC was experimenting with mass storage.

The first mass storage products were based on Vermont Research Drums. The Type 23 parallel and Type 24 serial drums offered 131,072 words of storage with rapid access. But the drums were big (two six-foot cabinets for the Type 23, one for the Type 24), expensive, and inflexible: storage was tied to the computer. This didn't fit with the typical use of the 18b computers as "personal" or serially shared systems.

To find a solution, DEC again turned to Lincoln Labs. In 1962, Wes Clark had demonstrated the prototype of the LINC computer. It featured LINCtape, a block-replaceable tape system with a simple, rugged transport and small, inexpensive tape reels. The LINCtape concept offered exactly the kind of "personal" storage needed to complement DEC's computers. With some changes in tape format, DEC offered "MicroTape" (later renamed DECtape) on the PDP-1 and PDP-4 in 1963. The product also included a stand-alone program librarian, Microtrieve. DECtape was to remain the dominant form of mass storage on DEC's 12b and 18b systems into the early 1970's, when it was supplanted by the RK05 (2315-style) cartridge disk drive.

The PDP-7

According to the history of the 18b series in Computer Architecture, the PDP-4 was not a success. The use of slower logic yielded a system that was 5/8 the performance of the PDP-1 at ½ the price. What the market required was a system that was both higher performance and lower cost. That system was the PDP-7. Implemented (primarily) in 10Mhz logic, its basic 1.75 usec cycle time was almost three times the speed of the PDP-1, at 1/3 the cost.

The PDP-7's basic architecture consisted of minor refinements of the PDP-4's instruction set, accompanied by one interesting architectural extensions: multi-user protection, the first in the 18b family. The PDP-7 also was the first 18b PDP to use ASCII coding.

Arithmetic Systems and Character Sets

The PDP-7's arithmetic systems were identical to the PDP-4. The console typewriter was an ASR-33 Teletype. Its eight-bit character set was an early version of ASCII, with the high order bit always forced on. The character set supported both upper and lower case letters, although the console only supported upper case. The line printer's SIXBIT character set was derived from ASCII by truncating codes 040 - 0137 to six bits. The rapid evolution of character sets in the 18b family was embodied in the PDP-7's DECtape-based operating system DECsys. DECsys stored information in FIODEC, Baudot, and SIXBIT, depending on whether the underlying software was derived from the PDP-4 or newly written.

Instruction Set and I/O Architecture

The PDP-7 used the same instruction set architecture as the PDP-4, including the EAE. The extended memory model was the same as the PDP-4's. A new feature was a primitive form of multi-user protection called trap mode. If trapping was enabled, IOT's and HLT became privileged instructions. If extend mode was simultaneously disabled, indirect addresses were confined to the current bank. This allowed for simple time-sharing, with each user in a separate memory bank. (An option, the KA70A, added a small bounds control register to protect memory within a bank.)

The PDP-7's I/O architecture was identical to the PDP-4's, and it used the same controllers for major I/O devices such as DECtape, magnetic tape, and the serial drum. A few new IOT's were added, for management of the trap system. The PDP-7 featured an interprocessor link; this device set the model for the general purpose parallel I/O options in subsequent DEC computers. Like the

PDP-1 (but unlike the PDP-4), the PDP-7 console featured a “read-in” switch, to automate system bootstrapping from paper tape. The “read-in” function did not use the PDP-4’s RIM format but instead loaded memory sequentially from the tape. Therefore, loading software required three steps: use the “read-in” switch to load the RIM loader; use the RIM loader to load the binary loader; and finally use the binary loader to load the software.

Software

The PDP-7 offered DEC’s first mass-storage operating system, the DECtape-based DECsys. (DECsys also ran on the PDP-4.) DECsys was a modest first step in operating system development. It consisted of a simple memory-resident DECtape I/O library, a keyboard monitor, a Fortran II compiler, an assembler, a linking loader, and a symbolic debugger. All of the components were based on PDP-4 and PDP-7 paper-tape counterparts, with calls to the DECtape I/O library replacing paper-tape I/O. The internals of DECsys reflect its heterogeneous origins, with directory information stored in Baudot and source files in FIODEC.

A DECsys system tape contained the bootstrap monitor in blocks 0 and 1, and the directory in block 2. The first word of the directory contained the directory length; the last word contained the address of the first free block on the tape. Directory entries consisted of 5 or 6 words:

Word 1:	Type (1 for **S**ystem, 2 for **W**orking)
Words 2-3:	File name, in Baudot
S, word 4:	starting block on tape
S, word 5:	starting address in memory
W, word 4:	starting block on tape for F (Fortran) version
W, word 5:	starting block on tape for A (assembler) version
W, word 6:	starting block on tape for R (relocatable binary) version

Files were simply linked DECtape blocks, with the first word of a block pointing to the next; a pointer of 0 signified end of file.

As far as the author can tell, all copies of DECsys have vanished. This is equally true of an even more historic system for the PDP-7, UNIX. The PDP-7’s multi-user protection, crude as it was, sufficed for implementation of the first version of UNIX, making the PDP-7 a significant system in the history of computing. Unfortunately, all copies of UNIX for the PDP-7 have been lost. Some details of the PDP-7 version can be found on Dennis Ritchie’s personal web site.

The PDP-9

The PDP-7 was considerably more successful than its predecessors, selling more than 100 systems thanks to its significant price/performance improvements. The PDP-9 was intended to carry the line forward. The arithmetic system and character sets were unchanged, and the instruction set and I/O architecture changed only minimally. The I/O subsystem changed from a radial to a bus design, necessitating redesign of all peripherals. Interfaces to programmed I/O peripherals (paper tape, console, line printer) remained basically the same as the PDP-4 and PDP-7; however, interfaces to mass storage peripherals (magnetic tape, DECtape) changed significantly. An entirely new multi-level interrupt option, called the Automatic Priority Interrupt (API), was designed. The PDP-9 carried over little of the PDP-7’s admittedly small software base.

Instruction Set and I/O Architecture

The PDP-9 introduced a more flexible form of memory management, with a bounds register separating user (lower) memory from system (upper) memory. The PDP-7's trap flag now became the PDP-9's user mode flag.

Although intended to be upward compatible with the PDP-7, the PDP-9 introduced a number of differences:

- Auto-indexing. In the PDP-7, each bank of memory had auto-index registers. In the PDP-9, only bank 0 had auto-index registers, and indirect references through addresses 00010-00017 were forced to reference bank 0.
- Extend mode restore. The PDP-7 used EMIR to prepare the system to restore extend mode at the end of an interrupt. The PDP-9 introduced the more ambitious RES, which prepared the system to restore the link, extend mode, and memory protect mode. This removed two instructions from the end of all interrupt routines.
- Extend mode in traps. The PDP-7 set extend mode on a protection trap but cleared it on an interrupt; the PDP-9 cleared it on both.

The PDP-9's I/O architecture contained some modest improvements in flexibility and error detection. Status flags were added for reader and punch errors. The line printer controller implemented a device-specific interrupt enable/disable. The new DECtape, magnetic tape, and fixed head disk controllers implemented better programming models than their PDP-7 counterparts, and used up fewer device numbers in the process.

The PDP-9 also implemented an entirely new design for multi-level interrupts. Called the Automatic Priority Interrupt (API) option, the API separated the concept of interrupt channel from priority. The API option supported 32 channels (interrupting devices), but the channels were grouped into eight priority levels. Four channels, on the four lowest priorities, were reserved for software interrupts. When an API break occurred, the memory location corresponding to the channel was executed. The location had to contain a JMS to an interrupt service routine; use of other instructions was not supported. The API was carried over unchanged to the PDP-15.

Software

The PDP-9's close compatibility with the PDP-7 allowed the latter's software to be brought forward. However, that code base, dating from the PDP-4, was considered inadequate and relegated to use in the smallest systems. For mainstream use, a new software suite was written from scratch. The three-step software loading process was simplified by eliminating the intermediate RIM loader. The hodge-podge of I/O routines and libraries was replaced by a standard I/O executive that maintained compatible interfaces from the paper-tape environment through the mass-storage based operating systems (Advanced Monitor System, its foreground/background extension, and DOS). The PDP-4/7 assembler syntax and binary formats were scrapped and replaced with a new macro assembler, Macro 9. Fortran II was replaced by Fortran IV. The console was changed from software echoing of input characters to hardware echoing. The intent versus the practice for PDP-9 software is illustrated by the changes in the manual set. The examples in the Systems Reference Manual all follow PDP-7 assembler syntax, but most surviving software is written in Macro 9.

The PDP-15

The PDP-15 introduced the most significant set of architectural changes in the 18b product line since the transition from the PDP-1 to the PDP-4. It represented a major technology shift, from discrete transistors to TTL integrated circuits. The PDP-15 was the fastest and most popular 18b computer in Digital's history. It was also the last.

Instruction Set and I/O Architecture

The PDP-15 introduced four architectural extensions:

- two new registers, an 18b index register and a 9b limit register
- extended addressing to 128K words
- memory relocation and protection
- hardware floating point option

The introduction of the index register made the PDP-15 more competitive with contemporary machines such as the SDS 940 and DDP 516, both of which had indexing. To get an index register select into the memory reference instructions, the directly addressable memory range was reduced from 8K to 4K:

```
  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|   op   |in| x|                address               | mem reference
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

Direct addressing beyond 4K words could be done by indirect addressing (maximum 32K words), or by indexing (maximum 128K words). However, return addresses remained limited to 15b; thus the maximum practical code segment size remained 32K words. Extended memory worked best with the new memory relocation and protection option; in that environment, multiple 32K word programs could reside in memory simultaneously.

The addition of indexing created a serious compatibility problem with the PDP-9. To ameliorate migration issues, the PDP-15 redefined the PDP-7's and PDP-9's extend mode flag as PDP-9 compatibility mode, or bank mode. If bank mode was enabled, memory reference decoding was identical to the PDP-9, without index capability. The PDP-15 did not implement the PDP-9's extend mode capability within bank mode, because extend mode, which was a compatibility aid for PDP-4 and PDP-7 programs, was no longer needed.

The hardware floating point unit was another new addition to the architecture. It dramatically improved the performance of the system in scientific applications. To support indexing and floating point, the PDP-15 introduced two new instructions, both carved out of the IOT instruction. Bits <4:5> of the IOT instruction had been defined as sub-device selects but in practice were unused. The PDP-15 used them to differentiate between IOT instructions (<4:5> = 00), floating point instructions (<4:5> = 01),

```
  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| 1  1  1  0  0  1|           subopcode               | floating point
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|in|                   address                        |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

and index operate instructions (<4:5> = 1x):

```
  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| 1  1  1  0  1| subopcode |        immediate         | index operate
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

In addition to the major changes outlined above, the PDP-15 had its own set of tweaks and incompatibilities compared to its predecessor. Two meaningless operates were redefined as IAC (increment AC) and BSW (byte swap). The former facilitated a one-instruction 2's complement, thereby correcting a hole in the arithmetic system. On the PDP-9, DBR and RES were triggered by a JMP indirect, on the PDP-15 by any indirect. The PDP-15 implemented new IOT skips for bank mode. A mid-life ECO (called the "re-entrancy ECO") added two additional IOT's to inhibit and enable interrupts, respectively.

From a programming viewpoint, the PDP-15's I/O architecture was the same as the PDP-9's, but the implementations were quite different. The PDP-15 implemented a separate I/O processor, providing greater expandability and flexibility, and a different I/O bus. It had more powerful peripherals, including the RP15/RP02 disk pack and the LP15 DMA line printer. Some PDP-9 controllers, such as the TC09 DECtape controller and the RF09 fixed head disk controller, were redesigned to connect directly to the PDP-15's I/O bus; others were interfaced by a backwards-compatible bus converter.

Although the PDP-15 was more successful than any prior 18b system, compared to the PDP-11 its volume was low. This made continuing investment in new technology and options difficult. The CPU was never re-implemented to take advantage of advances in component integration. Investments in new peripheral types and controllers had to be limited. The PDP-15 group responded with great ingenuity to these constraints. Notable developments included:

- Multiprocessing. Two CPU's could share memory and I/O subsystems, for increased throughput in a multiprogramming environment.
- PDP-11 add-on processor. The Unichannel-15 was a PDP-11/05 CPU that functioned as an I/O controller. The Unibus tied in directly to the PDP-15's memory system, using the two data parity lines as extra data lines. This gave the PDP-15 access to inexpensive PDP-11 peripherals, such as the RK05 and LP11.
- XVM memory manager. The XVM project was the final spin on the PDP-15. It replaced the initial memory relocation option with a more sophisticated unit. The new relocation unit allowed individual programs to extend beyond 32K words.

These structural innovations stretched the lifetime of the product line but could not reverse its status as a niche rather than a volume product. By the mid 1970's, the PDP-15's position in DEC's product line was eclipsed by the success of the more flexible PDP-11 (as the position of the PDP-10 would be by the VAX). In 1977, the PDP-15 was retired, ending the history of the 18b product family.

Software

The PDP-15 built on the PDP-9's software base. The Advanced Monitor System was retained and extended to create DOS-15 and its batch extension, BOS-15. A new real-time operating system, RSX15, evolved from an execution-only environment into a full-featured multiprogramming system, RSX15-Plus III, that exploitied the memory relocation hardware and multiprocessing capabilities to provide simultaneous timesharing, batch, and real-time capabilities. Another notable system was MUMPS (**M**GH **U**tility **M**ulti **P**rogramming **S**ystem), a timesharing system developed at Massachusetts General hospital for processing medical records. Descendents of MUMPS (now known as the M language) continue to be used today in medical systems. DOS, RSX-Plus III, and MUMPS were all substantially rewritten in the mid-70's to take advantage of XVM memory management.

18b Systems Today

Because of the low numbers produced (< 1500), and the early retirement of the product line, relatively few examples of the DEC 18b computers are still extent (a fate shared by the early 36b products as well). Surviving systems are scattered and often in private collections, making an accurate census difficult.

- PDP-1: The Computer History Museum (Mountain View, Ca) has three PDP-1's. One of these was running as recently as 1995 and will (hopefully) be restored to operation. The other two are from DEC's history collection.
- PDP-4: The Computer History Museum has three PDP-4's, all from DEC's history collection. None are considered restorable.
- PDP-7: The Computer History Museum has a PDP-7, from DEC's history collection. Max Burnet (Sydney, Australia) has a PDP-7 in his collection. Neither is considered restorable. There is a partially running PDP-7 in Norway and, incredibly, one still in operation in Oregon.
- PDP-9, 9/L: The Computer History Museum has both a PDP-9 and a –9/L. Max Burnet also has one of each, and the PDP-9/L works. The Rhode Island Computer Museum has a PDP-9, which is being restored. There are two PDP-9's at ACONIT (Grenoble, France); Hans Pufal and his team have restored one to working order.
- PDP-15: Multiple examples in private hands.

Sources

The primary source for this article was DEC's documentation archive. The author was fortunate to have access to the archive while it was still being staffed and maintained (Compaq dismissed the archive staff and dispersed the documents; HP is in process of donating the archive to the Computer History Museum). Max Burnet has graciously shared his unique collection of DEC documents and hardware. In addition, Al Kossow and Dave Gesswein have done the field of "computer archaeology" a tremendous service by scanning, and publishing online, surviving documents, DECtapes, and paper-tapes from the 18b family. Last, but hardly least, the staff of the Computer History Museum has made available its significant archive of DEC material. Among the items consulted:

Family

1973 Field Service Census of Systems under contract – Computer History Museum

PDP-1

PDP-1 Handbook (F-15, 1960 edition) – online
PDP-1 Handbook (F-15B, 1961 edition) – online
PDP-1 Handbook (F-15C, 1962 edition) – Max Burnet's collection, now online
PDP-1 Handbook (F-15D, 1963 edition) – Computer History Museum, now online
PDP-1 Maintenance Manual (F-17) – Max Burnet's collection, now online
PDP-1 Input-Output Systems Manual (F-25) – DEC archive, now online

PDP-4

PDP-4 Handbook (F-45, 1962 edition) – DEC archive, now online
PDP-4 Maintenance Manual (F-47) – Max Burnet's collection, now online
PDP-4 Technical Specification (DEC memo M-1142) – online
PDP-4 Fortran Users' Manual (J-4FT) – DEC library, now online

PDP-4 EAE Option Bulletin (F-43(18)P) – Computer History Museum
PDP-4 Paper, Gordon Bell, August 1977 – Computer History Museum

PDP-7

PDP-7 Reference Manual (F-75, 1964 edition) – DEC archive, now online
PDP-7 Maintenance Manual and logic prints (F-77) – Max Burnet's collection
DECSYS-7 Operating Manual (7-5-S) – DEC library, now online

PDP-9

PDP-9 User's Handbook (F-95, 1968 edition) – online
PDP-9 Maintenance Manual (F-97) – online
PDP-9 logic prints – online
KE09A Extended Arithmetic Element Instruction Manual – online
PDP-9 – Design History, Don Vonada, undated – Computer History Museum

PDP-15

PDP-15 Reference Manual (first and sixth editions) – online
XVM System Reference Manual – online
PDP-15 processor diagnostics – online
PDP-15 Development Project History, Jerry Butler, September 1977 – Computer History Museum

Another critical source was Computer Engineering: A DEC View Of Hardware Systems Design. The article "The PDP-1 and Other 18-Bit Computers", by Gordon Bell, Gerald Butler, Robert Gray, John McNamara, Donald Vonada, and Ronald Wilson, contains unique hardware, marketing, and technology information about the 18b family. The book, out of print for years, is now online, thanks to the efforts of Gordon Bell.

Lastly, the author had the benefit of the recollections of people who worked on the 18b family, including Gordon Bell, Dennis Ritchie, and Barry Rubinson, as well as access to the surviving archive of PDP-7 software from Applied Data Research.

18b PDP Web Sites

Gordon Greene's PDP-1 web site, http://www.dbit.com/~greeng3/pdp1/

Barry and Brian Silverman's Java-based emulator for PDP-1 Spacewar, http://mevard.www.media.mit.edu/groups/el/projects/spacewar/

Al Kossow's "Minicomputer Orphanage", including the 18b PDP's, http://www.spies.com/~aek/orphanage.html

Dennis Ritchie and Ken Thompson memoir of early UNIX, http://www.bell-labs.com/history/unix/pdp7.html

Hans Pufal's site about the restored PDP-9 at ACONIT, http://www.aconit.org/hbp/PDP9/

C. Fletcher

PROGRAMMED DATA PROCESSOR-1

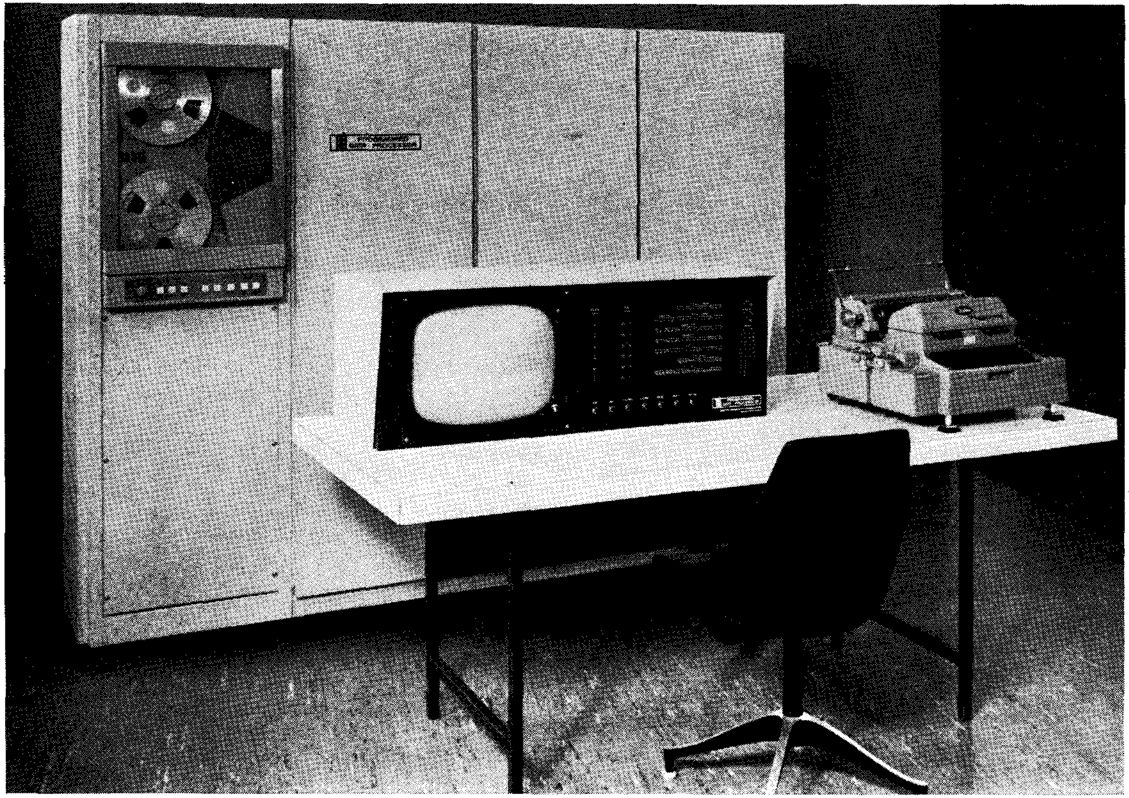
C. Fletcher
PK1/E33

PROGRAMMED DATA PROCESSOR-1

digital equipment corporation

MAYNARD, MASSACHUSETTS

TABLE OF CONTENTS



Programmed Data Processor-1

I. INTRODUCTION

The Programmed Data Processor (PDP-1) is a high speed, solid state digital computer designed to operate with several types of input-output devices, with no internal machine changes. It is a single address, single instruction, stored program computer with powerful program features. Five-megacycle circuits, a magnetic core memory, and fully parallel processing make possible a computation rate of 100,000 additions per second (about 2.5 times the speed of most large computers in use today, and more than 100 times the speed of magnetic drum computers). The PDP-1 is unusually versatile. It is easy to install, operate and maintain. Conventional 110-volt power is used, neither air conditioning nor floor reinforcement is necessary, and preventive maintenance is provided for by built-in marginal checking circuits.

PDP-1 circuits are based on the designs of DEC's highly successful and reliable System Building Blocks. Flip-flops and most switches use saturating transistors. Primary active elements are Micro-Alloy and Micro-Alloy-Diffused transistors.

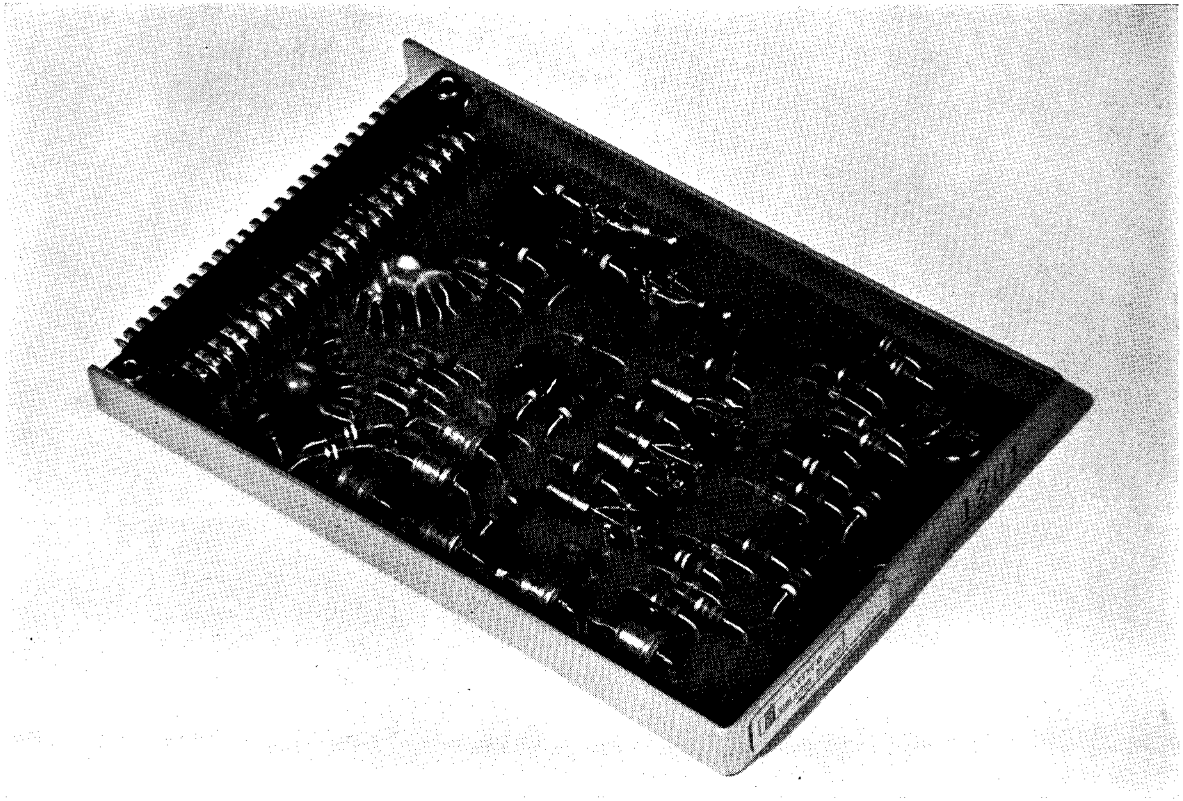
The entire computer occupies only 32 square feet of floor space. It consists of a seven-foot console-desk and three equipment frames.

CENTRAL PROCESSOR

The Central Processor contains the control, arithmetic and memory addressing elements and the memory buffer register. The word length is 18 binary digits. Instructions are carried out in multiples of the memory cycle time of five microseconds. Add, subtract, deposit, and load, for example, are two-cycle instructions requiring 10 microseconds. Multiplication, by subroutine, requires 350 microseconds on the average. Program features include: single address instructions, multiple step indirect addressing and logical arithmetic commands. Console features include: flip-flop indicators grouped for convenient octal reading, six program flags for automatic setting and computer sensing and six sense switches for manual setting and computer sensing.

MEMORY SYSTEM

The coincident-current, magnetic core memory holds 4096 words of 18 bits each. Additional memory units of the same capacity may be readily added to the machine; a memory field switch instruction built into PDP-1 will then select the correct memory module. The



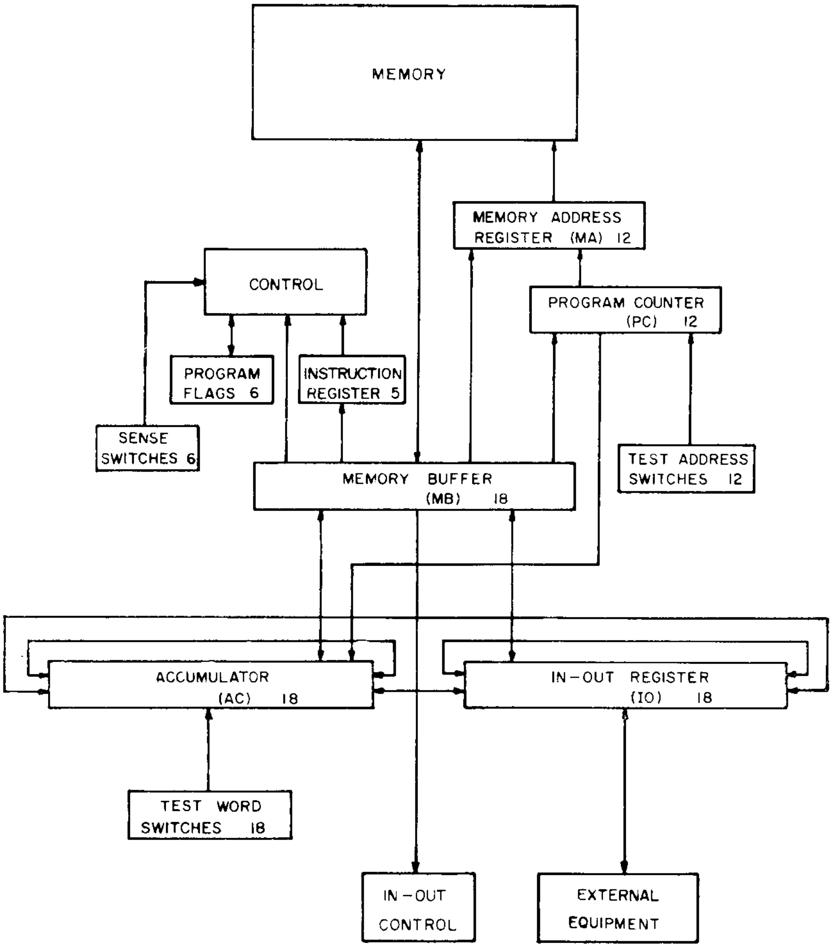
A Standard DEC System Building Block used in PDP-1 read-rewrite time of the memory is five microseconds, the basic computer rate. Driving currents are automatically adjusted to compensate for temperature variations between 55 and 100 degrees fahrenheit. The core memory storage may be supplemented by up to 64 magnetic tape transports and a tape control unit that serves them all.

INPUT-OUTPUT

PDP-1 is designed to operate a variety of input-output devices. Standard equipment consists of a paper tape reader with a read speed of 300 lines (100 18-bit words) per second, a typewriter for on-line operation in both input and output and a paper-tape punch (alphanumeric or binary) with a nominal speed of 20 characters per second. Optional external equipment includes: compatible magnetic tape (75 inches per second, alphanumeric or binary); 16-inch cathode ray tube for graphic or tabular displays; light pen input; line printer (600 lines per minute); analog to digital and digital to analog converters; and a real time clock. All in-out operations are performed through the In-Out Register.

Of particular interest is the ease with which new, and perhaps unusual, external equipment can be added to PDP-1. Space is provided for additional gates to, and buffers from, the In-Out Register. The in-out system is sufficiently simple so that little control circuitry is needed for additional devices.

The PDP-1 is also available with the optional Sequence Break System. This is a 16-channel (or more, when needed) automatic interrupt feature which permits concurrent operation of several in-out devices.



PDP-1 System Block Diagram

II. PROGRAMMING PDP-1

The Central Processor of PDP-1 contains the Control Element, the Memory Buffer Register, the Arithmetic Element, and the Memory Addressing Element. The Control Element governs the complete operation of the computer including memory timing, instruction performance and the initiation of input-output commands. The Arithmetic Element, which includes the Accumulator and the In-Out Register, performs the arithmetic operations. The Memory Addressing Element, which includes the Program Counter and the Memory Address Register, performs address bookkeeping and modification.

The powerful program features of PDP-1 include multiple step indirect addressing, Boolean operations, twelve variations of arithmetic and logical shifting, and ten conditional instructions. Six independent flip-flops, called "program flags," are available for use as program switches or special in-out synchronizers. Two special instructions, Multiply Step and Divide Step, are included in the Instruction List. Multiply and divide subroutines using these instructions operate in about 350 and 600 microseconds respectively.

NUMBER SYSTEM

The PDP-1 is a "fixed point" machine using binary arithmetic. Negative numbers are represented as the 1's complement of the positive numbers. Bit 0 is the sign bit which is ZERO for positive numbers. Bits 1 to 17 are magnitude bits, with Bit 1 being the most significant and Bit 17 being the least significant.

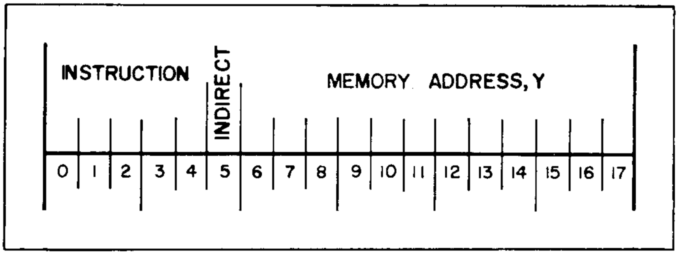
The actual position of the binary point may be arbitrarily assigned to best suit the problem in hand. Two common conventions in the placement of the binary point are:

The binary point is to the right of the least significant digit; thus, numbers represent integers.

The binary point is to the right of the sign digit; thus, the numbers represent a fraction which lies between ± 1.

The conversion of decimal numbers into the binary system for use by the machine may be performed automatically by subroutines. Similarly the output conversion of binary numbers into decimals is done by subroutine. Operations for floating point numbers are handled by interpretive programming. The utility program system provides for automatic insertion of the routines required to perform floating point operations and number base conversion.

INSTRUCTION FORMAT



PDP-1 INSTRUCTION FORMAT

The Bits 0 through 4 define the instruction code; thus there are 32 possible instruction codes, not all of which are used. The instructions may be divided into two classes:

Memory reference instructions

Augmented instructions

In the memory reference instructions, Bit 5 is the indirect address bit. The instruction memory address, Y, is in Bits 6 through 17. These digits are sufficient to address 4096 words of memory.

The augmented instructions use Bits 5 through 17, to specify variations of the basic instruction. For example, in the shift instruction, Bit 5 specifies direction of shift, Bit 6 specifies the character of the shift (arithmetic or logical), Bits 7 and 8 enable the registers (01 = AC, 10 = IO, and 11 = both) and Bits 9 through 17 specify the number of steps.

INDIRECT ADDRESSING

A memory reference instruction which is to use an indirect address will have a ONE in Bit 5 of the instruction word. The original address, Y, of the instruction will not be used to locate the operand, jump location, etc., of the instruction, as is the normal case. Instead, it is used to locate a memory register whose contents in Bits 6 through 17 will be used as the address of the original instruction. Thus, Y is not the location of the operand but the location of the location of the operand. If the memory register containing the indirect address also has a ONE in Bit 5, the indirect addressing procedure is repeated and a third address is located. There is no limit to the number of times this process can be repeated.

OPERATING SPEEDS

Operating times of PDP-1 instructions are multiples of the memory cycle of 5 microseconds. Two-cycle instructions refer twice to memory and thus require 10 microseconds for completion. Examples of this are add, subtract, deposit, load, etc. The jump instruction and the augmented instructions need only one call on memory and are performed in 5 microseconds.

In-Out Transfer instructions that do not include the optional wait function require 5 microseconds. If the in-out device requires a wait time for completion, the operating time depends upon the device being used.

Each step of indirect addressing requires an additional 5 microseconds.

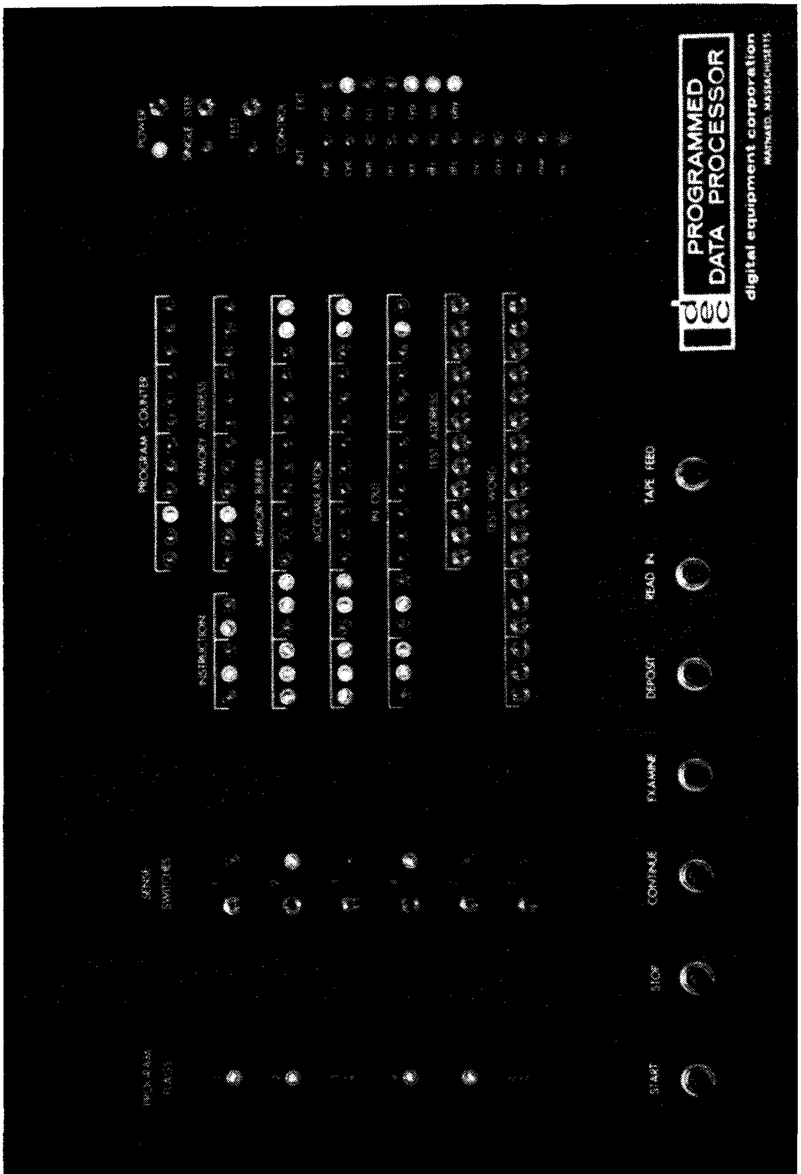
MANUAL CONTROLS

The Console of PDP-1 has controls and indicators for the use of the operator. All computer flip-flops have indicator lights on the Console. These indicators are primarily for use when the machine has stopped or when the machine is being operated one step at a time. While the machine is running, the brightness of an indicator bears some relationship to the relative duty factor of that particular flip-flop.

Three registers of toggle switches are available on the Console. These are the Test Address (12 bits), the Test Word (18 bits), and the Sense Switches (6 bits). The first two are used in conjunction with the operating push buttons. The Sense Switches are present for manual intervention. The use of these switches is determined by the program.

Operating Push Buttons

START	The computer will start. The first instruction comes from the memory location indicated in the Test Address Switches.
STOP	The computer will come to a halt at the completion of the current memory cycle.
CONTINUE	The computer will resume operation starting at the state indicated by the lights.
EXAMINE	The contents of the memory register indicated in the Test Address will be displayed in the Accumulator and the Memory Buffer lights.
DEPOSIT	The word selected by the Test Word Switches will be put in the memory location indicated by the Test Address Switches.
READ-IN	The photoelectric paper tape reader will start operating in the Read-In mode.



PDP-1 Control Panel

Toggle Switches

SINGLE CYCLE SWITCH	When the Single Cycle Switch is on, the computer will halt at the completion of each memory cycle. This switch is particularly useful in debugging programs. Repeated operation of the Continue Switch Button will step the program one cycle at a time. The programmer is thus able to examine the state of the machine at each step.
TEST SWITCH	When the Test Switch is on, the computer will perform the instruction indicated in the Test Address location, repeating this instruction at either the normal or single cycle rate, if the Single Cycle Switch is up. This switch is primarily useful in maintenance.

INSTRUCTION LIST

This list includes the title of the instruction, the normal execution time of the instruction, *i.e.*, the time with no indirect address, the mnemonic code of the instruction, and the operation code number. In the following list, the contents of a register are indicated by C( ). Thus C(Y) means the contents of memory at Address Y; C(AC) means the contents of the accumulator; C(IO) means the contents of the in-out register. An alphabetical and numerical listing of the instructions is contained on Pages 27 to 29.

Memory Reference Instructions

ARITHMETIC INSTRUCTIONS

Add (*10 μsec*)
add Y Operation Code 40

The new C(AC) are the sum of C(Y) and the original C(AC). The C(Y) are unchanged. The addition is performed with 1's complement arithmetic. If the sum exceeds the capacity of the Accumulator Register, the overflow flip-flop will be set (see Skip Group instructions).

Subtract (*10 μsec*)
sub Y Operation Code 42

The new C(AC) are the original C(AC) minus the C(Y). The C(Y) are unchanged. The subtraction is performed using 1's complement arithmetic. If the difference exceeds the capacity of the Accumulator, the overflow flip-flop will be set (see Skip Group instructions).

Multiply Step (*10 μsec*)
mus Y Operation Code 54

If Bit 17 of the In-Out Register is a ONE, the C(Y) are added to C(AC). If IO Bit 17 is a ZERO, the addition does not take place. In either case, the C(AC) and C(IO) are shifted right one place. AC Bit 0 is made ZERO by this shift. This instruction is used in the multiply subroutine.

Divide Step (*10 μsec*)
dis Y Operation Code 56

The Accumulator and the In-Out Register are rotated left one place. IO Bit 17 receives the complement of AC Bit 0. If IO Bit 17 is ONE, the C(Y) are subtracted from C(AC). If IO Bit 17 is ZERO, C(Y) + 1 are added to C(AC). This instruction is used in the divide subroutine.

Index (*10 μsec*)
idx Y Operation Code 44

The C(Y) are replaced by C(Y) + 1. The C(Y) + 1 are left in the Accumulator. The previous C(AC) are lost. Overflow is not indicated.

Index and Skip if Positive (*10 μsec*)
isp Y Operation Code 46

The C(Y) are replaced by C(Y) + 1. The C(Y) + 1 are left in the Accumulator. The previous C(AC) are lost. If, after the addition, C(Y) + 1 are positive, the Program Counter is advanced one extra position and the next instruction in the sequence is skipped. Overflow is not indicated.

Logical Instructions

Logical AND (*10 μsec*)
and Y Operation Code 02

The bits of C(Y) operate on the corresponding bits of the Accumulator to form the logical AND. The result is left in the Accumulator. The C(Y) are unaffected by this instruction.

Logical AND Table

AC Bit	*Y Bit*	*Result*
0	0	0
0	1	0
1	0	0
1	1	1

Exclusive OR (*10 μsec*)
xor Y Operation Code 06

The bits of C(Y) operate on the corresponding bits of the Accumulator to form the exclusive OR. The result is left in the Accumulator. The C(Y) are unaffected by this order.

Exclusive OR Table

AC Bit	*Y Bit*	*Result*
0	0	0
0	1	1
1	0	1
1	1	0

Inclusive OR *(10 μsec)*
ior Y Operation Code 04

The bits of C(Y) operate on the corresponding bits of the Accumulator to form the inclusive OR. The result is left in the Accumulator. The C(Y) are unaffected by this order.

INCLUSIVE OR TABLE

AC Bit	*Y Bit*	*Result*
0	0	0
0	1	1
1	0	1
1	1	1

GENERAL INSTRUCTIONS

Load Accumulator *(10 μsec)*
lac Y Operation Code 20

The C(Y) are placed in the Accumulator. The C(Y) are unchanged. The original C(AC) are lost.

Deposit Accumulator *(10 μsec)*
dac Y Operation Code 24

The C(AC) replace the C(Y) in the memory. The C(AC) are left unchanged by this instruction. The original C(Y) are lost.

Deposit Address Part *(10 μsec)*
dap Y Operation Code 26

Bits 6 through 17 of the Accumulator replace the corresponding digits of memory register Y. C(AC) are unchanged as are the contents of Bits 0 through 5 of Y. The original contents of Bits 6 through 17 of Y are lost.

Deposit Instruction Part *(10 μsec)*
dip Y Operation Code 30

Bits 0 through 5 of the Accumulator replace the corresponding digits of memory register Y. The Accumulator is unchanged as are Bits 6 through 17 of Y. The original contents of Bits 0 through 5 of Y are lost.

Load In-Out Register *(10 μsec)*
lio Y Operation Code 22

The C(Y) are placed in the In-Out Register. C(Y) are unchanged. The original C(IO) are lost.

Deposit In-Out Register *(10 μsec)*
dio Y Operation Code 32

The C(IO) replace the C(Y) in memory. The C(IO) are unaffected by this instruction. The original C(Y) are lost.

Jump *(5 μsec)*
jmp Y Operation Code 60

The Program Counter is reset to Address Y. The next instruction that will be executed will be taken from Memory Register Y. The original contents of the Program Counter are lost.

Jump and Save Program Counter *(5 μsec)*
jsp Y Operation Code 62

The contents of the Program Counter are transferred to the Accumulator. When the transfer takes place, the Program Counter holds the address of the instruction following the jsp. The Program Counter is then reset to Address Y. The next instruction that will be executed will be taken from Memory Register Y. The original C(AC) are lost.

Skip if Accumulator and Y differ *(10 μsec)*
sad Y Operation Code 50

The C(Y) are compared with the C(AC). If the two numbers are different, the Program Counter is indexed one extra position and the next instruction in the sequence is skipped. The C(AC) and the C(Y) are unaffected by this operation.

Skip if Accumulator and Y are the same *(10 μsec)*
sas Y Operation Code 52

The C(Y) are compared with the C(AC). If the two numbers are identical, the Program Counter is indexed one extra position and the next instruction in the sequence is skipped. The C(AC) and C(Y) are unaffected by this operation.

Augmented Instructions

Load Accumulator with N *(5 μsec)*
law N Operation Code 70

The number in the memory address bits of the instruction word is placed in the Accumulator. If the indirect address bit is ONE, the complement of N (−N) is put in the Accumulator.

Shift Group *(5 μsec)*
sft Operation Code 66

This group of instructions will rotate or shift the Accumulator and/or the In-Out Register. When the two registers operate combined, the In-Out Register is considered to be an 18-bit magnitude extension of the right end of the Accumulator.

Rotate is a non-arithmetic cyclic shift. That is, the two ends of the register are logically tied together and information is rotated as though the register were a ring.

Shift is an arithmetic operation and is, in effect, multiplication of the number in the register by $2^{\pm N}$, where N is the number of shifts; plus is left and minus is right.

The number of shift or rotate steps to be performed (N) is indicated by the number of ONES in Bits 9 thru 17 of the instruction word. Thus, Rotate Accumulator Right nine times is 671777. A shift or rotate of one place can be indicated nine different ways. The usual convention is to use the right end of the instruction word (rar 1 = 671001).

Rotate Accumulator Right *(5 μsec)*
rar N Operation Code 671

Rotates the bits of the Accumulator right N positions, where N is the number of ONES in Bits 9-17 of the instruction word.

Rotate Accumulator Left (*5 μsec*)
ral N Operation Code 661

Rotates the bits of the Accumulator left N positions, where N is the number of ONES in Bits 9-17 of the instruction word.

Shift Accumulator Right (*5 μsec*)
sar N Operation Code 675

Shifts the contents of the Accumulator right N positions, where N is the number of ONES in Bits 9-17 of the instruction word.

Shift Accumulator Left (*5 μsec*)
sal N Operation Code 665

Shifts the contents of the Accumulator left N positions, where N is the number of ONES in Bits 9-17 of the instruction word.

Rotate In-Out Register Right (*5 μsec*)
rir N Operation Code 672

Rotates the bits of the In-Out Register right N positions, where N is the number of ONES in Bits 9-17 of the instruction word.

Rotate In-Out Register Left (*5 μsec*)
ril N Operation Code 662

Rotates the bits of the In-Out Register left N positions, where N is the number of ONES in Bits 9-17 of the instruction word.

Shift In-Out Register Right (*5 μsec*)
sir N Operation Code 676

Shifts the contents of the In-Out Register right N positions, where N is the number of ONES in Bits 9-17 of the instruction word.

Shift In-Out Register Left (*5 μsec*)
sil N Operation Code 666

Shifts the contents of the In-Out Register left N positions, where N is the number of ONES in Bits 9-17 of the instruction word.

Rotate AC and IO Right (*5 μsec*)
rcr N Operation Code 673

Rotates the bits of the combined registers right in a single ring N positions, where N is the number of ONES in bits 9-17 of the instruction word.

Rotate AC and IO Left (*5 μsec*)
rcl N Operation Code 663

Rotates the bits of the combined registers left in a single ring N positions, where N is the number of ONES in Bits 9-17 of the instruction word.

Shift AC and IO Right (*5 μsec*)
scr N Operation Code 677

Shifts the contents of the combined registers right N positions, where N is the number of ONES in Bits 9-17 of the instruction word.

Shift AC and IO Left (*5 μsec*)
scl N Operation Code 667

Shifts the contents of the combined registers left N positions, where N is the number of ONES in Bits 9-17 of the instruction word.

Skip Group (*5 μsec*)
skp Operation Code 64

This group of instructions senses the state of various flip-flops and switches in the machine. The address portion of the instruction selects the particular function to be sensed. All members of this group have the same operation code.

The instructions in the Skip Group may be combined to form the inclusive OR of the separate skips. Thus, if Address 3000 is selected, the skip would occur if the overflow flip-flop equals ZERO or if the In-Out Register is positive. The combined instruction would still take 5 microseconds.

Skip on ZERO Accumulator (*5 μsec*)
sza Address 100

If the Accumulator is equal to plus ZERO (all bits are ZERO), the Program Counter is advanced one extra position and the next instruction in the sequence is skipped.

Skip on Plus Accumulator (*5 μsec*)
spa Address 200

If the sign bit of the Accumulator is ZERO, the Program Counter is advanced one extra position and the next instruction in the sequence is skipped.

Skip on Minus Accumulator (*5 μsec*)
sma Address 400

If the sign bit of the Accumulator is ONE, the Program Counter is advanced one extra position and the next instruction in the sequence is skipped.

Skip on ZERO Overflow (*5 μsec*)
szo Address 1000

If the overflow flip-flop is a ZERO, the Program Counter is advanced one extra position and the next instruction in the sequence will be skipped. The overflow flip-flop is cleared by the instruction. This flip-flop is set by an addition or subtration that exceeds the capacity of the Accumulator. The overflow flip-flop is not cleared by arithmetic operations which do not cause an overflow. Thus, a whole series of arithmetic operations can be checked for correctness by a single szo. The overflow flip-flop is cleared by the "Start" Switch.

Skip on Plus In-Out Register (*5 μsec*)
spi Address 2000

If the sign digit of the In-Out Register is ZERO, the Program Counter is indexed one extra position and the next instruction in sequence is skipped.

Skip on ZERO Switch *(5 μsec)*
szs Addresses 10, 20 70

If the selected Sense Switch is ZERO, the Program Counter is advanced one extra position and the next instruction in the sequence will be skipped. Address 10 senses the position of Sense Switch 1, Address 20 Switch 2, etc. Address 70 senses all the switches. If 70 is selected all 6 switches must be ZERO to cause the skip.

Skip on ZERO Program Flag *(5 μsec)*
szf Addresses 0 to 7 inclusive

If the selected program flag is a ZERO, the Program Counter is advanced one extra position and the next instruction in the sequence will be skipped. Address 0 is no selection. Address 1 selects Program Flag 1, etc. Address 7 selects all program flags. All flags must be ZERO to cause the skip.

Operate Group *(5 μsec)*
opr Operation Code 76

This instruction group performs miscellaneous operations on various Central Processor Registers. The address portion of the instruction specifies the action to be performed.

The instructions in the Operate Group can be combined to give the union of the functions. The instruction opr 3200 will clear the AC, put TW to AC, and complement AC. If the number minus zero is interpreted as an instruction, the IO is cleared, AC gets the complement of the TW switches, all program flags are set and the computer halts.

Clear In-Out Register *(5 μsec)*
cli Address 4000

Clears (sets equal to plus zero) the In-Out Register.

Load Accumulator from Test Word *(5 usec)*
lat Address 2000

Forms the inclusive OR of the C(AC) and the contents of the Test Word. This instruction is usually combined with address 200 (clear Accumulator), so that C(AC) will equal the contents of the Test Word Switches.

Complement Accumulator *(5 μsec)*
cma Address 1000

Complements (makes negative) the contents of the Accumulator.

Halt
hlt Address 400

Stops the computer.

Clear Accumulator *(5 μsec)*
cla Address 200

Clears (sets equal to plus zero) the contents of the Accumulator.

Clear Selected Program Flag *(5 μsec)*
clf Address 01 to 07 inclusive

Clears the selected program flag. Address 01 clears Program Flag 1, 02 clears Program Flag 2, etc. Address 07 clears all program flags.

Set Selected Program Flag *(5 μsec)*
stf Addresses 11 to 17 inclusive

Sets the selected program flag. Address 11 sets Program Flag 1; 12 sets Program Flag 2, etc. Address 17 sets all program flags.

In-Out Transfer Group *(5 μsec without in-out wait)*
iot Operation Code 72

The variations within this group of instructions perform all the in-out control and information transfer functions. If Bit 5 (normally the Indirect Address bit) is a ONE, the computer will halt and wait for the completion pulse from the device activated. When this device delivers its completion, the computer will resume operation of the instruction sequence.

An incidental fact which may be of importance in certain scientific or real time control applications is that the time origin of operations following an in-out completion pulse is identical with the time of that pulse.

Most in-out operations require a known minimum time before completion. This time may be utilized for programming. The appropriate In-Out Transfer is given with no in-out wait (Bit 5 a ZERO). The instruction sequence then continues. This sequence must include an iot instruction which performs nothing but the in-out wait, and the instruction must occur before the safe minimum time. A table of minimum times for all in-out devices is delivered with the computer: it lists minimum time before completion pulse and minimum In-Out Register free time.

III. INPUT-OUT EQUIPMENT

STANDARD EQUIPMENT

Paper Tape Reader

The paper tape reader of the PDP-1 is a photoelectric device capable of reading 300 lines per second. Three lines form the standard 18-bit word when reading binary punched eight-hole tape. Five, six, and seven-hole tape may also be read.

Read Paper Tape, Alphanumeric

rpa iot 1

In this mode, one line of tape is read for each In-Out Transfer. All eight holes of the line are read. The information is left in the right eight bits of the In-Out Register, the remainder of the register being left clear.

The code of the off-line tape preparation typewriter (Friden FPC-8 "Flexowriter") contains an odd parity bit. This bit may be checked by the read-in program. The Friden Code is then converted to a concise six-bit code. This conversion squeezes out the fifth bit (parity) and drops the eighth bit. The carriage return character (Friden 200) is converted to 77.

The more concise code is used by the on-line typewriter, printer, and magnetic tape. A list of characters and their codes is found on Pages 30 and 31.

Read Paper Tape Binary

rpb iot 2

For each In-Out Transfer instruction, three lines of paper tape are read and assembled in the In-Out Register to form a full computer word. For a line to be recognized in this mode, the eighth hole must be punched; *i.e.*, lines with no eighth hole will be skipped over. The seventh hole is ignored. The pattern of holes in the binary tape is arranged so as to be easily interpreted visually in terms of machine instruction.

Read-In Mode

This is a special mode activated by the "Read-In" switch on the console. It provides a means of entering programs which neither rely on programs in memory nor require a plug board. Pushing the "Read-In" switch starts the reader in the binary mode. The first group of three lines, and alternate succeeding groups of three lines, are interpreted as "Read-In" mode instructions. Even-numbered groups of three lines are data. The "Read-In" mode instructions must be either "deposit in-out" (dio Y) or "jump" (jmp Y). If the instruction is dio Y, the next group of three binary lines will be stored in memory location Y and the reader continues moving. If the instruction is jmp Y, the "Read-In" mode is terminated and the computer will commence operation at the address of the jump instruction.

Paper Tape Punch

The standard PDP-1 paper tape punch has a nominal speed of 20 lines per second. It can operate in either the alphanumeric mode or the binary mode.

Punch Paper Tape, Alphanumeric

ppa iot 5

For each In-Out Transfer instruction one line of tape is punched. In-Out Register Bit 17 conditions Hole 1. Bit 16 conditions Hole 2, etc. Bit 10 conditions Hole 8.

Punch Paper Tape, Binary

ppb iot 6

For each In-Out Transfer instruction one line of tape is punched. In-Out Register Bit 5 conditions Hole 1. Bit 4 conditions Hole 2, etc. Bit 0 conditions Hole 6. Hole 7 is left blank. Hole 8 is always punched in this mode.

Typewriter

The typewriter will operate in the input mode or the output mode.

Type Out

tyo iot 3

For each In-Out Transfer instruction one character is typed. The character is specified by the right six bits of the In-Out Register.

Type In

tyi iot 4

This operation is completely asynchronous and is therefore handled differently than any of the preceding in-out operations.

When a typewriter key is struck, Program Flag 1 is set. At the same time the code for the struck key is presented to gates connected to the right six bits of the In-Out Register. This information will remain at the gate for a relatively long time by virtue of the slow mechanical action. A program designed to accept typed-in data would periodically check the status of Program Flag 1. If at any time Program Flag 1 is found to be set, an In-Out Transfer instruction with Address 4 must be executed for information to be transferred. This In-Out Transfer should not use the optional in-out halt. The information contained in the typewriter's coder is then read into the right six bits of the In-Out Register. tyi does not clear the IO. The tyi is usually preceded by cli and clf-1.

OPTIONAL EQUIPMENT

Magnetic Tape

The magnetic tape system consists of the magnetic tape control unit and one or more tape transport units which contain the read and write circuits. The tape control unit contains the equipment to select the active transport and the logic necessary to control the system.

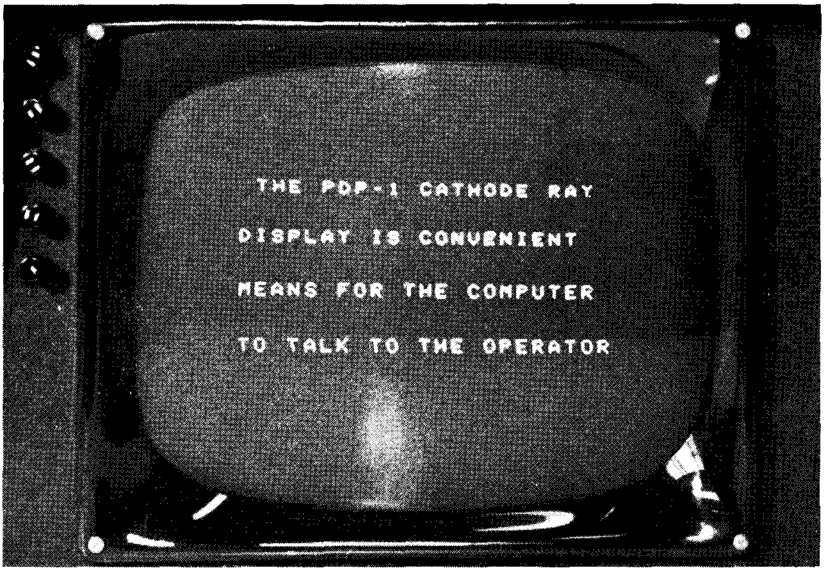
The method of recording is non-return-to-zero. Each flux change represents a binary ONE. The reading is done at two levels of sensitivity. A check is performed at each level. Thus the high level of sensitivity detects the presence of excessive noise and the low sensitivity detects weak ONES.

The transports operate at 75 inches per second with a recording density of 200 bits to the inch. The format is the same as for the IBM 729 I. Seven tracks are written: six are binary or alphanumeric bits, and a seventh is used as a lateral parity. At the completion of a record, which may be of arbitrary length, a longitudinal parity is written.

Real Time Clock

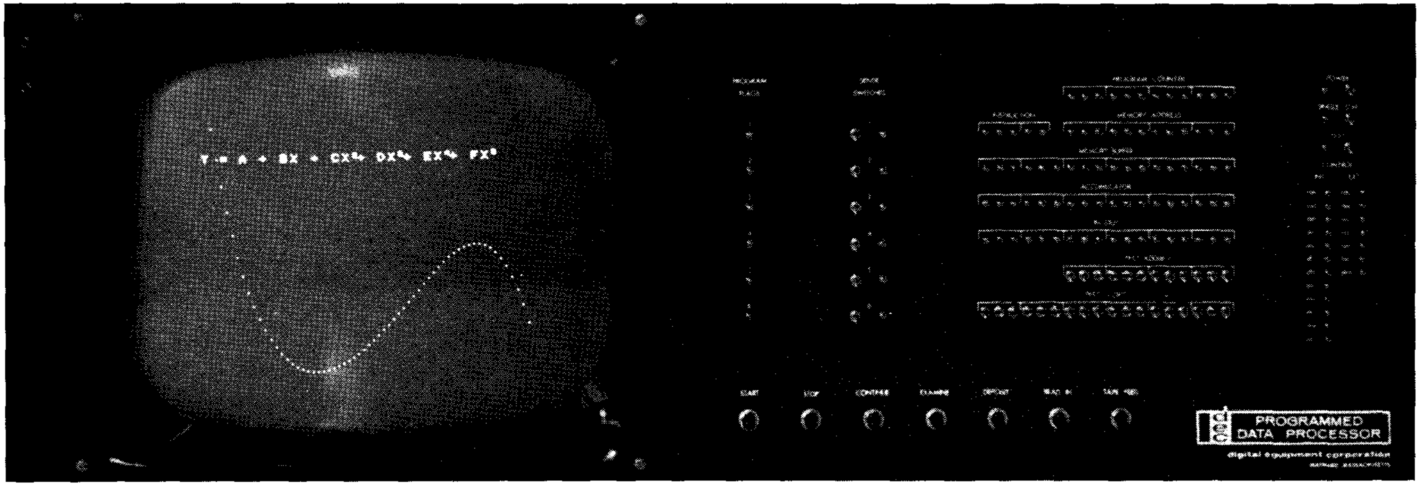
A special input register may be connected to operate as a real time clock. This is a counting register operated by a crystal controlled oscillator.

The state of this counter may be read at any time by the appropriate In-Out Transfer instruction. The computer stops only long enough to provide synchronization with the clock oscillator, then resumes operation in phase with it.



Cathode Ray Tube Display

The PDP-1 cathode ray tube display is useful for presentation of graphical or tabular data to the operator. For each In-Out Transfer instruction, one point is displayed. The first 10 bits of the In-Out Register, Bits 0-9, are the X coordinate of the point. Bits 0-9 of the Accumulator are the Y coordinate of the point.



CATHODE RAY X-Y POINT PLOTTER

An additional display option is a light pen. By use of this device the computer is signaled that the operator is interested in the last point displayed. Thus the program can take appropriate action such as changing the display or shifting operation to another program.

Line Printer

A 72-column line printer is available as an on-line printing station. The operating speed is 450 lines per minute. A simple one-line buffer is part of this equipment. The appropriate In-Out Transfer instruction is repeated to fill the buffer. The order to print is then given. Following the completion of the line print, the printer returns a completion pulse and spaces the paper.

Analog Equipment

Equipment providing analog input to and output from the computer can be provided. This equipment can take the form either of high speed electronic equipment or shaft position conversion equipment. In either case, multiplexing can be provided.

Other Optional Equipment

Additional in-out devices may be added to PDP-1 with, at most, a few hour's work on the machine. Sockets for several In-Out Transfer variation pulse commands are prewired. Space is provided for additional gates to and buffers from the In-Out Register. The in-out system is sufficiently simple so that the control circuitry needed for any additional device is minimal.

Sequence Break System

An optional in-out control is available for PDP-1. This control, termed the Sequence Break System, allows concurrent operation of several in-out devices and the main sequence. The system has, nominally, 16 automatic interrupt channels arranged in a priority chain.

A break to a particular sequence may be initiated by the completion of an in-out device, the program, or any external signal. If this sequence has priority, the C(AC), C(IO), C(PC), and the contents of the memory field flip-flops (if present) are stored in adjacent fixed locations unique to that sequence. The Program Counter is reset to the address contained in a fourth fixed location. The program is now operating in the new sequence. This new sequence may be broken by a higher priority sequence. A typical program loop for handling an in-out sequence would contain 3 to 5 instructions, including the appropriate iot. These are followed by load AC and load IO from the fixed locations and an indirect jump to location of the previous C(PC). This last instruction terminates the sequence.

The Sequence Break System provides PDP-1 with much of the power of a multiple sequence machine or of a computer having in-out synchronizers or automatic trunks.

IV. UTILITY PROGRAMS

The Utility Programs for PDP-1 are designed to provide the nucleus of a growing system of programs. Programs available upon delivery of the machine are:

SYMBOLIC ADDRESS ASSEMBLY PROGRAM is the basic element in the utility system. It is designed for maximum flexibility consistent with adequate indication of program errors. Numerous macro instructions are included such as floating point add, subtract, multiply, divide, decimal-to-binary conversion, and binary-to-decimal conversion.

MEMORY PRINT-OUT can appear either on the typewriter or on the line printer, if connected.

BINARY PUNCH will punch out a specified region of memory. Check characters are included on the tape. The program is available in both the binary read-in format and the read-in mode format.

BINARY READ-IN reads the tapes prepared by the Binary Punch Program. Several versions of this program are available. They differ in the region of memory in which the read-in program is written. Thus, various read-in programs are available in both the binary read-in format and the read-in mode format.

MAINTENANCE PROGRAMS include programs for checking memories, input-output equipment, and operation of the Central Processor.

V. APPENDIX

ABBREVIATED INSTRUCTION LIST

Basic Instructions

Instruction	*Code #*	*Explanation*	*Oper. Time* (*μsec*)	*Page Ref.*
add Y	40	Add C(Y) to C(AC)	10	13
and Y	02	Logical AND C(Y) with C(AC)	10	14
dac Y	24	Put C(AC) in Y	10	15
dap Y	26	Put contents of address part of AC in Y	10	15
dio Y	32	Put C(IO) in Y	10	15
dip Y	30	Put contents of instruction part of AC in Y	10	15
dis Y	56	Divide step	10	14
idx Y	44	Index (add one) C(Y), leave in Y & AC	10	14
ior Y	04	Inclusive OR C(Y) with C(AC)	10	15
iot Y	72	In-out transfer, see below		20
isp Y	46	Index and skip if result is positive	10	14
jmp Y	60	Take next instruction from Y	5	15
jsp Y	62	Jump to Y and save program counter in AC	5	16
lac Y	20	Load the AC with C(Y)	10	14
law N	70	Load the AC with the number N	5	16
law −N	71	Load the AC with the number −N	5	16
lio Y	22	Load IO with C(Y)	10	15
mus Y	54	Multiply step	10	14
opr	76	Operate, see below	5	19
sad Y	50	Skip next instruction if $C(AC) \neq C(Y)$	10	16
sas Y	52	Skip next instruction if $C(AC) = C(Y)$	10	16
shift	66	See below	5	16
skp	64	Skip, see below	5	18
sub Y	42	Subtract C(Y) from C(AC)	10	13
xor Y	06	Exclusive OR C(Y) with C(AC)	10	14

Operate Group

Instruction	*Code #*	*Explanation*	*Oper. Time (μsec)*	*Page Ref.*
cla	760200	Clear AC	5	19
clf	760001-7	Clear selected Program Flag	5	20
cli	764000	Clear IO	5	19
cma	761000	Complement AC	5	19
hlt	760400	Halt	5	19
lat	762200	Load AC from Test Word switches	5	19
stf	760011-7	Set selected Program Flag	5	20

In-Out Transfer Group

ppa	730005	Punch paper tape alphanumeric		22
ppb	730006	Punch paper tape binary		22
rpa	730001	Read paper tape alphanumeric		21
rpb	730002	Read paper tape binary		21
tyi	720004	Read typewriter input switches	5	22
tyo	730003	Type out		22

Skip Group

sma	640400	Skip on minus AC	5	18
spa	640200	Skip on plus AC	5	18
spi	642000	Skip on plus IO	5	18
sza	640100	Skip on ZERO (+0) AC	5	18
szf	64000f	Skip on ZERO flag (f = flag #)	5	19
szo	641000	Skip on ZERO overflow (and clear overflow)	5	18
szs	6400S0	Skip on ZERO sense switch (S = switch #)	5	19

Shift/Rotate Group

ral	661	Rotate AC left	5	17
rar	671	Rotate AC right	5	16
rcl	663	Rotate combined AC & IO left	5	17
rcr	673	Rotate combined AC & IO right	5	17
ril	662	Rotate IO left	5	17
rir	672	Rotate IO right	5	17
sal	665	Shift AC left	5	17

Shift/Rotate Group (Continued)

Instruction	*Code #*	*Explanation*	*Oper. Time (μsec)*	*Page Ref.*
sar	675	Shift AC right	5	17
scl	667	Shift combined AC & IO left	5	18
scr	677	Shift combined AC & IO right	5	17
sil	666	Shift IO left	5	17
sir	676	Shift IO right	5	17

NUMERICAL INSTRUCTION LIST

Code	*Instruction*	*Code*	*Instruction*
00	*	40	add
02	and	42	sub
04	ior	44	idx
06	xor	46	isp
10	*	50	sad
12	*	52	sas
14	*	54	mus
16	*	56	dis
20	lac	60	jmp
22	lio	62	jsp
24	dac	64	skp
26	dap	66	Shift
30	dip	70	law
32	dio	72	iot
34	*	74	*
36	*	76	opr

* spare code, computer will halt

ALPHANUMERIC CODES

TABLE I

Character	*Friden Code*	*Concise Code*	*Character*	*Friden Code*	*Concise Code*
a A	141	61	y Y	070	30
b B	142	62	z Z	051	31
c C	163	63	0)	040	20
d D	144	64	1 ′	001	01
e E	165	65	2 @	002	02
f F	166	66	3 #	023	03
g G	147	67	4 =	004	04
h H	150	70	5 %	025	05
i I	171	71	6 ¢	026	06
j J	121	41	7 ?	007	07
k K	122	42	8 *	010	10
l L	103	43	9 (	031	11
m M	124	44	Space	020	00
n N	105	45	, ,	073	33
o O	106	46	. .	153	73
p P	127	47	/ :	061	21
q Q	130	50	& ;	160	60
r R	111	51	$ —	133	53
s S	062	22	- ″	100	40
t T	043	23	Upper Case	174	74
u U	064	24	Lower Case	172	72
v V	045	25	Tab.	076	36
w W	046	26	Carr. Ret.	200	77
x X	067	27	Tape Feed	177	—

TABLE II

Friden	*Character*	*Friden*	*Character*
001	1 ′	043	t T
002	2 @	045	v V
004	4 =	046	w W
007	7 ?	051	z Z
010	8 *	061	/ :
020	Space	062	s S
023	3 #	064	u U
025	5 %	067	x X
026	6 ¢	070	y Y
031	9 (	073	, ,
040	0)	076	Tab.

TABLE II (Continued)

Friden	*Character*	*Friden*	*Character*
100	– ″	144	d D
103	l L	147	g G
105	n N	150	h H
106	o O	153	. .
111	r R	160	& ;
121	j J	163	c C
122	k K	165	e E
124	m M	166	f F
127	p P	171	i I
130	q Q	172	Lower Case
133	$ —	174	Upper Case
141	a A	177	Tape Feed
142	b B	200	Carr. Ret.

TABLE III

Concise Code	*Character*	*Concise Code*	*Character*
00	Space	42	k K
01	1 ′	43	l L
02	2 @	44	m M
03	3 #	45	n N
04	4 =	46	o O
05	5 %	47	p P
06	6 ¢	50	q Q
07	7 ?	51	r R
10	8 *	53	$ —
11	9 (	60	& ;
20	0)	61	a A
21	/ :	62	b B
22	s S	63	c C
23	t T	64	d D
24	u U	65	e E
25	v V	66	f F
26	w W	67	g G
27	x X	70	h H
30	y Y	71	i I
31	z Z	72	Lower Case
33	, ,	73	. .
36	Tab.	74	Upper Case
40	– ″	77	Carr. Ret.
41	j J		

F-15B

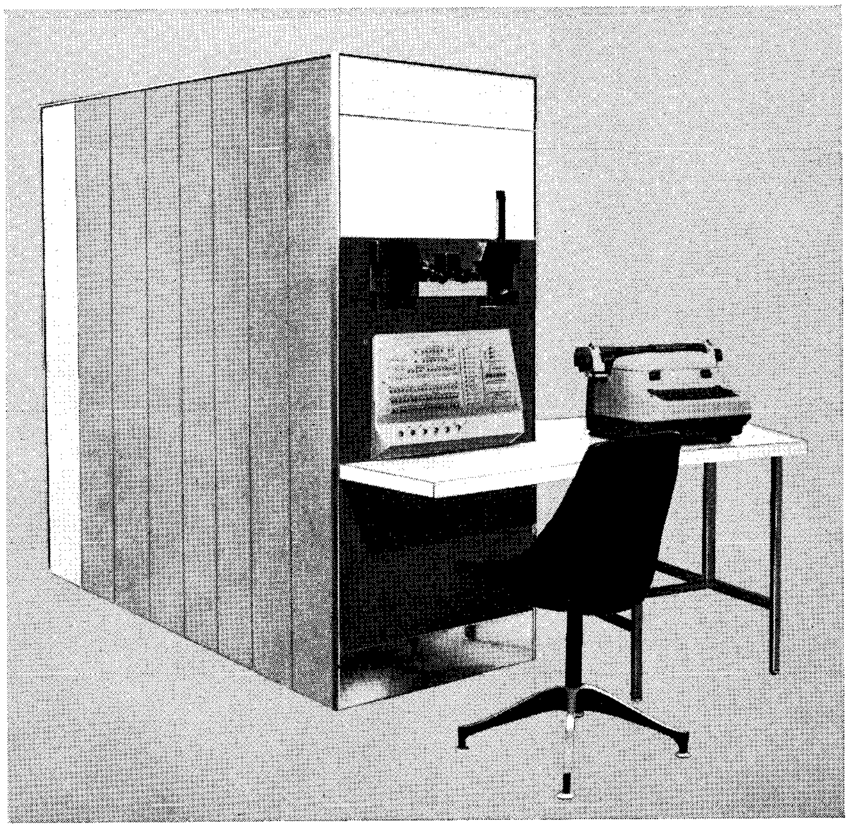
PROGRAMMED DATA PROCESSOR-1 MANUAL

C. [illegible] PK1/E33

PROGRAMMED DATA PROCESSOR-1 MANUAL

DIGITAL EQUIPMENT CORPORATION

Maynard · Massachusetts



Programmed Data Processor-1

TABLE OF CONTENTS

I. INTRODUCTION

The Programmed Data Processor (PDP-1) is a high speed, solid state digital computer designed to operate with several types of input-output devices, with no internal machine changes. It is a single address, single instruction, stored program computer with powerful program features. Five-megacycle circuits, a magnetic core memory, and fully parallel processing make possible a computation rate of 100,000 additions per second (about 2.5 times the speed of most large computers in use today, and more than 100 times the speed of magnetic drum computers). The PDP-1 is unusually versatile. It is easy to install, operate and maintain. Conventional 110-volt power is used, neither air conditioning nor floor reinforcement is necessary, and preventive maintenance is provided for by built-in marginal checking circuits.

PDP-1 circuits are based on the designs of DEC's highly successful and reliable System Modules. Flip-flops and most switches use saturating transistors. Primary active elements are Micro-Alloy and Micro-Alloy-Diffused transistors.

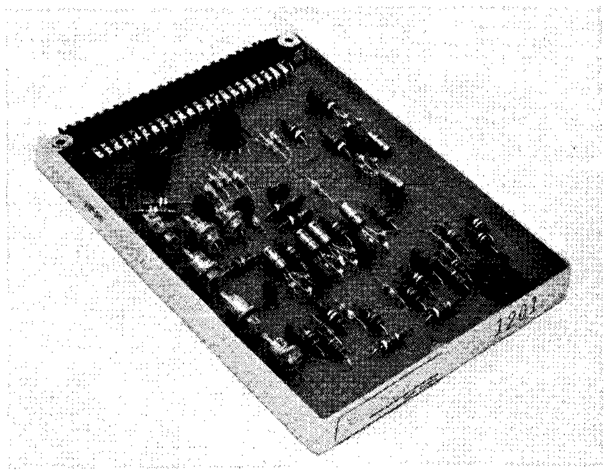
The entire computer occupies only 17 square feet of floor space. It consists of four equipment frames, one of which is used as the operating station.

CENTRAL PROCESSOR

The Central Processor contains the control, arithmetic and memory addressing elements and the memory buffer register. The word length is 18 binary digits. Instructions are carried out in multiples of the memory cycle time of five microseconds. Add, subtract, deposit, and load, for example, are two-cycle instructions requiring 10 microseconds. Multiplication, by subroutine, requires 325 microseconds on the average. Program features include: single address instructions, multiple step indirect addressing and logical arithmetic commands. Console features include: flip-flop indicators grouped for convenient octal reading, six program flags for automatic setting and computer sensing and six sense switches for manual setting and computer sensing.

MEMORY SYSTEM

The coincident-current, magnetic core memory holds 4096 words of 18 bits each. Up to eight additional memory units of the same capacity may be readily added to the machine; a memory field switch instruction built into PDP-1 will then select the correct memory module. The read-rewrite time of the memory is five microseconds, the basic computer rate. Driving currents are automatically adjusted to compensate for temperature variations



A Standard DEC System Module used in PDP-1

between 50 and 110 degrees Fahrenheit. The core memory storage may be supplemented by up to 24 magnetic tape transports.

INPUT-OUTPUT

PDP-1 is designed to operate a variety of input-output devices. Standard equipment consists of a punched tape reader with a read speed of 400 lines per second, an alphanumeric typewriter for on-line operation in both input and output and a punched tape punch (alphanumeric or binary) with a speed of 63 lines per second. Optional external equipment includes: compatible magnetic tape (75 inches per second, BCD or binary); 16-inch cathode ray tube for graphic or tabular displays; light pen input; line printer (150 or 600 lines per minute); punched cards (input and output at speeds of 100 cards per minute); and a real time clock. All in-out operations are performed through the In-Out Register or through High Speed Input-Output Channels.

Of particular interest is the ease with which new, and perhaps unusual, external equipment can be added to PDP-1. Space is provided for additional gates to, and buffers from, the In-Out Register. The in-out system is sufficiently simple so that little control circuitry is needed for additional devices. New input-output instructions can be implemented easily at the Input-Output Instruction Control Panel.

The PDP-1 is also available with the optional Sequence Break System. This is a 16-channel automatic interrupt feature which permits concurrent operation of several in-out devices. A one-channel Sequence Break System is included in the standard PDP-1.

II. PROGRAMMING PDP-1

The Central Processor of PDP-1 contains the Control Element, the Memory Buffer Register, the Arithmetic Element, and the Memory Addressing Element. The Control Element governs the complete operation of the computer including memory timing, instruction performance and the initiation of input-output commands. The Arithmetic Element, which includes the Accumulator and the In-Out Register, performs the arithmetic operations. The Memory Addressing Element, which includes the Program Counter and the Memory Address Register, performs address bookkeeping and modification.

The powerful program features of PDP-1 include:

- Multiple step indirect addressing
- Boolean operations
- Twelve variations of arithmetic and logical shifting, operating on 18 or 36 bits
- Fifteen conditional instructions (expandable by combining to form the inclusive OR of the separate conditions)
- Three different subroutine calling instructions
- Combinable housekeeping instructions
- Index and Index-Conditional instructions
- Execute instruction
- Load-immediate instructions

Six independent flip-flops, called "program flags," are available for use as program switches or special in-out synchronizers. Two special instructions, Multiply Step and Divide Step, are included in the Instruction List. Multiply and divide subroutines using these instructions operate in about 325 and 440 microseconds respectively.

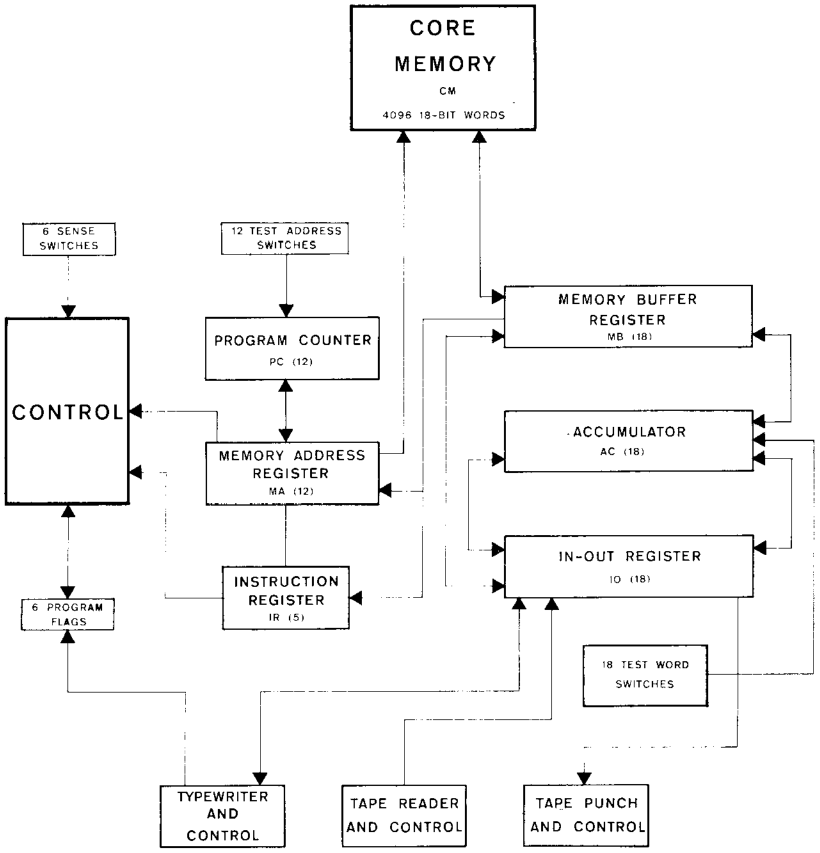
NUMBER SYSTEM

The PDP-1 is a "fixed point" machine using binary arithmetic. Negative numbers are represented as the 1's complement of the positive numbers. Bit 0 is the sign bit which is ZERO for positive numbers. Bits 1 to 17 are magnitude bits, with Bit 1 being the most significant and Bit 17 being the least significant.

The actual position of the binary point may be arbitrarily assigned to best suit the problem in hand. Two common conventions in the placement of the binary point are:

The binary point is to the right of the least significant digit; thus, numbers represent integers.

The binary point is to the right of the sign digit; thus, the numbers represent a fraction which lies between ± 1.



PDP-1 System Block Diagram

The conversion of decimal numbers into the binary system for use by the machine may be performed automatically by subroutines. Similarly the output conversion of binary numbers into decimals is done by subroutine. Operations for floating point numbers are handled by interpretive programming. The PDP-1 Compiler-Assembler System provides for automatic insertion of the routines required to perform floating point operations and number base conversion.

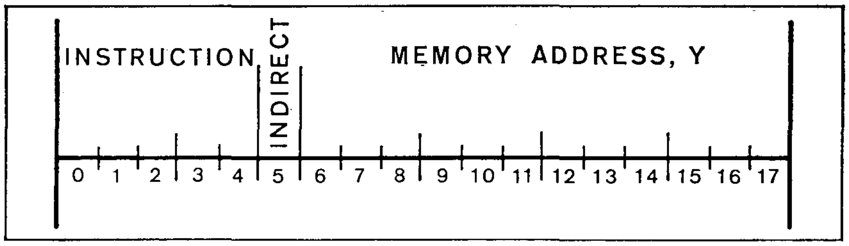
INSTRUCTION FORMAT

The Bits 0 through 4 define the instruction code; thus there are 32 possible instruction codes, not all of which are used. The instructions may be divided into two classes:

Memory reference instructions

Augmented instructions

In the memory reference instructions, Bit 5 is the indirect address bit. The instruction memory address, Y, is in Bits 6 through 17. These digits are sufficient to address 4096 words of memory.



PDP-1 Instruction Format

The augmented instructions use Bits 5 through 17, to specify variations of the basic instruction. For example, in the shift instruction, Bit 5 specifies direction of shift, Bit 6 specifies the character of the shift (arithmetic or logical), Bits 7 and 8 enable the registers (01 = AC, 10 = IO, and 11 = both) and Bits 9 through 17 specify the number of steps.

INDIRECT ADDRESSING

A memory reference instruction which is to use an indirect address will have a ONE in Bit 5 of the instruction word. The original address, Y, of the instruction will not be used to locate the operand, jump location, etc., of the instruction, as is the normal case. Instead, it is used to locate a memory register whose contents in Bits 6 through 17 will be used as the address of the original instruction. Thus, Y is not the location of the operand but the location of the location of the operand. If the memory register containing the indirect address also has a ONE in Bit 5, the indirect addressing procedure is repeated and a third address is located. There is no limit to the number of times this process can be repeated.

OPERATING SPEEDS

Operating times of PDP-1 instructions are multiples of the memory cycle of 5 microseconds. Two-cycle instructions refer twice to memory and thus require 10 microseconds for completion. Examples of this are add, subtract, deposit, load, etc. The jump, augmented and combined augmented instructions need only one call on memory and are performed in 5 microseconds.

In-Out Transfer instructions that do not include the optional wait function require 5 microseconds. If the in-out device requires a wait time for completion, the operating time depends upon the device being used.

Each step of indirect addressing requires an additional 5 microseconds.

MANUAL CONTROLS

The Console of PDP-1 has controls and indicators for the use of the operator. All computer flip-flops have indicator lights on the Console. These indicators are primarily for use when the machine has stopped or when the machine is being operated one step at a time. While the machine is running, the brightness of an indicator bears some relationship to the relative duty factor of that particular flip-flop.

Three registers of toggle switches are available on the Console. These are the Data Field-Instruction Field-Address (18 bits), the Test Word (18 bits), and the Sense Switches (6 bits). The first two are primarily used in conjunction with the operating push buttons. The Sense Switches are present for manual intervention. The use of these switches is determined by the program.

Operating Push Buttons

Start	The computer will start. The first instruction comes from the memory location indicated in the Field and Address Switches.
Stop	The computer will come to a halt at the completion of the current memory cycle.
Continue	The computer will resume operation starting at the state indicated by the lights.
Examine	The contents of the memory register indicated by the Field and Address Switches will be displayed in the Accumulator and the Memory Buffer lights.
Deposit	The word selected by the Test Word Switches will be put in the memory location indicated by the Field and Address Switches.
Read In	The photoelectric punched tape reader will start operating in the Read-In mode.

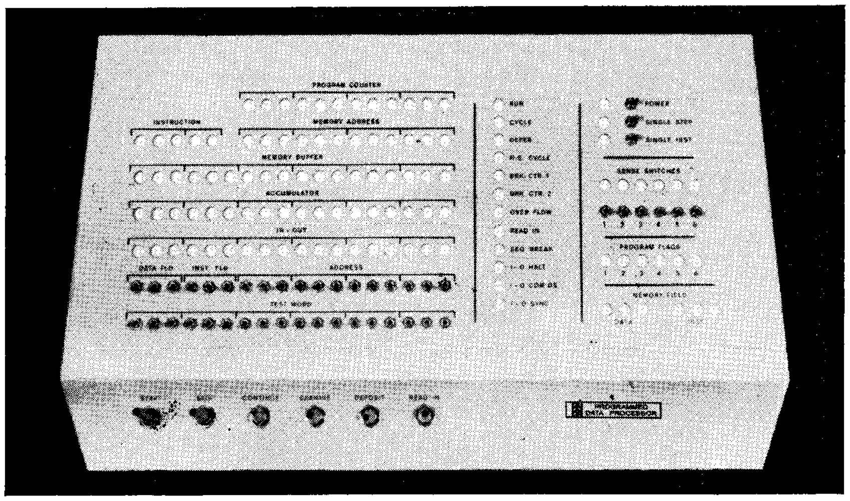
Operating Toggle Switches

Power	Turns all power to the computer on and off.
Single Step	When the Single Step Switch is on, the computer will halt at the completion of each memory cycle. This switch is particularly useful in debugging programs. Repeated opera-

	tion of the Continue Push Button will step the program one cycle at a time. The programmer is thus able to examine the state of the machine at each step.
Single Instruction	Same as Single Step except that entire instructions are stepped one at a time, regardless of the number of cycles required for their completion. (If Single Step and Single Instruction toggles are both on, the mode of operation will be single step.)

Operating Indicator Lights

Run	On while the computer is executing instructions.
Cycle	On after the completion of one or more instruction cycles with one or more to follow.
Defer	On immediately prior to the execution of any deferred cycle.
High Speed Cycle	On while the computer is executing a high speed channel, Input-Output Transfer instruction.
Break Counter 1	On while the computer is executing Cycle 1 (deposit Accumulator) and Cycle 3 (deposit Input-Output Register) of a sequence break.
Break Counter 2	On while the computer is executing Cycle 2 (deposit Program Counter) and Cycle 3 of a sequence break.



PDP-1 Control Panel

Overflow	On if overflow has occurred. (Can only be turned off or cleared by executing the Skip on Zero Overflow instruction or pressing Start.)
Read In	On while the computer is reading or trying to read punched tape in the Read-In mode.
Sequence Break	On while the computer is using the Sequence Break System.
In-Out Halt	On while the computer is executing a deferred Input-Output Transfer instruction.
In-Out Commands	On while the computer is executing any Input-Output Transfer instruction.
In-Out Sync	Used for maintenance purposes only.
Program Flags	On after the computer has executed the Set Selected Program Flag instruction or an in-out device has been activated, indicating its readiness to be serviced. (Can only be turned off or cleared by executing the Clear Selected Program Flag instruction.)
Memory Field	These indicate which memory field is currently in use.

Register Indicator Lights

Program Counter	Displays 12 bits which represent the address of the next instruction to be executed.
Instruction	Displays 5 bits which represent the basic operation code of the instruction being executed.
Memory Address	Displays 12 bits which represent the address of the instruction being executed (after Cycle 1) or the address of the operand (after succeeding cycles).
Memory Buffer	Displays 18 bits which represent the instruction being executed (operation code and address part after Cycle 1) or the 18-bit operand (after succeeding cycles).
Accumulator	Displays 18 bits which represent the results of arithmetic and logical operations.
In-Out	Displays 18 bits which represent information just transferred in or out of the computer or the results of certain arithmetic and logical operations.

INSTRUCTION LIST

This list includes the title of the instruction, the normal execution time of the instruction, *i.e.*, the time with no indirect address, the mnemonic code of the instruction, and the operation code number. In the following list, the contents of a register are indicated by C( ). Thus C(Y) means the contents of memory at Address Y; C(AC) means the contents of the accumulator; C(IO) means the contents of the in-out register. An alphabetical and numerical listing of the instructions is contained on Pages 32 to 39.

Memory Reference Instructions

ARITHMETIC INSTRUCTIONS

Add *(10 μsec)*
add Y Operation Code 40
The new C(AC) are the sum of C(Y) and the original C(AC). The C(Y) are unchanged. The addition is performed with 1's complement arithmetic. If the sum exceeds the capacity of the Accumulator Register, the overflow flip-flop will be set (see Skip Group instructions).

Subtract *(10 μsec)*
sub Y Operation Code 42
The new C(AC) are the original C(AC) minus the C(Y). The C(Y) are unchanged. The subtraction is performed using 1's complement arithmetic. If the difference exceeds the capacity of the Accumulator, the overflow flip-flop will be set (see Skip Group instructions).

Multiply Step *(10 μsec)*
mus Y Operation Code 54
If Bit 17 of the In-Out Register is a ONE, the C(Y) are added to C(AC). If IO Bit 17 is a ZERO, the addition does not take place. In either case, the C(AC) and C(IO) are shifted right one place. AC Bit 0 is made ZERO by this shift. This instruction is used in the multiply subroutine.

Divide Step *(10 μsec)*
dis Y Operation Code 56
The Accumulator and the In-Out Register are rotated left one place. IO Bit 17 receives the complement of AC Bit 0. If IO Bit 17 is ONE, the C(Y) are subtracted from C(AC). If IO Bit 17 is ZERO, C(Y) + 1 are added to C(AC). This instruction is used in the divide subroutine.

Index *(10 μsec)*
idx Y Operation Code 44
The C(Y) are replaced by C(Y) + 1. The C(Y) + 1 are left in the Accumulator. The previous C(AC) are lost. Overflow is not indicated.

Index and Skip if Positive *(10 μsec)*
isp Y Operation Code 46
The C(Y) are replaced by C(Y) + 1. The C(Y) + 1 are left in the Accumulator. The previous C(AC) are lost. If, after the addition, C(Y)

+ 1 are positive, the Program Counter is advanced one extra position and the next instruction in the sequence is skipped. Overflow is not indicated.

LOGICAL INSTRUCTIONS

Logical AND (*10 μsec*)
and Y Operation Code 02

The bits of C(Y) operate on the corresponding bits of the Accumulator to form the logical AND. The result is left in the Accumulator. The C(Y) are unaffected by this instruction.

LOGICAL AND TABLE

AC Bit	*Y Bit*	*Result*
0	0	0
0	1	0
1	0	0
1	1	1

Exclusive OR (*10 μsec*)
xor Y Operation Code 06

The bits of C(Y) operate on the corresponding bits of the Accumulator to form the exclusive OR. The result is left in the Accumulator. The C(Y) are unaffected by this order.

EXCLUSIVE OR TABLE

AC Bit	*Y Bit*	*Result*
0	0	0
0	1	1
1	0	1
1	1	0

Inclusive OR (*10 μsec*)
ior Y Operation Code 04

The bits of C(Y) operate on the corresponding bits of the Accumulator to form the inclusive OR. The result is left in the Accumulator. The C(Y) are unaffected by this order.

INCLUSIVE OR TABLE

AC Bit	*Y Bit*	*Result*
0	0	0
0	1	1
1	0	1
1	1	1

GENERAL INSTRUCTIONS

Load Accumulator (*10 μsec*)
lac Y Operation Code 20

The C(Y) are placed in the Accumulator. The C(Y) are unchanged. The original C(AC) are lost.

Deposit Accumulator (*10 μsec*)
dac Y Operation Code 24

The C(AC) replace the C(Y) in the memory. The C(AC) are left unchanged by this instruction. The original C(Y) are lost.

Deposit Address Part *(10 μsec)*
dap Y Operation Code 26

Bits 6 through 17 of the Accumulator replace the corresponding digits of memory register Y. C(AC) are unchanged as are the contents of Bits 0 through 5 of Y. The original contents of Bits 6 through 17 of Y are lost.

Deposit Instruction Part *(10 μsec)*
dip Y Operation Code 30

Bits 0 through 5 of the Accumulator replace the corresponding digits of memory register Y. The Accumulator is unchanged as are Bits 6 through 17 of Y. The original contents of Bits 0 through 5 of Y are lost.

Load In-Out Register *(10 μsec)*
lio Y Operation Code 22

The C(Y) are placed in the In-Out Register. C(Y) are unchanged. The original C(IO) are lost.

Deposit In-Out Register *(10 μsec)*
dio Y Operation Code 32

The C(IO) replace the C(Y) in memory. The C(IO) are unaffected by this instruction. The original C(Y) are lost.

Deposit Zero in Memory *(10 μsec)*
dzm Y Operation Code 34

Clears (sets equal to plus zero) the contents of register Y.

Execute *(5 μsec plus time of instruction executed)*
xct Y Operation Code 10

The instruction located in register Y is executed. The Program Counter remains unchanged (unless a jump or skip were executed). Execute may be indirectly addressed, and the instruction being executed may use indirect addressing. An *xct* instruction may execute other *xct* commands.

Jump *(5 μsec)*
jmp Y Operation Code 60

The Program Counter is reset to Address Y. The next instruction that will be executed will be taken from Memory Register Y. The original contents of the Program Counter are lost.

Jump and Save Program Counter *(5 μsec)*
jsp Y Operation Code 62

The contents of the Program Counter are transferred to the Accumulator. When the transfer takes place, the Program Counter holds the address of the instruction following the *jsp*. The Program Counter is then reset to Address Y. The next instruction that will be executed will be taken from Memory Register Y. The original C(AC) are lost.

Call Subroutine *(10 μsec)*
cal Y Operation Code 16

The address part of the instruction, Y, is ignored. The contents of the Accumulator are deposited in Memory Register 100. The contents of the

Program Counter (holding the address of the instruction following the *cal*) are transferred to the Accumulator. The next instruction that will be executed is taken from Memory Register 101. This instruction requires that the indirect bit be ZERO. The instruction may be used as part of a master routine to call subroutines.

Jump and Deposit Accumulator *(10 μsec)*
jda Y Operation Code 17

The contents of the Accumulator are deposited in Memory Register Y. The contents of the Program Counter (holding the address of the instruction following the *jda*) are transferred to the Accumulator. The next instruction that will be executed is taken from Memory Register Y + 1. This instruction requires that the indirect bit be a ONE. The instruction is equivalent to the instructions *dac Y*, followed by *jsp Y* + 1.

Skip if Accumulator and Y differ *(10 μsec)*
sad Y Operation Code 50

The C(Y) are compared with the C(AC). If the two numbers are different, the Program Counter is indexed one extra position and the next instruction in the sequence is skipped. The C(AC) and the C(Y) are unaffected by this operation.

Skip if Accumulator and Y are the same *(10 μsec)*
sas Y Operation Code 52

The C(Y) are compared with the C(AC). If the two numbers are identical, the Program Counter is indexed one extra position and the next instruction in the sequence is skipped. The C(AC) and C(Y) are unaffected by this operation.

Augmented Instructions

Load Accumulator with N *(5 μsec)*
law N Operation Code 70

The number in the memory address bits of the instruction word is placed in the Accumulator. If the indirect address bit is ONE, the complement of N (−N) is put in the Accumulator.

Shift Group *(5 μsec)*
sft Operation Code 66

This group of instructions will rotate or shift the Accumulator and/or the In-Out Register. When the two registers operate combined, the In-Out Register is considered to be an 18-bit magnitude extension of the right end of the Accumulator.

Rotate is a non-arithmetic cyclic shift. That is, the two ends of the register are logically tied together and information is rotated as though the register were a ring.

Shift is an arithmetic operation and is, in effect, multiplication of the number in the register by $2^{\pm N}$, where N is the number of shifts; plus is left and minus is right.

The number of shift or rotate steps to be performed (N) is indicated by the number of ONE's in Bits 9 thru 17 of the instruction word. Thus, Rotate Accumulator Right nine times is 671777. A shift or rotate of one place can be indicated nine different ways. The usual convention is to use the right end of the instruction word (*rar 1* = 671001).

Rotate Accumulator Right *(5 μsec)*
rar N Operation Code 671

Rotates the bits of the Accumulator right N positions, where N is the number of ONE's in Bits 9-17 of the instruction word.

Rotate Accumulator Left *(5 μsec)*
ral N Operation Code 661

Rotates the bits of the Accumulator left N positions, where N is the number of ONE's in Bits 9-17 of the instruction word.

Shift Accumulator Right *(5 μsec)*
sar N Operation Code 675

Shifts the contents of the Accumulator right N positions, where N is the number of ONE's in Bits 9-17 of the instruction word.

Shift Accumulator Left *(5 μsec)*
sal N Operation Code 665

Shifts the contents of the Accumulator left N positions, where N is the number of ONE's in Bits 9-17 of the instruction word.

Rotate In-Out Register Right *(5 μsec)*
rir N Operation Code 672

Rotates the bits of the In-Out Register right N positions, where N is the number of ONE's in Bits 9-17 of the instruction word.

Rotate In-Out Register Left *(5 μsec)*
ril N Operation Code 662

Rotates the bits of the In-Out Register left N positions, where N is the number of ONE's in Bits 9-17 of the instruction word.

Shift In-Out Register Right *(5 μsec)*
sir N Operation Code 676

Shifts the contents of the In-Out Register right N positions, where N is the number of ONE's in Bits 9-17 of the instruction word.

Shift In-Out Register Left *(5 μsec)*
sil N Operation Code 666

Shifts the contents of the In-Out Register left N positions, where N is the number of ONE's in Bits 9-17 of the instruction word.

Rotate AC and IO Right *(5 μsec)*
rcr N Operation Code 673

Rotates the bits of the combined registers right in a single ring N positions, where N is the number of ONE's in Bits 9-17 of the instruction word.

Rotate AC and IO Left *(5 μsec)*
rcl N Operation Code 663

Rotates the bits of the combined registers left in a single ring N positions, where N is the number of ONE's in Bits 9-17 of the instruction word.

Shift AC and IO Right (*5 μsec*)
scr N Operation Code 677

Shifts the contents of the combined registers right N positions, where N is the number of ONE's in Bits 9-17 of the instruction word.

Shift AC and IO Left (*5 μsec*)
scl N Operation Code 667

Shifts the contents of the combined registers left N positions, where N is the number of ONE's in Bits 9-17 of the instruction word.

Skip Group (*5 μsec*)
skp Operation Code 64

This group of instructions senses the state of various flip-flops and switches in the machine. The address portion of the instruction selects the particular function to be sensed. All members of this group have the same operation code.

The instructions in the Skip Group may be combined to form the inclusive OR of the separate skips. Thus, if Address 3000 is selected, the skip would occur if the overflow flip-flop equals ZERO or if the In-Out Register is positive. The combined instruction would still take 5 microseconds.

The intent of any skip instruction can be reversed by making Bit 5 (normally the Deferred Address Bit) equal to ONE. For example, the Skip on Zero Accumulator instruction, if deferred, becomes Do Not Skip on Zero Accumulator.

Skip on ZERO Accumulator (*5 μsec*)
sza Address 100

If the Accumulator is equal to plus ZERO (all bits are ZERO), the Program Counter is advanced one extra position and the next instruction in the sequence is skipped.

Skip on Plus Accumulator (*5 μsec*)
spa Address 200

If the sign bit of the Accumulator is ZERO, the Program Counter is advanced one extra position and the next instruction in the sequence is skipped.

Skip on Minus Accumulator (*5 μsec*)
sma Address 400

If the sign bit of the Accumulator is ONE, the Program Counter is advanced one extra position and the next instruction in the sequence is skipped.

Skip on ZERO Overflow (*5 μsec*)
szo Address 1000

If the overflow flip-flop is a ZERO, the Program Counter is advanced one extra position and the next instruction in the sequence will be skipped. The overflow flip-flop is cleared by the instruction. This flip-flop is set by an addition or subtraction that exceeds the capacity of the Accumulator. The overflow flip-flop is not cleared by arithmetic operations which do not cause an overflow. Thus, a whole series of arithmetic operations can be checked for correctness by a single *szo*. The overflow flip-flop is cleared by the "Start" Switch.

Skip on Plus In-Out Register *(5 μsec)*
spi Address 2000

If the sign digit of the In-Out Register is ZERO, the Program Counter is indexed one extra position and the next instruction in sequence is skipped.

Skip on ZERO Switch *(5 μsec)*
szs Addresses 10, 20 70

If the selected Sense Switch is ZERO, the Program Counter is advanced one extra position and the next instruction in the sequence will be skipped. Address 10 senses the position of Sense Switch 1, Address 20 Switch 2, etc. Address 70 senses all the switches. If 70 is selected all 6 switches must be ZERO to cause the skip.

Skip on ZERO Program Flag *(5 μsec)*
szf Addresses 0 to 7 inclusive

If the selected program flag is a ZERO, the Program Counter is advanced one extra position and the next instruction in the sequence will be skipped. Address 0 is no selection. Address 1 selects Program Flag 1, etc. Address 7 selects all program flags. All flags must be ZERO to cause the skip.

Operate Group *(5 μsec)*
opr Operation Code 76

This instruction group performs miscellaneous operations on various Central Processor Registers. The address portion of the instruction specifies the action to be performed.

The instructions in the Operate Group can be combined to give the union of the functions. The instruction *opr 3200* will clear the AC, put TW to AC, and complement AC. If the number minus zero is interpreted as an instruction, the IO is cleared, AC gets the complement of the TW switches, all program flags are set and the computer halts.

Clear In-Out Register *(5 μsec)*
cli Address 4000

Clears (sets equal to plus zero) the In-Out Register.

Load Accumulator from Test Word *(5 μsec)*
lat Address 2000

Forms the inclusive OR of the C(AC) and the contents of the Test Word. This instruction is usually combined with Address 200 (Clear Accumulator), so that C(AC) will equal the contents of the Test Word Switches.

Complement Accumulator *(5 μsec)*
cma Address 1000

Complements (makes negative) the contents of the Accumulator.

Halt
hlt Address 400

Stops the computer.

Clear Accumulator (*5 μsec*)
cla Address 200

Clears (sets equal to plus zero) the contents of the Accumulator.

Clear Selected Program Flag (*5 μsec*)
clf Address 01 to 07 inclusive

Clears the selected program flag. Address 01 clears Program Flag 1, 02 clears Program Flag 2, etc. Address 07 clears all program flags.

Set Selected Program Flag (*5 μsec*)
stf Addresses 11 to 17 inclusive

Sets the selected program flag. Address 11 sets Program Flag 1; 12 sets Program Flag 2, etc. Address 17 sets all program flags.

No Operation (*5 μsec*)
nop Address 0000

The state of the computer is unaffected by this operation, and the Program Counter continues in sequence.

In-Out Transfer Group (*5 μsec without in-out wait*)
iot Operation Code 72

The variations within this group of instructions perform all the in-out control and information transfer functions. If Bit 5 (normally the Indirect Address bit) is a ONE, the computer will halt and wait for the completion pulse from the device activated. When this device delivers its completion, the computer will resume operation of the instruction sequence.

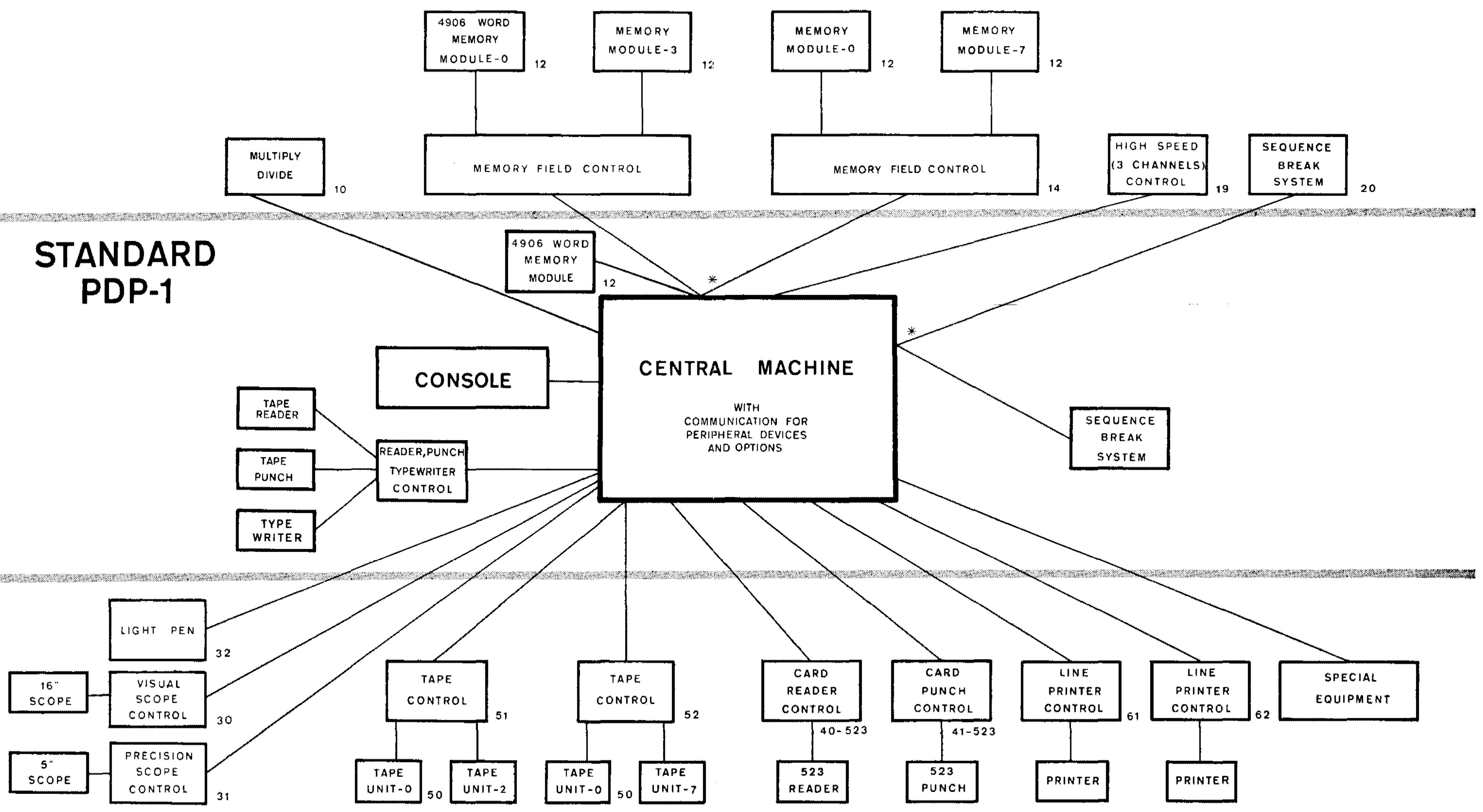
An incidental fact which may be of importance in certain scientific or real time control applications is that the time origin of operations following an in-out completion pulse is identical with the time of that pulse.

Most in-out operations require a known minimum time before completion. This time may be utilized for programming. The appropriate In-Out Transfer is given with no in-out wait (Bit 5 a ZERO and Bit 6 a ONE). The instruction sequence then continues. This sequence must include an *iot* instruction 730000 which performs nothing but the in-out wait, and the instruction must occur before the safe minimum time. A table of minimum times for all in-out devices is delivered with the computer: it lists minimum time before completion pulse and minimum In-Out Register free time.

Bit 6 determines whether a completion pulse will or will not be received from the in-out device. When it is different than Bit 5, a completion pulse will be received. When it is the same as Bit 5, a completion pulse will not be received.

In addition to the control functions of Bits 5 and 6, Bits 7 through 11 are also used as control bits serving to extend greatly the power of the *iot* instructions. For example, Bits 12 through 17, which are used to designate a class of input or output devices such as typewriters, may be further defined by Bits 7 through 11 as referring to Typewriter 1, 2, 3, etc., and whether or not the Sequence Break System is to be used.

CENTRAL PROCESSOR OPTIONS



INPUT-OUTPUT OPTIONS

NOTE:

OUTER NUMBERS DENOTE OPTION TYPES

✳ ONLY ONE OPTION MAY BE CONNECTED FOR A MACHINE

PDP-1 System Configuration Diagram

III. STANDARD AND OPTIONAL EQUIPMENT

STANDARD EQUIPMENT

Punched Tape Reader

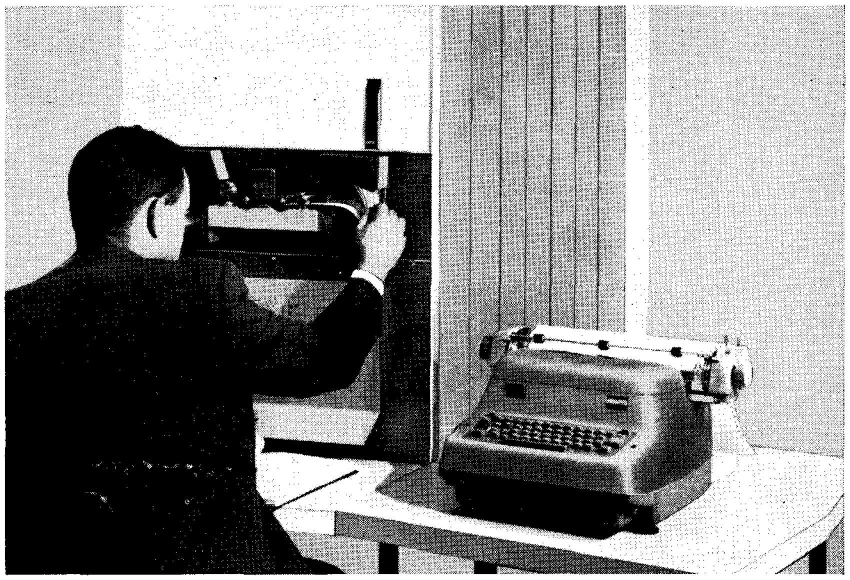
The punched tape reader of the PDP-1 is a photoelectric device capable of reading 400 lines per second. Three lines form the standard 18-bit word when reading binary punched eight-hole tape. Five, six, and seven-hole tape may also be read.

Read Punched Tape, Alphanumeric
rpa Address 0001

In this mode, one line of tape is read for each In-Out Transfer. All eight holes of the line are read. The information is left in the right eight bits of the In-Out Register, the remainder of the register being left clear.

The code of the off-line tape preparation typewriter (Friden FIO-DEC Recorder-Reproducer) contains an odd parity bit. This bit may be checked by the read-in program. The FIO-DEC Code can then be converted to the concise six-bit code used by PDP-1 merely by dropping the eighth bit (parity).

A list of characters and their FIO-DEC and Concise Codes is found on Pages 37 through 39.



High Speed Punched Tape Reader

Read Punched Tape, Binary

rpb Address 0002

For each In-Out Transfer instruction, three lines of punched tape are read and assembled in the In-Out Register to form a full computer word. For a line to be recognized in this mode, the eighth hole must be punched; *i.e.,* lines with no eighth hole will be skipped over. The seventh hole is ignored. The pattern of holes in the binary tape is arranged so as to be easily interpreted visually in terms of machine instruction.

Read Reader Buffer

rrb Address 0030

When operating in the Sequence Break Mode, the *rpa* and *rpb* instructions operate as usual but do not transfer information from the reader buffer to the IO Register. To accomplish the transfer, these instructions must be followed by an *rrb* instruction.

Read-In Mode

This is a special mode activated by the "Read-In" switch on the console. It provides a means of entering programs which neither rely on programs in memory nor require a plug board. Pushing the "Read-In" switch starts the reader in the binary mode. The first group of three lines, and alternate succeeding groups of three lines, are interpreted as "Read-In" mode instructions. Even-numbered groups of three lines are data. The "Read-In" mode instructions must be either "deposit in-out" (*dio Y*) or "jump" (*jmp Y*). If the instruction is *dio Y*, the next group of three binary lines will be stored in memory location *Y* and the reader continues moving. If the instruction is *jmp Y*, the "Read-In" mode is terminated, and the computer will commence operation at the address of the jump instruction.

Punched Tape Punch

The standard PDP-1 punched tape punch operates at a speed of 63 lines per second. It can operate in either the alphanumeric mode or the binary mode.

Punch Punched Tape, Alphanumeric

ppa Address 0005

For each In-Out Transfer instruction one line of tape is punched. In-Out Register Bit 17 conditions Hole 1. Bit 16 conditions Hole 2, etc. Bit 10 conditions Hole 8.

Punch Punched Tape, Binary

ppb Address 0006

For each In-Out Transfer instruction one line of tape is punched. In-Out Register Bit 5 conditions Hole 1. Bit 4 conditions Hole 2, etc. Bit 0 conditions Hole 6. Hole 7 is left blank. Hole 8 is always punched in this mode.

Alphanumeric Typewriter

The typewriter will operate in the input mode or the output mode.

Type Out
tyo Address 0003

For each In-Out Transfer instruction one character is typed. The character is specified by the right six bits of the In-Out Register.

Type In
tyi Address 0004

This operation is completely asynchronous and is therefore handled differently than any of the preceding in-out operations.

When a typewriter key is struck, Program Flag 1 is set. At the same time the code for the struck key is presented to gates connected to the right six bits of the In-Out Register. This information will remain at the gate for a relatively long time by virtue of the slow mechanical action. A program designed to accept typed-in data would periodically check the status of Program Flag 1. If at any time Program Flag 1 is found to be set, an In-Out Transfer instruction with Address 4 must be executed for information to be transferred. This In-Out Transfer should not use the optional in-out halt. The information contained in the typewriter's coder is then read into the right six bits of the In-Out Register. The *tyi* instruction automatically clears the IO before transferring the information. The *tyi* instruction is usually preceded by a Clear Selected Program Flag 1 instruction.

Sequence Break Mode

Two instructions are associated directly with the One-Channel Sequence Break System on the standard PDP-1.

Enter Sequence Break Mode
esm Address 0055

This instruction turns on the Sequence Break System, allowing automatic interrupts to the main sequence to occur. The contents of the Sequence Break flip-flops are unaffected by this instruction.

Leave Sequence Break Mode
lsm Address 0054

This instruction turns off the Sequence Break System, thus preventing interrupts to the main sequence. Should interrupts occur while the System is off, the Sequence Break flip-flops will, nevertheless, continue to be set.

Miscellaneous

Check Status
cks Address 0033

This instruction checks the status of various in-out devices and sets IO Bits 0 through 4 for subsequent program interrogation as follows:

IO Bit Positions	*If* ONE
0	Displayed point sensed by light pen
1	Punched tape reader busy
2	Typewriter busy
3	Typewriter key stuck
4	Punched tape punch busy

OPTIONAL EQUIPMENT

Automatic Multiply and Divide (Type 10)

This option replaces the Multiply Step and Divide Step instructions with the following instructions:

Multiply *(14 to 25 μsec)*
mul Y Operation Code 54

The product of C(AC) and C(Y) is formed in the AC and IO registers. The sign of the product is in the AC sign bit. IO Bit 17 also contains the sign of the product. The magnitude of the product is the 34-bit string from AC Bit 1 through IO Bit 16. The C(Y) are not affected by this instruction.

Divide *(30 to 40 μsec, except on overflow, 12 μsec)*
div Y Operation Code 56

The dividend must be in the AC and IO registers in the form indicated in the instruction, Multiply. IO Bit 17 is ignored. The divisor is the C(Y). At the completion of the instruction, the C(AC) are the quotient and the C(IO) are the remainder. The sign of the remainder is the sign of the dividend. If an overflow were to occur, the division does not take place, the C(AC) and C(IO) are unchanged, and the overflow indicator is set. The C(Y) are not affected by this instruction.

Memory Module (Type 12)

Each Memory Module consists of 4096 18-bit words. A maximum of eight modules may be connected to the PDP-1.

Memory Field Control (Type 13)

This control allows for memory expansion up to 16,384 18-bit words (i.e., four 4096-word memory modules). Each memory module is defined as consisting of two 2048-word fields. A select memory instruction, jump field according to the C(Y), *jfd Y*, selects any two fields to be connected to the PDP-1 and jumps to a specified location in one of the two fields.

A second instruction, change fields according to Y, *cfd Y*, replaces the contents of the two 3-bit field registers with Bits 6 through 11 of the *cfd* instruction. The Program Counter is unaffected and computation continues in sequence using the newly selected fields.

When High Speed Channel transfers are involved, the high speed channel specifies a 14-bit address for one of 16,384 words.

Memory Field Control (Type 14)

This control allows for memory expansion up to 32,768 18-bit words (i.e., eight 4096-word memory modules). Each memory module represents either a 4096 word Instruction Field or a 4096 word Data Field. A select memory instruction, jump field according to the C(Y), *jfd Y*, selects any two fields (one Instruction Field and one Data Field) to be connected to the PDP-1 and jumps to a specified location in the newly selected Instruction Field.

A second instruction, change data field according to Y, *cdf Y*, replaces the contents of the 3-bit Data Field Register with Bits 9 through 11 of the *cdf* instruction. The Program Counter and the Instruction Field Register are unaffected, and computation continues in sequence using the same Instruction Field.

When High Speed Channel transfers are involved, the high speed channel specifies a 15-bit address for one of 32,768 words.

High Speed Channel (Type 19)

The High Speed Channel is used to transfer blocks of words between memory and an in-out device, usually a high speed device such as magnetic tape. Such a channel is installed with the Tape Control Unit Type 52 or may be installed separately for special applications.

As many as three High Speed Channels may be added to the PDP-1. Each of these is automatically interrogated at the completion of each memory cycle on a priority basis. The priority is wired and fixed. The Sequence Break System has an over-all priority just below that of the lowest priority High Speed Channel.

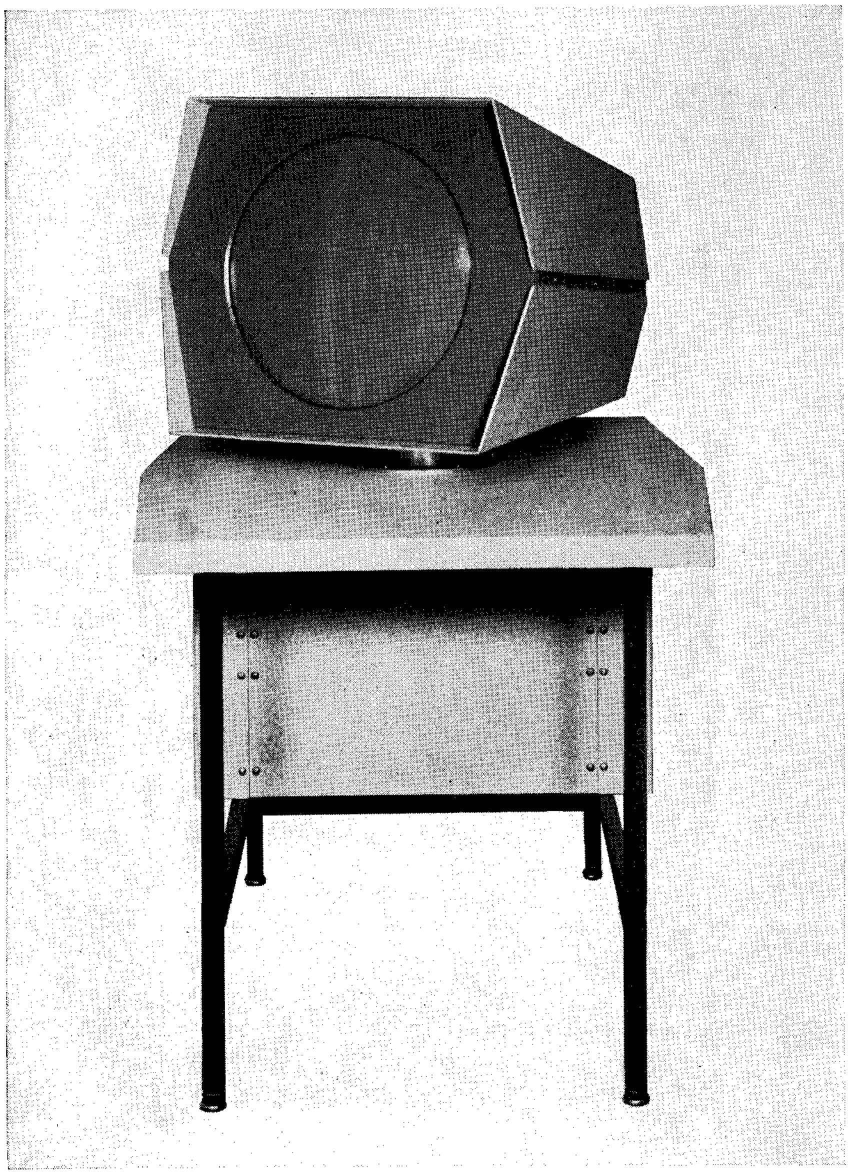
When wired to this channel, a device communicates directly with memory through the Memory Buffer Register, bypassing the IO Register. After proper initiation, data transfers proceed without disturbing the main program. If the channel has a word for or needs a word from the memory, the current program sequence pauses for one memory cycle in order to serve that channel, then continues.

Sequence Break System (Type 20)

An optional in-out control is available for PDP-1. This control, termed the Sequence Break System, allows concurrent operation of several in-out devices and the main sequence. The system has, nominally, 16 automatic interrupt channels arranged in a priority chain.

A break to a particular sequence may be initiated by the completion of an in-out device, the program or any external signal. If this sequence has priority, the C(AC), C(IO), C(PC), and the contents of the memory field flip-flops (if present) are stored in adjacent fixed locations unique to that sequence. The Program Counter is reset to the address contained in a fourth fixed location. The program is now operating in the new sequence. This new sequence may be broken by a higher priority sequence. A typical program loop for handling an in-out sequence would contain three to five instructions, including the appropriate *iot*. These are followed by load AC and load IO from the fixed locations and an indirect jump to location of the previous C(PC). This last instruction terminates the sequence.

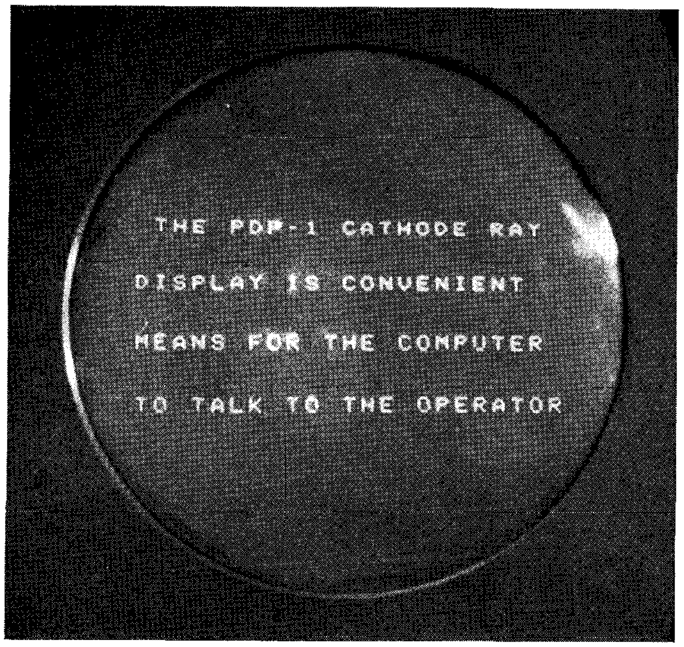
The Sequence Break System provides PDP-1 with much of the power of a multiple sequence machine or of a computer having in-out synchronizers or automatic trunks.


Cathode Ray X-Y Point Plotter

Visual CRT Display (Type 30)

The PDP-1 cathode ray tube display is useful for presentation of graphical or tabular data to the operator. For each Display instruction (730007), one point is displayed. The first 10 bits of the IO Register, Bits 0 through 9, are the Y coordinate of the point. Bits 0 through 9 of the Accumulator are the X coordinate of the point.

Information is displayed at a rate of 20,000 points per second, and the resolution is 1 part in 1024.



Cathode Ray Tube Display

Precision CRT Display (Type 31)

The operation of this 5-inch cathode ray tube display is similar to that of the Type 30. The resolution is 1 part in 4096. It comes equipped with mounting bezel to accept a camera or a photomultiplier device.

Light Pen (Type 32)

This accessory allows information to be "written" on the cathode ray tube. The pen detects displayed information, and the pen output sets a program flip-flop in the machine each time a pulse of light strikes the pen.

Card Punch Control (Type 40-523)

This control allows for on-line operation of standard card punching equipment. It contains an 80-bit buffer which is loaded from the IO Register, using an *iot* instruction for each card row punched. The control is for use with a 523 Summary Punch at speeds of 100 cards per minute.

Card Reader Control (Type 41-523)

This control provides for on-line operation of standard card reading equipment. It allows the read brush outputs to be directed to the IO Register. The control is for use with a 523 Summary Punch at speeds of 100 cards per minute.

Tape Transport (Type 50)

This transport is compatible with IBM tape formats with a recording density of 200 7-bit characters per inch and an inter-record gap of ¾ inch. The transfer rate is 15,000 characters per second at a tape speed of 75 inches per second. The method of recording is non-return-to-zero. A maximum of 24 tape transports may be connected to the PDP-1.

Programmed Tape Control Unit (Type 51)

This control transfers information between the computer and the tape one character at a time. All transfer operations, including timing, error checking and assembly of characters into computer words are performed by routines. The Type 51 allows a choice of tape format, including the standard IBM tape format described under Tape Transport (Type 50).

Automatic Tape Control Unit (Type 52)

This control automatically transfers information between the computer memory and the tape in variable-length blocks of characters. It allows computation to continue while the transfer is in process by using the High Speed Channel, which is part of the control unit. Special features include scatter-read and gather-write; automatic, bit-by-bit read-compare with core memory; automatic parity error detection while reading and writing; and rapid tape searching through its ability to skip a pre-selected number of blocks. Tape format is standard IBM as described under Tape Transport (Type 50).

Programmed Line Printer and Control (Type 61)

This is an on-line printing station capable of operating at 150 lines per minute (120 columns per line with 64 characters per column). All transfer operations, formatting and control functions are under program control.

Automatic Line Printer and Control (Type 62)

This is an on-line printing station capable of operating at 600 lines per minute (120 columns per line with 64 characters per column). A simple one-line buffer is used. The appropriate *iot* instruction is repeated to fill the buffer. The order to print is then given. Following the completion of the line print, the printer returns a completion pulse and spaces the paper.

Real Time Clock

A special input register may be connected to operate as a real time clock. This is a counting register operated by a crystal controlled oscillator.

The state of this counter may be read at any time by the appropriate In-Out Transfer instruction. The computer stops only long enough to provide synchronization with the clock oscillator, then resumes operation in phase with it.

IV. PROGRAM LIBRARY

The Basic Program Library for PDP-1 is designed to provide the nucleus of a growing system of programs and subroutines. Among the programs in use are:

DECAL

Digital Equipment Compiler, Assembler and Linking Loader is an integrated program for PDP-1 incorporating in one system all of the essential features of advanced compilers, assemblers and loaders. The outstanding features of DECAL are:

- *Open-Ended Programming System.* DECAL can be modified without a detailed understanding of its internal operation by means of a recursive definition facility based on a skeleton compiler with a basic set of logical capabilities. The skeleton compiler can act as a bootstrap for implementing additional features or deleting or changing existing facilities.
- *Efficient Use of Storage Capacity.* All available memory space will be used before DECAL runs out of space. This is accomplished by means of a single table that meets all of the storage needs of DECAL. When any feature of DECAL is removed, all of the associated space is again made available.
- *Efficient Use of Time.* DECAL processing takes full advantage of the speeds of the standard PDP-1 input-output equipment (400 lines per second reader and 63 lines per second punch).
- *One Pass Compiler-Assembler.* The symbolic tape (usually prepared off-line) is only read one time. It is not stored in memory. A relocatable tape ready for the linking loader is punched as the symbolic tape is read.
- *Efficient Object Program.* Since DECAL allows compiler and assembler statements to be freely intermixed at the discretion of the programmer, the object program can be as efficient as desired.
- *Variable Length Symbols.* Symbol lengths are not restrictive and can be hundreds of characters long if necessary.
- *Program Integration and Relocation.* At load time, individual routines and subroutines can be loaded in any order. All cross-referencing is done automatically, and the resulting program is arranged compactly starting at any specified origin. The names of subroutines required at run time and not yet loaded are listed on the typewriter so that they may be subsequently loaded.
- *Use of ALGOL.* DECAL includes that part of ALGOL which is compatible with the PDP-1 Computer. Compiler (algebraic)

and assembler statements can be written taking full advantage of the ALGOL reference language due to the unique character set used by the PDP-1.

- *Recursively Defined Macro Instructions.* Full use of such complex items as recursively defined macro instructions can be made when writing DECAL assembler statements.
- *Use of Arbitrary Languages.* Arbitrary reference languages can be defined and incorporated into DECAL to handle special problems.
- *Use of Floating Point Systems.* DECAL-compatible, single and double precision floating point systems can be incorporated.
- *Generalized subscripting, indexing and address arithmetic facilities* can be incorporated.

OTHER PROGRAMS

FRAP Assembly Program is a basic assembler designed to operate with a very minimum amount of internal storage.

Standard Function Generator Subroutines for single precision fixed point arithmetic. These include: sine, cosine, arctangent, square root, exponential and logarithmic subroutines.

Single Precision Floating Point Package for arithmetic using an 18-bit fraction and an 18-bit exponent. The package includes standard function generator subroutines.

Double Precision Floating Point Package for arithmetic using a 36-bit fraction and an 18-bit exponent. The package includes standard function generator subroutines.

Basic Double Precision Fixed Point Subroutines including addition, subtraction, multiplication and division.

Utility Routine Package including a wide range of input-output subroutines and debugging aids.

V. APPENDIX

ABBREVIATED INSTRUCTION LIST

Basic Instructions

Instruction	*Code #*	*Explanation*	*Oper. Time (μsec)*	*Page Ref.*
add Y	40	Add C(Y) to C(AC)	10	12
and Y	02	Logical AND C(Y) with C(AC)	10	13
cal Y	16	Equals jda 100	10	14
dac Y	24	Put C(AC) in Y	10	13
dap Y	26	Put contents of address part of AC in Y	10	14
dio Y	32	Put C(IO) in Y	10	14
dip Y	30	Put contents of instruction part of AC in Y	10	14
dis Y	56	Divide step	10	12
dzm Y	34	Make C(Y) zero	10	14
idx Y	44	Index (add one) C(Y), leave in Y & AC	10	12
ior Y	04	Inclusive OR C(Y) with C(AC)	10	13
iot Y	72	In-out transfer, see below		19
isp Y	46	Index and skip if result is positive	10	13
jda Y	17	Equals dac Y and jsp Y+1	10	19
jfd Y	12	Jump memory field according to C(Y)	10	26
jmp Y	60	Take next instruction from Y	5	14
jsp Y	62	Jump to Y and save program counter in AC	5	14
lac Y	20	Load the AC with C(Y)	10	13
law N	70	Load the AC with the number N	5	15

Basic Instructions (Continued)

Instruction	*Code #*	*Explanation*	*Oper. Time (μsec)*	*Page Ref.*
law −N	71	Load the AC with the number −N	5	15
lio Y	22	Load IO with C(Y)	10	14
mus Y	54	Multiply step	10	12
opr	76	Operate, see below	5	18
sad Y	50	Skip next instruction if C(AC) $\neq$ C(Y)	10	15
sas Y	52	Skip next instruction if C(AC) = C(Y)	10	15
sft	66	Shift, see below	5	15
skp	64	Skip, see below	5	17
sub Y	42	Subtract C(Y) from C(AC)	10	12
xct Y	10	Perform instruction in Y	5+extra	14
xor Y	06	Exclusive OR C(Y) with C(AC)	10	13

Operate Group

cla	760200	Clear AC	5	19
clf	760001-7	Clear selected Program Flag	5	19
cli	764000	Clear IO	5	18
cma	761000	Complement AC	5	18
hlt	760400	Halt	5	18
lat	762200	Load AC from Test Word switches	5	18
nop	760000	No operation	5	19
stf	760011-7	Set selected Program Flag	5	19

In-Out Transfer Group

Instruction	*Code #*	*Explanation*	*Oper. Time (μsec)*	*Page Ref.*
cdf	720X74	Change data field	5	26
cfd	72XX74	Change fields	5	25
cks	730033	Check status	5	24
dpy	730007	Display one point on CRT	50	27
esm	720055	Enter Sequence Break Mode	5	24
lsm	720054	Leave Sequence Break Mode	5	24
ppa	730005	Punch punched tape alphanumeric		23
ppb	730006	Punch punched tape binary		23
rpa	730001	Read punched tape alphanumeric		22
rpb	730002	Read punched tape binary		23
rrb	720030	Read Reader Buffer	5	23
tyi	720004	Read typewriter input switches	5	24
tyo	730003	Type out		24

Skip Group

sma	640400	Skip on minus AC	5	17
spa	640200	Skip on plus AC	5	17
spi	642000	Skip on plus IO	5	18
sza	640100	Skip on ZERO (+0) AC	5	17
szf	64000F	Skip on ZERO flag (F = flag #)	5	18
szo	641000	Skip on ZERO overflow (and clear overflow)	5	17
szs	6400S0	Skip on ZERO sense switch (S = switch #)	5	18

Shift/Rotate Group

Instruction	*Code #*	*Explanation*	*Oper. Time (μsec)*	*Page Ref.*
ral	661	Rotate AC left	5	16
rar	671	Rotate AC right	5	16
rcl	663	Rotate combined AC & IO left	5	16
rcr	673	Rotate combined AC & IO right	5	16
ril	662	Rotate IO left	5	16
rir	672	Rotate IO right	5	16
sal	665	Shift AC left	5	16
sar	675	Shift AC right	5	16
scl	667	Shift combined AC & IO left	5	17
scr	677	Shift combined AC & IO right	5	17
sil	666	Shift IO left	5	16
sir	676	Shift IO right	5	16

NUMERICAL INSTRUCTION LIST

Code	*Instruction*	*Code*	*Instruction*
00	*	40	add
02	and	42	sub
04	ior	44	idx
06	xor	46	isp
10	xct	50	sad
12	jfd	52	sas
14	*	54	mus
16	cal	56	dis
17	jda	60	jmp
20	lac	62	jsp
22	lio	64	skp
24	dac	66	Shift
26	dap	70	law
30	dip	71	law
32	dio	72	iot
34	dzm	74	*
36	*	76	opr

* Spare code, computer will halt.

ALPHANUMERIC CODES

Character		*FIO-DEC Code*	*Concise Code*
a	A	61	61
b	B	62	62
c	C	263	63
d	D	64	64
e	E	265	65
f	F	266	66
g	G	67	67
h	H	70	70
i	I	271	71
j	J	241	41
k	K	242	42
l	L	43	43
m	M	244	44
n	N	45	45
o	O	46	46
p	P	247	47
q	Q	250	50
r	R	51	51
s	S	222	22
t	T	23	23
u	U	224	24
v	V	25	25
w	W	26	26
x	X	227	27
y	Y	230	30
z	Z	31	31

Alphanumeric Codes (Continued)

Character			*FIO-DEC Code*	*Concise Code*
0	$\rightarrow$	(right arrow)	20	20
1	″	(double quotes)	01	01
2	′	(single quote)	02	02
3	$\sim$	(not)	203	03
4	$\supset$	(implies)	04	04
5	$\vee$	(or)	205	05
6	$\wedge$	(and)	206	06
7	$<$	(less than)	07	07
8	$>$	(greater than)	10	10
9	$\uparrow$	(up arrow)	211	11
(	[		57	57
)	]		255	55
—	|	(non-spacing overstrike and vertical)	256	56
$-$	$+$	(minus and plus)	54	54
·	—	(non-spacing middle dot and underline)	40	40
,	=		233	33
.	$\times$	(period and multiply)	73	73
/	?		221	21
Lower Case			272	72
Upper Case			274	74
Space			200	00
Backspace			75	75
Tab			236	36

Alphanumeric Codes (Continued)

Character	*FIO-DEC Code*	*Concise Code*
Carriage Return	277	77
Tape Feed	00	00
Red *	—	35
Black *	—	34
Stop Code	13	—
Delete	100	—

*Used on type-out only, not on keyboard.

DEC TECHNICAL BULLETINS

NOTE: DEC digital circuit packages are being renamed "Modules." As new bulletins are published, the two basic product lines will be referred to as "Laboratory Modules" instead of "Digital Test Equipment" and as "System Modules" rather than "System Building Blocks."

DIGITAL MODULES (A-702) — short form catalog listing DEC's complete product line.

DEC 10 Megacycle Building Blocks (A-710) — describes new 5000 Series Digital Test Equipment and 6000 Series System Building Blocks.

Expanded 100 Series DEC Digital Test Equipment (B-100) — describes DEC's 5 megacycle patchcord units.

New 3000 Series DEC Digital Test Equipment (B-3000) — describes DEC's 500 kilocycle patchcord units.

Expanded 1000 Series DEC System Building Blocks (C-1000A) — describes DEC's 5 megacycle plug-in units.

New 4000 Series DEC System Building Blocks (C-4000A)—describes DEC's 500 kilocycle plug-in units.

DEC Basic Logic Kit (E-150) —describes a basic selection of DEC Digital Test Equipment and Accessories which can be used to perform a variety of logical operations.

DEC Programmed Data Processor (F-11A)—describes DEC's PDP-1 high-speed, solid state, general purpose computer.

DEC Memory Tester Type 1512 (F-1512A) — describes DEC's 1500 Series testers for coincident current core memories.

DEC Memory Tester Type 1514 (F-1514) — describes DEC's 1500 Series testers for word address and coincident current core memories.

DEC Automatic Memory Core Tester Type 2101 (F-2101) — describes DEC's automatic tester for ferrite magnetic memory cores.

DEC Memory Exerciser Type 2201 (F-2201) — describes DEC's exercisers for coincident current core memory systems.

Copies of the above bulletins are available on request from the DEC Sales Department, 146 Main Street, Maynard, Massachusetts, or 8820 Sepulveda Boulevard, Los Angeles 45, California.

PDP-1

INPUT-OUTPUT SYSTEMS MANUAL

(PRELIMINARY MANUAL)

F-25

INDEX

INDEX

THE ELECTRICAL INTER-CONNECTION AND PROGRAMMING FOR DEVICES CONNECTED WITH PDP-1

Introduction

This is a discussion of the electrical, physical, and programming aspects of devices connected to PDP-1.

The PDP-1 is composed of standard DEC building blocks whose logical characteristics and capabilities are discussed in DEC literature.* The following comments on the inter-connection of equipment will assume the reader has a fairly basic knowledge of Boolean Algebra and has vaguely perused the DEC Logic Handbook, A-400-B. The reader should also be familiar with the register layout of the PDP-1 and its programming as described in the Programmed Data Processor manual, F-15A.

Information and Control

Two general types of signal flow are in PDP-1. These are for information transfers and for control pulses. An example of an information transfer in PDP-1 is when the contents of the Memory Buffer register (MB) are read into the Accumulator (AC). In this case, when the transfer is made, a whole register, or 18 bits of information, is transfered simultaneously. The command that the Memory Buffer register is to be placed in the Accumulator is done by a single line which is called a control line. That is, the 18 bits of the Memory Buffer register go to the Accumulator, and the control line "samples" these lines at an appropriate time.

Basically, external information controls are handled in the same manner. Information is presented to external devices, and the device is given a control signal under program control which says to take the information and proceed. A symbolic language program may be written for the paper tape punch to perforate one line of paper tape:

```
character     77                   ,code to be punched on tape
               .
               .
               .
punch         lio character        ,contents of "character" replace
                                    In Out register
              ppa                  ,command to punch one line of tape
```

In the above program, if the program is started in register, "punch", the instruction, lio character, would place the octal

77, in the In Out register (IO). The ppa or punch paper alphanumeric instruction would transfer the last 8 bits of a character, 00 111 111 (77), to the punch logic. The punch would then perforate a line of the tape. Information is transfered to paper tape just as information is transfered from a memory location to the arithmetic element. The character 00 111 111 would correspond to the information and any control signals generated by the command punch paper alpha (ppa), being given would constitute the control to the punch logic.

The in-out transfer command has the operation code 72XXXX. The address portion of the command has special meaning. The six bits, 12-17, address one of 64 devices. Bits 5 and 6 control synchronization and bits 7-11 may be used for special purposes.

The in-out transfer command is the basic method of transfering information between PDP-1 and other devices.

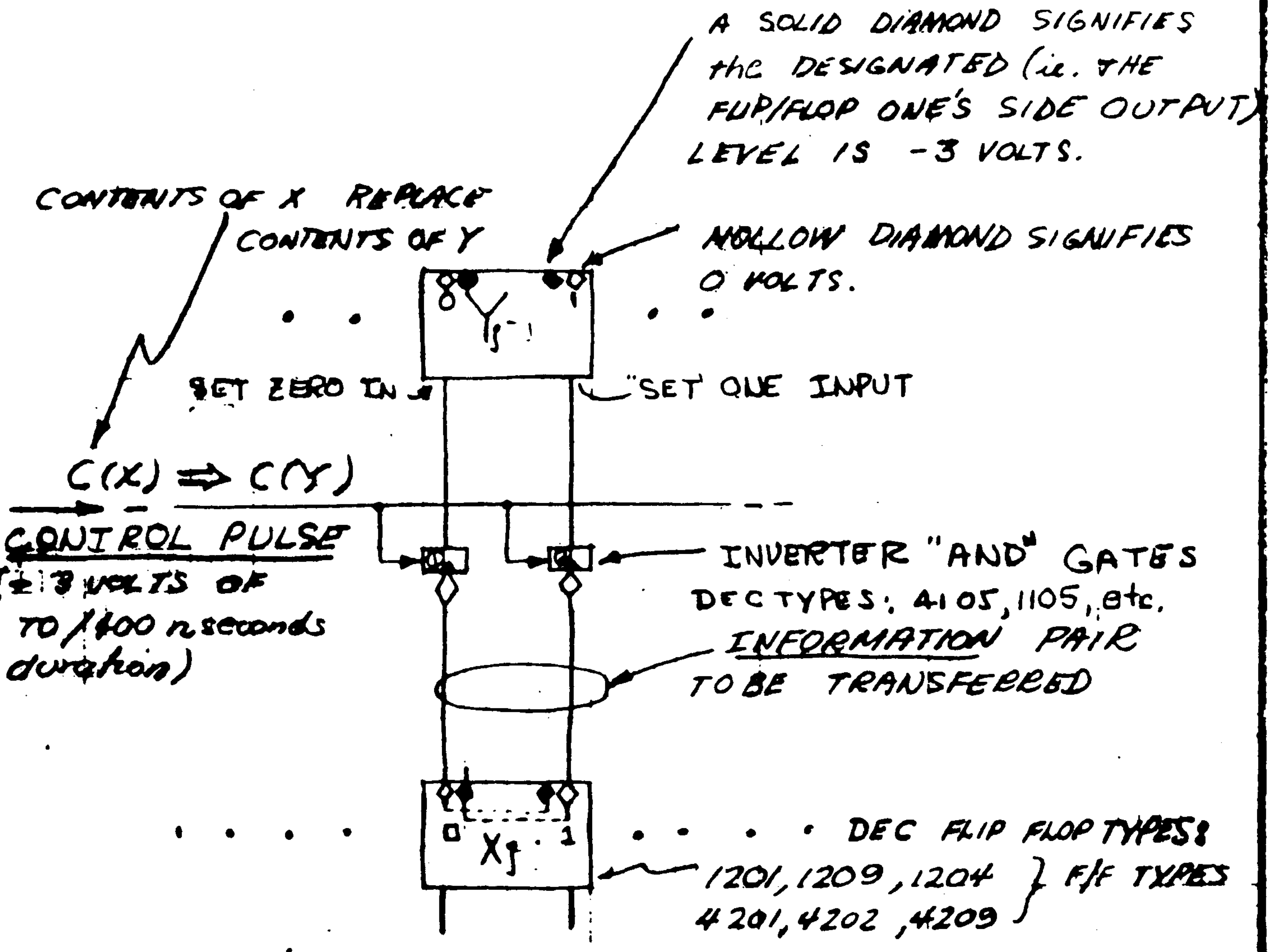
There are several methods (some of which are special options) of transfering information that relieve the program of the details associated with the transfer. Logic within a device may request that information be transfered directly from core memory while the program is running. That is, the program is momentarily interrupted while data are transfered. This is called a high speed data channel.

The Sequence Break System permits a program sequence to be interrupted at a time a condition has been satisfied. The Sequence Break relieves the program of the details associated with the status of input-output devices.

Fundamental Transfer of Information

Fundamentally, information can be transfered between one register and another register, as shown in figure 1. This method, which involves only one step is known as a jam transfer. Figure 1 shows the jth flip-flops for registers Y and X. At a given time, information is to be transfered from register X to register Y. The control event is entitled "C(X) = C(Y)", or, the contents of register X replace the contents of register Y. A pulse on the control line will transfer the information from register X to register Y by "sampling" the output voltages of register X. This process may also be referred to as "strobing" , "sampling", "transfering" or "reading in".

If register X contains a one, then the transistor "and" gate labeled 2 will conduct when the control pulse is given causing the



A SOLID DIAMOND SIGNIFIES the DESIGNATED (i.e. THE FLIP/FLOP ONE'S SIDE OUTPUT) LEVEL IS -3 VOLTS.
CONTENTS OF X REPLACE CONTENTS OF Y
HOLLOW DIAMOND SIGNIFIES 0 VOLTS.
0
1
Yj
SET ZERO IN
SET ONE INPUT
C(X) ⇒ C(Y)
CONTROL PULSE (± 3 VOLTS OF 70/400 n seconds duration)
INVERTER "AND" GATES DEC TYPES: 4105, 1105, etc.
INFORMATION PAIR TO BE TRANSFERRED
0
Xj
1
DEC FLIP FLOP TYPES: 1201, 1209, 1204 4201, 4202, 4209 } F/F TYPES

JAM TRANSFER OF INFORMATION BETWEEN TWO REGISTERS

FIGURE 1

set one input to be pulsed. Transistors 1 and 2 act as "and" gates. The new contents of Y do not depend on the previous contents. The contents of X are unaffected by the information transfer. The logical conventions of figure 1 will be used throughout DEC PDP-1 diagrams.

Figure 2 shows the two step method of transmitting information between two flip-flop registers. By allowing the transfer to be effected in two steps, the number of "and" gates has been halved.

First, each bit of the register is set to zero. A short time later (0.2 microseconds for 5 MC logic and 2 microseconds for 500 KC logic), a second pulse transfers (strobes, or reads in) the information into the one side of the flip-flops. Using this system, information need only be specified by the polarity on a single line rather than two lines.

<u>The Use of In-Out Transfer Commands For Control Purposes</u>

The PDP-1 uses the address of in out transfer (iot) instructions to select various devices. Actually, the decoding for the instruction, 72XXXX, is as follows:

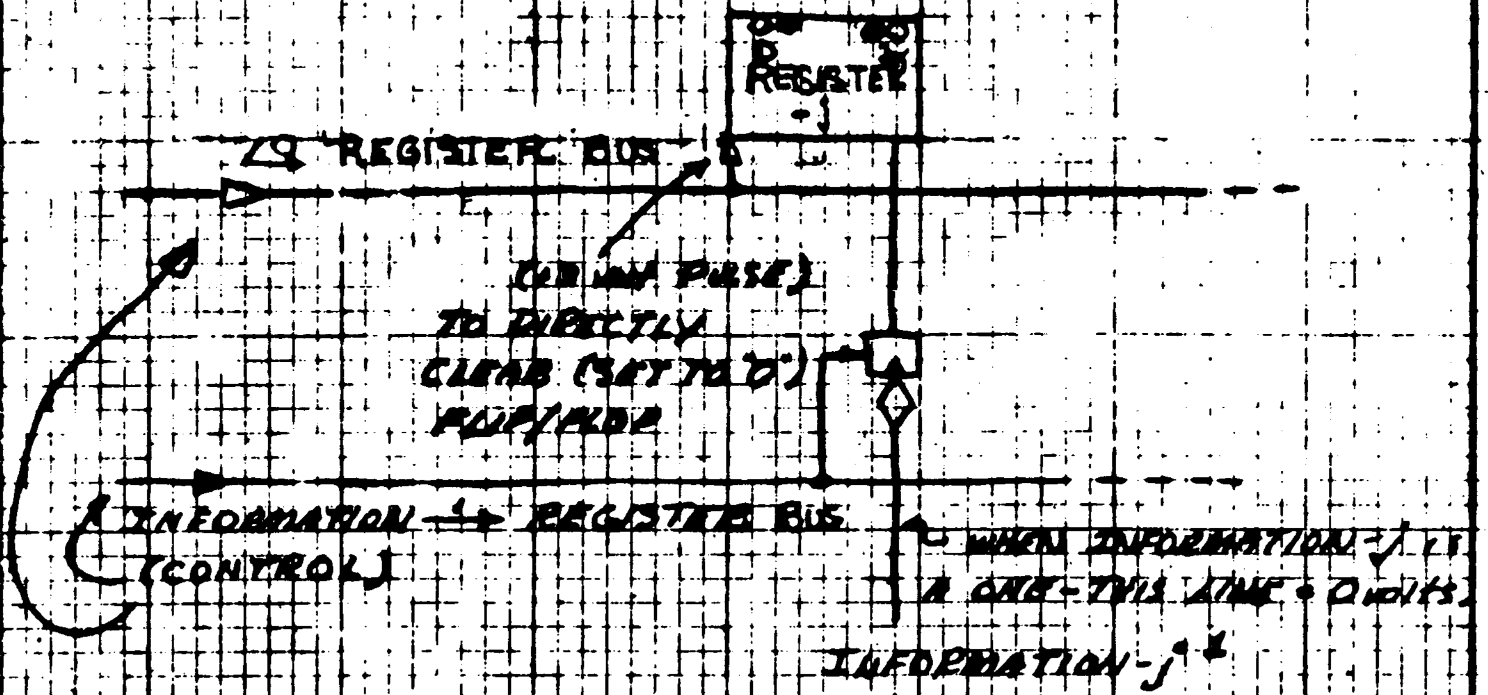
Bits	Use
0-4	11101 Instruction bit code for in out transfer
5-6	Used for device synchronization
7-11	Unused - May be anything
12-17	Addresses 1 of 64 devices

The address decoding scheme will be described in greater detail in the following sections. One of 64 pulses will be emitted on a particular wire corresponding to the address or last 6 bits of the in out transfer instruction.

Really, 64 pulse pairs may be formed. The pulse pair, which corresponds to an address is sent on two separate wires and provides the following:

1. 2.5 microseconds after the command is given, a pulse is available for clearing a register.

2. 2.5 microseconds later, or 5 microseconds after the beginning of the command, a pulse is available to transfer the information. Thus, the two step transfer method may be used.



SEQUENCE OF EVENTS:

1. REGISTER IS CLEARED
   (0 → REGISTER)
2. INFORMATION IS TRANSFERED
   (STROBED/SAMPLED) INTO
   REGISTER (INFO → REGISTER)

TWO STEP METHOD TO TRANSFER INFORMATION TO A REGISTER
FIGURE 2

For example, an in-out transfer command, turn on (ton), may be defined in the program assembly language and connected within PDP-1 with the operation code 72 0010. Now, each time a program gives a ton command, a pulse of 0.4 microseconds duration and 3 volt amplitude will be emitted on a line labeled "ton". Similarly, a turn off pulse may be emitted for a tof command (code 72 0011). The following program with figure 3 forms a 50 KC square wave generator:

```
on      72 0010              ,sets the flip-flop to a one state
        jmp off              ,dummy instruction - goes to next
                             ,instruction
off     tof                  ,sets the flip-flop to the zero state
        jmp on               ,returns to the register labeled begin
                             ,for repeating the sequence
```

A pulse appears on the ton line at some time. Five microseconds elapse as the "jmp off" command is effected. Five microseconds later, tof occurs and a pulse clears the flip-flop. The jmp instruction requires 5 microseconds and the process repeats. Every 10 microseconds, the flip-flop either gets set or cleared. Thus, output of the flip-flop is a square wave with a period of 20 microseconds. The times for an on or an off level can be varied in 5 microseconds intervals.

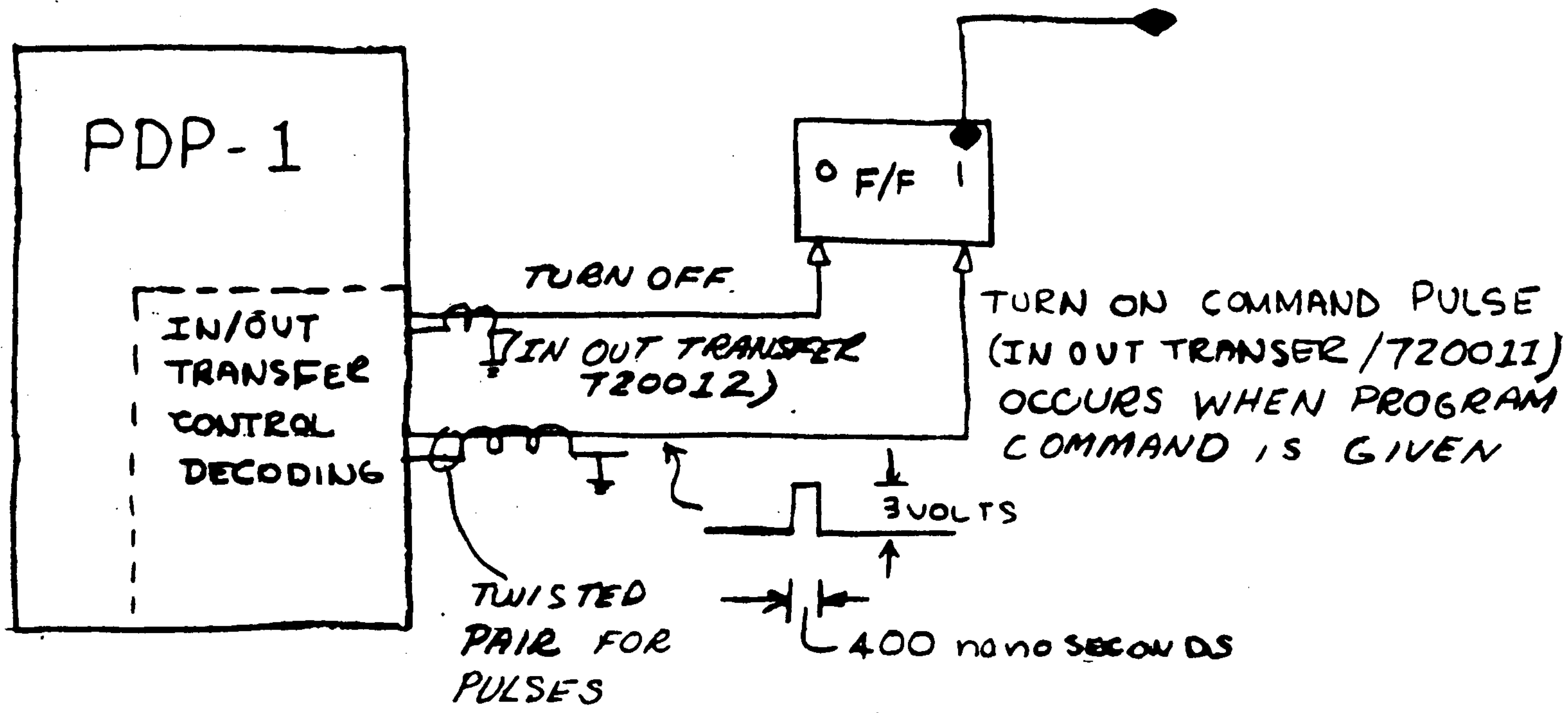
TRANSMITTING INFORMATION TO A DEVICE WITH THE PROGRAM

One of the most common devices that can be connected to the PDP-1 is a flip-flop register of up to 18 bits. The register, for example, might drive a digital to analog decoding network to supply an analog voltage, or a series of relays, or a visual display buffer from which decimal numbers are decoded and viewed, etc.

Given, there is 18 bits of information in the in-out register, we wish to transfer these bits (information) to an external register.

Figure 4 shows information of the In-Out register Bit-j being transmitted to the 18-bit external buffer register (EB). To transmit information to External Buffer Register, an in out command address is assigned which we will call teb, with operation code 72 0010. The one's side outputs of the In-Out register (IO), are sent to EB. When a bit of the IO is a one then the respective line is 0 volts, and when the IO register bit is a zero, the line is at -3 volts. When the program gives the teb, two things happen:

1. 2.5 microseconds after the command begins, a positive pulse clears EB.


PDP-1
IN/OUT TRANSFER CONTROL DECODING
0 F/F 1
TURN OFF
(IN OUT TRANSFER 720012)
TURN ON COMMAND PULSE (IN OUT TRANSER/720011) OCCURS WHEN PROGRAM COMMAND IS GIVEN
3 VOLTS
400 nano SECONDS
TWISTED PAIR FOR PULSES

PDP-1 WITH TWO CONTROL PULSES

FIGURE 3

2. 2.5 microseconds later, or 5 microseconds after the beginning of teb, the information from the one's side outputs of the in-out register is read into (or transfered to) EB. Thus, 5 microseconds after the beginning of teb, EB contains the same information as IO.

A block diagram of the connection is shown in figure 4B. There are 18 lines of In-Out register information, and 2 control pulses. Each of the control pulses requires either a coaxial cable or a pair of wires which are twisted, thus, a total of 22 wires form the interconnection.

RECEIVING INFORMATION FROM A DEVICE WITH THE PROGRAM

When information is received from a remote device, the roles of transmitter and receiver mentioned above are reversed. A register of information must be transmitted to PDP-1, as in the case of an analog to digital converter attached to the PDP-1.

Figure 5 shows how one bit of information would be read into the in-out register under program command. In this case, a switch position is to be sampled, and 18 switches form a register. In this case, a program executes a command which emits pulses that:

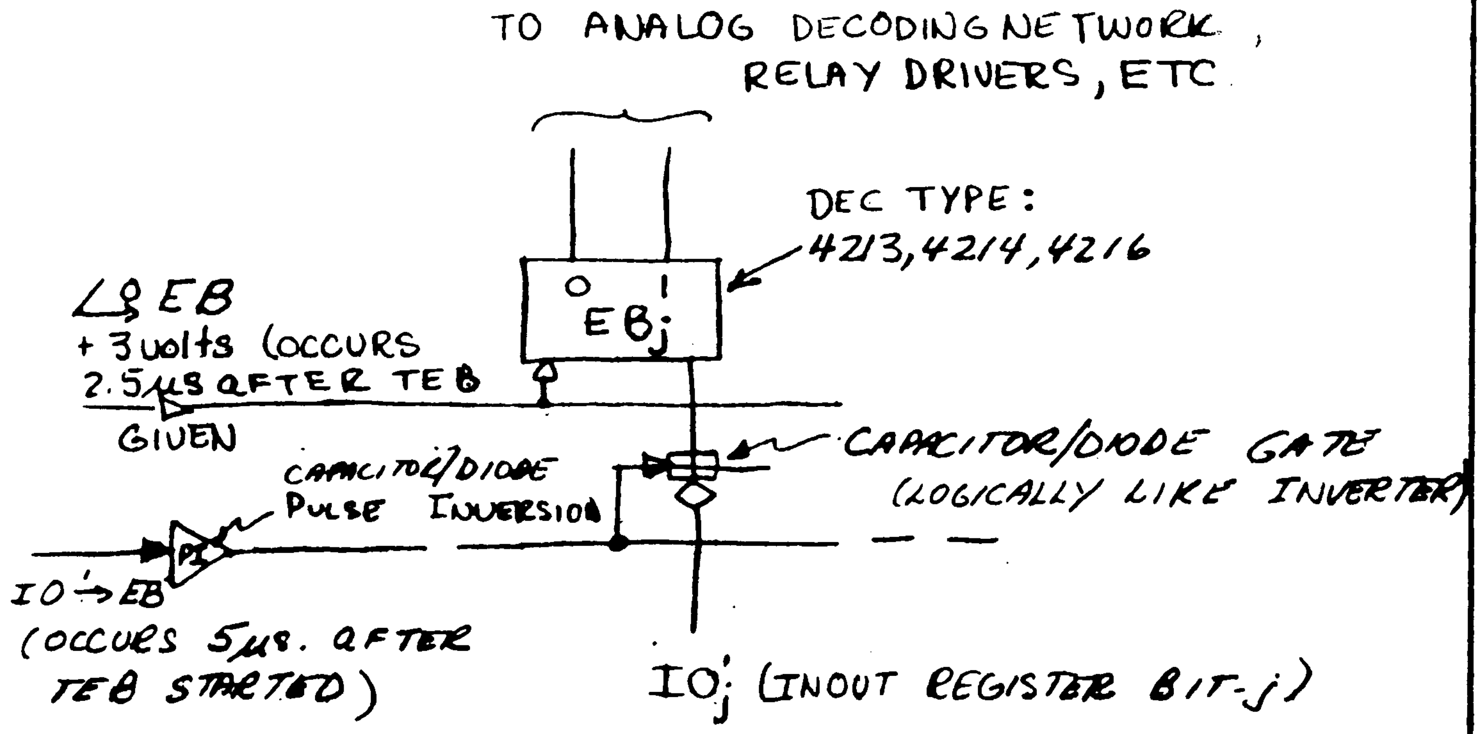
1. Clear the in-out register.

2. Sample the information and place it into the in-out register.

The mechanism for the information gating is shown in figure 5. That is, a level enables a capacitor diode gate, then a control pulse "ands" with the level to conditionally form a pulse for each information bit, thus, the IO register flip-flop is set to a one if the gate is enabled.

The Connection of An Analog To Digital Converter With PDP-1

The connection of a 12 bit analog to digital converter to operate with PDP-1 is shown in figure 6. The line labeled "start convert" is a control pulse to the converter that commands a conversion. From the time the "convert" command is given, the converter requires 40 microseconds to determine a 12-bit digital number proportional to the analog input voltage. The convert command, cnv, has the operation code 72 0041.



TO ANALOG DECODING NETWORK, RELAY DRIVERS, ETC
DEC TYPE: 4213, 4214, 4216
0 1
EB_j
↓0 EB
+3 volts (OCCURS 2.5μs AFTER TEB GIVEN
CAPACITOR/DIODE GATE (LOGICALLY LIKE INVERTER)
CAPACITOR/DIODE PULSE INVERSION
PI
IO' → EB
(OCCURS 5μs. AFTER TEB STARTED)
IO'_j (INOUT REGISTER BIT-j)

FIGURE 4a

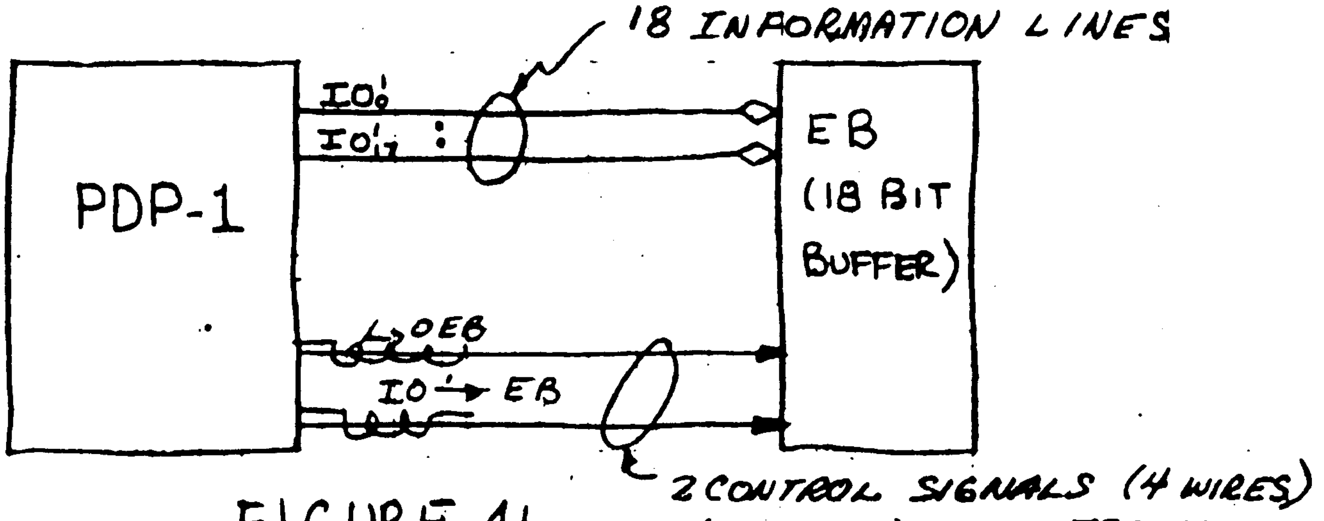
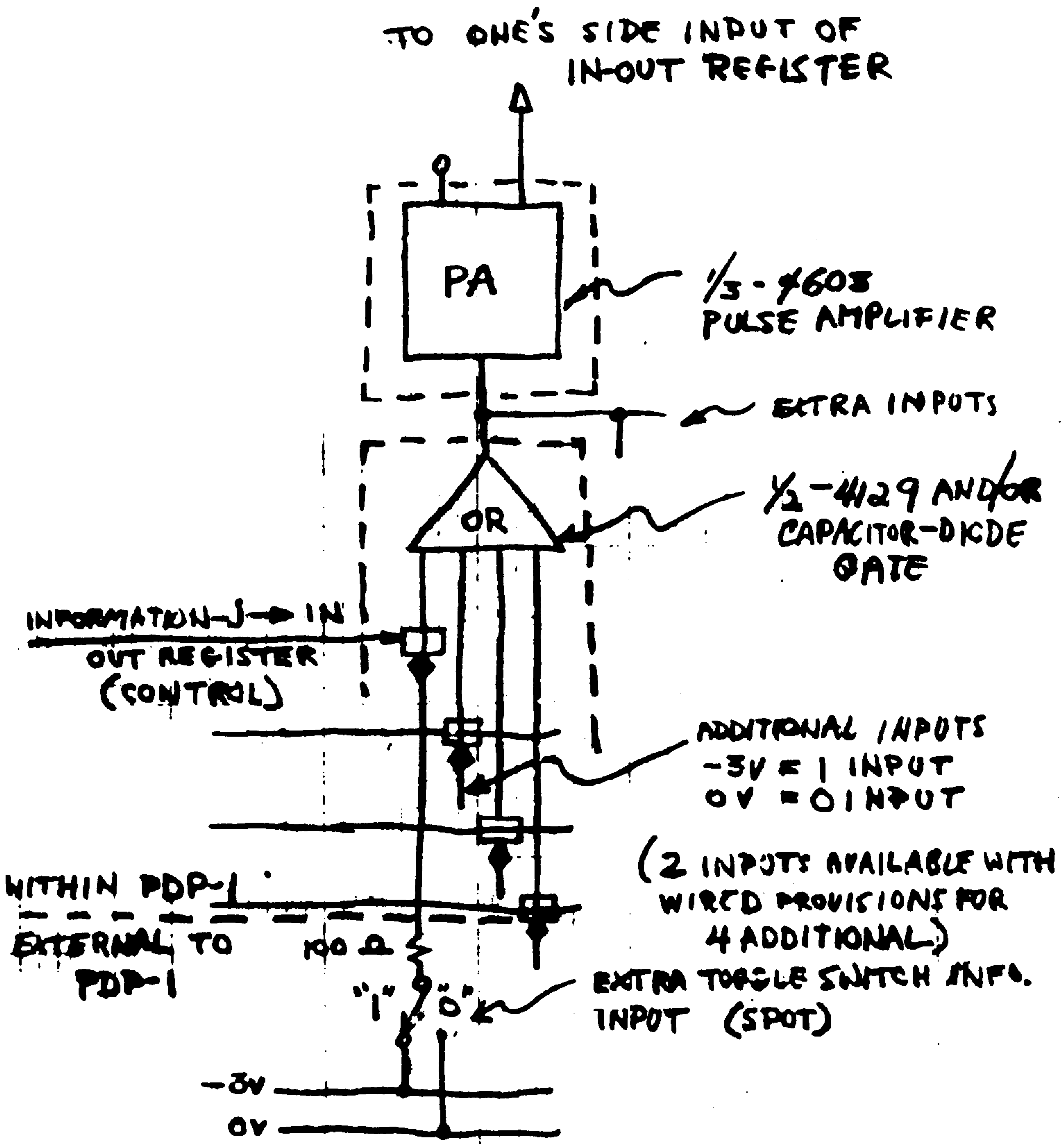


FIGURE 4b

TRANSMITTING INFORMATION TO EXTERNAL BUFFER (EB) USING COMMAND: TRANSFER TO EB (TEB)


TO ONE'S SIDE INPUT OF IN-OUT REGISTER
PA
1/3 - 4603 PULSE AMPLIFIER
EXTRA INPUTS
OR
1/2 - 4129 AND/OR CAPACITOR-DIODE GATE
INFORMATION → IN OUT REGISTER (CONTROL)
ADDITIONAL INPUTS -3V = 1 INPUT 0V = 0 INPUT
(2 INPUTS AVAILABLE WITH WIRED PROVISIONS FOR 4 ADDITIONAL)
WITHIN PDP-1
EXTERNAL TO PDP-1
100 Ω
"1"
"0"
EXTRA TOGGLE SWITCH INFO. INPUT (SPOT)
-3V
0V

IN-OUT REGISTER INPUT MIXER FOR ONE BIT

FIGURE 5

The digital number is transfered from the converter to the In-Out register under program control. The command, read converter buffer, (rcb = 72 0031) reads the converted bits into the 12 most significant bits of the In-Out register.

Figure 6 shows the decoding within PDP-1 necessary for control. The information is read into the input mixer (which sets the In-Out) and three pulse lines are decoded which:

1. Clear the in-out register (clear portion of rcb).

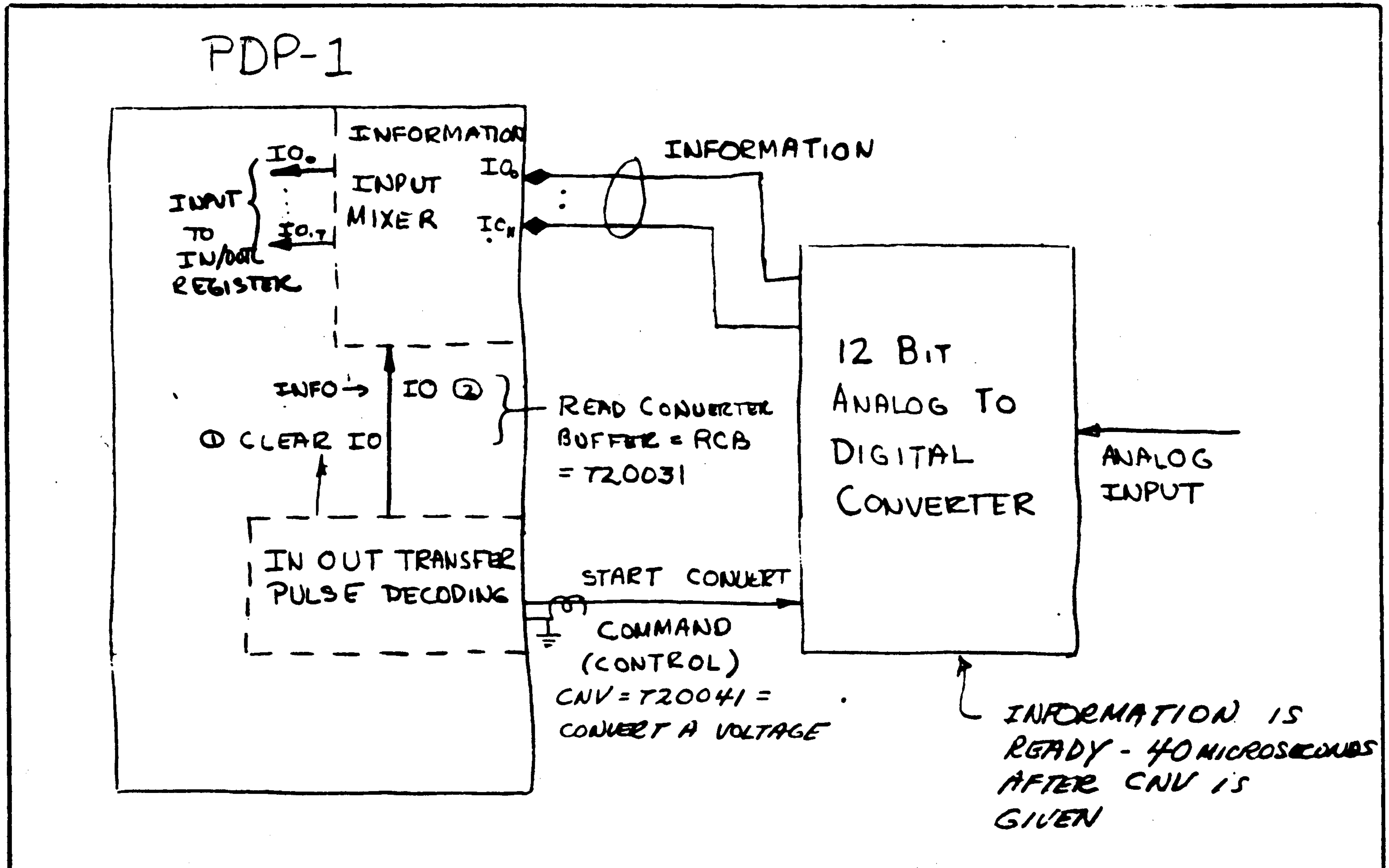
2. Sample the converter buffer outputs to set the various in-out bits (sample portion of rcb).

3. Start the conversion (CNV).

Pulse (2) occurs 2.5 microseconds after pulse (1) above.

The following subroutine uses the analog to digital converter, and is called by the command "jsp fill". The subroutine in symbolic assembly language samples an analog input 512 times at a 20 KC rate, and stores the results in registers: "function", "function & 1", ..., "function & 511".

```
,Subroutine to sample a continuous voltage 512 times, each 50
,microseconds and store in "function" table.  The subroutine starts
,at register "fill".

fill          dap exit
              law function - 1
              dap storsam          ,initialize
loop          cnv                  ,starts converter = 72 0041
              nop
              idx storsam
              sad ftest
exit          jmp                  ,exit location
              rcb                  ,rcb = 72 0031 place voltage sample
                                   ,in IO
storsam       dio                  ,store sample
              jmp loop
ftest         dio function + 1000
function      0                    ,first sample
   .          .
   .          .
   .          .
function +777 0                    ,1000₈ sample
```


PDP-1
INFORMATION
INPUT MIXER
IO₀
IO₁₇
INPUT TO IN/OUT REGISTER
IC₀
ICₙ
INFORMATION
12 BIT ANALOG TO DIGITAL CONVERTER
INFO →
IO ②
① CLEAR IO
READ CONVERTER BUFFER = RCB = 720031
ANALOG INPUT
IN OUT TRANSFER PULSE DECODING
START CONVERT
COMMAND (CONTROL)
CNV = 720041 = CONVERT A VOLTAGE
INFORMATION IS READY - 40 MICROSECONDS AFTER CNV IS GIVEN

PDP-1 / A-D CONVERTER CONNECTIONS
FIGURE 6

OPERATION OF THE STANDARD IN OUT EQUIPMENT

Synchronization and Programming

The straight forward use of the in-out transfer commands, e.g., the connection of analog to digital or digital to analog converters has been described above. In these cases, the transfer of information took on a very simple form since the program controlled the information transfers relative to the action of the device.

Quite often, input-output operations must be synchronized. That is, information is transfered certainly and efficiently, which may cause the computer (or a device) to "wait", and then proceed in synchronism. In other cases, when several in-out devices operate simultaneously, the synchronization is essential. The synchronization of the standard in-out equipment (oscilliscope, paper tape punch, photoelectric tape reader, and typewriter) is done simply and the control for this synchronization is coded in each in-out transfer command. Appendix II contains a current list of the assigned in-out transfer commands for the PDP-1 equipment.

Bacally, the program must always operate faster than a device. Thus, a program can halt, then continue in synchronism with a device. For example, the command to punch a line of paper tape is given, the paper tape punch effects the punching when able, then signals the waiting computer to proceed. Sometimes it would be desirable for the computer to give the command to punch a line of paper tape and not stop but continue calculations. This method is certain unless a second command is given before the punch has finished the previous task.

A safe method would issue a command, proceed with a safe number of calculations, then issue a waiting command which re-synchronizes the completion of the in-out transfer and the program. The re-synchronization takes place in 5 microseconds intervals.

The decoding for the in-out transfer command which allows various possibilities is shown in Table I:

TABLE I

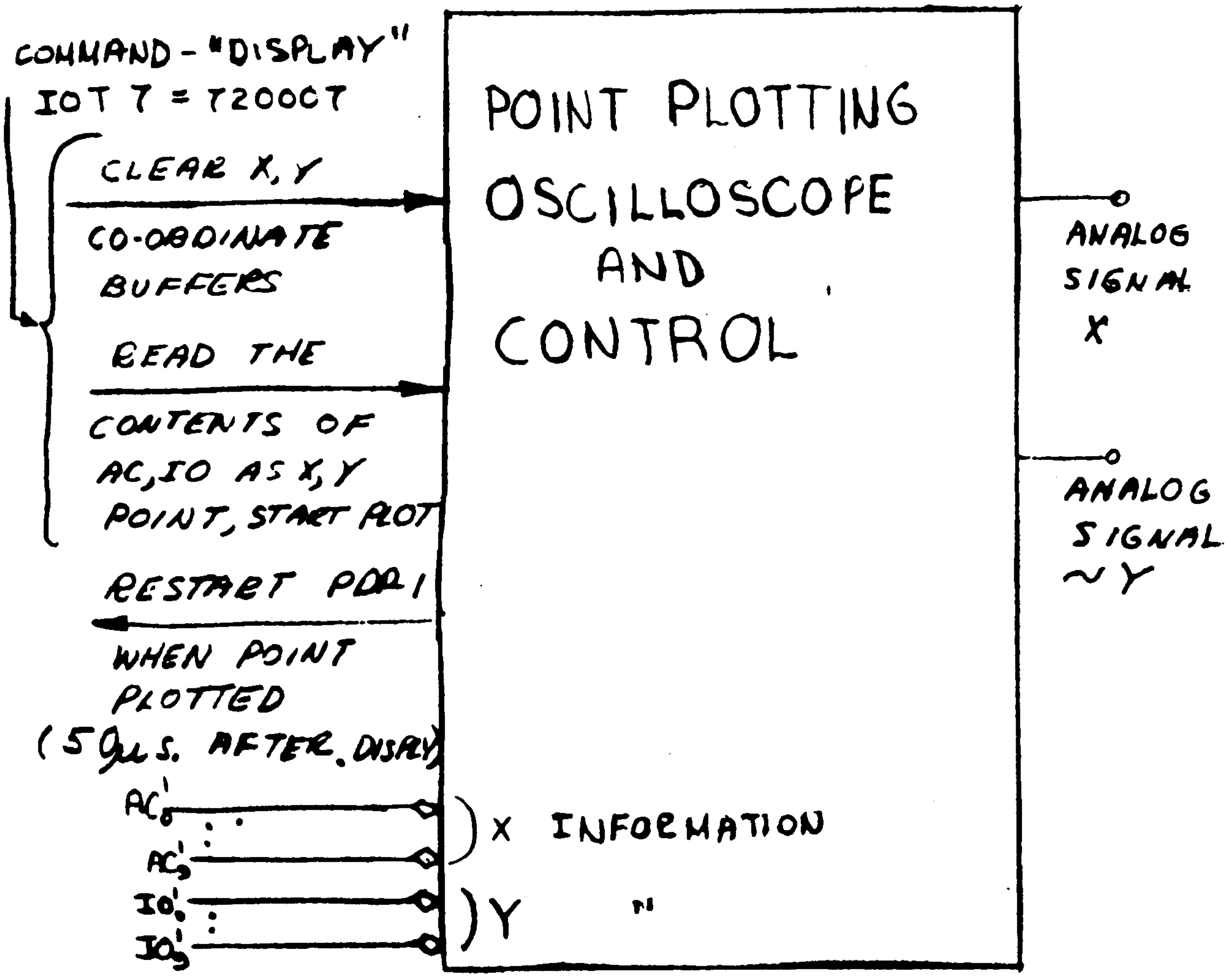
In-Out Transfer Command Bits 5	6	Wait for Completion Pulse for Restart/Continue without wait	Enable/Disable-Completion (Done Pulse Signal
0	0	Continue, no wait	Disable
0	1	Continue, no wait	Enable
1	0	Wait, then continue	Enable
1	1	Wait, then continue	Disable

Bit 5 of the in-out transfer command disignates whether the program is to wait for a completion pulse before continuing. The exclusive or of bits 5, 6 of the command specify whether the completion pulse return signal is to be enabled or disabled.

Point Plotting Oscilliscope

Figure 7 shows the diagram of the scope inter-connections to PDP-1. The oscilliscope operates on a point by point basis and is activated by the computer giving the command, "display", iot 7, or 72 0007 (or 73 0007, etc.). The display command first clears a 10 bit X coordinate buffer, a 10 bit Y coordinate buffer and 2.5 microseconds later reads the contents of the 10 most significant bits of the Accumulator and In-Out registers into the X and Y buffers and then intensifies the point. The plotting of a single point requires 50 microseconds. The cathode ray tube has a P7 phosphor, thus the point persists for a relatively long time. At the completion of the plotting, a pulse will be emitted on the restart line to PDP-1 which is used for the computer restart. The restart pulse occurs 50 microseconds after the display command is given.

The following program shows how the various in-out transfer commands are used to effect savings in timing, and handle synchronization. The first program displays a horizontal line at some Y coordinate, 1/2 the width of the scope. The line consists of 512 points, and is plotted starting at the point X = 0 going right to the point X = 377_8. The program requires (50 + 20) x 512 microseconds. This program uses the most straight forward in-out transfers. The "display" command, 73 0007, is given and the computer waits until the point is displayed before the machine restarts and continues calculations.



OSCILLOSCOPE FOR PDP-1

FIGURE 7

```
,Display a horizontal line of 512 points
,line plotted center to right edge - height of Y
start          lio y                 ,y holds height at line
               cla
loop           730007                ,display a point and wait till done
               add increment
               sma
               jmp loop              ,return until ac = 400000
               --                    ,done
               .
               .
increment      400                   ,increment of line
y              --                    ,y coordinate
```

The following program computes while the point is plotted:

```
start          lio Y
               cla
               jmp loop & 1
loop           730000                ,dummy - halts till previous completion
               724007                ,displays a point/computer proceeds
               add increment         ,display completion enabled
               sma
               jmp loop
               .
               .
               .
```

The above program plots the 512 point line in (50 + 5) x 512 microseconds. The program commands a point to be displayed, (without disabling the completion pulse), proceeds with the calculations, then finally gives a synchronizing "wait" instruction which synchronizes the display.

The only case not included above, but mentioned in Table I, is the case of bits 5 and 6 both zero. Here, the program does not halt and the completion pulse of the device is disabled. Thus, the program must only refrain from giving the display commands too frequently.

The technique of inhibiting completion is used when several devices are operating concurrently. The inhibit command completion is used with the sequence break system.

Paper Tape Punch For PDP-1

A block diagram of the paper tape punch logic is shown in figure 8. The punch action is similar to display. Two separate commands are given which transfer information to a buffer register for the paper tape and they are:

1. Punch paper alphanumeric format, (ppa) 72 0005, (or 73 0005).

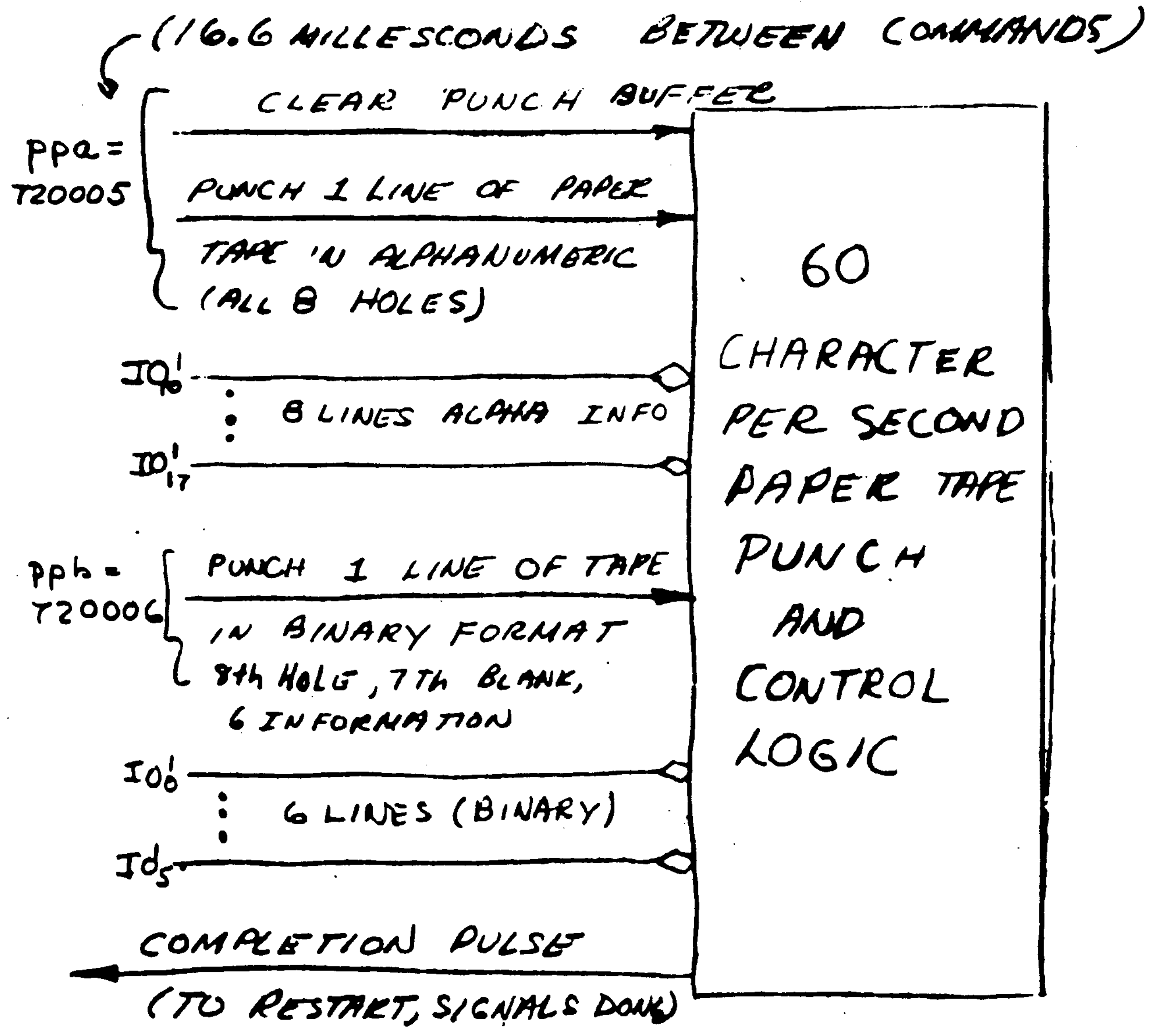
2. Punch paper binary format, (ppb) 72 0006, (or 73 0006).

When either the ppa or ppb command is given, a pulse first clears the punch flip-flop buffer register then 2.5 microseconds later a pulse will occur on either the punch paper alpha line or the punch paper binary control line and the punching action is initiated. Alphanumeric information consists of in-out register bits 10 to 17 for the 8 holes on paper tape. The ppb command always punched hole 8, ignore information for hole 7, and punched IO bits 0 - 5. A pulse occurs when the punch has finished an assigned task. This may occur within 4.0 milliseconds after the command is given.

The paper tape punch is synchronous, and only during certain intervals can a character be punched. This condition is sensed by the control logic when the punch cams reach a "start" position. If a character is to be punched, information must be static for 4.0 milliseconds after the "start" punch position and solenoid driving current are enabled. The completion pulse is returned when the punch buffer is free to receive the next character. Thus, if a punch paper command is given and it appears at the correct time, the punching appears to require only 4.0 milliseconds. If a punch command is given and the punch has just passed "start", (the synchronizing position) the completion pulse may require 4.0 + 15.8 milliseconds.

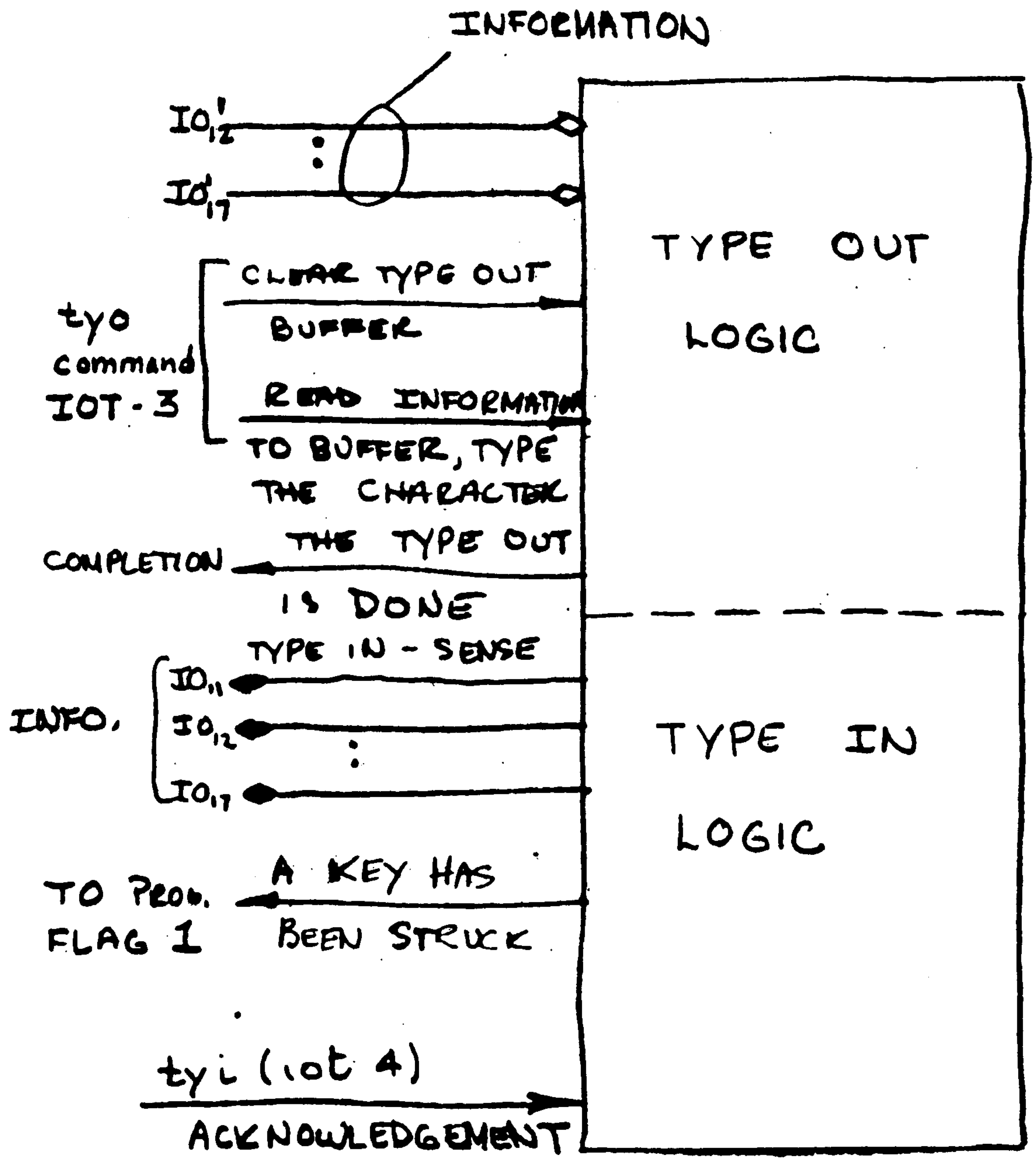
Typewriter for Output of Information

The input-output typewriter for PDP-1 is shown in figure 9. The typewriter is best explained by separating the logic into output and input. For typing out, the typewriter acts like the display or the punch. The type out command, tyo or 720003, (or 730003) first clears the typewriter buffer then 2.5 microseconds later, the information of IO bits 12-17 is read into the typewriter buffer, and the typing action initiated.


(16.6 MILLESCONDS BETWEEN COMMANDS)
ppa = 720005
CLEAR PUNCH BUFFER
PUNCH 1 LINE OF PAPER TAPE IN ALPHANUMERIC (ALL 8 HOLES)
IO'0
8 LINES ALPHA INFO
IO'17
ppb = 720006
PUNCH 1 LINE OF TAPE IN BINARY FORMAT 8th HOLE, 7Th BLANK, 6 INFORMATION
IO'0
6 LINES (BINARY)
IO5
COMPLETION PULSE (TO RESTART, SIGNALS DONE)
60 CHARACTER PER SECOND PAPER TAPE PUNCH AND CONTROL LOGIC

PAPER TAPE PUNCH FOR PDP-1

FIGURE 8



TYPEWRITER FOR PDP-1
FIGURE 9

The procedure for typing a character would be:

1. Load the in-out register bits 12-17 with the code for the character to be typed.

2. Issue the command tyo.

The typeout portion of the logic has a completion pulse which is emitted when a character has been typed. The typewriter output rate is 9.5 characters per second, thus, the typeout completion pulse occurs approximately 105 milliseconds after the tyo is given.

The typeout completed signal can be used to restart the program if the typewriter is used synchronously within a program. The completion pulse, of course, can be used in the same manner discussed with the display system. The time saving is emphasized more in the case of the typewriter because the 105 ms interval between characters will allow up to 20,000 operations.

Typewriter Input

The typewriter input logic is also shown in figure 9. This logic is similar to the sampling of an analog to digital converter. When a key is struck, a pulse is emitted on the line labelled "A key has been struck". This pulse is wired to set program flag 1. A 6 bit buffer (the same used for type out) holds the code for the character struck. When flag 1 is set, the program may give the command, type in, tyi = iot 4 (72 0004) never 73 0004. The command, tyi, first clears the IO then reads the 6 information bits, which code the character typed, and the type in sense level into IO bits 11-17. Tyi first clears the in-out register, then 2.5 microseconds later the 7 levels of the typewriter are read into the IO.

The tyi command notifies the typewriter logic that the character has been accepted by the program. The type in acknowledgement resets the type in sense line, thus, if another character is not typed and the program gives the type in instruction, bit 11 will be a 1 instead of 0. In this way, a program can tell if the same character has been read more than once.

The following program excerpt uses tyi. The program begins in register "look".

```
look          szf * 1          ,650001 skip on flag set
              jmp look         ,repeat look, look + 1 till key struck
              tyi              ,720004 IO contains code for key strucl
```

Instructions "look" and "look + 1" are repeated over and over again until the program flag 1 is set by a key being struck. When the key is struck, the program flag 1 is sensed and the instruction in look + 2 is obeyed, tyi, and the 6 bit code for the character typed is placed in the in-out register. If the key typed was a carriage return, then an octal 77 would appear in the IO. If a second tyi command is given before a new character is struck, the in-out register would contain the code 177.

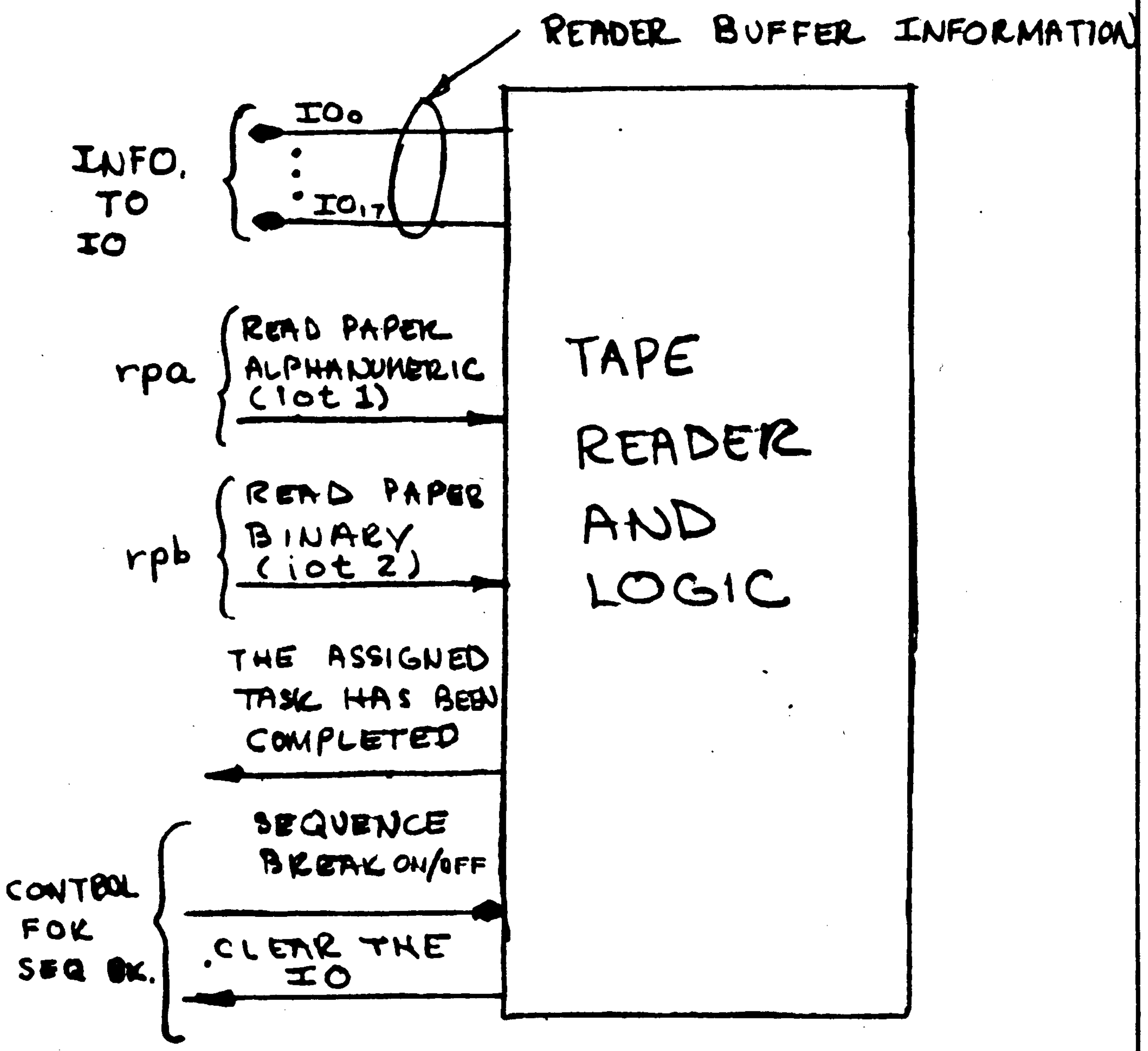
<u>The Photoelectric Tape Reader</u>

The logic for the photoelectric tape reader is shown in figure 9. Five, 6, 7, or 8 hole tapes are read at a 400 line per second rate. The paper tape reader has an 18 bit buffer which holds the information that is gathered from tape. There are two modes of operation, read paper alphanumeric (rpa) and read paper binary (rpb). When the command rap 720001 or 730001 is given, one line of tape is read. All eight holes of the line go into the right 8 bits of a buffer. When the 8 bits are assembled in the buffer, a pulse appears on the line clear the In-Out register, then 2.5 microseconds later the completion pulse reads the reader buffer information into the IO.

The command read paper binary, or rpb = 720002 or 730002, etc, reads three lines of paper tape into the reader buffer and then returns a completion pulse. For a line of tape to be recognized in this mode, the eighth hole must be punched while the seventh is ignored. The information is packed in the buffer as three 6-bit characters.

If the sequence break system is "on" the completion pulse still occurs, but the In-Out register is not cleared prior to the completion pulse. In this case, the reader buffer contents may be transfered to the IO by the command read reader buffer, rrb = iot 31. The sequence break system will be discussed below.

Computation can proceed during the 2.5 or 7.5 milliseconds an rpa or rpb is being carried out. Synchronization must be handled as previously described.


READER BUFFER INFORMATION
INFO. TO IO
IO0
IO17
rpa
READ PAPER ALPHANUMERIC (iot 1)
rpb
READ PAPER BINARY (iot 2)
TAPE READER AND LOGIC
THE ASSIGNED TASK HAS BEEN COMPLETED
SEQUENCE BREAK ON/OFF
CONTROL FOR SEQ BK.
CLEAR THE IO

400 LINE PER SECOND TAPE READER

FIGURE 10

THE GENERAL CONNECTION OF PROGRAMMED IN-OUT TRANSFERS

Outgoing Information

Lines are available from the In-Out register output for transfer of information to a device. The individual lines of each bit of a register form a bus to which connections may be made. Appendix I describes the loading restrictions of these lines.

The In-Out register may connect with output devices. The scheme for affecting this connection to the output bus is shown in figure 11. Here, each bit of the In-Out register, i.e., the one's side output is available at a taper pin connector block, being suitably buffered with a bus driver. This taper pin block allows connections to be made in parallel to devices which desire information.

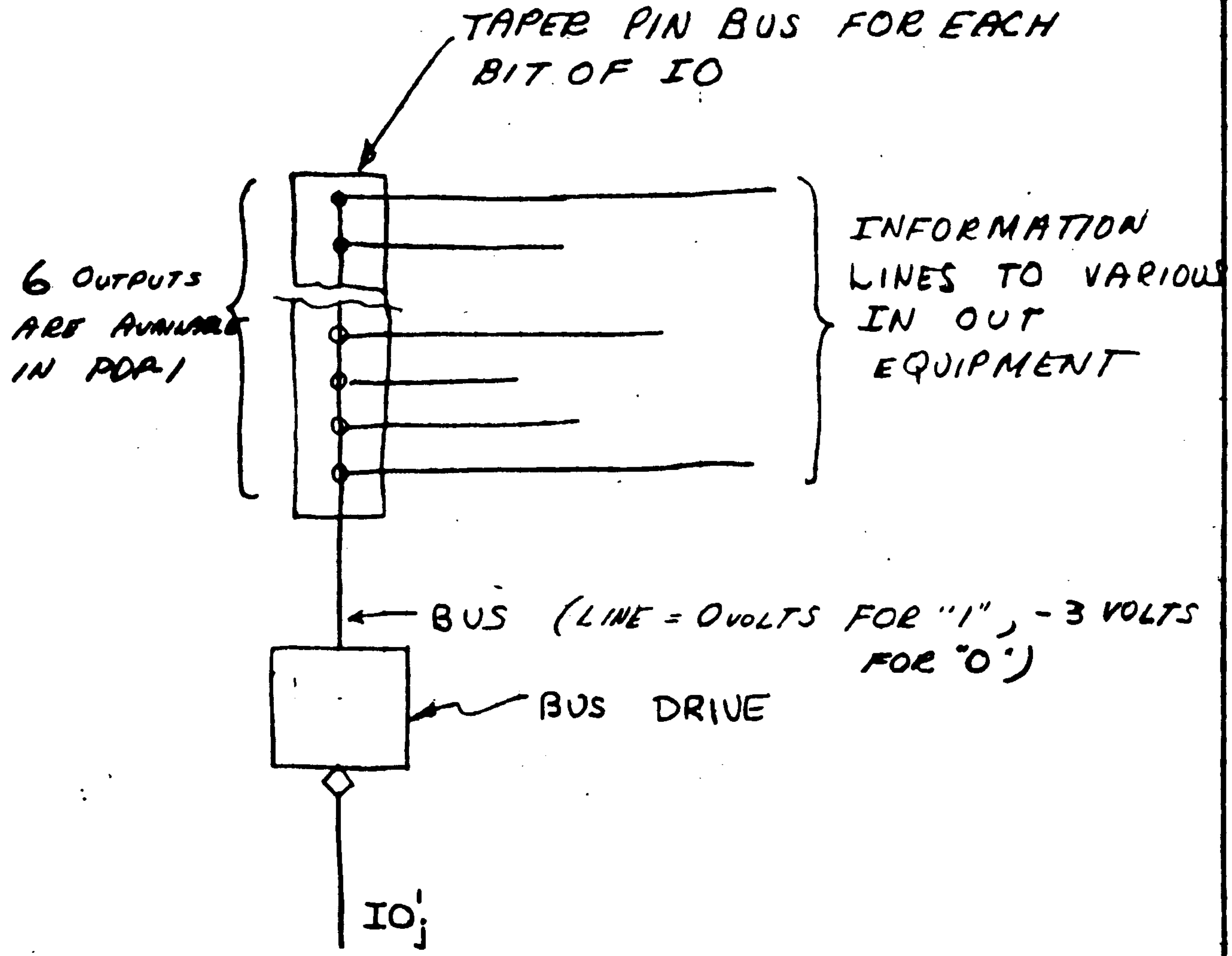
Incoming Information

A similar taper pin block arrangement is available for connecting the outputs of a device to the inputs of the In-Out register mixer. The In-Out Mixer reads in the various information bits to the In-Out register. The input levels "and" with program in out transfer pulses to form a pulse which is mixed with similar pulses through an "OR" circuit. One bit of the In-Out register Input Mixer is shown in figure 5. Eight taper pin inputs exist for several devices. The specifications for the input mixer are summarized in Appendix I.

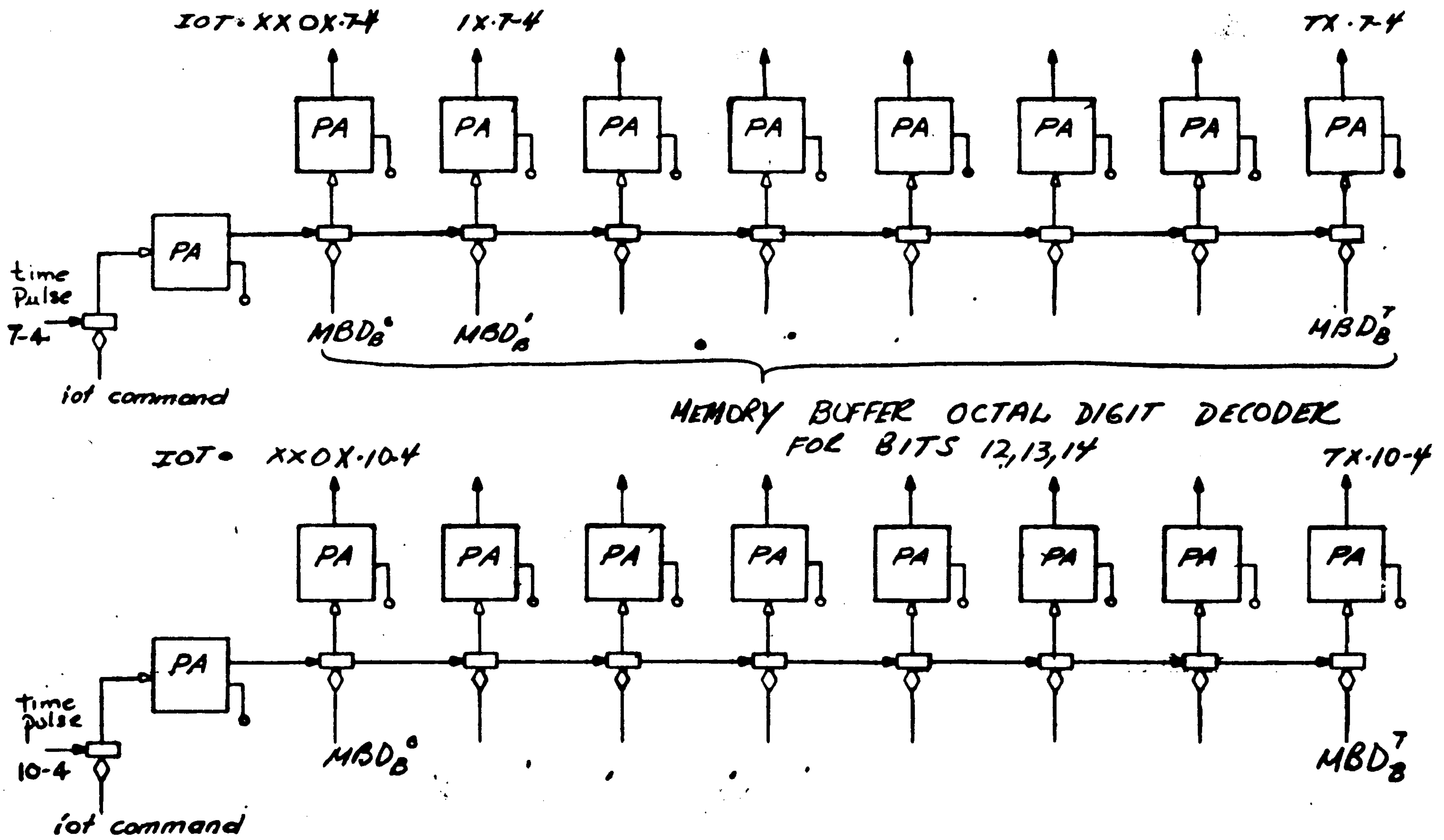
Formation of Programmed In-Out Transfer Pulses

A portion of the In-Out Transfer Control block schematic is shown in figure 12. This diagram shows the formation of the standard In-Out transfer pulses, (or in some cases pairs of pulses). These include rpa, rpb, tyo, tyi, etc., i.e., all those pulses discussed above which communicate with the conventional equipment.

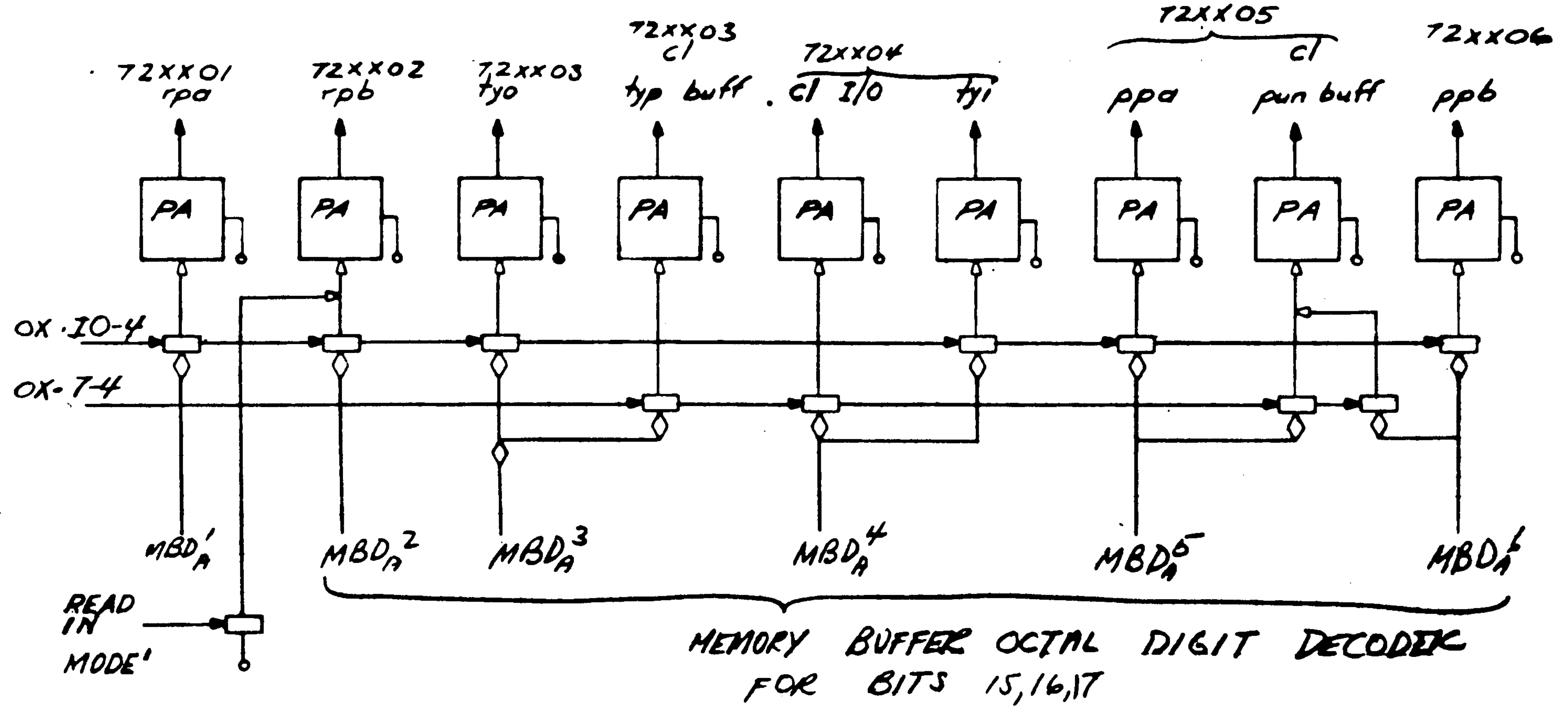
Appendix I describes the Memory Buffer Lines which constitute the basic decoding for the in-out transfer pulses. In general, the levels are not needed externally. The lines, MBD with subscripts, A, B, C, and D, and superscripts 0 - 7, refer to the decoding of the Memory Buffer octal digits, MBD, or Memory Buffer Decoders. Decoder A is connected to bits 15 - 17, decoder B to 12 - 14, etc. The superscript number refers to the decoder output lines.


TAPER PIN BUS FOR EACH BIT OF IO
6 OUTPUTS ARE AVAILABLE IN PDP-1
INFORMATION LINES TO VARIOUS IN OUT EQUIPMENT
BUS (LINE = 0 VOLTS FOR "1", -3 VOLTS FOR "0")
BUS DRIVE
IO'j

IN OUT REGISTER
OUTPUT BUS
FIGURE 11



FIRST STAGE IOT ADDRESSING
FIGURE 12a



FINAL STAGE IOT ADDRESSING

FIGURE 12b

Two pulse lines are used to form basic in-out transfer pulses. They are called tp7-4 and tp10-4. These line pulses are fixed relative to the computer's memory cycle time, with tp7-4 and tp10-4 occuring 2.5 and 5.0 microseconds after the beginning of a memory cycle. The -4 indicates that the pulse is a DEC 0.4 microsecond pulse. The iot instruction level is shown and is a one when an In-Out transfer command is given.

The pulse formed by "anding" tp7-4 (or tp10-4) with the In-Out transfer command is again "anded" with MBD_B of figure 12a to form 8 basic pairs of pulses.

The 8 by 8 decoding scheme allows up to 64 addressable pulses (or pairs) to be easily formed. Any or all of these pulses may be further subdivided by using MBD_C or MBD_D. Thus, a particular iot may be connected to an additional 8 by 8 array to give size to 64 sub-orders. If the physical space limitations are somehow circumvented, 4096 distinct in-out instructions can be implemented. A list of the in-out transfer commands is given in Appendix II.

The iot control pulse amplifier labeled "read paper alpha" emits a pulse when Memory Buffer Decoder A is a 1, ($MBD1_A$, the 3 least significant bits in the In-Out transfer instruction are 001), and when $MBD0_B$ (or the 3 next least significant bits are 000) and when an iot command is given.

Synchronization of Device Completion and Computer Restarting

The computer must synchronize with some in-out devices. This synchronization is determined by the use of bits 5 and 6, in the iot command (see page 8). The implementation of this function is shown in figures 13a and b. A flip-flop is used as a switch for each device which is to operate in this mode. Thus, the various return pulses (completion pulses) may or may not affect the restart action.

Restart synchronization need not use the flip-flop switch arrangement. In this case, a restart pulse can go directly to the in-out transfer done, pulse amplifier to restart the machine.

The action of bit 5 to indicate a wait words on any iot instruction. The enable/disable feature of the completion pulse is only applicable to devices having the extra logic of figure 13a. The standard devices (reader, punch, and typewriter output) have this feature. The oscilliscope display has this feature.

If the Sequence Break System is included, the device completion pulse may also go to an appropriate SBS channel.

Certain iot commands will never use the wait feature and thus the completion enable/disable need not be included. In such a case, bit 6 of the command may be used in the same manner as bits 7-11.

Special Levels and Pulses

Appendix I describes several connections which are useful for input-output equipment.

Accumulator Outputs: The Accumulator bits 0-11 outputs are available. The driving power of the AC is limited and loadings should be carefully observed.

Memory Buffer and Octal Memory Buffer Decoders: The various Memory Buffer Octal digit levels and Memory Buffer levels are available. When iot commands are given, these may be used either for information or additional decoding.

Program Flags: An external device can set a program flag. Either of two inputs may be used. The output of the program flag is available for control purposes.

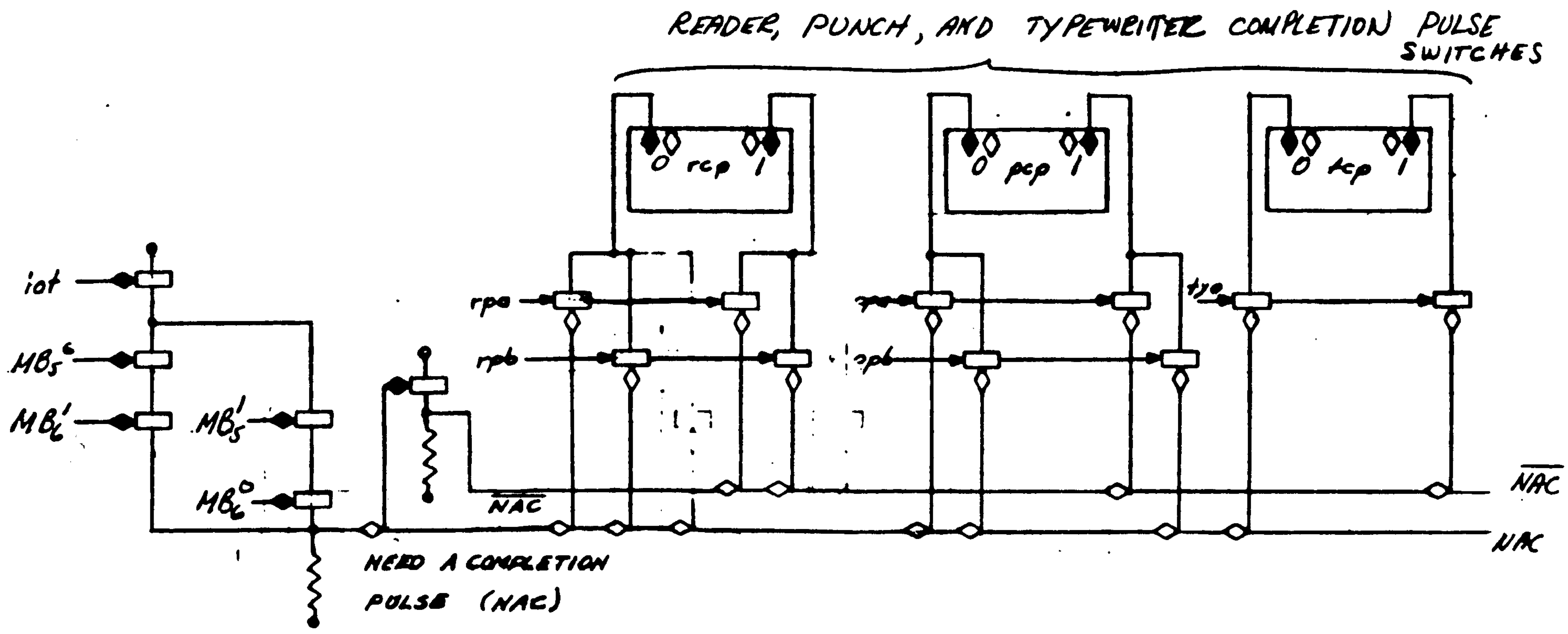
Program Counter: A pulse can add one to the program counter. In this way, an instruction could be skipped if an in-out condition is not met.

Synchronization Pulses: Timing pulses are available which may be used to synchronize extra equipment. The time pulses occur at 1,2.5 and 5.0 microseconds after the start of a memory cycle. The start and stop pulses are available and occur when the console start or stop keys are pressed.

BASIC SEQUENCE BREAK SYSTEM

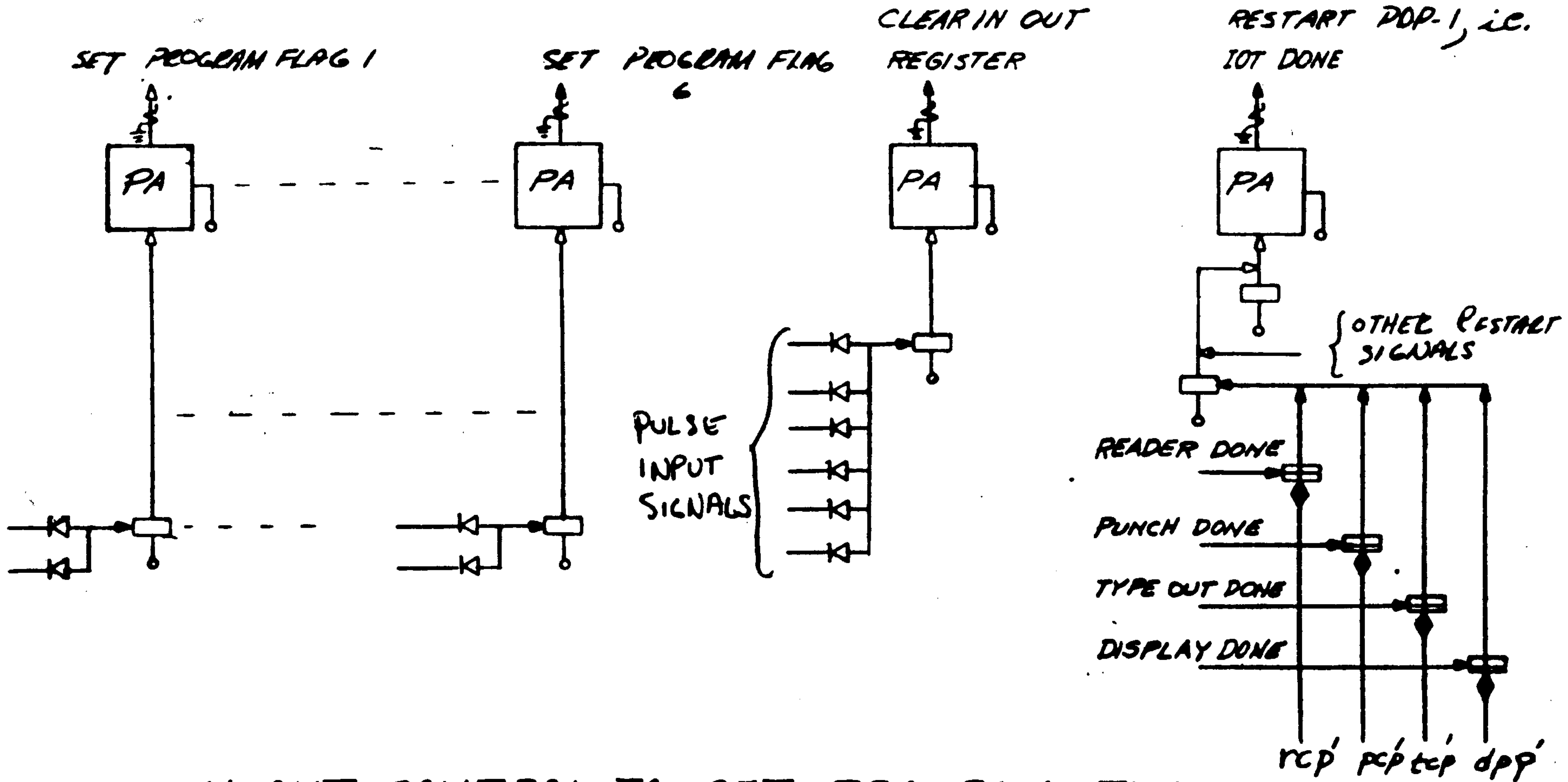
Programmed In-Out Synchronization

When a device communicates in a random fashion with PDP-1, a


READER, PUNCH, AND TYPEWRITER COMPLETION PULSE SWITCHES
0 rcp 1
0 pcp 1
0 tcp 1
iot
MB_5^6
MB_6^1
MB_5^1
MB_6^0
rpa
rpb
tyo
$\overline{NAC}$
NAC
NEED A COMPLETION PULSE (NAC)

CONTROL SWITCHES FOR COMPLETION PULSES

FIGURE 13a



IN OUT CONTROL TO SET PROGRAM FLAGS, CLEAR IN OUT REGISTER & RESTART COMPUTER

FIGURE 13b

means is required to sample the device's readiness, then to synchronize information transfers. The program flags may be set by external devices and sensed by program. If the machine must always "wait" for device completion (or action), before continuing a sequence, periodic sampling of the program flags, or any other external states (e.g., flip-flops) must be affected.

On page 12, a program loop was given which continually checked the typewriter input flag (program flag 1). When a character has been typed, a program sequence can continue, handling the character. The limitations of this technique are:

1. Operating time is consumed sensing and waiting for flag.

2. Priorities among devices must be established by the program. This often requires careful considerations.

3. A program loop (loop contains one sense flag instruction) can be no longer than the repetition period of a device.

Automatic Program Interruption

The Sequence Break System (SBS) enables a program sequence to be interrupted, i.e., the sequence of instructions broken, each time a device needs attention. Thus, a program loop need not be used to sense a condition, but rather calculations may proceed, and are only interrupted when communications need exist (e.g., data transfers) between the machine and a device.

The basic Sequence Break System has provisions for 12-level inputs. Thus, when any of the 12 levels are present, (a one) an interruption occurs. This program interruption takes the following form:

1. The contents of the AC are stored in register 0.

2. The contents of the program counter (and memory field information) are stored in register 1.

3. The contents of IO are stored in register 2.

4. The program resumes in register 3.

The above steps require 3 x 5 microseconds. Since one of 12 devices may cause the interruption, a program must find the device. The command, check status bits, cks, 72 0032, causes a set of status bits to be read into the IO. By examining the various IO bits, the appropriate device may be found. While status bits 0-4 are taken with conventional in out equipment, the remaining bits may be free for use.

When an interruption occurs, the action of other devices connected to other SBS inputs are ignored, until the break is terminated properly. This is accomplished by restoring the C(AC), C(IO) and returning to the previous sequence. This termination may be affected by the following program:

```
lac 0
lio 2
jmp * 1
```

The last instruction, 61 0001, is the one which actually affects the termination, and must be given exactly as shown above. Two additional instructions turn on or turn off the sequence break mode.

1. Enter sequence break mode, esm, iot 55.

2. Leave sequence break mode, lsm, iot 54.

When the sequence break mode is off, no interruptions may occur.

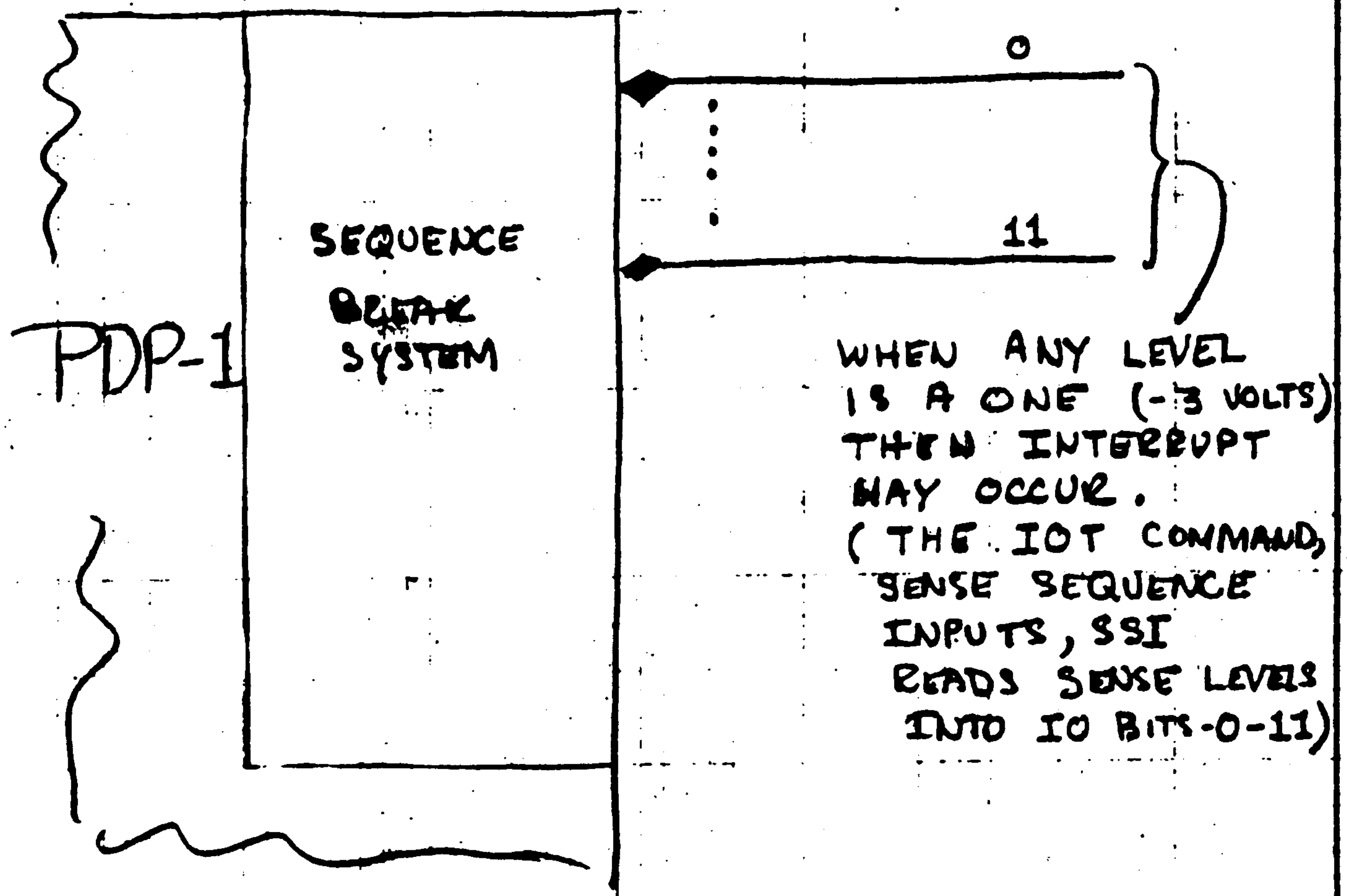
The sequence break mode can be turned on or off by the console "start" switch, either by being pushed up for on or down for off. A block diagram of the sequence break inputs is shown in figure 14.

The program flags may be wired to act as inputs to the SBS System.

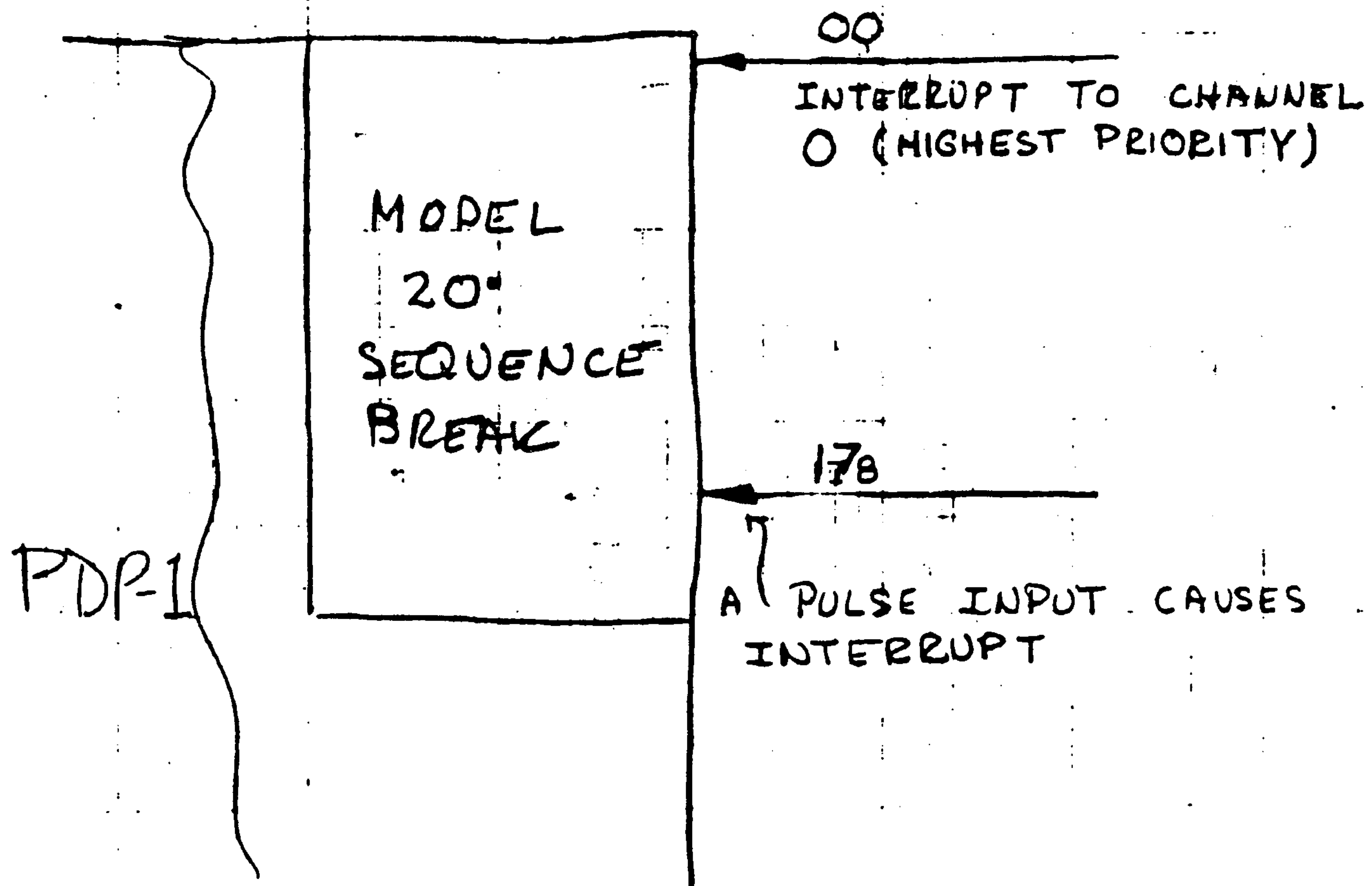
TYPE 20 - 16 CHANNEL SEQUENCE BREAK SYSTEM

This system allows 16 devices to operate simultaneously, and in a preassigned priority. The 16 separate channels allow a device to interrupt to 1 of 16 unique locations (instead of one for the basic break system). Thus, 16 x 4 memory locations are assigned to this sequence break. The channels are arranged in a priority chain.

The block diagram for the system is given in figure 15. In


0
11
PDP-1
SEQUENCE
~~BREAK~~
SYSTEM
WHEN ANY LEVEL IS A ONE (-3 VOLTS) THEN INTERRUPT MAY OCCUR.
(THE IOT COMMAND, SENSE SEQUENCE INPUTS, SSI READS SENSE LEVELS INTO IO BITS-0-11)

CONVENTIONAL SEQUENCE BREAK SYSTEM
FIGURE 14


00
INTERRUPT TO CHANNEL 0 (HIGHEST PRIORITY)
MODEL 20 SEQUENCE BREAK
17₈
A PULSE INPUT CAUSES INTERRUPT
PDP-1

MODEL 20 - 16 CHANNEL SEQUENCE BREAK SYSTEM

FIGURE 15

this case, a pulse or a negative transition can cause a break to a sequence. Program commands may cause these pulses to be ignored, i.e., a channel may be "on" or "off". Since the 16 channels are in a priority chain, a higher priority break may interrupt the sequence of a lower one. The following action occurs if a pulse enters the ith channel input:

1. C(AC) = C(4i)
2. C(PC) = C(4i + 1)
3. C(IO) = C(4i + 2)
4. Program resumes in 4i +3 by a jmp command.
   where i = 0, 1, ... 17_8

The break is terminated by the following:

```
lac 4i
lio 4i + 2
jmp * 4i + 1
```

The termination is affected by the last instruction.

<u>Flip-Flops of Type 20 Sequence Break</u>

There are 16 x 4 flip-flops which have significance and are visible to the user. The flip-flops are:

1. Channel ON
2. Break Synchronize
3. Waiting Break
4. Break Started

Channel On is used as a switch and must be in the "one" state to allow device completion pulses to set Break Synchronize.

Waiting Break and Break Synchronize act together. Waiting Break is set to a one when Break Synchronize is a one. Two microseconds after this event, Break Synchronize is cleared.

Waiting Break provides a signal that a particular channel would like an interruption. This interruption may occur provided that a break has not started (Break Started = 1) for the same or a higher channel and no higher channel is currently Waiting Break.

When a break occurs to a channel, Break Started is set to a one. At this time, Waiting Break is cleared. Higher priority channels may interrupt lower channels. Break Started remains a one until the break is dismissed by giving a jmp * instruction as previously discussed.

The instructions for Model 20 Sequence Break Systems are:

1. esm, Enter Sequence Break System mode, iot 55, (see basic sequence break).

2. lsm, Leave Sequence Break mode, iot 54. (See basic sequence break).

3. isb, Initiate Sequence Break: Channel i is selected by bits 8-11 of the command, iot 52. This command allows the program to initiate a break, or simulate an external pulse action. This break is not conditioned by "channel on".

4. asc, Activate Sequence Break: Channel i is selected by bits 8-11 of the command iot 51. Allows channel i to be active, i.e., pulses to effect breaks).

5. dsc, Deactivate Sequence Break, iot 50. Channel i is selected by bits 8-11 of the command, iot 50. A channel is turned off thus any incoming break pulses will be ignored.

HIGH SPEED CHANNELS FOR DIRECT MEMORY DATA TRANSFERS CONCURRENT WITH COMPUTATIONS

A high speed channels feature is an option which may be used to transfer blocks of words between memory and an in-out device, e.g., a device such as magnetic tape, which is time consuming to program on a character by character (or word by word) basis.

This optional feature is installed with tape control, Type 52 or may be installed separately for special applications. When the feature is added, the internal machine facilities for three channels are available. The three channels are serviced on a demand basis.

A common information transfer requirement is that successive words go to or from the computer at high speed. The use of program flags, the sequence break, or a program loop of in-out transfers, all effect the speed and efficiency of the transfers. In most cases, the number of program steps required to transfer each word is small with respect to the number of program steps over the whole period between transfers. For high speed devices, such as magnetic tape, the reduction in available time between transfers makes it desirable to reduce the transfer sequence time and improve the ratio of available computation time to total running time.

Each of the three channels are interrogated regularly at the completion of each memory cycle, on a priority basis. The priority is wired and fixed. The sequence break system has a priority just below the last high speed channel. If a channel has a word for memory, or requires a word from memory, a memory cycle interruption occurs. The diagram of one high speed channel (of 3) is shown in figure 16.

The lines operate as follows:

1. The memory address and field lines select the memory register for the device.

2. Information

   a. Memory buffer outputs contain the information which is to be transmitted to a device.

   b. The high speed channel input mixer is similar to the in-out register input mixer and is used for incoming information.

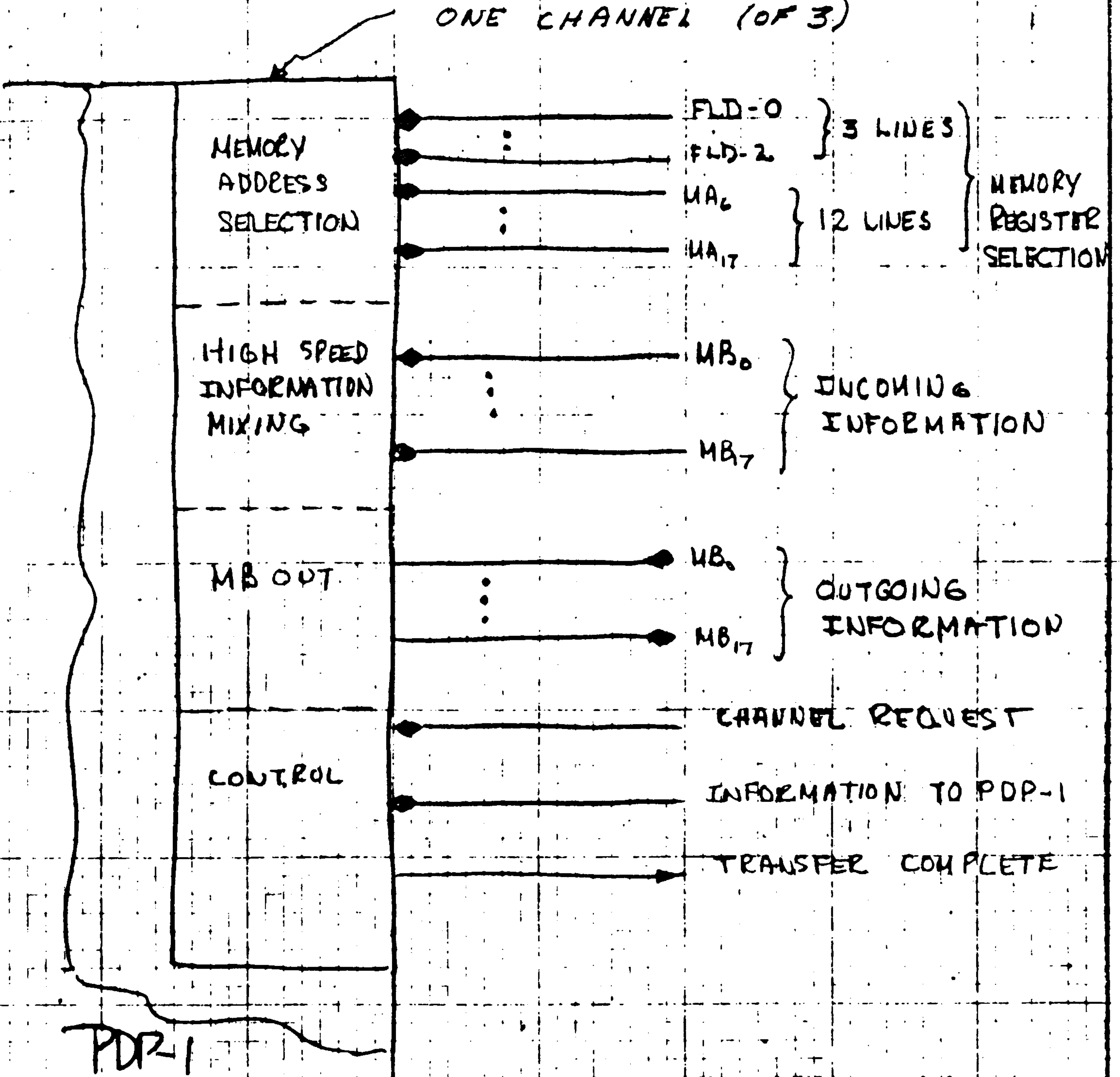
3. Control

   a. Request - When the line is a one, a memory interruption may take place.

   b. Transfer Direction Lines - This line specifies whether information is to go to or from PDP-1.

   c. Transfer Done - A pulse occurs on this line to acknowledge the request completion. The request line must be removed to prevent another interruption.

The timing diagram for a high speed channel is shown in figure 17.



HIGH SPEED CHANNEL CONNECTIONS
(1 CHANNEL)
FIGURE 16

APPENDIX I

LIST OF IN OUT TRANSFER COMMANDS

Instruction	Code	Definition
Photoelectric Punched Tape Reader		
rpa	iot** X001	Read punched tape, alphanumeric forward direction
rpr	iot X101	Read punched tape, reverse direction
rpb	iot XX02	Read punched tape, bi-octal
rrb	iot XX30	Read reader buffer (for sequence break operations)
cks	iot XX33	Check status bits. Miscellaneous control bits are read into IO. Bit 0 - Displayed point seen by light pen 1 - Paper tape reader busy 2 - Typewriter busy 3 - Typewriter key struck 4 - Punch busy
Tape Punch		
ppa	iot XX05	Punch punched tape, alphanumeric
ppb	iot XX06	Punch punched tape, bi-octal
Typewriter		
tyo	iot XX03	Type out
tyi	iot XX04	Type in

*Symbolic assembly mnemonic code.
**iot has the value 720000 in symbolic assembly language.
X denotes octal digit may be anything.

Instruction	Code	Definition
Display - Type 30		
dpy	iot XX07	Display one point on CRT
(The above commands, excepting rpr, are standard for all PDP-1's)		
Card Reader - Type 41-523		
rac	iot XX41	Read a card
rsc	iot XX42	Row synchronize and clear field counter
raf	iot XX32	Read a field of 18 columns and index field counter
Card Punch - Type 40-523		
pac	iot XX43	Punch a card
psc	iot XX44	Row synchronize and prepare to punch first field
pag	iot XX22	Punch a field of 18 columns and index field counter
Basic Magnetic Tape - Type 51		
mcb	iot XX70	Magnetic tape clear buffer
mwc	iot XX71	Magnetic tape write character
mrc	iot XX72	Magnetic tape read character
mcs	iot XX34	Magnetic tape check status
msm	iot XX73	Magnetic tape select mode
High Speed Magnetic Tape - Type 52		
muf	iot kk75	Magnetic tape unit and final address, kk specifies control information

Instruction	Code	Definition
mic	iot kk76	Magnetic tape initial address and command
mel	iot kk35	Magnetic tape examine location
mes	iot kk36	Magnetic tape examine status
mri	iot kk66	Magnetic tape reset initial address
mrf	iot kk67	Magnetic tape reset final address
Clock		
rsk	iot XX47	Reset the clock
rdk	iot XX37	Read clock time into IO
Timer		
stm	iot XX24	Set timer with IO
Relay Buffer		
srb	iot XX21	Set relay buffer
Analog to Digital Converter		
cnv	iot XX41	Convert a voltage
rcb	iot XX31	Read converter buffer
Sequence Break System - Type 20		
esm	iot XX55	Enter sequence break mode
lsm	iot XX54	Leave sequence break mode
asc	iot NN51	Activate sequence break channel NN

Instruction	Code	Definition
dsc	iot NN50	Deactivate sequence break channel NN
isb	iot NN52	Initiate sequence break to channel NN
Core Memory Expansion		
cfd	iot kk74	Change fields, Type 11 expansion kk specifies new field
cdf	iot Xk74	Change data field, Type 14 expansion k specifies new data field
Line Printer - Type 62		
lgo	iot X145	Line printer go (print)
lfb	iot X245	Fill line printer buffer
lsp	iot X345	Space (vertically) line printer

APPENDIX II - LIST OF INTERCONNECTIONS WITH PDP-1

LINE	IN-OUT	NUMBER	POLARITY	DEC CIRCUIT AT PDP-1	SUGGESTED LOAD	REMARKS
IO_{0-17}	out	7	—◆ = 1	1685	100 Type 4128 etc. cap. diode gates - Less than 1000 ft. cable	General programmed output transfers
AC_0	out	1	—◆ = 1	4113	cap. diode gates	For oscilloscope, but available
AC_{1-11}	out	1	—◆ = 1	4113	cap. diode gates	For oscilloscope, but available
MB_{0-17}	out	3	—◆ = 1	1685	cap. diode gates	For In Out and High Speed Channel Transfers
$MBD_A \cdot$ iot	out	8	→	4603		Used only for forming in out transfers (2 sets available, 2.5 and 5.0 microseconds after start of cycle).
MBD_B	out	8	—◆ = 1	4113		Used only for forming in out transfers
$MBD_{C,D}$	out	8	—◆ = 1	4113	cap. diode gates	For iot commands, sequence break
program flag	out	1 x 6	—◆ = 1	1209		For external level output
IO_{0-17}	In	1	—◆ = 1	4129		For incoming information, levels must be present 2.0 microseconds (4 wired and available with additional 9 - 4129)
MB_{0-17}	In	3	—◆ = 1	4129		For incoming information of High Speed Channels
Memory fields MA_{6-17}	In	3	—◆ = 1	4129		To select register for High Speed channels
4o IO	In	6	→	4110		Clear IO
Restart Computer	In	6	→	4110		Restarts computer when in out halting
program flags	In	2 x 6	→	4112		Any device may set flag

APPENDIX II - LIST OF INTERCONNECTIONS WITH PDP-1
Page 2

LINE	IN-OUT	NUMBER	POLARITY	DEC CIRCUIT AT PDP-1	SUGGESTED LOAD	REMARKS
↳+ 1 PC	In	6	→	4110		Advance program counter
Basic Sequence Break	In	12	—◆ = 1	4110		Any level causes Sequence Break
Time Pul. 2.4	Out	1	→	4603		Pulses must be buffered by 4603 before driving separate lines occurs 1 microsecond after start of Memory Cycle
7.4	Out	1	→	4603		occurs 2.5 microseconds after start of Memory Cycle
10.4	Out	1	→	4603		occurs 5.0 microseconds after start of Memory Cycle
Power on	Out	1	→	4603		occurs when power comes on
stop	Out	1	→	4603		occurs when console stop button is pushed
start	Out	1	→	4603		occurs when console start button is pushed
Sequence Break Type 20	In	16	→	4128		Pulse causes Sequence Break, Type 20 only
High Sp. Channel requests	In	1 x 3	—◆ = 1	4106		Requests a channel transfer
High Sp. Channels In/Out	In	1 x 3	—◆ = 1	4106		Direction of a channel transfer
High Sp. Channels Complete	Out	1 x 3	→	4603		Specifies completion of transfer

When a break occurs to a channel, Break Started is set to a one. At this time, Waiting Break is cleared. Higher priority channels may interrupt lower channels. Break Started remains a one until the break is dismissed by giving a jmp * instruction as previously discussed.

The instructions for Model 20 Sequence Break Systems are:

1. esm, Enter Sequence Break System mode, iot 55, (see basic sequence break).

2. lsm, Leave Sequence Break mode, iot 54. (See basic sequence break).

3. isb, Initiate Sequence Break: Channel i is selected by bits 8-11 of the command, iot 52. This command allows the program to initiate a break, or simulate an external pulse action. This break is not conditioned by "channel on".

4. asc, Activate Sequence Break: Channel i is selected by bits 8-11 of the command iot 51. Allows channel i to be active, i.e., pulses to effect breaks).

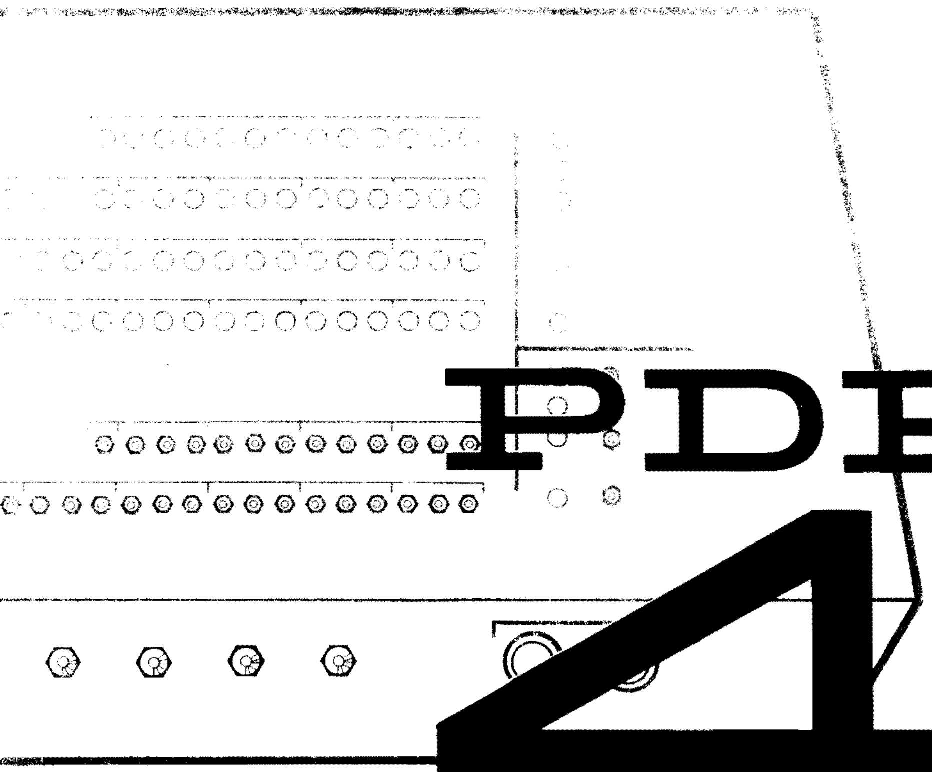
5. dsc, Deactivate Sequence Break, iot 50. Channel i is selected by bits 8-11 of the command, iot 50. A channel is turned off thus any incoming break pulses will be ignored.

HIGH SPEED CHANNELS FOR DIRECT MEMORY DATA TRANSFERS CONCURRENT WITH COMPUTATIONS

A high speed channels feature is an option which may be used to transfer blocks of words between memory and an in-out device, e.g., a device such as magnetic tape, which is time consuming to program on a character by character (or word by word) basis.

This optional feature is installed with tape control, Type 52 or may be installed separately for special applications. When the feature is added, the internal machine facilities for three channels are available. The three channels are serviced on a demand basis.

A common information transfer requirement is that successive words go to or from the computer at high speed. The use of program flags, the sequence break, or a program loop of in-out transfers, all effect the speed and efficiency of the transfers. In most cases,



PDP 4 MANUAL

PROGRAMMED DATA PROCESSOR-4 MANUAL

DIGITAL EQUIPMENT CORPORATION • MAYNARD, MASSACHUSETTS

Foreword

This manual is for programmers and users of the Programmed Data Processor-4, a high speed, stored program, digital computer manufactured by the Digital Equipment Corporation. Chapters 2 and 3 contain the detailed information necessary to make use of the machine. Chapter 1 summarizes the machine's electrical and logical design. Chapter 4 presents information helpful in making the electrical connections to input-output devices. Appendices provide detailed data which may be helpful in specific programming assignments. Although program examples are given in this document, no attempt has been made to teach programming techniques. However, Appendix 4 explains the meaning and use of special characters used in the programming examples.

Table Of Contents

Typical PDP-4 System

CHAPTER 1

SYSTEM DESCRIPTION

Summary

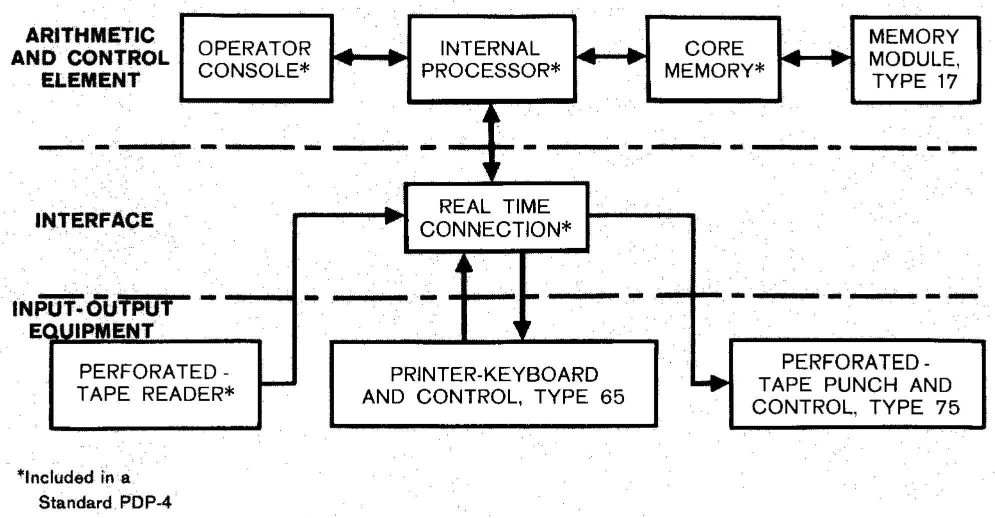
The Digital Equipment Corporation Programmed Data Processor-4 (PDP-4) is designed to be the control element in an information processing system. PDP-4 is a single address, parallel, binary machine with an 18-bit word length using 1's or 2's complement arithmetic. Standard features of the machine are stored program operation, a random access magnetic-core memory, a complete order code, and indirect addressing.

Flexible, high-capacity input-output capabilities of the PDP-4 enable it to operate in conjunction with a variety of peripheral devices, such as perforated-tape readers and punches, punched-card readers and punches, Teletype printer-keyboard, line printers, magnetic tape transports, and analog-to-digital converters.

The machine is completely self-contained, requiring no special power sources, air conditioning, or floor bracing. From a single source of 115-volt, 60-cycle, single-phase power, PDP-4 produces circuit operating dc voltages of -15 volts (±1) and $+10$ volts (±1) which are varied for marginal checking. Total power consumption is 900 watts. It is constructed with standard DEC 4000 series system modules and power supplies. Solid-state components and built-in marginal checking facilities insure reliable machine operation.

System Description

The basic PDP-4 system is shown diagramatically in Figure 1. Three portions of the system are delineated according to function: the Arithmetic and Control Element, the Interface, and the Input-Output Equipment. Information originates not only from peripheral devices but can be entered manually and modified at the Operator Console.



ARITHMETIC AND CONTROL ELEMENT
OPERATOR CONSOLE*
INTERNAL PROCESSOR*
CORE MEMORY*
MEMORY MODULE, TYPE 17
INTERFACE
REAL TIME CONNECTION*
INPUT-OUTPUT EQUIPMENT
PERFORATED - TAPE READER*
PRINTER-KEYBOARD AND CONTROL, TYPE 65
PERFORATED - TAPE PUNCH AND CONTROL, TYPE 75
*Included in a Standard PDP-4

Figure 1 — PDP-4 System with Real-Time Connection

ARITHMETIC AND CONTROL ELEMENT

The Operator Console, Internal Processor, and Core Memory constitute the Arithmetic and Control Element. The Internal Processor carries out the arithmetic and logical operations and controls the Real-Time Connection and the Core Memory. Binary arithmetic with a fixed point is employed. The optional Extended Arithmetic Control Unit, Type 22, gives PDP-4 a multiply, divide, and arithmetic shifting capability without the use of subroutines.

The Console is used to observe and control the action of the program and the Internal Processor, and to alter the contents of Internal Processor registers. The contents of Core Memory can be examined or new information deposited. All Internal Processor registers are displayed continuously.

Memory capacities of from 1,024 to 32,768 words are available for PDP-4. The cycle time (the time required to read information from memory and rewrite information back into memory) is 8 microseconds. The access time (the time required to read information from memory) is 2 microseconds. In the event of power failure, the contents of the Core Memory remain unaltered. See Chapter 2 for detailed functions of the Arithmetic and Control Element.

INTERFACE

The Real-Time Connection, furnished as standard equipment, provides communication between the Internal Processor and the Perforated-Tape Reader, the Perforated-Tape Punch and Control, Type 75, and the Printer-

Keyboard and Control, Type 65. The Real-Time Option, Type 25 gives the system the additional capability to operate efficiently over a wide range of information handling rates (from seconds per event to 125,000 words per second) and with a large variety of input-output devices (see Figure 2). The Real-Time Option consists of a Device Selector, an Information Collector, an Information Distributor, an Input-Output Skip connection, a Program Interrupt facility, a Data Interrupt facility, and a Clock/Timer. See Chapter 3 for details of functions.

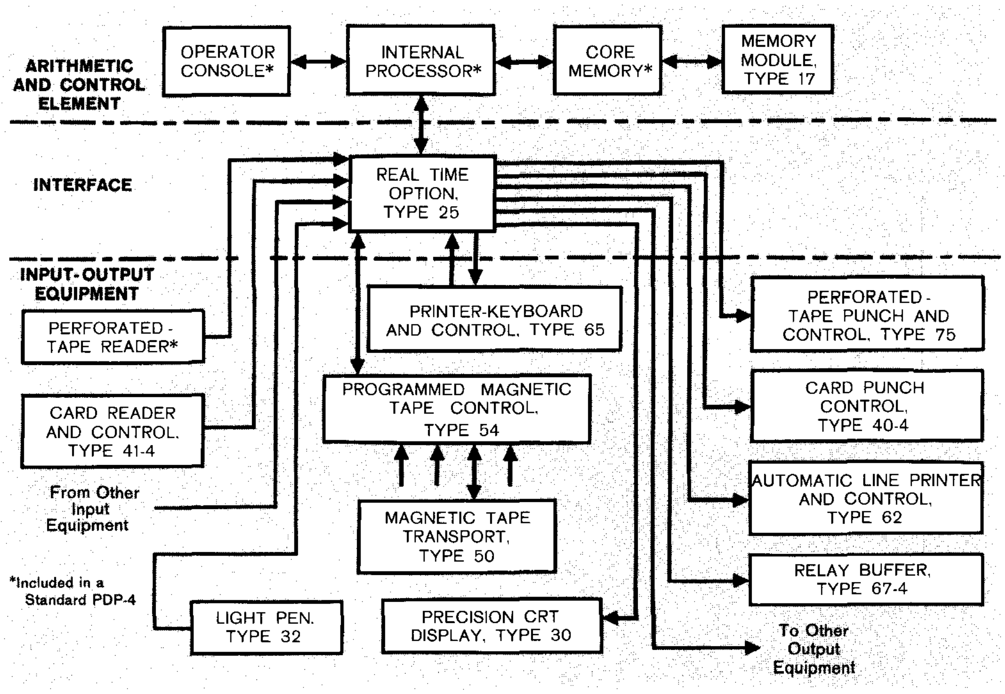
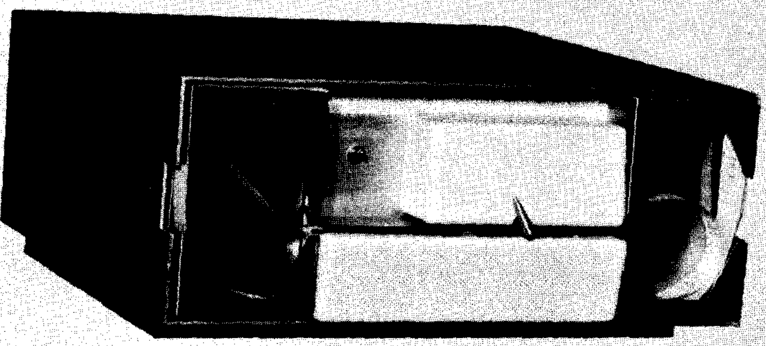
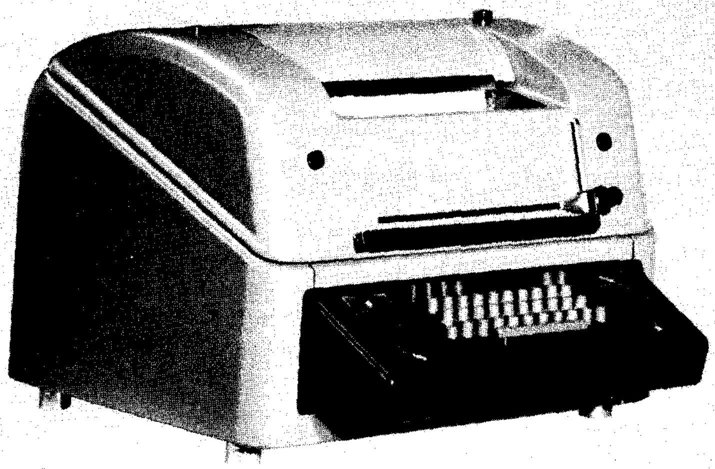

ARITHMETIC AND CONTROL ELEMENT
OPERATOR CONSOLE*
INTERNAL PROCESSOR*
CORE MEMORY*
MEMORY MODULE, TYPE 17
INTERFACE
REAL TIME OPTION, TYPE 25
INPUT-OUTPUT EQUIPMENT
PERFORATED-TAPE READER*
PRINTER-KEYBOARD AND CONTROL, TYPE 65
PERFORATED-TAPE PUNCH AND CONTROL, TYPE 75
PROGRAMMED MAGNETIC TAPE CONTROL, TYPE 54
CARD READER AND CONTROL, TYPE 41-4
CARD PUNCH CONTROL, TYPE 40-4
From Other Input Equipment
MAGNETIC TAPE TRANSPORT, TYPE 50
AUTOMATIC LINE PRINTER AND CONTROL, TYPE 62
*Included in a Standard PDP-4
RELAY BUFFER, TYPE 67-4
LIGHT PEN, TYPE 32
PRECISION CRT DISPLAY, TYPE 30
To Other Output Equipment

Figure 2 — PDP-4 System with Real-Time Option

THE DEVICE SELECTOR consists of decoding elements to select and establish the state of an external device when the program issues an input-output transfer instruction. The direction of information transfer (in or out of the Internal Processor) is controlled by signals produced by the Device Selector. Up to 64 input-output devices can be selected and these, in turn, may cause the selection of many more. The standard Device Selector has provisions for twenty selector elements.

THE INFORMATION COLLECTOR receives information from input devices (selected by the Device Selector) and transfers the information to the Internal Processor. Up to 18 bits of information can be collected simultaneously; 8 × 18 bits of information may be collected, broken into variable-sized words.





The Perforated-Tape Reader (top) and Printer-Keyboard (bottom).

THE INFORMATION DISTRIBUTOR distributes information from the Internal Processor to all output devices. Only the output device selected (or addressed) by the Device Selector samples and reads in the information contained in the Information Distributor. Up to 8 $\times$ 18 bits may be distributed.

THE INPUT-OUTPUT SKIP CONNECTION provides a program skip instruction conditioned by the state of a given input-output device logic line. The instruction following the skip instruction will not be executed if the line is a 1. Eight skip conditions may be sampled.

THE PROGRAM INTERRUPT permits one of 11 lines (conditions) or input-output devices to interrupt the program and initiate a subroutine which may return to the original program when the cause for interruption has been processed. The machine state is preserved during a Program Interrupt. This type of interrupt is suited for information or event rates in the range of 0 to 2,000 cycles per second.

THE DATA INTERRUPT allows a device to automatically interrupt the program and deposit or extract data from the Core Memory at an address specified by the device. The Data Interrupt is suited for high speed information transfers; up to 125,000 18-bit words may be transferred per second.

THE CLOCK/TIMER produces a signal which increments a Core Memory register at a rate of 60 cycles per second. When the register overflows, a Program Interrupt occurs.

INPUT-OUTPUT DEVICES

All of the input-output devices are optional except the Perforated-Tape Reader.

THE PERFORATED-TAPE READER senses 5-, 7-, or 8-hole perforated-tape at the rate of 300 lines per second. Either one line of tape (alphanumeric) or 3 lines of tape (binary word) may be read.

THE PERFORATED-TAPE PUNCH AND CONTROL, TYPE 75, perforates 5-, 7-, or 8-hole paper tape at a rate of 63.3 lines per second.

THE PRINTER-KEYBOARD AND CONTROL, TYPE 65, includes a Teletype Model KSR-28 Printer and Keyboard with an allowable input or printing rate of ten characters per second. Typed information may be monitored by a program. A program may print information.

THE PRECISION CRT DISPLAY, TYPE 30, displays data on a 9¼" by 9¼" area. Information is plotted point by point to form either graphical or tabular data. Operation of this device requires the Real-Time Option.

THE LIGHT PEN, TYPE 32, is a photoelectric device which detects information displayed on the Type 30 Visual CRT Display. Upon signal from the Light Pen, the computer carries out previously programmed instructions. Requires Real-Time Option.

THE 18-BIT RELAY BUFFER, TYPE 67-4, provides contacts which operate devices of higher power rating. The relays have form "D" contacts, which open and close in approximately 3 milliseconds. Requires Real-Time Option.

THE PROGRAMMED MAGNETIC TAPE CONTROL, TYPE 54, controls up to four Magnetic Tape Transports, Type 50. Information is read from or written on the tape. The format on the tape may be programmed to be compatible with IBM tapes having a density of 200, 6 + 1 bit characters per inch. Requires Real-Time Option.

THE MAGNETIC TAPE TRANSPORTS, TYPE 50, are used with the Programmed Magnetic Tape Control, Type 54.

THE AUTOMATIC LINE PRINTER AND CONTROL, TYPE 62, operates at up to 600 lines per minute, 120 columns per line. Each column may print one of 64 characters. Spacing format is controlled by a punched format tape in the Printer. Once a command to print or space is given, the Internal Processor is not required. Approximately one per cent of program running time is required to operate the Line Printer at a 600-line-per-minute rate. Requires Real-Time Option.

THE CARD READER AND CONTROL, TYPE 41-4, operates at a rate of up to 200 cards per minute. Cards are read column by column. Column information may be read in alpha-numeric or binary mode. The alpha-numeric mode converts the 12-bit Hollerith Code of one column into the six-bit binary-coded decimal code with code validity checking. The binary mode reads a 12-bit column directly into the PDP-4. Approximately one per cent of a Card Reader program running time is required to read the 80 columns of information at the 200 cards per minute rate. Requires Real-Time Option.

THE CARD PUNCH CONTROL, TYPE 40-4, enables the operation of a standard IBM Type 523 Summary Punch with PDP-4. Cards are punched row by row at a rate of 100 cards per minute. Approximately 0.3 per cent of program running time is required to operate the Card Punch at the 100-card-per-minute rate. Requires Real-Time Option.

PROGRAMMING AIDS

Several programs are supplied with each PDP-4 to assist the programmer in routine tasks. They include: The PDP-4 Assembly Program, the DDT-4 debugging tape, double-precision floating point routines, maintenance routines, a tape reproducer, punch routines, an octal debugging routine, an algebraic compiler, and a floating point functions program which will enable various functions, such as double precision floating-point sine, to be computed. See Appendix 6.

CHAPTER 2

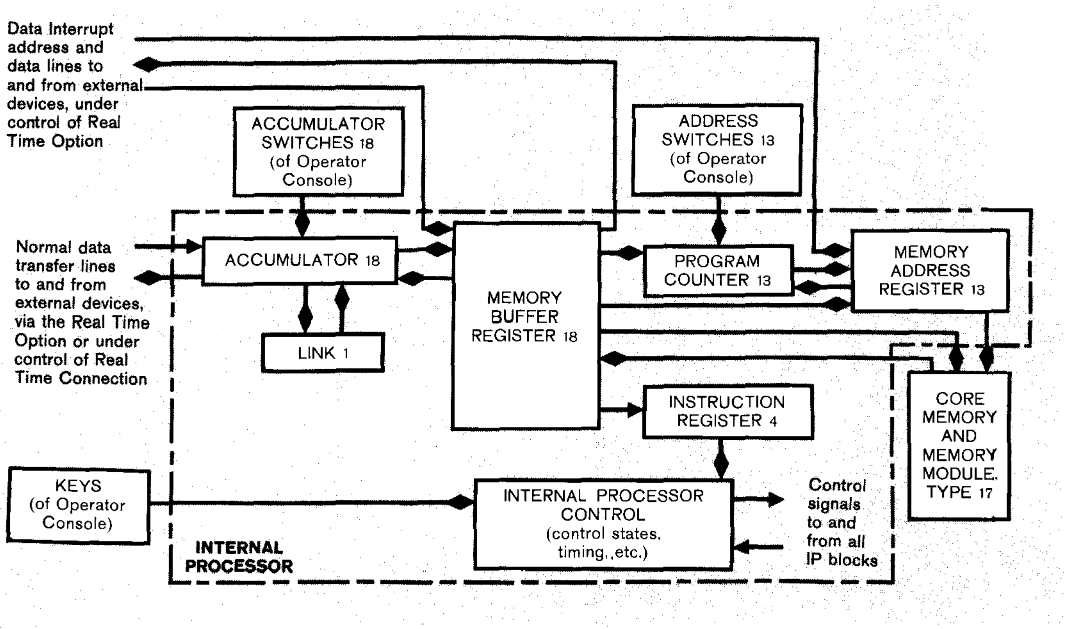
ARITHMETIC AND CONTROL ELEMENT

In this chapter the functions of the Arithmetic and Control Element are described in detail. The operations of the machine instructions are explained and listed.

Functions

INTERNAL PROCESSOR

The Internal Processor performs arithmetic operations, controls memory access, and handles information entering and leaving the machine. It consists of the Information Processor Control, which oversees all activities, and six registers: Accumulator, Link, Memory Buffer, Memory Address, Instruction, and Program Counter. The elements of the Internal Processor are shown within the broken line in Figure 3.



Data Interrupt address and data lines to and from external devices, under control of Real Time Option
ACCUMULATOR SWITCHES 18 (of Operator Console)
ADDRESS SWITCHES 13 (of Operator Console)
Normal data transfer lines to and from external devices, via the Real Time Option or under control of Real Time Connection
ACCUMULATOR 18
LINK 1
MEMORY BUFFER REGISTER 18
PROGRAM COUNTER 13
MEMORY ADDRESS REGISTER 13
INSTRUCTION REGISTER 4
CORE MEMORY AND MEMORY MODULE, TYPE 17
KEYS (of Operator Console)
INTERNAL PROCESSOR CONTROL (control states, timing,.etc.)
Control signals to and from all IP blocks
INTERNAL PROCESSOR

Figure 3 — Arithmetic and Control Element

ACCUMULATOR (AC): Arithmetic operations are performed in this 18-bit register. The AC may be cleared and complemented. Its contents may be rotated right or left with the Link. The contents of the Memory Buffer may be added to the contents of the AC with the result left in the AC. The contents of both these registers may be combined by the logical operations AND and Exclusive OR, the result remaining in the AC. The Inclusive OR may be formed between the AC and the Accumulator Switches on the Operator Console (see below), and the result left in the AC.

The Accumulator also acts as an input-output register. Under normal operation all information transfers between core memory and an external device must pass through the Accumulator.

LINK (L): This is a one-bit register used to extend the arithmetic facility of the Accumulator. In 1's complement arithmetic, the Link is an overflow indicator; in 2's complement it functions as a carry register. The Link may be cleared and complemented and its state sensed independent of the AC. It is included with the AC in rotate operations.

MEMORY BUFFER (MB): All information transferred between Core Memory and the AC, Instruction Register, or Program Counter passes through the MB. Information is read from a memory cell into the MB and rewritten into the cell in one cycle time (8 microseconds). Instructions are brought from memory into the MB to be decoded. The MB serves also as a buffer for information transferred between Core Memory and an external device in a Data Interrupt. The contents of the MB may be incremented by one.

MEMORY ADDRESS REGISTER (MA): The address of the Core Memory cell currently being accessed is contained in the 13-bit MA. Information may enter the MA from the MB, Program Counter, or from an external device operating in a Data Interrupt.

INSTRUCTION REGISTER (IR): This is a 4-bit register which contains the operation code of the instruction currently being performed by the computer. Information enters the IR from the MB.

PROGRAM COUNTER (PC): The program sequence, that is, the order in which instructions are performed, is determined by the PC. This 13-bit register contains the address of the memory cell from which the next instruction will be taken. Information may enter the PC from the MB, MA, or the Address Switches of the Operator Console.

MEMORY

The memory contains stored information for processing, and the instructions of the program being run. Memory capacities of from 1,024 to 32,768 words are available in PDP-4. Standard models PDP-4A and PDP-4B come with 1024-word and 4096-word memories, respectively. The two models are identical in all other respects. The smaller memory has a 32 by 32 by 18 core array, the larger a 64 by 64 by 18 core array. A Memory Module Type 17, containing a 64 by 64 by 18 core array may be added to PDP-4B to give it an 8192-word capacity. With the addition of the Magnetic Core Memory Extension Control Type 16, memory modules may be added to build a memory of 32,768 words. Further increase in storage capacity can be gained by adding the Magnetic Drum System Type 24, available in three capacities: 16,384, 32,768, and 65,536 words.

OPERATOR CONSOLE

The Operator Console contains all the switches and controls necessary to run the machine, and lights which indicate the current status of the Internal Processor. The functions of the lights and controls are described in the following tables.

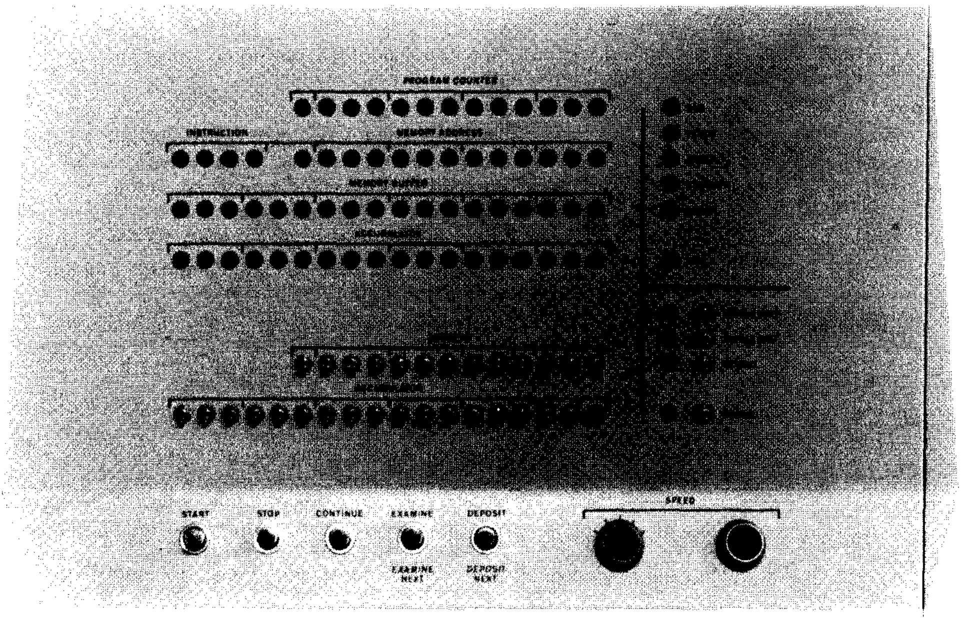


Figure 4 — Operator Console

Console Switches	Function
ADDRESS	A group of 13 switches which establishes the memory address for the START, EXAMINE, and DEPOSIT operations.
ACCUMULATOR	A group of 18 switches, the setting of which determines the word to be placed in memory by the DEPOSIT and DEPOSIT NEXT operations, or to be placed in the AC under program control.
POWER	Controls the primary power to the computer and all external devices attached to it.
SINGLE STEP	Causes the computer to stop at the completion of each memory cycle. Repeated operation of CONTINUE while this switch is on will step the program one cycle at a time.
SINGLE INSTRUCTION	Causes the computer to stop at the completion of each instruction. Repeated operation of CONTINUE while this switch is on will step the program one instruction at a time. When both switches are on, SINGLE STEP takes precedence over SINGLE INSTRUCTION.
REPEAT	Causes the operations initiated by pressing CONTINUE, EXAMINE NEXT, or DEPOSIT NEXT, to be repeated as long as the key is held on. The rate of repetition is controlled by the setting of the SPEED knobs.
SPEED	Two controls that vary the REPEAT interval from approximately 40 microseconds to 8 seconds. The left knob is a five-position coarse control, the right knob a continuously variable fine control. For both knobs, slowest speed is obtained in extreme left position.

Console Light	Indication
ACCUMULATOR	The contents of the AC.
MEMORY BUFFER	The contents of the MB.
LINK	The contents of the Link.
MEMORY ADDRESS	The contents of the MA register.
INSTRUCTION	The contents of the IR.
PROGRAM COUNTER	The contents of the PC.
RUN	The computer is executing instructions.
FETCH, DEFER, EXECUTE, BREAK	The primary control state of the next memory cycle.

Console Key	Function
START	Starts the processor. The first instruction is taken from memory cell specified by the setting of the ADDRESS switches. The START operation clears the AC and Link, and turns off the Program Interrupt.
STOP	Stops the processor at the completion of the memory cycle in progress at the time of key operation.
CONTINUE	Causes the computer to resume operation from the point at which it was stopped by the last previous operation of STOP or one of the EXAMINE or DEPOSIT keys. Besides the normal off and momentary on positions, CONTINUE has a latched on position obtained by raising the key instead of depressing it.
EXAMINE	Places the contents of the memory cell specified by the ADDRESS switches in the AC and MB. The contents of the ADDRESS switches appear in the MA. The PC contains the address of the next cell.
EXAMINE NEXT	Places the contents of the cell specified by the PC in the MB and AC. The C(PC) are incremented by one. The MA contains the address of the register examined.
DEPOSIT	Deposits the contents of the AC switches in the memory cell specified by the ADDRESS switches. The C(AC switches) remain in the AC and MB. The contents of the ADDRESS switches appear in the MA. The PC contains the address of the next cell.
DEPOSIT NEXT	Deposits the contents of the AC switches in the memory cell specified by the PC. The C(PC) are then incremented by one. The C(AC), C(MB), and C(MA) are the same as for DEPOSIT.

Control States

The PDP-4 operates in one of four primary control states during a memory cycle: Fetch, Defer, Execute, or Break. The next control state is established at the completion of the current one. All states except Break are determined by the instructions themselves.

FETCH: A new instruction is obtained when this state occurs. The contents of the memory cell specified by the PC are placed in the MB, and the instruction part (bits 0-4) of this word are placed in the IR. The C(PC) are then incremented by one.

If a two-cycle instruction is fetched, the following control state will be either Defer or Execute. If a one-cycle instruction is fetched, the operations specified will be performed during the last part of the Fetch cycle. The next state will be Fetch.

DEFER: When bit 4 of a memory reference instruction is a 1, the Defer state is entered to perform the indirect addressing. The process of indirect addressing is often referred to as deferring, in the sense that access to the operand is deferred once to another memory cell. This is why the primary control state in which this operation is performed is called Defer. Bit 4 of a memory reference instruction is referred to interchangeably as the Indirect or the Defer Bit.

EXECUTE: This state is established only when a memory reference instruction is being performed. The contents of the memory cell addressed are brought into the MB, and the operation specified by the C(IR) is performed.

BREAK: When this state is established, the sequence of instructions is broken for a Data Interrupt or a Program Interrupt. In both cases, the break occurs only at the completion of the current instruction.

The Data Interrupt allows information to be transferred between memory and an external device; when this transfer has been completed, the program sequence is resumed from the point of the break. The Program Interrupt causes the sequence to be altered. The C(PC) and the C(L) are stored in location 0000 and the program continues from location 0001.

Instructions

The instruction code is specified by bits 0-3 of a word. There are two types of instructions: Memory Reference and Augmented.

MEMORY REFERENCE INSTRUCTIONS

The bit assignment of the memory reference instruction is shown in Figure 5. Bits 0-3 determine the operation to be performed. Bits 5-17 specify the address of the memory cell containing the operand. If bit 4 is a 1, then indirect addressing occurs. In the following discussion, i is the mnemonic symbol used to indicate indirect addressing.

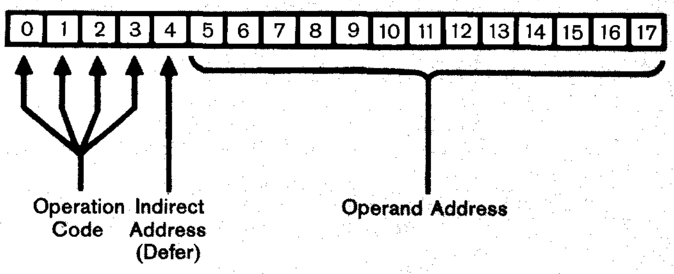


Figure 5 — Memory reference instruction format

INDIRECT ADDRESSING

When indirect addressing is specified, the address part (bits 5-17) of a memory reference instruction is interpreted as the address of a cell containing not the operand, but the address of the operand. Consider the instruction add A. Normally, A is interpreted as the address of the cell containing the quantity be be added to the AC. Thus, if cell 100 contains the number 576, the instruction

    add 100

will cause the quantity 576 to be added to the AC. Now suppose that cell 576 contains the number 1135. The instruction

    add i 100

(where i signifies indirect addressing) will cause the computer to take the number 576, which is in cell 100, as the effective address of the instruction, and the number in cell 576 as the operand. Hence this instruction will result in the quantity 1135 being added to the AC.

If, when indirect addressing is indicated, the memory cell addressed by the instruction is one of those in locations 10-17, the contents of that cell are incremented by one and the result taken as the effective address. This feature is called auto-indexing. If memory cell 12 contains the number 200, the instruction

    add i 12

will cause the number in cell 200 + 1 to be added to the AC.

1'S COMPLEMENT ARITHMETIC

When two numbers are added together in 1's complement arithmetic (see add instruction in following table), a 1 carried out of the high-order position will be added to the low-order digit, as follows:

```
   110101001100011
   011001010111101
   ---------------
 1 001110100100000
 └──────────────►1
   ---------------
   001110100100001
```

Since bit 0 of a word is used for the sign of a number, the largest positive number that can be represented is $2^{17}-1$. If, in 1's complement addition, the addends are of like sign and the sign of the sum is different, overflow is said to have occurred and the Link is set to 1.

2'S COMPLEMENT ARITHMETIC

In 2's complement addition (see tad instruction), a carry out of the high-order bit is not added into the low order position. Instead, if a carry occurs, the Link is complemented. The signs of the addends and sum are not examined. Two's complement addition is used primarily in multiple precision arithmetic.

All memory reference instructions require an Execute cycle (see Control States above) to transfer data between Core Memory and the MB and execute the instruction. When indirect addressing is specified, an extra cycle is required to determine the effective address. The jmp instruction, while it requires an address, does not require an operand; an Execute cycle is thus not needed, and the instruction is performed in only one cycle.

MEMORY REFERENCE INSTRUCTIONS

Explanation of Special Terms

C(A)	contents of A	$\veebar$	exclusive OR
A => B	A replaces B	$\vee$	inclusive OR
Y_{1-4}	bits 1 - 4 of Y	$\wedge$	AND
Y_j	a given bit in Y	$\bar{A}$	1's complement of A

MNEMONIC SYMBOL	OCTAL CODE (BITS 0-3)	TIME (μsec)	OPERATION
lac Y	20	16	Load AC. The C(Y) are loaded into the AC. The previous C(AC) are lost. C(Y) => C(AC).
dac Y	04	16	Deposit AC. The C(AC) are deposited in the memory cell at location Y. The previous C(Y) are lost; the C(AC) are unchanged. C(AC) => C(Y).
dzm Y	14	16	Deposit Zero in Memory. Zero is deposited in memory cell Y. The original C(Y) are lost. The AC is unaffected by this operation. 0 => C(Y).
add Y	30	16	Add (1's complement). The C(Y) are added to the C(AC) in 1's complement arithmetic. The result is left in the AC and the original C(AC) are lost. The C(Y) are unchanged. The Link is set to 1 on overflow. (See text). C(Y) + C(AC) => C(AC).

MNEMONIC SYMBOL	OCTAL CODE (BITS 0-3)	TIME (μsec)	OPERATION
tad Y	34	16	Two's complement Add. The C(Y) are added to the C(AC) in 2's complement arithmetic. The result is left in the AC and the original C(AC) are lost. The C(Y) are unchanged. A carry out of the 0 bit complements the Link. C(Y) + C(AC) => C(AC).
xor Y	24	16	Exclusive OR. The logical operation Exclusive OR is performed between the C(Y) and the C(AC). The result is left in the AC and the original C(AC) are lost. The C(Y) are unchanged. Corresponding bits are compared independently. $C(Y_j) \vee C(AC_j) => C(AC_j)$.

Example

$C(AC)_j$ original	$C(Y)_j$	$C(AC)_j$ final
0	0	0
0	1	1
1	0	1
1	1	0

MNEMONIC SYMBOL	OCTAL CODE (BITS 0-3)	TIME (μsec)	OPERATION
and Y	50	16	AND. The logical operation AND is performed between the C(Y) and the C(AC). The result is left in the AC, and the original C(AC) are lost. The C(Y) are unchanged. Corresponding bits are compared independently. $C(Y_j) \wedge C(AC_j) => C(AC_j)$

Example

$C(AC)_j$ original	$C(Y)_j$	$C(AC)_j$ final
0	0	0
0	1	0
1	0	0
1	1	1

MNEMONIC SYMBOL	OCTAL CODE (BITS 0-3)	TIME (μsec)	OPERATION
sad Y	54	16	Skip if AC is Different from Y. The C(Y) are compared with the C(AC). If the numbers are the same, the computer proceeds to the next instruction. If the numbers are different, the next instruction is skipped. The C(AC) and the C(Y) are unchanged. If $C(AC) \neq C(Y)$ then C(PC) + 1 => C(PC).

MNEMONIC SYMBOL	OCTAL CODE (BITS 0-3)	TIME (μsec)	OPERATION
isz Y	44	16	Index and Skip if Zero. The C(Y) are incremented by one in 2's complement arithmetic. If the result is 0, the next instruction is skipped. If not, the computer proceeds to the next instruction. The C(AC) are unaffected. C(Y) + 1 => C(Y). If result = 0, C(PC) + 1 => C(PC).
jmp Y	60	8	Jump to Y. The next instruction to be executed is taken from memory cell Y. Y => C(PC).
jms Y	10	16	Jump to Subroutine. The C(PC) and the C(L) are deposited in memory cell Y. The next instruction is taken from cell Y + 1. C(L) => C(Y_0). 0 => C(Y_{1-4}). C(PC) => C(Y_{5-17}). Y + 1 => C(PC).
cal	00	16	Call Subroutine. The address portion of this instruction is ignored. The action is identical to jms 20. The instruction cal i is equivalent to jms i 20.
xct Y	40	8 + time of instruction being executed	Execute. The instruction in memory cell Y will be executed. The computer will act as if the instruction located in Y were in the place of the xct.

AUGMENTED INSTRUCTIONS

None of the augmented instructions require a memory reference. Bits 4–17 of an augmented instruction are used to specify operations, many of which may be combined in a single instruction. There are three classes of augmented instructions:

a. Operate class: includes operations on the AC and Link, the skip group, and the halt instruction.

b. The special instruction, law.

c. Input-output transfer class: includes all the instructions which initiate transfers of information between the Internal Processor and an external device and those that sense the status of the devices.

OPERATE CLASS

The instructions of the Operate class require one cycle for their execution. The octal code (bits 0–3) for this class is 74. The operations specified by bits 4–17 are called micro-instructions. The functions of each micro-instruction are described in the following table. The Event Time indicates when the operation is performed in the course of the cycle. Times 0, 1, and 2 occur in that order in the latter part of the cycle.

Except for the restrictions indicated at the end of the table, micro-instructions may be combined in a single instruction. The bit assignment of the Operate class micro-instructions is shown in Figure 6.

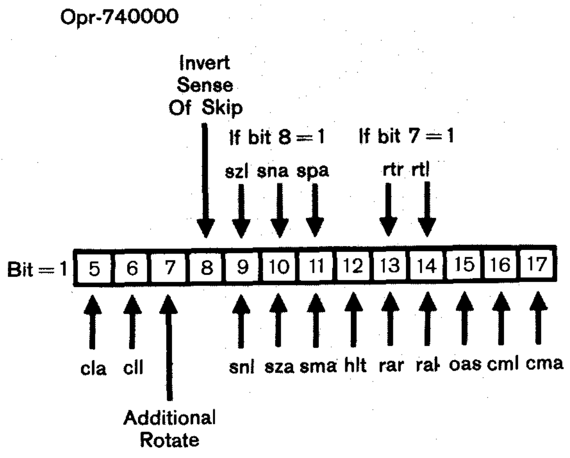


Figure 6 — Operate class instruction — bit assignment

MNEMONIC SYMBOL	OCTAL CODE	EVENT TIME	OPERATION
opr	740000		Operate. Indicates the Operate class. When used alone, performs no operation; the computer proceeds to the next instruction.
cla	750000	2	Clear AC. The AC is cleared to 0. 0 => C(AC).
cma	740001	3	Complement AC. Each bit of the AC is complemented. $\overline{C(AC)}$ => C(AC).
cll	744000	2	Clear Link. Link is set to 0. 0 => C(L).
cml	740002	3	Complement Link. $\overline{C(L)}$ => C(L).

MNEMONIC SYMBOL	OCTAL CODE	EVENT TIME	OPERATION
ral	740010	3	Rotate AC Left. The C(AC) and the C(L) are rotated left one place. $C(AC_j) => C(AC_{j-1})$ $C(AC_0) => C(L)$. $C(L) => C(AC_{17})$
rtl	742010	2, 3	Rotate Two places Left. Equivalent to two successive ral's.
rar	740020	3	Rotate AC Right. The C(AC) and the C(L) are rotated one place right. $C(AC_j) => C(AC_{j-1})$ $C(L) => C(AC_0)$ $C(AC_{17}) => C(L)$
rtr	742020	2, 3	Rotate Two Places Right. Action taken is equivalent to two successive rar's.
oas	740004	3	OR AC Switches. The Inclusive OR of the C(AC) and the C(AC switches) is placed in the AC. A switch up is interpreted as a 1. $C(AC\ Switches) \vee C(AC) => C(AC)$.

Example

$C(AC)_j$ original	$C(Y)_j$	$C(AC)_j$ final
0	0	0
0	1	1
1	0	1
1	1	1

MNEMONIC SYMBOL	OCTAL CODE	EVENT TIME	OPERATION
sma	740100	1	Skip if Minus AC. If the AC is negative, the next instruction is skipped. If $AC_0 = 1$, then $C(PC) + 1 => C(PC)$.
spa	741100	1	Skip if Plus AC. If the AC is positive, the next instruction is skipped. If $AC_0 = 0$, then $C(PC) + 1 => C(PC)$.
sza	740200	1	Skip if Zero AC. If C(AC) are 0, the next instruction is skipped. If $C(AC) = 0$, then $C(PC) + 1 => C(PC)$.
sna	741200	1	Skip if Non-zero AC. If $C(AC) \neq 0$, then $C(PC) + 1 => C(PC)$.

MNEMONIC SYMBOL	OCTAL CODE	EVENT TIME	OPERATION
snl	740400	1	Skip if Non-zero Link. If C(L) is 1, the next instruction is skipped. If C(L) $\neq$ 0, then C(PC) + 1 => C(PC).
szl	741400	1	Skip if Zero Link. If C(L) = 0, then C(PC) + 1 => C(PC).
hlt	740040	immediately after the completion of the cycle.	Halt. Stops the computer.

If skips are combined in a single instruction, the Inclusive OR of the conditions to be met will determine the skip. For instance, if both sza and snl are indicated (octal code 740600), the next instruction will be skipped if either the AC is zero or the Link is non-zero, or both.

If ral or rar is specified, cma, cml, oas may not be specified, and conversely. If rtl or rtr is specified, cma, cml, cla, cll, oas may not be specified, and conversely.

THE INSTRUCTION, law

The octal code for this instruction is 760000. Bits 5-17 are used to specify a quantity to be placed in the AC. The effect of the law instruction is to place itself in the AC.

law Y	76	8 μsec	Load AC With law Y.

INPUT-OUTPUT TRANSFER CLASS

The instructions in this class are used to effect information transfers between the Internal Processor and external devices, via the Interface.

iot	760000	8 μsec	Input-Output Transfer. Bits 4-13 of an iot instruction determine the device and subdevice to be selected. The presence of a 1 in bit 14 will cause the AC to be cleared at Event time 1. Bits 15-17 determine when pulses are to be sent to the selected device.

The bit assignment of the iot instruction is shown in Figure 7. The instructions of the iot class are described in Chapter 3.

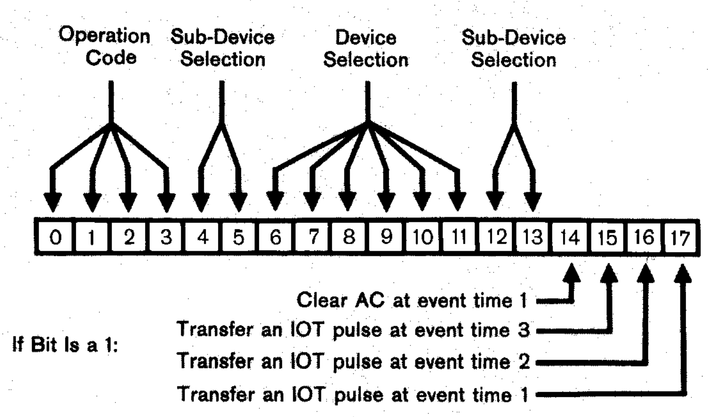


Figure 7 — Bit assignment for input-output transfer instruction (iot)

CHAPTER 3

INPUT-OUTPUT EQUIPMENT FUNCTIONS AND PROGRAMMING

PDP-4 is capable of operating with the ten input-output devices described in Chapter 1 and with a variety of others. The computer can operate with most of the devices simultaneously. The Interface, consisting of the Real-Time Connection or the Real-Time Option, issues commands to the devices, monitors their state of availability, transfers information to them, and receives information from them. Since the Internal Processor can store or read out data much faster than the devices can operate, the Interface and the individual devices provide buffering to minimize the amount of program time consumed in transfers.

The Real-Time Connection, furnished as standard equipment, provides communication between the Internal Processor and the Perforated-Tape Reader, the Perforated-Tape Punch, and the Keyboard-Printer. The Real-Time Option, Type 25, gives the system the additional capability to operate efficiently over a wide range of information handling rates, from seconds per event to 125,000 words per second, and with a large variety of input-output devices. The Real-Time Option consists of the Device Selector, the Information Collector, the Information Distributor, the Input-Output Skip Facility, the Program Interrupt Control, the Data Interrupt Control, and the Clock/Timer (see Figure 8).

The coupling of input-output equipment to PDP-4 is similar for all devices. The electrical characteristics of the coupling are discussed in Chapter 4. The logical functions and programming instructions are given below.

Input-Output Commands

DEVICE SELECTOR (DS)

The input-output transfer (iot) augmented instruction causes the Interface to produce pulses which select IO devices and transfer information. Upon receipt of an instruction, the Device Selector in the Interface performs one of the following functions:

(a) Starts a device (e.g.asks for a line of perforated tape to be read and assembled into a word, a card to be moved to a reading or punching station, etc.)

(b) Transfers data from the information buffer of an input device to the AC, through the Information Collector

(c) Transfers information from the AC, through the Information Distributor to the buffer of an output device

(d) Senses the flag(s) associated with a device to determine its availability

(e) Resets the flags. These commands dismiss a device without asking for additional action.

The flags referred to above are signals generated by an external device upon completion of its assigned task. This technique allows the Internal Processor to resume its arithmetic operations after issuing an instruction to a relatively slow input-output device (data rate of less than 20,000 words per second). When a flag is set to 1 by the device, it signifies that:

(a) an output action (punch out, etc.) has been completed; the Arithmetic and Control Element may transmit data to the device.

(b) an input action (card or tape input, etc.) has occurred; information is available for the Arithmetic and Control Element.

(c) an alarm condition exists.

Flags may be sensed, and a program skip take place, using the Output Skip Facility (see below). Flags may be read into the AC using the iors (in-out read status) instruction. Most flags are connected to the Program Interrupt (see below).

The Device Selector selects an input-output device or subdevice according to the address code of the device in bits 4-13 in the iot instruction. It then generates IO pulses at event times 0, 1, and 2 if the appropriate micro-instruction code bits are present in bits 17, 16, and 15. Pulse iot 0 occurs near the end of an iot instruction, followed by iot 1 in 2 microseconds. Pulse iot 2 occurs at the beginning of the next instruction, 1.2 microseconds after iot 1. This timing enables one iot instruction to perform multiple operations.

INFORMATION COLLECTOR (IC)

The Information Collector enables information to be collected from eight 18-bit word input devices. The AC must contain 0 at the time the inputs are sampled. A word can be broken into smaller words according to the word size requirements of the input device. The program steps for reading the contents of a group of static parallel data bits are:

cla	Clear the AC (AC must equal 0)
iot	Selected device (sample the selected device outputs)
dac Y	Deposit C(AC). The C(AC) are sent to a particular memory cell, Y. (the first two steps may be microprogrammed together in one instruction)

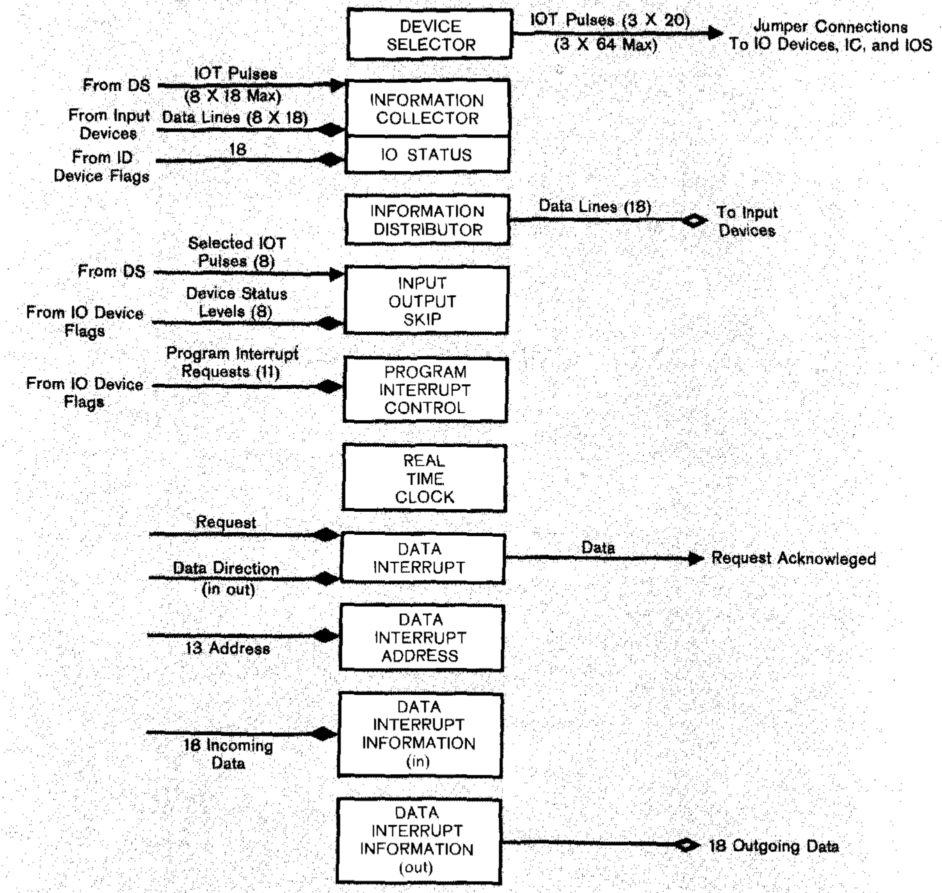


Figure 8 — Real Time Option, Type 25

INFORMATION DISTRIBUTOR (ID)

The Information Distributor presents the static data contained in the AC to each output device requiring AC information. The devices sample the Information Distributor using the program-controlled pulses from the Device Selector. The program steps for transmitting information from a particular memory cell are:

lac Y	Load the AC with C(Y)
iot	Clear selected output register to prepare for information
iot transmit	The information is sampled and placed in the register of the input-output device. *(the second two steps may be microprogrammed together in one instruction)*

INPUT-OUTPUT SKIP FACILITY (IOS)

The Input-Output Skip facility enables the program to skip (or branch) according to various external device states. There are eight inputs to the Skip facility. The iot pulses from the Device Selector strobe an input line and if a logic condition is present, the instruction following the iot is skipped. The iot skip pulse must occur at event time 1.

PROGRAM INTERRUPT CONTROL (PIC)

The program interrupt allows a logic line state to interrupt the program. It is used to speed the processing of input-output device information, or to allow certain alarm conditions to be sensed by the computer. The interrupt may be enabled or disabled by the program.

When the interrupt occurs, the contents of the Program Counter and the Link are stored in memory location 0 (bits 0, 5 . . . 17) and an interrupt program begins in memory location 1. This action disables the interrupt mode. The interrupt program is responsible for finding the signal causing the interruption, for removing the condition, and for returning to the original program.

When the condition for interruption is removed, an iot signal to re-enable the Program Interrupt is given, followed by the instruction, jmp indirect 0, or 620000. The interrupt program will then resume. If a Program Interrupt request is waiting, it will be serviced after the 620000 instruction. If a second interruption condition occurs and the interrupt program is running, the signal will have no effect; that is, there is only one level of interruption. The START key disables the Program Interrupt system. The iot instructions for the program interrupt are:

iof — 700002 — Disable the Program Interrupt

ion — 700042 — Enable the Program Interrupt

INPUT-OUTPUT STATUS INSTRUCTION

The iors (in-out read status) instruction, 700314, enables the status of all IO devices to be read into the AC and sampled. Various IO device states are indicated by the presence of a 1 or 0 in the bit positions allocated for that device (see Figure 9).

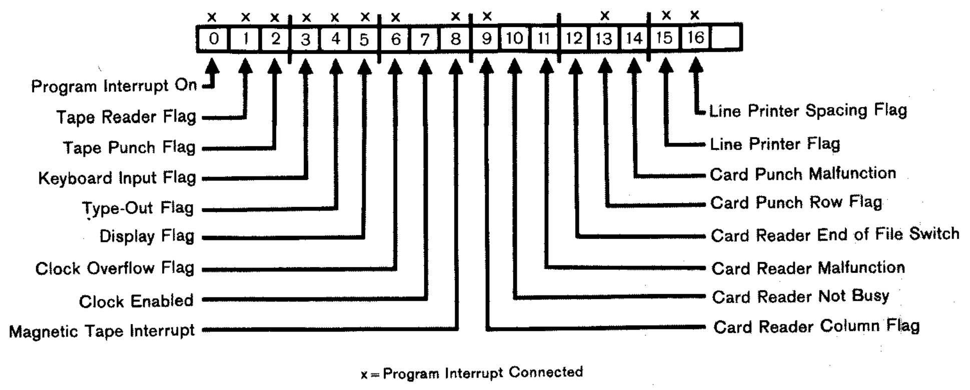


Figure 9 — Input-Output Status instruction, bit assignment

CLOCK/TIMER

The Clock produces a pulse every 1/60 second (16.6 milliseconds) which temporarily interrupts the program (in the same manner as the data interrupt) and a 1 is added to the contents of memory cell 7 using 2's complement addition. If the contents of memory cell 7 are 0 after the addition, the Clock flag is set to 1, which initiates a Program Interrupt if the Interrupt is on. Depressing the START key on the Operator Console clears the Clock flag and disables the Clock. The iot instructions associated with the Clock are:

csf — 700001 — Skip the next instruction if the Clock flag is a 1

cof — 700004 — Disable the Clock and clear the Clock flag

con — 700044 — Enable the Clock and clear the Clock flag

Register 7 is identical to other core memory registers, that is, its contents may be examined or modified. By presetting register 7 to a number, a Program Interrupt will occur when the register overflows after a timed interval.

Input-Output Devices

All of the Input-Output Devices discussed below can be controlled by the Real-Time Option, Type 25. The Real-Time Connection, furnished as standard equipment, provides communication between the Internal Processor and the Perforated-Tape Reader, the Perforated-Tape Punch and Control, and the Printer-Keyboard and Control. All devices except the Perforated-Tape Reader are optional. This section is arranged in the order of increasing complexity of connection.

PRECISION CRT DISPLAY, TYPE 30A

Data points are displayed on a 9¼ inch by 9¼ inch area. Information is plotted point by point to form either graphical or tabular data. Two digital-to-analog converters drive the deflection yokes in the X and Y directions. Data can be plotted at a 20 kc rate, or every 50 miscroseconds.

The program loads the AC with a point to be plotted. Bits 0 through 8 specify the X co-ordinate of the point and Bits 9 through 17 the Y co-ordinate. The C(AC) are then transferred to the Display Buffer. The specifying of the point initiates the plotting of the point on the CRT.

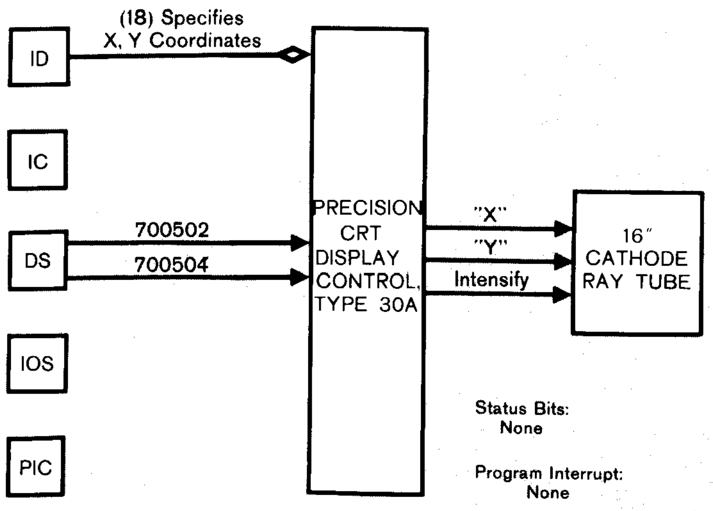


Figure 10 — Precision CRT Display, Type 30A programming logic

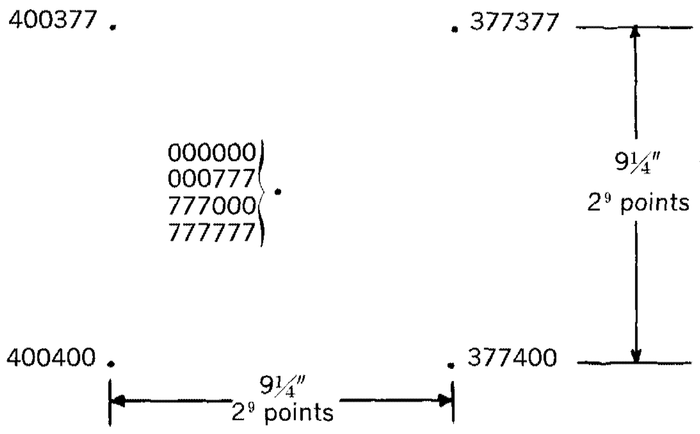
The CRT, Type 30A is selected when the numbers 0 and 5 (octal) are specified in bits 8 and 9 respectively, of the iot instruction. The display commands are:

dls — 700506 — Load the Display Buffer and select the display. The program loads the Display Buffer from the AC. A point is plotted as specified by the C(Display Buffer). The plotting requires 50 microseconds, after which another dls can be given. The Light Pen flag or Display flag is cleared with dls.

700502 — Clear the X and Y display buffers. 0 => C(Display Buffer).

700504 — C(AC) V C(Display Buffer)=>C(Display Buffer). Plot the point specified by the C(Display Buffer).

The points specified in the AC are plotted as unsigned quantities, beginning in the lower left hand corner of the cathode ray tube. The point locations are:



A program sequence is given in PDP-4 Assembly language below. The program begins in register 40, and plots a point, XY, as specified by Core Memory register 10.

PROGRAM SEQUENCE

```
/display a point 30a
10/  ...                          /xy bits 0-8, bits 9-17 y.
40/  lac 10                       /place xy co-ordinate in ac
                                  /display the point, next dls command
                                  /must wait 50 microsec.
```

LIGHT PEN, TYPE 32

The Light Pen is a photosensitive device which detects the presence of information displayed on a CRT. If the Light Pen is held in front of the CRT at a point displayed, the Display flag will be set to a 1. The Pen is specified by 0 and 5 in bits 8 and 9 of the iot instruction. The commands are:

dsf — 700501 — Skip if Display flag is a 1.

dcf — 700502 — Reset the Display flag to a 0.

The Display flag is connected to bit 5 of the iors instruction, and to the Program Interrupt.

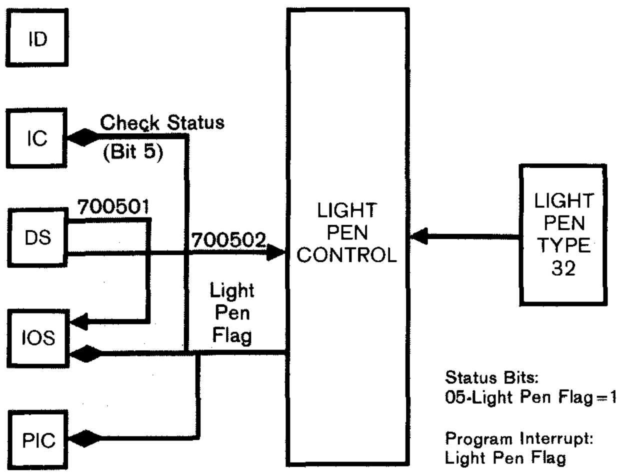


Figure 11 — Light Pen programming logic

PRECISION CRT DISPLAY, TYPE 30D AND LIGHT PEN, TYPE 32

The Type 30D display plots points at a 20kc rate. The X and Y co-ordinate buffers (XB and YB) are loaded from the 10 bits, AC_{8-17}.

The instructions are:

dsf — 700501 — Skip if the Display flag is a 1. The Display Flag is set to 1 when the Light Pen senses light.

dcf — 700601 — Clear the Display flag.

dxl — 700506 — Load the C(XB) with C(AC_{8-17}).

dyl — 700606 — Load the C(YB) with C(AC_{8-17}).

dxs — 700546 — Load the C(XB) with C(AC_{8-17}). Plot the point: C(XB), C(YB).

dys — 700646 — Load the C(YB) with C(AC_{8-17}). Plot the point: C(XB), C(YB).

dlb — 700706 — Load the Brightness Register with AC bits 15-17. The bits of AC specify the brightness of the points displayed. Clear the Display flag.

700502 — Clear XB.

700504 — C(SB) $\vee$ C(AC) => C(XB). Display a point.

700602 — Clear YB.

700604 — C(YB) $\vee$ C(AC) => C(YB). Display a point.

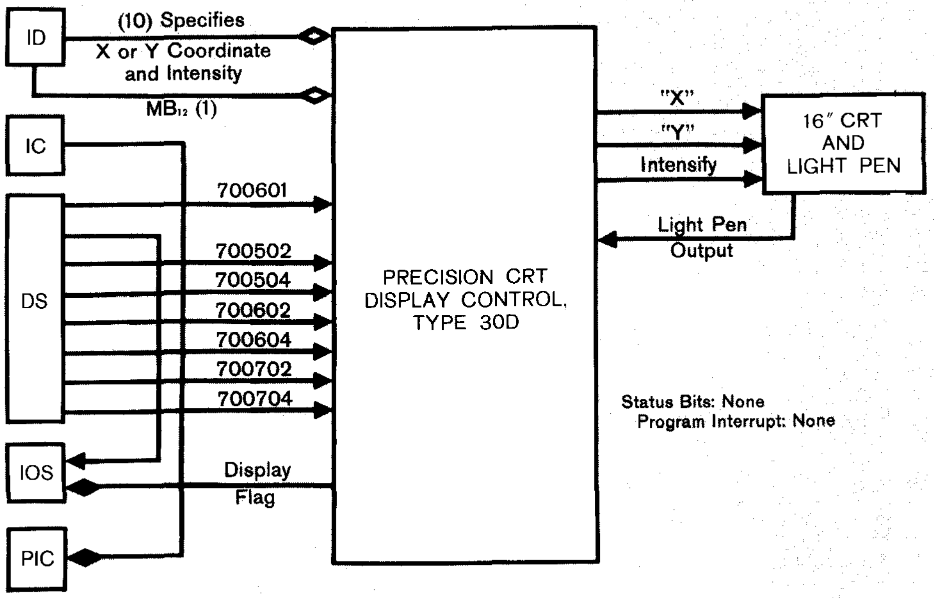
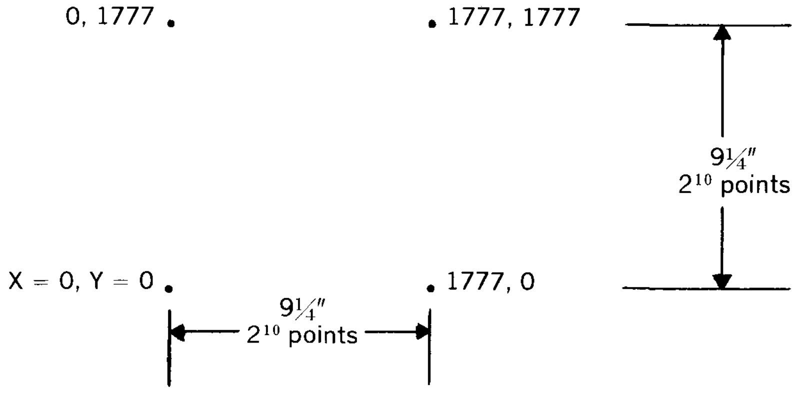


Figure 12 — Precision CRT Display, Type 30D, and Light Pen, Type 32

The Display flag is connected to the Program Interrupt and to bit 5 of the iors instruction. The co-ordinates of the corners are:



PROGRAM SEQUENCE

```
/display a point 30d
10/  . .            /x bits 8-17
     . .            /y
40/  lac 10
     dxl            /load x
     lac 11
     dys            /load y and plot the point
```

HIGH SPEED ANALOG-TO-DIGITAL CONVERTER (TYPICAL INPUT DEVICE)

An analog-to-digital converter with a resolution of 8 bits and a conversion time of 2 microseconds may be connected to the Real-Time Option. The input-output transfer instructions, series 11, for the converter are:

sci — 701115 — Sample the analog input. Convert the sampled quantity to digital form and load the AC with the converted number.

701101 — This micro-instruction starts the converter. In a period of 2 microseconds the converter will form an 8-bit number proportional to the analog input.

701104 — C(A-D converter) $\vee$ C(AC) => A(AC).

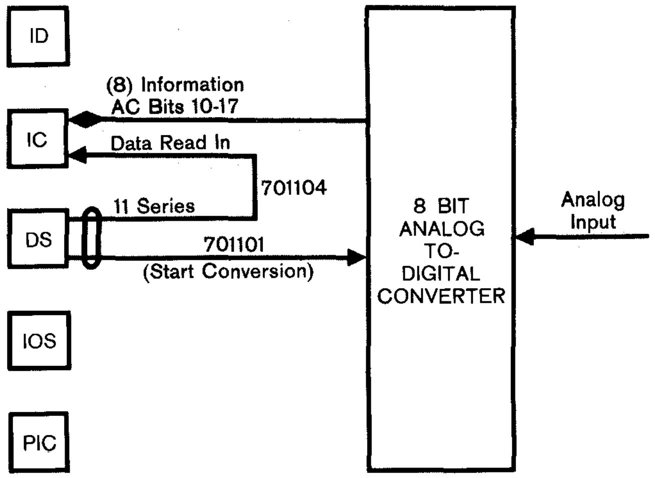


Figure 13 — High-speed analog-to-digital converter programming logic

A program sequence to sample a function at the input to the converter, and store the result in memory register 10 would be:

PROGRAM SEQUENCE

```
/analog-to-digital converter
10/                  /location of sampled result
42/   sci            /places sample in AC
      dac 10         /deposit result
```

LOW SPEED ANALOG-TO-DIGITAL CONVERTER (TYPICAL INPUT DEVICE)

An analog-to-digital converter with a resolution of 12 bits and a conversion time of 60 microseconds can be connected to PDP-4. The converter is given an iot command to sample the analog function, and in 60 microseconds the converter will contain a 12-bit number proportional to the input. At the completion of the sample, the converter flag is set to a 1, signifying that the input data is ready.

The contents of the converter buffer are read into the AC with a program command. The action which transfers the information from the converter to the AC also resets the converter flag. An iot skip instruction is used which skips if the conversion is complete; i.e., the converter flag is a 1. The program instructions, iot series 11, are:

asf — 701101 — Skip if the converter flag is a 1.

arb — 701112 — Read converter buffer and clear converter flag.

ase — 701104 — Start the converter and clear the converter flag.

701102 — A micro-instruction which clears the converter flag, and C(converter buffer) V C(AC) => C(AC).

The converter flag might connect to the Program Interrupt.

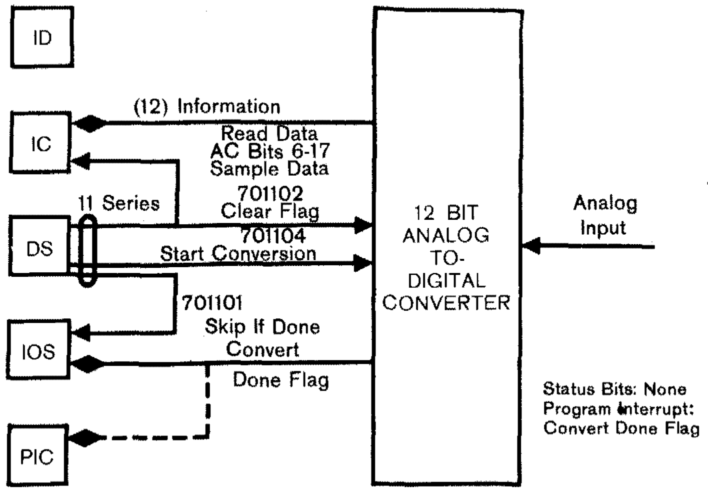


Figure 14 — Slow-speed analog-to-digital converter programming logic

PERFORATED-TAPE READER

The Tape Reader senses 5-, 7-, or 8-hole perforated-paper (or Mylar) tape photoelectrically at 300 characters (or lines) per second. The Reader control requests Reader movement, assembles data from the Reader into a Reader Buffer (RB), and signals the computer when incoming data is present. Reader tape movement is started by the Reader control request to release the Reader brake and simultaneously engage the clutch.

In addition to the Reader movement control logic, the control unit contains an 18-bit Reader Buffer (RB) which can collect one or three lines from the tape. The C(RB) can be read into the AC. The Reader flag becomes a 1 when a character or word has been assembled in the RB.

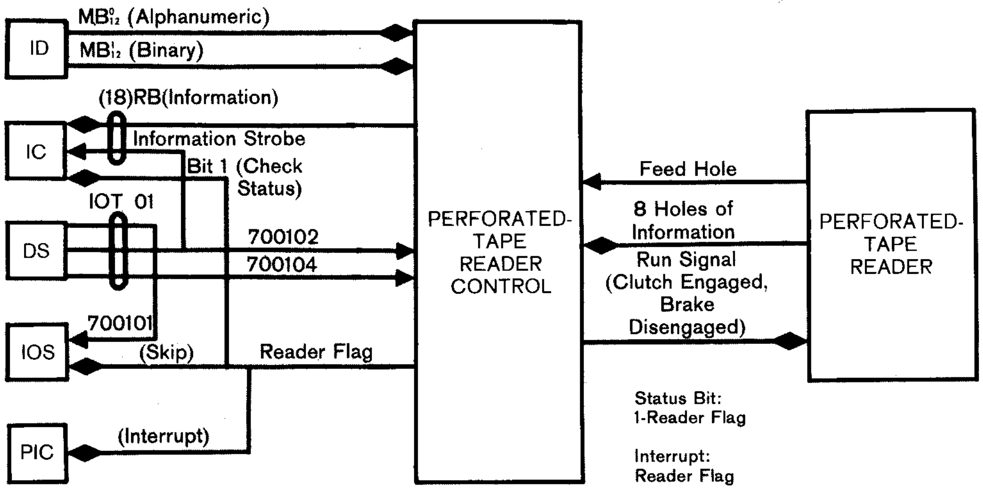


Figure 15 — Perforated —Tape Reader programming logic

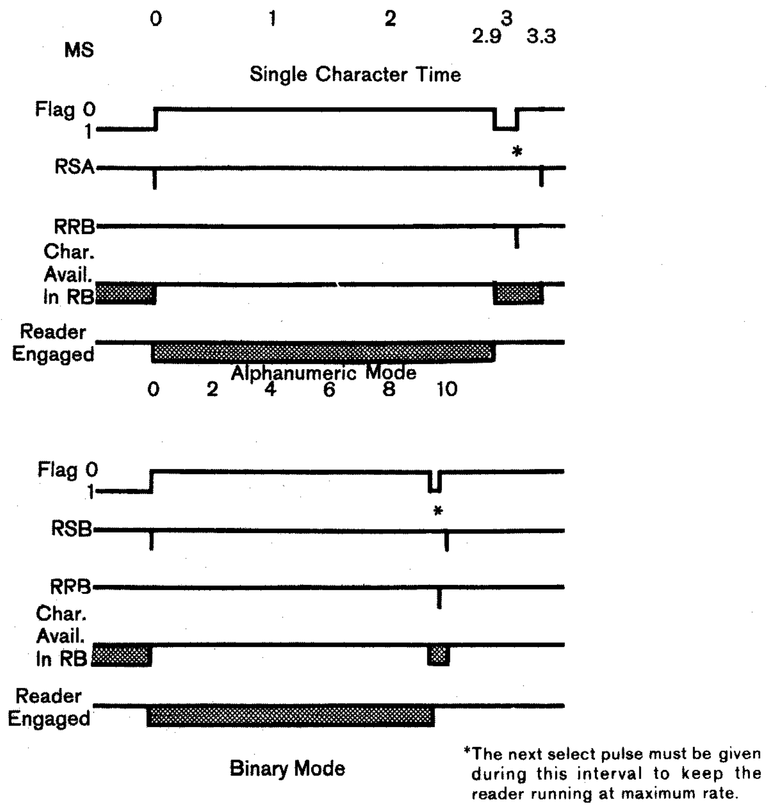


Figure 16 — Perforated-Tape Reader timing

An alphanumeric character is one line (5, 7, or 8 holes) on tape. A binary word consists of three consecutive characters (18 bits) on tape which have the 8th hole present. Only 8-hole tape is used in the binary mode; the 7th hole is ignored. The first, second, and third six-bit characters are the left, middle, and right thirds, respectively, of the 18-bit word. The reader commands, iot select series 01, are:

rsf — 700101 —	Skip if Reader flag is a 1, i.e., character or word present.
rsa — 700104 —	Select Reader and fetch one alphanumeric character from tape. Clear the Reader flag. Reset RB. The character is read into RB bits 10-17. Turn on the Reader flag when character is present.
rsb — 700144 —	Select Reader and fetch a binary word from tape. Clear the Reader flag. Reset the RB. Fetch the next three characters (with 8th holes present) from perforated tape and place in RB bits 0-5, 6-11, and 12-17. Turn on Reader flag when a word is assembled.
rrb — 700112 —	Read RB. Clear the Reader flag, and transfer the contents of RB to the AC.
rcf — 700102 —	Clear the Reader flag. C(RB) $\vee$ C(AC) => C(AC)

The Reader flag is connected to the Program Interrupt Control and to bit 0 of the iors instruction. Several methods may be used to program the Reader. The following sequence reads a character from tape and places it in the AC. Up to 400 microseconds of computation time are available between the end of the sequence and the next command to read a character or word from tape. The sequence, starting in register 40 is:

PROGRAM SEQUENCE

```
/perforated-tape reader
40/         rsa         /select reader alphanumeric
            rsf         /begin loop to look for character arrival
            jmp 41      /end loop to look for arrival
            rrb         /fetch character from reader buffer
```

By changing instruction 40 to rsb the sequence would fetch a binary word.

PRINTER-KEYBOARD AND CONTROL, TYPE 65

The Printer-Keyboard is a Teletype Model 28 KSR (keyboard send-receive) which can print or receive ten characters per second. A five-bit code, given in Appendix 2, represents the characters. The printing (output) and keyboard (input) functions have separate commands and control logic.

The signals to and from the KSR to the control logic are standard serial, 7.5-unit-code Teletype signals. The signals are: start (1.0 unit), information, bits 1-5 (1.0 unit each), and stop (1.5 units). Figure 17 illustrates the current pattern produced by the binary code 10110.

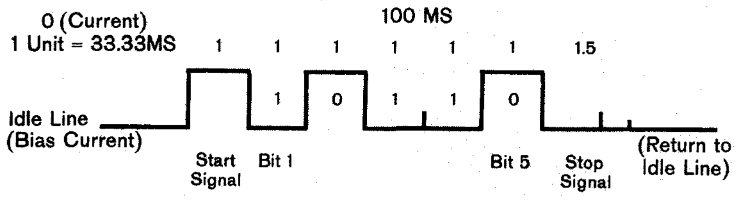


Figure 17 — Teletype timing of information code 10110

KEYBOARD

The Keyboard control contains a 5-bit buffer (KB) which holds the code for the last key struck. The Keyboard flag signifies that a character has been typed and its code is present in the Keyboard buffer. The Keyboard flag and Keyboard buffer are cleared each time a character starts to appear on the Teletype line. The Keyboard flag becomes a 1, signifying the buffer is full 0.5 ± 0.125 units after the end of information bit 5, or 86.6 milliseconds after key strike time. The instructions to manipulate the Keyboard are:

ksf — 700301 — Skip if the Keyboard flag is a 1, i.e., character present.

krb — 700312 — Read Keyboard buffer. Clear the Keyboard flag. C(KB)=> C(AC)

700302 — Clear the Keyboard flag. C(KB) ∨ C(AC) => C(AC)

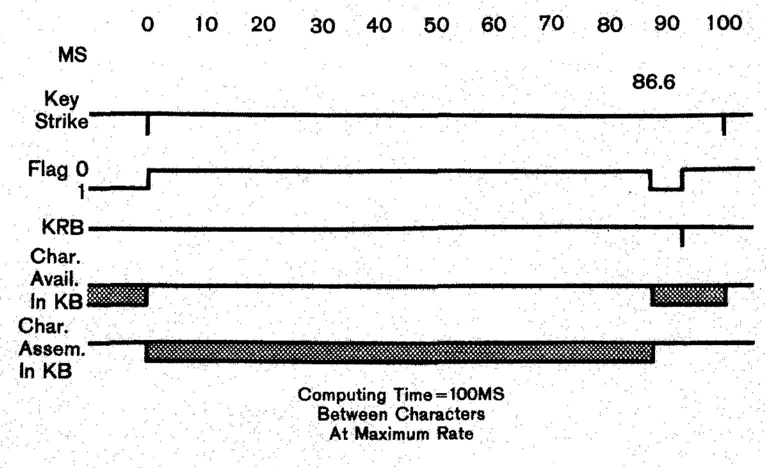


Figure 18 — Keyboard timing

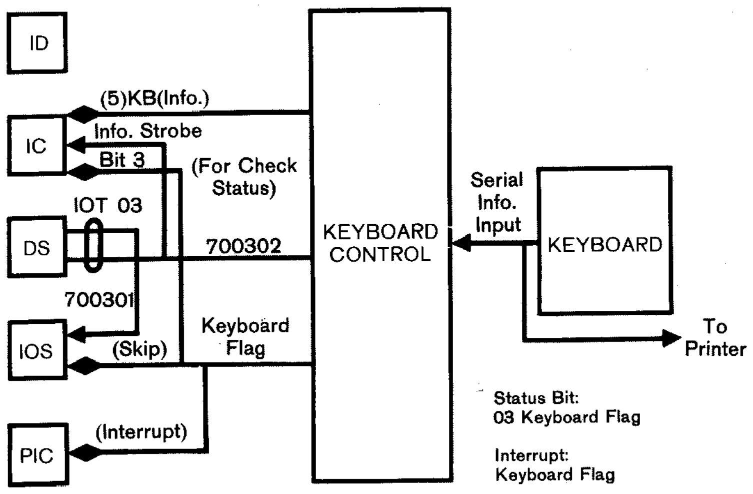


Figure 19 — Keyboard programming logic

The Keyboard flag is connected to the Program Interrupt Control and the iors instruction, bit 3. A simple sequence which "listens" for keyboard inputs is:

PROGRAM SEQUENCE

```
/listen loop for keyboard

400/        ksf         /skip when a character arrives from keyboard

            jmp 400

            krb         /read in the character
```

The sequence following the listen sequence beginning in 403 may operate for up to 100 + 13.3 milliseconds before returning to listen for the next character without missing the next character. The average computing time between any two characters must be less than 100 milliseconds (for an input rate of 10 characters per second).

TELEPRINTER

The Teleprinter is given 5 bits of information from AC bits 13 to 17, coding the character to be printed. The teleprinter Buffer (TB) receives this information, transmits it to the Teleprinter serially, and when finished turns on the Teleprinter flag. The Flag is connected to the Program Interrupt and to bit 4 of the iors instruction. The printing rate is ten characters per second. The instructions for the printer are:

tsf — 700401 — Skip if Teleprinter flag is a 1.

tls — 700406 — Load the Teleprinter from AC bits 13-17, clear the Teleprinter flag. Select the Teleprinter for printing.

tcf — 700402 — Clear the Teleprinter flag.

700404 — C(AC) ∨ C(TB). Print a character.

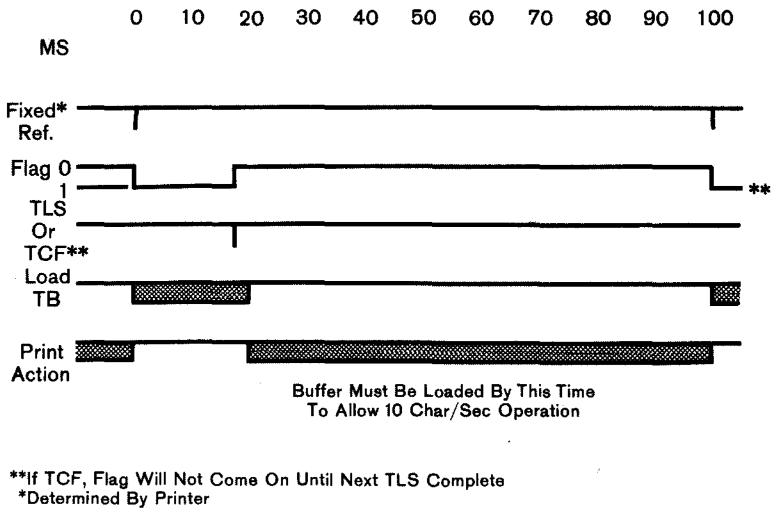

0 10 20 30 40 50 60 70 80 90 100
MS
Fixed* Ref.
Flag 0
1
**
TLS Or TCF**
Load TB
Print Action
Buffer Must Be Loaded By This Time To Allow 10 Char/Sec Operation
**If TCF, Flag Will Not Come On Until Next TLS Complete
*Determined By Printer

Figure 20 — Printer timing

PROGRAM SEQUENCES

```
/print and wait for Teleprinter
           tls       /print the character from AC bits 13-17
           tsf       /begin listen loop for printing completion
           jmp.-1    /return to previous instruction or listen loop
                     /again
             .
             .
```

```
/wait for previously printed character completion, then print
          tsf       /wait loop until previous character printed
          jmp.-1    /return to wait loop beginning
          tls       /print the new character
            .
            .
```

In the first sequence above, 20 milliseconds of program time is available between that tls and the next one that can be given. In the second sequence, 100 milliseconds of program time is available between that tls and the next one that can be given.

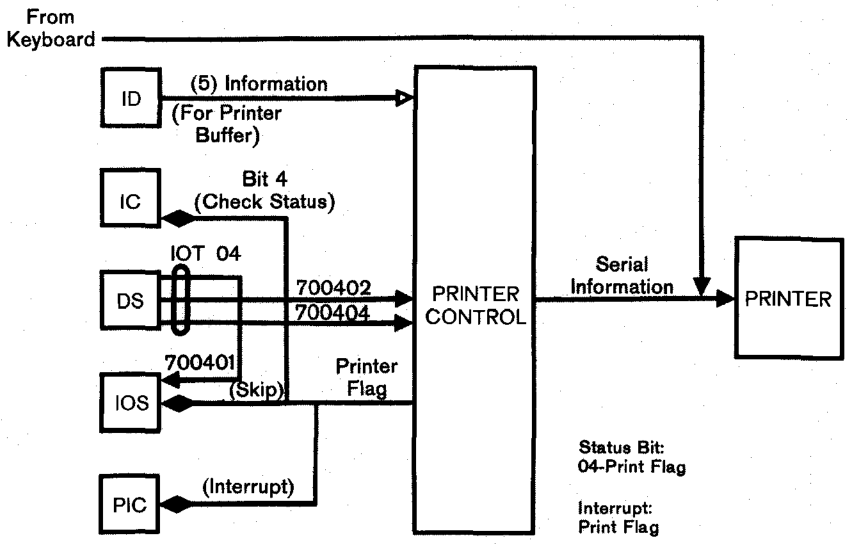


Figure 21 — Printer programming logic

PERFORATED-TAPE PUNCH AND CONTROL, TYPE 75

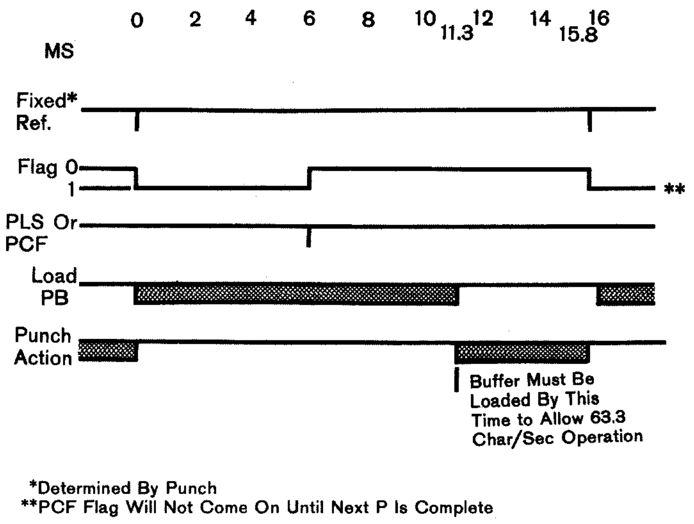
The Teletype BRPE paper-tape punch perforates 5-, 7-, or 8-hole tape at 63.3 characters (lines) per second. Information to be punched on a line of tape is loaded on an 8-bit buffer (PB) from the AC bits 10 through 17. The Punch flag becomes a 1 at the completion of punching action, signaling that new information may be read into Punch Buffer (PB) (and punching initiated). The Punch flag is connected to the Program Interrupt and to the iors instruction bit 2. The Punch instructions, iot series 02, are:

psf — 700201 — Skip if the Punch flag is a 1.

pcf — 700202 — Clear the Punch flag.

pls — 700206 — Load a character into PB from AC bits 10-17. Clear the Punch flag. Punch the specified character.

700204 — C(PB) V C(AC)=> C(PB). Punch the C(PB).



MS
0
2
4
6
8
10
11.3
12
14
15.8
16
Fixed* Ref.
Flag 0
1
**
PLS Or PCF
Load PB
Punch Action
Buffer Must Be Loaded By This Time to Allow 63.3 Char/Sec Operation
*Determined By Punch
**PCF Flag Will Not Come On Until Next P Is Complete

Figure 22 — Perforated-Tape Punch timing

PROGRAM SEQUENCES

```
/punch the contents of AC and wait
            pls         /punches AC 10-17
            psf         /wait till done loop beginning
            jmp.-1      /wait till done loop end
/wait for previous punching, then punch next
            psf         /wait loop for previous character punching
            jmp.-1      /wait loop end
            pls         /punch the next character on tape
```

In the first sequence above, 11.3 milliseconds of program time is available between the instruction following the wait loop and the next pls that can be given. In the second sequence, 15.8 milliseconds or more program time is available between the pls and the next time a pls can be given.

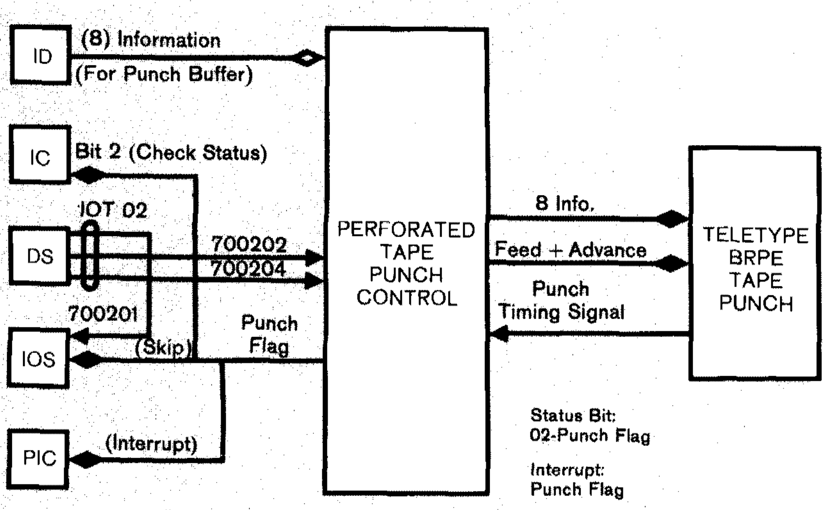


Figure 23 — Perforated-Tape Punch programming logic

CARD READER AND CONTROL, TYPE 41-4

The control of the Card Reader is different than the control of other input devices, in that the timing of the read-in sequence is dictated by the device. Once the command to fetch a card is given, the Reader will read all 80 columns of information in order. To read a column, the program must respond to a flag set as each new column is started. The instruction to read the column must come within 300 microseconds after the flag is set. The interval between flags is 2.3 milliseconds. The commands for the Card Reader, iot series 67, are:

crsf — 706701 — Skip if Card Reader flag is a 1. If a card column is present for reading, the instruction will skip.

crrb — 706712 — Read the card column buffer information into AC and clear the Card Reader flag. One crrb reads alphanumeric information. Two crrb instructions read the upper and lower column binary information.

crsa — 706704 — Select a card in alphanumeric mode. Select the card reader and start a card moving. Information will appear in alphanumeric form.

crsb — 706714 — Select a card in binary mode. Select the card reader and start a card moving. Information will appear in binary form.

Upon instruction to read the Card Reader buffer, 6 information bits are placed into AC bits 12-17. Alphanumeric (or Hollerith) information on the card is encoded or represented with these six bits. The binary mode enables the 12 bits (or rows) of each column to be obtained. The first read buffer instruction transfers the upper six rows (Y, X, 0, 1, 2, and 3), the second instruction transfers the lower six rows (4, 5, 6, 7, 8, and 9). The mode is specified with the Card Read Select instruction. The mode can be changed while the card is being read.

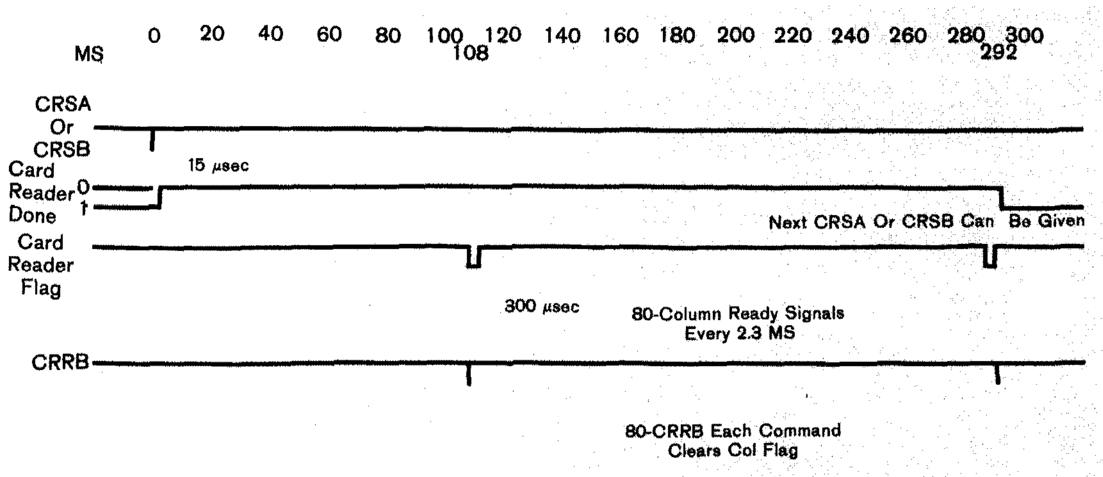


Figure 24 — Card Reader timing

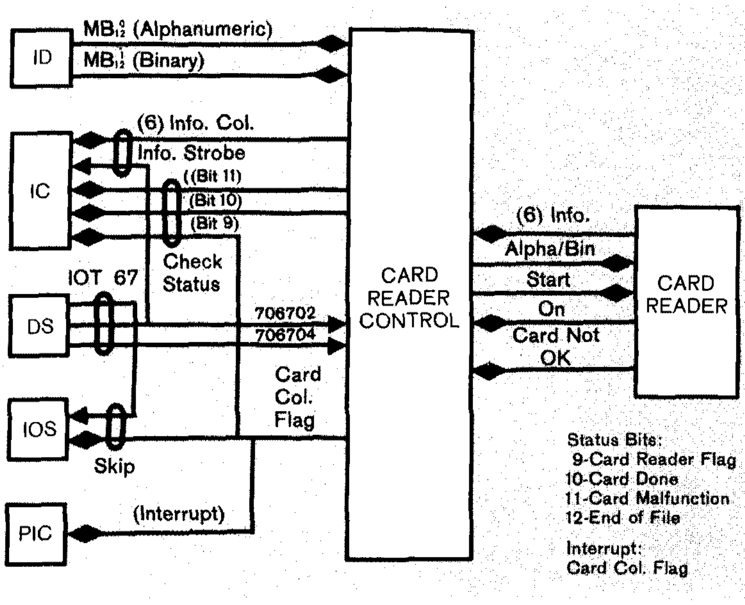


Figure 25 — Card Reader programming logic

The Card Read Flag is connected to the Program Interrupt Control and to bit 9 of the iors instruction. The Card Read Done status level bit is connected to bit 10 of the iors instruction. A Card Read Malfunction status is connected to bit 11 of the iors instruction. Card Read Malfunction status indicates one or more of the following conditions: Reader not ready (power off, etc.), hopper empty, stacker full, card jam, validity check error (if validity is on), or real circuit failure.

Bit 12 of the iors instruction is connected to the END OF FILE switch at the Card Reader. The switch is activated manually, and when depressed, holds until the RESET END OF FILE switch is depressed.

PROGRAM SEQUENCE

```
/sequence to read an 80-column card and place alphanumeric codes
/in register 1000-1117 (octal).  Program begins in register cardrd.
cardrd,     crsa          /read card in alphanumeric mode
            lac cardlo    /initialize card location table
            dac 10        /place in indexable register
            lac cardct    /initialize card count 80 (decimal)
            dac temp
cdloop,     crsf          /wait for column loop
            jmp cdloop
            crrb          /place column information in AC
            dac i 10      /info to 1000, 1001 ...1117
            isz temp
            jmp cdloop
            hlt           /finish of card, and halt
cardlo,     1000-1        /location of card table
cardct,     -120+1        /80 column counter initial value
temp,       0             /reserved for column counter
```

CARD PUNCH CONTROL, TYPE 40-4

The Card Punch dictates the timing of a read-out sequence, much as the Card Reader controls the read-in timing. Once a card has started, all 12 rows are punched at intervals of 40 milliseconds. Punching time for each row is 24 milliseconds, leaving 16 milliseconds to load the buffer for the next row. A flag indicates that the buffer is ready to load. The commands for the Card Punch Control, iot series 64, are:

cpsf — 706401 —	Skip if Card Punch flag is a 1. The Card Punch flag indicates the Punch buffer is available, and should be loaded.
cpcf — 706402 —	Clear Card Punch flag.
cpse — 706442 —	Select the Card Punch. Transmit a card to the 80-column punch die from the hopper.
cplb — 706406 —	Load the Card Punch buffer from the C(AC). Five load instructions must be given to fill the buffer.

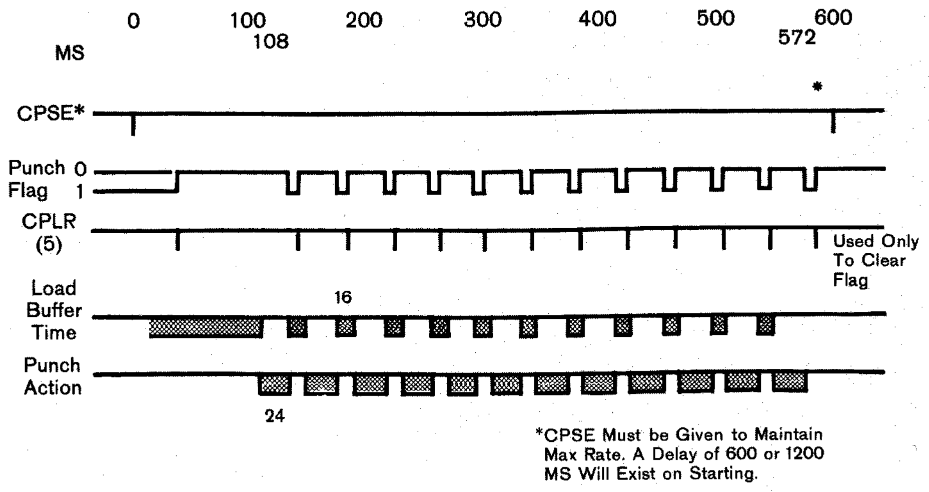


Figure 26 — Card Punch timing

Since 18 bits are transmitted with each iot instruction, 5 iot instructions must be issued to load the 80-bit row buffer. The first four loading instruction fill the first 72 bits (or columns); the fifth loads the remaining 8 bits of the buffer from AC bits 10-17.

After the last row punching is complete, 28 milliseconds are available to select the next card for continuous punching. If the next card is not requested in this interval, the Card Punch will stop. The maximum rate of the Punch is 100 cards per minute in continuous operation. A delay of 1308 milliseconds follows the command to select the first card; a delay of 108 milliseconds separates the reading of cards in continuous operation.

The Card Punch flag is connected to the Program Interrupt, and to bit 13 of the iors instruction. Faults occurring in the punch are detected by status bit 14 of the iors and signify the punch is disabled, the stacker is full, or the hopper is empty.

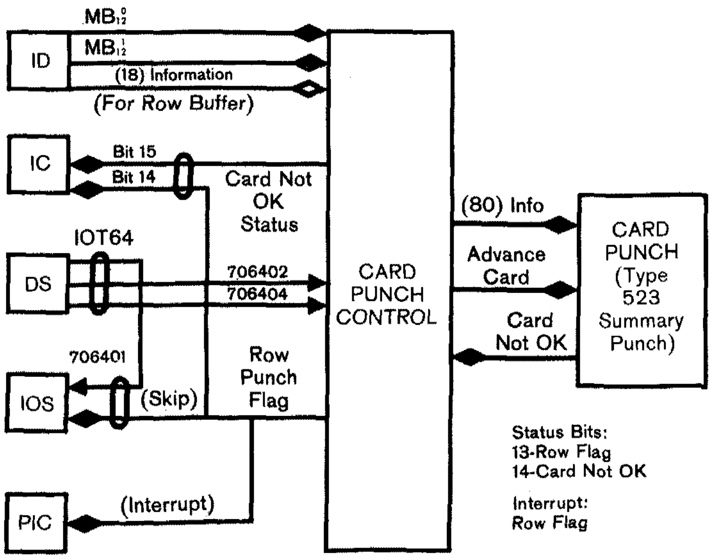


Figure 27 — Card Punch programming logic

PROGRAM SEQUENCE

```
/sequence to punch 12 rows of data on a card.  Each row is stored in
/5 consecutive registers beginning in location 100.  The program begins
/in register cardph.
cardph,         cpse            /select the card
                lac punloc      /initialize the card image
                dac 10
                lac rowct
                dac temp1       /initialize the row counts, 12.
/loop1,         lac grpct       /initialize the 5 groups per row
                dac temp2
                cpsf            /sense punch load availability
                jmp.-1
loop2,          lac i 10        /5 groups of 18 bit per row
                cplr            /load buffer command
```

```
                  isz temp2
                  jmp loop2
                  isz temp1           /test for 12 rows
                  jmp loop1
                  hlt                 /end punching 1 card
punloc,           100-1               /location of card image
rowct,            -14+1               /12 rows per card
grpct,            -5+1                /5 groups per row
temp1,            0                   /row counter
temp2,            0                   /group counter
```

AUTOMATIC LINE PRINTER AND CONTROL, TYPE 62

The Line Printer can print 600 lines of 120 columns per minute. Each column has 64 characters. Spacing rate is approximately 132 lines (or two 66-line pages) per second.

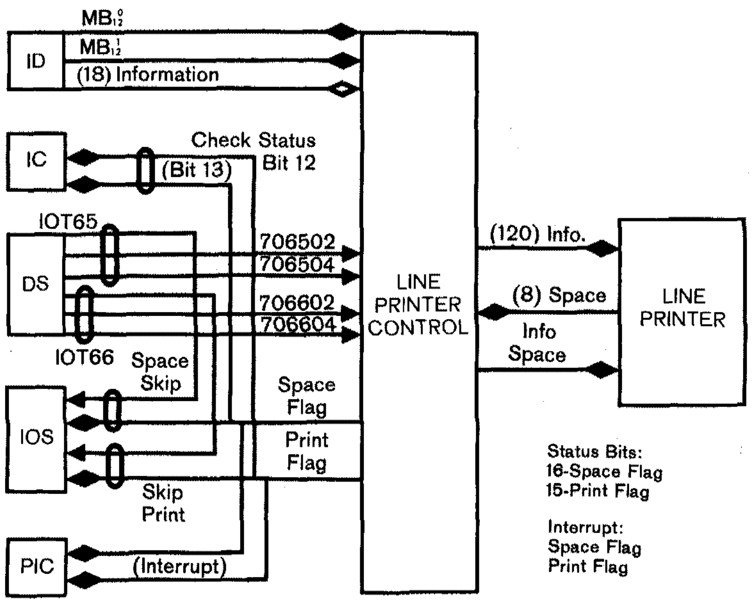


Figure 28 — Line Printer programming logic

A complete line, or 120 columns of information, is placed in the printing buffer. Six bits specify each character (the codes are given in Appendix 2). The information is transferred to the printing buffer through the AC, three characters at a time from AC bits 0-5, 6-11, and 12-17. Forty load print buffer instructions fill the 120-column line.

After the printing buffer is loaded, a print instruction is given which prints the contents of the buffer. The action of printing does not disturb the printing buffer. When a column of information has been printed, the printing flag becomes a 1. Approximately 80 milliseconds are required to print one line.

An eight-channel format-control tape inside the Printer moves in synchronism with the paper and specifies how far the paper is to be spaced. Holes punched in each channel of the format tape signify the next paper position. The channel is selected by placing a three-bit code in AC bits 15-17, and giving an instruction to space paper. The spacing flag becomes a 1 when the spacing action is complete. A recommended control tape has the following characteristics, where the middle column indicates the number of lines between successive holes in the channel:

Channel	Spacing	Time
0	1 line	16 ms
1	2 lines	$<2 \times 16$ ms
2	3 lines	$<3 \times 16$ ms
3	6 lines	$<6 \times 16$ ms
4	11 lines (1/6 page)	$<11 \times 16$ ms
5	22 lines (1/3 page)	$<22 \times 16$ ms
6	33 lines (1/2 page)	$<33 \times 16$ ms
7	restores page	520 ms for 66 lines

The Line Printer printing and spacing instructions, iot series 65 and 66, are:

lpsf — 706501 — Skip if the printing flag is a 1.

lpcf — 706502 — Clear the printing flag.

lpld — 706542 — Load the Printing buffer.

lpse — 706506 — Select the Printer. Print the contents of the Printing buffer. Clear the printing flag. (The printing flag becomes a 1 at the completion of the printing.)

lssf — 706506 — Skip when the spacing flag becomes a 1.

lscf — 706602 — Clear the spacing flag.

lsls — 706606 — Load the spacing buffer from AC bits 15-17 and select spacing. Clear the spacing flag. (The spacing flag becomes a 1 when spacing is complete.)

The printing and spacing flags are connected to the Program Interrupt and to the iors instruction bits 15 and 16.

PROGRAM SEQUENCE

```
/sequence to print a line of 120 columns.  Output stored 3
/characters per word.
/Data begins in register 2000. Sequence assumes printer is
/in process of printing a line previously assigned.  "print" is
/begin of prog.
print,        lpsf          /wait till previous printing done
              jmp.-1
              lsls + 10     /space 1 line (0 in AC)iot 10 clears
                            /AC
              lac (2000-1   /location of data
              dac 10        /print table initialize
              lac(-50+1     /40x3 characters
              dac temp
ldloop,       lac i 10      /load print buffer loop
              lpld          /load from AC
              isz temp
              jmp ldloop
space,        lssf          /test for spacing done before
                            /proceeding
              jmp space
              lpse          /print activate...end of printing
                            /a line
```

CHAPTER 4

THE INTERFACE ELECTRICAL CHARACTERISTICS

As explained in previous sections, the standard Interface contains the Real-Time Connection, which can operate only with the Perforated-Tape Reader, the Perforated-Tape Punch, and the Printer-Keyboard. The Real-Time Option can operate with a variety of external devices over a wide range of information handling rates. In this section the location of the Real-Time Option, its electrical characteristics, and its connections to input-output devices are presented.

Real-Time Option

A coordinate system locates modules and connectors in PDP-4 with a four-place, alphanumeric code. Bays are numbered 1 and 2, panels are lettered alphabetically downward, connectors or modules are numbered left to right in the panels (blank spaces included), and terminals are lettered alphabetically downward on the connectors or modules. The Real-Time Option is located in panels 2E, 2F, and 2H. Connections to external control units are made through a cable connector in positions 2J1-6.

DEVICE SELECTOR (LOCATION 2F6-25)

The standard Device Selector contains provisions for up to 20 selector modules, each of which is a Pulse Amplifier, Type 4605. The amplifiers are pulsed with standard DEC 4000 Series negative logic pulses which can drive 18 units of base load.

Each module is wired to respond to one address code only (see example, Figure 29). The 6-bit address portion of the iot instruction will therefore pass only through the six-level AND gate of those modules wired to the same combination of ones and zeros. The output of the AND gate enables three AND gates to pass the common iot 1, 2, and 3 pulses. These pulses are available at terminals E, H, and K, respectively, of modules 2F6-25.

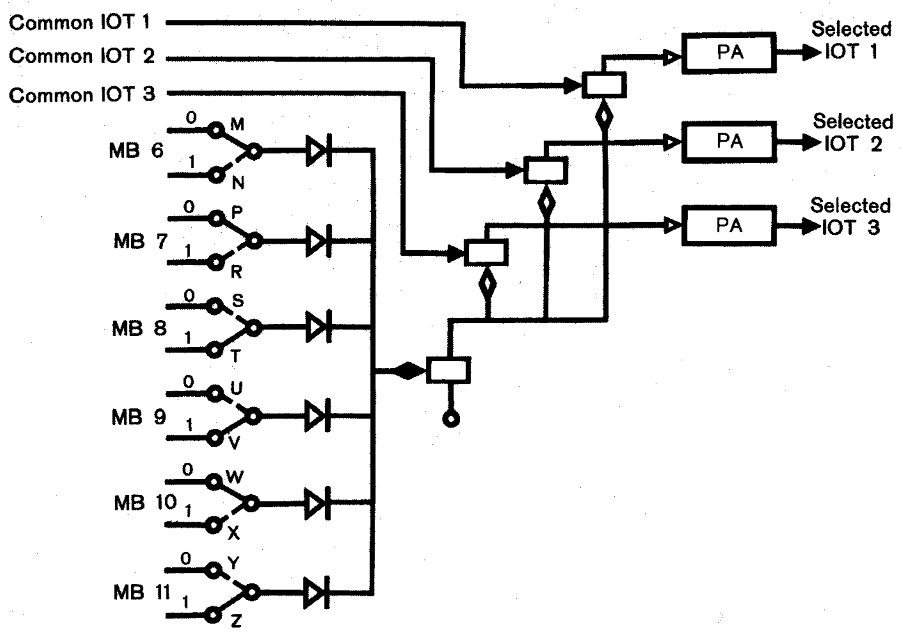


Figure 29-Typical Pulse Amplifier, Type 4605, used in PDP-4 Device Selector. Example shown is wired to pass the iot address 001101. The six-level AND gate will pass only that address if it is present in the instruction word from the Memory Buffer, thus enabling three AND gates to pass three IO pulses to the pulse amplifier.

The Device Selector modules are delivered with jumpers across the address terminals. The user can remove appropriate jumpers to establish the module select mode according to the table below.

Instruction Word Bit	ZERO Input Terminal	ONE Input Terminal
6	M	N
7	P	R
8	S	T
9	U	V
10	W	X
11	Y	Z

INFORMATION COLLECTOR (LOCATION 2H8-25)

The information collecting sequence begins with an iot pulse from the Device Selector applied to the strobe input of the Information Collector. The IC then ANDs with the input device information present level and the results are transmitted to the AC. The results of the AND functions are mixed, or ORed together, to enable eight 18-bit-word devices to read data into the AC. Two or more devices requiring less than 18 bits could share a word, provided their bit-position requirements did not conflict. In such cases, more than eight input devices could be handled by the IC. The incoming information signal polarities are:

0 volts	0 bit transmitted to AC
−3 volts	1 bit transmitted to AC

The IC consists of 18 modules, one for each bit of the word, starting with bit 0 in module 2H8. All eight input channels are wired to each module. The convention for designating bits is $IC_{j,k}$, where j specifies the bit number and k the channel number. The eight input-level terminals and associated iot-pulse terminals are:

Channel (k)	Data-Bit Input	Associated iot Input
0	E	F
1	H	J
2	K	L
3	M	N
4	S	T
5	U	V
6	W	X
7	Y	Z

INFORMATION DISTRIBUTOR (LOCATION 2H1-3)

The Information Distributor presents the static data contained in the AC to an output device when the Device Selector commands the device to sample the ID. The signal polarities are:

−3 volts	AC bit contains a 0
0 volts	AC bit contains a 1

Eight groups of 18 outputs are available in the ID. The module driving the output bus is a Type 1690 or 1685 Bus Driver supplying up to 15 ma at 0 or —3 volts. All eight groups must share the bus.

Connections to the ID are made at three taper-pin terminal blocks, 2H1, 2H2, 2H3. Each block has 3 columns of 20 terminals each. Each column represents a group; the first 18 terminals (A-U) in the column represent AC bits 0-17 and the last two (V, W) the bipolar bit 12 in the Memory Buffer. V and W may be used to select a subdevice. The terminals are tied together horizontally to form 20 rows.

INPUT-OUTPUT SKIP FACILITY (LOCATION 2H06)

There are 8 inputs to Input-Output Skip. The iot pulses from the Device Selector strobe an input line and if a logic condition is present, the instruction following the iot will be skipped. The conditions for skipping are:

−3 volts	skip
0 volts	do not skip

The iot skip pulse <u>must</u> occur at event time 1 of the iot instruction.

The IOS consists of a Capacitor-Diode Gate, Type 4129. The input connections are:

IO Device Input Connection	Device Selector Pulse Connection
F	E
J	H
L	K
N	M
T	S
V	U
X	W
Z	Y

PROGRAM INTERRUPT CONTROL (LOCATION 2H05)

Eleven Program Interrupt lines are available. Any one of the 11 signals may cause an interruption of a program. All signals are identical; the polarities are:

−3 volts	interrupt the program
0 volts	no effect

The connections from IO devices which request program interrupt are made to module 2H05 at pins E, F, H, J, S, T, U, W, X, Y, and Z.

DATA INTERRUPT CONTROL (LOCATION 2E13)

The signal levels associated with the DI are shown in Figure 30. In transferring data, the Memory Address is first transmitted to the Memory Address Register on 13 lines from the external source. Data is next transferred to or from the MB on 18 + 18 lines.

Incoming data is received from 18 lines and placed in the Memory Buffer and on into Memory.

Outgoing data from the Core Memory addressed is transferred to the Memory Buffer and appears on 18 lines for sampling by the IO device.

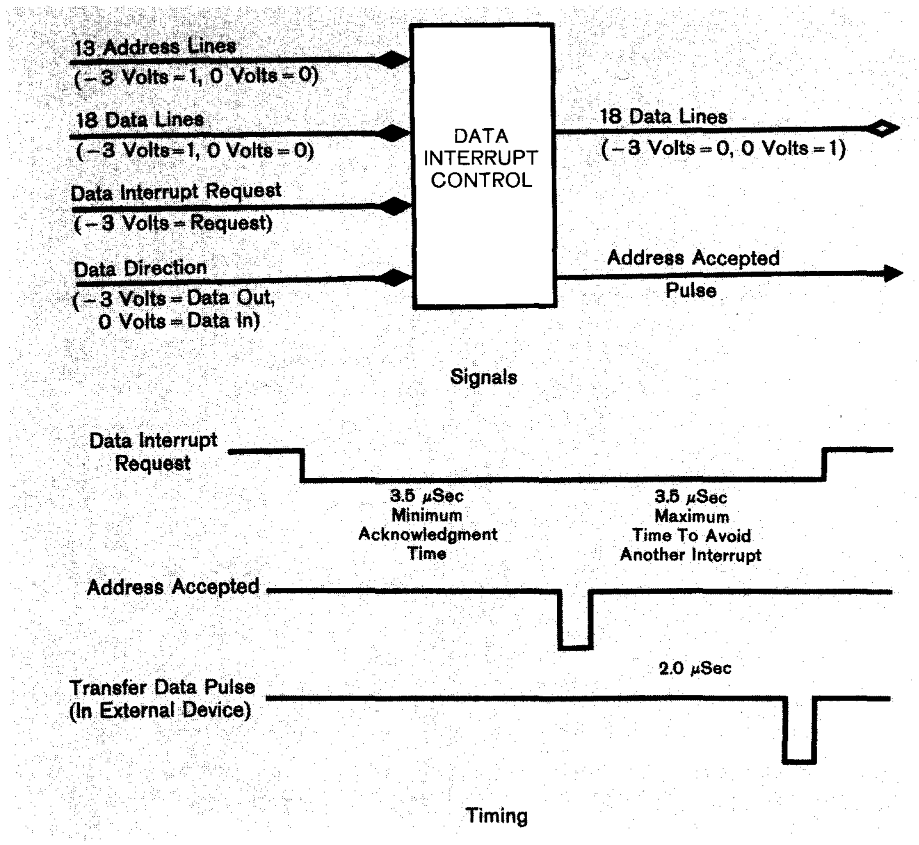

13 Address Lines
(−3 Volts = 1, 0 Volts = 0)
18 Data Lines
(−3 Volts = 1, 0 Volts = 0)
Data Interrupt Request
(−3 Volts = Request)
Data Direction
(−3 Volts = Data Out, 0 Volts = Data In)
DATA INTERRUPT CONTROL
18 Data Lines
(−3 Volts = 0, 0 Volts = 1)
Address Accepted Pulse
Signals
Data Interrupt Request
3.5 μSec Minimum Acknowledgment Time
3.5 μSec Maximum Time To Avoid Another Interrupt
Address Accepted
2.0 μSec
Transfer Data Pulse (In External Device)
Timing

Figure 30 — Data Interrupt Control signals and timing

APPENDIX 1

Instruction Lists

MEMORY REFERENCE INSTRUCTIONS

MNEMONIC CODE	OCTAL CODE	TIME (μsec)	OPERATION
cal Y	00	16	Call Subroutine. Y is ignored jms 20 if bit 4 = 0, jms i 20 if bit 4 = 1.
dac Y	04	16	Deposit AC. C(AC) => C(Y)
jms Y	10	16	Jump to subroutine. C(PC) => C(Y_{5-17}), C(L) => C(Y_0), Y + 1 => C(PC)
dzm Y	14	16	Deposit zero in memory. 0 => C(Y)
lac Y	20	16	Load AC. C(Y) => C(AC)
xor Y	24	16	Exclusive OR. C(AC) $\vee$ C(Y) => C(AC)
add Y	30	16	Add (1's complement). C(AC) + C(Y) => C(AC)
tad Y	34	16	2's complement add. C(AC) + C(Y) => C(AC)
xct Y	40	8+	Execute.
isz Y	44	16	Index and skip if 0. C(Y) + 1 => C(Y), if C(Y) + 1 = 0, then C(PC) + 1 => C(PC)
and Y	50	16	AND. C(AC) $\wedge$ C(Y) => C(AC)
sad Y	54	16	Skip if AC and Y differ. If C(AC) = C(Y), then C(PC) + 1 => C(PC)
jmp Y	60	8	Jump. Y => C(PC)
law N	76	8	Load AC with law N. 1 => C(AC_{0-4}), N => C(AC_{5-17})

OPERATE INSTRUCTIONS

MNEMONIC CODE	OCTAL CODE	EVENT TIME	OPERATION
opr	740000	—	Operate.
nop	740000	—	No Operation.
cma	740001	3	Complement, $\overline{C(AC)} => C(AC)$
cml	740002	3	Complement Link, $\overline{C(L)} => C(L)$
oas	740004	3	Inclusive OR AC Switches. $C(ACS) \vee C(AC) => C(AC)$
las	750004	2, 3	Load AC from Switches. $C(ACS) => C(AC)$
ral	740010	3	Rotate AC + Link left one place. $C(AC_j) => C(AC_{j-1})$, $C(L) => C(AC_{17})$, $C(AC_0) => C(L)$
rcl	744010	2, 3	Clear Link, then ral. 0 => C(L), then ral
rtl	742010	2, 3	Rotate AC left twice. Same as two ral instructions
rar	740020	2	Rotate AC + Link right one place. $C(AC_j) => C(AC_{j+1})$, $C(L) => C(AC_0)$, $C(AC_{17}) => C(L)$
rcr	744020	2, 3	Clear Link, then rar. 0 => C(L), then rar
rtr	742020	2, 3	Rotate AC right twice. Same as two rar instructions
hlt	740040	4	Halt. 0 => RUN
sza	740200	1	Skip on zero AC. Skip if C(AC) = positive zero
sna	741200	1	Skip on non-zero AC. Skip if C(AC) $\neq$ positive zero
spa	741100	1	Skip on positive AC. Skip if $C(AC_0) = 0$
sma	740100	1	Skip on negative AC. Skip if $C(AC_0) = 1$
szl	741400	1	Skip on zero Link. Skip if C(L) = 0
snl	740400	1	Skip on non-zero Link. Skip if C(L) = 1
skp	741000	1	Skip, unconditional. Always skip
cll	744000	2	Clear Link. 0 => C(L)
stl	744002	2, 3	Set the Link. 1 => L
cla	750000	2	Clear AC. 0 => C(AC)
clc	750001	2, 3	Clear and Complement AC. −0 => C(AC)
glk	750020	2, 3	Get Link. 0 => C(AC), $C(L) => C(AC_{17})$

BASIC IOT INSTRUCTIONS

MNEMONIC CODE	OCTAL CODE	OPERATION
		Interrupt
iof	700002	turn off interrupt
ion	700042	turn on interrupt
		IO Equipment
iors	700314	read status of io equipment
		Clock
clsf	700001	skip if clock flag is 1
clof	700004	turn off clock, clear clock flag
clon	700044	turn on clock, clear clock flag
		Paper tape reader
rsf	700101	skip if reader flag is a 1
rsa	700104	select reader for alphanumeric, clear reader flag
rsb	700144	select reader for bry, clear reader flag
rrb	700112	read the reader buffer into AC, clear reader flag
		Paper tape punch
psf	700201	skip if punch flag is a 1
pls	700206	load punch buffer and select punch, clear punch flag
pcf	700202	clear punch flag
		Keyboard input from teleprinter
ksf	700301	skip if keyboard flag is a 1
krb	700312	read the beyboard buffer into the AC, clear keyboard flag
		Teleprinter
tsf	700401	skip if teleprinter flag is a 1
tls	700406	load teleprinter buffer and select, clear teleprinter flag
tcf	700402	clear the teleprinter flag
		Display type 30A
dsf	700501	skip if display flag is a 1
dls	700506	load display buffer and select, clear display flag
dcf	700502	clear display flag
		Display type 30D
dsf	700501	skip if display flag is a 1 (light pen)
dcf	700601	clear display flag
dxl	700506	load x co-ordinate
dxs	700546	load x co-ordinate and select
dyl	700606	load y co-ordinate
dys	700646	load y co-ordinate and select
dlb	700706	load brightness register

BASIC IOT INSTRUCTIONS

(continued)

MNEMONIC CODE	OCTAL CODE	OPERATION
		Magnetic tape type 54
mci	707001	clear tape instruction and character buffer
mrs	707012	read tape status into AC
mli	707005	load instruction buffer
msc	707101	skip if character is present for reading
msi	707201	clear interrupt flag and select interrupt
msf	707301	skip if the tape flag is a 1 (end of record)
mrl	707112	clear AC, read character buffer into AC left clear character buffer
mrm	707202	read character buffer into AC middle clear character buffer
mrr	707302	read character buffer into AC right clear character buffer
mwl	707104	write a character from AC left
mwm	707204	write a character from AC middle
mwr	707304	write a character from AC right
		Card reader
crsf	706701	skip if reader character flag is a 1
crsa	706704	select card reader for alphanumeric
crsb	706744	select card reader for binary
crrb	706712	read card column buffer into AC
		Card punch
cpsf	706401	skip if the card punch flag is a 1
cpse	706444	select a card, set card punch flag
cplr	706406	load row buffer, clear punch flag
cpcf	706442	clear punch flag
		Line printer
lpsf	706501	skip if printing flag is a 1
lpcf	706502	clear printing flag
lpld	706542	load the printing buffer
lpse	706506	select printing, clear printing flag
lssf	706601	skip if spacing flag is a 1
lscf	706602	clear spacing flag
lsls	706606	load spacing buffer and select spacing, clear spacing flag

APPENDIX 2

Codes

FIO-DEC CODE

a	A	61
b	B	62
c	C	63
d	D	64
e	E	65
f	F	66
g	G	67
h	H	70
i	I	71
k	J	41
k	K	42
l	L	43
m	M	44
n	N	45
o	O	46
p	P	47
q	Q	50
r	R	51
s	S	22
t	T	23
u	U	24
v	V	25
w	W	26
x	X	27
y	Y	30
z	Z	31
0	→	20
1	″	01
2	′	02
3	∽	03
4	⊃	04
5	∨	05
6	∧	06
7	<	07
8	>	10
9	↑	11

Low order bits	High order bits 00	01	10	11
0000	space	0 →	. _	
0001	1 ″	/ ?	j J	a A
0010	2 ′	s S	k K	b B
0011	3 ~	t T	l L	c C
0100	4 ⊃	u U	m M	d D
0101	5 ∨	v V	n N	e E
0110	6 ∧	w W	o O	f F
0111	7 <	x X	p P	g G
1000	8 >	y Y	q Q	h H
1001	9 ↑	z Z	r R	i I
1010				lower case
1011	stop	, =		. ×
1100		black	— +	upper case
1101		red	)]	backspace
1110		tab	‾ |	
1111			([	car ret

/	?	21
,	=	33
.	×	73
—	+	54
)	]	55
(	[	57
·	_	40
‾	|	56

stop code	13
lower case	72
upper case	74
black	34
red	35
tab	36
backspace	75
carriage return	77
space	00

code delete punches seventh channel

TELETYPE CODE

Low order bits	High order bits 00	01	10	11
000		line feed	E 3	A —
001	T 5	L)	Z "	W 2
010	car ret	R 4	D $	J '
011	O 9	G &	B ?	figures
100	space	I 8	S bell	U 7
101	H #	P 0	Y 6	Q 1
110	N ,	C :	F !	K (
111	M .	V ;	X /	letters

letters	37	figures	33
A	30	0	15
B	23	1	35
C	16	2	31
D	22	3	20
E	20	4	12
F	26	5	01
G	13	6	25
H	05	7	34
I	14	8	14
J	32	9	03
K	36	(	36
L	11	)	11
M	07	.	07
N	06	,	06
O	03	–	30
P	15	?	23
Q	35	:	16
R	12	$	22
S	24	bell	24
T	01	&	13
U	34	#	05
V	17	'	32
W	31	;	17
X	27	/	27
Y	25	!	26
Z	21	"	21
space	04	carriage return	02
line feed	10		

CARD READER CODE

A	61
B	62
C	63
D	64
E	65
F	66
G	67
H	70
I	71
J	41
K	42
L	43
M	44
N	45
O	46
P	47
Q	50
R	51
S	22
T	23
U	24
V	25
W	26
X	27
Y	30
Z	31
0	12
1	01
2	02
3	03
4	04
5	05
6	06
7	07
8	10
9	11
+	60
—	40
/	21
=	13
,	33
$	53
.	73
'	14
(	34
*	54
)	74
blank	00

	High order bits			
Low order bits	00	01	10	11
0000		blank	—	+ [&]
0001	1	/	J	A
0010	2	S	K	B
0011	3	T	L	C
0100	4	U	M	D
0101	5	V	N	E
0110	6	W	O	F
0111	7	X	P	G
1000	8	Y	Q	H
1001	9	Z	R	I
1010	0			
1011	= [#]	,	$	.
1100	' [@]	([%]	*	) [□]

HOLLERITH CARD CODE

digit	Zone			
	no zone	12	11	0
no punch	blank	+ [&]	—	0
1	1	A	J	/
2	2	B	K	S
3	3	C	L	T
4	4	D	M	U
5	5	E	N	V
6	6	F	O	W
7	7	G	P	X
8	8	H	Q	Y
9	9	I	R	Z
8-3	= [#]	.	$	,
8-4	' [@]	) [□]	*	([%]

LINE PRINTER CODE

A	61
B	62
C	63
D	64
E	65
F	66
G	67
H	70
I	71
J	41
K	42
L	43
M	44
N	45
O	46
P	47
Q	50
R	51
S	22
T	23
U	24
V	25
W	26
X	27
Y	30
Z	31
0	20
1	01
2	02
3	03
4	04
5	05
6	06
7	07
8	10
9	11
°	40
/	21
′	12
∽	13
⊃	14
∨	15
∧	16
<	17
$	52
=	53
—	54
)	55
(	57
¯	56

space	00
_	60
″	32
,	33
>	34
↑	35
→	36
?	37
×	72
.	73
+	74
]	75
[	77
|	76

	High order bits			
Low order bits	00	01	10	11
0000	space	0	°	_
0001	1	/	J	A
0010	2	S	K	B
0011	3	T	L	C
0100	4	U	M	D
0101	5	V	N	E
0110	6	W	O	F
0111	7	X	P	G
1000	8	Y	Q	H
1001	9	Z	R	I
1010	′	″	$	×
1011	∽	,	=	.
1100	⊃	>	¯	+
1101	∨	↑	)	]
1110	∧	→	—	|
1111	<	?	(	[

APPENDIX 3

Read-In Mode Sequence

The initial data input to PDP-4 is made using the keys and switches on the Operator Console. A small program read in manually can be used to read in a somewhat larger program from perforated tape. An example of such a routine is given below. It can also be used to read in other programs from perforated tape.

READ-IN LOADER

The purpose of the read-in loader is to load programs punched in "read-in mode," such as the block format loader described below. The read-in loader must be loaded by means of the console toggle switches. It loads tapes of the following format:

```
dac A
c(A)
dac B
c(B)
.
.
.
jmp Y
dummy word
```

Read-in mode tapes consist of word pairs giving a dac into an address, followed by the contents of that address. They are terminated by a jmp to the program followed by a dummy word (e.g., 0).

To load a read-in mode tape, place the tape in the reader, set the ADDRESS switches to 7770, and press START.

```
LOCATION  OCTAL CODE        MNEMONIC    REMARKS
 7762/      0         r,     0          /read one binary word
 7763/      700101           rsf
 7764/      607763           jmp . - 1  /wait for word to come in
 7765/      700112           rrb        /read buffer
 7766/      700144           rsb        /read another word
 7767/      627762           jmp i r    /exit subroutine
 7770/      700144    go,    rsb        /enter here, start reader going
 7771/      107762    g,     jms r      /get next binary word
 7772/      47775            dac out
 7773/      407775           xct out    /execute control word
 7774/      107762           jms r      /get data word
 7775/      0         out,   0          /store data word
 7776/      607771           jmp g      /continue
```

BLOCK FORMAT LOADER

The block format loader will read a block format binary tape of the following format:

dac A	A is the address of the first data word
−N	/complement of number of data words in block
N data words	/data words
Check sum	/sum of every word in block, except check sum

The routine occupies register 7737 to 7761, and uses the read-in loader subroutine to read each binary word. Upon completing a block, the computed check sum is compared with the read check sum and the loader halts if these differ. The block may be re-read by pulling the tape back to the beginning of the block and pressing the CONTINUE switch on the console.

```
LOCATION    MNEMONIC                      REMARKS
 7737/      rsb
      a,    jms r          /block format loader
            dac s
            xct s
            dac cks
            jms r
            dac out
      b,    add cks        /loop
            dac cks
            jms r
            isz out        /check count, last word read is check sum
            jmp s
            sad cks
            jmp a          /sum checks, continue
            hlt            /stop on check sum error
            jmp a − 1      /out
      s,    xx
            isz s
            jmp b
            cks = 7777
```

APPENDIX 4

PDP-4 Assembly Program

The more important characteristics of the PDP-4 Assembly Program are mentioned briefly here to provide the background necessary to understand the programming examples in this manual. The program and its complete description are furnished to purchasers of PDP-4.

CHARACTER SET: The character set includes digits 0 through 9, letters a through z, and the following punctuation characters:

Punctation Characters	Meaning
+ plus	add values
– minus	subtract values
Δ space	add values
∧ and	combine values by logical AND
∨ or	combine values by inclusive OR
(left parenthesis	enclose constant word
) right parenthesis	enclose constant word
. period	has value of current address
, comma	assign address tag
= equals sign	assign symbol on left of =
/ slash	begin comments; set current address
↵ carriage return	termination character
⇥ tab	termination character
— overbar	variable indicator

The characters Δ , ↵ , and ⇥ are nonprinting.

NUMBERS: Any sequence of digits delimited on the left and right by a punctuation character.

SYMBOLS: Any sequence of alphanumeric characters, the first of which must be a letter. Symbols are identified by the first six characters only.

'Value symbols' are those symbols which have a numerical value assigned to them, either in the permanent symbol table, or during assembly. Value symbols may be assigned by the use of a comma, indicating the symbol to the left of the comma is an address tag; or by an equals sign, indicating the symbol to the left of the equals sign is to be assigned the value of the word to the right of the equals sign.

```
Example:   a,          dzm 100
           b = -1
           c = a + b
```

SYLLABLES: A syllable can take several forms. It can be a value symbol, a period (.), a flexowriter input pseudo-instruction (flex or char), or a constant (a word enclosed in parentheses).

Examples:

```
al
100
1z2
flex abc
flex now
(add a + 1)
lac
abcdef
```

WORDS: A word is a string of syllables connected by the arithmetic operators plus, minus, space, AND or OR, delimited on the left by tab, carriage return, left parenthesis, or equals sign, and on the right by a tab or carriage return. A word may be a single number or symbol so delimited, or a string of symbols connected by the operators. If the word is delimited on the left by an equals sign then the symbol to the left of the equals sign is assigned a value equal to that of the word. Otherwise, the word is a storage word and will become part of the binary version of the program being assembled. The arithmetic operators, plus and space both mean add, while the operator minus means subtract.

Examples:

```
sad K ↵
lac a ↵
1000 − 20↵
add b + 2↵
jmp . − 2↵
a + b − c − 2↵
lac (add a + 1)↵
```

THE CHARACTER SLASH (/): The slash has two meanings. If immediately preceded by a tab or carriage return then slash initiates a comment, which is terminated by the next tab or carriage return. If slash is preceded by a word, then the address part of the word indicates the address into which the next instruction or data word will go. Normally, the first instruction or data word goes into register 22 and succeeding instructions or data words into succeeding registers. If the programmer wishes to break this sequence or wishes to start translating into some register other than 22, then a slash may be used to set the new address.

INDIRECT ADDRESSING: Indirect addressing is indicated by the symbol, i which has the value 20000.

Example: `lac i abc`

THE CHARACTER PERIOD (.): The period (.) has as its value the current address.

Example: `dac .` is equivalent to
`a, dac a`

PSEUDO INSTRUCTIONS

FLEXOWRITER INPUT PSEUDO INSTRUCTIONS: The pseudo-instruction, flex $\Delta\alpha\beta\gamma$ causes the (six-bit) FIO-DEC codes for the three characters following the space (Δ) to be read into one word which is taken as the value of the syllable. The code for the character α will go into bits 0-5 of the word, for β into bits 6-11, and for γ into bits 12-17. The code is a six-bit character, the first five of which are the FIO-DEC code, the sixth a 1 for upper case or a 0 for lower case.

Example: flex Δ boy

The pseudo-instruction, char $\Delta Z\gamma$ causes the (six-bit) FIO-DEC character, γ to be assembled into the left, middle, or right six bits of the word, depending on whether Z is r, m, or l.

```
Example:    char r0
            char m(
            char la
```

CONSTANTS: The MACRO assembly system has available a facility by which the program constants may be automatically stored. A constant must follow the rules for a word and is delimited on the left by a left parenthesis. The right delimiter may be a right parenthesis, carriage return, or tab. The value of the syllable, (α) is the address of the register containing α. The constant α will be stored in a constants block at the end of the program, and the address of α will replace (α).

Examples of the use of constants:

```
add (1)↵
lac (add z 1)↵
lac (−760000)↵
lac (flexo abc)↵
```

START: The pseudo-instruction, start indicates the end of the English tape. Instruction, start A must be followed by a carriage return. "A" is the address at which execution of the program is to begin, and causes a jmp A instruction punched at the end of the binary tape on pass two.

DECIMAL: The pseudo-instruction, decimal indicates all numbers are to be considered decimal.

OCTAL: The pseudo-instruction, octal indicates all numbers are to be considered octal.

APPENDIX 5

Multiply and Divide Subroutines

MULTIPLY SUBROUTINE

```
/PDP-4 ones complement single precision multiplication subroutine
/calling sequence: /lac multiplier
                   /jms mult
                   /lac multiplicand
                   /return; low order product in AC, high order product in mp5
/time = 2.6 msec. for non-zero cases, approximately 100 microsec. for zero.

mult,     0
          dzm mp5
          sna
          jmp mpz
          spa + cll - opr
          cma + cml - opr
          dac mpl
          xct i mult
          sna
          jmp mpz
          spa
          cma + cml - opr
          dac mp2
          lac (360000
          ral
          dac mpsign
          lac (-21
          dac mp3

mp4,      lac mpl
          rar
          dac mpl
          lac mp5
          spl + cll - opr
          tad mp2
          rar
          dac mp5
          isz mp3
          jmp mp4

mpsign,   0
          dac mp5
          lac mp1
          rar
          xct mpsign

mpz,      isz mult
          jmp i mult

start
```

DIVIDE SUBROUTINE

```
/PDP-4 ones complement divide subroutine
/calling sequence: /lac high order dividend
                   /jms divide
                   /lac low order dividend
                   /lac divisor
                   /return; quot. in AC, rem. in dvd. if high dividend is
/greater than divisor, no divide takes place and L=>1. Time = 3.1 ms

divide, 0
        spa + cll — opr
        cma + cml — opr
        dac dvd
        xct i divide
        spl
        cma
        dac quo
        jms dv5

dv5,    0          /remainder has sign of dividend
        isz divide
        xct i divide
        sma + cml — opr
        cma + cml — opr
        jms dv4

dv4,    0
        cll
        tad (1
        dac dvs
        tad dvd
        isz divide
        spl
        jmp i divide
        lac (—22
        dac dv1
        jmp dv2

dv3,    lac dvd
        ral
        dac dvd
        tad dvs
        spl
        dac dvd
```

DIVIDE SUBROUTINE
(continued)

```
dv2,    lac quo
        ral
        dac quo
        isz dv1
        jmp dv3

        lac dv5
        ral
        lac dvd
        spl
        cma
        dac dvd
        lac dv4
        ral
        lac quo
        spl
        cma + cll — opr
        jmp i divide

start
```

APPENDIX 6
Programming Aids

The following programming aids are supplied with the PDP-4.

PDP-4 ASSEMBLY PROGRAM — A one-pass assembler which allows mnemonic symbols to be used for addresses and instructions. Constants are automatically assigned. Text statements may be written for printing at run time, and a decimal mode may be specified. Up to six character symbols may be used, and the symbol table may be punched on paper tape for use with the debugging tape below.

DDT-4 DEBUGGING TAPE — Provides communication with a program via the on-line typewriter. Registers may be examined (using mnemonic codes) and modified. Communication is entirely in symbolic language. Programs may have break points inserted and then run under DDT-4 control, similar to a tracing routine. A program may be searched for particular words.

DOUBLE-PRECISION FLOATING POINT PACKAGE — Provides floating-point arithmetic with a 36-bit mantissa and 18-bit exponent. The routines include plus, minus, divide, multiply, fix-to-float, and float-to-fix, with decimal input and output.

MAINTENANCE ROUTINES — There are five maintenance routines. These tests are also used as DEC's standard acceptance test routines.

(a) CONTEST (CONtinuous TEST) — Verifies that all machine functions are performing properly. Each instruction is tested, a core checkerboard pattern is run, a tape is punched and read, and a message is typed. The test then repeats itself.

(b) INSTEP (INStruction TEst Programs) — Test all machine instructions under various modes.

(c) Checkerboard Program — Provides continuous memory testing with four different patterns.

(d) Reader and Punch Test — Checks the start time of the reader and checks the reader using different patterns and variable times. The punch is tested by providing tapes for the reader test.

(e) Teleprinter Test.

TAPE REPRODUCER — Reproduces tape using the Interrupt Mode.

PUNCH ROUTINES — Allow punching in either block format or read-in mode format.

OCTAL DEBUG — A simple debugging routine.

MISCELLANEOUS INPUT-OUTPUT ROUTINES —Octal, decimal, double precision input and output and special Teletype conversion routines.

DEMONSTRATION PROGRAMS — Included are: Three Point Display (Tri-Pos), Pen Follow, Type-in Character Display, and Character Punch.

FLOATING POINT FUNCTIONS — Allows various functions to be computed, such as double precision sine, cosine, tangent, exponents, log base e, and square root. Inquire at DEC for the completion date of these subroutines.

ALGEBRAIC COMPILER — Inquire at DEC for the completion date of this FORTRAN compiler.

APPENDIX 7
Powers Of Two

2^n	n	2^{-n}
1	0	1.0
2	1	0.5
4	2	0.25
8	3	0.125
16	4	0.062 5
32	5	0.031 25
64	6	0.015 625
128	7	0.007 812 5
256	8	0.003 906 25
512	9	0.001 953 125
1 024	10	0.000 976 562 5
2 048	11	0.000 488 281 25
4 096	12	0.000 244 140 625
8 192	13	0.000 122 070 312 5
16 384	14	0.000 061 035 156 25
32 768	15	0.000 030 517 578 125
65 536	16	0.000 015 258 789 062 5
131 072	17	0.000 007 629 394 531 25
262 144	18	0.000 003 814 697 265 625
524 288	19	0.000 001 907 348 632 812 5
1 048 576	20	0.000 000 953 674 316 406 25
2 097 152	21	0.000 000 476 837 158 203 125
4 194 304	22	0.000 000 238 418 579 101 562 5
8 388 608	23	0.000 000 119 209 289 550 781 25
16 777 216	24	0.000 000 059 604 644 775 390 625
33 554 432	25	0.000 000 029 802 322 387 695 312 5
67 108 864	26	0.000 000 014 901 161 193 847 656 25
134 217 728	27	0.000 000 007 450 580 596 923 808 125
268 435 456	28	0.000 000 003 725 290 298 461 914 062 5
536 870 912	29	0.000 000 001 862 645 149 230 957 031 25
1 073 741 824	30	0.000 000 000 931 322 574 615 478 515 625
2 147 483 848	31	0.000 000 000 465 661 287 307 739 257 812 5
4 294 967 296	32	0.000 000 000 232 830 643 653 869 628 906 25
8 589 934 592	33	0.000 000 000 116 415 321 826 934 814 453 125
17 179 869 184	34	0.000 000 000 058 207 660 913 467 407 226 562 5
34 359 738 368	35	0.000 000 000 029 103 830 456 733 703 613 081 25
68 719 476 736	36	0.000 000 000 014 551 915 228 366 851 806 640 625
137 438 953 472	37	0.000 000 000 007 275 957 614 183 425 903 320 312 5
274 877 906 944	38	0.000 000 000 003 637 978 807 091 712 951 660 156 25
549 755 813 888	39	0.000 000 000 001 818 989 403 545 856 475 830 078 125
1 099 511 627 776	40	0.000 000 000 000 909 494 701 772 928 237 915 039 062 5
2 199 023 255 552	41	0.000 000 000 000 454 747 350 886 464 118 957 519 531 25
4 398 046 511 104	42	0.000 000 000 000 227 373 675 443 232 059 478 759 765 625
8 796 093 022 208	43	0.000 000 000 000 113 686 837 721 616 029 739 379 882 812 5
17 592 186 044 416	44	0.000 000 000 000 056 843 418 860 808 014 869 689 941 406 25
35 184 372 088 832	45	0.000 000 000 000 028 421 709 430 404 007 434 844 970 703 125
70 368 744 177 664	46	0.000 000 000 000 014 210 854 715 202 003 717 422 485 351 562 5
140 737 488 355 328	47	0.000 000 000 000 007 105 427 357 601 001 858 711 242 675 781 25
281 474 976 710 656	48	0.000 000 000 000 003 552 713 678 800 500 929 355 621 337 890 625
562 949 953 421 312	49	0.000 000 000 000 001 776 356 839 400 250 464 677 810 668 945 312 5
1 125 899 906 842 634	50	0.000 000 000 000 000 888 178 419 700 125 232 338 905 334 472 656 25
2 251 799 813 985 248	51	0.000 000 000 000 000 444 089 209 850 062 616 169 452 667 236 328 125
4 503 599 627 370 496	52	0.000 000 000 000 000 222 044 604 925 031 308 084 726 333 668 164 062 5
9 007 199 254 740 992	53	0.000 000 000 000 000 111 022 302 462 515 654 042 363 166 834 582 031 25
18 014 398 509 481 984	54	0.000 000 000 000 000 055 511 151 231 257 827 021 171 513 417 041 015 625
36 028 797 018 963 968	55	0.000 000 000 000 000 027 755 575 615 628 913 510 590 791 708 520 507 812 5
72 057 594 037 927 936	56	0.000 000 000 000 000 013 877 787 807 814 456 755 215 395 854 260 253 906 25
144 115 188 075 855 872	57	0.000 000 000 000 000 006 938 893 903 907 228 377 647 697 927 130 126 953 125
288 230 376 151 711 744	58	0.000 000 000 000 000 003 469 446 951 953 614 188 823 848 963 565 063 476 562 5
576 460 752 303 423 488	59	0.000 000 000 000 000 001 734 723 475 976 807 094 411 924 481 782 531 738 281 25
1 152 921 504 606 846 976	60	0.000 000 000 000 000 000 867 361 737 988 403 547 205 962 240 891 265 869 140 625

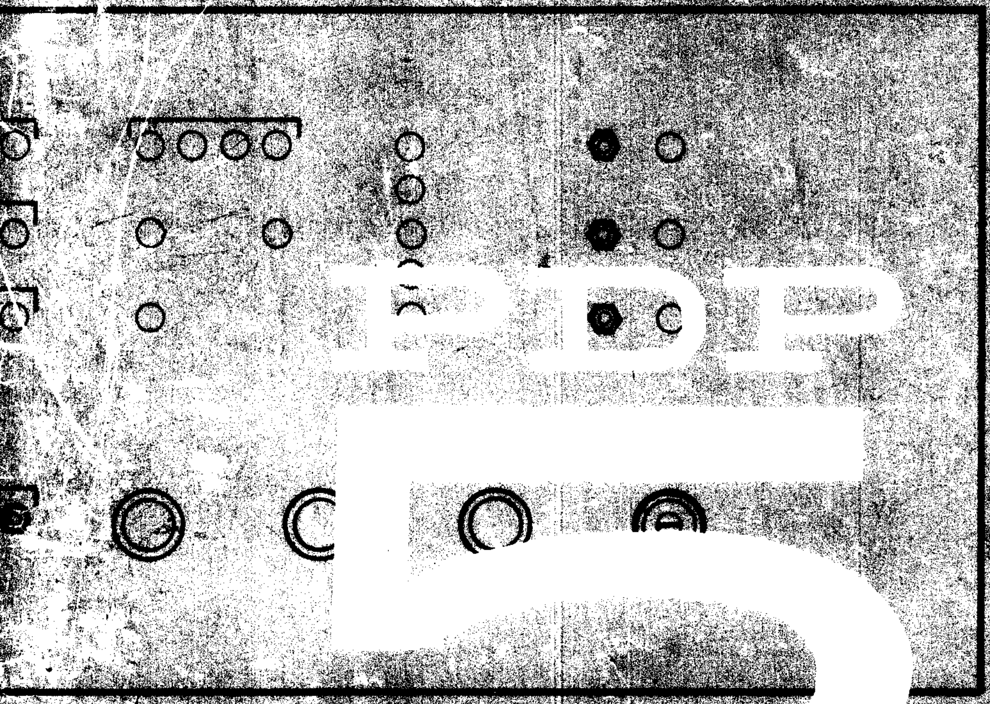
digital

DIGITAL EQUIPMENT CORPORATION • MAYNARD, MASSACHUSETTS

F-45 Printed in U.S.A. 2M—10/63

Gordon Bell

F-55
2/64



PDP 5 HANDBOOK

PROGRAMMED DATA PROCESSOR-5 HANDBOOK

DIGITAL EQUIPMENT CORPORATION • MAYNARD, MASSACHUSETTS

FOREWORD

This handbook concerns programming and operating the Programmed Data Processor-5, a high-speed, stored program, digital computer manufactured by the Digital Equipment Corporation. Chapter 1 summarizes the electrical and logical features of the computing system and analyzes it into three major functional elements: arithmetic and control, input-output control, and input-output devices. Chapters 2, 3, and 4 present detailed information on the function, instructions, and programming of the three major system elements. Practical information for making electrical connections between any input-output device and the computing system at the input-output control is presented in Chapter 5. Appendixes provide detailed information which may be helpful in specific programming assignments. Although program examples are given in this document, no attempt has been made to teach programming techniques. The meaning and use of special characters employed in the programming examples are explained in the description of the Program Assembly Language, available from the DEC Program Library.

Table of Contents

Table of Contents (continued)

List of Tables

List of Illustrations

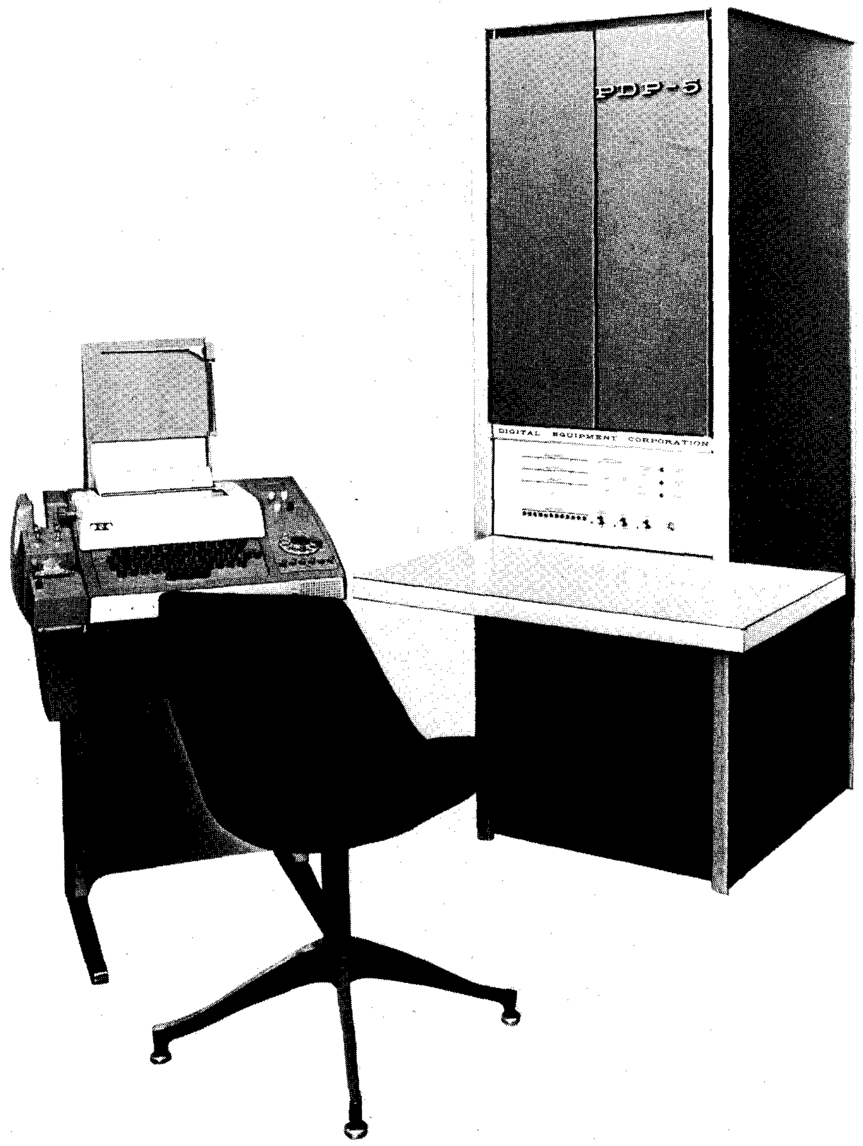


Figure 1 Typical PDP-5 Computing System

CHAPTER 1
SYSTEM INTRODUCTION

The Digital Equipment Corporation Programmed Data Processor-5 (PDP-5) is designed for use as a small-scale general-purpose computer, an independent information handling facility in a larger computer system, or as the control element in a complex processing system. The PDP-5 is a one-address, fixed word length, parallel computer using 12 bit, twos complement arithmetic. Cycle time of the 1024- or 4096-word random address magnetic-core memory is 6 microseconds. Standard features of the system include indirect addressing and facilities for instruction skipping, program interruption, or program halting as functions of input-output device conditions.

The 6-microsecond cycle time of the machine provides a computation rate of 55,555 additions per second. Addition is performed in 18 microseconds (with one number in the accumulator) and subtraction is performed in 30 microseconds (with the subtrahend in the accumulator). Multiplication is performed in approximately 2.0 milliseconds by a subroutine that operates on two 12-bit numbers to produce a 24-bit product, leaving the 12 most significant bits in the accumulator. Division of two 12-bit numbers is performed in approximately 3.5 milliseconds by a subroutine that produces a 12-bit quotient in the accumulator and a 12-bit remainder in core memory.

Flexible, high-capacity, input-output capabilities of the computer allow it to operate a variety of peripheral equipment. In addition to standard Teletype and perforated-tape equipment, the system is capable of operating in conjunction with a number of optional devices such as high-speed perforated-tape readers and punches, card equipment, a line printer, analog-to-digital converters, cathode-ray tube displays, and magnetic-tape equipment. The system is easily adapted for connection to equipment of special design.

PDP-5 is completely self-contained, requiring no special power sources or environmental conditions. A single source of 115-volt, 60-cycle, single-phase power is required to operate the machine. Internal power supplies produce all of the operating voltages required. Solid-state system modules and built-in provisions for marginal checking insure reliable operation in ambient temperatures between 50 and 105 degrees Fahrenheit.

The primary functions of the PDP-5 system are performed by an arithmetic and control element, an input-output control element, and the input-output devices. Figure 2 shows the relationship of these elements.

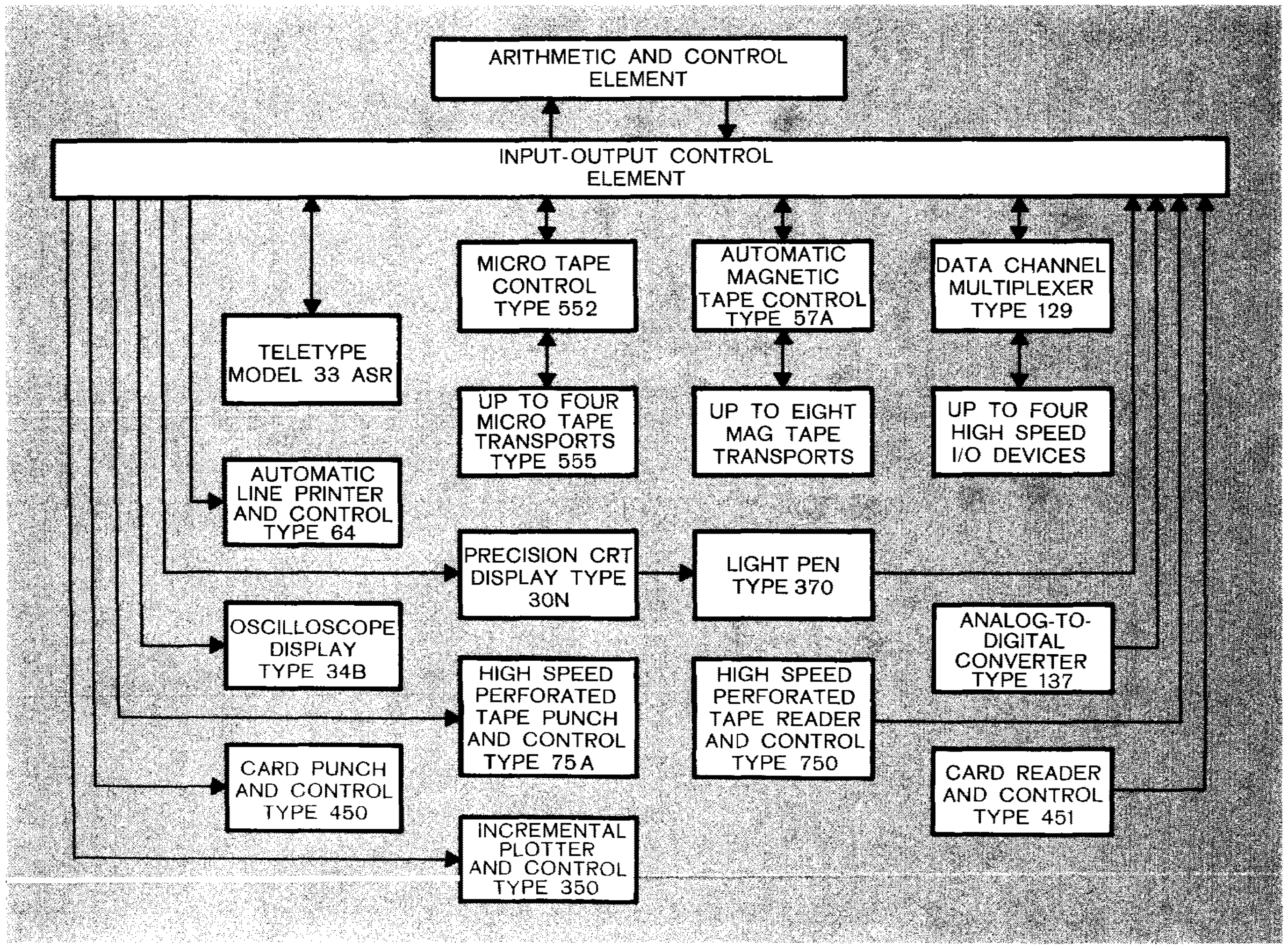

ARITHMETIC AND CONTROL ELEMENT
INPUT-OUTPUT CONTROL ELEMENT
TELETYPE MODEL 33 ASR
MICRO TAPE CONTROL TYPE 552
UP TO FOUR MICRO TAPE TRANSPORTS TYPE 555
AUTOMATIC MAGNETIC TAPE CONTROL TYPE 57A
UP TO EIGHT MAG TAPE TRANSPORTS
DATA CHANNEL MULTIPLEXER TYPE 129
UP TO FOUR HIGH SPEED I/O DEVICES
AUTOMATIC LINE PRINTER AND CONTROL TYPE 64
PRECISION CRT DISPLAY TYPE 30N
LIGHT PEN TYPE 370
OSCILLOSCOPE DISPLAY TYPE 34B
ANALOG-TO-DIGITAL CONVERTER TYPE 137
HIGH SPEED PERFORATED TAPE PUNCH AND CONTROL TYPE 75A
HIGH SPEED PERFORATED TAPE READER AND CONTROL TYPE 750
CARD PUNCH AND CONTROL TYPE 450
CARD READER AND CONTROL TYPE 451
INCREMENTAL PLOTTER AND CONTROL TYPE 350

Figure 2 PDP-5 System Components

The arithmetic and control element contains all of the registers that perform arithmetic and logic operations, the core memory for storage and retrieval of data and instructions, and the operator console, which indicates the contents of registers and provides a means of modifying data. Chapter 2 of this handbook describes these functions in detail.

The input-output control element provides communications between the arithmetic and control element and standard, optional, or special input-output devices. Components of this control may be housed in the main computer cabinet or with the I/O device. I/O device selection, input-output skip, program interrupt, input-output halt, and data break control are features of this element and are discussed in Chapter 3.

All of the input-output devices are optional except the Teletype Model 33 ASR. Standard and optional I/O equipment programming information is presented in Chapter 4.

THE TELETYPE MODEL 33 ASR provides a means of supplying data to the computer from perforated tape or a keyboard, or of receiving output information from the computer in the form of perforated tape or typed copy. Maximum speed of these operations is ten characters per second.

THE HIGH SPEED PERFORATED TAPE PUNCH AND CONTROL TYPE 75A perforates 8-hole paper tape at a rate of 63.3 lines per second.

THE HIGH SPEED PERFORATED TAPE READER AND CONTROL TYPE 750 senses 8-hole perforated paper tape photoelectrically at the rate of 300 lines per second.

THE ANALOG-TO-DIGITAL CONVERTER TYPE 137 is wired into each system. Modules to activate this feature are optional. The converter operates in the normal successive approximation fashion, using existing computer registers as the shift register and buffer register. The converter provides a 12-bit word; however, the last bit is insignificant.

THE CARD READER AND CONTROL TYPE 451 operates at a rate of 200 or 800 cards per minute. Cards are read column by column. Column information may be read in alphanumeric or binary mode. The alphanumeric mode converts the 12-bit Hollerith Code of one column into the 6-bit binary-coded decimal code with code validity checking. The binary mode reads a 12-bit column directly into the PDP-5. Approximately one percent of a Card Reader program running time is required to read the 80 columns of information at the 200 cards per minute rate.

THE CARD PUNCH CONTROL TYPE 450 permits operation of a standard IBM Type 523 Summary Punch with the PDP-5. Punching can occur at a rate of 100 cards per minute. Cards are punched one row at a time at 40-millisecond intervals.

THE AUTOMATIC LINE PRINTER AND CONTROL TYPE 64 prints a selection of 63 characters at up to 300 lines of 120 characters per minute. Printing of one group of 120 characters can be carried out while the next 120 characters are being loaded into the printer. Loading, printing, and format are under program control. Format is program selected from a punched format tape in the printer.

THE INCREMENTAL PLOTTER AND CONTROL TYPE 350 provides high-speed plotting of points, continuous curves, points connected by curves, curve identification symbols, letters, and numerals under program control.

THE DATA CHANNEL MULTIPLEXER TYPE 129 automatically transfers data directly between the computer core memory and up to four I/O devices. The computer core memory address of each transfer is specified by the I/O device. Transfers are made through the normal data break facilities and breaks are performed in accordance with an assigned I/O device priority.

THE OSCILLOSCOPE DISPLAY TYPE 34B plots data point by point on a high resolution oscilloscope, such as the Tektronix Model RM 503. Each axis is determined by 10 binary bits.

THE PRECISION CRT DISPLAY TYPE 30N displays data on a 9¼ inch by 9¼ inch area. Information is plotted point by point to form either graphical or tabular data. The X and Y coordinates are each controlled by a separate 10-bit word.

THE LIGHT TYPE PEN 370 is a photoelectric device which signals the computer when it detects information displayed on the Type 30N Precision CRT Display. Upon signal from the light pen, the computer carries out previously programmed instructions.

THE DUAL MICRO TAPE SYSTEM TYPE 555-552 provides a fixed-address magnetic-tape facility for high-speed loading, readout, and program updating. A system consists of a Type 555 Micro Tape Transport and a Type 552 Micro Tape Control. Each transport contains two independent tape drivers. Up to four transports (8 drives) can be used with one control. Each reel, containing up to four-million bits of data, can be written or read under program control.

THE AUTOMATIC MAGNETIC TAPE CONTROL TYPE 57A reads and writes high and low density, IBM compatible magnetic tape at a transfer rate of 15,000 characters per second.

The following special terms are used throughout this handbook in the explanation of equipment functions and instructions:

Term	Explanation
C(A)	Contents of A
$A => B$	A replaces B or B is set to A
Y	Any core memory location
Y_i	Any given bit in Y
Y_{1-4}	Bits 1 through 4 of Y
Y^1_i	The 1 output of bit *j* of register Y
N_r	Number N to the radix r
$C(A)_{0-5} => C(Y)_{6-11}$	The contents of bits 6 through 11 of core memory location Y are set to correspond with the contents of bits 0 through 5 of register A
$\veebar$	Exclusive OR
V	Inclusive OR
$\wedge$	AND
$\bar{A}$	Ones complement of A

CHAPTER 2
ARITHMETIC AND CONTROL

Functions

To perform the required arithmetic, logic, and data processing operations and to store, retrieve, control, and modify information the arithmetic and control element uses the logic components shown in Figure 3 and described in the following paragraphs.

ACCUMULATOR (AC)

Arithmetic operations are performed in this 12-bit register. The AC can be cleared or complemented. Its contents can be rotated right or left with the link. The contents of the memory buffer register can be added to the contents of the AC and the result left in the AC. The contents of both these registers may be combined by the logical operation AND, the result remaining in the AC. The memory buffer register and the AC also have gates which allow them to be used together as the shift register and buffer register of a successive approximation analog-to-digital converter. The inclusive OR may be formed between the AC and the switch register on the operator console and the result left in the AC.

The accumulator also acts as an input-output register. All programmed information transfers between core memory and an external device pass through the accumulator.

LINK (L)

This one-bit register is used to extend the arithmetic facilities of the accumulator. It is used as the carry register for twos complement arithmetic. This feature greatly simplifies multiple precision arithmetic. The link can be cleared and complemented, and it can be rotated as part of the accumulator.

MEMORY BUFFER REGISTER (MB)

All information transfers between the computer registers and the core memory are temporarily held in the MB. Information can be transferred into MB from the accumulator or memory address register. The MB can be cleared, incremented by one or two, or shifted right. Information can be set into the

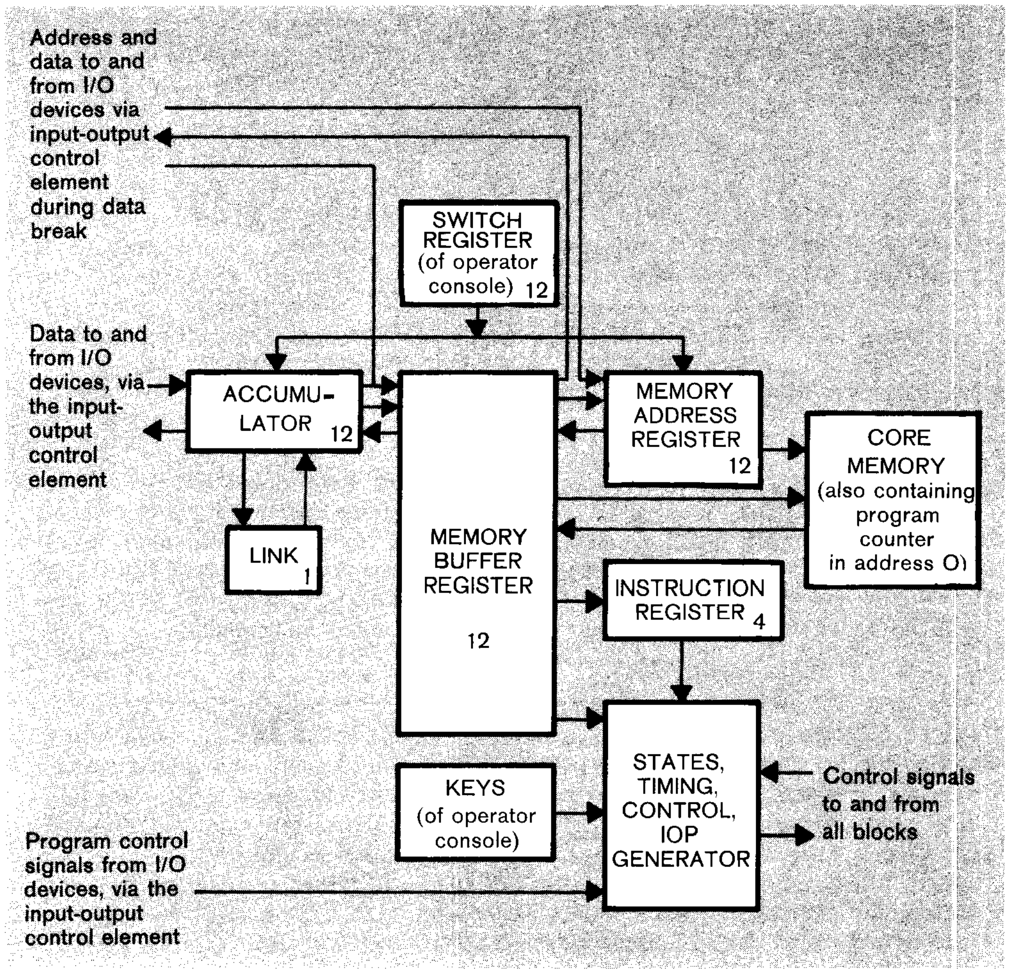

Address and data to and from I/O devices via input-output control element during data break
SWITCH REGISTER (of operator console) 12
Data to and from I/O devices, via the input-output control element
ACCUMU-LATOR 12
MEMORY ADDRESS REGISTER 12
CORE MEMORY (also containing program counter in address O)
LINK 1
MEMORY BUFFER REGISTER 12
INSTRUCTION REGISTER 4
KEYS (of operator console)
STATES, TIMING, CONTROL, IOP GENERATOR
Control signals to and from all blocks
Program control signals from I/O devices, via the input-output control element

Figure 3 Arithmetic and Control Element

MB from an external device during a data break or from core memory, via the sense amplifiers. Information is read from a memory location in 2 microseconds and rewritten in the same location in another 2 microseconds of one 6-microsecond memory cycle.

MEMORY ADDRESS REGISTER (MA)

The address in core memory which is currently selected for reading or writing is contained in this 12-bit register. Therefore, all 4096 words of core memory can be addressed directly by this register. The MA can be cleared or incremented by one. Data can be set into it from the memory buffer register, from the switch register, or from an I/O device. The output can be disabled (i.e. forced to indicate all binary zeros) without affecting the contents of the register.

INSTRUCTION REGISTER (IR)

This 4-bit register contains the operation code of the instruction currently being performed by the machine. Information is loaded into the IR from the memory buffer register during a Fetch cycle. The contents of the three most significant bits of the IR are decoded to produce the eight instructions, and affect the cycles and states entered at each step in the program. The least significant bit (the indirect bit) is used in addressing core memory to specify a defer cycle in addressable instructions and to differentiate the two groups operate instructions.

PROGRAM COUNTER (PC)

The program sequence, that is, the order in which instructions are performed, is determined by the PC. This 12-bit core memory register contains the address of the core memory location from which the last instruction was taken. Information enters the PC from the MB, since core memory address 0 is used as the PC. Because the PC is in core memory, it can be manipulated by the program in the same manner as any other core memory location.

CORE MEMORY

The core memory provides storage for instructions to be performed and information to be processed or distributed. This random addressable magnetic core memory holds either 1024 or 4096 12-bit words. Memory location 0 is used as the program counter, location 1 is used to store the contents of the PC following a program interrupt, and location 2 is used to store the first instruction to be executed following a program interrupt. (When a program interrupt occurs, the contents of the PC are stored in location 1; and program control is transferred to location 2 automatically.) Locations 10 through 17 are used for auto-indexing. All other locations can be used to store instructions or data.

STATES, TIMING, CONTROL, AND IOP GENERATOR

This logic component of the computer establishes the basic timing of all computer operations, controls the operation of all previously mentioned registers, and generates the three IOP pulses which are supplied to the device selectors in the input-output control element. It also establishes the cycles or primary control states entered to accomplish each instruction. The control state entered next is determined at the completion of the current one. All states except break are determined by the instruction.

PROGRAM COUNTER (P): This state reads the contents of the program counter from core memory location 0 into the MB, increments the contents of the MB by 1 (or 2 for a skip instruction), and rewrites the contents of the MB back in location 0. The incremented contents of the PC remain in the MB as the address of the current instruction. During a jump or jump to subroutine instruction, the effective address specified by the jmp or jms is written into location 0 to transfer program control. Completion of a P cycle initiates a Fetch cycle.

FETCH (F): During this state an instruction word is read from the core memory location specified by the contents of the program counter.

EXECUTE 1 (E_1): This state occurs for all instructions requiring an operand from core memory. The contents of the core memory location specified by the least significant bits of the instruction are read into the memory buffer register and the operation specified by bits 0 through 2 of the instruction is performed.

EXECUTE 2 (E_2): When a jump to subroutine instruction is being executed, this state is entered to write the contents of the program counter into core memory location Y.

DEFER (D): When a 1 is present in bit 3 of a memory reference instruction, the defer state is entered to obtain the full 12-bit address of the operand from the address in the current page or page 0 specified by bits 4 through 11 of the instruction. The process of address deferring is called indirect addressing because access to the operand is addressed indirectly, or deferred, to another memory location.

BREAK (B): When this state is established, the sequence of instructions is broken for a data interrupt. The break normally occurs at the completion of the current instruction. If the interrupt occurs during a jump or jump to subroutine instruction, the break begins only after two instructions have been completed (the instruction jumped to is executed). The data interrupt allows information to be transferred directly between core memory and an external device. When this transfer has been completed, the program sequence is resumed from the point of the break.

OPERATOR CONSOLE

Switches and keys on the operator console allow manual program and information insertion or modification. Indicator lamps display the status of the machine and the contents of major registers. Register indicators light to denote the presence of a 1 in a specific bit. While a program is running, the brightness of an indicator is related to the percentage of time that the related bit holds a 1.

Figure 4 shows the operator console and the following tables list the function of switches, keys, and indicators.

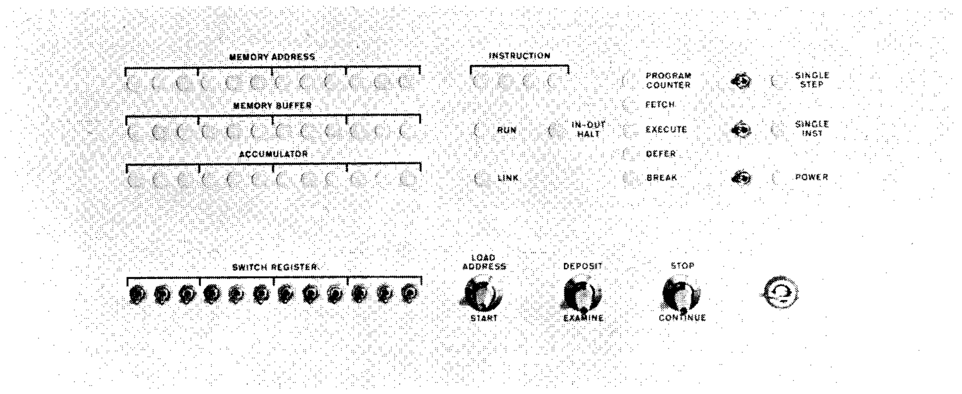


Figure 4 Operator Console

CONSOLE SWITCH FUNCTIONS

Switch	Function
SWITCH REGISTER	Provides a means of manually setting a 12-bit word into the machine. Switches in the up position correspond to ones, down, to zeros. Contents of this register are loaded into the memory address register by the LOAD ADDRESS key, or into the memory buffer register and core memory by the DEPOSIT key. The contents of the switch register (SR) can be set into the accumulator under program control.
SINGLE STEP	Causes the computer to halt at the completion of each memory cycle. Repeated operation of the CONTINUE key steps the program one cycle at a time so that the state of the machine can be examined at each step.
SINGLE INST	Causes the computer to stop at the completion of each instruction.
POWER	Controls primary power in the computer.

CONSOLE KEY FUNCTIONS

Key	Functions
LOAD ADDRESS	Deposits the contents of the switch register into the memory address register.

CONSOLE KEY FUNCTIONS (continued)

Key	Functions
START	Starts the computer after turning off the program interrupt system and clearing both the AC and L. The first instruction is taken from the core memory at the address presently in the memory address register.
DEPOSIT	Sets the word contained in the switch register into the core memory at the location specified by the memory address register. The results remain in the memory buffer register. The memory address register is then incremented by one, allowing rapid data deposits in sequential core memory locations.
EXAMINE	Sets the contents of the core memory location selected by the memory address register into the accumulator and the memory buffer register. The memory address register is then incremented by one, allowing rapid examination of data in sequential core memory locations.
STOP	Causes the computer to stop at the completion of the memory cycle in progress at the time of key operation.
CONTINUE	Causes the computer to resume execution of the instruction at the address held in the PC, from the program state indicated by the panel lamps.
LOCK SWITCH	Disables all console keys and switches except the SR to prevent inadvertent power turn-off or program interference while a program is in progress.

CONSOLE LAMP INDICATIONS

Lamp(s)	Indications
MEMORY ADDRESS	Indicate the contents of the memory address register.
MEMORY BUFFER	Indicate the contents of the memory buffer register.
ACCUMULATOR	Indicate the contents of the accumulator.
INSTRUCTION	Indicate the contents of the instruction register.
RUN	Indicates that the computer is executing instructions.

CONSOLE LAMP INDICATIONS (continued)

Lamp(s)	Indications
IN-OUT HALT	Indicates that the computer is waiting for an input-output device to complete its operation.
LINK	Indicates the contents of the carry link.
PROGRAM COUNTER, FETCH, EXECUTE, DEFER, BREAK	Indicate the primary control state of the machine and that the next memory cycle will be a program counter, fetch, execute, defer, or break cycle respectively.
SINGLE STEP and SINGLE INST	Indicate that the SINGLE STEP or SINGLE INST switch is on the ON position.
POWER	Indicates that power is turned on in the computer.

Instructions

Instruction words are of two types: memory reference and augmented. Memory reference instructions store or retrieve data from core memory, while augmented instructions do not. All instructions utilize bits 0 through 2 to specify the operation code. Operation codes of 0_8 through 5_8 specify memory reference instructions, and codes of 6_8 and 7_8 specify augmented instructions. Instruction execution times are multiples of the 6-microsecond computer cycle time. Memory reference instructions require 12, 18, or 24 microseconds for execution. Indirect addressing increases the execution time of a memory reference instruction by 6 microseconds. The augmented instructions, input-output transfer and operate, are performed in 12 microseconds.

MEMORY REFERENCE INSTRUCTIONS

Word format of memory reference instructions is shown in Figure 5, and the instructions are explained in the Memory Reference Instructions Table.

Since this system can contain a 4096-word memory, 12 bits are required to address all locations. To simplify addressing, the memory is divided into blocks, or pages, of 128 words (200_8 addresses). Pages are numbered 0_8 through 37_8, a 1024-word memory having pages 0_8 through 7_8, and a 4096-word memory using all 32 pages. The seven address bits (bits 5 through 11) of a memory reference instruction can address any location in the page on which the current instruction is located by placing a 1 in bit 4 of the instruction. By placing a 0 in bit 4 of the instruction, any location in page 0 can be addressed directly from any page of core memory. All other core memory locations must be addressed indirectly by placing a 1 in bit 3 and placing a

7-bit effective address in bits 5 through 11 of the instruction to specify the location in the current page or page 0, which contains the full 12-bit absolute address of the operand.

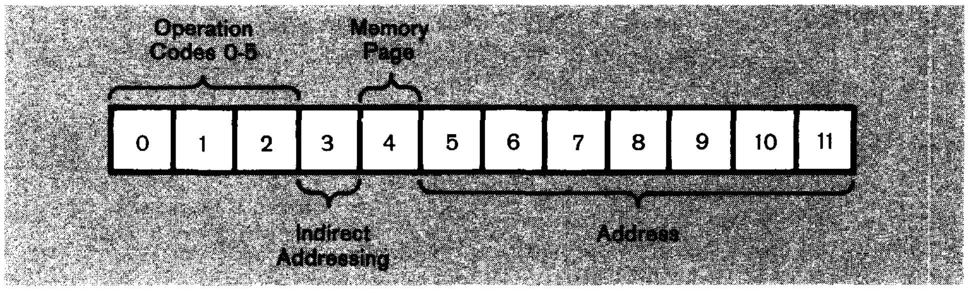


Figure 5 Memory Reference Instruction Bit Assignments

MEMORY REFERENCE INSTRUCTIONS

Mnemonic Symbol	Octal Code	Time (μsec)	Operation
and Y	0	18	Logical AND. The AND operation is performed between the C(Y) and the C(AC). The result is left in the AC, and the original C(AC) are lost. The C(Y) are unchanged. Corresponding bits are compared independently. This instruction, often called extract or mask, can be considered as a bit-by-bit multiplication. $C(Y)_i \wedge C(AC)_i => C(AC)_i$

Example

$C(AC)_i$ original	$C(Y)_i$	$C(AC)_i$ final
0	0	0
0	1	0
1	0	0
1	1	1

Mnemonic Symbol	Octal Code	Time (μsec)	Operation
tad Y	1	18	Twos complement add. The C(Y) are added to the C(AC) in twos complement arithmetic. The result is left in the AC and the original C(AC) are lost. The C(Y) are unchanged. If there is a carry from AC_0, the link is complemented. This feature is useful in multiple precision arithmetic. $C(Y) + C(AC) => C(AC)$.

Mnemonic Symbol	Operation Code	Time (μsec)	Operation
isz Y	2	18	Index and skip if zero. The C(Y) are incremented by one in twos complement arithmetic. If the resultant C(Y) = 0, the next instruction is skipped. If the resultant C(Y) $\neq$ 0, the program proceeds to the next instruction. The C(AC) are unaffected. C(Y) + 1 = > C(Y) . if result = 0, C(PC) + 1 = > C(PC).
dca Y	3	18	Deposit and clear AC. The C(AC) are deposited in core memory location Y and the AC is then cleared. The previous C(Y) are lost. C(AC) => C(Y), then 0 => C(AC).
jms Y	4	24	Jump to subroutine. The C(PC) contained in core memory location 0 are deposited in core memory location Y. The next instruction is taken from location Y + 1. C(PC) + 1 => C(Y) Y + 1 => C(PC)
jmp Y	5	12	Jump to Y. The C(PC) contained in core memory location 0 are set to address Y. The next instruction is taken from core memory location Y. The original C(PC) are lost. Y => C(PC).

AUGMENTED INSTRUCTIONS

There are two augmented instructions or instructions which do not reference core memory. They are the input-output transfer, which has an operation code of 6, and the operate, which has an operation code of 7. Bits 3 through 11 within these instructions function as an extension of the operation code and can be microprogrammed to perform several operations with one instruction. Augmented instructions are two-cycle (P, F) instructions requiring 12 microseconds for execution. During the second cycle, three clock pulses are available to initiate operations as a function of bit microprogramming. These clock pulses are designated event times 1, 2 and 3 and are separated by 1 microsecond.

INPUT-OUTPUT TRANSFER INSTRUCTION: Microinstructions of the input-output transfer (iot) instruction effect information transfers between the arithmetic and control element and an input-output device via the input-output control element. Specifically, when operation code 6 is detected, the IOP generator is enabled to produce IOP 1, 2, and 4 pulses as a function of the contents of bits 9 through 11. These pulses are gated in the device selector of the selected I/O device to produce the IOT pulses which enact a transfer.

The format of the iot instruction is shown in Figure 6. Bits 3 through 8 are used to select the I/O device; and bits 9 through 11 enable generation of IOP pulses during event times 3, 2, and 1, respectively. Operations performed by iot microinstructions are explained in Chapter 4.

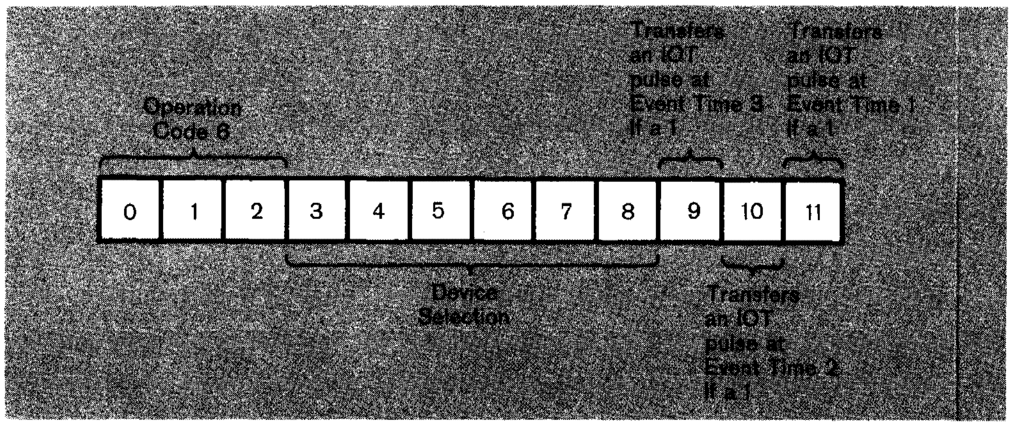

Operation Code 6
Transfers an IOT pulse at Event Time 3 If a 1
Transfers an IOT pulse at Event Time 1 If a 1
0
1
2
3
4
5
6
7
8
9
10
11
Device Selection
Transfers an IOT pulse at Event Time 2 If a 1

Figure 6 IOT Instruction Bit Assignments

OPERATE INSTRUCTION: The operate instruction consists of two groups of microinstructions. Group 1 (opr 1) is principally for clear, complement, rotate, and increment operations and is designated by the presence of a 0 in bit 3. *Group 2 (opr 2) is used principally in checking the contents of the accumulator* and link and continuing to or skipping the next instruction based on the check. A 1 in bit 3 designates an opr 2 microinstruction.

Group 1 operate microinstruction format is shown in Figure 7, and the microinstructions are listed in the table below. Any logical combination of bits within this group can be combined into one microinstruction. For example, it is possible to assign ones to bits 5, 6, and 11; but it is not logical to assign ones to bits 8 and 9 simultaneously since they specify conflicting operations. If ral or rar is specified, neither cma or cml may be specified, and conversely. If rtl or rtr is specified, neither cma, cml, or iac may be specified, and conversely.

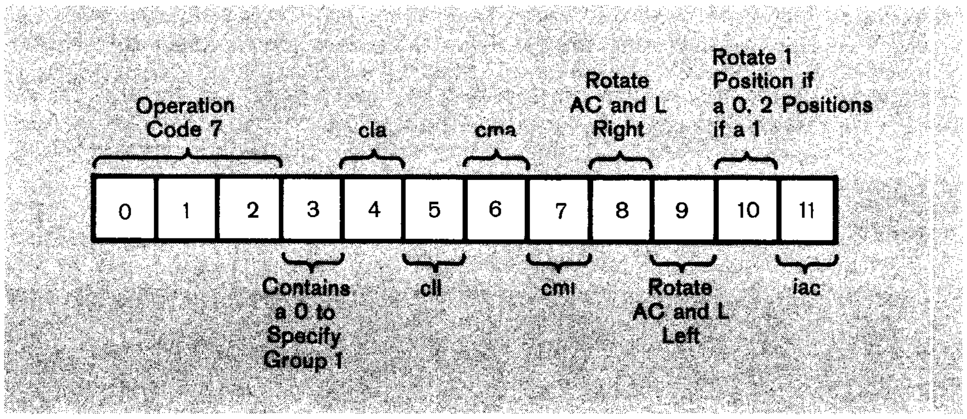


Figure 7 Group 1 Operate Instruction Bit Assignments

GROUP 1 OPERATE MICROINSTRUCTIONS

Mnemonic Symbol	Octal Code	Event Time	Operation
cla	7200	1	Clear AC. To be used alone or in opr 1 combinations. $0 => C(AC)$.
cll	7100	1	Clear L. $0 => C(L)$.
cma	7040	2	Complement AC. The C(AC) are set to the ones complement of C(AC). $C(\overline{AC}) => C(AC)$.
cml	7020	2	Complement L. $C(\overline{L}) => C(L)$.
rar	7010	2	Rotate AC and L right. The C(AC) and the C(L) are rotated right one place. $C(AC)_i => C(AC)_{i+1}$ $C(AC)_{11} => C(L)$ $C(L) => C(AC)_0$
ral	7004	2	Rotate AC and L left. The C(AC) and the C(L) are rotated left one place. $C(AC)_i => C(AC)_{i-1}$ $C(AC)_0 => C(L)$ $C(L) => C(AC)_{11}$
rtr	7012	2,3	Rotate two places to the right. Equivalent to two successive rar operations.

Mnemonic Symbol	Octal Code	Event Time	Operation
ral	7004	2	Rotate AC and L left. The C(AC) and the C(L) are rotated left one place. $C(AC)_i => C(AC)_{i-1}$ $C(AC)_0 => C(L)$ $C(L) => C(AC)_{11}$
rtr	7012	2,3	Rotate two places to the right. Equivalent to two successive rar operations.
rtl	7006	2,3	Rotate two places to the left. Equivalent to two successive ral operations.
iac	7001	3	Index AC. The C(AC) are incremented by one in twos complement arithmetic. $C(AC) + 1 => C(AC)$.
nop	7000	—	No operation. Causes a 12μsec program delay.

Group 2 operate microinstruction format is shown in Figure 8 and the microinstructions are listed in the table below. Any logical combination of bits within this group can be composed in one microinstruction.

If skips are combined in a single instruction, the inclusive OR of the conditions determines the skip. For example, if ones are designated in bits 6 and 7 (sza and snl), the next instruction is skipped if either C(AC) = 0, or C(L) = 1, or both. The cla microinstruction from group 1 can be combined with group 2 commands. This command occurs at event time 2 with respect to the event times listed in the following table.

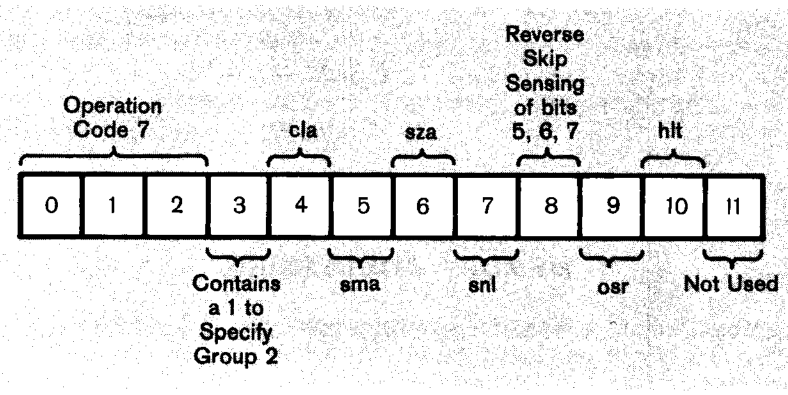

Operation Code 7
cla
sza
Reverse Skip Sensing of bits 5, 6, 7
hlt
0
1
2
3
4
5
6
7
8
9
10
11
Contains a 1 to Specify Group 2
sma
snl
osr
Not Used

Figure 8 Group 2 Operate Instruction Bit Assignments

GROUP 2 OPERATE MICROINSTRUCTIONS

Mnemonic Symbol	Octal Code	Event Time	Operation
cla	7600	2	Clear AC. To be used alone or in opr 2 combinations. $0 => C(AC)$
spa	7510	1	Skip on positive AC. If the C(AC) is a positive number, the next instruction is skipped. If $C(AC)_0 = 0$, then $C(PC) + 1 => C(PC)$
sma	7500	1	Skip on minus AC. If the C(AC) is a negative number, the next instruction is skipped. If $C(AC)_0 = 1$, then $C(PC) + 1 => C(PC)$
sna	7450	1	Skip on non-zero AC. If $C(AC) \neq 0$, then $C(PC) + 1 => C(PC)$
sza	7440	1	Skip on zero AC. If $C(AC) = 0$, the $C(PC) +1 => C(PC)$
szl	7430	1	Skip on zero L. If $C(L) = 0$, the next instruction is skipped. If $C(L) = 0$, the $C(PC) + 1 => C(PC)$
snl	7420	1	Skip on non-zero L. If $C(L) = 1$, then $C(PC) + 1 => C(PC)$
skp	7410	1	Skip, unconditional. The next instruction is skipped. $C(PC) + 1 => C(PC)$
osr	7404	3	OR with switch register. $C(SR) \vee C(AC) => C(AC)$
hlt	7402	3	Halt. Stops the program. If this instruction is combined with others in the opr 2 group, the computer stops immediately after completion of the cycle in process.

Programming

MEMORY ADDRESSING

The following terms are used in memory address programming:

Term	Definition
Page	A block of 128 core memory locations (200_8 addresses).

Term	Definition
Current Page	The page containing the instruction being executed; as determined by bits 0 through 5 of the program counter.
Page Address	An 8-bit number contained in bits 4 through 11 of an instruction which designates one of 256 core memory locations. Bit 4 of a page address indicates that the location is in(the current page when a 1, or indicates it is in) page 0 when a 0. Bits 5 through 11 designate one of the 128 locations in the page determined by bit 4.
Absolute Address	A 12-bit number used to address any location in core memory.
Effective Address	The address of the operand. When the address of the operand is in the current page or in page 0, the effective address is a page address. Otherwise, the effective address is an absolute address stored in the current page or page 0 and obtained by indirect addressing.

Organization of the core memory is summarized as follows:

Total locations (decimal)	1024	4096
Total addresses (octal)	0-1777	0-7777
Number of pages (decimal)	8	32
Page designations (octal)	0-7	0-37
Number of locations per page (decimal)	128	128
Addresses within a page (octal)	0-177	0-177

Four methods of obtaining the effective address are used as specified by combinations of bits 3 and 4.

Bit 3	Bit 4	Effective Address
0	0	The operand is in page 0 at the address specified by bits 5 through 11.
0	1	The operand is in the current page at the address specified by bits 5 through 11.
1	0	The absolute address of the operand is taken from the contents of the location in page 0 designated by bits 5 through 11.
1	1	The absolute address of the operand is taken from the contents of the location in the current page designated by bits 5 through 11.

The following example indicates the use of bits 3 and 4 to address any location in core memory. Suppose it is desired to add the contents of locations A, B, C, and D to the contents of the accumulator by means of a routine stored in page 2. The instructions in this example indicate the operation code, the contents of bit 4, the contents of bit 3, and a 7-bit address. This routine would take the following form:

Page 0		Page 1		Page 2		
Location	Contents	Location	Contents	Location	Contents	Remarks
				R	tad 00 A	Direct to data in page 0
				S	tad 01 B	Direct to data in same page
				T	tad 10 M	Indirect to address specified in page 0
				U	tad 11 N	Indirect to address specified in same page
					•	
					•	
					•	
					•	
A	xxxx	C	xxxx	B	xxxx	
M	C	D	xxxx	N	D	

Routines, using 128 instructions or less, can be written in one page using direct addresses for looping and using indirect addresses for data stored in other pages. When planning the location of instructions and data in core memory, remember that the following locations are reserved for special purposes:

Address	Purpose
0_8	Is the program counter.
1_8	Stores the contents of the program counter following a program interrupt.
2_8	Stores the first instruction to be executed following a program interrupt.
10_8 through 17_8	Auto-indexing.

INDIRECT ADDRESSING: When indirect addressing is specified, the address part (bits 5-11) of a memory reference instruction is interpreted as the address of a location containing not the operand, but the address of the operand. Consider the instruction tad A. Normally, A is interpreted as the address of the location containing the quantity to be added to the AC. Thus, if location 100 contains the number 5432, the instruction tad 100 causes the quantity 5432 to be added to the AC. Now suppose that location 5432 contains the number 6543. The instruction tad i 100 (where i signifies indirect addressing) causes the computer to take the number 5432, which is in location 100, as the effective address of the instruction and the number in location 5432 as the operand. Hence, this instruction results in the quantity 6543 being added to the contents of the AC.

AUTO-INDEXING: When a location between 10_8 and 17_8 in page 0 is specified as the address in an instruction, and bit 3 is a 1, the contents of that location are read, incremented by one rewritten in the same location, and then taken as the effective address of the instruction. This feature is called auto-indexing. If location 12_8 contains the number 5432 and the instruction dca i 12 is given, the contents of the accumulator are deposited in core memory location 5433, and the number 5433 is stored in location 12.

STORING AND LOADING

Data is stored in any core memory location by use of the dca Y instruction. This instruction clears the AC to simplify loading of the next datum. If the data deposited is required in the AC for the next program operation, the dca must be followed by a tad Y for the same address.

All loading of core memory information into the AC is accomplished by means of the tad Y instructions, preceded by an instruction that clears the AC such as cla or dca.

PROGRAM CONTROL

Transfer of program control to any core memory location uses the jmp or jms instructions. The jmp i (indirect address, 1 in bit 3) is used to address any location in core memory which is not in the current page or page 0.

The jms Y is used to enter a subroutine which starts at location Y+1. The C(PC) + 1 => C(Y) and Y + 1 => C(PC). To exit a subroutine the last instruction is a jmp i Y, which returns a program control to C(Y).

Since the program counter is in core memory location 0, the program flow can be altered by depositing some number in location 0. If the number X is deposited in 0, the next instruction is taken from location X+1.

INDEXING OPERATIONS

The isz instruction is used to count repetitive program operations without disturbing the contents of the accumulator. Counting is performed by storing a twos complement negative number equal to the number of program loops to be counted. Each time the operation is performed, the isz instruction is used to increment the contents of this stored number and check the result. When the stored number becomes zero, C(Y) = 0, the specified number of operations have occurred and the program skips out of the loop and back to the main sequence.

This instruction is also used for other routines in which the contents of a memory location are incremented without disturbing the contents of the accumulator, such as storing information from an I/O device in sequential memory locations or using core memory locations to count I/O device events.

LOGIC OPERATIONS

The PDP-5 instruction list includes the logic instruction, and Y. From this instruction short routines can be written to perform the inclusive and exclusive OR operations.

LOGIC AND: The logic AND operation between the contents of the Accumulator and the contents of a core memory location Y is performed directly by means of the and Y instruction.

INCLUSIVE OR: Assuming value A is in the AC and value B is stored in a known core memory address, the following sequence performs the inclusive OR. The sequence is stated as a utility subroutine called ior.

```
/calling sequence                              jms ior
/                                              (address of B)
/                                              (return)
/enter with argument in AC; exit with logical result in AC

          ior,      0
                    dca tem 1
                    tad i ior
                    dca tem2
                    tad i tem2
                    cma
                    and tem1
                    tad i tem2
                    isz ior
                    jmp i ior
          tem1,     0
          tem2,     0
```

EXCLUSIVE OR: The exclusive OR operation for two numbers, A and B, can be performed by a subroutine called by the mnemonic code xor. In the following general purpose xor subroutine, the value A is assumed to be in the AC, and the address of the value B is assumed to be stored in a known core memory location.

```
/calling sequence                              jms xor
/                                              (address of B)
/                                              (return)
/enter with argument in AC; exit with logical result in AC

        xor,     0
                 dca tem1
                 tad i xor
                 dca tem2
                 tad tem1
                 and i tem2
                 cma V iac
                 cll V ral
                 tad tem1
                 tad i tem2
                 isz xor
                 jmp i xor
        tem1,    0
        tem2,    0
```

An xor subroutine can be written using fewer core memory locations by making use of the ior subroutine; however, such a subroutine requires longer to execute. A faster xor subroutine can be written by storing the value B in the second instruction of the calling sequence instead of the address of B; however, the resulting subroutine is not as utilitarian as the routine given here.

ARITHMETIC OPERATIONS

One arithmetic instruction is included in the PDP-5 order code, the twos complement add: tad Y. Using this instruction, routines can easily be written to perform addition, subtraction, multiplication, and division in twos complement arithmetic.

TWOS COMPLEMENT ARITHMETIC: In twos complement arithmetic, addition, subtraction, multiplication, and division of binary numbers is performed in accordance with the common rules of binary arithmetic. In PDP-5, as in other machines utilizing complementation techniques, negative numbers are represented as the complement of positive numbers, and subtraction is achieved by complement addition. Representation of negative values in ones complement arithmetic is slightly different from that in twos complement arithmetic.

The ones complement of a number is the complement of the absolute positive value; that is, all ones are replaced by zeros and all zeros are replaced by ones. The twos complement of a number is equal to the ones complement of the positive value plus one.

In ones complement arithmetic a carry from the sign bit (most significant bit) is added to the least significant bit in an end-around carry. In twos complement arithmetic a carry from the sign bit complements the link (a carry would set the link to one if it were properly cleared before the operation), and there is no end-around carry.

A ones complement representation of a negative number is always one less than the twos complement representation of the same number. Differences between ones and twos complement representations are indicated in the following list.

Number	1s Complement	2s Complement
+5	000000000101	000000000101
+4	000000000100	000000000100
+3	000000000011	000000000011
+2	000000000010	000000000010
+1	000000000001	000000000001
+0	000000000000	000000000000
−0	111111111111	Nonexistent
−1	111111111110	111111111111
−2	111111111101	111111111110
−3	111111111100	111111111101
−4	111111111011	111111111100
−5	111111111010	111111111011

Note that in twos complement there is only one representation for the number which has the value zero, while in ones complement there are two representations. Note also that complementation does not interfere with sign notation in either ones complement or twos complement arithmetic; bit 0 remains a 0 for positive numbers and a 1 for negative numbers.

To form the twos complement of any number in the PDP-5, the ones complement is formed, and the result is incremented by one. This is accomplished by the instruction cma combined with an iac instruction. Since both of these instructions are functions of the opr 1 instruction and the actions occur at different event times, they can be combined to form:

cia — 7041 — 2, 3   Complement and index AC

ADDITION: The addition of a number contained in a core memory location and the number contained in the accumulator is performed directly by using the tad Y instruction, assuming that the binary point is in the same position and that both numbers are properly represented in twos complement arithmetic. Addition can be performed without regard for the sign of either the augend or the addend. Overflow is possible, in which case the result will have an incorrect sign, although the 11 least significant bits will be correct.

SUBTRACTION: Subtraction is performed by complementing the subtrahend and adding the minuend. As in addition, if both numbers are represented by their twos complement, subtraction can be performed without regard for the signs of either number. Assuming that both numbers are stored in core memory, a routine to find the value of A-B follows:

```
cla
tad B              /Load subtrahend into AC
cia                /Complement and increment B (cma V iac)
tad A              /C(AC) = A - B
```

CHAPTER 3
INPUT-OUTPUT CONTROL

Functions

Selected input-output devices are controlled by iot (in-out transfer) instructions. The iot instruction is microprogrammed to allow one basic instruction to handle many devices (by changing the bits of the command). The command pulses occur at various times to allow flags to be sampled (and an instruction skipped), buffers to be cleared, and data to be transmitted to or from the accumulator. Operational circuits of the input-output control element are shown in Figure 9.

DEVICE SELECTOR (DS)

Input-output equipment connected into the system is controlled by various Device Selector pulses. These pulses:

a. Sample Device flag conditions which are fed into the input-output skip facility.
b. Reset external register.
c. write information into external registers from the AC output.
d. Read information from external register into the AC.
e. Control the I/O device.
f. Halt the computer until the external device has finished its operation.

The iot instruction causes the arithmetic and control element to produce IOP pulses based on the contents of bits 9 through 11 of the instruction. These pulses are designated IOP 4, 2, and 1, respectively, and occur at 1-microsecond intervals, which are identified as event times. Binary ones in the instruction word cause the IOP pulses to be generated as follows:

Instruction Bit	IOP Pulse	IOT Pulse	Event Time	Computer Cycle Time
11	IOP 1	IOT 1	1	T4
10	IOP 2	IOT 2	2	T5
9	IOP 4	IOT 4	3	T6

A device selector module exists for each I/O device or external register to be addressed separately. The DS is a gating element which receives both the 1 and 0 information from bits 3 through 8 of an instruction (MB_{3-8}) and the IOP

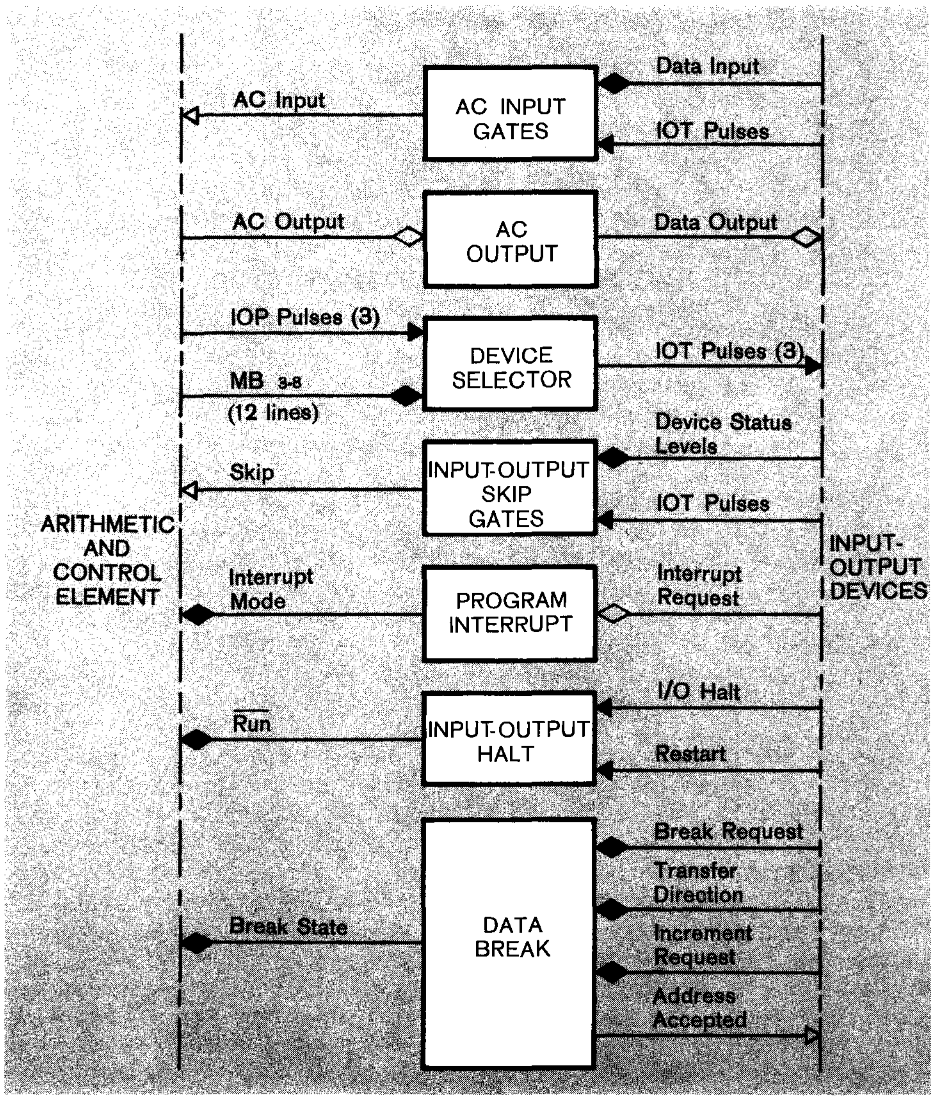


Figure 9 Input-Output Control Element

pulses. Each DS is wired to pass IOP pulses to a specific I/O device only when the I/O device selection bits are set to the code which specifies operation of the associated I/O device. The gated IOP pulses at the output of a DS are called IOT pulses and can be used to set or reset control flip-flops, gate information into the AC from external registers, gate information into external registers from the AC, or skip instructions.

ACCUMULATOR INPUT

Capacitor-diode gates are provided at the inputs to the accumulator to allow gated information to be written into the PDP-5 from several sources. Information levels from 12 separate bits of an external register can be simultaneously set into the AC by an IOT pulse. The AC must be clear at the time information is written in. Information pulses supplied to the AC input bus must drive it to ground potential to write a 1 in an accumulator bit.

ACCUMULATOR OUTPUT

A static level is available at an output bus from each bit of the accumulator. These static levels are ground potential for a binary 1 and −3 volts for a binary 0. Data supplied to an external register is strobed into it by means of IOT pulses.

INPUT-OUTPUT SKIP (IOS)

The IOS facility allows the program to skip (or branch) according to the condition of various external devices. An IOT pulse is used to strobe the external device, such as a flag, and sample its state. If the gating of the Device flag and IOT pulse drives the IOS bus to ground, the instruction following the iot instruction which issued the strobe is skipped. If the input is a −3 volt potential, the program sequence is not altered.

PROGRAM INTERRUPT

The program interrupt feature allows certain external conditions to interrupt the computer program. It is used to speed the information processing of input-output devices or to allow certain alarms to halt the program in progress and initiate another routine. When a program interrupt request is made, the computer completes execution of the instruction in progress before entering the interrupt mode. A program interrupt is similar to a jms to location 1; that is, the contents of the program counter are stored in location 1, and the program resumes operation in location 2. The interrupt program commencing in location 2 is responsible for finding the signal causing the interruption, for removing the condition, and for returning to the original program. Exit from the interrupt program, back to the original program, can be accomplished by a jmp i 1 instruction.

INPUT-OUTPUT HALT (IOH)

The input-output halt facility allows the computer to be halted during the time that external devices are operating and then restarted by a pulse from the device. The IOH state occurs when Type 137 Analog-to-Digital Converter is operating and may be wired to occur during the operation time of any other device.

A specific iot instruction initiates operation of an I/O device. The I/O device supplies an I/O-Halt pulse to the IOH that inhibits program advance. When the I/O device completes the programmed operation, it produces a Restart pulse which is received by the IOH to clear the IOH mode and to allow program advance to the next instruction.

DATA BREAK

This facility allows transmission of data directly between an external device and core memory, via the memory buffer register. During a data break, the program is halted but the contents of the accumulator, instruction register, and program counter are not disturbed. Therefore, when a data transfer is complete, the program resumes from exactly the same condition which existed before the break.

Data breaks require receipt of three control signals: Break Request, Transfer Direction, and Increment Request. When a Break Request signal is received from an I/O device, the computer completes execution of the instruction in progress and then enters the data break mode. If a jmp or jms instruction is in progress when the request is received, the current instruction is completed, and the next instruction is performed before the break is instituted. The direction of transfer and the core memory address of each transferred word are specified by the I/O device when the break request is made. The Transfer Direction signal controls the read or write cycle of the computer, and the address is set directly into the memory address register. Data transfer then takes place between the memory buffer register and the I/O device. When the transfer is completed, the I/O device signals the computer to leave the break mode by removing the Break Request. If additional transfers are to occur, a new address must be specified to the memory address register or an Increment Request signal must be supplied to transfer data at sequential core memory location. Figure 10 indicates the timing of these signals. The levels of these signals are:

Signal	−3 Volts	0 Volts
Break Request	No request	Request Break
Transfer Direction	Data into PDP-5	Data out of PDP-5
Increment Request	Request increment	No request
Address	0	1
Data	0	1

Break Request, Transfer Direction, and Address Information signals should be supplied simultaneously for the first transfer. When the computer enters the data break mode, it supplies an Address Accepted pulse to the I/O device. When the direction of transfer is into the PDP-5 from the device, data must be supplied to the memory buffer register input no later than 1 microsecond after the Address Accepted pulse occurs and must be present for more than 2 microseconds. To discontinue the data break mode, the Break Request signal must be removed no later than 4 microseconds after the address accepted pulse occurs, or the computer will enter another cycle in the data break mode. The Transfer Direction signal must be present when the break request is made and cannot be changed until 4 microseconds after the Address Accepted pulse occurs. Address information must also be present when the request is made, but can be changed any time after the address is accepted. To transfer data at sequential core memory locations the first transfer address must be supplied to the memory address register by the I/O device, and successive addresses can be specified by the Increment Request signal. This signal cannot occur before 1 microsecond after the address is accepted for the first transfer and must be present no later than 4 microseconds after the address is accepted. The maximum and minimum limits of this signal timing are indicated in Figure 10.

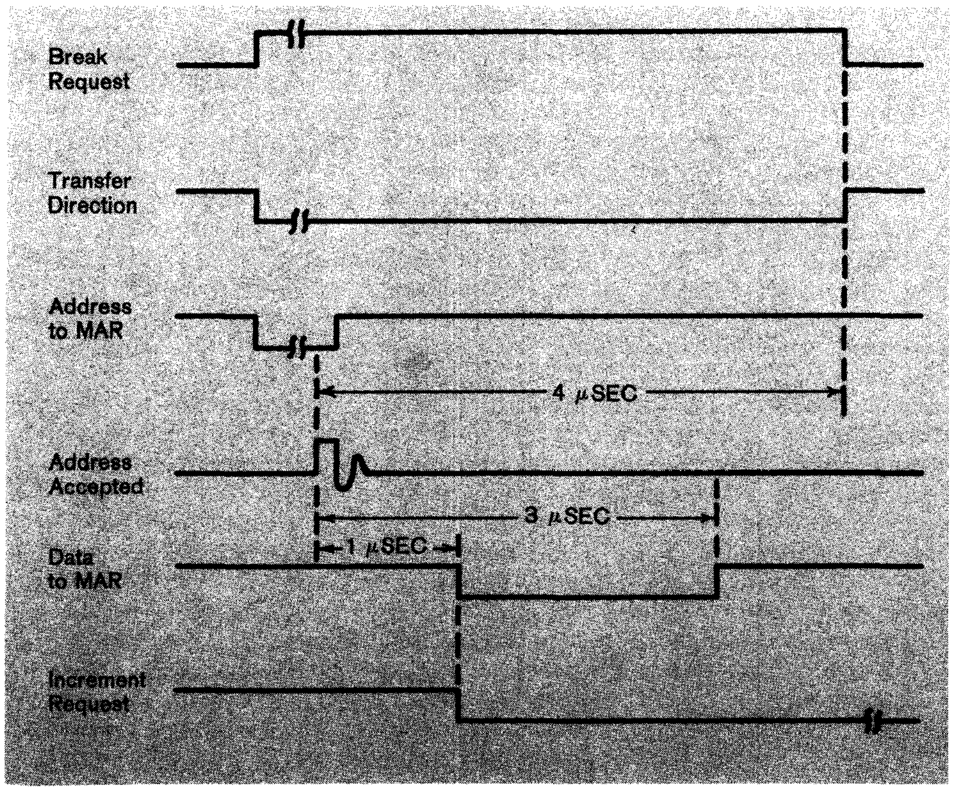


Figure 10 Data Break Timing

Instructions

Two types of instructions are associated directly with the input-output controls: those concerning the input-output skip and those concerning the program interrupt. The skip instructions are listed in Chapter 4 of this handbook with the instructions for the device whose status is checked. There are two instructions for the program interrupt. They are:

ion — 6001 —	Turn interrupt on and enable the computer to respond to an interrupt request. When this instruction is given, the computer executes the next instruction, then enables the interrupt. The additional instruction allows exit from the interrupt subroutine before allowing another interrupt to occur.
iof — 6002 —	Turn interrupt off i.e. disable the interrupt.

Programming

When an interrupt request is acknowledged, the interrupt is automatically disabled by the program interrupt circuits (not by instructions). The next instruction is taken from core memory location 2. Usually the instruction stored in location 2 is a jmp, which transfers program control to a subroutine, which services the interrupt. At some time during this subroutine an ion instruction must be given. The ion can be given at the end of the subroutine to allow other interrupts to be serviced after program control is transferred back to the original program. In this application, the ion instruction immediately precedes the last instruction in the routine. A delay of one instruction (regardless of length) is inherent in the ion instruction to allow transfer of program control back to the original program before enabling the interrupt. Usually exit from the subroutine is accomplished by a jmp i 1 instruction.

The ion can be given during the subroutine as soon as it has determined the I/O device causing the interrupt. This latter method allows the subroutine, which is handling a low priority interrupt, to be interrupted, possibly by a high priority device. Programming of an interrupt subroutine, which checks for priority and allows itself to be interrupted, must make provisions to relocate the contents of the program counter stored in location 1; so that if interrupted, the contents of the PC during the subroutine are stored in location 1, and the contents of the PC during the original program are not lost.

CHAPTER 4
INPUT-OUTPUT DEVICES

Use of the standard and optional equipment in a PDP-5 system is discussed in this section. The Teletype equipment is the only standard I/O device supplied with PDP-5. All other equipment is purchased at the option of the customer to compose a system tailored to his computing needs.

Teletype Model 33 ASR

The standard Teletype Model 33 ASR (automatic send-receive) can be used to type in or print out information at a rate of up to ten characters per second, or to read in or punch out perforated paper tape at a ten characters per second rate. Signals transferred between the 33 ASR and the keyboard printer control logic are standard serial, 11 unit code Teletype signals. The signals consist of marks and spaces which correspond to idle and bias current in the Teletype and zeros and ones in the control and computer. The start mark and subsequent eight character bits are one unit of time duration and are followed by the stop mark which is two units. Appendix 2 lists the character code for the Teletype. Punched tape format is as follows:

	Tape Channel			
	87	654	S	321
Binary Code (Punch = 1)	10	110		100
Octal Code	2	6		4

Toggle switches on the right side of the Teletype console control primary power and allow the 33 ASR to communicate with the computer in on line operation or to prepare tapes, etc., without disturbing the computer program in local operations. Separate pushbutton switches on the punch are used to control power, release the mechanism to allow insertion and removal of tape, and allow backspacing to correct errors. A three position switch on the reader advances the tape, stops the tape, or allows free wheeling of the mechanism for tape insertion and removal.

KEYBOARD/READER

The keyboard and tape reader control contains an 8-bit buffer (LUI) which assembles and holds the code for the last character struck on the keyboard or read from the tape. The Keyboard flag becomes a 1 to signify that a character has been assembled and is ready for transfer to the accumulator. When the flag is a 1, a relay contact opens to disable the reader. This flag is connected to the computer program interrupt and input-output skip facility. It is cleared by command. Instructions for use in supplying data to the computer from the Teletype are:

ksr — 6031 — Skip if Keyboard flag is a 1.
kcc — 6032 — Clear AC and clear Keyboard flag.
krs — 6034 — Read keyboard buffer static. (This is a static command in that neither the AC nor the Keyboard flag is cleared.)
$C(LUI) \vee C(AC)_{4-11} => C(AC)_{4-11}$
krb — 6036 — Clear AC, clear Keyboard flag, and read the contents of the keyboard buffer into $C(AC)_{4-11}$.

A program sequence loop to look for a Teletype keyboard or tape reader character can be written as follows:

```
200     6031     look,     ksf          /skip when LUI is full
201     5200               jmp look
202     6036               krb          /read LUI into AC
```

TELEPRINTER/PUNCH

The teleprinter and tape punch control contains an 8-bit buffer (LUO) which receives a character to be printed and/or punched from AC bits 4 through 11. The LUO receives the 8-bit code from the AC in parallel and transmits it to the teleprinter and tape punch serially. When the last bit has been transmitted, the Teleprinter flag is set to 1. This flag is connected to the computer program interrupt and input-output skip facility. It is cleared by programmed command. The instruction list for printing or punching is:

tsf — 6041 — Skip if Teleprinter flag is a 1.
tcf — 6042 — Clear Teleprinter flag.
tpc — 6044 — Load the LUO from the $C(AC)_{4-11}$ and print and/or punch the character.
tls — 6046 — Load the LUO from the $C(AC)_{4-11}$, clear the Teleprinter flag, and print and/or punch the character.

A program sequence loop to print and/or punch a character when the LUO is free can be written as follows:

```
free,     tsf          /skip when free
          jmp free
          tls          /load LUO, print or punch
```

High Speed Perforated Tape Reader and Control Type 750

This device senses 8-hole perforated paper or Mylar tape photoelectrically at 300 characters per second. The reader control requests reader movement, transfers data from the reader into the reader buffer (RB), and signals the computer when incoming data is present. Reader tape movement is started by a reader control request to simultaneously release the brake and engage the clutch. The 8-bit reader buffer sets the Reader flag to 1 when it has been filled from the reader and transfers data into bits 4 through 11 of the accumulator under computer command. The Reader flag is connected to the computer program interrupt and input-output skip facility. It is cleared by IOT pulses. Computer instructions for the reader are:

rsf — 6011 — Skip if Reader flag is a 1.
rrb — 6012 — Read the contents of the reader buffer and clear the Reader flag. (This instruction does not clear the AC.)
$C(RB) \vee C(AC)_{4-11} => C(AC)_{4-11}$
rfc — 6014 — Clear Reader flag and reader buffer, fetch one character from tape and load it into the reader buffer, and set the Reader flag when done.

A program sequence loop to look for a reader character can be written as follows:

```
              rfc          /fetch character from tape
look,         rsf          /skip when RB full
              jmp look
              cla
              rrb          /load AC from RB
```

High Speed Perforated Tape Punch and Control Type 75A

The Teletype BRPE paper tape punch perforates 8-hole tape at 63.3 characters per second. Information to be punched on a line of tape is loaded on an 8-bit punch buffer (PB) from the AC bits 4 through 11. The Punch flag becomes a 1 at the completion of punching action, signaling that new information may be read into punch buffer (PB) (and punching initiated). The Punch flag is connected to the computer program interrupt and input-output skip facility. The punch instructions are:

psf — 6021 — Skip if Punch flag is a 1.
pcf — 6022 — Clear Punch flag and punch buffer.
ppc — 6024 — Load the punch buffer from bits 4 through 11 of the AC and punch the character. (This instruction does not clear the Punch flag or punch buffer.)
$C(AC)_{4-11} \vee C(PB) => C(PB)$
pls — 6026 — Clear the Punch flag, clear the punch buffer, load the punch buffer from the contents of bits 4 through 11 of the accumulator, punch the character, and set the Punch flag to 1 when done.

A program sequence loop to punch a character when the punch buffer is "free" can be written as follows:

```
free,         psf          /skip when free
              jmp free
              pls          /load PB from AC and punch character
```

Analog-To-Digital Converter Type 137

This converter operates in the conventional successive approximation manner, using the memory buffer register as a shift register and using the accumulator as the buffer register. An IOT pulse from the device selector starts the conversion and initiates an input-output halt. At the end of the conversion the converter produces a Restart pulse which is supplied to the input-output halt facility. At this time the digital equivalent of the Analog Input signal is contained in the accumulator as a 12-bit binary number. Insignificant magnitude bits can be rotated out of the AC by an instruction such as 7110 (rar and cll).

To save program running time, the converter should be adjusted to provide only the accuracy required by the program application. Instructions for adjusting the accuracy are given in the maintenance manual covering Type 137. Maximum error of the converter is equal to the switching point error plus the quantization error. Maximum quantization error is equal to the least significant bit. Switching point error and total conversion time are functions of the adjusted accuracy of the converter.

Adjusted Bit Accuracy	Switching Point Error	Conversion Time per Bit (in μsec)	Total Conversion Time (in μsec)
6	±1.6%	3.5	24.5
7	±0.8%	4.0	32.0
8	±0.4%	4.5	40.5
9	±0.2%	5.0	50.0
10	±0.1%	6.0	66.0
11	±0.05%	11.0	132.0

There is only one instruction associated with the converter:

adc — 6004 — Convert the Analog Input signal to a digital value.

Card Reader and Control Type 451

The control of the card reader differs from the control of other input devices, in that the timing of the read-in sequence is dictated by the device. Once the command to fetch a card is given, the card reader reads all 80 columns of information in sequence. To read a column, the program must respond to a flag set as each new column is started. The instruction to read the column must come within 2.3 milliseconds. The commands for the card reader are:

crsf — 6632 —	Skip if Card Reader flag is a 1. If a card column is present for reading, the next instruction is skipped.
cers — 6634 —	Card equipment read status. Reads the status of the Card Reader flag and status levels into bits 6 through 9 of the AC.
crrb — 6671 —	Read the card column buffer information into the AC and clear the Card Reader flag. One crrb reads alphanumeric information. Two crrb instructions read the upper and lower column binary information.

crsa	6672	Select a card in alphanumeric mode. Select the card reader and start a card moving. Information appears in alphanumeric form.
crsb	6674	Select a card in binary mode. Select the card reader and start a card moving. Information appears in binary form.

Upon instruction to read the card reader buffer, 6 information bits are placed into AC bits 6 through 11. Alphanumeric (or Hollerith) information on the card is encoded or represented with these six bits. The binary mode enables the 12 bits (or rows) of each column to be obtained. The first read buffer instruction transfers the upper six rows (Y, X, 0, 1, 2, and 3); the second instruction transfers the lower six rows (4, 5, 6, 7, 8, and 9). The mode is specified with the card read select instruction. The mode can be changed while the card is being read.

Card Punch Control Type 450

The card punch dictates the timing of a read-out sequence, much as the card reader controls the read-in timing. Once a card leaves the hopper, all 12 rows are punched at intervals of 40 milliseconds. Punching time for each row is 24 milliseconds, leaving 16 milliseconds to load the buffer for the next row. A flag indicates that the buffer is ready to be loaded. The commands for the card punch control are:

cpsf	6631	Skip if Card Punch flag is a 1. The Card Punch flag indicates the punch buffer is available, and should be loaded.
cers	6634	Card equipment read status. Reads the status of the Card Punch flag and the Card Punch error level into the contents of bits 10 and 11 of the AC, respectively.
cpcf	6641	Clear Card Punch flag.
cpse	6642	Select the card punch. Transmit a card to the 80-column punch die from the hopper.
cplb	6644	Load the card punch buffer from the C(AC). Seven load instructions must be given to fill the buffer.

Since 12 bits are transmitted with each iot instruction, 7 iot instructions must be issued to load the 80-bit row buffer. The first six loading instructions fill the first 72 bits (or columns); the seventh loads the remaining 8 bits of the buffer from AC bits 4 through 11.

After the last row of punching is complete, 28 milliseconds are available to select the next card for continuous punching. If the next card is not requested in this interval, the card punch will stop. The maximum rate of the punch is 100 cards per minute in continuous operation. A delay of 1308 milliseconds follows the command to select the first card; a delay of 108 milliseconds separates the punching of cards in continuous operation.

The Card Punch flag is connected to the program interrupt and to bit 10 of the cers instruction. Faults occurring in the punch are detected by status bit 11 of the cers and signify the punch is disabled, the stacker is full, or the hopper is empty.

A program sequence to punch 12 rows of data on a card can be written as follows, assuming the data to be punched in each row is stored in seven consecutive core memory locations beginning in location 100.The program begins in register pnch.

```
pnch,       cpse          /select the card
            cla
            tad loc       /initialize the card image
            dca 10
            tad rcnt
            dca tem1      /initialize the row counts, 12
lp1,        cla
            tad gpct      /initialize the 7 groups per row
            dca tem2
            cpsf          /sense punch load availability
            jmp .-1
lp2,        cla
            tad i 10      /7 groups of 12 bits per row
            cplr          /load buffer command
            isz tem2
            jmp lp2
            isz tem1      /test for 12 rows
            jmp lp1
            hlt           /end punching 1 card
loc,        77            /location of card image
rcnt,       -14           /12 rows per card
gpct,       -7            /7 groups per row
tem1,       0             /row counter
tem2,       0             /group counter
```

Automatic Line Printer And Control Type 64

The line printer can print 300 lines of 120 characters per minute. Each character is selected, from a set of 63 available, by means of a 6-bit binary code (Appendix 2 lists the character specified by each code). Each 6-bit code is loaded separately into a printing buffer from bits 6 through 11 of the AC. The printing buffer is divided into two sections; each section can hold 120 codes. Therefore, 120 load instructions can be given to fill one section of the printing buffer. A print command causes the characters specified by the contents of the print buffer section last loaded to be printed on one line. As printing is in progress, the alternative section of the printing buffer can be reloaded. After the last character in a line is printed, the section of the printing buffer from which characters were just printed is cleared automatically. The section of the printing buffer that is loaded and printed is alternated automatically within the printer and is not program specified.

A 3-bit format register within the printer is loaded from bits 9 through 11 of the AC during a print command. This register selects one of eight channels of a perforated tape to control spacing of the paper. The tape moves in synchronism with the paper until a hole is sensed in the selected channel to halt paper advance. A recommended tape has the following characteristics:

Channel	Spacing
0	1 line
1	2 lines
2	3 lines
3	¼ page
4	½ page
5	¾ page
6 or 7	top of form

Loading of a 6-bit code into the printing buffer requires approximately 1.6 milliseconds. When the transfer of a code is completed, the Line Printed flag rises to indicate that the printer is ready to receive another code. The Line Printer flag is connected to the program interrupt facility.

The iot microinstructions which command the line printer are:

lcf — 6652 — Clear Line Printer flag.
lpr — 6655 — Clear the format register, load the format register from the $C(AC)_{9-11}$, print the line contained in the section of the printer buffer loaded last, and advance the paper in accordance with the selected channel of the format tape if the $C(AC)_8 = 1$. If the $C(AC)^8 = 0$, the line is printed and paper advance is inhibited.
lsf — 6661 — Skip if Line Printer flag is a 1.
lcb — 6662 — Clear both sections of the printing buffer.
lld — 6664 — Load printing buffer from the $C(AC)_{6-11}$.

The following routine demonstrates the use of these commands in a sequence which prints an unspecified number of 120-character lines. This sequence assumes that the printer is not in operation, that the paper is manually positioned for the first line of print, and that one-character words are stored in sequential core memory locations beginning at 2000. The "print" location starts the routine.

```
print,     lcb            /initialize printing buffer
           cla
           tad loc        /load initial character address
           dca 10         /store in auto-index register
lrpt,      tad cnt        /initialize character counter
           dca temp
loop,      lsf            /wait until printing buffer ready
           jmp loop
           tad i 10       /load AC from current character address
           lld            /load printing buffer
           cla
           isz temp       /test for 120 characters loaded
           jmp loop
           tad frm        /load spacing control and
           lpr            /print a line
           cla            /ready for next line
           jmp lrpt       /jump to print another line
loc,       1777           /initial character address —1
cnt,       —170           /character counter
```

```
temp,        0            /current character address
frm,         10           /spacing control and format
```

Oscilloscope Display Type 34B

Type 34B is a two axis digital-to-analog converter and an intensifying circuit, which provides the Deflection and Intensify signals needed to plot data on an oscilloscope. Coordinate data is loaded into an X buffer (XB) or a Y buffer (YB) from bits 2 through 11 of the accumulator. The binary data in these buffers is converted to a −10 to 0 volt Analog Deflection signal. The 30 volt, 10-microsecond Intensify signal is connected to the grid of the oscilloscope CRT. Points can be plotted at approximately a 25-kilocycle rate. The instructions for this display are:

dcx — 6051 — Clear X coordinate buffer.
dxl — 6053 — Clear and load X coordinate buffer.
$C(AC)_{2-11} => C(XB)$
dcy — 6061 — Clear Y coordinate buffer.
dyl — 6063 — Clear and load Y coordinate buffer.
$C(AC)_{2-11} => C(YB)$
dix — 6054 — Intensify the point defined by the contents of the X and Y coordinate buffers.
diy — 6064 — Intensify the point defined by the contents of the X and Y coordinate buffers.
dxs — 6057 — Executes the combined functions of dxl followed by dix.
dys — 6067 — Executes the combined functions of dyl followed by diy.

The following program sequence to display a point begins at location 200, and assumes that the X and Y coordinate data is stored in absolute addresses 176 and 177.

```
176             X,
177             Y,
200     7200    beg,        cla
201     1176                tad X       /load AC with X
202     6053                dxl         /clear and load XB
203     7200                cla
204     1177                tad Y       /load AC with Y
205     6067                dys         /clear and load YB, and display point
```

Precision CRT Display Type 30N

Type 30N functions are similar to those of the Type 34B Oscilloscope Display in plotting points on a self-contained 16-inch cathode-ray tube. A 3-bit brightness register is contained in Type 30N to control the amplitude of the Intensify signal supplied to the CRT. This register is loaded by jam transfer (transfer ones and zeros so that clearing is not required) from the AC by the instruction:

dlb — 6074 — Load brightness register (BR) from bits 9 through 11 of the AC.
$C(AC)_{9-11} => C(BR)$

All other instructions and the instruction sequence are similar to those used in the Type 34B.

Light Pen Type 370

The light pen is a photosensitive device which detects the presence of information displayed on a CRT. If the light pen is held in front of the CRT at a point displayed, the Display flag will be set to a 1. The commands are:

dsf — 6071 — Skip if Display flag is a 1.
dcf — 6072 — Clear the Display flag to a 0.

The Display flag is connected to the input-output skip facility, and to the program interrupt.

Incremental Plotter and Control Type 350

Four models of California Computer Products Digital Incremental Recorder can be operated from a DEC Type 350 Incremental Plotter Control. Characteristics of the four recorders are:

Model	Step Size (inches)	Speed (steps/minute)	Paper Width (inches)
563	0.01	12,000	31
564	0.005	18,000	31
565	0.01	18,000	12
566	0.005	18,000	12

The principles of operation are the same for each of the four models of Digital Incremental Recorders. Bidirectional rotary step motors are employed for both the X and Y axes. Recording is produced by movement of a pen relative to the surface of the graph paper, with each instruction causing an incremental step. X-axis deflection is produced by motion of the drum; Y-axis deflection, by motion of the pen carriage. Instructions are used to raise and lower the pen from the surface of the paper. Each incremental step can be in any one of eight directions through appropriate combinations of the X and Y axis instructions. All recording (discrete points, continuous curves, or symbols) is accomplished by the incremental stepping action of the drum and carriage. Front panel controls permit single-step or continuous-step manual operation of the drum and carriage, and manual control of the pen solenoid. The recorder and control are connected to the computer program interrupt and input-output skip facility.

Instructions for the recorder and control are:

plsf — 6501 — Skip if Plotter flag is a 1.
plcf — 6502 — Clear Plotter flag.
plpu — 6504 — Plotter pen up. Raise pen off of paper.
plpr — 6511 — Plotter pen right.
pldu — 6512 — Plotter drum (paper) upward.

pldd	6514	Plotter drum (paper) downward.
plpl	6521	Plotter pen left.
pldu	6522	Plotter drum (paper) upward. (Same as 6512.)
plpd	6524	Plotter pen down. Lower pen on to paper.

Program sequence must assume that the pen location is known at the start of a routine since there is no means of specifying an absolute pen location in an incremental plotter. Pen location can be preset by the manual controls on the recorder. During a subroutine, the PDP-5 can track the location of the pen on the paper by counting the instructions that increment the pen and the drum.

Automatic Magnetic Tape Control Type 57A

This control, operating through interface logic, such as Type 520, 521, or 522, transfers information between PDP-5 and up to eight tape transports. Data transmission format is compatible with IBM high and low densities (800-556 and 200 characters per inch, respectively) in either BCD or binary parity modes. Transports can be DEC Type 50 or Type 570, or IBM Types 729 II, IV, V, VI, or (with certain restrictions) the 7330. The transports are capable of operating at the following densities: 200 cpi only, Type 50; 200 and 556 cpi only, Type 570 or IBM Types 729 II and 7330; all three densities, IBM Type 729 V.

The following functions are controlled by various combinations of iot instructions:

Write
Write End of File
Write Blank Tape
Read
Read Compare
Space Forward
Space Backward
Rewind
Rewind/Unload
Write Continuous
Read Continuous

Tape transport motion is governed by one of two control modes: normal, in which tape motion starts upon command and stops automatically at the end of the record; and continuous, in which tape motion starts on command and continues until stopped by the program as a function of synchronizing flags if status conditions appear.

All data transfers are under control of the PDP-5 data break facility; and commands issued during a transfer control, operate, and monitor Type 57A functions by means of the PDP-5 program interrupt facility. Assembled, 12-bit, PDP-5, data words pass between the computer MBR and the control final data buffer register. The core memory address of each word transferred is specified to the computer MAR by the control current address register. Use of the program interrupt facility allows the main computer program to continue during long tape operations without running in a loop which waits for Tape flags. The program interrupt subroutine for Type 57A loads the AC with numbers, then issues iot instructions to the control. Specific tape control modes are interpreted from the contents of the AC during some iot instructions. In addition, the current address (CA) register and the word count (WC) registers of the control are loaded from the AC.

Tape functions can be monitored by the program either during or at the end of an operation. They can be altered during operation to a limited degree. The control senses for several types of possible error condition throughout an operation.The results of this sensing can be interrogated by the subroutine at any time.

Two crystal clocks are used to generate one of three character writing rates, depending on the density (200, 556, 800) specified by the programmer. In writing or reading, a composite 12-bit binary word passes between the computer and the control; that is, bits 0 through 5 constitute one tape character, and bits 6 through 11 constitute a second tape character.

In normal operation, six iot commands initiate reading or writing of one record. When the word count exceeds the number stored in the WC, the transport is stopped and the control is free for another command. In continuous operation, any number of records is written or read without the need for further transport commands except stop.

The following automatic safeguards are inherent in the design of Type 57A:

END POINT: If the end point is reached during reading or writing, the control ignores the end point and finishes the operation (ample tape is allowed).
Beyond the end point, tape commands specifying forward direction are illegal, and the tape will not respond to such commands. If the end point is passed during spacing, the transport is shut down regardless of word count.

LOAD POINT: If the load point is reached during back spacing, the transport is stopped regardless of word count. At load point, a space back command is legal, and the tape may be unloaded. When the write command is given at load point, the tape is erased 3 inches beyond the load point before writing the first record. After giving a read command at the load point, the read logic is disabled until the load point marker is past the read head before the read logic is turned on.

WRITE LOCK RING: Without the write lock ring in the tape reel, writing is illegal and the transport will not respond to a write command.

FORMAT CONTROL: If the PDP-5 halt command is given during normal reading or read comparing, the tape proceeds to the end of record, and the control shuts down the transport. If a halt is given in continuous reading or read comparing, the transport will proceed to end of tape and shut down. If a halt command is given in normal spacing, the transport will proceed to EOR and shut down. If halt is given during continuous spacing, the transport will proceed until WC overflows or until it senses a file marker, load point, or end point, then shut down.

If halt is given during writing in the normal mode, the last word to be transferred is written, the rest of the record is written as zeros, and the transport is shut down. If halt is given during writing in the continuous mode, the record is completed; then zeros are written to the end of the tape. If a WC overflow occurs during a normal read or read compare, the transport proceeds to EOR before shutting down.

The functions of Type 57A Automatic Magnetic Tape Control are controlled by combinations of the following iot instructions:

mscr — 6701 — Skip if the tape control Ready (TCR) level is 1. A 1 is added to the contents of the program counter if the tape control is free to accept a command. The TCR flag is connected to the program interrupt.

mcd — 6702 — Disable the TCR flag from the program interrupt and clear command register. Clear Word Count Overflow (WCO) flag. Clear End of Record (EOR) flag. This instruction should be immediately preceded by the two instructions cla and tad (4000) to obtain the operation indicated.

mts — 6706 — Disable the TCR flag from the program interrupt, turn off the WCO flag and EOR flag and select the unit, the mode of parity, and the density from the contents of the AC. The AC bit assignments are:

AC_1
(Type 521 and 522 interface only)
0=high sense level
1=low sense level

AC_2
0=200 or 556 density
1=800 or 556 density

AC_8
0=200 density
1=556 density

AC_2	AC_8	Density
0	0	200
0	1	556
1	0	800
1	1	556

AC_7
0 = even parity (BCD)
1 = odd parity (binary)

AC_{9-11}
These three bits select one of eight tape units, addresses 0

msur — 6711 — Skip if the tape transport is ready (TTR). The selected tape unit is checked, using this command, and must be free before the following mtc command is given.

mnc — 6712 — Terminate the continuous mode. This instruction clears the AC at completion. It should be immediately preceded by the two instructions cla and tad (4000) to obtain the operation indicated.

mtc — 6716 — Place $C(AC)_{3-6}$ in the tape control command register and start tape motion. Bit 6 selects motion mode.

AC_6
- 0 = Normal
- 1 = Continuous

AC_{3-5} are decoded as follows:
- 0 = no operation
- 1 = rewind
- 2 = write
- 3 = write end of file (EOF)
- 4 = read compare
- 5 = read
- 6 = space forward
- 7 = space backward

mswf — 6721 — Skip if the WCO flag is a 1. The flag is connected to the program interrupt.

mdwf — 6722 — Disable WCO flag.

mcwf — 6722 — Clear WCO flag. This instruction should be immediately preceded by the two instructions cla and tad (2000) to obtain the operation indicated.

mewf — 6722 — Enable WCO flag. This instruction should be immediately preceded by the two instructions cla and tad (4000) to obtain the operation indicated.

miwf — 6722 — Initialize WCO flag. This instruction should be immediately preceded by the two instructions cla and tad (6000) to obtain the operation indicated.

msef — 6731 — Skip if the EOR flag is a 1. This flag is connected to the program interrupt.

mdef — 6732 — Disable ERF.

mced — 6732 — Clear ERF. This instruction should be immediately preceded by the two instructions cla and tad (2000) to obtain the operation indicated.

meef — 6732 — Enable ERF. This instruction should be immediately preceded by the two instructions cla and tad (4000) to obtain the operation indicated.

mief — 6732 — Initialize ERF, clear and enable. This instruction should be immediately preceded by the two instructions cla and tad (6000) to obtain the operation indicated.

mtrs — 6734 — Read tape status bits into the contents of the AC. This instruction should be immediately preceded by a cla instruction to obtain the operation indicated. The bit assignments are:
- 0 = data request late
- 1 = tape parity error
- 2 = read compare error
- 3 = end of File flag set
- 4 = write lock ring out
- 5 = tape at load point

6 = tape at end point
7 = tape near end point (Type 520)
7 = last operation write (Type 521 and 522 interfaces)
8 = tape near load point (Type 520)
8 = write echo (Type 522 interface)
8 = B control using transporting (Type 521 interface with multiplex transport)
9 = transport rewinding
10 = tape miss character

mcc — 6741 — Clear CA and WC.

mrwc — 6742 — Transfer $C(AC)_{0-11}$ to $C(WC)_{0-11}$

mrca — 6744 — Transfer $C(CA)_{0-11}$ to $C(AC)_{0-11}$. This instruction should be immediately preceded by a cla instruction to obtain the operation indicated.

mca — 6745 — Clear CA and WC, and transfer $C(AC)_{0-11}$ to $C(CA)_{0-11}$.

All operations begin with the program events indicated in the following basic program sequence. When the main program branches to this sequence (having received, for example, a high priority data break request from the tape control), the control and transport are interrogated for availability (mscr, msur) and if ready are instructed to carry out the specified task (mts, mtc). If the task is one of the eight listed in the instruction list under mtc, the mscr instruction completes the program sequence; if not, the program branches at "begin" to another routine (write, read, etc.), returning afterwards to "wait" in the basic program.

```
begin,      mscr            /skip if tape control free
            jmp.-1          /tape control not free, jump back to mscr
                            /instruction
            cla
            tad ia-1        /load AC with initial address minus one
            mca             /transfer AC to CA
            cla
            tad-n+1         /load AC with complement of number of
                            /words to be transferred plus one
            mrwc            /transfer AC to WC
            cla
            tad (*)         /load AC with selected information*
            mts             /transfer AC to control with parity density
                            /and unit number
            msur            /skip if tape transport ready
            jmp.-1          /transport not ready, jump back to msur
                            /instruction
            mtc             /transfer AC to control with command
                            /and tape motion mode
wait,       mscr            /wait for tape function to complete
            jmp.-1          /tape function not complete, jump back
                            /to mscr
            hlt             /operation completion
```

*A set of mnemonics that specifies all tape operations is furnished with the Type 57A.

When programming in the interrupt mode, the TCR flag causes an interrupt in the operating program and the flag may be tested by using the mscr instruction. The TCR flag must be cleared with the mcd command before dismissing the interrupt. WCO and ERF flags must be disabled before dismissing the interrupt, with the option of clearing or not clearing the flags.

CHAPTER 5

INTERFACE ELECTRICAL CHARACTERISTICS

One of the strong features of the PDP-5 is the relative ease of input-output device connection. Input-output devices can be connected into the system up to the limits specified in this section. Refer to the Digital Modules catalog A-705 for an explanation of standard DEC signals and loading definitions used in this section.

A coordinate system is used to locate cabinets, racks, modules and cable connectors, and terminals in the PDP-5. Cabinets are numbered beginning with the cabinet containing the operator console. Each position on the front of the cabinet is assigned a capital letter, beginning with A at the top, as indicated on Figure 11. Modules are numbered from 1 through 25 from left to right in a rack, as viewed from the wiring side. Connectors are numbered from 1 through 6, from left to right as viewed from the front of the machine. Blank module and connector locations are numbered. Terminals on a module connector are designated by capital letters from top to bottom, omitting G, I, O, and Q. Therefore, 1D05F is in cabinet 1, the fourth location from the top (D), the fifth module from the left (05), and the six (F) terminal from the top of the module.

Two 50-terminal cable connectors are available on the connector panel (1J01 and 1J02) for connection to I/O devices. Additional connector locations (1J03-1J05) are available for installation of connectors, as needed. Corresponding terminals of 1J01 and 1J02 are connected together and routed to signal origins or destinations in the machine logic. In the following discussions, origins of output signals and destinations of input signals are given with the terminal connection at 1J02. In this manner, the connections of both 1J01 and 1J02 are explained, and wiring to a new signal connector can be planned for bus connection to 1J02 or direct connection to the logic. Connections to 1J01 and 1J02 are summarized in Appendix B.

Device Selector

The device selector function is performed by a Type 4605 Pulse Amplifier for each I/O device or external register, which is individually selected. Each I/O device added to the system must contain a Type 4605 module, which has been

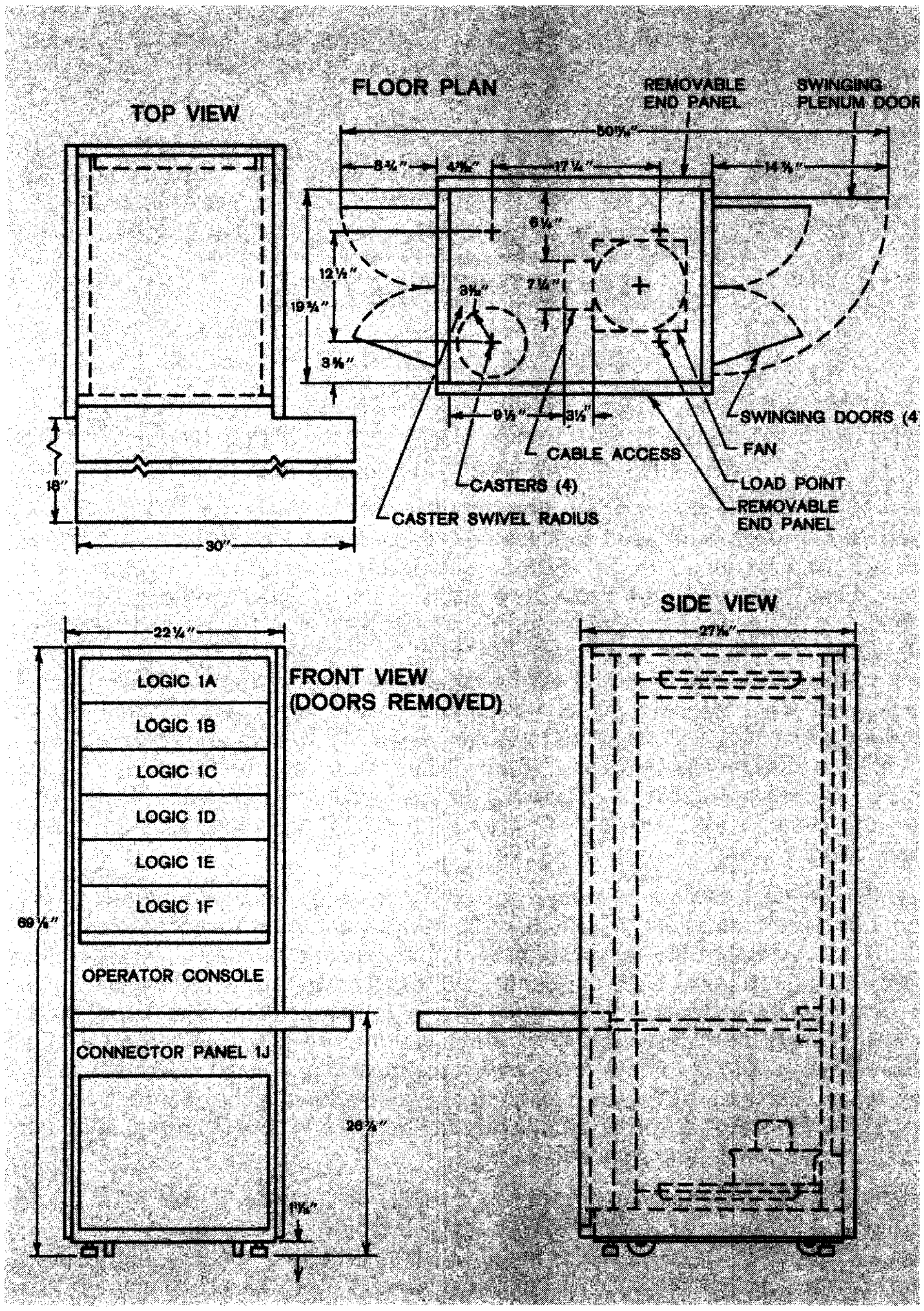


Figure 11 Component Location and Installation Diagram

prepared to select the device for a given combination of bits 3 through 8 of an iot instruction. When selected in this manner, Type 4605 produces IOT pulses related to the IOP pulses which are generated in accordance with the presence of ones in bits 9, 10, and 11 of the iot instruction. These IOT pulses, in turn, must be wired to initiate operation of the I/O device.

Therefore, cable connections must supply inputs to each Type 4605 from both the 1 and 0 output of memory buffer register bits 3 through 8 (12 lines) and from the three IOP generator outputs (6 lines or 3 twisted pairs). Connections are then made directly from the three output terminals of Type 4605 directly to the logic circuits of the I/O device. The input and output terminals of Type 4605 module are indicated in the logic diagram shown in Figure 12.

Type 4605 Pulse Amplifier modules are delivered with a jumper wire from both complementary inputs of each MB bit connected to one of the six inputs of the -AND diode gate. (Jumpers are indicated as dotted lines in Figure 12). The user must remove one jumper from each -AND gate input to establish the appropriate select code. (Both jumpers may be removed if the selection code requires it.) This system allows select codes to be changed in the module and not in cable connections. As delivered, these modules are also wired to produce negative IOT pulses. Positive IOT pulses can be obtained by reversing both jumper wire connections of a pulse transformer secondary winding.

Note that the input connections to Type 4605 must be as specified in Figure 12 and cannot be modified to operate more than one pulse amplifier (per module) at the same time. Should an I/O device require coincident positive and negative IOT pulses, two separate Type 4605 modules must be used, or an IOT pulse can be used to trigger external positive and negative pulse amplifiers. Note also that positive IOT pulses cannot be inverted to produce negative IOT pulses but can be used to trigger a pulse amplifier, such as Type 4604 or 4606 modules.

Output pulses from a Type 4605 Pulse Amplifier are standard for the DEC 4000 Series systems modules (2.5 volts, 0.4 microsecond). Each output is capable of driving 16 units of pulse load.

Memory Buffer Register

Bits 3 through 8 of an iot instruction are used to select the I/O device addressed by the instruction. During the F cycle, the instruction word is read from memory and placed in the memory buffer register. Complementary outputs from flip-flop bits MB_{3-8} are wired to input terminals of each device selector module connector. When the device selector is located within the I/O device, these MB lines must be connected to a cable connector.

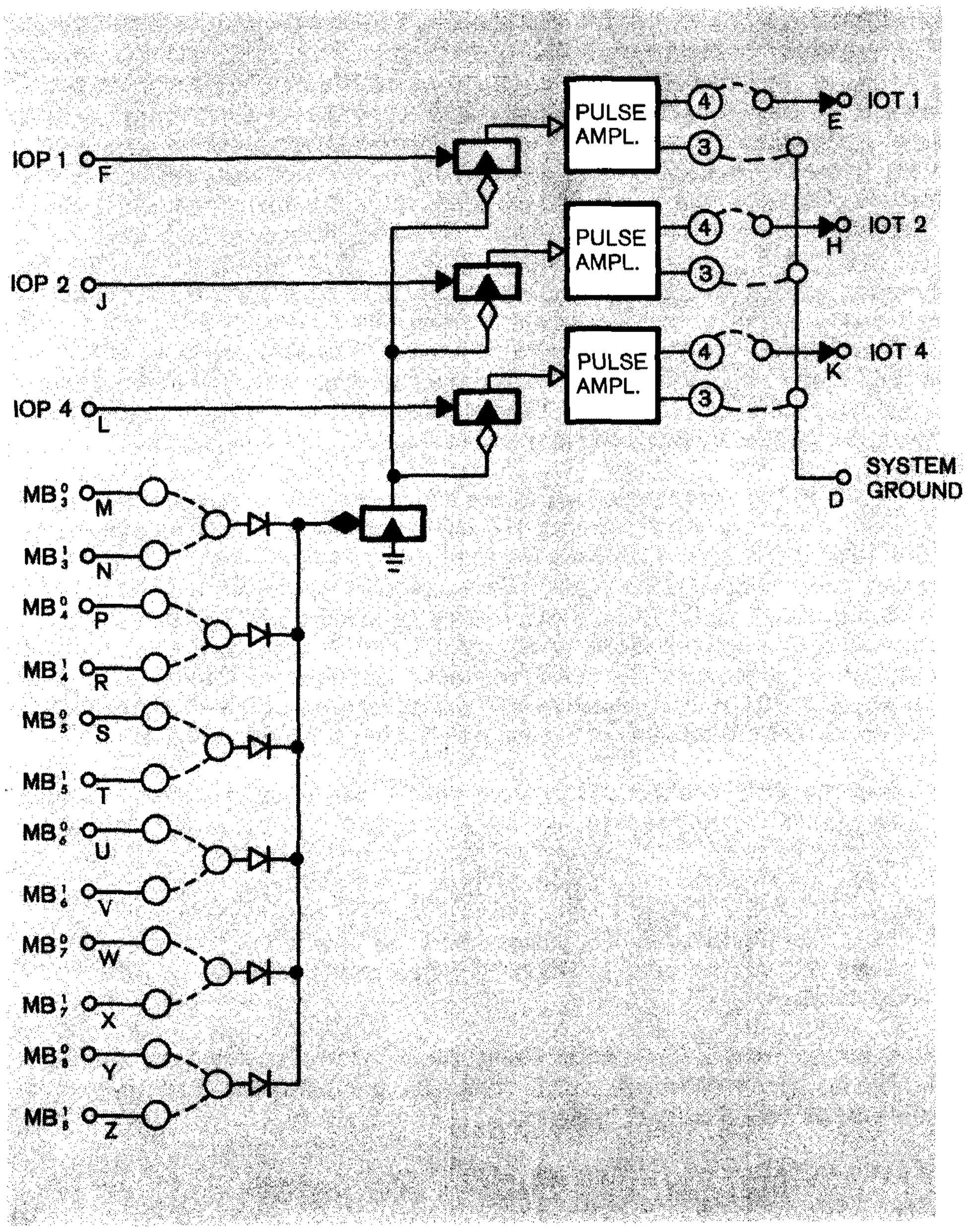


Figure 12 Type 4605 Pulse Amplifier Logic Diagram

The terminal locations for this connection are:

Signal	Origin	Bus Driver Output	Connection	Signal	Origin	Bus Driver Output	Connection
MB_3^0	1B05L	1F09L	1J02-27	MB_3^1	1B05K	1F09N	1J02-28
MB_4^0	1B06L	1F09T	1J02-29	MB_4^1	1B06K	1F09R	1J02-30
MB_5^0	1B07L	1F10L	1J02-31	MB_5^1	1B07K	1F10N	1J02-32
MB_6^0	1B08L	1F10T	1J02-33	MB_6^1	1B08K	1F10R	1J02-34
MB_7^0	1B09L	1F11L	1J02-35	MB_7^1	1B09K	1F11N	1J02-36
MB_8^0	1B10L	1F11T	1J02-37	MB_8^1	1B10K	1F11R	1J02-33

Memory buffer register outputs are wired from their point of origin in a Type 4206 Triple Flip-Flop module at locations 1B05 through 1B10 to connectors at 1F09 through 1F11. Normally, locations 1F09 through 1F11 contain dummy plugs which jumper terminals corresponding to the input and output of a Type 1684 Bus Driver. Therefore, when sufficient device selectors are added to the system to overload the normal driving capabilities of the Type 4206 modules, these dummy plugs can be removed and replaced by Type 1684 Bus Driver Modules. Each Type 4206 output can drive four Type 4605 Pulse Amplifier modules in the device selector. When the bus drivers are inserted in the system, each MB signal can drive at least 12 Type 4605 Pulse Amplifier modules, since Type 1684 can supply ± 15 milliamperes, and each Type 4605 requires 1.25 milliamperes shared among the grounded inputs. Under most circumstances, a single Type 1684 output can drive more than 12 Type 4605 modules because the load presented by a Type 4605 is shared by Type 1684 modules that drive it. To determine the maximum number of Type 4605 modules which can be driven by Type 1684 modules look for the condition where the minimum number of bus drivers is holding the maximum number of outputs at ground level. Under these conditions, the current delivered by each driver in a Type 1684 is equal to 1.25 milliamperes times the number of loads, divided by the number of bus drivers. This current must not exceed 15 milliamperes per driver circuit.

IOP Generator

The IOP pulses trigger the selected pulse amplifiers in the device selector located in the I/O device. These pulses are produced in a Type 4606 Pulse Amplifier module in location 1D25 and are routed as twisted-wire pairs to the appropriate input terminals of all Type 4605 Pulse Amplifier module connectors. Each IOP pulse can drive 16 Type 4605 modules.

Specific connection points for IOP pulses are:

Signal	Origin	Connection
IOP 1	1D25H	1J02-39*, 40
IOP 2	1D25P	1J02-41*, 42
IOP 4	1D25W	1J02-43*, 44

*Ground side of pulse amplifier transformer secondary winding to be connected to terminal D of the Type 4605 module in the device selector.

Accumulator Outputs

Data contained in the AC is available as static levels to supply information to I/O devices. These static levels can be strobed into an I/O device register by IOT pulses from the associated DS. Binary designation for the static output levels of the AC is:

−3 volts when AC bit contains a 0
0 volts when AC bit contains a 1

Connection points for these outputs are:

Signal	Origin	Bus Driver Output	Connection
AC^{1}_{0}	1B02E	1F06L	1J02-1
AC^{1}_{1}	1B03E	1F06N	1J02-2
AC^{1}_{2}	1B04E	1F06T	1J02-3
AC^{1}_{3}	1B05E	1F06R	1J02-4
AC^{1}_{4}	1B06E	1F07L	1J02-5
AC^{1}_{5}	1B07E	1F07N	1J02-6
AC^{1}_{6}	1B08E	1F07T	1J02-7
AC^{1}_{7}	1B09E	1F07R	1J02-8
AC^{1}_{8}	1B10E	1F08L	1J02-9
AC^{1}_{9}	1B11E	1F08N	1J02-10
AC^{1}_{10}	1B12E	1F08T	1J02-11
AC^{1}_{11}	1B13E	1F08R	1J02-12

Accumulator outputs are wired from their point of origin in a Type 4206 Triple Flip-Flop to module connectors at locations 1F06, 07, and 08. Normally these locations contain dummy plugs which jumper terminals corresponding to the input and output of a Type 1685 Bus Driver. When sufficient I/O devices are connected to the AC to overload Type 4206 modules, these dummy plugs can be removed and replaced by Type 1685 Bus Driver modules.

With the dummy plugs in the system each AC output signal is capable of driving:

six 1500-ohm capacitor-diode gate level inputs or
ten units of 5MC base load or
six units of 500KC base load or
two units of DC emitter load.

With the dummy plugs replaced by bus drivers each AC output signal is capable of driving:

one hundred 1500-ohm capacitor-diode gate level inputs or
fifteen units of base load or
twelve negative OR diode gates.

Each output can supply ±15 milliamperes. The rise and fall times of the output signals are approximately 1 microsecond. For more than a 5000-picofarad output load, the maximum rise or fall time in microseconds is equal to the capacitance in picofarads divided by 5000. Maximum rise or fall time of a bus driver output should be limited to 10 microseconds.

Accumulator Inputs

Transfer of data from an I/O device to the PDP-5 is normally received at the AC input. The AC input is accessible only through a pulse input to Type 4130 Capacitor-Diode Gate modules at locations 1E10 through 1E15. The level input to these gates is permanently connected to system ground and the pulse input is clamped at −3 volts by the Type 1000 Clamped Load Resistor module at location 1E16. Therefore, gated register outputs from many I/O devices can be connected to the AC input, so that IOT pulses set the information into the PDP-5. The input terminals are:

Signal	Connection	Load	Destination
AC^1_0	1J02-13	1E16E	1E10M
AC^1_1	1J02-14	1E16F	1E10Y
AC^1_2	1J02-15	1E16H	1E11M
AC^1_3	1J02-16	1E16J	1E12M
AC^1_4	1J02-17	1E16K	1E12M
AC^1_5	1J02-18	1E16L	1E12Y
AC^1_6	1J02-19	1E16M	1E13M
AC^1_7	1J02-20	1E16N	1E13Y
AC^1_8	1J02-21	1E16P	1E14M
AC^1_9	1J02-22	1E16R	1E14Y
AC^1_{10}	1J02-23	1E16S	1E15M
AC^1_{11}	1J02-24	1E16T	1E15Y

Driving any AC input connection point to ground potential sets a 1 into the corresponding AC flip-flop. The input change should be a maximum of 0.5 volts to avoid setting a flip-flop to a 1, and must be at least 2 volts with a rise time of less than 0.3 microseconds to reliably set a 1 into the AC. Each input presents a load of one standard clamped load resistor in parallel with 330 picofarads to ground.

Input-Output Skip

A skip bus is available for input connections to the PDP-5 from gated Skip pulses generated in I/O equipment. Input Skip pulses are usually produced by a flag or device status level which is strobed or sampled by an IOT pulse. The IOT pulse from the DS strobes the flag; and if it is in the preselected binary condition, the instruction following the iot is skipped.

Connection points for IOS are:

Signal	Connection	Load	Destination
IOS	1J02-25	1C04R	1D03E

To cause an instruction to be skipped, the IOS bus must be driven to ground potential for 0.4 microseconds by a pulse with a rise time of less than 0.2 microseconds. This pulse must originate in a high-impedance source, such as a transistor in a standard DEC inverter, diode gate, or capacitor-diode gate. The source of the IOS pulse cannot exhibit more than 1000 picofarads for the driving transistor.

These input pulses provide the complement input to the Type 4215 Four-Bit Counter module at location 1D03. Within the equipment this point is clamped at −3 volts by the collector load resistor of a Type 4129 Negative Capacitor-Diode Gate at location 1C04.

Program Interrupt

Signals from I/O devices, which interrupt the program in progress, are connected to a bus on the PDP-5. Connections to this bus must be in the form of static levels: ground potential to interrupt, −3 volts for no effect. The PI connection points are:

Signal	Connection	Destination
PI	1J02-26	1E04Y

The PI signal level is clamped at −3 volts by the collector load of the Type 4114 Diode NOR at location 1D04, is inverted and isolated by the Type 4102 Inverter at location 1E04, and is supplied to one input of the Type 4115 Diode AND at location 1D05 as the primary condition for initiating the internal interrupt gate. Connection to the PI bus represents 1 unit of dc emitter load. The maximum total leakage current from all sources connected to the PI bus must not exceed 6 milliamperes.

Input-Output Halt

The IOH facility provides a means of halting the advance of the program for an undetermined length of time while an I/O device executes a programmed operation. A specific iot instruction is decoded in the I/O device DS to produce IOT pulses which initiate device operation and return to the PDP-5 as an I/O Halt pulse. The I/O Halt pulse sets the I/O Halt flip-flop to 1, which in turn sets the run flip-flop to 0, so that the program stops. When the I/O device completes the operation specified by the iot instruction, it supplies a Restart pulse to the PDP-5 which returns the run flip-flop to the one state to continue the program and sets the I/O Halt flip-flop to 0.

These connections are:

Signal	Connection	Destination
I/O Hlt	1J02-46	1D12Y
Restart	1J02-48	1E02Y

I/O Halt pulses must be Standard DEC Negative Pulses (– 2.5 volts, 0.4 microsecond) or equivalent. The dc load presented to the signal by the input is ⅛ unit of dc emitter load. This load is shared by those inputs which are at ground. The transient load presented to a pulse input is 1 unit pulse load. I/O Halt pulses are received by a Type 4116 Diode module at location 1D12 which functions as a negative OR gate. The inverted output of this gate sets the I/O halt flip-flop when it is at ground potential. This flip-flop is contained in the Type 4215 module at location 1D01. The 1 output at the I/O halt flip-flop sets the run flip-flop to 0. The run flip-flop is also contained in the module at location 1D01.

The Restart pulse is received at the pulse input of a Type 4129 (negative) Capacitor-Diode Gate at location 1E02. The conditioning level input to this gate is provided by the one status of the I/O halt flip-flop. The Restart pulse may be driven from a Standard DEC 0.4 microsecond –2.5 volt Negative Pulse, or it may be driven from a negative-going level change. The level change should be 2.5 to 3.3 volts, with a maximum fall time of 0.4 microseconds. The input represents 3 units of pulse load.

Cabling

Power and signal cables enter the computer cabinet through a port in the bottom. The power cable is permanently wired to the equipment and signal cables mate with connectors, which are mounted on the front of the cabinet, facing the center of the machine.

Power cables for the computer and for most peripheral equipment are supplied with twist-lock connectors, rated at 30 amperes. To mate with the power cables, power sources should be provided with Hubbell 7310B, or equivalent twist-lock, flush receptacles rated at 30 amperes, 115 volts alternating current. Note that the receptacle terminal stamped GR or marked with green paint must be grounded.

Signal cables are 50-wire, shielded, with Amphenol 115-114P male connectors and 1391 shells on both ends. To mate with a signal cable, special equipment in the system must be provided with Amphenol 115-114S female connectors. Unless otherwise specified by the user, power cables are supplied in 20 foot lengths; signal cables, in 25 foot lengths. Power cables are 11/16 inch in diameter; signal cables are 13/16 inch in diameter.

I/O Halt pulses must be Standard DEC Negative Pulses (−2.5 volts, 0.4 microsecond) or equivalent. The dc load presented to the signal by the input is ⅛ unit of dc emitter load. This load is shared by those inputs which are at ground. The transient load presented to a pulse input is 1 unit pulse load. I/O Halt pulses are received by a Type 4116 Diode module at location 1D12 which functions as a negative OR gate. The inverted output of this gate sets the I/O halt flip-flop when it is at ground potential. This flip-flop is contained in the Type 4215 module at location 1D01. The 1 output at the I/O halt flip-flop sets the run flip-flop to 0. The run flip-flop is also contained in the module at location 1D01.

The Restart pulse is received at the pulse input of a Type 4129 (negative) Capacitor-Diode Gate at location 1E02. The conditioning level input to this gate is provided by the one status of the I/O halt flip-flop. The Restart pulse may be driven from a Standard DEC 0.4 microsecond −2.5 volt Negative Pulse, or it may be driven from a negative-going level change. The level change should be 2.5 to 3.3 volts, with a maximum fall time of 0.4 microseconds. The input represents 3 units of pulse load.

Cabling

Power and signal cables enter the computer cabinet through a port in the bottom. The power cable is permanently wired to the equipment and signal cables mate with connectors, which are mounted on the front of the cabinet, facing the center of the machine.

Power cables for the computer and for most peripheral equipment are supplied with twist-lock connectors, rated at 30 amperes. To mate with the power cables, power sources should be provided with Hubbell 7310B, or equivalent twist-lock, flush receptacles rated at 30 amperes, 115 volts alternating current. Note that the receptacle terminal stamped GR or marked with green paint must be grounded.

Signal cables are 50-wire, shielded, with Amphenol 115-114P male connectors and 1391 shells on both ends. To mate with a signal cable, special equipment in the system must be provided with Amphenol 115-114S female connectors. Unless otherwise specified by the user, power cables are supplied in 20 foot lengths; signal cables, in 25 foot lengths. Power cables are 11/16 inch in diameter; signal cables are 13/16 inch in diameter.

APPENDIX 1

INSTRUCTIONS

MEMORY REFERENCE INSTRUCTIONS

Mnemonic Symbol	Operation Code	Time (μsec)	Operation
and Y	0	18	Logical AND. The AND operation is performed between the C(Y) and the C(AC). $C(Y)_i \wedge C(AC)_i => C(AC)_i$.
tad Y	1	18	Twos complement add. The C(Y) are added to the C(AC) in twos complement arithmetic. $C(Y) + C(AC) => C(AC)$.
isz Y	2	18	Index and skip if zero. The C(Y) are incremented by one in twos complement arithmetic. If the resultant C(Y) = 0, the next instruction is skipped. $C(Y) + 1 => C(Y)$. If result = 0, $C(PC) + 1 => C(PC)$.
dca Y	3	18	Deposit and clear AC. The C(AC) are deposited in core memory location Y and the AC is cleared. $C(AC) => C(Y)$, then $0 => C(AC)$.
jms Y	4	24	Jump to subroutine. The C(PC) are deposited in core memory location Y. The next instruction is taken from location Y+1. $C(PC) + 1 => C(Y)$ $Y + 1 => C(PC)$
jmp Y	5	12	Jump to Y. The C(PC) are set to address Y. The next instruction is taken from core memory location Y. $Y => C(PC)$.

BASIC IOT MICROINSTRUCTIONS

Mnemonic Symbol	Octal Code	Operation
		PROGRAM INTERRUPT
ion	6001	Turn interrupt on
iof	6002	Turn interrupt off
		ANALOG-TO-DIGITAL CONVERTER
adc	6004	Convert analog to digital
		HIGH SPEED PERFORATED TAPE READER
rsf	6011	Skip if Photoreader flag = 1
rrb	6012	Read the contents of the photoreader buffer into $C(AC)_{4-11}$ and clear the Photoreader flag
rfc	6014	Clear Photoreader flag and buffer, fetch one character from tape and load it into the photoreader buffer, and set the Photoreader flag when done.
		HIGH SPEED PERFORATED TAPE PUNCH
psf	6021	Skip if High Speed Punch flag = 1
pcf	6022	Clear the Punch flag and buffer
ppc	6024	Load the punch buffer from $C(AC)_{4-11}$ and punch the character (this instruction does not clear the High Speed Punch flag or buffer).
pls	6026	Clear the Punch flag and buffer, load the punch buffer from $C(AC)_{4-11}$, punch the character, and set the Punch flag when done.
		TELETYPE KEYBOARD/READER
ksf	6031	Skip if Keyboard flag = 1
kcc	6032	Clear AC and Keyboard flag.
krs	6034	Read the contents of the keyboard buffer into $C(AC)_{4-11}$ (does not clear AC or flag.)
krb	6036	Clear AC, read keyboard buffer into AC, clear Keyboard flag.
		TELETYPE TELEPRINTER/PUNCH
tsf	6041	Skip if Teleprinter flag = 1
tcf	6042	Clear Teleprinter flag
tls	6046	Load the LUO from the $C(AC)_{4-11}$, clear Teleprinter flag, and print and/or punch the character.

BASIC IOT MICROINSTRUCTIONS (continued)

Mnemonic Symbol	Octal Code	Operation
		OSCILLOSCOPE DISPLAY AND PRECISION CRT DISPLAY
dcx	6051	Clear X buffer
dxl	6053	Clear and load the X buffer $C(AC)_{2-11} => C(YB)$.
dcy	6061	Clear Y buffer
dyl	6063	Clear and load the Y buffer $C(AC)_{2-11} => C(YB)$.
dix	6054	Intensify the point defined by C(XB) and C(YB)
diy	6064	Intensify the point defined by C(XB) and C(YB)
dxs	6057	Executes the combined functions of dxl followed by dix
dys	6067	Executes the combined functions of dyl followed by diy.
dsf	6071	Skip if Display flag = 1
dcf	6072	Clear Display flag
dlb	6074	Load brightness register. $C(AC)_{8-11} => C(BR)$
		INCREMENTAL PLOTTER
plsf	6501	Skip if Plotter flag = 1
plcf	6502	Clear Plotter flag
plpu	6504	Plotter pen up
plpr	6511	Plotter pen right
pldu	6512	Plotter drum upward
pldd	6514	Plotter drum downward
plpl	6521	Plotter pen left
pldu	6522	Plotter drum upward
plpd	6524	Plotter pen down
		LINE PRINTER
lcf	6652	Clear Line Printer flag.
lpr	6655	Clear the format register. Load the format register from $C(AC)_{9-11}$, print the line contained in the last half of the printing buffer, and advance the paper according to the contents of the format register if $C(AC)_8 = 1$.
lsf	6661	Skip if Line Printer flag = 1.

BASIC IOT MICROINSTRUCTIONS (continued)

Mnemonic Symbol	Octal Code	Operation
		LINE PRINTER (continued)
lcb	6662	Clear printing buffer.
lld	6664	Load printing buffer from $C(AC)_{6-11}$.
		CARD READER AND CONTROL
crsf	6632	Skip if Card Reader flag = 1.
cers	6634	Card equipment read status. Reads the status of the card reader into $C(AC)_{6-9}$.
crrb	6671	Read the contents of the card column buffer into the C(AC) and clear the Card Reader flag.
crsa	6672	Select a card in alphanumeric mode.
crsb	6674	Select a card in binary mode.
		CARD PUNCH CONTROL
cpsf	6631	Skip if Card Punch flag = 1.
cers	6634	Card equipment read status. Reads the status of the Card Punch flag into bit 10 and the card punch error level into bit 11 of the AC.
cpcf	6641	Clear Card Punch flag.
cpse	6642	Select the card punch and transmit a card from the hopper to the 80-column punch die.
cplb	6644	Load the card punch buffer from C(AC).
		AUTOMATIC MAGNETIC TAPE CONTROL
mscr	6701	Skip if tape control unit is ready. If TCR = 1, then C(PC) + 1 => C(PC)
mcd	6702	Disable the TCR flag from the program interrupt; clear command register, WCO, and EOR. Used when C(AC) = 4000.
mts	6706	Disable the TCR flag from the program interrupt, clear WCO and EOR. Select unit, parity mode, and density.
msur	6711	Skip if tape transport unit is ready. If TTR = 1, then C(PC) + 1 => C(PC)
mnc	6712	Terminate continuous mode. Used when C(AC) = 4000.
mtc	6716	Load tape control unit command register, start tape motion,

BASIC IOT MICROINSTRUCTIONS (continued)

Mnemonic Symbol	Octal Code	Operation
		AUTOMATIC MAGNETIC TAPE CONTROL (continued)
mswf	6721	Skip if WCO flag = 1 and clear AC.
mdwf	6722	Disable WCO flag. Used when C(AC) = 2000.
mewf	6722	Enable WCO flag. Used when C(AC) = 4000.
miwf	6722	Initialize WCO flag (clear, enable). Used when C(AC) = 600.
msef	6731	Skip if EOR flag = 1.
mdef	6732	Disable ERF.
mced	6732	Clear ERF. Used when C(AC) = 2000.
meef	6732	Enable ERF. Used when C(AC) = 4000.
mief	6732	Initialize ERF (clear, enable). Used when C(AC) = 6000.
mtrs	6734	Read tape status bits into C(AC). Used when C(AC) = 0000.
mcc	6741	Clear CA and WC.
mrwc	6742	Read word counter. C(WC) => $C(AC)_{0-11}$
mrca	6744	Read current address. Used when C(AC) = 0000. C(CA) => $C(AC)_{0-11}$
mca	6745	Read current address, and clear CA and WC. Executes the combined functions of mcc with mrca.

GROUP 1 OPERATE MICROINSTRUCTIONS

Mnemonic Symbol	Octal Code	Event Time	Operation
nop	7000	—	No operation. Causes a 12 μsec program delay.
iac	7001	3	Index AC. C(AC) +1=> C(AC)
ral	7004	2	Rotate the C(AC) and the C(L) left one place. $C(AC)_i$ => $C(AC)_{i-1}$ C(L) => $C(AC)_{11}$ $C(AC)_0$ => C(L)
rtl	7006	2, 3	Rotate two left.

GROUP 1 OPERATE MICROINSTRUCTIONS (continued)

Mnemonic Symbol	Octal Code	Event Time	Operation
rar	7010	2	Rotate the C(AC) and the C(L) right one place. $C(AC)_i => C(AC)_{i+1}$ $C(AC)_{11} => C(L)$ $C(L) => C(AC)_0$
rtr	7012	2, 3	Rotate two right.
cml	7020	2	Complement L. $C(\overline{L}) => C(L)$
cma	7040	2	Complement AC. $C(\overline{AC}) => C(AC)$
cll	7100	1	Clear L. $0 => C(L)$
cla	7200	1	Clear AC. $0 => C(AC)$

GROUP 2 OPERATE MICROINSTRUCTIONS

Mnemonic Symbol	Octal Code	Event Time	Operation
hlt	7402	3	Halt. Stops the program.
osr	7404	3	OR with Switch Register $C(SR) \vee C(AC) = > (CAC)$
skp	7410	1	Skip, unconditional. $C(PC) + 1 = > C(PC)$
snl	7420	1	Skip on non-zero L. If $C(L) = 1$, then $C(PC) + 1 = > C(PC)$
szl	7430	1	Skip on zero L. If $C(L) = 0$, then $C(PC) + 1 = > C(PC)$
sza	7440	1	Skip on zero AC. If $C(AC) = 0$, then $C(PC) + 1 = > C(PC)$
sna	7450	1	Skip on non-zero AC. If $C(AC) \neq 0$, then $C(PC) +1 = > C(PC)$
sma	7500	1	Skip on minus AC. If $C(AC)_0 = 1$, then $C(PC) + 1 => C(PC)$
spa	7510	1	Skip on positive AC. If $C(AC)_0 = 0$, then $C(PC) + 1 => C(PC)$
cla	7600	2	Clear AC $0 = > C(AC)$

APPENDIX 2

CODES

TELETYPE CODE

Character	8-Bit Code (in octal)	6-Bit Trimmed Code (in octal)	Character	8-Bit Code (in octal)	6-Bit Trimmed Code (in octal)
A	301	01	!	241	41
B	302	02	"	242	42
C	303	03	#	243	43
D	304	04	$	244	44
E	305	05	%	245	45
F	306	06	&	246	46
G	307	07	'	247	47
H	310	10	(	250	50
I	311	11	)	251	51
J	312	12	*	252	52
K	313	13	+	253	53
L	314	14	,	254	54
M	315	15	-	255	55
N	316	16	.	256	56
O	317	17	/	257	57
P	320	20	:	272	72
Q	321	21	;	273	73
R	322	22	<	274	74
S	323	23	=	275	75
T	324	24	>	276	76
U	325	25	?	277	77
V	326	26	@	300	00
W	327	27	[	333	33
X	330	30	\	334	34
Y	331	31	]	335	35
Z	332	32	↑	336	36
0	260	60	←	337	37
1	261	61	EOT	204	—
2	262	62	W RU	205	—
3	263	63	RU	206	—
4	264	64	BELL	207	—
5	265	65	Line Feed	212	—
6	266	66	Return	215	—
7	267	67	Space	240	40
8	270	70	ACK	374	—
9	271	71	ALT MODE	375	—
			Rub Out	377	—

CARD READER AND LINE PRINTER OCTAL CODES

Octal Code	Card Reader Character	Line Printer Character	Octal Code	Card Reader Character	Line Printer Character	Octal Code	Card Reader Character	Line Printer Character
00	....	space	25	V	V	53	$	=
01	1	1	26	W	W	54	*	.
02	2	2	27	X	X	55	....	)
03	3	3	30	Y	Y	56	....	—
04	4	4	31	Z	Z	57	....	(
05	5	5	32	....	"	60	+	—
06	6	6	33	'	'	61	A	A
07	7	7	34	(	>	62	B	B
10	8	8	35	....	↑	63	C	C
11	9	9	36	....	→	64	D	D
12	0	'	37	....	?	65	E	E
13	=	~	40	....	°	66	F	F
14	;	....	41	J	J	67	G	G
15	....	....	42	K	K	70	H	H
16	....	....	43	L	L	71	I	I
17	....	<	44	M	M	72	....	×
20	....	0	45	N	N	73	....	.
21	/	/	46	O	O	74	)	+
22	S	S	47	P	P	75	....	]
23	T	T	50	Q	Q	76	....	|
24	U	U	51	R	R	77	....	[

CARD READER AND LINE PRINTER BINARY CODES

Low Order Bits	High Order Bits 00 Card Reader Character	00 Line Printer Character	01 Card Reader Character	01 Line Reader Character	10 Card Reader Character	10 Line Reader Character	11 Card Reader Character	11 Line Reader Character
0000	....	space		0	—	°	+[&]	—
0001	1	1	/	/	J	J	A	A
0010	2	2	S	S	K	K	B	B
0011	3	3	T	T	L	L	C	C
0100	4	4	U	U	M	M	D	D
0101	5	5	V	V	N	N	E	E
0110	6	6	W	W	O	O	F	F
0111	7	7	X	X	P	P	G	G
1000	8	8	Y	Y	Q	Q	H	H
1001	9	9	Z	Z	R	R	I	I
1010	0	′	....	″	....	....	....	×
1011	=[#]	~	,	,	$	=	.	.
1100	′[@]	⊃	([%]	>	*	—	)[□]	+
1101	....	V	....	↑	....	)	....	]
1110	....	Λ	....	→	....	—	....	I
1111	....	<	....	?	....	(	....	[

HOLLERITH CARD CODE

Digit	No Zone	Zone 12	11	0
no punch	blank	+[&]	—	0
1	1	A	J	/
2	2	B	K	S
3	3	C	L	T
4	4	D	M	U
5	5	E	N	V
6	6	F	O	W
7	7	G	P	X
8	8	H	Q	Y
9	9	I	R	Z
8-3	=[#]	.	$	,
8-4	,[@]	)[□]	*	([%]

APPENDIX 3

PERFORATED-TAPE LOADER SEQUENCES

READIN MODE LOADER

The readin mode (RIM) loader is a minimum length, basic, perforated-tape reader for the PDP-5. It is initially stored in memory by manual use of the operator console keys and switches. The loader is permanently stored in 17 locations of the highest numbered page.

A perforated tape to be read by the RIM loader must be in RIM format:

Tape Channel 8 7 6 5 4 S 3 2 1	Format
1 0 0 0 0 . 0 0 0	Leader-trailer code
0 1 A1 . A2 0 0 A3 . A4	Absolute address to contain next 4 digits
0 0 X1 . X2 0 0 X3 . X4	Contents of previous 4-digit address
0 1 A1 . A2 0 0 A3 . A4	Address
0 0 X1 . X2 0 0 X3 . X4	Contents
(Etc.)	(Etc.)
1 0 0 0 0 . 0 0 0	Leader-trailer code

A tape in RIM format is generally concluded with address = 0000 and content = SA-1, where SA indicates starting address. In this way, the SA of the routine just loaded is stored in the program counter of the PDP-5. The next instruction to be executed will then be taken from the SA, (i.e., the program counter is incremented, then used as the address of the instruction). Therefore, the loaded routine is self-starting. It is suggested that this procedure always be used. If it is not desirable for the routine to be self-starting, simply store a halt instruction in the SA. Pressing the CONTINUE key then starts the routine.

The RIM loader can only be used in conjunction with the 33 ASR reader (not the high-speed perforated-tape reader). Because a tape in RIM format is, in effect, twice as long as it need be, it is suggested that the RIM loader be used only to read the binary loader when using the 33 ASR.

The complete PDP-5 RIM loader (SA = 1700 in systems with IK memory or 7700 in systems with 4K memory) is as follows:

Addr.	Octal Contents	Tag	Inst'n I Z	Comments
700,	6032	beg,	kcc	/clear AC and flag
701,	6031		rsf	/skip if flag = 1
702,	5301		jmp .−1	/looking for char
703,	6036		krb	/read buffer
704,	7106		cll rtl	
705,	7006		rtl	/ch 8 in AC_0
706,	7510		spa	/checking for leader
707,	5301		jmp beg +1	/found leader
710,	7006		rtl	/OK, ch 7 in link
711,	6031		ksf	
712,	5311		jmp .−1	/read, do not clear
713,	6034		krs	/checking for address
714,	7420		snl	
715,	3720		dca i temp	/store contents
716,	3320		dca temp	/store address
717,	5300		jmp beg	/next word
720,		temp,		/temp storage

Placing the RIM loader in core memory by way of the operator console keys and switches is accomplished as follows:

1. Set the appropriate starting address in the switch register (SR).
2. Press LOAD ADDRESS key.
3. Set the first instruction in the SR.
4. Press the DEPOSIT key.
5. Set the next instruction in the SR.
6. Press DEPOSIT key.
7. Repeat steps 5 and 6 until all 16 instructions have been deposited.

To load a tape in RIM format, place the tape in the reader, set the SR to the appropriate starting address, press the LOAD ADDRESS key, press the START key, and start the Teletype reader.

BINARY LOADER

The binary loader (BIN) is used to read machine language tapes (in binary format) produced by the program assembly language (PAL). A tape in binary format is about one half the length of the comparable RIM format tape. It can, therefore, be read about twice as fast as a RIM tape and is, for this reason, the more desirable format to use with the 10 cps 33 ASR reader.

The format of a binary tape is as follows:

LEADER: about 2 feet of leader-trailer codes.

BODY: characters representing the absolute, machine language program in easy-to-read binary (or octal) form. The section of tape may contain characters representing instructions (channels 8 and 7 not punched) or origin resettings (channel 8 not punched, channel 7 punched) and is concluded by 2 characters (channels 8 and 7 not punched) that represent a check-sum for the entire section.

TRAILER: same as leader

Example of the format of a binary tape:

Tape Channel 8 7 6 5 4 S 3 2 1	Memory Location	Contents
1 0 0 0 0 . 0 0 0	leader-trailer code	
0 1 0 0 0 . 0 1 0		
0 0 0 0 0 . 0 0 0		
0 0 1 1 1 . 0 1 0		
0 0 0 0 0 . 0 0 0	0200	cla
0 0 0 0 1 . 0 1 0		
0 0 1 1 1 . 1 1 1	0201	tad 277
0 0 0 1 1 . 0 1 0		
0 0 1 1 1 . 1 1 0	0202	dca 276
0 0 1 1 1 . 1 0 0		
0 0 0 0 0 . 0 1 0	0203	hlt
0 1 0 0 0 . 0 1 0		
0 0 1 1 1 . 1 1 1	original setting at 0277	
0 0 0 0 0 . 0 0 0		
0 0 1 0 1 . 0 1 1	0277	0053
0 0 0 0 1 . 0 0 0		
0 0 0 0 0 . 1 1 1	sum check 1007	
1 0 0 0 0 . 0 0 0	leader-trailer code	

After a BIN tape has been read in, one of the two following conditions exists:

a. No check-sum error: halt with AC = 0
b. Check-sum error: halt with AC = (computed check-sum) − (tape check-sum)

The BIN loader in no way depends upon or uses the RIM leader. To load a tape in BIN format place the tape in the reader, set the SR to 1777, press the LOAD ADDRESS key, press the START key, and start the tape reader.

APPENDIX 4

SOFTWARE

A programming parcel is supplied to each user of the PDP-5. Each parcel consists of program descriptions and perforated-paper tapes applicable to a particular system, selected from the DEC Program Library. The following programs are included in each package:

a. Program Assembly Language (PAL)
b. Readin Mode and Binary Tape Loaders
c. Symbolic Tape Editor
d. Mnemonic-Octal Debugging Routine
e. Multiply and Divide Subroutines, single and double precision
f. Square Root, Sine, and Cosine Subroutines
g. Binary-to-Decimal and Decimal-to-Binary Conversion Subroutines
h. Interpretive Floating Point Package
i. Floating Point I/O Package
j. Teletype Output Package
k. Maintenance Programs

New techniques, routines, and programs are constantly being developed, field-tested, and documented in the DEC Program Library for incorporation in users' systems.

APPENDIX 5

TABLE OF POWERS OF TWO

2^n	n	2^{-n}
1	0	1.0
2	1	0.5
4	2	0.25
8	3	0.125
16	4	0.062 5
32	5	0.031 25
64	6	0.015 625
128	7	0.007 812 5
256	8	0.003 906 25
512	9	0.001 953 125
1 024	10	0.000 976 562 5
2 048	11	0.000 488 281 25
4 096	12	0.000 244 140 625
8 192	13	0.000 122 070 312 5
16 384	14	0.000 061 035 156 25
32 768	15	0.000 030 517 578 125
65 536	16	0.000 015 258 789 062 5
131 072	17	0.000 007 629 394 531 25
262 144	18	0.000 003 814 697 265 625
524 288	19	0.000 001 907 348 632 812 5
1 048 576	20	0.000 000 953 674 316 406 25
2 097 152	21	0.000 000 476 837 158 203 125
4 194 304	22	0.000 000 238 418 579 101 562 5
8 388 608	23	0.000 000 119 209 289 550 781 25
16 777 216	24	0.000 000 059 604 644 775 390 625
33 554 432	25	0.000 000 029 802 322 387 695 312 5
67 108 864	26	0.000 000 014 901 161 193 847 656 25
134 217 728	27	0.000 000 007 450 580 596 923 828 125
268 435 456	28	0.000 000 003 725 290 298 461 914 062 5
536 870 912	29	0.000 000 001 862 645 149 230 957 031 25
1 073 741 824	30	0.000 000 000 931 322 574 615 478 515 625
2 147 483 648	31	0.000 000 000 465 661 287 307 739 257 812 5
4 294 967 296	32	0.000 000 000 232 830 643 653 869 628 906 25
8 589 934 592	33	0.000 000 000 116 415 321 826 934 814 453 125
17 179 869 184	34	0.000 000 000 058 207 660 913 467 407 226 562 5
34 359 738 368	35	0.000 000 000 029 103 830 456 733 703 613 281 25
68 719 476 736	36	0.000 000 000 014 551 915 228 366 851 806 640 625
137 438 953 472	37	0.000 000 000 007 275 957 614 183 425 903 320 312 5
274 877 906 944	38	0.000 000 000 003 637 978 807 091 712 951 660 156 25
549 755 813 888	39	0.000 000 000 001 818 989 403 545 856 475 830 078 125
1 099 511 627 776	40	0.000 000 000 000 909 494 701 772 928 237 915 039 062 5
2 199 023 255 552	41	0.000 000 000 000 454 747 350 886 464 118 957 519 531 25
4 398 046 511 104	42	0.000 000 000 000 227 373 675 443 232 059 478 759 765 625
8 796 093 022 208	43	0.000 000 000 000 113 686 837 721 616 029 739 379 882 812 5
17 592 186 044 416	44	0.000 000 000 000 056 843 418 860 808 014 869 689 941 406 25
35 184 372 088 832	45	0.000 000 000 000 028 421 709 430 404 007 434 844 970 703 125
70 368 744 177 664	46	0.000 000 000 000 014 210 854 715 202 003 717 422 485 351 562 5
140 737 488 355 328	47	0.000 000 000 000 007 105 427 357 601 001 858 711 242 675 781 25
281 474 976 710 656	48	0.000 000 000 000 003 552 713 678 800 500 929 355 621 337 890 625
562 949 953 421 312	49	0.000 000 000 000 001 776 356 839 400 250 464 677 810 668 945 312 5
1 125 899 906 842 624	50	0.000 000 000 000 000 888 178 419 700 125 232 338 905 334 472 656 25
2 251 799 813 685 248	51	0.000 000 000 000 000 444 089 209 850 062 616 169 452 667 236 328 125
4 503 599 627 370 496	52	0.000 000 000 000 000 222 044 604 925 031 308 084 726 333 618 164 062 5
9 007 199 254 740 992	53	0.000 000 000 000 000 111 022 302 462 515 654 042 363 166 809 082 031 25
18 014 398 509 481 984	54	0.000 000 000 000 000 055 511 151 231 257 827 021 181 583 404 541 015 625
36 028 797 018 963 968	55	0.000 000 000 000 000 027 755 575 615 628 913 510 590 791 702 270 507 812 5
72 057 594 037 927 936	56	0.000 000 000 000 000 013 877 787 807 814 456 755 295 395 851 135 253 906 25
144 115 188 075 855 872	57	0.000 000 000 000 000 006 938 893 903 907 228 377 647 697 925 567 626 953 125
288 230 376 151 711 744	58	0.000 000 000 000 000 003 469 446 951 953 614 188 823 848 962 783 813 476 562 5
576 460 752 303 423 488	59	0.000 000 000 000 000 001 734 723 475 976 807 094 411 924 481 391 906 738 281 25
1 152 921 504 606 846 976	60	0.000 000 000 000 000 000 867 361 737 988 403 547 205 962 240 695 953 369 140 625
2 305 843 009 213 693 952	61	0.000 000 000 000 000 000 433 680 868 994 201 773 602 981 120 347 976 684 570 312 5
4 611 686 018 427 387 904	62	0.000 000 000 000 000 000 216 840 434 497 100 886 801 490 560 173 988 342 285 156 25
9 223 372 036 854 775 808	63	0.000 000 000 000 000 000 108 420 217 248 550 443 400 745 280 086 994 171 142 578 125
18 446 744 073 709 551 616	64	0.000 000 000 000 000 000 054 210 108 624 275 221 700 372 640 043 497 085 571 289 062 5
36 893 488 147 419 103 232	65	0.000 000 000 000 000 000 027 105 054 312 137 610 850 186 320 021 748 542 785 644 531 25
73 786 976 294 838 206 464	66	0.000 000 000 000 000 000 013 552 527 156 068 805 425 093 160 010 874 271 392 822 265 625
147 573 952 589 676 412 928	67	0.000 000 000 000 000 000 006 776 263 578 034 402 712 546 580 005 437 135 696 411 132 812 5
295 147 905 179 352 825 856	68	0.000 000 000 000 000 000 003 388 131 789 017 201 356 273 290 002 718 567 848 205 566 406 25
590 295 810 358 705 651 712	69	0.000 000 000 000 000 000 001 694 065 894 508 600 678 136 645 001 359 283 924 102 783 203 125
1 180 591 620 717 411 303 424	70	0.000 000 000 000 000 000 000 847 032 947 254 300 339 068 322 500 679 641 962 051 391 601 562 5
2 361 183 241 434 822 606 848	71	0.000 000 000 000 000 000 000 423 516 473 627 150 169 534 161 250 339 820 981 025 695 800 781 25
4 722 366 482 869 645 213 696	72	0.000 000 000 000 000 000 000 211 758 236 813 575 084 767 080 625 169 910 490 512 847 900 390 625

APPENDIX 6

INTERFACE CONNECTIONS

Connection points for interface signals at the I/O connectors 1J01 and 1J02 are listed in the following table.

PDP-5 I/O CONNECTIONS

Signal	Symbol	Terminal	Signal	Symbol	Terminal
AC ${}^{1}_{0}$ Out	—◇	1	AC ${}^{1}_{4}$ In	—▷	17
AC ${}^{1}_{1}$ Out	—◇	2	AC ${}^{1}_{5}$ In	—▷	18
AC ${}^{1}_{2}$ Out	—◇	3	AC ${}^{1}_{6}$ In	—▷	19
AC ${}^{1}_{3}$ Out	—◇	4	AC ${}^{1}_{7}$ In	—▷	20
AC ${}^{1}_{4}$ Out	—◇	5	AC ${}^{1}_{8}$ In	—▷	21
AC ${}^{1}_{5}$ Out	—◇	6	AC ${}^{1}_{9}$ In	—▷	22
AC ${}^{1}_{6}$ Out	—◇	7	AC ${}^{1}_{10}$ In	—▷	23
AC ${}^{1}_{7}$ Out	—◇	8	AC ${}^{1}_{11}$ In	—▷	24
AC ${}^{1}_{8}$ Out	—◇	9	IOS	—▷	25
AC ${}^{1}_{9}$ Out	—◇	10	PI	—◇	26
AC ${}^{1}_{10}$ Out	—◇	11	MB ${}^{0}_{3}$	—◇	27
AC ${}^{1}_{11}$ Out	—◇	12	MB ${}^{1}_{3}$	—◆	28
AC ${}^{1}_{0}$ In	—▷	13	MB ${}^{0}_{4}$	—◆	29
AC ${}^{1}_{1}$ In	—▷	14	MB ${}^{1}_{4}$	—◆	30
AC ${}^{1}_{2}$ In	—▷	15	MB ${}^{0}_{5}$	—◆	31
AC ${}^{1}_{3}$ In	—▷	16	MB ${}^{1}_{5}$	—◆	32

PDP-5 I/O CONNECTIONS (continued)

Signal	Symbol	Terminal	Signal	Symbol	Terminal
MB $^{0}_{6}$	—◆	33	IOP 4*	---	43
MB $^{1}_{6}$	—◆	34	IOP 4	—▶	44
MB $^{0}_{7}$	—◆	35	1 MC clock	—▶	45
MB $^{1}_{7}$	—◆	36	I/O Hlt	—▶	46
MB $^{0}_{8}$	—◆	37	AC Clear	—▷	47
MB $^{1}_{8}$	—◆	38	Restart	—▶	48
IOP 1*	---	39	Power Clear	—▶	49
IOP 1	—▶	40	Ground	⏚	50
IOP 2*	---	41			
IOP 2	—	42			

*Ground side of pulse transformer secondary winding.

Connection points for data break signals at connector 1J03 are presented in the following table.

PDP-5 DATA BREAK CONNECTIONS

Signal	Symbol	Terminal	Signal	Symbol	Terminal
MB $^{1}_{0}$ Out	—◇	1	MB $^{1}_{11}$ In	—◇	24
MB $^{1}_{1}$ Out	—◇	2	Data Addr. Bit 0	—◇	26
MB $^{1}_{2}$ Out	—◇	3	Data Addr. Bit 1	—◇	27
MB $^{1}_{3}$ Out	—◇	4	Data Addr. Bit 2	—◇	28
MB $^{1}_{4}$ Out	—◇	5	Data Addr. Bit 3	—◇	29
MB $^{1}_{5}$ Out	—◇	6	Data Addr. Bit 4	—◇	30
MB $^{1}_{6}$ Out	—◇	7	Data Addr. Bit 5	—◇	31
MB $^{1}_{7}$ Out	—◇	8	Data Addr. Bit 6	—◇	32
MB $^{1}_{8}$ Out	—◇	9	Data Addr. Bit 7	—◇	33
MB $^{1}_{9}$ Out	—◇	10	Data Addr. Bit 8	—◇	34
MB $^{1}_{10}$ Out	—◇	11	Data Addr. Bit 9	—◇	35
MB $^{1}_{11}$ Out	—◇	12	Data Addr. Bit 10	—◇	36
MB $^{1}_{0}$ In	—◇	13	Data Addr. Bit 11	—◇	37
MB $^{1}_{1}$ In	—◇	14	Break^{1} State	—◆	41
MB $^{1}_{2}$ In	—◇	15	Run^{1} State	—◆	42
MB $^{1}_{3}$ In	—◇	16	Break Request	—◇	43
MB $^{1}_{4}$ In	—◇	17	Transfer Direction (Into PDP-5)	—◆	44
MB $^{1}_{5}$ In	—◇	18	Increment Request	—◆	45
MB $^{1}_{6}$ In	—◇	19	SP 0	—▶	46
MB $^{1}_{7}$ In	—◇	20	Power Clear	—▶	47
MB $^{1}_{8}$ In	—◇	21	Data = > MB	—▷	48
MB $^{1}_{9}$ In	—◇	22	Address Accepted	—▷	49
MB $^{1}_{10}$ In	—◇	23	Ground	⏚	50

G. Bell

dec INTEROFFICE MEMORANDUM

DATE January 28, 1963

SUBJECT PDP-3

TO Ken Olsen

FROM Gordon Bell

cc: H. Anderson
D. Mone
N. Mazzarese

Beginning now, A. Kotok should be assigned full time to PDP-3.

The character of a machine influences our growth tremendously since day to day development decisions are always made around existing machines (eg. BBN system).

PDP-3 might be useful if it is: (I'm sure it could be placed in the same space as PDP-1)

1. Built to sell for under $200,000
2. 5 μsec cycle
3. Expandable (similar to BBN system)
4. Capable of running 704, 7040, 7044, 709, 7090, 7094 programs.
5. Built as if we intend to stay with it a while.
6. Entirely serial logic in the processor.
7. Complete systems approach:
   a) allow many memories
   b) allow many processors of various types.
   c) First processors might be very simple with complete trapping facilities to handle _most_ every instruction, and provide only a very skeleton processor.
   d) Provide an encore (made with a faster parallel version).
8. Use new logic (if we have an extra 9 months for the project.)

dec INTEROFFICE MEMORANDUM

DATE February 21, 1963

SUBJECT New Computer Design Philosophy

TO Tom Stockebrand FROM Kenneth H. Olsen

A new computer is long overdue at DEC but we have not been in a position to build one because we have been so long in winding up the details from our present computers. However, now we do have the techniques and the time and the money for a new computer, I think we should go ahead and make one in a reasonably fast time schedule.

The proposal is to do all aspects of the computer design in parallel. This means that at the end of the time schedule whether it is four or six months, the job should be done. Then after a rest of a month or two we could if we wanted to go off and make another computer. Here is a list of the items which should be carried on in parallel:

Design and Build Central Processor
Write FORTRAN with Assembler and Simulator
Design and Build Tape Control Unit
Write All Manuals

We have never looked at competition before but I think as a result we have lost out because we don't know the points in which our machines are significantly better than others. I think that we should consider doing this parallel effort sub-contracting a survey out to someone like I.I.I. to compare our machine in detail with others.

Kenneth H. Olsen

dec INTEROFFICE MEMORANDUM

DATE February 21, 1963

SUBJECT Random Notes on New Computer

TO Tom Stockebrand FROM Kenneth H. Olsen

We received a quote from Amphenol on a 36 pin connector for use in large system plug-in units but this will not work out well because it has to be thicker and therefore will not fit in our standard construction. Loren Prentice is now making a model of a double width plug-in unit which will have two 22 pin connectors on it which will make a total of 44. This looks like a reasonable approach to a large plug-in unit.

Gordon Bell suggests that we do all our register transfers through one common register. This is the way the MTC Computer worked originally. This would cut down the number of gates and they might end up using the very high speed transistor gates.

I asked Bob Savell to consider repackaging the reader, punch and typewriter control panels to make them less expensive. We might put much of it on a very small number of large plug-in units. We might also include the micro-tape logic in the same place.

I told Bob Savell to start working on the new punch timing control for PDP-1 but to plan to have it in the new computer.

Dit Morse feels that the teletype typewriter is a satisfactory typewriter for computer use. He of course would like a more extensive character set but a typewriter that works has a very definite advantage. I can't see that we'll have time to evaluate any other typewriter in time.

Loren Prentice has been working on a new design for the PDP-1 and PDP-4 console fronts. I suggested that they drop all work on that and work on the console front for the new computer. This one should include space for punch, reader, LINC and control panel.

Some people like the idea of having an extra register to store the contents of the accumulator when it is not being used. This would allow the accumulator to be used for index adding and other things. The extra register could then be used as a carry register which would allow very fast multiply. If this carry register is used as an accumulator buffer, the accumulator might then be used as the register which transfers information between registers. Several people have told me they would like to have a pointer register.

We have to decide whether we want indicators on all flip-flops or not. I have asked Jack Smith to estimate what it would cost to add an indicator.

It is a real chore to change cabinet design. Our present mounting panels hold 25 plug-in units and if we move the marginal checking panel, it will hold 26. It would therefore be convenient to keep the digit length of the machine 26 or less bits long.

File – PDP-3

INTEROFFICE MEMORANDUM

DATE February 22, 1963

SUBJECT New Computer Design

TO Ken Olsen

FROM Tom Stockebrand

My apologies for form and content of this memo, it is a rushed job. In particular it does not include enough evaluation of the competition nor enough filtering of the ideas presented. While I am on vacation, I will try to sketch out more of the machine design.

Commitments on delivery dates, price and so on should be to Ken Olsen and the company and not to customers.

This machine should be specifically designed to do the job as listed below superlatively well rather than to in any way "look like the competition" or be an answer for them.

This machine is to fill a vacuum we believe to exist at the present time in the computer market.

We must make no compromises in carrying out the ideas which are involved in its design. The implication of the above is that, as is usual with DEC effort, the ideas shall be limited to those which are eminently easy to do, general, straightforward extensions of the art..... In fact, "today's technology today." ------God.

The sources of the ideas presented in this note are indicated in an effort to provide "source data" while I'm gone. If the general ideas are agreed upon, future administration of the project will be vastly improved.

If we are to turn out machines regularly, we need some more official advanced development - that is answers to specific how-can-we-do-this-job questions. (Coax delays, micro-logic, serial, majority logic circuits, etc.)

THE IMPORTANT NOTIONS

It is time the Programmer was given real power in sub-routine writing ability so that no modifications of instructions are ordinarily necessary during program relocation.

Multi-programming, time sharing, fast break-in or what-you-will is necessary in the eyes of most users of our equipment and in fact necessary (though they don't know it) to many users who are comtemplating using our equipment.

Data words need to match today's data requirements in accuracy. The analog people are almost entirely concerned with 14 bit accuracy for what they call four significant digit precision.

Large memories are here. Index registers are here.

Some fair expansion of the machine should be planned for at the beginning though we understand that wholesale revisions of the machine are out of order.

The rest of this memo is a list of specifics pertaining to the generalities listed above.

Routine Relocation Power - The ability to operate routines wherever they may be located in memory after a dump from, say, the drum can be provided by the ability to (1) modify each memory reference by a constant while (2) checking that result against specified bounds and trapping to a particular memory location or executive program if the required location is outside of the bounded area. This feature can be achieved reasonably easily during the initial design of a machine by allowing the index adder, or its equivalent, to do the work. Dit says this feature would make programming "ten to a thousand times easier." Ed says that if you can use the arithmetic element more and memory less, you're way ahead and this feature would leap in that direction. (Dit, Shelley, Kotok, Ed and Ben.) This feature is considered by advanced type people to be crucial to the machine design.

Trapping - Trapping meaning to execute and instruction located at, for instance, the address indicated on the op code. This trapping would be done on non-used instructions or memory addresses outside of the bounds set by the executive routine in the relocation of power indicated above (Dit, Ben.)

Character Handling Power - The ability, in one form or another, to address characters stored in memory hopefully to deal with character strings in I/O transfers such as is done in the Lisp and Comet Programs. Dit, Ben and Ed are in favor of this, Ivan goes even further and says that bit addressing features are of great power. However, Len disagrees.

On Obsolescence - Trapping also allows optional expansion by do-it-now-with-program, later with wires. Also de-bugging and checking power is automatically incorporated. The machine should be built of modular parts of course like different memories and AE's and an extra bit or two should be assigned in the instruction word for future variations not thought of now when you absolutely have to have that bit!

Multi-Processing - Multi-Programming - First and foremost, a fast break - this means primarily no need for many accesses of a clean-up variety to store away stuff in preparation for operations in response to a break request. The most potent feature here seems to be an extra register in the AE to allow either exchanges with the AE for saving purposes, or as an address calculator (Dit) or as a multiply index by, or as an addend register, or as a carry register depending on your exact orientation. The second thing which would help this process out is probably a separate index adder though I believe a machine try should be made to use one adder for everything. Since it is reasonably certain that two groups of wide modules will be used, however, it is probably not unreasonable to suggest the index adder. In the future, that means perhaps with the development of another machine, separate program counters may be in order. For now, core program counters should certainly be enough if they are necessary. To hell with data gather. The idea here is to eliminate control problems from the channel and put them in the program where they belong.

Channels should be only high-speed data gathering devices. (Dit) System capability is an okay phrase. (Dit)

List Processing - This is a program technique which has general power which goes well together with our ideas of a processor with general power. It requires index registers and increment and decrement by more than one and, ideally, registers which can be packed with several addresses each (that is, word length equal to two times the address length.) However, I think a clever use of the relocation feature or of Dit's multiple indexing (1+ 2+ 4 scheme) will allow the shorter pack base address that this too short word machine will have. (Len) In general, this processing seems to be for the next machine though a small look into the future is probably in order. Similarly, floating point AE's will probably have to wait until the next machine or at the very best, be planned as a different kind of AE attachable to this memory.

Index Registers - These are clearly necessary. Dit feels that three register which could be added together in a micro-program fashion that is, any combination of the three according to MACRO programmed bits in the word, would be of more use than seven registers addressed directly by the same three bits though Kotok disagrees. I have no feelings. Whether the three could be added together and in fact the complete design of the index adder might depend crucially on the ability to build a simple circuit which would detect four out of seven to provide carry for carries. If this circuit were easily available I believe that five registers could be added together simultaneously and stored in a fifth and the sketch accompanying this memo shows the powerful use that could be made of this feature.

Addressable Registers - These would be very useful according to Len for much easy processing without complicated instruction and could perhaps be implemented to do the character addressing without using extra bits in the word by allowing certain kinds of character type transfers between registers. The most important addressable registers would perhaps be the in/out registers such as, for example, the scope buffer for use with the light pen -- especially if it were an incremental scope plus generator type. In this case too, the feature would allow sine, cosine and hyperbolic and parabolic function generation with no extra hardware. It would save on the IOT read-in bits but cost some address decoding.

Data Channel - Fast break SI, Data Channel SI, I/O Channel, no, - do it with program. (Dit)

Cute Instructions - Ben feels that load and deposit AC in push down list would be a useful instruction at least to the prospects of a clever turn of mind if not to real users. Instruction (Y+)AC)) ---- AC is reasonably necessary for multi-dimentional matrices when indexing is not readily available and would implement easier list processing. Ben likes an instruction called execute effective address however, Len doesn't go along with him. Dit makes the comment that we should avoid doing things in little pieces.

Word Length - There are two criterion for word length, one is the data word that will usually be of necessity, and the other one is the number of bits that you need in your instruction. For floating point work, 48 bits seems to be a minimum and for graceful manipulation of the text this also seems like an appropriate word length. I do not believe that it is necessary to have precisely a multiple of six though this may be, in some cases, graceful for character processing. Many people would just love to have an extra bit or two to indicate whether this set of characters is to be considered in the list and for other marking purposes, ask Dit for example. I, myself, have run into this problem many times when programming character strings. Len will also agree I think. As far as the packaging limitations go, I agree that it is essential to keep the packaging the same which means no more than 25 units in a rack panel wide; notice that if the address portion is 16 or 17 bits, even, there are 8 bits left over in the mounting panel supporting the "short-word" AE in which to provide extensions of the full register portion of the AE. Since the floating point people need 48 bits and we can't possibly take this much of a jump in the present machinery, we should either leave them out of consideration or consider two-word data accesses floating point words. To this end, Dit suggests a single bit in the data words to tell whether the word is to be interpreted as floating point or not. This might be an example of the use of a spare bit location in the word for use when a floating point processor might become available. How about word lengths for ordinary users of fixed point type calculations? The competition seems to feel that 24 bits is a reasonable length however, I submit that in many practical cases 14 or so bits is a reasonable length based on my discussions with various analog and hybrid types. This is because 14 bits represents four decimal digits which is the current okay number in that industry, though there as here okay numbers do not necessarily represent the best in engineering philosophy or power. Analog people further state that they need higher data rates than we can get and if we are to capitalize on our parallel computing and data handling power in order to try to overcome some of the taint of the current serial flap, we should consider, I think, 28 bits minimum so as to be able to pack two 14 bit words per register and thus, double our data output rate to digital to analog converters and the like - also to scopes.

Now on to word length as determined by the instructions. Certainly 16 bits represents a reasonable address length to address 65 kilowords of memory. Everyone agrees that this would be a desirable number. 3 bits for index register seems about right and one bit for deferring. 6 bits seems like a minimum for op code, 1 bit for a programmed operator - primarily to catch up to the competition of SDS. I insist on one spare bit and many people who feel character addressing is important would want to use my spare bit plus two others to do the character addressing in those instructions where it matters, and leave it for instruction modifications where it does not matter. This would give a total of 28 or 30 bits depending how you look at it. If you really believe that there should be a multiple of six, then I would recommend a 30 bit machine. However, 28 bits I think is my current recommendation. Incidentally, if you allow 7 bit characters for 128 character set, which is quite a reasonable number, and a "step forward",then this even meets the criterion that 2 bits of character addressing is enough and comes out even. In any case we have room for 33 bits and 17 address bits in the two mounting panels which have double trays so this gives us three extra slots for odds and ends.
I STRONGLY RECOMMEND A 28 OR A 30 BIT WORD.

Concurrent Programming - In this area I am not an expert but Dit seems to feel that the FORTRAN four language, which looks like the ALGOL language is the language to use for all programming. I am not aware of the details of the character set required or like that.

He wants to do it all in ALGOL. I would have a good discussion with Dit on the subject. All agree that a full-time programmer should be working from the start of the project.

More Work - Very soon, more work should be done in the following areas before the design is completely hard.

1. A careful compilation and discussion of the competition's ideas and features, also of LINC and other semi-competitive machines.

2. Whether an analog input is a necessity - I believe it may be.

3. Whether serial methods of computation would give us any real advantage. It may be that in the shorter worded index adder, the multiple additions that will sometimes go on could be done very efficiently this way in the event that a majority logic circuit did not work out as a good idea. This would allow many additions in only the time to circulate one word plus N extra bit times. Furthermore, I am not sure of the best AE design. I am convinced that we should have one programmer (hopefully Lennie) working full time along with the design of this machine so that it is on cards or back panel wiring or like that right from the start. This, I think, will eliminate in the future bottle necks which we are certainly going to run into if we plan to turn out new type machines regularly.

Conclusions -

Relocation
Independence of AE and Memories
Trapping
Time Sharing or Multi-Processing or Addressable Register or Multi-Programming
Character Handling Power

I think a tentative example of the breakdown of parallel tasks in the developments of this machine would be somewhat as follows:

1. Programming with a good man such as Dit

2. Manual Design and Development along with the development of the machine with Stu Grover

3. AE design under Dit and Gordon

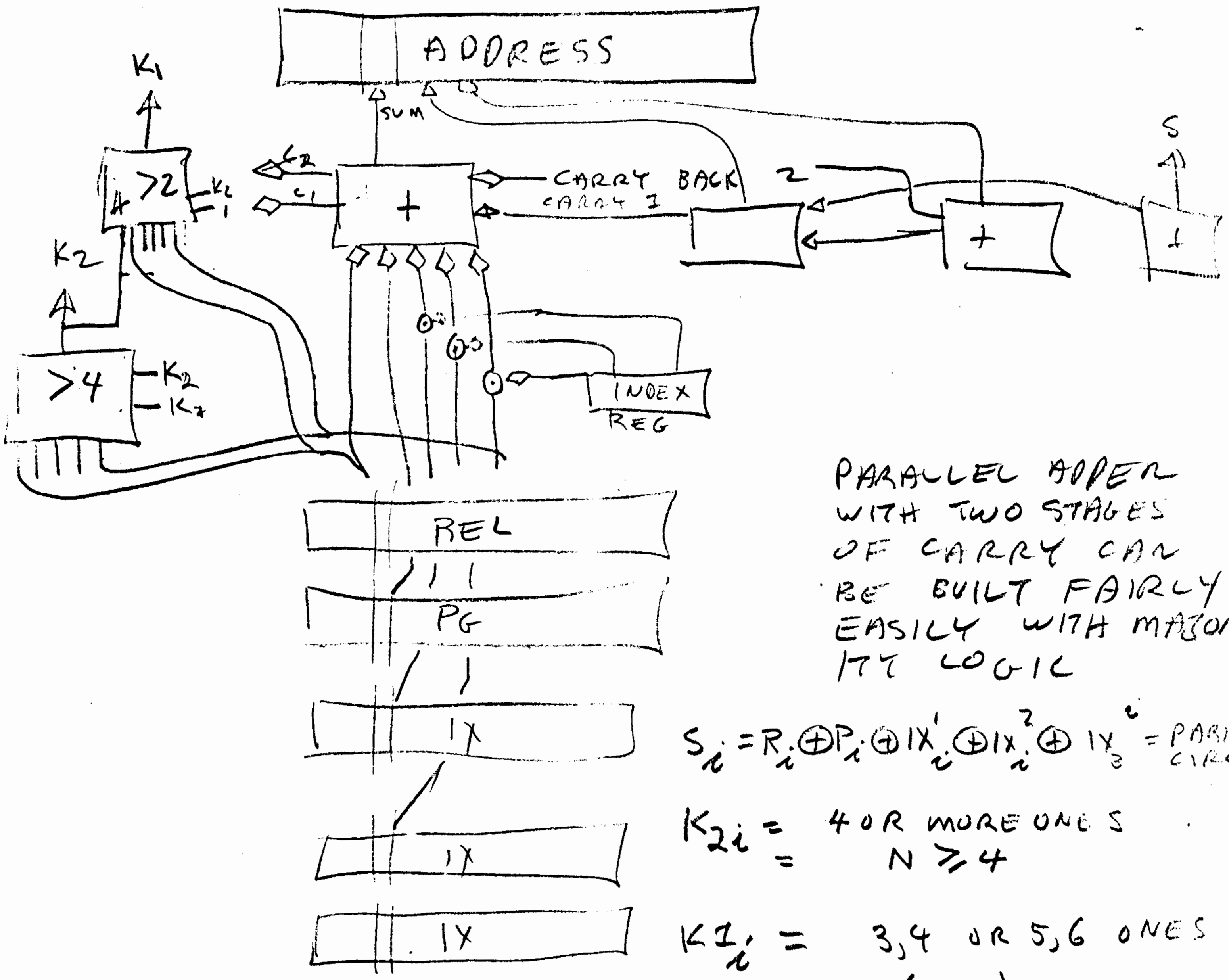
4. Machine design under Gordon and I

5. Programming toward aiding the design of the machine under Len

6. A small amount of research under Emile or Russ Doane in the form of coaxial serial parallel conversion and multi-plexing and majority logic circuitry.

7. I/O development under Roland Boisvert or perhaps even better Mel Arsenault.

PG PARALLEL



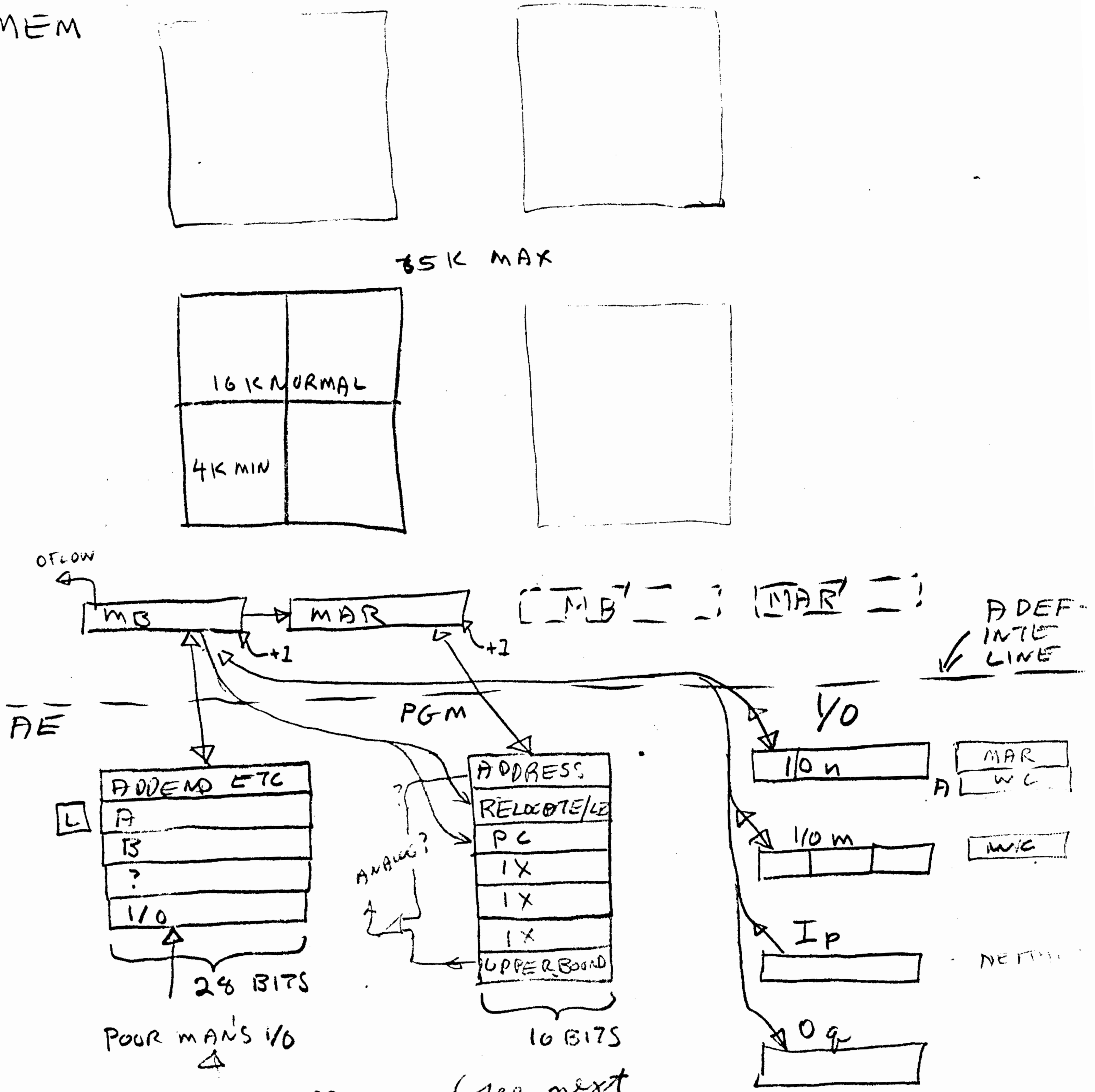
PARALLEL ADDER WITH TWO STAGES OF CARRY CAN BE BUILT FAIRLY EASILY WITH MAJOR ITY LOGIC

$S_i = R_i \oplus P_i \oplus IX_i^1 \oplus IX_i^2 \oplus IX_3^i$ = PARITY CIRCUIT

K_{2i} = 4 OR MORE ONES
= $N \geq 4$

$K1_i$ = 3, 4 OR 5, 6 ONES
$= -\left(\frac{K_{2i}}{2}\right) + N \geq 3$

FLOW



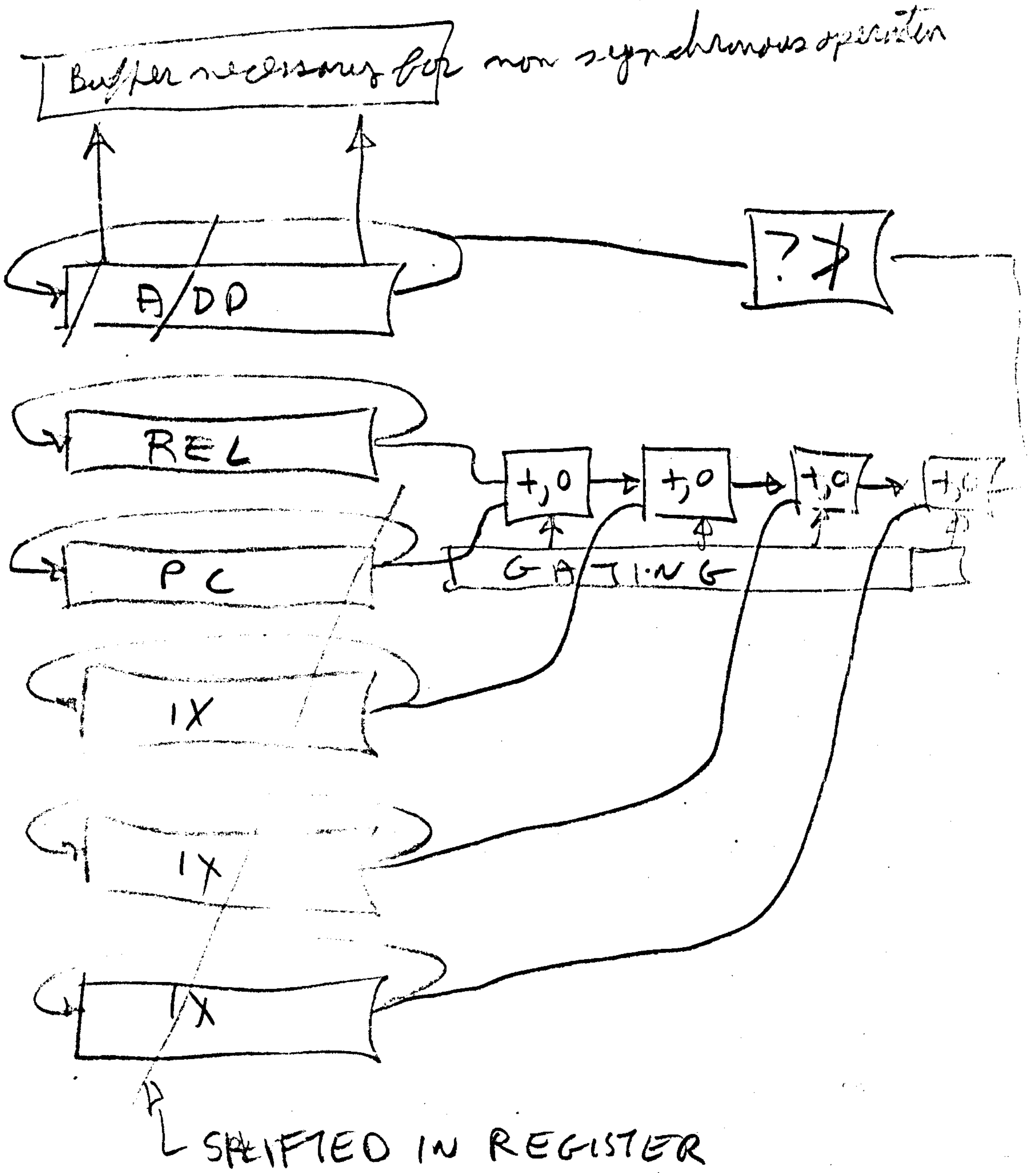
TRANSFERS MB → ADD
A → MB
A → B

(see next page for serial Parallel)
MB → REC
MB → PC

DATA REQUST CYCLE
TYPE A, B, C, D { WD CT, MAR, DATA, PGM

MAXS NEXT MACHINE [illegible]
DES = DIFFERENTIA EQN SOLVER

SERIAL



G Bell ✓

Questions:
1. Memory too small address size
2. Kludg (1 x register in 16 bit mode)
3. 8 bit a bitch to implement (all control)
4. 32 " " " " for 8 bit mode
5. poor 32 bit, hot for an 8 bit, good for a 16 bit, but lots of needless instrs.
6. Can't do address arith with X ic.
7. Bad shifts
8. Tim Share provision
9. FL PT?
10A 2 addresses don't fit in a word.
10B addressing is bad
11. Indexing doesn't work, must X by 4.
12. No form for pure procedures
13. 6 disparities
~~a. Bytes (ok)~~
~~b. BLF~~
c. BLKI-O
d Test (2)
e Boolean (64)
f. XR stuff (AOBJF
g. negate, magnitude
h integer arith.
i. exch
j. FSC
k good tests
l. arith compares
m push/pop

INTEROFFICE MEMORANDUM

DATE October 14, 1965

SUBJECT PRELIMINARY THOUGHTS ON A NEW COMPUTER LINE

TO N Mazzarese FROM E De Castro

I believe that we should start fairly soon to develop both hardware and software for a completely new line of small computers. Our current machines, because of their limited organization, have made it impossible for us to add features which cost very little and yet are standard equipment on most competitive machines. The following are some of the most predominant deficiencies in our line:

1. We are unable to offer our customers the ability to replace a small machine with a larger one as his requirements grow without asking him to undertake a complete reprogramming job.

2. We do not have a full line and therefore are precluded from a fair segment of the market.

3. We have yet to build a computer small enough and inexpensive enough to fully satisfy the OEM, educational and small laboratory markets.

4. We do not have compatible interfaces and therefore must develop and maintain different peripherals for each computer.

5. We do not have program compatibility and as new programming concepts evolve or new applications areas become interesting we must either duplicate our efforts or forego the competitive advantage on one machine or the other.

Completely replacing a computer line is certainly a large undertaking but we now have several advantages which we have not enjoyed during the recent past. These are as follows:

1. A large order backlog for standard products which can be produced with a minimum of engineering assistance.

2. A competitive line which with only minor modifications can probably be sold successfully for another year.

3. An adequate programming system which, although not fully competitive is complete enough so as not to detract seriously from sales in the short run.

4. Sufficient personnel in the small computer group capable in circuit design, system design and programming.

If we are going to avoid serious fluctuations in our production rate and still allow development to be done in a thorough and orderly manner we must start now to plan the products which will take over as PDP-7 and 8 phase out.

DESIGN OBJECTIVES

For a new computer line to be successful in the market it must meet several objectives some of which are in conflict and therefore compromises must be made. We must have a low cost basic configuration yet it must not be so inept that peripherals are prohibitively expensive or extremely unwieldy to attach. We must have machines that closely approach theaccepted standards yet not so complex in organization that we are unable to sell at a price slightly below that of competition for a computer of equal memory speed and word length. We must do everything possible to get the most mileage out of our engineering and programming effort. To further this objective central processors must all have an identical interface so that one line of peripherals may be designed to connect to any processor. C.P. organization should be such that software may be transferred without change from one machine to another. In achieving this degree of compatibility we must not make it impossible for efficient programs to be written for each machine in the series although this does not mean that the most efficient program for one machine is necessarily optimum for another.

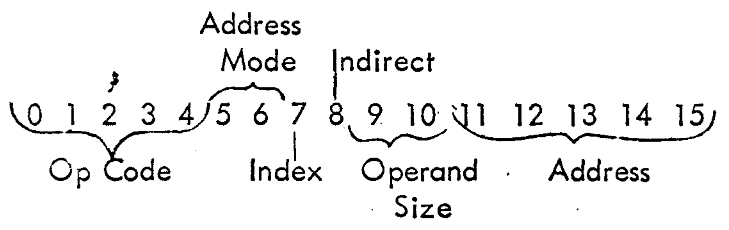
GENERAL CHARACTERISTICS

The line should consist of three computers having word lengths of 8, 16 and 32 bits respectively. Each machine will have a parallel memory and be capable of performing arithmetic and logical functions in parallel on operands equal to or smaller than the basic word length. In addition the two smaller machines will be able to perform 16 and 32 bit operations by processing operands in serial. For example, if the small machine were programmed to add two 32 bit numbers it would make 4 calls on memory to obtain operands and would add each 8 bit segment individually to the appropriate section of the accumulator using the same adding circuitry for each step. The 16 bit machine would require only two such steps. To achieve compatibility in the other direction the larger machines will be capable of dealing with words consisting of 1, 2 or 4 - 8 bit bytes. Thus the op code which causes the small machine to add a single word will be interpreted by the large machine as a command to add a single byte.

It is desirable to make the 32 bit machine capable of performing some instruction which will not be included in the repertoire of the smaller ones. To maintain compatibility all unused op codes will trap, i.e., cause the program to branch to a fixed location where a subroutine to simulate the non-existant instruction may be located. Some additional storage is thus required in the smaller machines to simulate these instructions.

INSTRUCTION FORMAT

All instructions are either 16 or 32 bits in length and are fetched from memory in 1, 2 or 4 cycles as required. The small machine must make at least 2 references to memory for each instruction while the large machine may have 2 instructions in a single word. The 16 bit memory reference instruction word format is as follows:



The 32 bit word format is:

```
                                Index Register
            Address                Selection
               Mode  Indirect
 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
 Op Code  Index Operand                             Address
                 Size
```

Crock

"The OP Code portion" is used in the traditional sense and merely selects the instruction to be performed.

"The Address Mode" is decoded as follows:

0 = Immediate i.e. operand is contained in the next 2 bytes immediately following the instruction or in the same word on the 32 bit machine.

[illegible]

1 = Relative forward. Add the contents of the address portion to the current P.C. to obtain the address of the operand.

2 = Relative reverse. Subtract the contents of the address portion from the current P.C. to obtain the address of the operand.

3 = Full address. Fetch the next two bytes to obtain the address of the operand.

Modes 0, 1 and 2 specify 16 bit instructions whereas mode 3 specifies a 32 bit instruction.

"The Index bit" if a one indicates that the contents of the index register will be added to the address after any relative address calculation has been made.

"The Indirect bit" specifies deferred addressing in the usual sense. Multi level indirect addressing is possible. During a defer cycle the address mode, index and indirect bits of each word are obeyed.

"The Operand Size portion" indicates that the operand will be 8, 16 or 32 bits long.

"The Index Register selection bits" allow any one of 8 index registers to be specified in the full address mode. In any other address mode only index register 0 may be used.

"<u>The Address portion</u>" is used to select the first of the 1, 2 or 4 bytes which will be used as the operand. Thus in the 8 bit machine the address portion is equivalent to the memory address. In the 32 bit machine the least significant 2 bits are not used to address memory but rather are used as a byte pointer to select the desired portion of the word.

INSTRUCTION REPERTOIRE

The instruction set is designed to be complete but straightforward. Many of the instructions can be implemented at very small cost over and above the most basic useful set because they use existing gating and transfer paths. The following list represents a starting point and probably can be improved upon. Instructions are grouped by major function.

1. Memory Reference

   Arithmetic

   - Add to accumulator (#)
   - Add to memory
   - Subtract from accumulator
   - Multiply (optional)
   - Divide (optional)

   Logical

   - AND
   - Inclusive OR
   - Exclusive OR

   Store and Load

   - Load Accumulator
   - Store Accumulator
   - Store Zero in memory
   - Load MQ (optional)
   - Store MQ (optional)

   Index

   - Increment Memory and skip if 0
   - Decrement Memory and skip if 0

   Compare

   - Skip if same
   - Skip if different

Branching

- Jump conditional #1
- Jump conditional #2
- Jump to subroutine
- Jump and save P C in index register

In-Out

- Transmit memory on IO bus
- Transmit IO bus to memory
- Test and jump

Miscellaneous

- Execute

2. Augmented instructions

Shifts and Rotates

- Logical Shift right (1 or 8 places)
- Logical Shift left (1 or 8 places)
- Arithmetic Shift right (1 or 8 places)
- Rotate left (1 or 8 places)
- Rotate right (1 or 8 places)
- Long Shift right (optional) ✓
- Long Shift left (optional) ✓
- Normalize (optional) ✓

Clears and Complements

- Clear accumulator
- Complement accumulator
- Clear overflow
- Complement overflow

Coun ting

- Increment accumulator
- Decrement accumulator

Miscellaneous

Halt

Read switches into accumulator

In-Out

Select device

Transmit AC on IO bus

Transmit IO bus to AC

Most of the instructions listed above are quite conventional. However the jump instructions require further explanation. Since the operand size portion has no meaning for these instructions it will be used to specify the condition for jumping. Conditions are decoded as follows:

Jump #1

0 = unconditional

1 = if AC = 0

2 = if AC $\neq$ 0

3 = if overflow = 1

Jump #2

0 = if AC is positive

1 = if AC is negative

2 = if overflow = 0

4 = not used

Test and Jump

0 = if device flag 0 is a 1

1 = if device flag 1 is a 1

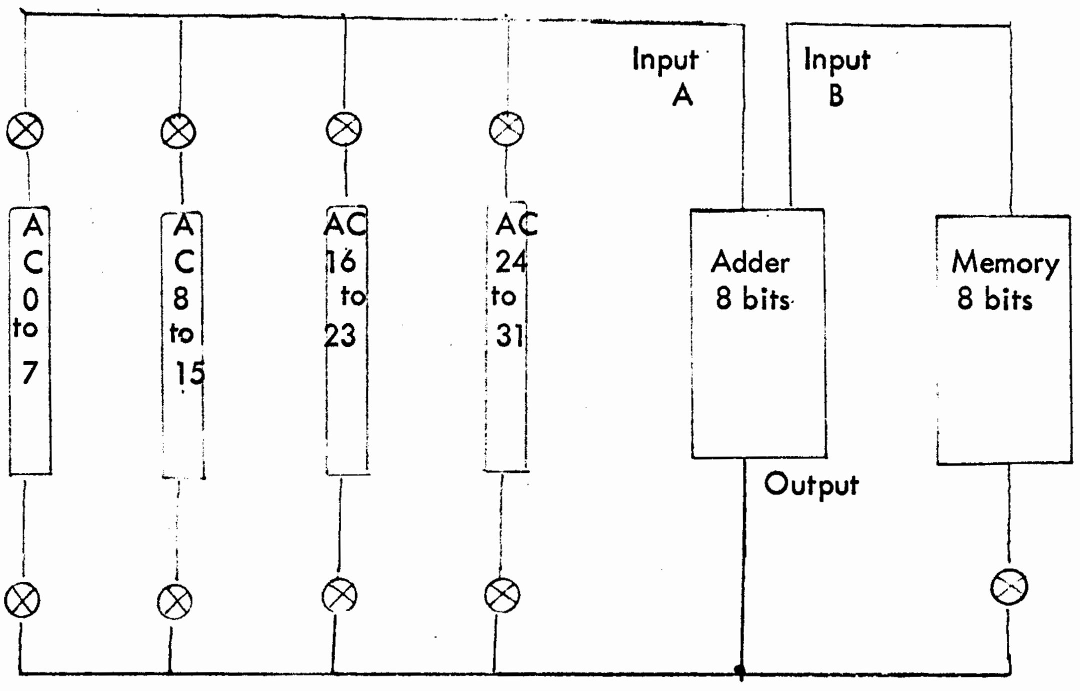
2 = if device flag 2 is a 1

3 = if device flag 3 is a 1

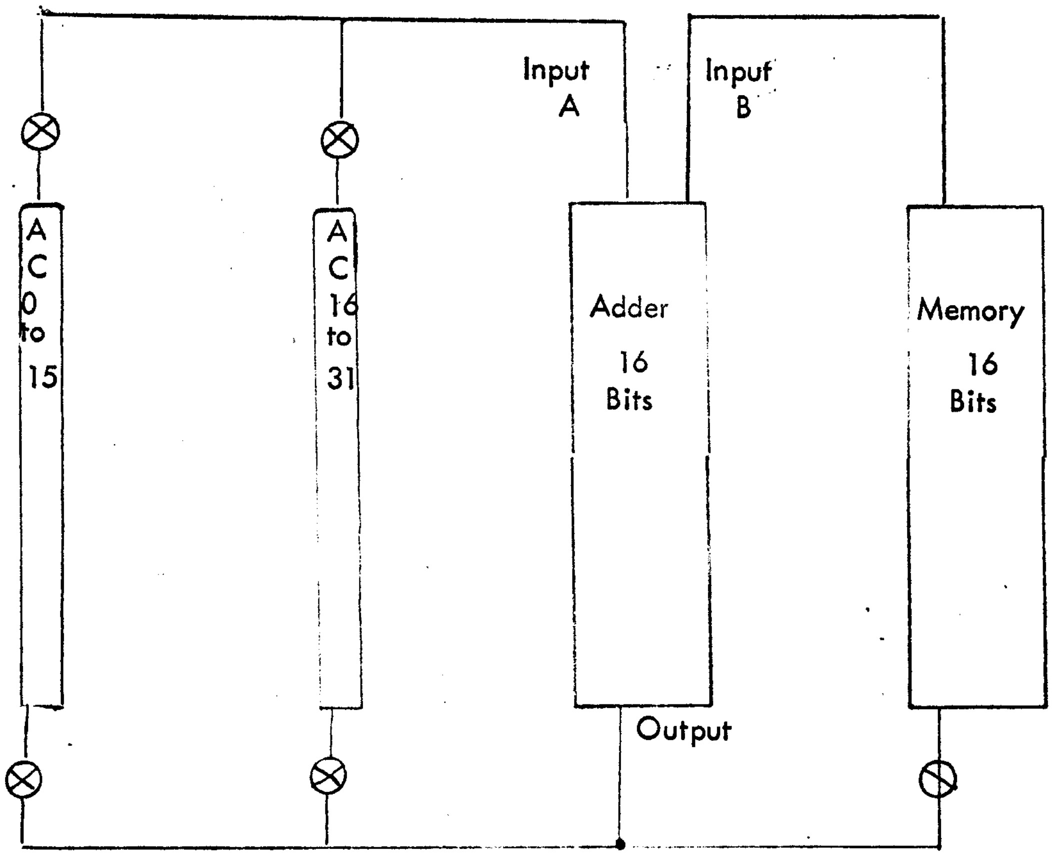
DATA HANDLING

Internal data is normally handled by moving it from memory to the accumulator where it is processed and then returned to memory. In all machines the accumulator is a full 32 bit register. However its organization and transfer paths differ. The block diagrams below illustrate the organization of each member of the family.

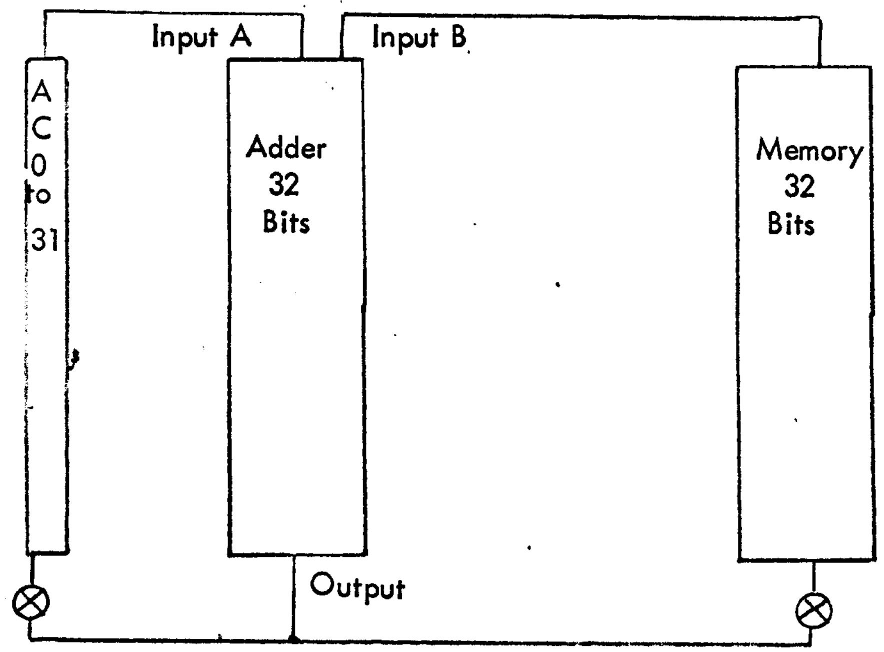
8 Bit Organization



16 Bit Organization



32 Bit Organization



It can be seen that in order to process a 32 bit number with an 8 bit machine, 4 passes must be made through the adder in serial. This of course takes 4 times as long but also substantially reduces the cost since all of the complex operations are done in the adder. The accumulator flip flops themselves are really quite simple and inexpensive. Carries out of any of the lower order portions of the accumulator will propagate into the next higher order part. Carries from the most significant bit will set the overflow flip flop.

INDEXREGISTERS

Eight index registers are provided and are normally located in core memory. They may however be replaced by flip flop registers as an option. Each index register is 16 bits long including a sign bit. During an index cycle the sign bit will be obeyed, i.e., if it is negative the index register will be subtracted from the address. If it is positive it will be added. In addition if subtraction is specified and the index register is equal to 0 the next instruction will be skipped.

INPUT OUTPUT

All IO operations will be done on a bus system. Data transmission is normally accomplished as a 2 step operation. The first step is to load the selection register and the second is to transmit the data. The selection register is 8 bits long and its contents are transmitted to each device. Whenever a device recognizes its own code on the selection lines it will make a DC connection to the bus. Actual data transfers may be made with the accumulator using an augmented instruction or with memory using a memory reference instruction. If the transfer is with memory the instruction may be indexed and thus blocks of data may be conveniently transmitted or received. Either 1, 2 or 4 bytes will be transferred depending on the operand size portion of the instruction.

Device status may be tested by use of the test and jump instruction. This instruction will sample any one of 4 status lines on the IO bus. Since the selected device will have previously connected its status information to the bus the program may be branched in accordance with any of 4 different conditions from any of 256 devices.

ADVANTAGES

An organization along these lines gives us many advantages in return for a small amount of added complexity to maintain compatibility. The most important of these are as follows:

1. A 32 bit arithmetic capability. This will drastically reduce the amount of double and triple precision computations required and thus speed processing and reduce storage requirements.
2. A fairly powerful order code structure which will allow us to write programs to operate in smaller memories.
3. A more efficient method of handling data which allows easy character packing and does not require use of more memory than necessary for data of a given length.
4. A full line with the possibilityof replacing a small machine with a larger one as requirements change.
5. A fully compatible line of peripherals which may be transferred from one machine to the next if the processor is replaced. This will also reduce the engineering cost of peripheral equipment.
6. A fully compatible programming system. This will allow us to invest all of our programming effort in a single language and thus we will be able to develop better software at lower cost.
7. Reduced module costs since all machines will use the same circuits and thus volume will be much higher.

EDEC:ASJ

CC
K H Olsen, J Jones, R L Best,
G Bell, L Hantman

G. Bell

dec INTEROFFICE MEMORANDUM

DATE December 7, 1966

SUBJECT Proposal for the PDP-14X$_{GB}$ - Logical Structure of the 16 bit Processor.

TO		FROM
K. Olsen	E. DeCastro	Gordon Bell
N. Mazzarese	M. Ford	
W. Hindle	H. Burkhardt	
S. Olsen	J. Jones	
S. Dinman	L. Portner	
A. Kotok	T. Johnson	
L. Seligman	R. Lane	

Having attended a rather hectic, but stimulating meeting, at DEC on November 23, 1966, I decided to write down thoughts about the machine(s), generally. Those are included in the memo "New Machines Design Parameters". That memo deals with parameterizing the design, with attempts to list the goals. Having gone that far, I couldn't resist trying to specify a machine, and that's included.

The most important decisions in the machine(s), I believe, are:

1. Index Registers
1.a Are Index Registers, AC, MQ identical, general?
1.b Number of general registers?
2. Addressing Storage, how many modes? The desirable abilities are:

 a. Using a 32 bit instruction, directly address any work in memory, in connection with at least one index register. The instruction should be contiguous, so the assembler doesn't have to worry about building the 2nd half of it (with the address part) somewhere else nearby.

 b. Be able to transfer to a nearby address using a 16 bit instruction (nearby = -16 ± 64 words).

 c. Pick up common 16 or 32 bit constants or data nearby for a common routine in a 16 bit instruction.

 d. Get at least a constant or immediate data of 2^5 for directly specifying shifts, selecting an I/O device, etc. in a 16 bit instruction.

e. Directly or indirectly address any of the general registers in a 16 bit instruction.

f. Address such that temporary data is stored in an "impure part" so that subroutines are all re-entrant.

g. Provide "immediate" data in a 32 bit instruction to avoid having assembler page difficulties.

3. Calling subroutines - can the subroutines be naturally re-entrant? Need they?

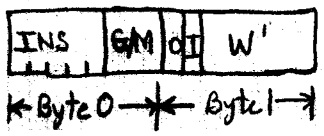
4. Extra codes/SYSPOP/UUOS/or Programmed Operators - Can these be implemented so that desirable order codes be implemented with little overhead in time, and interpretive programming provided for?

5. Address space - Is 2^{15} or 2^{16} large enough for fore-seeable market?

6. Multiple users? Protection and Relocation Scheme.

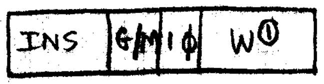
7. Should page or relative addressing be used for short addresses?

BASIC INSTRUCTION LAYOUT PDP-14X$_{GB}$



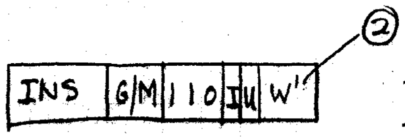
DIRECT OR INDIRECT USING THIS PAGE

I $\begin{cases} \text{DATA} \leftarrow \text{Memory (This page + W)} \\ \text{DATA} \leftarrow \text{Memory (memory (this page + W))} \end{cases}$



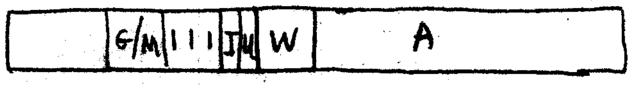
IMMEDIATE DATA

DATA ← W



DIRECT OR INDIRECT TO GENERAL REGISTER (W')

I $\begin{cases} \text{DATA} \leftarrow \text{memory } (W') \\ \text{DATA} \leftarrow \text{Memory(memory } (W')) \end{cases}$



DIRECT OR INDIRECT, WITH INDEXING, TO SPECIFY DATA

I③ $\begin{cases} \text{DATA} \leftarrow \text{Memory (A + General Register (w'))} \\ \text{DATA} \leftarrow \text{Memory (Memory (A + General Register (w')))} \end{cases}$

INS = INSTRUCTION PART

G/M = General REGISTER / MODE SELECTOR

I = INDIRECT BIT

W = WORD TO SPECIFY ADDRESS ON THIS PAGE OR DATA

U = UNUSED OR DIRECT DATA OR AUTO-INDEX BIT

W' = SHORT WORD TO SELECT A GENERAL REGISTER

A = ADDRESS LENGTH DATA

① Preferrably sign extended ③ may or may not repeat indirect
② Memory (w') ≡ GR(w')

CGB 12/3/66

Expanded INSTRUCTION LAYOUT PDP-14X$_{GB}$

INS	G/M	00	W			DATA ← Memory (THIS PAGE + W) – DIRECT TO PAGE
← byte 0 →		← byte 1 →				
INS	G/M	01	W			DATA ← Memory (Memory (This Page + W)) – INDIRECT TO PAGE
INS	G/M	10	W			DATA ← W① – DIRECT DATA
INS	G/M	110	00	W'		DATA ← Memory (W') – DIRECT TO Registers (or memory)
INS	G/M	110	01	W'		DATA ← Memory (memory (W')) – Indirect to registers (or memory)
INS	G/M	110	10	W'		③
INS	G/M	110	11	W'		③
INS	G/M	111	00	W'	A	DATA ← Memory (A + General Register (W')) – Direct to memory, indexed.
					← bytes 2, 3 →	
INS	G/M	111	01	W'	A	Data ← Memory (Memory (A + Gen. Reg. (W'))) – Indirect to memory indexed④
INS	G/M	111	10	W'	A	③
INS	G/M	111	11	W'	A	③

① Preferably sign extended
② Memory(W') ≡ GR(W')
③ (Unused or Direct Data or Auto-index) bit
④ MAY OR MAY NOT Need Repeated indirect

CGB-12/3/66

LOGICAL MACHINE STATE

; General Registers
<u>Register Array</u> General Registers [0:15, 0:7] ; General registers

; Program Counter
<u>REGISTER IDENTITY</u> PROGRAM COUNTER [0:15]; General Register [0:15, ϕ]

; MEMORY
<u>Register array</u> memory [0:15, 0: 177777_8] ;

<u>Register array identity</u> memory [0:15, ϕ-7], general Registers [0:15, ϕ-7]

; FLAGS
<u>BIT</u> OVERFLOW FLAG ; ARITHMETIC OVERFLOW
<u>BIT</u> CARRY FLAG; IF SUM $\geq 2^{16}$
<u>BIT</u> ZERO FLAG ; IF NUMBER $\equiv$ 0
<u>BIT</u> SIGN FLAG ; 1 IF NEGATIVE

<u>BIT</u> INTERRUPT ON
<u>BIT</u> INTERRUPT PROCESSING

<u>Operations</u>

1. TWO'S COMPLEMENT NUMBERS
2. FLAGS ABOVE <u>MAY</u> BE SET ON* INSTRUCTIONS AS A FUNCTION OF THE RESULTS

Instructions with a GR - Part, and Operands

	LOAD	$GR(G) \leftarrow DATA$
	ADD*	$GR(G) \leftarrow GR(G) + DATA$
	SUBTRACT*	$GR(G) \leftarrow GR(G) - DATA$
	OR*	$GR(G) \leftarrow GR(G) \vee DATA$
	AND*	$GR(G) \leftarrow GR(G) \wedge DATA$
③	XOR*	$GR(G) \leftarrow GR(G) \veebar DATA$
	COMPARE*	$GR(G) - DATA$
①	MULTIPLY*	$GR(G), GR(G+1) \leftarrow GR(G) * DATA$
①	DIVIDE*	$GR(G), GR(G+1) \leftarrow GR(G), GR(G+1) / DATA$
①	ROTATE	$GR(G) \leftarrow f_1(Data, GR(G))$
①	SHIFT*	$GR(G) \leftarrow f_2(Data, GR(G))$
③	ROTATE DOUBLE	$GR(G), GR(G+1) \leftarrow f_1'(DATA, GR(G), GR(G+1))$
③	SHIFT DOUBLE	$GR(G), GR(G+1) \leftarrow f_2'(Data, GR(G), GR(G+1))$
	STORE	$DATA \leftarrow GR(G)$
②	LOAD STACK	$GR(G) \leftarrow GR(G)+1;\ Mem(GR(G)) \leftarrow DATA$
②	LOAD STACK, JUMP	$GR(G) \leftarrow GR(G)+1;\ Mem(GR(G)) \leftarrow PC$; $PC \leftarrow DATA$
②	STORE STACK	$DATA \leftarrow Mem(GR(G));\ GR(G) \leftarrow GR(G)-1$
②	STORE STACK, RETURN	$PC \leftarrow Mem(GR(G)) + DATA;$ $GR(G) \leftarrow GR(G)-1$
③	CALL OPERATORS-TYPE 1 ([illegible])	$GR(G) \leftarrow GR(G)+1;$ $Mem(GR(G)) \leftarrow PC;$ $GR(G) \leftarrow GR(G)+1;$ $Mem(GR(G)) \leftarrow DATA$; $PC \leftarrow OPCODE + FIXED\ LOCATION(S)$
④	CALL OPERATORS-TYPE 2	$Mem(FIXED) \leftarrow PC$; $Mem(FIXED+1) \leftarrow DATA$; $PC \leftarrow FIXED+2$

See Key on next page

each instruction is to a different location (fixed)

cgB/12/3/63

INSTRUCTIONS WITH An M(MODE) PART

JUMP (M = CONDITIONS OF FLAGS TO JUMP ON) (ϕ,-, flags)
IF f(M, FLAGS = 1) THEN PC ← DATA ELSE NULL.

3 EXECUTE (M = CONDITIONS OF FLAGS)
IF f(M, FLAGS = 1) THEN INSTRUCTION ← Memory (DATA) ELSE NULL.

RESET Memory WITH AN M-MODIFIED ZERO
DATA ← f(M, ϕ) (yields 0, +1, -1,
RESET Memory WITH AN M- MODIFIED memory
$\text{DATA}^* \leftarrow f(M, \text{DATA})$

m = 3 BIT MICRO CODE
$1_{t=0}$ - COMPLEMENT
$1_{t=1}$ - +1
$1_{t=2}$ - COMPLEMENT
} yields: null, +1, -1, COMPLEMENT NEGATE

JUMP TO SUBROUTINE (M specifies: storage of FLAGS, INTERRUPT RESTORE

Memory (DATA) ← FLAGS ← mode select 1
Memory (DATA+1) ← PC
PC ← DATA + 2

3 RESTORE FROM SUBROUTINE
M = RETURN WITH/WITHOUT FLAGS. 2 - Un-necessary UNLESS THE PREVIOUS INSTN IS FULLY IMPLEMENTED

IN/OUT M = 8 CONDITIONS LIKE PDP-10

W = DEVICE SELECT

Memory (PC+1) = DATA POINTER TO TRANSFER

KEY
* - sets FLAGS
1 - OPTIONAL - (ROTATE/SHIFT WOULD THEN BE 1, OR 2 BITS)
2 - DESIREABLE
3 - NICE
4 - MANDATORY - IF AVAILABLE, THEN 2 and 3 COULD be sub-routines

cgb 12/3/68

G. Bell

INTEROFFICE MEMORANDUM

DATE February 6, 1967

SUBJECT Possibility of making many Peripherals at DEC with a Common Interface to all present and future computers.

TO

Ken Olsen
Nick Mazzarese
Win Hindle
Stan Olsen

FROM

Gordon Bell

CC: A. Kotok
R. Savell

From time to time this has been considered, but has not been practical because the interface has been at the computer-peripheral control boundry. Also, because the designers want to optimize each system there is a tendency to design each control to tune a system. A common interface would benefit software design, as well as giving production flexibility, and minimizing system designs. I think that due to increased emphasis on remote terminals there is a trend (good one) to be able to remote any device, and as such the specialized interface will hopefully vanish from our universe. For example, IBM will shortly announce a card reader, card punch, line printer combination that connects to a standard Data phone.

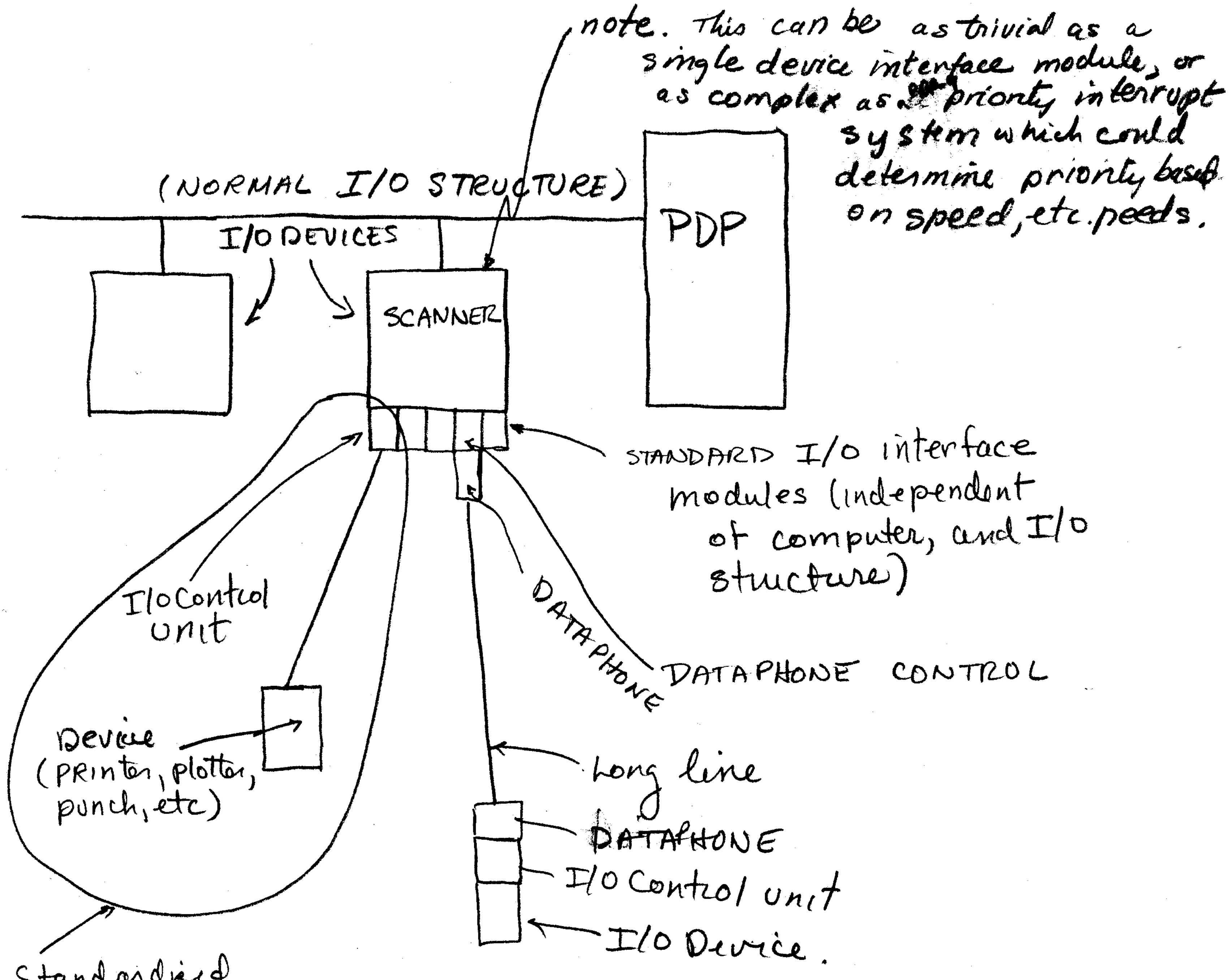
Therefore, I hope that since PDP-9,10, and 8I are in their pre-peripheral design phases, such an approach be studied as a means of having common I/O controllers across all computers and lines including new ones. Obviously, not all equipment fits the mold.

The equipment which looks most likely:

A-D-A
Paper Tape Readers & Punches
Card Readers and Punches
Printers
Plotters

Teletypes, Typewriters
Dataphones, and Phone Transmission stuff
slow displays
audio units
computer-to-computer buffers
relays, etc. (digital I/O)
Discs, Drums, mag. tape, DECtape, and Displays are undoubtedly too fast.

One possibility for such a system would be: (See attached sketch.)


note. This can be as trivial as a single device interface module, or as complex as a priority interrupt system which could determine priority based on speed, etc. needs.
(NORMAL I/O STRUCTURE)
PDP
I/O DEVICES
SCANNER
STANDARD I/O interface modules (independent of computer, and I/O structure)
DATAPHONE
DATAPHONE CONTROL
I/O Control unit
Device (PRinter, plotter, punch, etc)
Long line
DATAPHONE
I/O Control unit
I/O Device.
Standardized option

Methods of Constructing Standard I/O Devices

CGB 1/28/67

digital INTEROFFICE MEMORANDUM

DATE: 26 February 1968

SUBJECT: Visit to DEC 15 February 1968

TO: J. A. Jones
Stan Olsen
Nick Mazzarese
Ed deCastro
Mike Ford
Win Hindle
Ken Olsen

FROM: Gordon Bell

After spending a day talking about computers, I'm reacting by trying to write down my version of what transpired. I hope others will do the same, as I felt a tremendous need to try to put things into a framework. Also, since Mike Ford asked me what machines to build, I wanted to write an answer.

To begin with, I'm sorry to hear that the X has been killed, since it potentially could have formed the basis for a compatible series. However, since it implied a large number of compromises in each group, it probably is not possible

Ultimately, it would have removed the 9 and 10 as product lines and no one likes to be part of a vanishing product line.

In Summary

My favorite suggestions (although I'd like some other points looked at) now are:

1. Form a product-planning group.

2. Patent the Homogeneous Read-Only plus Read-Write Memory (described below).

3. Don't build a 24-bit computer, fast. If you have to, you might look at the PDP-X, which has both 16 and 32-bit instructions for an average of 24-bits only it's better than most 24-bit computers.

4. Build an 8/I around larger boards and lower cost and better cabinet fabrication (see Data Machines/and Mike Ford's suggestions). Incorporating options for:

   a. Lower basic cost.
   b. Use of Read-Only Memory.
   c. Not moving the computer on slides, drawers, or books.

5. Build 10/I Develop 10/I Memory for use in 9/I, 10/I.

6. A nice, modular, vast 9:

   a. Very lost cost.
   b. Modular memory system a la X in upper models.
   c. Multi-processor at high end.
   d. Use Read-Only Memory (either internal or main) to increase speed of arithmetic, so that it competes with 24-bit computers.
   e. Add XR's and some scratch pad, a la Ed deCastro's large 16 X 32 bits (18 X 32) or (18 X 16).
   f. Make a processor for interpreting PDP-10 instructions and handling PDP-10 I/O devices using Read-Only Memory internally.

7. Build the 9-bit controller. As a stand-alone computer, and a controller to 9, and 10 devices.

8. Try to build special, total systems, based on software packages for existing machines (e.g. TS8; TS9; Administrative Terminal System-like thing (IBM's Multi-terminal editor);)

9. Do something to consolidate market planning across product lines.

10. Data Communications can still be yours, don't drop it!

Other Comments

Although my following arguments need to be based on cost/performance curves, I think our sales result from other factors, too: inertia, (IBM effect); lowest cost; and cost/performance.

The X group came up with some nice analytical relationships (e.g., instruction set utilization, performance of machines, checkout costs, etc.), especially when it was needed to back a decision. I would like to endorse their analysis and would hope the several machines that are being started could all be done on such underlying thoroughness. I'm suggesting a number of machines, and I'd really like to see cost/performance) memory size curves for each of them. I'm enclosing examples I did on the 360.

I'd like to put the following into a better framework then the linear list following, but think that's up to #1, below. The items are:

1. Set up a market-study group to try to consider the company as a whole, and have it connect with each product-marketing group. I would prefer to call and use the existing marketing groups for sales support, and sales, and information collection. Such a group would be more along the lines of product-planning group, doing market/cost analysis with a combination of design, production, and market inputs and would help guide product planning.

2. Try to increase the parts which are produced in common for all computers (for production, sales promotion, customer learning, and training reasons through some formal organizational body (maybe product planning)). (For example: parts of memories, peripherals, and peripheral controllers.) The structure of the 8, and 9 make it virtually idiotic not to have common controllers. The advent of the larger logic cards, then LSI, really necessitates this. Specifically Ed deCastro wound up with a 16 X 32 array, fast memory that could be used in the 9I+ and 10I. These parts include software (see 5 below). About a year ago (memo Feb 6, 1967), I suggested such a scheme for common peripherals, the arguments are still valid.

3. Start patent proceedings on the Homogeneous Read-Only, Read-Write Memory scheme, described below, which was developed on the 15 February meeting. It seems to be an effective way to get a nice local improvement in speed, in the case of simple processors like PDP-8, 9. I've looked at the PDP-8/I logic, and if you can wait long enough $\geqslant 1\frac{1}{2}$ years, I will make it go at .3 μsec/ read-only cycle, with only 15% more integrated circuits.

4. It is very difficult to measure the cost-benefit of another product in the line. I'm against any machine which is only incremental and does not try to better consolidate all DEC computers because I believe the cost of development and maintenance (especially software) is too high. For the same amount of development $'s, I believe system applications software has better payoff, i.e., a computer is converted to a particular device (a la typesetting, etc.).

5. Along the lines of 4, DEC could start collecting FORTRAN programs from places like SHARE, GUIDE, etc., which can be run on both the 9, and 10, and maybe 8. In fact, I think the generalized applications packages (e.g., a MATH-pak, or a STAT-pak, etc.) are the only reasons one would buy an IBM small 360 or 1130/1800 over DEC. This can be overcome by getting these packages into the DECUS library. A policy to use FORTRAN to code these packages seems like a good, long-range policy. Most such packages are available, free, now. (For example, all CALCOMP plotter programs exist in FORTRAN).

6. Investigate several design alternatives thoroughly. (The only implementation which traded-off cost for performance to come from DEC has been the PDP-8/S.) I'd like for these to be investigated.

8/I-1 (lower cost using larger boards, and different bus structure to lower cost).

8/I-2 (lower cost-lower performance - possibly a serial version to run at 2 μsec/word or so)

8/I-3 (a rope memory control which allows some small set of core or flip-flop memory to be added along the lines of the homogeneous rope-core below).

8/I-4 (8/I-1+ 8/I-B).

9/I-1 (lower cost 9 - may or may not use rope control like the 9).

9/I-2 (fancier 9 structure with local MB and MA in a memory). The memory options would be based on some X designs and include:

(1) Memory box with connection or port to one processor with 4K, 8K or 16K.

(2) Memory box with connection or ports to two processors or a processor and controller with 4K, 8K, or 16K.

(3) Box to allow multiple (4-8) processors or controllers to connect to a memory port. The processor might use rope control.

9/I-3 (processor with a homogeneous read-only core structure in which Read-Only structure might include programmed floating point or FORTRAN operating system interpreter to speed up numerical calculations. This structure could do numerical work faster than a single 24-bit machine). The main memory structure would be along lines of 9/I-2 in which some modules would be rope.

9/I-4 (A fancier processor with rope control, along the lines of the 9, but a larger rope so that floating point and other common ops could be sped up.) Such a structure would also allow control functions, such as DECtape, Magtape, 680-like teletypes, high speed line concentrator, to be included.

This feature would be sold to customers for their use.

9/I-5 ≡ Mini-processor 10-2

A processor which would connect to 9/I-2 Memories, and PDP-10 I/O bus, and interpret only PDP-10 code (using rope memory). 16K X 18-bits would be minimum memory size. Use 10/10 + software.

9/I-6 ≡ Mini-processor 10-2

A processor which would connect to PDP-10 Memories, and PDP-10 I/O bus and be 18-bits wide, and interpret PDP-10 code. 16K X 18-bits would be minimum. Use 10/10 + software.

9/I-7 A multi-processor 9 (where multi $\geq$ 2), this should not only out perform a 24-bit computer, but should be cheaper, and more reliable.

9/I Increments

From a future product planning point of view, the 9 can be spruced up a bit, e. g.

(1) Three-core index registers.

(2) Replacement of first 16-core register to speed up operations using index registers temporary, and auto-index registers.

(3) Investigate if MIT's (Lee), and Harvard's PDP-9 time-sharing system is marketable.

(4) Incorporate Edinburg's PDP-7 MACRO Assembler in software.

(5) See why the PDP-9 FORTRAN is so bulky, and slow.

10/I	Integration of processor, compatible with 10. Integrate other components, attempt to use 9/I sub-components.
X-1	Smaller scale version of X.
24	Another computer.
9-bit computer	A smaller than PDP-8 computer which would be part of a series of weakly, compatible machines of our 9, 18, 36-bit series. This would stand alone as a minimum computer.

Also it would be specifically designed to serve as the <u>controller</u> for elaborate devices, or a group of devices which could be used on the 9 and 10, (also, 8 if desirable). It would be a front-end controller for communication lines for the 9 and 10 (scanning and buffering). This could be an important product, if it can be designed.

Note: This computer is along the lines of one we'd like built for here. I sent Mike Ford a copy of an 8-bit computer, along these lines which we thought could be built for $3K at Carnegie. I would like to remind people that the tasks which are done in 8-bit chunks, can be done nicely in a 9-bit computer. In fact, it may be a 'silly 1-bit longer'.

8-bit computer Although this is also minimum, it doesn't look very good as a controller to an 18 or 36-bit machine. I've never felt that 8 is an especially good base, and base (2^9)'s has 100% more states than an 8-bit base.

7. Do something about Data Communications Market (product) planning, before it's too late! Although it still isn't too late, waiting another year before starting to plan may be. (See memos of about 1½ years back). This is just right for DEC as a market (especially with the new 9×10^6 bit disk). This includes both telegraph message switching, and display (text-keyboard) at 2400-bits/sec concentration. Respond positively, creatively, and correctly to ARPA's RFQ for their network switching computers. This job may take a PDP-9, and the present DEC organizational structure precludes thinking of the problem this way.

Right now IBM has just announced an option to connect to the 360/25 to give 64 telegraph lines in and two high speed lines out in a concentrator and the price isn't awfully unreasonable, especially since they rent.

The proposed 24-bit machine

I think this machine isn't especially good as it's a compromise between a medium computer (16/18-bits) and a reasonably large one (32/36-bits). Although a 24-bit machine will out perform an 18-bit machine (for the same level of technology - i.e., memory speed) due to added index register and extra instructions. I don't give one (e.g., 910-920-like) more than a factor of 2 over a PDP-9 for the same memory speed, although one can build a 24-bit computer that performs like a large computer (e.g. CDC 3200).

Mostly, I don't like the idea of another product which has no chance of bringing the other product's production, programming, or sales training any closer together. (I can show you a real mess at IBM prior to the 360 in which slightly better, non-computible products kept getting stuck in cost/performance, cost, or performance holes.)

I agree that thereis a significant hole between the 9 and 10. This hole can be filled with existing product parts rather than introduce another incompatible series. In both the 9 and 10, there exists the possibilities for a nice filler. There is a discussion of the 360 as an example of filling.

The issue of whether a multi-processor 9 is better than a mini-processor 10 (9/I-5 or 9/I-6) should be based on cost/performance comparing say space/time for FORTRAN in the two machines, peripheral costs, instruction set power, and the fact that 10 software is already pretty far advanced. (Such a machine would use a memory of 16K words). I dont believe that the PDP-10 group is capable of making such a design or evaluating thefeasability.

Again, I think $'s should be spent on support software instead of basic software like maintenance routines, compilers, etc. A three or four year extensive effort to get DEC to the level of the SDS 900 series. Also, I believe that if any present 24-bit manufacturers want to, they could wipe you out! On the other hand, with a dual processor 18-bit machine, you could make things rougher on them.

I looked at some sample SDS 900 series programs, and though admittedly not typical, in 100 instructions I counted, an 8-bit address was sufficient 75% of the time. This compares favorably with the statistics in the instructions measured by the X group. I don't believe the small address hack is a hack, but rather an efficient use of bits.

360 Lessons

Enclosed are some notes on a talk given by Fred Brooks, one of the IBM 360 designers at a talk at IBM Poughkeepsie. I have also enclosed my IBM 360 cost/performance graphs, as I believe this kind of analysis is necessary to find a filler between the 9 and 10. The issue of ROS and multi-processors can be seen from the 360. For example, the utilization of memory

$$= \frac{\text{number of memory cycles used}}{\text{number of memory cycles available}}$$

Model	Memory Utilization
30	.2
40	.4
44	.55
50	.5
65	.37 - .18
75	.54 - .27

This is low compared to the PDP-8,9 machines, but on the other hand, the complex 360 instructions do move. Their 1130 and 1800 are like .75. ROS causes part of the problem, but the complex instructions do too. The 10 would probably be pretty low too, due to floating point, and multiple memories (in fact, a 32K system would put it below .5).

I proposed a smaller set of 360 processor primitives which would give better cost/performance in the 360, and I think these also apply to the 9+, 10-, 24-bit issue. These are given below.

An Alternative Series of Processors to Cover the Range of Computing Power.

Graph 4 indicates that an alternative approach based on multiple Pc's is feasible. Suppose the following Pc's are chosen as primitives:

Model	Power
C(20)	1
C(30')	4
C(44)	30

Then by combining primitives, the performance values of the present computer line can be obtained, as shown below:

Model	Power	Pc Cost
C(20)	1	.00049
2-C(20)	2	.00098
C(30)	2	.00125
C(30')	4	.00125
C(40)	6	.00295
2-C(30')	8	.00250
C(50)	15	.011
4-C(30')	16	.005
C(44)	30	.004
2-C(44)	60	.008
C(65)	60	.022
C(75)	80	.0365
3-C(44)	90	.012
16-C(44)	480	.064
C(91)	500	.09

Note that in every case, the multiple Pc approach performs significantly better than the uni-processor, at a lower cost. (The multiple Pc interconnection cost with Mp, and the problem of breaking the task apart has been ignored.)

INTEROFFICE MEMORANDUM

PM 11-3

DATE: 1 April 1968

SUBJECT: Reexamination of 24-bits

TO: Nick Mazzarese
Win Hindle
Ken Olsen
Ed DeCastro
Gordon Bell✓
Larry Portner

FROM: John Cohen

I. Background

Since the demise of the PDP-X, a number of possibilities for new products have been discussed. One of these is a medium-scale 24-bit machine. Initial reaction was very negative - in fact, everyone I spoke to was against it. The feeling was that the market was tending away from existing 24-bit machines and no one was sure who would buy such a machine. However, further consideration, especially a technical comment by Ed DeCastro, make me want to bring the issue up again.

Ed points out that memory speeds are increasing faster than hardware speeds and that this trend is expected to continue over the next few years. The implication is that it will become more and more difficult to design the hardware to keep up with the memory. The simpler the addressing scheme and instruction set, the easier it is to achieve hardware speeds. An instruction length of 16 or fewer bits naturally leads to a complicated addressing scheme - along with the associated hardware complexity. A 24-bit machine can be simply and directly addressed - thus warranting its further consideration.

The next section reviews a number of technical and marketing considerations which seem to lead to 24-bits. Finally, section III contains a specific proposal to build a 24-bit product line.

II. Technical and Marketing Considerations

A. Objectives

After talking with many people, I tend to feel that there are three valid reasons for building a new computer line. In order of estimated importance, these are:

--bridge the "cost gap".
A PDP-10 typically sells for more than 175K, while a PDP-9 most often sells for 125K or less. There is a void between these two machines which should be filled. There is some question as to whether a big 9 or a small 10 could do this job effectively.

--Get better performance/cost.
New concepts of machine organization make it possible to produce a computer with better performance/cost than the PDP-9. Although performance/cost is not necessarily the thing that sells computers, an improvement would not hurt.

--Make programming easier and cheaper.
Without any question, one of the problems with our present small and medium scale computers is that they are difficult to program. Since we are likely to provide more applications software to our customers in future years, this can be a real difficulty. On a long term basis, we would save money with a more "programable" computer.

B. Hardware/Memory Speed

According to Ed DeCastro, the current trend is for hardware speed to increase more slowly than memory speed. From a cost/performance point of view, a computer is optimally designed when its memory speed is nearly balanced by its logic speed. This is seen to be true from the following reasoning - suppose a computer memory is much faster than the hardware. Then the memory could be replaced by a slower (and cheaper) memory without substantially changing the performance of the machine. A similar argument holds if the hardware is much faster than the memory. Henry Burkhardt points out that the Sigma 2 is mismatched in the sense that their hardware is considerably slower than the memory. They could replace the fast memory by a slower one without hurting the through-put capabilities.

The implication of the hardware/memory trend is that it will become more difficult, over the next few years, to design hardware to keep up with memory. The more complex an instruction set, the more this effect will be amplified. A 16-bit computer must naturally have complex effective address calculation. For example, on the PDP-X, a check first must be made to determine if an instruction is basic or extended. If it is basic, it must be further determined whether it is short or long form addressing. Then it must be determined if the addressing is immediate or memory reference. The point is that this type of scheme will become more expensive to implement as memory speeds increase.

This leads naturally to the consideration of 24-bits. First of all, I think any new machine we build should have a word length which is a multiple of 8. This is becoming fairly standard in the industry and people I talk to uniformally agree that it will help us sell systems which interface with other equipment. A 24-bit instruction allows direct memory referencing without paging (PDP-8) or relative addressing (PDP-X). Without question, this is a major hardware and software simplification. It makes a machine more easy to understand for all involved - engineers, programmers, salesmen and customers.

If the hardware speed/memory speed argument is correct, the manufacturers of 16-bit computers may be switching to 24-bits in the next 2-4 years. If we are not committed to 16-bits, there is no reason why we should not be the first to "switch" to 24.

C. The Waste of Memory Argument

Computers use one of two addressing techniques which could be called direct and non-direct. Many medium scale and large systems use the direct approach. Each instruction contains enough address bits to directly reference all of core memory. For example, the PDP-10 instructions have 18 bits to address a maximum of 256K. The advantages of such a scheme are that it is relatively easy to implement in hardware and that it is convenient for programming. However, many people say that it has the disadvantage of wasting instruction bits. The claim is that most memory references refer to locations which are relatively near the instruction. Thus, the claim is made that bits can be saved if addresses are given relative to the issuing instruction, or "local" to a memory page.

Two commonly used non-direct methods are the page scheme and the relative address scheme. For example, the PDP-8 memory is divided into 256 word pages. In each memory reference, the program must specify whether the effective address is in the current page or in a special, fixed page. If the address is in neither of these, then the reference must be made indirectly, using another word.

In the PDP-X, a "short form" was used if the effective address was located within 128 words of the instruction. If not, then another full 16-bit word was necessary to specify the address.

The proponents of the non-direct addressing schemes claim that 30 to 40 per cent of the bits in direct reference computers are wasted. The opposing view holds that paging and relative schemes make the computer inherently more complicated and cause programming to be more difficult and costly.

It is my personal feeling that the waste of memory argument is a "red herring". To be sure, certain programs can be coded in, say, 30% fewer bits in a non-direct computer. However, if the program is half data and half instructions, the savings is only 15%. In addition, there is no need to make relatively small programs even smaller if part of the computer memory isn't used. Therefore I believe that the 30% figure has to be discounted to a 10% savings. My feeling is that the advantages of this savings are more than out-weighed by the increased hardware complexity and software development difficulties.

D. How Many Registers?

If we do build a 24-bit machine, I feel that it should have one accumulator and one index register. This would be cheaper to implement and would make the programming easier. Minor disadvantages would be slightly larger programs and the possibility of unfavorable comparisons on competitive checklists. An example of a computer with multiple accumulators and index registers is the PDP-10. I agree that the multiple registers give the capability of generating smaller and more efficient programs. However, I feel that the extra costs involved out-weight the advantages. Hardware development and maintenance is more expensive.

Possibly the best argument against multiple registers is in software development. In my experience, I have found that most programmers work better on machines which offer them only one method to perform a given function. If there is more than one method, they will spend much time and effort trying to optimize. The real objective in programming usually is to produce a program that works, rather than a program which works and is the fastest program possible and is the smallest program possible. This objective can be reached most easily on a simple computer with only one index register and one accumulator.

Unfortunately I don't have any solid figures to support the contentions made here, but I strongly suggest that a single index register and accumulator is to our advantage. It "forces" easy programming and makes the hardware easier to build and maintain. In addition, the nature of such a machine causes software systems to be more simply organized and thus easier to maintain.

E. The Use of Read Only Memory

There are basically two alternatives for internal computer structure - conventional orgainization and read-only memory control. The latter has the advantages of being flexible and cheaper for complex instruction sets. It has the disadvantage of being inherently slower than hardware. Computers built without ROM control tend to be faster, but inflexible in instruction set. However, if the instruction set is simple, conventional organization is cheaper than ROM control.

Since the discussion here is about a simple 24-bit machine, I think we are lead to the conclusion that read-only memory control should not be used.

III. A Recommendation

About a month ago, I suggested that DEC built both a 16 and a 32 bit computer. Because of the considerations above I'd recommend shelving the 16 and 32 ideas and focussing on 24. I think it promises the most immediate pay-off and will interfere least with existing product lines. Specifically I recommend:

A. 24-Bit Processor

We should build a 24-bit computer with direct memory addressing (no paging or relative addressing). There should be one accumulator and one index register (plus an additional register for multiply and divide operations).

The instruction set should be much simplier than the PDP-10 or PDP-X. An example of what I have in mind appears in the appendix.

The system should not be organized around time sharing. Multiply, Divide and Floating Point should be optional hardware. The computer should not be micro-programable via read-only memory.

B. 8-bit Peripheral Controller

We should also build a simple 8-bit micro-programable processor to be used primarily as a peripheral controller. It should be of intermediate internal complexity-more complex than the Interdata II, less complex than the PDP-X - about at the IBM 360/30 level. The machine would have a secondary use as an emulator for the 24-bit machine. It would probably sell at 1/3 the cost and run at 1/10 the speed. It could presumably be built before the 24-bit processor and could be used for software development for the larger machine.

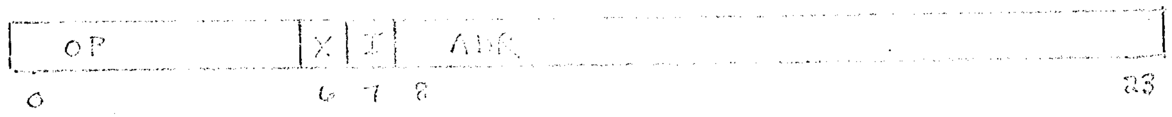
C. Interfacing Standards

Computer Technology Limited has, in theory, a product line with exceedingly rigid interface standardization. I have the impression that we have never put in enough effort in this area and have had difficulties when trying to configure non-standard systems. Our engineers should look closely at the CTL Modular One and at functionally large macro-modules. Neither of these may be the answer to our interface standardization problems, but they should provide us with a starting point.

APPENDIX

This appendix is a description of a simple 24-bit instruction set. Input/output and interrupt instructions are not considered. From the point of view of software development, I would be extremely happy to work with such a machine.

The instruction format is:



where OP is a 6-bit op code (64 possibilities)
X is the index register specification
I is the indirect address specification
ADR is a 16-bit address (up to 64K 24-bit words)

Location 0 refers to the accumulator. Location one refers to the multiply/divide register, when the option is present. Location two is the subroutine linkage register and location three the index register (similar to the PDP-X). An undefined operation code causes the program counter plus one to be stored in location four and a branch to location five.

The instructions are:

LDA	load accumulator
STA	store accumulator
ADD	add
SUB	subtract
INC	increment
NEG	negate
COM	complement
TST	test
BRU	branch unconditionally
BAL	branck and link
BCT	branch on carry true
BCF	branch on carry false
BZT	branch on zero true
BZF	branch on zero false
BNT	branch on negative true
BNF	branch on negative false
CLR	clear
AND	and
ORA	or
XOR	exclusive or
SHF	shift
BLM	block move
CML	compare logical
CMA	compare arithmetic

LDC	load character
STC	store character
ICP	increment character pointer
MUL	multiply
DIV	divide
LML	logical multiply
LDV	logical divide

LDA and STA load and store the accumulator. ADD and SUB add and subtract the effective word to the accumulator. INC adds one to the effective word. NEG and COM negate and complement the effective word, respectively.

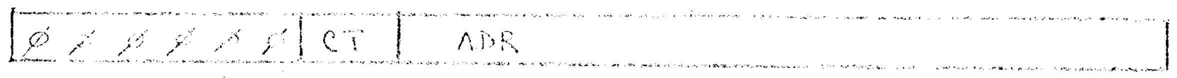
There are three condition code flip-flops as in the PDP-X. One is carry or overflow, the second is zero result and the third negative result. TST sets these condition codes (except carry) based on the status of the effective word. BRU causes an unconditional branch to the effective address. BAL causes the program counter plus one to be stored in the subroutine linkage register and then a branch to the effective address.

BCT, BCF, BZT, BZF, BNT and BNF cause conditional branches on the condition code status.

CLR sets the effective word to zero. AND, ORA and XOR perform logical operations on the effective word and the accumulator, leaving the result in the accumulator. SHF shifts the accumulator as specified by the effective address. Zeros are shifted in from the right or left. No provision is made for shifting in one's or for rotating the accumulator.

BLM is a block move instruction, which can be an option. The effective address points to a three word block containing a destination address, a source address and the count. The number of words specified by the count is moved from the source address block to the destination address block. CML and CMA compare the accumulator to the effective word and set the condition codes appropriately. CML does a logical compare while CMA does an arithmetic compare.

Three instructions for character manipulation are included, possibly as an option. They operate on a character pointer with the following format:



where CT is zero, one or two indicating, the first, second, and third characters in the word.
ADR is the address of the word containing the referenced character.

For example, if CT is one and address is 1,000, the pointer refers to the second or middle character in memory location 1,000. The effective word of an LDC instruction must be a character pointer. The appropriate character is moved into the accumulator bits 16-23. Bit 0 through 15 of the accumulator are set to zero. STC takes bits 16-23 of the accumulator and stores them as specified by the character pointer in the effective word. ICT increments the character pointer. If CT is zero or one, one is added to CT. If CT is greater than one, CT is set to zero and one is added to ADR.

The optional multiply/divide hardware has four instructions - arithmetic multiply and divide and logical multiply and divide.

ljh

digital INTEROFFICE MEMORANDUM

DATE: May 24, 1968

SUBJECT: PDP-10 COMPARABLE WITH SIGMA 5

TO: Gordon Bell FROM: Ken Olsen

SDS seems to have outsmarted us when they came out with their Sigma 5. They now have the lowest priced computer of this size, and are taking many orders away from our PDP-10.

What can we cut out of the PDP-10 to make a useful, low-priced computer?

Ken

February 26
...I'm glad you ask that question. On ~~[illegible]~~ 1968 I wrote a number of possible ~~solutions~~ which came out of my ~~visit~~ on 15 February, and on 21 of February 1968 Seligman wrote a memo on PDP-9 which said roughly the same thing.

Namely, we say: The Sigma 5 came out because of the PDP-10 and is presumably selling better because of lower cost, or 32 bits or people not able to sell the 10 IO structure or the 10 IO structure may not be as good. The 10 people probably aren't interested in any compatibility with the 9, or are interested in a scaled down version of the 10. I expect future plans for the 10 to be more extensive in its ability to time share, with better (more exotic) program mapping to cut down on the monitor overhead.///... All of these take the machine up the price scale, but make it more useful or better for timesharing. I don't think the 10 designers are particularly interested in the problem...or ~~maybe~~ should they be? One thing that Seleigman and I agree on in this regard, is that a very useful, PDP-10 compatible ~~machine~~ Processor could be made which is probably based on a Read Only Store to interpret the PDP-10 order code. It would be slow by 10 standards, but on the other hand could ~~let~~/ maybe exist for as little as 20-30 k$....and be only 1/4 as fsst. Just integrating might help on the cost question.

In regard to the stuff taht John Cohen is doing we are obtaining very interesting results by coding various problems, and intend to find the best ~~for~~/ machines for various problem classes. ~~/I/ord~~I am asking John to add coding for the PDP-10 to see how it s fares, .. or am trying to find just what it is the large machine is good at. Several weeks ago, I started examing the possibility of connecting many of these computers together, to see if it helps the power problem, with the hopes of wiping out large machines for most problems, and it appears as if this may just be possible. Therefore, in the 8 bit machine, that sells for ~~$1~~/$2k I want to put some escape mechanism so that it can exist in a high performance version with 32 or more bits, and floating point. As of now, the problem seems easy, but may not look so good as we get closer to the solution. So ~~are~~ far the 8 bit (cough) $2k machine is better than and x, a 9, the 8,/all 16 bitters, and maybe the 10, now maybe we ought to look at a problem it doesnt do as well

have Seligman write up of this summer

F-85
5/66

**PROGRAMMED DATA
PROCESSOR - 8**

PDP-8
USERS HANDBOOK

DIGITAL EQUIPMENT CORPORATION • MAYNARD, MASSACHUSETTS

PROGRAMMED DATA PROCESSOR-8 USERS HANDBOOK

DIGITAL EQUIPMENT CORPORATION • MAYNARD, MASSACHUSETTS

PREFACE

This handbook concerns programming, operating, and interfacing the Programmed Data Processor-8; a high-speed stored-program digital computer manufactured by the Digital Equipment Corporation. Section A describes the functional operation, instructions, and basic machine-language programming of the PDP-8 processor, core memory, processor options, and core memory options. Section B is devoted to the functional operation, instructions, and basic programming of standard and optional input/output equipment of a PDP-8 system. Section C presents information on operating the basic system and its options. Section D serves as an interface and installation manual, and contains information on planning and implementing the design and installation of any electrical interface required to connect a special device into a PDP-8 system.

Appendixes at the end of this handbook provide detailed information which may be helpful in specific programming assignments. Although program examples are given in this document, no attempt has been made to teach programming techniques. The meaning and use of special characters employed in the programming examples are explained in the description of the Program Assembly Language, available from the Digital Program Library.

CONTENTS

CONTENTS (continued)

CONTENTS (continued)

CONTENTS (continued)

CONTENTS (continued)

ILLUSTRATIONS

ILLUSTRATIONS (continued)

TABLES

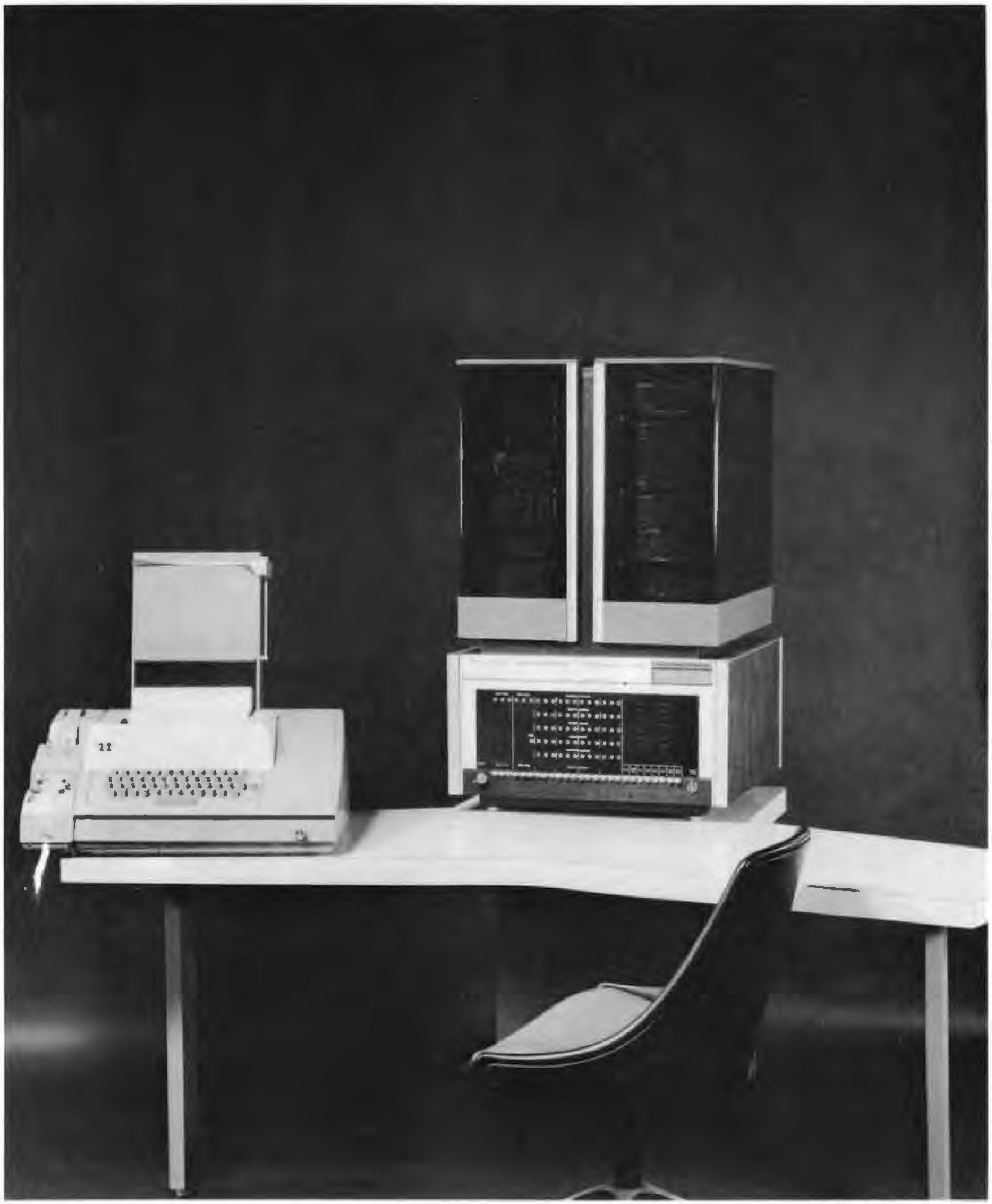


Figure 1   Typical PDP-8 in Table Top Configuration

SYSTEM INTRODUCTION

The Digital Equipment Corporation Programmed Data Processor-8 (PDP-8) is designed for use as a small-scale general-purpose computer, an independent information handling facility in a larger computer system, or as the control element in a complex processing system. The PDP-8 is a one-address, fixed word length, parallel computer using 12 bit, two's complement arithmetic. Cycle time of the 4096-word random-address magnetic-core memory is 1.5 microseconds. Standard features of the system include indirect addressing and facilities for instruction skipping and program interruption as functions of input-output device conditions.

The 1.5-microsecond cycle time of the machine provides a computation rate of 333,333 additions per second. Addition is performed in 3.0 microseconds (with one number in the accumulator) and subtraction is performed in 6.0 microseconds (with the subtrahend in the accumulator). Multiplication is performed in approximately 315 microseconds by a subroutine that operates on two signed 12-bit numbers to produce a 24-bit product, leaving the 12 most significant bits in the accumulator. Division of two signed 12-bit numbers is performed in approximately 444 microseconds by a subroutine that produces a 12-bit quotient in the accumulator and a 12-bit remainder in core memory. Similar multiplication and division operations are performed by means of the optional extended arithmetic element in approximately 15 and 30 microseconds, respectively.

Flexible, high-capacity, input-output capabilities of the computer allow it to operate a variety of peripheral equipment. In addition to standard Teletype and perforated tape equipment, the system is capable of operating in conjunction with a number of optional devices such as high-speed perforated tape readers and punches, card equipment, a line printer, analog-to-digital converters, cathode-ray-tube displays, magnetic drum systems, and magnetic-tape equipment. Equipment of special design is easily adapted for connection into the PDP-8 system. The computer is not modified with the addition of peripheral devices.

PDP-8 is completely self-contained, requiring no special power sources or environmental conditions. A single source of 115-volt, 60-cycle, single-phase power is required to operate the machine. Internal power supplies produce all of the operating voltages required. FLIP CHIP modules utilizing hybrid silicon circuits and built-in provisions for marginal checking insure reliable operation in ambient temperatures between 32 and 130 degrees Fahrenheit.

Computer Organization

The PDP-8 system is organized into a processor, core memory, and input/output equipment and facilities. All arithmetic, logic, and system control operations of the standard PDP-8 are performed by the processor. Permanent (longer than one instruction time) local information storage and retrieval operations are performed by the core memory. The memory is continuously cycling, automatically performing a read and write operation during each computer cycle. Input and output address and data buffering for the core memory is performed by registers of the processor, and operation of the memory is under control of timing signals produced by the processor. Due to the close relationship of operations performed by the processor and the core memory, these two elements are described together in Section A of this handbook.

Interface circuits for the processor allow bussed connections to a variety of peripheral equipment. Each input/output device is responsible for detecting its own select code and for providing any necessary input or output gating. Individually programmed data transfers between the processor and peripheral equipment take place through the processor accumulator. Data transfers can be initiated by peripheral equipment, rather than by the program, by means of the data break facilities. Standard features of the PDP-8 also allow peripheral equipment to perform certain control functions such as instruction skipping, and a transfer of program control initiated by a program interrupt.

Standard peripheral equipment provided with each PDP-8 system consists of a Teletype Model 33 Automatic Send Receive set and a Teletype control. The Teletype unit is a standard machine operating from serial 11-unit-code characters at a rate of ten characters per second. The Teletype provides a means of supplying data to the computer from perforated tape or by means of a keyboard, and supplies data as an output from the computer in the form of perforated tape or typed copy. The Teletype control serves as a serial-to-parallel converter for Teletype inputs to the computer and serves as a parallel-to-serial converter for computer output signals to the Teletype unit.

The Teletype and all optional input/output equipment is discussed in Section B of this handbook.

Symbols

The following special symbols are used throughout this handbook to explain the function of equipment and instructions:

Symbol	Explanation
$A => B$	The content of register A is transferred into register B
$0 => A$	Register A is cleared to contain all binary zeros
Aj	Any given bit in A
A5	The content of bit 5 of register A
A5(1)	Bit 5 of register A contains a 1
A6 − 11	The content of bits 6 through 11 of register A
$A6 - 11 => B0 - 5$	The content of bits 6 through 11 of register A is transferred into bits 0 through 5 of register B
Y	The content of any core memory location
V	Inclusive OR
$\not\vee$	Exclusive OR
$\wedge$	AND
$\overline{A}$	Ones complement of the content of A

SECTION A

MEMORY AND PROCESSOR

CHAPTER 1

MEMORY AND PROCESSOR FUNCTIONAL DESCRIPTION

Major Registers

To store, retrieve, control, and modify information and to perform the required logical, arithmetic, and data processing operations, the core memory and the processor employ the logic components shown in Figure 2 and described in the following paragraphs.

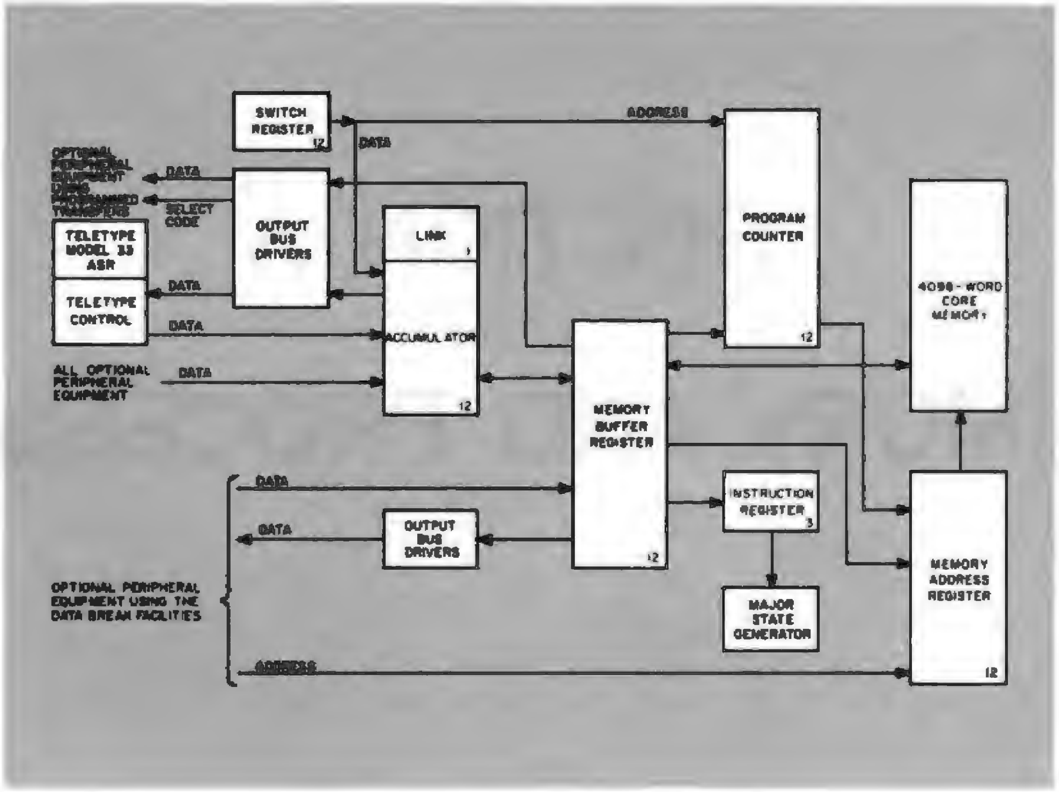


Figure 2 PDP-8 Major Register Block Diagram

ACCUMULATOR (AC)

Arithmetic and logic operations are performed in this 12-bit register. Under program control the AC can be cleared or complemented, its content can be rotated right or left with the link. The content of the memory buffer register can be added to the content of the AC and the result left in the AC. The content of both of these registers may be combined by the logical operation AND, the result remaining in the AC. The memory buffer register and the AC also have gates which allow them to be used together as the shift register and buffer register of a successive approximation analog-to-digital converter. The inclusive OR may be performed between the AC and the switch register on the operator console and the result left in the AC.

The accumulator also serves as an input-output register. All programmed information transfers between core memory and an external device pass through the accumulator.

LINK (L)

This one-bit register is used to extend the arithmetic facilities of the accumulator. It is used as the carry register for two's complement arithmetic. Overflow into the L from the AC can be checked by the program to greatly simplify and speed up single and multiple precision arithmetic routines. Under program control the link can be cleared and complemented, and it can be rotated as part of the accumulator.

PROGRAM COUNTER (PC)

The program sequence, that is the order in which instructions are performed, is determined by the PC. This 12-bit register contains the address of the core memory location from which the next instruction will be taken. Information enters the PC from the core memory, via the memory buffer register, and from the switch register on the operator console. Information in the PC is transferred into the memory address register to determine the core memory address from which each instruction is taken. Incrementation of the content of the PC establishes the successive core memory locations of the program and provides skipping of an instruction based upon a programmed test of information or conditions.

MEMORY ADDRESS REGISTER (MA)

The address in core memory which is currently selected for reading or writing is contained in this 12-bit register. Therefore, all 4096 words of core memory can be addressed directly by this register. Data can be set into it from the memory buffer register, from the program counter, or from an I/O device using the data break facilities.

SWITCH REGISTER (SR)

Information can be manually set into the switch register for transfer into the PC as an address by means of the LOAD ADDRESS key, or into the AC as data to be stored in core memory by means of the DEPOSIT key.

CORE MEMORY

The core memory provides storage for instructions to be performed and information to be processed or distributed. This random address magnetic core memory holds 4096 12-bit words in the standard PDP-8. Optional equipment extends the storage capacity in fields of 4096 words or expands the word length to 13 bits to provide parity checking. Memory location 0_8 is used to store the content of the PC following a program interrupt, and location 1_8 is used to store the first instruction to be executed following a program interrupt. (When a program interrupt occurs, the content of the PC is stored in location 0_8, and program control is transferred to location 1 automatically.) Locations 10_8 through 17_8 are used for auto-indexing. All other locations can be used to store instructions or data.

Core memory contains numerous circuits such as read-write switches, address decoders, inhibit drivers, and sense amplifiers. These circuits perform the electrical conversions necessary to transfer information into or out of the core array and perform no arithmetic or logic operations upon the data. Since their operation is not discernible by the programmer or operator of the PDP-8, these circuits are not described here in detail.

MEMORY BUFFER REGISTER (MB)

All information transfers between the processor registers and the core memory are temporarily held in the MB. Information can be transferred into the MB from the accumulator or memory address register. The MB can be cleared, incremented by one or two, or shifted right. Information can be set into the MB from an external device during a data break or form core memory, via the sense amplifiers. Information is read from a memory location in 0.8 microsecond and rewritten in the same location in another 0.8 microsecond of one 1.6-microsecond memory cycle.

INSTRUCTION REGISTER (IR)

This 3-bit register contains the operation code of the instruction currently being performed by the machine. The three most significant bits of the current instruction are loaded into the IR from the memory buffer register during a Fetch cycle. The content of the IR is decoded to produce the eight basic instructions, and affect the cycles and states entered at each step in the program.

MAJOR STATE GENERATOR

One or more major states are entered serially to execute programmed instructions or to effect a data break. The major state generator establishes one state for each computer timing cycle. The Fetch, Defer, and Execute states are entered to determine and execute instructions. Entry into these states is produced as a function of the current instruction and the current state. The Word Count, Current Address, and Break states are entered during a data break. The Break state or all three of these states are entered based upon request signals received from peripheral I/O equipment.

Fetch

During this state an instruction is read into the MB from core memory at the address specified by the content of the PC. The instruction is restored in core memory and retained in the MB. The operation code of the instruction is transferred into the IR to cause enactment, and the content of the PC is incremented by one.

If a multiple-cycle instruction is fetched, the following major state will be either Defer or Execute. If a one-cycle instruction is fetched, the operations specified are performed during the last part of the Fetch cycle and the next state will be another Fetch.

Defer

When a 1 is present in bit 3 of a memory reference instruction, the Defer state is entered to obtain the full 12-bit address of the operand from the address in the current page or page 0 specified by bits 4 through 11 of the instruction. The process of address deferring is called indirect addressing because access to the operand is addressed indirectly, or deferred, to another memory location.

Execute

This state is entered for all memory reference instructions except jump. During an AND, two's complement add, or increment and skip if zero instruction, the content of the core memory location specified by the address portion of the instruction is read into the MB and the operation specified by bits 0 through 2 of the instruction is performed. During a deposit and clear accumulator instruction the content of the AC is transferred into the MB and is stored in core memory at the address specified in the instruction. During a jump to subroutine instruction this state occurs to write the content of the PC into the core memory address designated by the instruction and to transfer this address into the PC to change program control.

Word Count

This state is entered when an external device supplies signals requesting a data break and specifying that the break should be a 3-cycle break. When this state occurs, a transfer word count in a core memory location designated by the device is read into the MB, is incremented by 1, and is rewritten in the same location. If the word count overflows, indicating that the desired number of data break transfers will be enacted at completion of the current break, the computer transmits a signal to the device. The Current Address state immediately follows the Word Count state.

Current Address

As the second cycle of a 3-cycle data break, this cycle establishes the address for the transfer that takes place in the following cycle (Break state). Normally the location following the word count is read from core memory into the MB, is incremented by 1 to establish sequential addresses for the transfers, and is transferred into the MA to determine the address selected for the next cycle. When the word count operation is not used, the device supplies an inhibit signal to the computer so that the word read during this cycle is not incremented. Transfers occur at sequential addresses due to incrementing during the Word Count state. During this sequence the word in the MB is rewritten at the same location and the MB is cleared at the end of the cycle. The Break state immediately follows the Current Address state.

Break

This state is entered to enact a data transfer between computer core memory and an external device, either as the only state of a 1-cycle data break or as the final state of a 3-cycle data break. When a break request signal arrives and the cycle select signal specifies a 1-cycle break, the computer enters the Break state at the completion of the current instruction. Information transfers occur between the external device and a device-specified core memory location, through the MB. When this transfer is complete, the program sequence resumes from the point of the break. The data break does not affect the content of the AC, L, and PC.

OUTPUT BUS DRIVERS

Output signals from the computer processor are power amplified by output bus driver modules of the standard PDP-8; allowing these signals to drive a heavy circuit load.

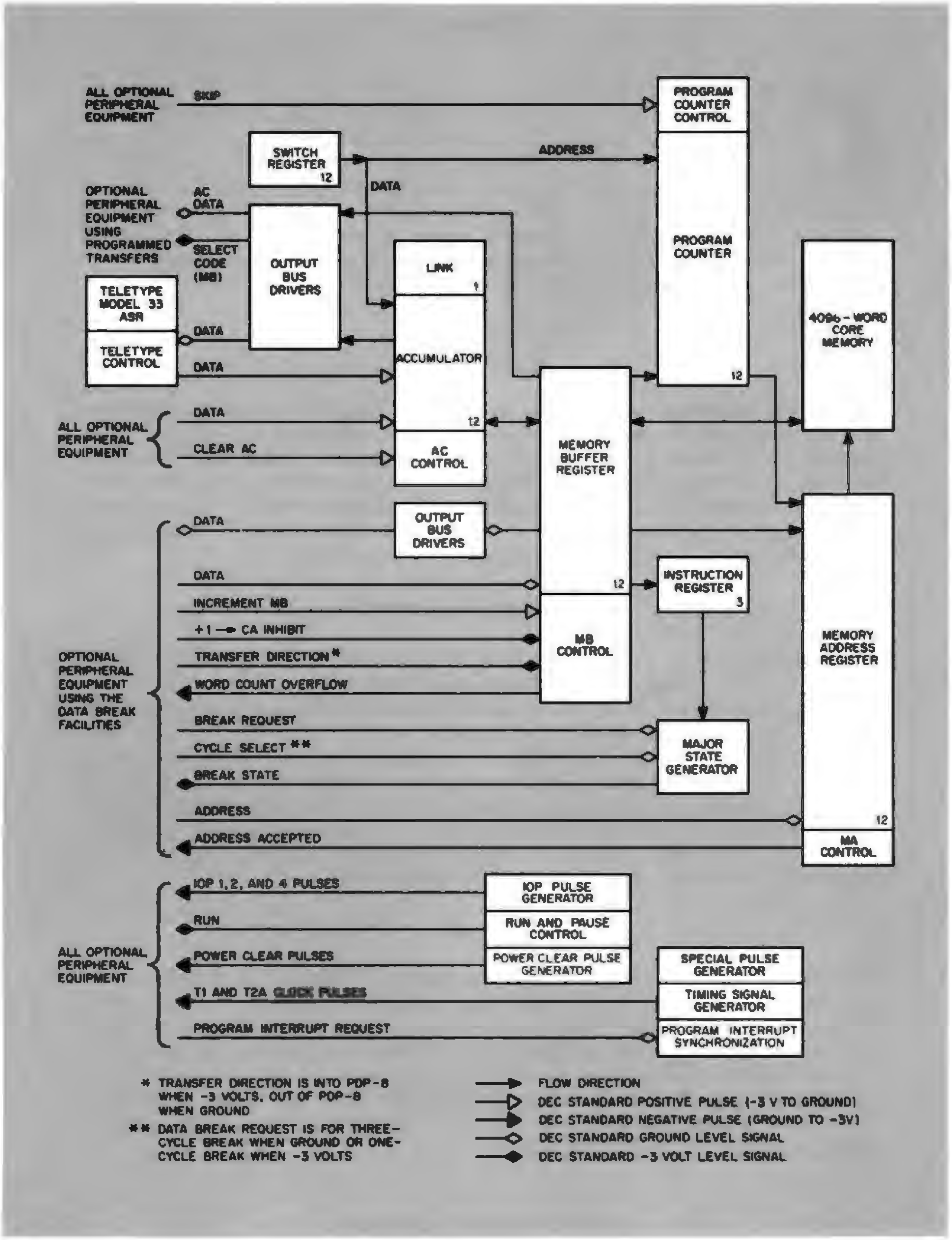


Figure 3 PDP-8 Timing and Control Element Block Diagram

FUNCTIONAL SUMMARY

Operation of the computer is accomplished on a limited scale by keys on the operator console. Operation in this manner is limited to address and data storage by means of the switch register, core memory data examination, the normal start/stop/continue control, and the single step or single instruction operation that allows a program to be monitored visually as a maintenance operation. Most of these manually initiated operations are performed by executing an instruction in the same manner as by automatic programming, except that the gating is performed by special pulses rather than by the normal clock pulses. In automatic operation, instructions stored in core memory are loaded into the memory buffer register and executed during one or more computer cycles. Each instruction determines the major control states that must be entered for its execution. Each control state lasts for one 1.5-microsecond computer cycle and is divided into distinct time states which can be used to perform sequential logical operations. Performance of any function of the computer is controlled by gating of a specific instruction during a specific major control state and a specific time state.

Timing and Control Elements

The circuit elements that determine the timing and control, of the operation of the major registers of the PDP-8 are added to Figure 2 to form Figure 3. Figure 3 shows the timing and control elements described in the succeeding paragraphs and indicates their relationship to the major registers. These elements can be grouped categorically into timing generators, register controls, and program controls.

TIMING GENERATORS

Timing pulses used to determine the computer cycle time and used to initiate sequential time-synchronized gating operations are produced by the timing signal generator. Timing pulses used during operations resulting from the use of the keys and switches on the operator console are produced by the special pulse generator. Pulses that reset registers and control circuits during power turn on and turn off operations are produced by the power clear pulse generator. Several of these pulses are available to, peripheral devices using programmed or data break information transfers.

REGISTER CONTROLS

Operation of the AC, MA, MB, and PC is controlled by an associated logic circuit. These circuits, in turn, transmit and receive control signals to and from I/O equipment. Programmed data transfer equipment can supply a pulse to the AC control to clear the AC prior to a data input and can supply a pulse to cause the content of the PC to be incremented, thus initiating an instruction skip. Equipment using the data break facility passes signals with the MA control and MB control to determine the direction and timing of data transfers in this mode.

PROGRAM CONTROLS

Circuits are also included in the PDP-8 that produce the IOP pulses which initiate operations involved in input/output transfers, determine the advance of the computer program, and allow peripheral equipment to cause a program interrupt of the main computer program to transfer program control to a subroutine which performs some service for the I/O device.

Interface

The input/output portion of the PDP-8 is extremely flexible and interfaces readily with special equipment, especially in real time data processing and control environments.

The PDP-8 utilizes a "bus" I/O system rather than the more conventional "radial" system. The "bus" system allows a single set of data and control lines to communicate with all I/O devices. The bus simply goes from one device to the next. No additional connections to the computer are required. A "radial" system requires that a different set of signals be transmitted to each device; and thus the computer must be modified when new devices are added. The PDP-8 need not be modified when adding new peripheral devices.

External devices receive two types of information from the computer: data and control signals. Computer output data is present as static levels on 12 lines. These levels represent a 12-bit word to be transmitted in parallel to a device. Data signals are received at all devices but are sampled only by the appropriate one in response to a control signal. Control signals are of two types: levels and timing pulses. Six static levels and their complement are supplied by the MB on 12 lines. These lines contain a code representing the device from which action is required. Each device recognizes its own code and performs its function only when this code is present. There are three timing pulses which may be programmed to occur. These IOP pulses are separated in time by one microsecond and are brought to all devices on 3 lines. These pulses are used by a device only when it is selected by the appropriate code on the level lines. They may be used to perform sequential functions in an external register, such as clear and read, or any other function requiring one, two, or three sequential pulses.

Peripheral devices transmit information to the computer on four types of "busses". These are the information bus, the clear AC bus, the skip bus, and the program interrupt bus. The information bus consists of 12 lines normally held at −3 volts by load resistors within the computer. Whenever one of these lines is brought to ground, a binary 1 will be placed in the corresponding accumulator bit. Each device may use the input bus only when it is selected; and thus, these input lines are time shared among all of the connected devices. The skip bus is electrically identical to the information bus. However, when it is driven to ground the next sequential instruction will be skipped. It too can be used only by the device currently selected and is effectively time shared. The program interrupt bus may be driven to ground at any time by any device whether currently selected or not. When more than one device is connected to the interrupt bus they should also be connected to the skip bus so the program can identify the device requesting program interruption.

The transmission of device selection levels and timing pulses is completely under program control. A single instruction can select any one of 64 devices and transmit up to three IOP timing pulses. Since the timing pulses are individually programmable, one might be used to strobe data into an external device buffer, another to transmit data to the computer, and the third to test a status flip-flop and drive the skip bus to ground if it is in the enabling state.

Data transfers may also be made directly with core memory at a high speed using the data break facility. This is a completely separate I/O system from the one described previously. It is standard equipment in every PDP-8 and is ordinarily used with fast I/O devices such as magnetic drums or tapes. Transfers through the data break facility are interlaced with the program in progress. They are initiated by a request from the peripheral device and not by programmed instruction. Thus, the device may transfer a word with memory whenever it is ready and does not have to wait for the program to issue an instruction. Computation may proceed on an interlaced basis with these transfers.

Interface signal characteristics are indicated in Section D of this handbook.

CHAPTER 2

MEMORY AND PROCESSOR INSTRUCTIONS

Instruction words are of two types: memory reference and augmented. Memory reference instructions store or retrieve data from core memory, while augmented instructions do not. All instructions utilize bits 0 through 2 to specify the operation code. Operation codes of 0_8 through 5_8 specify memory reference instructions, and codes of 6_8 and 7_8 specify augmented instructions. Memory reference instruction execution times are multiples of the 1.5-microsecond memory cycle. Indirect addressing increases the execution time of a memory reference instruction by 1.5 microseconds. The augmented instructions, input-output transfer and operate, are performed in 3.75 and 1.5 microseconds respectively.

Memory Reference Instructions

Since the PDP-8 system contains a 4096-word core memory, 12 bits are required to address all locations. To simplify addressing, the core memory is divided into blocks, or pages, of 128 words (200_8 addresses). Pages are numbered 0_8 through 37_8, each field of 4096-words of core memory uses 32 pages. The seven address bits (bits 5 through 11) of a memory reference instruction can address any location in the page on which the current instruction is located by placing a 1 in bit 4 of the instruction. By placing a 0 in bit 4 of the instruction, any location in page 0 can be addressed directly from any page of core memory. All other core memory locations can be addressed indirectly by placing a 1 in bit 3 and placing a 7-bit effective address in bits 5 through 11 of the instruction to specify the location in the current page or page 0 which contains the full 12-bit absolute address of the operand.

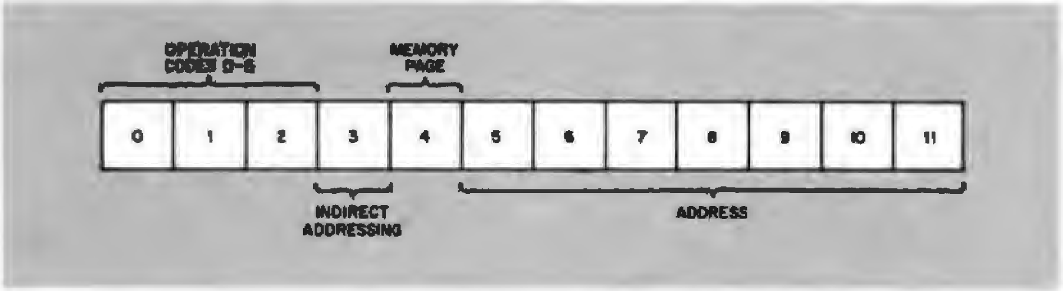


Figure 4 Memory Reference Instruction Bit Assignments

Word format of memory reference instructions is shown in Figure 4 and the instructions perform as follows:

LOGICAL AND (AND Y)

Octal Code: 0

Indicators: AND, FETCH, EXECUTE

Execution Time: 3.0 microseconds with direct addressing, 4.5 microseconds with indirect addressing.

Operation: The AND operation is performed between the content of memory location Y and the content of the AC. The result is left in the AC, the original content of the AC is lost, and the content of Y is restored. Corresponding bits of the AC and Y are operated upon independently. This instruction, often called extract or mask, can be considered as a bit-by-bit multiplication. Example:

Original ACj	Yj	Final ACj
0	0	0
0	1	0
1	0	0
1	1	1

Symbol: ACj $\wedge$ Yj => ACj

TWO'S COMPLEMENT ADD (TAD Y)

Octal Code: 1

Indicators: TAD, FETCH, EXECUTE

Execution Time: 3.0 microseconds with direct addressing, 4.5 microseconds with indirect addressing.

Operation: The content of memory location Y is added to the content of the AC in two's complement arithmetic. The result of this addition is held in the AC, the original content of the AC is lost, and the content of Y is restored. If there is a carry from AC0, the link is complemented. This feature is useful in multiple precision arithmetic.

Symbol: AC0 − 11 + Y0 − 11 => AC0 −11

INCREMENT AND SKIP IF ZERO (ISZ Y)

Octal Code: 2

Indicators: ISZ, FETCH, EXECUTE

Execution Time: 3.0 microseconds with direct addressing, 4.5 microseconds with indirect addressing.

Operation: The content of memory location Y is incremented by one in two's complement arithmetic. If the resultant content of Y equals zero, the content of the PC is incremented by one and the next instruction is skipped. If the resultant content of Y does not equal zero, the program proceeds to the next instruction. The incremented content of Y is restored to memory. The content of the AC is not affected by this instruction.

Symbol: Y + 1 => Y
If resultant Y0 − 11 = 0, then PC + 1 => PC

DEPOSIT AND CLEAR AC (DCA Y)

Octal Code: 3

Indicators: DCA, FETCH, EXECUTE

Execution Time: 3.0 microseconds with direct addressing, 4.5 microseconds with indirect addressing.

Operation: The content of the AC is deposited in core memory at address Y and the

AC is cleared. The previous content of memory location Y is lost.

Symbol: AC => Y
then 0 => AC

JUMP TO SUBROUTINE (JMS Y)

Octal Code: 4

Indicators: JMS, FETCH, EXECUTE

Execution Time: 3.0 microseconds with direct addressing, 4.5 microseconds with indirect addressing.

Operation: The content of the PC is deposited in core memory location Y and the next instruction is taken from core memory location Y + 1. The content of the AC is not affected by this instruction.

Symbol: PC + 1 => Y
Y + 1 => PC

JUMP TO Y (JMP Y)

Octal Code: 5

Indicators: JMP, FETCH

Execution Time: 1.5 microseconds with direct addressing, 3.0 microseconds with indirect addressing.

Operation: Address Y is set into the PC so that the next instruction is taken from core memory address Y. The original content of the PC is lost. The content of the AC is not affected by this instruction.

Symbol: Y => PC

Augmented Instructions

There are two augmented instructions which do not reference core memory. They are the input-output transfer, which has an operation code of 6, and the operate which has an operation code of 7. Bits 3 through 11 within these instructions function as an extension of the operation code and can be microprogrammed to perform several operations within one instruction. Augmented instructions are one-cycle (Fetch) instructions that initiate various operations as a function of bit microprogramming. The operations initiated by each bit occur at a specified time with respect to the computer cycle time and are designated as event times 1, 2, and 3. Three event times, separated by 1 microsecond, occur during the input-output transfer instruction. Two event times occur during the 1.5-microsecond cycle time of an operate instruction.

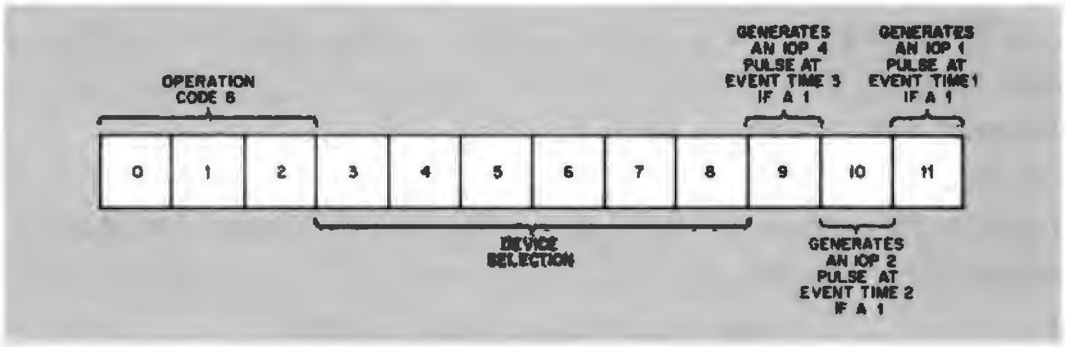
INPUT OUTPUT TRANSFER INSTRUCTION

Microinstructions of the input-output transfer (IOT) instruction initiate operation of peripheral equipment and effect information transfers between the processor and an I/O device. Specifically, when an operation code of 6 is detected, the PAUSE flip-flop is set and the IOP generator is enabled to produce IOP 1, IOP 2, and IOP 4 pulses as a function of the content of instruction bits 9 through 11. These pulses occur at 1-microsecond intervals designated as event times 3, 2, and 1 as follows:

Instruction Bit	IOP Pulse	IOT Pulse	Event Time
11	IOP 1	IOT 1	1
10	IOP 2	IOT 2	2
9	IOP 4	IOT 4	3

The IOP pulses are gated in the device selector of the program-selected equipment to produce IOT pulses that enact a data transfer or initiate a control operation. Selection of an equipment is accomplished by bits 3 through 8 of the IOT instruction. These bits form a 6-bit code that enables the device selector in a given device.

The format of the IOT instruction is shown in Figure 5. Operations performed by IOT microinstructions are explained in Section B of this handbook.



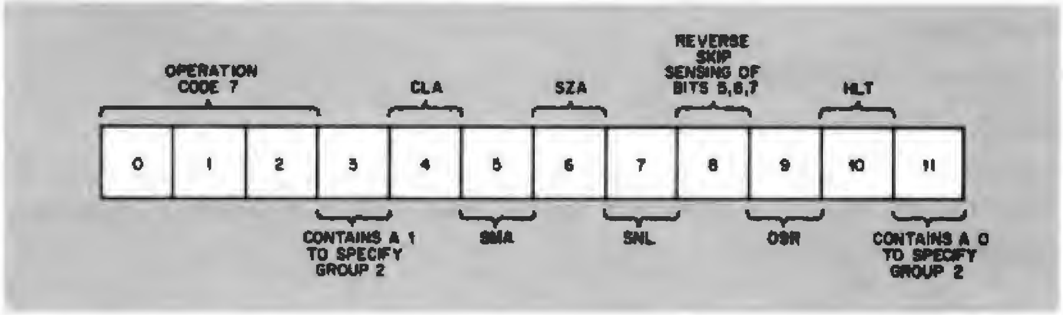
OPERATION CODE 6
GENERATES AN IOP 4 PULSE AT EVENT TIME 3 IF A 1
GENERATES AN IOP 1 PULSE AT EVENT TIME 1 IF A 1
0
1
2
3
4
5
6
7
8
9
10
11
DEVICE SELECTION
GENERATES AN IOP 2 PULSE AT EVENT TIME 2 IF A 1

Figure 5 IOT Instruction Bit Assignments

OPERATE INSTRUCTION

The operate instruction consists of two groups of microinstructions. Group 1 (OPR 1) is principally for clear, complement, rotate, and increment operations and is designated by the presence of a 0 in bit 3. Group 2 (OPR 2) is used principally in checking the content of the accumulator and link and continuing to, or skipping, the next instruction based on the check. A 1 in bit 3 designates an OPR 2 microinstruction.

Group 1 operate microinstruction format is shown in Figure 6 and the microinstructions are explained in the succeeding paragraphs. Any logical combination of bits within this group can be combined into one microinstruction. For example, it is possible to assign ones to bits 5, 6, and 11; but it is not logical to assign ones to bits 8 and 9 simultaneously since they specify conflicting operations. The only restriction on combining OPR 1 operations within one instruction, other than logical conflicts, is that a rotate operation (bits 8, 9, or 10) may not be combined with the increment AC operation (bit 11) since they are executed at the same event time.



OPERATION CODE 7
CLA
SZA
REVERSE SKIP SENSING OF BITS 5,6,7
HLT
0
1
2
3
4
5
6
7
8
9
10
11
CONTAINS A 1 TO SPECIFY GROUP 2
SMA
SNL
OSR
CONTAINS A 0 TO SPECIFY GROUP 2

Figure 6 Group 1 Operate Instruction Bit Assignments

NO OPERATION (NOP)

Octal Code: 7000

Event Time: None

Indicators: OPR, FETCH

Execution Time: 1.5 microseconds

Operation: This command causes a 1-cycle delay in the program and then the next sequential instruction is initiated. This command is used to add execution time to a program, such as to synchronize subroutine or loop timing with peripheral equipment timing.

Symbol: None

INCREMENT ACCUMULATOR (IAC)

Octal Code: 7001

Event Time: 2

Indicators: OPR, FETCH

Execution Time: 1.5 microseconds

Operation: The content of the AC is incremented by one in two's complement arithmetic.

Symbol: AC + 1 => AC

ROTATE ACCUMULATOR LEFT (RAL)

Octal Code: 7004

Event Time: 2

Indicators: OPR, FETCH

Execution Time: 1.5 microseconds

Operation: The content of the AC is rotated one binary position to the left with the content of the link. The content of bits AC1 — 11 are shifted to the next greater significant bit, the content of AC0 is shifted into the L, and the content of the L is shifted into AC11.

Symbol: ACj => ACj −1
ACO => L
L => AC11

ROTATE TWO LEFT (RTL)

Octal Code: 7006

Event Time: 2

Indicators: OPR, FETCH

Execution Time: 1.5 microseconds

Operation: The content of the AC is rotated two binary positions to the left with the content of the link. This instruction is logically equal to two successive RAL operations.

Symbol: ACj => ACj −2
AC1 => L
ACO => AC11
L => AC10

ROTATE ACCUMULATOR RIGHT (RAR)

Octal Code: 7010

Event Time: 2

Indicators: OPR, FETCH

Execution Time: 1.5 microseconds

Operation: The content of the AC is rotated one binary position to the right with the content of the link. The content of bits AC0 – 10 are shifted to the next less significant bit, the content of AC11 is shifted into the L, and the content of the L is shifted into AC0.

Symbol: ACj => ACj +1
AC11 => L
L => AC0

ROTATE TWO RIGHT (RTR)

Octal Code: 7012

Event Time: 2

Indicators: OPR, FETCH

Execution Time: 1.5 microseconds

Operation: The content of the AC is rotated two binary positions to the right with the content of the link. This instruction is logically equal to two successive RAR operations.

Symbol: ACj => ACj +2
AC10 = L
AC11 = AC0
L => AC1

COMPLEMENT LINK (CML)

Octal Code: 7020

Event Time: 1

Indicators: OPR, FETCH

Execution Time: 1.5 microseconds

Operation: The content of the L is complemented.

Symbol: $\bar{L}$ => L

COMPLEMENT ACCUMULATOR (CMA)

Octal Code: 7040

Event Time: 1

Indicators: OPR, FETCH

Execution Time: 1.5 microseconds

Operation: The content of the AC is set to the one's complement of the current content of the AC. The content of each bit of the AC is complemented individually.

Symbol: $\overline{ACj}$ => ACj

COMPLEMENT AND INCREMENT ACCUMULATOR (CIA)

Octal Code: 7041

Event Time: 1, 2

Indicators: OPR, FETCH

Execution Time: 1.5 microseconds

Operation: The content of the AC is converted from a binary value to its equivalent two's complement number. This conversion is accomplished by combining the CMA and IAC commands, thus the content of the AC is complemented during event time 1 and is incremented by one during event time 2.

Symbol: $\overline{ACj} => ACj$,
then $AC + 1 => AC$

CLEAR LINK (CLL)

Octal Code: 7100

Event Time: 1

Indicators: OPR, FETCH

Execution Time: 1.5 microseconds

Operation: The content of the L is cleared to contain a 0.

Symbol: $0 => L$

SET LINK (STL)

Octal Code: 7120

Event Time: 1

Indicators: OPR, FETCH

Execution Time: 1.5 microseconds

Operation: The L is set to contain a binary 1. This instruction is logically equal to combining the CLL and CML commands.

Symbol: $1 = > L$.

CLEAR ACCUMULATOR (CLA)

Octal Code: 7200

Event Time: 1

Indicators: OPR, FETCH

Execution Time: 1.5 microseconds

Operation: The content of each bit of the AC is cleared to contain a binary 0.

Symbol: $0 => AC$

SET ACCUMULATOR (STA)

Octal Code: 7240

Event Time: 1

Indicators: OPR, FETCH

Execution Time: 1.5 microseconds

Operation: Each bit of the AC is set to contain a binary 1. This operation is logically equal to combining the CLA and CMA commands.

Symbol: 1 = > ACj

Group 2 operate microinstruction format is shown in Figure 7 and the primary microinstructions are explained in the following paragraphs. Any logical combination of bits within this group can be composed into one microinstruction. (The instructions constructed by most logical command combinations are listed in Appendix 1.)

If skips are combined in a single instruction the inclusive OR of the conditions determines the skip when bit 8 is a 0; and the AND of the inverse of the conditions determines the skip when bit 8 is a 1. For example, if ones are designated in bits 6 and 7 (SZA and SNL), the next instruction is skipped if either the content of the AC = 0, or the content of L = 1. If ones are contained in bits 5, 7, and 8, the next instruction is skipped if the AC contains a positive number and the L contains a 0.

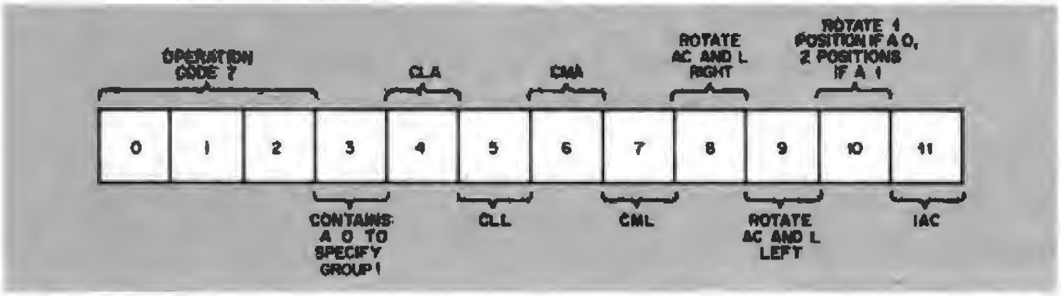


Figure 7 Group 2 Operate Instruction Bit Assignments

HALT (HLT)

Octal Code: 7402

Event Time: 1

Indicators: OPR, not RUN

Execution Time: 1.5 microseconds

Operation: Clears the RUN flip-flop at event time 1, so that the program stops at the conclusion of the current machine cycle. This command can be combined with others in the OPR 2 group that are executed during either event time 1, or 2, and so are performed before the program stops.

Symbol: 0 = > RUN

OR WITH SWITCH REGISTER (OSR)

Octal Code: 7404

Event Time: 2

Indicators: OPR, FETCH

Execution Time: 1.5 microseconds

Operation: The inclusive OR operation is performed between the content of the AC and the content of the SR. The result is left in the AC, the original content of the AC is lost, and the content of the SR is unaffected by this command. When combined with the CLA command, the OSR performs a transfer of the content of the SR into the AC.

Symbol: ACj V SRj = > ACj

SKIP, UNCONDITIONAL (SKP)

Octal Code: 7410

Event Time: 1

Indicators: OPR, FETCH

Execution Time: 1.5 microseconds

Operation: The content of the PC is incremented by one so that the next sequential instruction is skipped.

Symbol: PC + 1 => PC

SKIP ON NON-ZERO LINK (SNL)

Octal Code: 7420

Event Time: 1

Indicators: OPR, FETCH

Execution Time: 1.5 microseconds

Operation: The content of the L is sampled, and if it contains a 1 the content of the PC is incremented by one so that the next sequential instruction is skipped. If the L contains a 0, no operation occurs and the next sequential instruction is initiated.

Symbol: If L = 1, then PC + 1 => PC

SKIP ON ZERO LINK (SZL)

Octal Code: 7430

Event Time: 1

Indicators: OPR, FETCH

Execution Time: 1.5 microseconds

Operation: The content of the L is sampled, and if it contains a 0 the content of the PC is incremented by one so that the next sequential instruction is skipped. If the L contains a 1, no operation occurs and the next sequential instruction is initiated.

Symbol: If L = 0, then PC + 1 => PC

SKIP ON ZERO ACCUMULATOR (SZA)

Octal Code: 7440

Event Time: 1

Indicators: OPR, FETCH

Execution Time: 1.5 microseconds

Operation: The content of each bit of the AC is sampled, and if each bit contains a 0 the content of the PC is incremented by one so that the next sequential instruction is skipped. If any bit of the AC contains a 1, no operation occurs and the next sequential instruction is initiated.

Symbol: If $AC0 - 11 = 0$, then $PC + 1 => PC$

SKIP ON NON-ZERO ACCUMULATOR (SNA)

Octal Code: 7450

Event Time: 1

Indicators: OPR, FETCH

Execution Time: 1.5 microseconds

Operation: The content of each bit of the AC is sampled, and if any bit contains a 1 the content of the PC is incremented by one so that the next sequential instruction is skipped. If all bits of the AC contain a 0, no operation occurs and the next sequential instruction is initiated.

Symbol: If $AC0 - 11 \neq 0$, then $PC + 1 => PC$

SKIP ON MINUS ACCUMULATOR (SMA)

Octal Code: 7500

Event Time: 1

Indicators: OPR, FETCH

Execution Time: 1.5 microseconds

Operation: The content of the most significant bit of the AC is sampled, and if it contains a 1, indicating the AC contains a negative two's complement number, the content of the PC is incremented by one so that the next sequential instruction is skipped. If the AC contains a positive number no operation occurs and program control advances to the next sequential instruction.

Symbol: If $AC0 = 1$, then $PC + 1 => PC$

SKIP ON POSITIVE ACCUMULATOR (SPA)

Octal Code: 7510

Event Time: 1

Indicators: OPR, FETCH

Execution Time: 1.5 microseconds

Operation: The content of the most significant bit of the AC is sampled, and if it contains a 0, indicating a positive (or zero) two's complement number, the content of the PC is incremented by one so that the next sequential instruction is skipped. If the AC contains a negative number, no operation occurs and program control advances to the next sequential instruction.

Symbol: If AC0 = 0, then PC + 1 => PC

CLEAR ACCUMULATOR (CLA)

Octal Code: 7600

Event Time: 1

Indicators: OPR, FETCH

Execution Time: 1.5 microseconds

Operation: Each bit of the AC is cleared to contain a binary 0.

Symbol: 0 => AC

CHAPTER 3

MEMORY AND PROCESSOR BASIC PROGRAMMING

Memory Addressing

The following terms are used in memory address programming:

Term	Definition
Page	A block of 128 core memory locations (200_8 addresses).
Current Page	The page containing the instruction being executed; as determined by bits 0 through 4 of the program counter.
Page Address	An 8-bit number contained in bits 4 through 11 of an instruction which designates one of 256 core memory locations. Bit 4 of a page address indicates that the location is in the current page when a 1, or indicates it is in page 0 when a 0. Bits 5 through 11 designate one of the 128 locations in the page determined by bit 4.
Absolute Address	A 12-bit number used to address any location in core memory.
Effective Address	The address of the operand. When the address of the operand is in the current page or in page 0, the effective address is a page address. Otherwise, the effective address is an absolute address stored in the current page or page 0 and obtained by indirect addressing.

Organization of the standard core memory or any 4096-word field of extended memory is summarized as follows:

Total locations (decimal)	4096
Total addresses (octal)	7777
Number of pages (decimal)	32
Page designations (octal)	0-37
Number of locations per page (decimal)	128
Addresses within a page (octal)	0-177

Four methods of obtaining the effective address are used as specified by combinations of bits 3 and 4.

Bit 3	Bit 4	Effective Address
0	0	The operand is in page 0 at the address specified by bits 5 through 11.
0	1	The operand is in the current page at the address specified by bits 5 through 11.
1	0	The absolute address of the operand is taken from the content of the location in page 0 designated by bits 5 through 11.
1	1	The absolute address of the operand is taken from the content of the location in the current page designated by bits 5 through 11.

The following example indicates the use of bits 3 and 4 to address any location in core memory. Suppose it is desired to add the content of locations A, B, C, and D to the content of the accumulator by means of a routine stored in page 2. The instructions in this example indicate the operation code, the content of bit 4, the content of bit 3, and a 7-bit address. This routine would take the following form:

Page 0		Page 1		Page 2		
Location	Content	Location	Content	Location	Content	Remarks
				R	TAD 00 A	DIRECT TO DATA IN PAGE 0
				S	TAD 01 B	DIRECT TO DATA IN SAME PAGE
				T	TAD 10 M	INDIRECT TO ADDRESS SPECIFIED IN PAGE 0
				U	TAD 11 N	INDIRECT TO ADDRESS SPECIFIED IN SAME PAGE
					• • • •	
A	xxxx	C	xxxx	B	xxxx	
M	C	D	xxxx	N	D	

Routines using 128 instructions, or less, can be written in one page using direct addresses for looping and using indirect addresses for data stored in other pages. When planning the location of instructions and data in core memory, remember that the following locations are reserved for special purposes:

Address	Purpose
0_8	Stores the contents of the program counter following a program interrupt.
1_8	Stores the first instruction to be executed following a program interrupt.
10_8 through 17_8	Auto-indexing.

INDIRECT ADDRESSING

When indirect addressing is specified, the address part (bits 5-11) of a memory reference instruction is interpreted as the address of a location containing not the operand, but containing the address of the operand. Consider the instruction TAD A. Normally, A is interpreted as the address of the location containing the quantity to be added to the content of the AC. Thus, if location 100 contains the number 5432, the instruction TAD 100 causes the quantity 5432 to be added to the content of the AC. Now suppose that location 5432 contains the number 6543. The instruction TAD I 100 (where I signifies indirect addressing) causes the computer to take the number 5432, which is in location 100, as the effective address of the instruction and the number in location 5432 as the operand. Hence, this instruction results in the quantity 6543 being added to the content of the AC.

AUTO-INDEXING

When a location between 10_8 and 17_8 in page 0 of any core memroy field is addressed indirectly (by an instruction in which bit 3 is a 1) the content of that location is read, incremented by one, rewritten in the same location, and then taken as the effective address of the instruction. This feature is called auto-indexing. If location 12_8 contains the number 5432 and the instruction DCA I Z 12 is given, the number 5433 is stored in location 12, and the content of the accumulator is deposited in core memory location 5433.

Storing and Loading

Data is stored in any core memory location by use of the DCA Y instruction. This instruction clears the AC to simplify loading of the next datum. If the data deposited is required in the AC for the next program operation, the DCA must be followed by a TAD Y for the same address.

All loading of core memory information into the AC is accomplished by means of the TAD Y instruction, preceded by an instruction that clears the AC such as CLA or DCA.

Storing and loading of information in sequential core memory locations can make excellent use of an auto-index register to specify the core memory address.

Program Control

Transfer of program control to any core memory location uses the JMP or JMS instructions. The JMP I (indirect address, 1 in bit 3) is used to transfer program control to any location in core memory which is not in the current page or page 0.

The JMS Y is used to enter a subroutine which starts at location Y +1 in the current page or page 0. The content of the PC + 1 is stored in the specified address Y, and address Y + 1 is transferred into the PC. To exit a subroutine the last instruction is a JMP I Y, which returns program control to the location stored in Y.

Indexing Operations

External events can be counted by the program and the number can be stored in core memory. The core memory location used to store the event count can be initialized (cleared) by a CLA command followed by a DCA instruction. Each time the event occurs, the event count can be advanced by a sequence of commands such as CLA, TAD, IAC, and DCA.

The ISZ instruction is used to count repetitive program operations or external events without disturbing the content of the accumulator. Counting a specified number of operations is performed by storing a two's complement negative number equal to the number of iterations to be counted. Each time the operation is performed, the ISZ instruction is used to increment the content of this stored number and check the result. When the stored number becomes zero, the specified number of operations have occured and the program skips out of the loop and back to the main sequence.

This instruction is also used for other routines in which the content of a memory location is incremented without disturbing the content of the accumulator, such as storing information from an I/O device in sequential memory locations or using core memory locations to count I/O device events.

Logic Operations

The PDP-8 instruction list includes the logic instruction, AND Y. From this instruction short routines can be written to perform the inclusive OR and exclusive OR operations.

LOGICAL AND

The logic AND operation between the content of the accumulator and the content of a core memory location Y is performed directly by means of the AND Y instruction. The result remains in the AC, the original content of the AC is lost, and the content of Y is unaffected.

INCLUSIVE OR

Assuming value A is in the AC and value B is stored in a known core memory address, the following sequence performs the inclusive OR. The sequence is stated as a utility subroutine called IOR.

```
/CALLING SEQUENCE                              JMS IOR
/                                              (ADDRESS OF B)
/                                              (RETURN)
/ENTER WITH ARGUMENT IN AC; EXIT WITH LOGICAL RESULT IN AC

            IOR,      0
                      DCA TEM1
                      TAD I IOR
                      DCA TEM2
                      TAD TEM1
                      CMA
                      AND I TEM2
                      TAD TEM1
                      ISZ IOR
                      JMP I IOR
            TEM1,     0
            TEM2,     0
```

EXCLUSIVE OR

The exclusive OR operation for two numbers, A and B, can be performed by a subroutine called by the mnemonic code XOR. In the following general purpose XOR subroutine, the value A is assumed to be in the AC, and the address of the value B is assumed to be stored in a known core memory location.

```
/CALLING SEQUENCE                                   JMS XOR
/                                                   (ADDRESS OF B)
/                                                   (RETURN)
/ENTER WITH ARGUMENT IN AC; EXIT WITH LOGICAL RESULT IN AC

              XOR,       0
                         DCA TEM1
                         TAD I XOR
                         DCA TEM2
                         TAD TEM1
                         AND I TEM2
                         CMA IAC
                         CLL RAL
                         TAD TEM1
                         TAD I TEM2
                         ISZ XOR
                         JMP I XOR
              TEM1,      0
              TEM2,      0
```

An XOR subroutine can be written using fewer core memory locations by making use of the IOR subroutine; however, such a subroutine takes more time to execute. A faster XOR subroutine can be written by storing the value B in the second instruction of the calling sequence instead of the address of B; however, the resulting subroutine is not as utilitarian as the routine given here.

Arithmetic Operations

One arithmetic instruction is included in the PDP-8 order code, the two's complement add: TAD Y. Using this instruction, routines can easily be written to perform addition, subtraction, multiplication, and division in two's complement arithmetic.

TWO'S COMPLEMENT ARITHMETIC

In two's complement arithmetic addition, subtraction, multiplication, and division of binary numbers is performed in accordance with the common rules of binary arithmetic. In PDP-8, as in other machines utilizing complementation techniques, negative numbers are represented as the complement of positive numbers, and subtraction is achieved by complement addition. Representation of negative values in one's complement arithmetic is slightly different from that in two's complement arithmetic.

The one's complement of a number is the complement of the absolute positive value; that is, all ones are replaced by zeros and all zeros are replaced by ones. The two's complement of a number is equal to the one's complement of the positive value plus one.

In one's complement arithmetic a carry from the sign bit (most significant bit) is added to the least significant bit in an end-around carry. In two's complement arithmetic a carry from the sign bit complements the link (a carry would set the link to 1 if it were properly cleared before the operation), and there is no end-around carry.

A one's complement representation of a negative number is always one less than the two's complement representation of the same number. Differences between one's and two's complement representations are indicated in the following list.

Number	1's Complement	2's Complement
+5	000000000101	000000000101
+4	000000000100	000000000100
+3	000000000011	000000000011
+2	000000000010	000000000010
+1	000000000001	000000000001
+0	000000000000	000000000000
−0	111111111111	Nonexistent
−1	111111111110	111111111111
−2	111111111101	111111111110
−3	111111111100	111111111101
−4	111111111011	111111111100
−5	111111111010	111111111011

Note that in two's complement there is only one representation for the number which has the value zero, while in one's complement there are two representations. Note also that complementation does not interfere with sign notation in either one's complement or two's complement arithmetic; bit 0 remains a 0 for positive numbers and a 1 for negative numbers.

To form the two's complement of any number in the PDP-8, the one's complement is formed, and the result is incremented by one. This is accomplished by the instruction CMA combined with an IAC instruction. Since both of these instructions are functions of the OPR 1 instruction and the actions occur at different event times, they can be combined to form the instruction CIA, Complement and Increment AC.

ADDITION

The addition of a number contained in a core memory location and the number contained in the accumulator is performed directly by using the TAD Y instruction, assuming that the binary point is in the same position and that both numbers are properly represented in two's complement arithmetic. Addition can be performed without regard for the sign of either the augend or the addend. Overflow is possible, in which case the result will have an incorrect sign, although the 11 least significant bits will be correct. Following the addition a test for overflow can be made by using the SZL command.

SUBTRACTION

Subtraction is performed by complementing the subtrahend and adding the minuend. As in addition, if both numbers are represented by their two's complement, subtraction can be performed without regard for the sign of either number. Assuming that both numbers are stored in core memory, a routine to find the value of A-B follows:

```
CLA
TAD B        /LOAD SUBTRAHEND INTO AC
CIA          /COMPLEMENT AND INCREMENT B
TAD A        /AC = A - B
```

CHAPTER 4
PROGRAM INTERRUPT

The program interrupt feature allows certain external conditions to interrupt the computer program. It is used to speed the information processing of input-output devices or to allow certain alarms to halt the program in progress and initiate another routine. When a program interrupt request is made the computer completes execution of the instruction in progress before acknowledging the request and entering the interrupt mode. A program interrupt is similar to a JMS to location 0; that is, the content of the program counter is stored in location 0, and the program resumes operation in location 1. The interrupt program commencing in location 1 is responsible for identifying the signal causing the interruption, for removing the interrupt condition, and for returning to the original program. Exit from the interrupt program, back to the original program, can be accomplished by a JMP I Z 0 instruction.

Instructions

The two instructions associated with the program interrupt synchronization element are IOT microinstructions that do not use the IOP generator. These instructions are:

INTERRUPT TURN ON (ION)

Octal Code: 6001

Event Time: Not applicable

Indicators: IOT, FETCH, ION

Execution Time: 1.5 microseconds

Operation: This command enables the computer to respond to a program interrupt request. If the interrupt is disabled when this instruction is given, the computer executes the next instruction, then enables the interrupt. The additional instruction allows exit from the interrupt subroutine before allowing another interrupt to occur. This instruction has no affect upon the condition of the interrupt circuits if it is given when the interrupt is enabled.

Symbol: 1 => INT. ENABLE

INTERRUPT TURN OFF (IOF)

Octal Code: 6002

Event Time: Not applicable

Indicators: IOT, FETCH

Execution Time: 1.5 microseconds

Operation: This command disables the program interrupt synchronization element to prevent interruption of the current program.

Symbol: 0 => INT. ENABLE, INT. DELAY

Programming

When an interrupt request is acknowledged, the interrupt is automatically disabled by the program interrupt synchronization circuits (not by instructions). The next instruction is taken from core memory location 1. Usually the instruction stored in location 1 is a JMP, which transfers program control to a subroutine which services the interrupt. At some time during this subroutine an ION instruction must be given. The ION can be given at the end of the subroutine to allow other interrupts to be serviced after program control is transferred back to the original program. In this application, the ION instruction immediately precedes the last instruction in the routine. A delay of one instruction (regardless of the execution time of the following instruction) is inherent in the ION instruction to allow transfer of program control back to the original program before enabling the interrupt. Usually exit from the subroutine is accomplished by a JMP I Z 0 instruction.

The ION command can be given during the subroutine as soon as it has determined the I/O device causing the interrupt. This latter method allows the subroutine which is handling a low priority interrupt to be interrupted, possibly by a high priority device. Programming of an interrupt subroutine which checks for priority and allows itself to be interrupted, must make provisions to relocate the content of the program counter stored in location 0; so that if interrupted, the content of the PC during the subroutine is stored in location 0, and the content of the PC during the original program is not lost.

CHAPTER 5
DATA BREAK

Peripheral equipment connected to the data break facility can cause a temporary suspension in the program in progress to transfer information with the computer core memory, via the MB. One I/O device can be connected directly to the data break facility or up to seven devices can be connected to it through the Type DM01 Data Multiplexer. This cycle stealing mode of operation provides a high-speed transfer of individual words or blocks of information at core memory addresses specified by the I/O device. Since program execution is not involved in these transfers, the program counter, accumulator, and instruction register are not disturbed or involved in these transfers. The program is merely suspended at the conclusion of an instruction execution and the data break is entered to perform the transfer, then the Fetch state is entered to continue the main program.

Data breaks are of two basic types: single-cycle and three-cycle. In a single-cycle data break, registers in the device (or device interface) specify the core memory address of each transfer and count the number of transfers to determine the end of data blocks. In the three-cycle data break two computer core memory locations perform these functions, simplifying the device interface by omitting two hardware registers.

The computer receives the following signals from the device during a data break:

Signal	−3 Volts	0 Volts
Break Request	No break request	Break request
Cycle Select	One-cycle break	Three-cycle break
Transfer Direction	Data into PDP-8	Data out of PDP-8
Increment CA Inhibit	CA incremented	CA not incremented
Increment MB (pulse)	MB not incremented	MB incremented
Address (12 bits)	Binary 0	Binary 1
Data (12 bits)	Binary 0	Binary 1

The computer sends the following signals to the device during a data break:

Signal	Characteristics
Data (12 bits)	−3 volts = binary 0, 0 volts = binary 1
Address Accepted	400-nanosecond negative pulse beginning at memory done time
WC Overflow	400-nanosecond negative pulse occurring at T1 time
Buffered Break	−3 volts when in Break state

To initiate a data break an I/O device must supply four signals simultaneously to the data break facility. These signals are the Break Request signal, which sets the BRK SYNC flip-flop in the major state generator to control entry into the data break states (Word Count for a three-cycle data break or Break for a single-cycle data break); a Transfer Direction signal, supplied to the MB control element to allow data to be strobed into the MB from the peripheral equipment and to inhibit reading from core memory; a Cycle Select signal which controls gating in the major state generator to determine if the one-cycle or three-cycle data break is to be selected; and a core memory address of the transfer which is supplied to the input of the MA. When the break request is made, the data break replaces entry into the Fetch state of an instruction. Therefore the data break is entered at the conclusion of the Execute state of most memory reference instructions and at the conclusion of a Fetch state of augmented instructions. Having established the data break, each machine cycle is a Word Count, Current Address, or Break cycle until all data transfers have taken place, as indicated by removal of the Break Request signal by the peripheral equipment.

More exactly, the Break Request signal enables a diode-capacitor-diode gate at the binary 1 input of the BRK SYNC flip-flop. Midway through each computer cycle (T1) this gate is pulsed to set the flip-flop if the Break Request signal has been received.

At the beginning (T2) of each machine cycle the major state generator is set to establish the state for the cycle. At this time the status of the BRK SYNC flip-flop is sampled and the flip-flop is cleared. If the BRK SYNC flip-flop is in the 1 state at this time, the Word Count or Break state is set into the major state generator and a data break commences.

Therefore, to initiate a data break, the Break Request must be at ground potential for at least 400 nanoseconds preceding T1 of the cycle preceding the data break cycle. A Break Request signal should be supplied to the computer when the address, data, Transfer Direction and Cycle Select signals are supplied to the computer, and not before.

When a data break occurs, the address designated by the device is loaded into the MA during time T2 of the last cycle of the current instruction, and the major state generator is set to the Word Count state if the Cycle Select signal is at ground, or is set to the Break state if this signal is at −3 volts. The program is delayed for the duration of the data break, commencing in the following cycle. A break request is granted only after completion of the current instruction as specified by the following conditions:

1. At the end of the Fetch cycle of an OPR or IOT instruction, or a directly-addressed JMP instruction.

2. At the end of the Defer cycle of an indirectly addressed JMP instruction.

3. At the end of the Execute cycle of a JMS, DCA, ISZ, TAD, or AND instruction.

At the beginning of the Word Count cycle of a three-cycle data break or the Break cycle of a one-cycle data break the address supplied to the input of the MA is strobed into the MA and the computer supplies an Address Accepted pulse to the device. Entry into the Break cycle is indicated to the peripheral equipment by a Buffered Break signal and by an Address Accepted pulse that can be used to enable gates in the device to perform tasks associated with the transfers. The Addresss Accepted pulse is the most convenient control to be used by I/O equipment to disable the Break Request signal, since this signal must be removed at the end of T2 time to prevent continuance at the data break into the next cycle. Also at the beginning of the Break cycle, the MB is cleared in preparation for receipt of data from either the core memory or the external device. If the Transfer Direction signal establishes the direction as out of the computer, the content of the core memory register at the address specified is transferred into the MB and is immediately available for strobing by the peripheral equipment. If the Transfer Direction signal specifies a data direction into the PDP-8, reading from core memory is inhibited and data is transferred into the MB from peripheral equipment.

The status of the BRK SYNC flip-flop is sensed at the beginning of a Break cycle to determine if an additional Break cycle is required. If a Break Request signal has been received since T2, the Break state is maintained in the major state generator; if the Break Request signal has not been received by this time; the Fetch state is set into the major state generator to continue the program. The Break Request signal should be removed by the end of the Address Accepted signal if additional Break cycles are not required.

Single-Cycle Data Break

One-cycle breaks transfer a data word into the computer core memory from the device, transfer a data word into a device from the core memory, or increment the content of a device-specified core memory location. In each of these types of data break one computer cycle is stolen from the program by each transfer; Break cycles occur singly (interleaved with the program steps) or continuously (as in a block transfer), depending upon the timing of the Break Request signal.

During the memory strobe portion of the Break cycle, the content of the addressed cell is read into the MB if the transfer direction is out of the computer (into the I/O device). If the transfer direction is into the computer, generation of the Memory Strobe pulse is inhibited so that the MB (cleared during the previous cycle) remains cleared. Information is transferred from the output data register of the I/O device into the MB and is written into core memory during time T1 of the Break cycle. In an outward transfer, the write operation restores the original content of the address cell to memory.

The MB is cleared during time T2 of the Break cycle. If there is a further break request, another Break cycle is initiated. If there is no break request, the content of the PC is transferred into the MA, the IR is cleared, and the major state generator is set to Fetch. The program then executes the next instruction.

The increment MB facility is useful for counting iterations or events by means of a data break, so that the PC and AC are not disturbed. Within one Break cycle of 1.5 microseconds, a word is fetched from a device-specified core memory location, is incremented by one, and is restored to the same memory location. The Increment MB signal input must be supplied to the computer only during a Break cycle in which the direction of transfer is out of the PDP-8. These restrictions can be met by a simple AND gate in the device; an Increment MB signal is generated only when an event occurs, the Buffered Break signal from the computer is present, and the Transfer Direction signal supplied to the computer is at ground potential.

Three-Cycle Data Break

The three-cycle data break provides an economical method of controlling the transfer of data between the computer core memory and fast peripheral devices. Transfer rates in excess of 220 kc are possible using this feature of the PDP-8.

The three-cycle data break differs from the one-cycle break in that a ground-level Cycle Select signal is supplied so that when the data break conditions are fulfilled the program is suspended and the Word Count state is entered. The Word Count state is entered to increment the fixed core memory location containing the word count. The device requesting the break supplies this address as in the one-cycle break, except that this is a fixed address supplied by wired ground and -3v signals rather than from a register. The only restriction on this address is that it must be an even number (bit $11 = 0$).

Following the Word Count state a Current Address state occurs in which the location following the Word Count address (bit 11 = 1 after + 1 => MA) is read, incremented by one, restored to memory, and loaded into the MA to be used as the transfer address. Then the normal Break state is entered to effect the transfer between the device and the computer memory cell specified by the MA.

WORD COUNT STATE

When this state is entered the content of the core memory address specified by the external device is read into the MB during time state T1. The word count, established previously by instructions, is the 2's complement negative number equal to the required number of transfers. The word in the MB is incremented by 1 to advance the word count, and if the word becomes 0 when incremented, the computer generates a WC Overflow pulse and supplies it to the device. During time T2 the incremented word count is rewritten in memory, the MB is cleared, the content of the MA is incremented by 1 to establish the next location as the address for the following memory cycle, and the major state generator is set to the Current Address state.

CURRENT ADDRESS STATE

Operations during the second cycle of the three-cycle data break depend upon the condition of the Increment CA Inhibit (+1 → CA Inhibit) signal supplied to the computer from the I/O device. During T1 the address following the word count is read into the MB. If the Increment CA Inhibit signal is at ground potential, no further operations occur during T1. If this signal is at −3v, the content of the MB is incremented by 1 during T1 to advance the address of the transfer to the next sequential location. During T2, the content of the MB is rewritten into core memory, the address word in the MB is transferred into the MA to designate the address to be used in the succeeding memory cycle, the MB is cleared, and the major state generator is set to the Break state.

BREAK STATE

The actual transfer of data between the external device and the core memory, through the MB, occurs during the Break state as during a single-cycle data break, except that the address is determined by the current content of the MA rather than directly by the device.

CHAPTER 6

MEMORY EXTENSION CONTROL TYPE 183 AND MEMORY MODULE TYPE 184

Extension of the storage capacity of the standard 4096-word core memory is accomplished by adding fields of 4096-word core memories, each field being a Type 184 Memory Module. Field select control and address extension control for Type 184 Memory Modules are provided by the Type 183 Memory Extension Control. Up to seven fields can be added to the standard 4096-word memory, providing a maximum storage of 32,768 words. Direct addressing of 32,768 words require 15 binary bits ($2^{15} = 32,768$). However, since programs and data need not be directly addressed for execution of each instruction, a field can be program-selected, and all 12-bit addresses are then assumed to be within the current memory field. Program interrupt of a program in any field automatically specifies field 0, address 0 for storage of the program count. The memory extension control consists of several 3-bit flip-flop registers that extend addresses to 15 bits to establish or select a field.

Addition of a memory extension control to a standard PDP-8 requires a simple modification of the operator console to activate indicators and switches associated with the instruction field register and the data field register of the control. These switches function in the same manner as the switch register, to load information into associated registers when the LOAD ADDRESS key is pressed.

The seven functional circuit elements which comprise the memory extension control perform as follows:

Instruction Field Register (IF): The IF is a 3-bit register that serves as an extension of the PC. The content of the IF determines the field from which all instructions are taken and the field from which operands are taken in directly-addressed AND, TAD, ISZ, or DCA instructions. Operating the LOAD ADDRESS key clears the IF, then sets it by a transfer of ones from the INSTRUCTION FIELD switch register on the operator console. During a JMP or JMS instruction the IF is set by a transfer of information contained in the instruction buffer register. When a program interrupt occurs, the content of the IF is automatically stored in bits 0 through 2 of the save field register for restoration to the IF from the instruction buffer register at the conclusion of the program interrupt subroutine.

Data Field Register (DF): This 3-bit register determines the memory field from which operands are taken in indirectly-addressed AND, TAD, ISZ, or DCA instructions. The DF is cleared and set by a ones transfer of information contained in the DATA FIELD switch register by operation of the LOAD ADDRESS key. The DF is set by a transfer of information from bits 6 through 8 of the MB during a CDF microinstruction to establish a microprogrammed data field. When a program interrupt occurs, the content of the DF is automatically stored in the save field register. The DF is set by a transfer of information from bits 3 through 5 of the save field register by the RMF microinstruction to restore the data field at the conclusion of the program interrupt subroutine.

Instruction Buffer Register (IB): The IB serves as a 3- bit input buffer for the instruction field register. All field number transfers into the instruction field register are made through the instruction buffer, except transfers from the operator console switches. The IB is cleared and set by operation of the LOAD ADDRESS key in the same manner as the instruction field register. A CIF microinstruction loads the IB with the programmed field number contained in MB 6-8. An RMF microinstruction transfers the content of bits 0 through 2 of the save field register into the IB to restore the instruction field to the conditions that existed prior to a program interrupt.

Save Field Register (SF): When a program interrupt occurs, this 6-bit register is cleared, then loaded from the instruction field and data field registers. The RMF microinstruction can be given immediately prior to the exit from the program interrupt subroutine to restore the instruction field and data field by transferring the content of the SF into the instruction buffer and the data field register. The SF is cleared during the cycle in which the program count is stored at address 0000 of the JMS instruction forced by a program interrupt request, then the instruction field and data field are strobed into the SF.

Start Field Signal Generator: When the PDP-8 core memory capacity is extended, the standard memory is designated as field 0. This circuit produces the Enable Field 0 signal when data field 0 is selected, instruction field 0 is selected, or when break field 0 is selected. Similar circuits are provided for each of the other (up to seven) fields.

Accumulator Transfer Gating: This gating allows the content of the save field register, instruction field register, or the data field register to be strobed into the accumulator. Transfer of information in this manner is accomplished by circuits which sample the content of registers and supply positive pulses to the AC upon receipt of IOT command pulses. During an RIB microinstruction, bits 6 through 11 of the AC are set by the content of the save field register. During an RIF microinstruction, bits 6 through 8 of the AC are set by the content of the instruction field register. During an RDF microinstruction, bits 6 through 8 of the AC are set by the content of the data field register.

Device Selector: Bits 3 through 5 of the IOT instruction are decoded to produce the IOT command pulses for the memory extension control. Bits 6 through 8 of the instruction are not used for device selection since they specify a field number in some commands. Therefore, the select code for this device selector is designated as 2X.

Each Type 184 Memory Module consists of a core array, address selection circuits, inhibit selection circuits, sense amplifiers, and memory drivers which are identical with these in the standard PDP-8.

Instructions

The instructions for the Type 183 option do not use the IOP generator and extend the IOT instruction list to include the following:

CHANGE TO DATA FIELD N (CDF)

Octal Code: 62N1

Event Time: Not applicable

Indicators: IOT, FETCH

Execution Time: 1.5 microseconds

Operation: The data field register is loaded with the program-selected field number (N = 0 to 7). All subsequent memory requests for operands are automatically switched to that data field until the data field number is changed by a new CDF command, or during a program interrupt.

Symbol: MB6 − 8 = > DF

CHANGE INSTRUCTION FIELD (CIF)

Octal Code: 62N2

Event Time: Not applicable

Indicators: IOT, FETCH

Execution Time: 1.5 microseconds

Operation: The instruction buffer register is loaded with the program-selected field number (N = 0 to 7). The next JMP or JMS instruction causes the new field to be entered.

Symbol: MB6 − 8 = > IB

READ DATA FIELD (RDF)

Octal Code: 6214

Event Time: Not applicable

Indicators: IOT, FETCH

Execution Time: 1.5 microseconds

Operation: The content of the data field register is transferred into bits 6, 7, 8 of the AC. All other bits of the AC are unaffected.

Symbol: DF = > AC6 − 8

READ INSTRUCTION FIELD (RIF)

Octal Code: 6224

Event Time: Not applicable

Indicators: IOT, FETCH

Execution Time: 1.5 microseconds

Operation: The content of the instruction field register is transferred into bits 6, 7, 8 of the AC. All other bits of the AC are unaffected.

Symbol: IF = > AC6 − 8

READ INTERRUPT BUFFER (RIB)

Octal Code: 6234

Event Time: Not applicable

Indicators: IOT, FETCH

Execution Time: 1.5 microseconds

Operation: The instruction field and data field held in the save field register during a program interrupt are transferred into bits 6 through 8, and 9 through 11 of the AC respectively.

Symbol: SF 0 – 2 => AC6 – 8
SF 3 – 5 => AC9 – 11

RESTORE MEMORY FIELD (RMF)

Octal Code: 6244

Event Time: Not applicable

Indicators: IOT, FETCH

Execution Time: 1.5 microseconds

Operation: This command is used upon exit from the program interrupt subroutine in another field. The data and instruction fields that were interrupted by the subroutine are restored by transferring the content of the save field register into the instruction buffer and data field registers.

Symbol: SF 0 – 2 => IB
SF 3 – 5 => DF

Programming

Instructions and data are accessed from the currently assigned instruction and data fields, where instructions and data may be stored in the same or different memory fields. When indirect memory references are executed, the operand address refers first to the instruction field to obtain an effective address, which in turn, refers to a location in the currently assigned data field. All instructions and operands are obtained from the field designated by the content of the instruction field register, except for indirectly-addressed operands which are specified by the content of the data field register. In other words, the DF is effective only in the Execute cycle that is directly preceded by the Defer cycle of a memory reference instructions, as follows:

Indirect (Bit 3)	Page or Z Bit (Bit 0)	Field In IF	Field In DF	Effective Address
0	0	m	n	The operand is in page 0 of field m at the page address specified by bits 5 through 11.
0	1	m	n	The operand is in the current page of field m at the page address specified by bits 5 through 11.
1	0	m	n	The absolute address of the operand in field n is taken from the content of the location in page 0 of field m designated by bits 5 through 11.
1	1	m	n	The absolute address of the operand in field n is taken from the content of the location in the current page of field m designated by bits 5 through 11.

Each field of extended memory contains eight autoindex registers in addresses 10 through 17. For example, assume that a program in field 2 is running (IF = 2) and using operands in field 1 (DF = 1) when the instruction TAD I 10 is fetched. The Defer cycle is entered (bit 3 = 1) and the content of location 10 in field 2 is read, incremented, and rewritten. If address 10 in field 2 originally contained 4321, it now contains 4322. In the Execute cycle the operand is fetched from location 4322 of field 1.

Program control is transferred between memory fields by the CIF commands. This instruction does not change the instruction field directly, since this would make it impossible to execute the next sequential instruction. The CIF instruction sets the new instruction field into the IB for automatic transfer into the IF when either a JMP or JMS instruction is executed. The DF is unaffected by the JMP and JMS instructions. The 12-bit program counter is set in the normal manner and, since the IF is an extension on the most significant end of the PC, program sequence resumes in the new memory field following a JMP or JMS. Entry into a program interrupt is inhibited after the CIF instruction until a JMP or JMS is executed.

To call a subroutine that is out of the current field, the data field register is set to indicate the field of the calling JMS, which establishes the location of the operands as well as the identity of the return field. The instruction field is set to the field of the starting address of the subroutine. The following sequence returns program control to the main program from a subroutine that is out of the current field.

```
/PROGRAM OPERATIONS IN MEMORY FIELD 2
/INSTRUCTION FIELD = 2; DATA FIELD = 2
/CALL A SUBROUTINE IN MEMORY FIELD 1
/INDICATE CALLING FIELD LOCATION BY THE CONTENT OF THE DATA FIELD

              CIF       10              /CHANGE TO INSTRUCTION
                                        /FIELD 1 = 6212
              JMS       1  SUBRP        /SUBRP = ENTRY ADDRESS
              CDF       20              /RESTORE DATA FIELD
SUBRP,        SUBR                      /POINTER
/CALLED SUBROUTINE
              0                         /SUBR = PC + 1 AT CALLING POINT
              RDF                       /READ DATA FIELD INTO AC
              TAD  RETURN               /CONTENT OF THE AC = 6202 + DATA
                                        /FIELD BITS
              DCA  EXIT                 /STORE INSTRUCTION SUBROUTINE
               .
               .
               .
               .
EXIT,         0                         /A CIF INSTRUCTION
              JMP I SUBR                /RETURN
RETURN,       CIF
```

When a program interrupt occurs, the current instruction and data field numbers are automatically stored in the 6-bit save field register, then the IF and DF are cleared. The 12-bit program count is stored in location 0000 of field 0 and program control advances to location 0001 of field 0. At the end of the program interrupt subroutine the RMF instruction restores the IF and DF from the content of the SF. The following instruction sequence at the end of the program interrupt subroutine continues the interrupted program after the interrupt has been processed:

```
        .               /RESTORE MQ IF REQUIRED
        .
        .
        .
        .               /RESTORE L IF REQUIRED
        .
        .
        .
CLA
TAD  AC                 /RESTORE AC
RMF                     /LOAD IB FROM SF
ION                     /TURN ON INTERRUPT SYSTEM
JMP  I 0                /RESTORE PC WITH CONTENT OF
                        /LOCATION 0 AND LOAD IF FROM IB
```

A device using the computer data break facility supplies a 12-bit address to the MA and a 3-bit address extension to the Memory Extension Control Type 183. The address extension is received by a break field decoder which selects the memory field used for the data break.

CHAPTER 7
MEMORY PARITY TYPE 188

Data transmission checking of each word written in and read from core memory is provided by this option. The option replaces the 12-bit core memory with a 13-bit system (driving, inhibiting, sensing circuits as well as a core array constructed of 13 planes) and includes a parity generator and.a parity checking circuit. The parity generator produces the 13th bit for each 12-bit data word written in core memory so that the entire word contains an odd number of binary ones. The parity checking circuit monitors each word read from core memory to assure that the odd parity is maintained. If a word read contains an even number of ones a transmission error is indicated by setting a parity error flag. This flag is connected to the program interrupt synchronization element of the computer to initiate a program interrupt subroutine. This routine sequentially checks all equipment error flags to determine the option causing the interrupt and initiates an appropriate service and returns to the main program, or provides a suitable error printout and halts programmed operations. Upon determining that a memory parity error has occurred the program interrupt subroutine can repeat the main program step that caused the error to check the reliability of the error condition, can perform a simple write/read/check routine at the error address, or can determine the status of the machine when the error was detected and re-establish or print out these conditions and halt.

Instructions

Two instructions are associated with the Type 188 option. They are:

SKIP ON NO MEMORY PARITY ERROR (SMP)

Octal Code: 6104

Event Time: 3

Indicator: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The memory parity error flag is sensed and if it contains a 0 (signifying no error has been detected) the PC is incremented so that the next successive instruction is skipped.
Symbol: If Memory Parity Error Flag = 0, then PC + 1 => PC

CLEAR MEMORY PARITY ERROR FLAG (CMP)

Octal Code: 6102

Event Time: 2

Indicator: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The memory parity error flag is cleared.

Symbol: 0 => Memory Parity Error Flag

Programming

Both instructions for this option are used in the program interrupt subroutine and in diagnostic maintenance programs. The SMP command is used as a programmed check for memory parity error. In the program interrupt subroutine this command can be followed by a jump to a portion of the routine that services the memory parity option as described previously. The CMP command is used to initialize the memory parity option in preparation for normal programmed operation of the computer.

CHAPTER 8

EXTENDED ARITHMETIC ELEMENT TYPE 182

This option consists of circuits that perform parallel arithmetic operations on positive binary numbers. A 12-bit multiplier quotient register (MQ), a 5-stage step counter (SC), and various shifting and control logic constitute the option. The AC and MB are used in conjunction with these logic elements to perform arithmetic operations. With the addition of this option to a PDP-8 system, indicators on the operator console for the content of each bit of the MQ are activated and a class of instructions is added to the Group 2 Operate instruction list.

Instructions

The exended arithmetic element (EAE) microinstructions are specified by an operate instruction (operation code 7) in which bits 3 and 11 contain binary ones. Being augmented instructions, the EAE commands are microprogrammed and can be combined with each other to perform non-conflicting logical operations. Format and bit assignments of the EAE commands are indicated in Figure 8

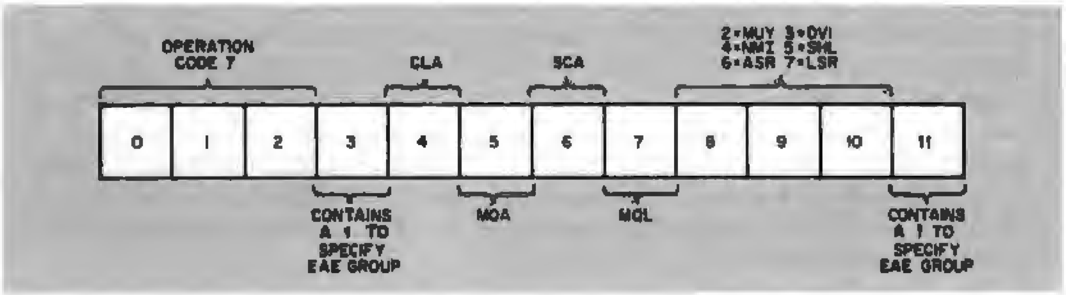


Figure 8 EAE Microinstruction Bit Assignments

MULTIPLY (MUY)

Octal Code: 7405

Event Time: 2

Indicators: OPR, FETCH, PAUSE

Execution Time: 9.0 to 21.0 microseconds

Operation: The number held in the MQ is multiplied by the number held in core memory location PC + 1 (or the next successive core memory location after the MUY command). At the conclusion of this command the link contains a 0, the most significant 12 bits of the product are contained in the AC and the least significant 12 bits of the product are contained in the MQ.

Symbol: Y × MQ = > AC, MQ
0 = > L

DIVIDE (DVI)

Octal Code: 7407

Event Time: 2

Indicators: OPR, FETCH, PAUSE

Execution Time: 36.5 microseconds or less

Operation: The 24-bit dividend held in the AC (most significant 12 bits) and the MQ (least significant 12 bits) is divided by the divisor held in core memory location PC + 1 (or the next successive core memory location following the DVI command). At the conclusion of this command the quotient is held in the MQ, the remainder is in the AC, and the L contains a 0. If the L contains a 1, divide overflow occurred so the operation was concluded after the first cycle of the division.

Symbol: AC, MQ $\div$ Y => MQ

NORMALIZE (NMI)

Octal Code: 7411

Event Time: 2

Indicators: OPR, FETCH, PAUSE

Execution Time: 1.5 microseconds + 0.5 microsecond for each shift

Operation: This instruction is used as part of the conversion of a binary number to a fraction and an exponent for use in floating-point arithmetic. The combined content of the AC and the MQ is shifted left by this one command until the content of AC0 is not equal to the content of AC1, or until 6000 0000 is contained in the combined AC and MQ, to form the fraction. Zeros are shifted into vacated MQ11 positions for each shift. At the conclusion of this operation, the step counter contains a number equal to the number of shifts performed, which can be loaded into the AC by an SCA command to form the exponent. The content of L is lost. Both positive and negative two's complement numbers can be normalized.

Symbol: ACj => ACj − 1
ACO => L
MQ0 => AC11
MQj = MQj − 1
0 => MQ11 until AC0 $\neq$ AC1 or until AC MQ = 6000 0000

SHIFT ARITHMETIC LEFT (SHL)

Octal Code: 7413

Event Time: 2

Indicators: OPR, FETCH, PAUSE

Execution Time: 3.0 microseconds + 0.5 microsecond for each shift

Operation: This instruction is used for scaling by shifting the combined content of the AC and MQ to the left one position more than the number of positions indicated by the content of core memory at address PC + 1 (or the next successive core memory location following the SHL command). During the shifting, zeros are shifted into vacated MQ11 positions. The L, AC, and MQ are treated as one long register during this operation. Bits shifted out of AC0 enter the L, and bits shifted out of the L are lost.

Symbol: Shift Y + 1 positions as follows:
ACj => ACj − 1
AC0 => L
MQ0 => AC11
MQj => MQj −1
0 => MQ11

ARITHMETIC SHIFT RIGHT (ASR)

Octal Code: 7415

Event Time: 2

Indicators: OPR, FETCH, PAUSE

Execution Time: 3.0 microseconds + 0.5 microsecond for each shift.

Operation: This instruction is used for scaling and treats the AC and MQ as one long register. The combined content of the AC and the MQ is shifted right one position more than the number contained in memory location PC + 1 (or the next successive core memory location following the ASR command). The sign bit, contained in AC0, enters vacated positions, the sign bit is preserved in the link, information shifted out of MQ11 is lost, and the L is set to correspond to the sign bit during this operation.

Symbol: Shift Y + 1 positions as follows:
AC0 = > L
AC0 = > AC0
ACj = > ACj + 1
AC11 = > MQ0
MQj = > MQj + 1

LOGICAL SHIFT RIGHT (LSR)

Octal Code: 7417

Event Time: 2

Indicators: OPR, FETCH, PAUSE

Execution Time: 3.0 microseconds + 0.5 microsecond for each shift.

Operation: This instruction is used for scaling and treats the AC and MQ as one long register. The combined content of the AC and MQ is shifted right one position more than the number contained in memory location PC + 1 (or the next successive core memory location following the LSR command). This command is similar to the ASR command except that zeros enter vacated positions instead of the sign bit entering these locations. Information shifted out of MQ11 is lost and the L is cleared during this operation.

Symbol: Shift Y + 1 positions as follows:

0 = > L

0 => AC0

ACj => ACj + 1

AC11 => MQ0

MQj => MQj + 1

LOAD MULTIPLIER QUOTIENT (MQL)

Octal Code: 7421

Event Time: 2

Indicators: OPR, FETCH

Execution Time: 1.5 microseconds

Operation: This command clears the MQ, loads the content of the AC into the MQ, then clears the AC. This operation is essential to initializing any multiply or divide routine and can be combined with a MUY or DVI command to perform the operation just prior to executing a multiplication or a division using a 12-bit dividend.

Symbol: 0 => MQ
AC => MQ
0 => AC

STEP COUNTER LOAD INTO ACCUMULATOR (SCA)

Octal Code: 7441

Event Time: 2

Indicators: OPR, FETCH

Execution Time: 1.5 microseconds

Operation: The content of the step counter is transferred into the AC. This command is used following an NMI command to establish the exponent of a normalized number to be used in floating point arithmetic. The AC should be cleared prior to issuing this command or the CLA command can be combined with the SCA to clear the AC then effect the transfer.

Symbol: SC V AC => AC

MULTIPLIER QUOTIENT LOAD INTO ACCUMULATOR (MQA)

Octal Code: 7501

Event Time: 2

Indicators: OPR, FETCH

Execution Time: 1.5 microseconds

Operation: The content of the MQ is transferred into the AC. This command is given to load the 12 least significant bits of the product into the AC following a multiplication or to load the quotient into the AC following a division. The AC should be cleared prior to issuing this command or the CLA command can be combined with the MQA to clear the AC then effect the transfer.

Symbol: MQ V AC => AC

CLEAR ACCUMULATOR (CLA)

Octal Code: 7601

Event Time: 1

Indicators: OPR, FETCH

Execution Time: 1.5 microseconds

Operation: The AC is cleared during event time 1, allowing this command to be combined with the other EAE commands that load the AC during event time 2 (such as SCA and MQA).

Symbol: 0 => AC

Programming

MULTIPLICATION

Multiplication is performed as follows:

1. Load the AC with the multiplier using the TAD instruction.
2. Transfer the content of the AC into the MQ using the MQL command.
3. Give the MUL command.

Note that steps 2 and 3 can be combined into one instruction.

The content of the MQ is then multiplied by the content of the next succesive core memory address (PC + 1). At the conclusion of the multiplication the most significant 12 bits of the product are held in the AC and the least significant 12 bits are held in the MQ. This operation takes a maximum of 21.0 microseconds, at the end of this time the next instruction is executed.

The following multiplication program examples indicate the operation of the Type 182 option in closed subroutines(routines which are incorporated into larger routines and are not written in a form which allows them to be called as a normal mathematical subroutine).

Multiplication of 12-Bit Unsigned Numbers

Enter with a 12-bit multiplicand in AC and a 12-bit multiplier in core memory. Exit with high order half of product in a core memory location labeled HIGH, and with low order half of product in the AC. Program time is from 13.5 to 25.5 microseconds.

```
MQL MUY         /LOAD MQ WITH MULTIPLICAND, INITIATE
                /MULTIPLICATION
MLTPLR          /MULTIPLIER
DCA HIGH        /STORE HIGH ORDER PRODUCT
MQA             /LOAD AC WITH LOW ORDER PRODUCT
```

Multiplication of 12-Bit Signed Numbers, 24-Bit Signed Product

Enter with a 12-bit multiplicand in AC and a 12-bit multiplier in core memory. Exit with signed 24-bit product in core memory locations designated HIGH and LOW. Program time is from 40.5 to 66.0 microseconds.

```
CLL
SPA                        /MULTIPLICAND POSITIVE?
CMA CML IAC                /NO. FORM TWO'S COMPLEMENT
MQL                        /LOAD MULTIPLICAND INTO MQ
TAD      MLTPLR
SPA                        /MULTIPLIER POSITIVE?
CMA CML IAC                /NO. FORM TWO'S COMPLEMENT
DCA      MLTPLR
RAL
```

```
          DCA      SIGN        /SAVE LINK AS SIGN INDICATOR
          MUY                  /MULTIPLY
MLTPLR,   0                    /MULTIPLIER
          DCA      HIGH
          TAD      SIGN
          RAR                  /LOAD LINK WITH SIGN INDICATOR
          MQA
          SNL                  /IS PRODUCT NEGATIVE?
          JMP      LAST        /NO
          CLL CMA  IAC         /YES
          DCA      LOW
          TAD      HIGH
          CMA
          SZL
          IAC
          DCA      HIGH
          SKP
LAST,     DCA      LOW
          .
          .
          .
```

DIVISION

Division is performed as follows:

1. Load the 12 least significant bits of the dividend into the AC using the TAD instruction, then transfer the content of the AC into the MQ using the MQL command.
2. Load the 12 most significant bit of the dividend into the AC.
3. Give the DVI command.

The 24-bit dividend contained in the AC and MQ is then divided by the 12-bit divisor contained in the next successive core memory address (PC + 1). This operation takes a maximum of 36.5 microseconds and is concluded with a 12-bit quotient held in the MQ, the 12-bit remainder in the AC, and the link holding a 0 if divide overflow did not occur. To prevent divide overflow, the divisor in the core memory must be greater than the 12-bits of the dividend held in the AC. When divide overflow occurs, the link is set and the division is concluded after only one cycle. Therefore the instruction following the divisor in core memory should be an SZL microinstruction to test for overflow. The instruction following the SZL can be a jump to a subroutine that services the overflow. This subroutine can cause the program to type out an error indication, rescale the divisor or the dividend, or perform other mathematical corrections and repeat the divide routine.

The following division program examples indicate the operation of the Type 182 option in closed subroutines.

Division of 12-Bit Unsigned Numbers

Enter with a 12-bit unsigned dividend in the AC and a 12-bit unsigned divisor in core memory. Exit with remainder in core memory location labeled REMAIN and with the quotient in the AC. Program time is a maximum of 44.0 microseconds.

```
CLL
MQL  DVI                /LOAD MQ, INITIATE DIVISION
DIVSOR                  /DIVISOR
SZL                     /OVERFLOW?
JMP                     /YES, EXIT
DCA REMAIN
MQL                     /LOAD AC WITH QUOTIENT
```

Division of a 12-Bit Signed Numbers

Enter with a 12-bit signed dividend in the AC and a 12-bit signed divisor in core memory. Exit with unsigned remainder in core memory location REMAIN and a 12-bit signed quotient in the AC. Program time is a maximum of 65.0 microseconds.

```
CLL
SPA                     /DIVIDEND POSITIVE?
CMA CML IAC             /NO
MQL
TAD .+11
SPA                     /DIVISOR POSITIVE?
CMA CML IAC             /NO
DCA .+6
SNL                     /QUOTIENT NEGATIVE?
CMA                     /NO
CLL
DCA SIGN                /SET SIGN INDICATOR
DVI
DIVSOR                  /DIVISOR
SZL                     /OVERFLOW
JMP                     /EXIT ON OVERFLOW
MQL
ISZ SIGN
CMA  IAC
```

CHAPTER 9
AUTOMATIC RESTART TYPE KR01

This prewired option protects an operating program in the event of failure of the source of computer primary power. If a power failure occurs, this option causes a program interrupt and enables continued operation for 1 millisecond, allowing the interrupt routine to detect the power low condition as initiator of the interrupt, and to store the content of active registers (AC, L, MQ, etc.) and the program count in known core memory locations. When power is restored, the power low flag clears and a routine beginning in address 0000 starts automatically. This routine restores the content of the active registers and program counter to the conditions that existed when the interrupt occurred, then continues the interrupted program.

The KR01 option consists of three logic circuits:

A power interrupt circuit monitors the status signal of the computer power supply, and sets a power low flag when power is interrupted (due to a power failure or due to the operation of the POWER lock on the operator console). This flag causes a program interrupt when an interruption in computer power is detected.

A restart circuit assures that when a power interrupt occurs the logic circuits of the computer continue operation for 1 millisecond to allow a program subroutine to store the content of the active registers; maintains the inoperative condition of the computer during periods of power fluctuation; and clears the power low flag and restarts the program when power conditions are suitable for computer operation. A manual RESTART switch on the processor marginal-check frame enables or disables the automatic restart operation. With this switch in the ON (down) position, the option clears the program counter immediately and produces a signal to simulate operation of the START key on the operator console 200 milliseconds after power conditions are satisfactory. The PC is cleared so that operation restarts by executing the instruction in address 0000. This instruction is a JMP to the starting address of the subroutine which restores the content of the active registers and the program counter to the conditions that existed prior to the power low interrupt. The 200-millisecond delay assures that slow mechanical devices, such as Teletype equipment, have come to a complete stop before the program is resumed. Simulation of the manual START function causes the processor to generate a Power Clear pulse to clear internal controls and I/O device registers. With the RESTART switch in the OFF (up) position, the power low flag is cleared but the program must be started manually, possibly after resetting peripheral equipment or by starting the interrupted program from the beginning.

A skip circuit provides programmed sensing of the condition of the power low flag by adding the following instruction to the computer repertoire:

SKIP ON POWER LOW (SPL)

Octal Code: 6102

Event Time: 2

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The content of the power low flag is sampled, and if it contains a 1 (indicating a power failure has been detected) the content of the PC is incremented by one so the next sequential instruction is skipped.

Symbol: If Power Low flag = 1, then PC +1 => PC

Since the time that operation of the computer can be extended after a power failure is limited to 1 millisecond, the condition of the power low flag should be the first status check made by the program interrupt subroutine. The beginning of the program interrupt subroutine, containing the SPL microinstruction and the power fail program sequence can be executed in 25.5 microseconds on a basic PDP-8 with an extended arithmetic element. The power fail program sequence stores the content of the active registers and program count in designated core memory locations, then relocates the calling instruction of the power restore subroutine to address 0000, as follows:

```
Address     Instruction     Remarks

0000            —           /STORAGE FOR PC AFTER PROGRAM INTERRUPT
0001        JMP FLAGS       /INSTRUCTION EXECUTED AFTER PROGRAM
                            /INTERRUPT
FLAGS,      SPL             /SKIP IF POWER LOW FLAG = 1
            JMP OTHER       /INTERRUPT NOT CAUSED BY POWER LOW,
                            /CHECK OTHER FLAGS
            DCA AC          /INTERRUPT WAS CAUSED BY POWER LOW,
                            /SAVE AC
            RAR             /GET LINK
            DCA LINK        /SAVE LINK
            MQA             /GET MQ
            DCA MQ          /SAVE MQ
            TAD 0000        /GET PC
            DCA PC          /SAVE PC
            TAD RESTRT      /GET RESTART LOCATION
            DCA 0000        /DEPOSIT RESTART LOCATION IN 0000
            HLT
RESTRT      JMP ABCD        /ABCD IS LOCATION OF RESTART ROUTINE
```

Automatic program restart begins by executing the instruction stored in address 0000 by the power fail routine. The power restore subroutine restores the content of the active registers, enables the program interrupt facility, and continues the interrupted program from the point at which it was interrupted, as follows:

```
Address     Instruction     Remarks

0000        JMP ABCD
ABCD,       TAD MQ          /GET MQ
            MQL             /RESTORE MQ
            TAD LINK        /GET LINK
            CLL RAL         /RESTORE LINK
            TAD AC          /RESTORE AC
            ION             /TURN ON INTERRUPT
            JMP I PC        /RETURN TO INTERRUPTED PROGRAM
```

SECTION B

INPUT-OUTPUT EQUIPMENT

CHAPTER 1

BASIC IOT PROGRAMMING

Three basic modes of programming can be used to transfer information or control signals between input/output devices and the PDP-8. These modes are programmed data transfers, program interrupt, and data break transfers. To understand the use which can be made of each of these control modes assume that the PDP-8 is connected as the primary control element in a system which contains several furnaces or kilns, continuous-belt conveyors which transport products through the furnaces, a visual monitor panel, and a drum memory. Assume that each furnace or zone of an oven contains a controller which has a digital temperature readout, and overtemperature and undertemperature alarms which can be set by a digital readin. Next, assume that the conveyor for each oven is automatically loaded with blocks of products spaced at given distances on the conveyor and that the conveyor has a normal speed, a high emergency speed, and a stop control. Finally, assume that the visual monitor and drum memory periodically cause data breaks to transfer temperature and controls status information from core memory into visual display devices and drum storage. The following explanation of the use of each of the three control modes in programming this hypothetical system can easily be translated into examples of data processing or other control system programs.

Programmed Data Transfers

All peripheral equipment transfers information to or from the computer by programmed instructions. This means of communication can be used as the sole method of transferring information or can be used to initialize equipment using the program interrupt or data break facilities. This mode of operation utilizes the IOT instruction which is divided into three parts. Bits 0, 1, and 2 contain an operation code of 6 to specify the IOT microinstruction. Bits 3 through 8 serve as a device selection code which is transmitted to all peripheral equipment and which activates only the equipment designated by a specific code number contained within these bits. Bits 9, 10, and 11 control the IOP generator within the processor and enable or disable generation of IOP1, IOP2, and IOP4 pulses during each IOT instruction. A device selector within each peripheral equipment monitors the device selection lines and enables pulse amplifiers when its assigned select code has been detected within bits 3 through 8 of an IOT instruction. When enabled in this manner the pulse amplifiers produce positive or negative IOT pulses when triggered by an associated IOP pulse. The IOT pulses,in turn, perform data transfers to or from the computer or perform control functions within the peripheral equipment.

Each peripheral equipment can contain one or more device selector. A device selector can consist of a Type 4605 Pulse Amplifier system module, a Type W103 Device Selector FLIP CHIP module, or can be constructed of three FLIP-CHIP modules such as the Type R603 Pulse Amplifier, R111 Diode Gate, and R002 Diode Cluster. Regardless of its circuit components, a device selector consists of a 6-input negative diode NAND gate which is enabled only when the select code of the specified device is contained in the instruction. The output from this NAND gate enables gating circuits at the input of each of three pulse amplifiers which are triggered by the IOP1, IOP2, or IOP4 pulse.

SENSE FOR DEVICE READY

In preparation for a normal data transfer the computer program normally checks the ready status of the transmitting or receiving device by means of a skip instruction. This skip instruction can skip on either the ready or not ready status of the device, depending upon the internal operations of that device. In our imaginary PDP-8 control system, assume that the control for each furnace contains an "up to temperature flag" and an associated instruction which provides a Skip pulse to the program counter control element when the operating temperature range has been attained. Therefore the programming can contain this skip on-up-to-temperature instruction followed by a JMP instruction which transfers program control back in a loop so that it repeatedly checks the up to temperature flag until the acceptable temperature range has been attained. Then the next step in the program proceeds with an operation, such as initiate loading of the conveyor and movement of the conveyor at the normal speed. In some instances the program will proceed to assemble data so that it can be transferred into or out of the computer.

ASSEMBLE DATA

The ready control described under the previous heading can indicate that a device is operating in a known control mode or that data is ready for transmission. In our example the former condition is true so that data assembly follows the sense for ready operation. To assemble data the program issues commands in one or more IOT instructions to clear and load a buffer with data to be transferred from the accumulator, or to clear and load the accumulator for data to be transferred to a peripheral equipment buffer. In the imaginary control system described earlier, data assembly could consist of an instruction which generates an IOT1 pulse to clear an information buffer, generates an IOT2 pulse to load a temperature setting into the six least significant bits of this buffer and loads the status of the undertemperature and overtemperature flags into the two most significant bits of this buffer.

EFFECT A TRANSFER

Actual data transfer between peripheral equipment and the PDP-8 accumulator is performed by an IOT instruction. Usually a transfer into the PDP-8 is performed by an instruction in which an IOT1 pulse is generated to clear the accumulator and a later IOT pulse is generated to strobe the content of an external buffer into the accumulator. In transfers from the accumulator to an external buffer the static accumulator data lines are used to condition gates at the input of a static buffer, then an IOT pulse is generated to actuate the gates and transfer the static conditions into the buffer. In the case of our hypothetical process control system an IOT instruction can be developed to clear the accumulator, then read the content of the data buffer into the accumulator.

The entire sequence of operations described under Programmed Data Transfers is summarized in the following program example, using legitimate memory reference and operate instructions, and using artificial IOT instructions.

```
                   CLA
                   TAD            /LOAD OVERTEMPERATURE VALUE INTO AC
                   LOT            /CLEAR AND LOAD OVERTEMPERATURE SET
Initialize                        /POINT
                   CLA
                   TAD            /LOAD UNDERTEMPERATURE VALUE INTO AC
                   LUT            /CLEAR AND LOAD UNDERTEMPERATURE SET
                                  /POINT
                   TON            /TURN ON FURNACE
Sense for          SUT            /SKIP ON DEVICE #1 UP TO TEMP.
device ready       JMP -1         /LOOP IF NOT UP TO TEMP
(one or more       IPC            /INITIATE PRODUCT TRANSPORT TO CONVEYOR
devices)                          /AND CONVEYOR OPERATION AT NORMAL
                                  /SPEED
Assemble           LDB            /CLEAR AND LOAD DEVICE BUFFER WITH DATA
Data                              /AND STATUS

Effect Transfer    RDB            /CLEAR AC AND READ DATA BUFFER INTO AC
                   RAL            /ROTATE OVERTEMP. CONTROL STATUS FROM
                                  /ACO INTO L
                   SNL OR SZL     /SENSE OVERTEMP. CONTROL
                   JMS            /JUMP TO OVERTEMP. SUBROUTINE WHICH
                                  /TURNS OFF FURNACE, STOPS PRODUCT
                                  /LOADING INTO CONVEYOR, ADVANCES
                                  /CONVEYOR AT EMERGENCY SPEED TO
                                  /REMOVE PRODUCTS FROM FURNACE, THEN
                                  /STOPS CONVEYOR
Process            RAL            /ROTATE UNDERTEMP. CONTROL STATUS
Data                              /INTO L
                   SNL OR SZL     /SENSE UNDERTEMP. CONTROL
                   JMS            /JUMP TO SUBROUTINE WHICH STOPS
                                  /CONVEYOR MOTION AND JUMPS BACK TO
                                  /THE BEGINNING OF THE MAIN ROUTINE TO
                                  /WAIT FOR UP TO TEMP.
                   RTR            /RELOCATE DATA
                   DCA            /STORE TEMPERATURE DATA
```

Program Interrupt

Urgent requirements for programmed data transfer or programmed control functions by peripheral equipment can be satisfied through use of the program interrupt facility. This facility allows an external device to cause the main computer program to be interrupted and a subroutine to be initiated to service the interrupting device. Use of this facility simplifies basic programming by eliminating the need for checking alarm conditions and allows the alarm conditions themselves to activate corrective operations, rather than waiting for cyclic checking by the main routine.

When the program interrupt feature is used address 0001 is automatically specified as the first address of a subroutine that checks and services the interrupt condition. Usually the instruction stored in this address is a jump to a location where the subroutine really begins. As designated in Chapter 4 of Section A of this handbook the program interrupt subroutine must locate the device causing the interruption, take some corrective action, restore or enable the program interrupt synchronization element of the computer by execution of an ION instruction, and return program control to the main program at the point at which the interrupt occurred. If only one device is connected to the program interrupt facility no checking is required to locate the interrupting device. However, if many devices are connected to the program interrupt bus (as normally is the case) the interrupt subroutine must perform repeated skip instructions to test the condition of the various devices. This testing should be accomplished by a program-established priority system so that the devices which need servicing in the least amount of time or which require servicing most frequently are checked first, depending upon the application.

In our hypothetical process control system if the overtemperature alarms for each furnace are connected to the program interrupt bus the interrupt subroutine can skip on the condition of the overtemperature flag to a portion of the routine which de-energizes the appropriate furnace and performs any required operations in the control of the conveyor for that oven (such as inhibiting additional loading of the conveyor, removing products from the oven by high speed advance of the conveyor, or by initiating other shut down procedures). The sequence of testing for the furnace causing the overtemperature alarm can proceed from the first furnace to the last furnace if it is most important to prevent unfired products from entering the defective oven, can be performed from the last to the first furnace if over firing cannot occur, or can be performed in the sequence determined by the various furnace temperatures.

Data Break Transfers

Peripheral equipment requiring rapid or periodic data transfers to or from the core memory of the PDP-8 can be connected to use the data break facilities. Where more than one such input/output device is used in the computer system they must be connected to the data break facility through a multiplexer switch (such as the Type DM01 option) which assigns a pre-established priority to each device. This facility allows the main computer program to be suspended for a time while individual or block data transfers occur between the memory buffer register and the peripheral equipment. In some cases these transfers occur to or from blocks of sequential core memory addresses or occur individually, but each transfer occurs at a device-specified address. Where individual transfers occur sporadically in time, each transfer is interleaved with portions of the main program. Following each data break the transferring equipment can issue a program interrupt to enter a sub-routine which reinitializes the device with the transfer direction and address information required for the succeeding break.

In our hypothetical process control system it can be assumed that a real-time clock in the visual display produces a data break periodically to transfer temperature and control status information from specific core memory addresses into visual readout or display devices. The drum memory can be assumed to initiate data breaks to read and store data from core memory so that it can be analyzed for quality control evaluation after running the heat. The frequency of the data break requests performed in this manner must be related to the program timing required to read this information into core memory from the temperature controlling devices, assuming the maximum amount of time for any program interrupts and data breaks during the program.

The data break is accomplished by supplying Break Request and Cycle Select signals to the major state generator, supplying a Transfer Direction signal to the memory buffer register control element, supplying an address to the memory address register, and (for a transfer into the PDP-8) supplying a data word to the memory buffer register. When a single-cycle data break is requested in this manner, at the conclusion of the current instruction the Break state is entered to transfer a data word. When the direction of transfer is into the PDP-8, the data word must be supplied the memory buffer register within the first half of the Break cycle. When the direction of transfer is out of the PDP-8, the data word is available for strobing by the device approximately 350 microseconds after entry into the Break state. When a three-cycle data break is requested, at the conclusion of the current instruction the Word Count state is entered to commence the data break; but the transfer does not occur until the the third (Break) cycle of the data break.

Timing and data requirements of the input/output device determine the number of consecutive data breaks that occur before control of computer operations is returned to the program. One Break cycle is required for each data word transfer, so very fast devices which require blocks of information can maintain the Break Request signal to perform consecutive data breaks until a device-specified block length transfer is completed. Devices using consecutive data breaks must synchronize their operations to the speed of the computer to transfer words with core memory at the single-cycle rate of 666 kc (one word every 1.5 microseconds) or at the three-cycle rate of 222 kc (one word every 4.5 microseconds). Normal clock pulses used with the computer are available to synchronize the operation of peripheral equipment with the PDP-8 during a data break. Slower equipment using the data break facilities must initiate a separate data break for each word transfer.

CHAPTER 2

TELETYPE AND CONTROL

Teletype Model 33 ASR

The standard Teletype Model 33 ASR (automatic send-receive) can be used to type in or print out information at a rate of up to ten characters per second, or to read in or punch out perforated paper tape at a ten characters per second rate. Signals transferred between the 33 ASR and the control logic are standard serial, 11 unit code Teletype signals. The signals consist of marks and spaces which correspond to idle and bias current in the Teletype,and to zeros and ones in the control and computer. The start mark and subsequent eight character bits are one unit of time duration and are followed by the stop mark which is two units.

The 8-bit code used by the Model 33 ASR Teletype unit is the American Standard Code for Information Interchange (ASCII) modified. To convert the ASCII code to Teletype code add 200 octal (ASCII + 200_8 = Teletype). This code is read in the reverse of the normal octal form used in the PDP-8 since bits are numbered from right to left, from 1 through 8, with bit 1 having the least significance. Therefore perforated tape is read:

$$8 \quad 7 \quad \underbrace{6 \quad 5 \quad 4}_{\substack{\text{Most Significant} \\ \text{Octal Bit}}} \quad S \quad \underbrace{3 \quad 2 \quad 1}_{\substack{\text{Least Significant} \\ \text{Octal Bit}}}$$

The Model 33 ASR set can generate all assigned codes except 340 through 374 and 376. Generally codes 207, 212, 215, 240 through 337, and 377 are sufficient for Teletype operation. The Model 33 ASR set can detect all characters, but does not interpret all of the codes that it can generate as commands. The standard number of characters printed per line is 72. The sequence for proceeding to the next line is a carriage return followed by a line feed (as opposed to a line feed followed by a carriage return). Appendix 2 lists the character code for the Teletype. Punched tape format is as follows:

	Tape Channel			
	87	654	S	321
Binary Code (Punch = 1)	10	110		100
Octal Code	2	6		4

Teletype Control

Serial information read or written by the Teletype unit is assembled or disassembled by the control for parallel transfer to the accumulator of the processor. The control also provides the program flags which cause a program interrupt or an instruction skip based upon the availability of the Teletype and the processor as a function of the program.

In all programmed operation, the Teletype unit and control are considered as a Teletype in (TTI) as a source of input intelligence from the keyboard or the perforated-tape reader and is considered a Teletype out (TTO) for computer output information to be printed and/or punched on tape. Therefore, two device selectors are used; the select code of 03 initiates operations associated with the keyboard/reader, and the device selector, assigned the select code of 04, performs operations associated with the teleprinter/punch. Parallel input and output functions are performed by corresponding IOT pulses produced by the two device selectors. Pulses produced by IOP1 pulse trigger skip gates; pulses produced by the IOP2 pulse clear the control flags and/or the accumulator; and pulses produced by the IOP4 pulse initiate data transfers to or from the control.

Keyboard/Reader

The keyboard and tape reader control contains an 8-bit buffer (TTI) which assembles and holds the code for the last character struck on the keyboard or read from the tape. Teletype characters from the keyboard/reader are received serially by the 8-bit shift register TTI. The character code of a Teletype character is loaded into the TTI so that spaces correspond with binary zeros and marks correspond to binary ones. Upon program command the content of the TTI is transferred in parallel to the accumulator. When a Teletype character starts to enter the TTI the control de-energizes a relay in the Teletype unit to release the tape feed latch. When released, the latch mechanism stops tape motion only when a complete character has been sensed, and before sensing of the next character is started. A keyboard flag is set to one, and causes a program interrupt when an 8-bit computer character has been assembled in the TTI from a Teletype character. The program senses the condition of this flag with a KSF microinstruction and issues a KRB microinstruction which clears the AC, clears the keyboard flag, transfers the content of the TTI into the AC, and enables advance of the tape feed mechanism. Instructions for use in supplying data to the computer from the Teletype are:

SKIP ON KEYBOARD FLAG (KSF)

Octal Code: 6031

Event Time: 1

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The keyboard flag is sensed, and if it contains a binary 1 the content of the PC is incremented by one so that the next sequential instruction is skipped.

Symbol: If Keyboard Flag = 1, then PC +1 => PC

CLEAR KEYBOARD FLAG (KCC)

Octal Code: 6032

Event Time: 2

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: Both the AC and the keyboard flag are cleared in preparation for transferring a Teletype character into the AC.

Symbol: 0 => AC
0 => Keyboard Flag

READ KEYBOARD BUFFER STATIC (KRS)

Octal Code: 6034

Event Time: 3

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The content of the TTI is transferred into bits 4 through 11 of the AC. This is a static command in that neither the AC nor the keyboard flag is cleared.

Symbol: TTI V AC 4-11 => AC 4-11

READ KEYBOARD BUFFER DYNAMIC (KRB)

Octal Code: 6036

Event Time: 2, 3

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The AC and the keyboard flag are both cleared, then the content of the TTI is transferred into bits 4 through 11 of the AC.

Symbol: 0 => AC, Keyboard Flag
TTI V AC 4-11 => AC 4-11

A program sequence loop to read input information into the computer from the Teletype keyboard or tape reader can be written as follows:

```
LOOK,    KSF              /SKIP WHEN TTI IS FULL
         JMP  LOOK
         KRB              /READ TTI INTO AC
```

Teleprinter/Punch

Eight-bit computer characters from the accumulator are loaded in parallel into the 8-bit flip-flop shift register TTO for transmission to the Teletype unit. The control generates the start space, then shifts the eight character bits into the printer selector magnets of the Teletype unit, and then produces the stop mark. This transfer of information from the TTO into the Teletype unit is accomplished in a serial manner at the normal Teletype rate. A teleprinter flag in the teleprinter control is set when the last bit of the Teletype code has been sent to the teleprinter/punch, indicating that the TTO is ready to receive a new character from the AC. The flag is connected to both the program interrupt synchronization element and the PC control (instruction skip) element. Upon detecting the set (binary one) condition of the flag by means of the TSF microinstruction the program issues a TLS microinstruction which clears the flag and loads a new computer character into the TTO.

The instruction list for printing or punching is:

SKIP ON TELEPRINTER FLAG (TSF)

Octal Code: 6041

Event Time: 1

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The teleprinter flag is sensed, and if it contains a binary 1 the content of the PC is incremented by one so that the next sequential instruction is skipped.

Symbol: If Teleprinter Flag = 1, then PC +1 => PC

CLEAR TELEPRINTER FLAG (TCF)

Octal Code: 6042

Event Time: 2

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The teleprinter flag is cleared to 0.

Symbol: 0 => Teleprinter Flag

LOAD TELEPRINTER AND PRINT (TPC)

Octal Code: 6044

Event Time: 3

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The TTO is loaded from the content of bits 4 through 11 of the AC; then the Teletype character just loaded is selected, and punched and/or printed.

Symbol: AC 4-11 => TTO

LOAD TELEPRINTER SEQUENCE (TLS)

Octal Code: 6046

Event Time: 2, 3

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The teleprinter flag is cleared; then a Teletype character code is transferred from the content of AC 4-11 into the TTO, the character is selected and punched and/or printed.

Symbol: 0 => Teleprinter Flag
AC 4-11 => TTO

A program sequence loop to print and/or punch a character when the TTO is free can be written as follows:

```
        FREE,          TSF              /SKIP WHEN FREE
                       JMP FREE
                       TLS              /LOAD TTO,  PRINT OR PUNCH
```

Teletype System Type LT08

The Teletype facility of the basic computer can be expanded to accommodate several Model 33 or Model 35 Automatic Send Receive or Keyboard Send Receive units by addition of the Type LT08 option. Each Teletype line added to the PDP-8 system contains logic elements that are functionally identical to those of the basic Teletype control. Therefore, instructions and programming for each line of an LT08 equipment are similar to those described previously for the basic Teletype unit. The following device select codes have been assigned for five lines of LT08 equipment:

Line Unit	Select Codes
1	40 and 41
2	42 and 43
3	44 and 45
4	46 and 47
5	11 and 12

Instruction mnemonics for Teletype equipment in the LT08 system are not recognized by the program assembler (PAL III) and must be defined by the programmer. Mnemonic codes can be defined by the mnemonic code of the comparable basic Teletype microinstruction, suffixed with "LT" and the line number. For example, the following instructions can be defined for line 3:

Mnemonic	Octal	Operation
TSFLT3	6441	Skip if teleprinter 3 flag is a 1.
TCPLT3	6442	Clear teleprinter 3 flag.
TPCLT3	6444	Load teleprinter 3 buffer (TTO3) from the content of AC4-11 and print and/or punch the character.
TLSLT3	6446	Load TTO3 from the content of AC4-11, clear teleprinter 3 flag, and print and/or punch the character.
KSFLT3	6451	Skip if keyboard 3 flag is a 1.
KCCLT3	6452	Clear AC and clear keyboard 3 flag.
KRSLT3	6454	Read keyboard 3 buffer (TTI3) static. The content of TTI3 is loaded into AC4-11 by an OR transfer.
KRBLT3	6456	Clear the AC, clear keyboard 3 flag, and read the content of TTI3 into AC4-11.

CHAPTER 3

HIGH SPEED PERFORATED TAPE READER AND CONTROL TYPE 750C

This device senses 8-hole perforated paper or Mylar tape photoelectrically at 300 characters per second. The reader control requests reader movement, transfers data from the reader into the reader buffer (RB), and signals the computer when incoming data is present. Reader tape movement is started by a reader control request to simultaneously release the brake and engage the clutch. The 8-bit reader buffer sets the reader flag to 1 when it has been filled from the reader and transfers data into bits 4 through 11 of the accumulator under program control. The reader flag is connected to the computer program interrupt and instruction skip facilities, and is cleared by IOT pulses. Tape format is as described for the Teletype unit. Computer instructions for the reader are:

SKIP ON READER FLAG (RSF)

Octal Code: 6011

Event Time: 1

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The reader flag is sensed, and if it contains a binary 1 the content of the PC is incremented by one so that the next sequential instruction is skipped.

Symbol: If Reader Flag = 1, then PC +1 => PC

READ READER BUFFER (RRB)

Octal Code: 6012

Event Time: 2

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The content of the reader buffer is transferred into bits 4 through 11 of the AC and the reader flag is cleared. This command does not clear the AC.

Symbol: RB V AC 4-11 => AC 4-11
0 => Reader Flag

READER FETCH CHARACTER (RFC)

Octal Code: 6014

Event Time: 3

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The reader flag and the reader buffer are both cleared, one character is loaded into the reader buffer from tape, and the reader flag is set when this operation is completed.

Symbol: 0 => Reader Flag, RB
Tape Data => RB
1 => Reader Flag when done

A program sequence loop to read a character from perforated tape can be written as follows:

```
                    RFC             /FETCH CHARACTER FROM TAPE
        LOOK,       RSF             /SKIP WHEN RB FULL
                    JMP LOOK
                    CLA
                    RRB             /LOAD AC FROM RB
```

CHAPTER 4

HIGH SPEED TAPE PUNCH CONTROL TYPE 75E

This option consists of a Teletype BRPE paper tape punch that perforates 8-hole tape at a rate of 63.3 characters per second. Information to be punched on a line of tape is loaded in an 8-bit punch buffer (PB) from AC bits 4 through 11. The punch flag becomes a 1 at the completion of punching action, signaling that new information may be transferred into the punch buffer, and punching initiated. The punch flag is connected to the computer program interrupt and instruction skip facility. Tape format is as described in Chapter 2. The punch instructions are:

SKIP ON PUNCH FLAG (PSF)

Octal Code: 6021

Event Time: 1

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The punch flag is sensed, and if it contains a binary 1 the content of the PC is incremented by one so that the next sequential instruction is skipped.

Symbol: If Punch Flag = 1, then PC + 1 => PC

CLEAR PUNCH FLAG (PCF)

Octal Code: 6022

Event Time: 2

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: Both the punch flag and the punch buffer are cleared in preparation for receiving a new character from the computer.

Symbol: 0 => Punch Flag, PB

LOAD PUNCH BUFFER AND PUNCH CHARACTER (PPC)

Octal Code: 6024

Event Time: 3

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: An 8-bit character is transferred from bits 4 through 11 of the AC into the punch buffer and then this character is punched. This command does not clear the punch flag or the punch buffer.

Symbol: AC4-11 V PB => PB

LOAD PUNCH BUFFER SEQUENCE (PLS)

Octal Code: 6026

Event Time: 2, 3

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The punch flag and punch buffer are both cleared, the content of bits 4 through 11 of the AC is transferred into the punch buffer, the character in the PB is punched in tape, and the punch flag is set when the operation is completed.

Symbol: 0 => Punch Flag, PB
AC4-11 => PB
1 => Punch Flag when done

A program sequence loop to punch a character when the punch buffer is "free" can be written as follows:

```
FREE,     PSF          /SKIP WHEN FREE
          JMP FREE
          PLS          /LOAD PB FROM AC AND PUNCH
                       /CHARACTER
```

CHAPTER 5

ANALOG-TO-DIGITAL CONVERTER

TYPE 189

This converter operates in the conventional successive approximation manner, using the memory buffer register as a distributor shift register and using the accumulator as the digital buffer register. Converter operation is initiated by an IOT command that produces a Pause pulse (as most IOT commands do) and starts the conversion process. With the AC cleared, this process starts by asuming that the value of the analog input signal is at mid scale by setting a binary 1 into the most significant bit of the accumulator and producing a voltage equal to the center of the input range of the converter (ground to −10 volts). This voltage is produced by a digital-to-analog converter which operates as a function of a number contained in the accumulator. This voltage is then compared with the analog input signal and the result of the comparison is used to clear the most significant bit of the accumulator if the approximated voltage generated is of greater amplitude than the analog input signal. This process is then repeated by setting the next least significant bit of the accumulator to the 1 state, generating the analog signal according to the content of the accumulator, and comparing this signal with the analog input signal to clear the bit of the AC which was just set previously if the generated signal is greater than the input signal being measured. This process is repeated a number of times depending upon the prewired accuracy of the conversion. Each approximation reduces the error of the resultant binary number in the AC by approximately one half. The bit of the accumulator which is first set and then evaluated, is controlled by the memory buffer register. During the first approximation, a binary 1 is set into most significant bit of the MB and is shifted right one place at the conclusion of each approximation. The bit of the accumulator which is processed is determined by the location of this binary 1 in the MB. Sensing of the location of this binary 1 in the MB is also used to control the number of approximations performed, and hence determines the accuracy of the conversion. Since the conversion is started at the time the binary 1 is shifted in the MB, one conversion takes place after the sensing of the 1 in the MB which discontinues the conversion process. At the conclusion of the conversion a Restart pulse is produced by the converter which clears the MB and continues the normal computer program. At this time the digital equivalent of the analog input signal is contained in the accumulator as a 12-bit unsigned binary number. Insignificant magnitude bits can be rotated out of the AC by an instruction such as 7110 (RAR and CLL).

To save program running time, the converter should be adjusted to provide only the accuracy required by the program application. Maximum error of the converter is equal to the switching point error plus the quantization error. Maximum quantization error is equal to the binary value of the least significant bit. Switching point error and total conversion time are functions of the adjusted accuracy of the converter as indicated in Table 1.

TABLE 1 ANALOG-TO-DIGITAL CONVERTER TYPE 189 CHARACTERISTICS

Adjusted Bit Accuracy	Switching Point Error (in per cent)	Conversion Time per Bit (in microseconds)	Total Conversion Time (in microseconds)	Instruction Execution Time (in microseconds)
6	±1.6	1.0	6	7.6
7	±0.8	1.85	13	14.6
8	±0.4	2.5	20	21.6
9	±0.2	2.7	24	25.6
10	±0.1	2.7	27	28.6
11	±0.05	4.1	45	46.6
12	±0.025	4.6	55	56.6

The ADC is the only instruction associated with the Type 189 converter.

CONVERT ANALOG TO DIGITAL (ADC)

Octal Code: 6004

Event Time: Not applicable

Indicators: IOT, FETCH, PAUSE

Execution Time: This time is related to the adjusted converter accuracy as listed in Table 1.

Operation: The analog input signal is converted to an unsigned digital value which is held in the AC at the end of the conversion.

Symbol: None

CHAPTER 6

ANALOG-TO-DIGITAL CONVERTER TYPE 138E AND MULTIPLEXER CONTROL TYPE 139E

The Type 138E/139E General-Purpose Analog-to-Digital Converter and Multiplexer Control combines a versatile, multipurpose converter with a multiplexer to provide a fast, automatic, multichannel scanning and conversion capability. It is intended for use in systems in which computers sample and process analog data from sensors or other external signal sources at high rates. For example, analog data on each of 64 channels can be accepted and converted into 12-bit digital numbers 415 times per second.* Switching point accuracy in this instance is 99.975 per cent, with an additional quantization error of half the least significant bit (LSB). If less resolution and accuracy is required, all 64 channels can be scanned and the analog signals on them converted into 6-bit digital numbers 1,360 times each second.** Switching point accuracy in this case is 99.2 per cent, again with the additional quantization error of half the digital value of the LSB.

The Type 139 Multiplexer Control can include from 1 to 32 series A100 Multiplexer modules determined by the user. Each module addresses one of two channels for a maximum of 64 channels per Type 139E control. In the Individal Address mode, the Type 139 routes the data from any selected channel to the Type 138E converter input. In the Sequential Address mode, the multiplexer advances its channel address by one each time it receives an increment command, returning to channel zero after scanning the last channel. Sequenced operations can be short-cycled when the number of channels in use is less than the maximum available.

*Conversion rate=$[(35+2.5)\ (10^{-6})\ (64)]^{-1}=415$ cycles/sec
**Conversion rate=$[(9+2.5)\ (10^{-6})\ (64)]^{-1}=1360$ cycles/sec

TABLE 2 ANALOG-TO-DIGITAL CONVERTER TYPE 139E CHARACTERISTICS

Word Length (in bits)	Switching Point Error*** (in percent)	Total Conversion Time (in microseconds)	Conversion Rate (in kc)
6	±1.6	9.0	110.0
7	±0.8	10.5	95.0
8	±0.4	12.0	83.0
9	±0.2	13.5	74.0
10	±0.1	17.0	58.5
11	±0.05	25.0	40.0
12	±0.025	35.0	28.5

***±½ LSB for quantizing error.

The Type 138E is a successive approximation converter that measures a 0 to 10 volt analog input signal and provides a binary output indication of the amplitude of the input signal. Output indication accuracy is a function of the conversion time, and is determined by a switch on the front panel. Each of the seven positions of the rotary switch establishes an output word length, conversion accuracy, and conversion time for operation of the converter. Overall conversion error equals switching point error plus a quantization of ±½ the digital value of the LSB. Converter characteristics selected for each switch position are specified in Table 2.

CONVERTER SPECIFICATIONS

Monotonicity: Guaranteed for all settings

Aperture Time: Same as conversion time

Converter Recovery Time: None

Analog Input: 0 to −10 volts is standard. Bipolar or specific amplitude range input can be accommodated on special request. If a different voltage range is desired, it is recommended that an amplifier be used at the source, since this will also provide a low driving impedance and reduce the possibilities of noise pickup between the source and the converter.

Input Loading: ±1 microampere and 125 picofarads for the standard 0 to −10 volt input.

Digital Output: A signed 6- to 12-bit binary number in 2's complement notation. A 0 volt input yields a digital output number of 4000_8; a −5 volt input produces 0000_8; and a −10 volt input gives an output of 3777_8.

Controls: Binary readout indicators and a seven-position rotary switch for selecting converter characteristics are provided on the front panel.

The Type 139E Multiplexer Control is intended for use with the Type 138E or Type 189 analog-to-digital conversion systems in applications where the PDP-8 must process sampled analog data from multiple sources at high speeds. Under program control the multiplexer can select from 2 to 64 analog input signal channels for connection to the input of an analog-to-digital converter. Channel selection is provided by Type A100, A101, A102, or A103 Multiplex Switch FLIP CHIP modules. These module types each have slightly different timing, impedance, and power characteristics so that multiplexers can be built for wide differences in application by selecting the appropriate module type. Each module contains two independent, floating, transistor switches letting the user select any multiple of two channels to a maximum of 64. In the individual address mode, the Type 139E routes the analog data from any program-selected channel to the converter input. In the sequential address mode, the multiplexer advances the channel address by one each time it receives an incrementing command, returning to channel zero after scanning the last channel. Sequenced operations can be short-cycled when the number of channels in use is less than the maximum available.

A 6-bit channel address register (CAR) specifies a channel number from 0-77_8. A channel address may be chosen in one of two ways. It can be specified by the content of bits 6-11 of the AC or by incrementing the content of the CAR.

MULTIPLEXER SPECIFICATIONS

Indicators: Six binary indicators on the front panel give visual indication of the selected channel.

Multiplexer Switching Time: The time required to switch from one channel to any program-specified channel, or to select the next adjacent channel when the content of the CAR is incremented is 2.5 microseconds. This time is measured from when either a select or increment command is received.

Both the converter and multiplexer circuits are constructed entirely of FLIP CHIP modules. Both the Type 138E converter and the Type 139E Multiplex Control (implemented to 24 input channels) circuits can be contained in one standard 64-connector module mounting panel.

The following IOT commands have been assigned to the Type 138E/139E converter system:

SKIP ON A-D FLAG (ADSF)

Octal Code: 6531

Event Time: 1

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The A-D converter flag is sensed, and if it contains a binary 1 (indicating that the conversion is complete) the content of the PC is incremented by one so that the next instruction is skipped.

Symbol: If A-D Flag = 1, then PC + 1 => PC

CONVERT ANALOG VOLTAGE TO DIGITAL VALUE (ADCV)

Octal Code: 6532

Event Time: 2

Indicators: IOT, FETCH, PAUSE

Execution Time: This time is a function of the accuracy and word length switch setting as listed in Table 2.

Operation: The A-D converter flag is cleared, the analog input voltage is converted to a digital value, and then the A-D converter flag is set to 1. The number of binary bits in the digital-value word and the accuracy of the word is determined by the preset switch position.

Symbol: 0 => A-D Flag at start of conversion, then
1 => A-D Flag when conversion is done.

READ A-D CONVERTER BUFFER (ADRB)

Octal Code: 6534

Event Time: 3

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The converted number contained in the converter buffer (ADCB) is transferred into the AC as a normalized word (shifted into the most significant bits), unused bits of the AC are cleared, and the A-D converter flag is cleared.

Symbol: ADCB => AC
0 => A-D Converter Flag

CLEAR MULTIPLEXER CHANNEL (ADCC)

Octal Code: 6541

Event Time: 1

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The channel address register (CAR) of the multiplexer is cleared in preparation for setting of a new channel.

Symbol: 0 => CAR

SET MULTIPLEXER CHANNEL (ADSC)

Octal Code: 6542

Event Time: 2

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The channel address register of the multiplexer is set to the channel specified by bits 6 through 11 of the AC. A maximum of 64 single-ended or 32 differential input channels can be used.

Symbol: AC 6-11 => CAR

INCREMENT MULTIPLEXER CHANNEL (ADIC)

Octal Code: 6544

Event Time: 3

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The content of the channel address register of the multiplexer is incremented by one. If the maximum address is contained in the register when this command is given, the minimum address (00) is selected.

Symbol: CAR +1 => CAR

A program to cycle through all channels of the converter a given number of times, storing the conversion values at successive core memory locations can be written as follows:

```
LOOP,       ADIC            /INCREMENT CAR
            ADCV            /INITIATE CONVERSION
            ADSF            /WAIT FOR FLAG
            JMP.-1
            ADRB            /READ A-D CONVERTER BUFFER
            DCA I Z 10      /STORE RESULT IN ADDRESS SPECIFIED
                            /BY AUTO-INDEX REGISTER 10
            ISZ CNTR        /INCREMENT CYCLE COUNTER
            JMP LOOP        /REPEAT CYCLE
                            /END OF LOOP
```

Executive of this program loop takes 25.5 microseconds plus the conversion time, which is 35 microseconds maximum. Therefore, the worst case conditions for this routine require a 60.5-microsecond execution time and a minimum conversion rate of 16.5 kc.

CHAPTER 7

DIGITAL-TO-ANALOG CONVERTER TYPE AA01A

The general purpose Digital-to-Analog Converter Type AA01A converts 12-bit binary computer output numbers to analog voltages. The basic option consists of three channels, each containing a 12-bit digital buffer register and a digital-to-analog converter (DAC). Digital input to all three registers is provided, in common, by one 12-bit input channel which receives bussed output connections from the PDP-8 accumulator. Appropriate precision voltage reference supplies are provided for the converters.

One IOT microinstruction simultaneously selects a channel and transfers a digital number into the selected register. Each converter operates continuously on the content of the associated register to provide an analog output voltage.

Type AA01A options can be specified in a wide range of basic configurations; e.g., with from one to three channels, with or without output operational amplifiers, and with internally or externally supplied reference voltages. Configurations with double buffer registers in each channel are also available.

Each single-buffered channel of the equipment is operated by a single IOT command. Select codes of 55, 56, and 57 are assigned to the AA01A, making it possible to operate nine single-buffered channels or various configurations of double-buffered channels. A typical instruction for the AA01A is:

LOAD DIGITAL-TO-ANALOG CONVERTER 1 (DAL1)

Octal Code: 6551

Event Time: 1

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The content of the accumulator is loaded into the digital buffer register of channel 1.

Symbol: AC => DAC1

The analog output voltage of a standard converter is from ground to −9.9976 volts (other voltages are available in equipment containing output operational amplifiers). All binary input numbers are assumed to be 12 bits in length with negative numbers represented in 2's complement notation. An input of 4000_8 yields an output of ground potential; an input of 0000_8 yields an output of −5 volts; and an input of 1777_8 yields an output of −10 volts minus the analog value of the least significant digital bit. Output accuracy is ±0.0125% of full scale and resolution is 0.025% of full scale value. Response time, measured directly at the converter output, is 3 microseconds for a full-scale step change to 1 least significant bit accuracy. Maximum buffer register loading rate is 2 megacycles.

CHAPTER 8
DISPLAY EQUIPMENT

Cathode-ray tube display equipment available for use with the PDP-8 includes the Oscilloscope Display Type 34D and the Precision Display Type 30N. The Light Pen Type 370 operates with either of these devices.

Oscilloscope Display Type 34D

Type 34D is a two axis digital-to-analog converter and an intensifying circuit, which provides the Deflection and Intensify signals needed to plot data on an oscilloscope. Coordinate data is loaded into an X buffer (XB) or a Y buffer (YB) from bits 2 through 11 of the accumulator. The binary data in these buffers is converted to a −10 to 0 volt Analog Deflection signal. The 30-volt Intensify signal is connected to the grid of the oscilloscope CRT. The duration of this signal, and hence the intensity of the point displayed, is determined by a 2-bit brightness register (BR). The content of the BR controls timing circuits that establish nominal durations of 1-, 2-, or 4-microsecond for the Intensify signal. The BR is loaded from a number contained in the appropriate IOT instruction. Application of power to the computer or pressing of the START key resets the BR to the maximum brightness. Points can be plotted at approximately a 30-kilocycle rate. The instructions for this display are:

CLEAR X COORDINATE BUFFER (DCX)

Octal Code: 6051

Event Time: 1

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The X coordinate buffer is cleared in preparation for receiving new X-axis display data.

Symbol: 0 => XB

CLEAR AND LOAD X COORDINATE BUFFER (DXL)

Octal Code: 6053

Event Time: 1, 2

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The X cordinate buffer is cleared, then loaded with new X-axis data from bits 2 through 11 of the AC.

Symbol: 0 => XB
AC2-11 => XB

CLEAR Y COORDINATE BUFFER (DCY)

Octal Code: 6061

Event Time: 1

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The Y coordinate buffer is cleared in preparation for receiving new Y-axis display data.

Symbol: 0 => YB

CLEAR AND LOAD Y COORDINATE BUFFER (DYL)

Octal Code: 6063

Event Time: 1, 2

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The Y coordinate buffer is cleared then loaded with new Y-axis data from bits 2 through 11 of the AC.

Symbol: 0 => YB
AC 2-11 => YB

INTENSIFY (DIX)

Octal Code: 6054

Event Time: 3

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: Intensify the point defined by the content of the X and Y coordinate buffers. This command can be combined with the DXL command.

Symbol: None

INTENSIFY (DIY)

Octal Code: 6064

Event Time: 3

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: Intensify the point defined by the content of the X and Y coordinate buffers. This command is identical to the DIX command except that it can be combined with the DYL command.

Symbol: None

X COORDINATE SEQUENCE (DXS)

Octal Code: 6057

Event Time: 1, 2, 3

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: This command executes the combined functions performed by the DXL and DIX commands. The X coordinate buffer is cleared then loaded from the content of AC2 through AC11, then the point defined by the content of the X and Y buffers is intensified.

Symbol: 0 => XB
AC 2-11 => XB
then intensify

Y COORDINATE SEQUENCE (DYS)

Octal Code: 6067

Event Time: 1, 2, 3

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: This command executes the combined functions performed by the DYL and DIY commands. The Y coordinate buffer is cleared,then loaded from the content of bits AC2 through 11, then the point defined by the content of the X and Y coordinate buffers is intensified.

Symbol: 0 => YB
AC 2-11 => YB
then intensify

SET BRIGHTNESS CONTROL (DSB)

Octal Code: 607X

Event Time: 3

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The brightness register (BR) is loaded from the content of bits 10 and 11 of the instruction. When the instruction is 6075 the minimum brightness (0.4 microsecond) is set, when 6076 the medium brightness (0.8 microsecond) is set, and when 6077 the maximum brightness (3.0 microseconds) is set.

Symbol: MB10-11 = >BR

The following program sequence to display a point assumes that the coordinate data is stored in known addresses X and Y .

```
X,
Y,
BEG,    CLA
        TAD X   /LOAD AC WITH X
        DXL     /CLEAR AND LOAD XB
        CLA
        TAD Y   /LOAD AC WITH Y
        DYS     /CLEAR AND LOAD YB, DISPLAY POINT
```

Precision CRT Display Type 30N

Type 30N functions are similar to those of the Type 34D Oscilloscope Display in plotting points on a self-contained 16-inch cathode ray tube. A 3-bit brightness register is contained in Type 30N to control the duration of the Intensify signal supplied to the CRT. The content of this register specifies the brightness of the point being displayed according to the following scale:

BR Content	Intensity
3	brightest
2	
1	
0	average
7	
6	
5	
4	dimmest

The BR register is loaded by jam transfer (transfer ones and zeros so that clearing is not required) from the AC by the instruction:

LOAD BRIGHTNESS REGISTER (DLB)

Octal Code: 6074

Event Time: 3

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The brightness register (BR) is loaded by a jam transfer of information contained in bits 9 through 11 of the AC.

Symbol: AC 9-11 => BR

All other instructions and the instruction sequence are similar to those used in the Type 34D.

Light Pen Type 370

The light pen is a photosensitive device which detects the presence of information displayed on a CRT. If the light pen is held against the face of the CRT at a point displayed, the display flag will be set to a 1. The light pen display flag is connected into the computer instruction skip facility. The commands are:

SKIP ON DISPLAY FLAG (DSF)

Octal Code: 6071

Event Time: 1

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The content of the display flag is sensed, and if it contains a 1 the content of the PC is incremented by one so that the next sequential instruction is skipped.

Symbol: If Display Flag = 1, then PC +1 => PC

CLEAR THE DISPLAY FLAG (DCF)

Octal Code: 6072

Event Time: 2

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The display flag is cleared in preparation for sensing another point on the CRT.

Symbol: 0 => Display Flag

CHAPTER 9

INCREMENTAL PLOTTER AND CONTROL TYPE 350B

Four models of California Computer Products Digital Incremental Recorder can be operated from a DEC Type 350 Increment Plotter Control. Characteristics of the four recorders are:

CCP Model	Step Size (inches)	Speed (steps/minute)	Paper Width (inches)
563	0.01 or 0.005	12,000	31
565	0.01 or 0.005	18,000	12

The principles of operation are the same for each of the four models of Digital Incremental Recorders. Bidirectional rotary step motors are employed for both the X and Y axes. Recording is produced by movement of a pen relative to the surface of the graph paper, with each instruction causing an incremental step. X-axis deflection is produced by motion of the drum; Y-axis deflection, by motion of the pen carriage. Instructions are used to raise and lower the pen from the surface of the paper. Each incremental step can be in any one of eight directions through appropriate combinations of the X and Y axis instructions. All recording (discrete points, continuous curves, or symbols) is accomplished by the incremental stepping action of the paper drum and pen carriage. Front panel controls permit single-step or continuous-step manual operation of the drum and carriage, and manual control of the pen solenoid. The recorder and control are connected to the computer program interrupt and instruction skip facility.

Instructions for the recorder and control are:

SKIP ON PLOTTER FLAG (PLSF)

Octal Code: 6501

Event Time: 1

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The plotter flag is sensed, and if it contains a 1 the content of the PC is incremented by one so the next sequential instruction is skipped.

Symbol: If Plotter Flag = 1, then PC +1 => PC

CLEAR PLOTTER FLAG (PLCF)

Octal Code: 6502

Event Time: 2

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The plotter flag is cleared in preparation for issuing a plotter operation command.

Symbol: 0 => Plotter Flag

PEN UP (PLPU)

Octal Code: 6504

Event Time: 3

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The plotter pen is raised from the surface of the paper.

Symbol: None

PEN RIGHT (PLPR)

Octal Code: 6511

Event Time: 1

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The plotter pen is moved to the right in either the raised or lowered position.

Symbol: None

DRUM UP (PLDU)

Octal Code: 6512

Event Time: 2

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The plotter paper drum is moved upward. This command can be combined with the PLPR and PLDD commands.

Symbol: None

DRUM DOWN (PLDD)

Octal Code: 6514

Event Time: 3

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The plotter paper drum is moved downward.

Symbol: None

PEN LEFT (PLPL)

Octal Code: 6521

Event Time: 1

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The plotter pen is moved to the left in either the raised or lowered position.

Symbol: None

DRUM UP (PLUD)

Octal Code: 6522

Event Time: 2

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The plotter paper drum is moved upward. This command is similar to command 6512 except that it can be combined with the PLPL or PLPD commands.

Symbol: None

PEN DOWN (PLPD)

Octal Code: 6524

Event Time: 3

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The plotter pen is lowered to the surface of the paper.

Symbol: None

Program sequence must assume that the pen location is known at the start of a routine since there is no means of specifying an absolute pen location in an incremental plotter. Pen location can be preset by the manual controls on the recorder. During a subroutine, the PDP-8 can track the location of the pen on the paper by counting the instructions that increment position of the pen and the drum.

CHAPTER 10
CARD READER AND CONTROL TYPE CR01C

The Card Reader and Control Type CR01C reads standard 12-row, 80-column punched cards at a maximum rate of 100 cards per minute. Cards are read by column, beginning with column 1. One select instruction starts the card moving past the read station. Once a card is in motion, all 80 columns are read. Data in a card column is sensed by mechanical star wheels which close an electrical contact when a hole (binary 1) is detected. Column information is read in one of two program selected modes: alphanumeric and binary. In the alphanumeric mode the 12 information bits in one column are automatically decoded and transferred into the least significant half of the accumulator as a 6-bit Hollerith code. Appendix 2 lists the Hollerith card codes. In the binary mode the 12 bits of a column are transferred directly into the accumulator so that the top row (12) is transferred into AC0 and the bottom row (9) is transferred into AC11. A punched hole is interpreted as a binary 1 and no hole is interpreted as a binary 0.

Three program flags indicate card reader conditions to the computer. The data ready flag rises and requests a program interrupt when a column of information is ready to be transferred into the AC. A read alphanumeric or read binary command must be issued within 1.5 milliseconds after the data ready flag rises to prevent data loss. The card done flag rises and requests a program interrupt when the card leaves the read station. A new select command must be issued within 25 milliseconds after the card done flag rises to keep the reader operating at maximum speed. Sensing of this flag can eliminate the need for counting columns, or combined with column counting can provide a check for data loss. The reader-not-ready flag can be sensed by a skip command to provide indication of card reader power off, no card in the read station, or that a reader failure has been detected. When this flag is raised the reader cannot be selected and select commands are ignored. The reader-not-ready flag is not connected to the program interrupt facility and cannot be cleared under program control. Manual intervention is required to clear the reader-not-ready flag. Instructions for the CR01C are:

SKIP ON DATA READY (RCSF)

Octal Code: 6631

Event Time: 1

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The content of the data ready flag is sensed, and if it contains a 1 (indicating that information for one card column is ready to be read) the content of the PC is incremented by one so the next sequential instruction is skipped.

Symbol: If Data Ready Flag = 1, then PC + 1 => PC

READ ALPHANUMERIC (RCRA)

Octal Code: 6632

Event Time: 2

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The 6-bit Hollerith code for the 12 bits of a card column are transferred into bits 6 through 11 of the AC, and the data ready flag is cleared.

Symbol: AC6-11 V Hollerith Code => AC6-11
0 => Data Ready Flag

READ BINARY (RCRB)

Octal Code: 6634

Event Time: 3

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The 12-bit binary code for a card column is transferred directly into the AC, and the data ready flag is cleared. Information from the card column is transferred into the AC so that card row 12 enters AC0, row 11 enters AC1, row 0 enters AC2, ... and row 9 enters AC 11.

Symbol: AC V Binary Code => AC
0 => Data Ready Flag

SKIP ON CARD DONE FLAG (RCSP)

Octal Code: 6671

Event Time: 1

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The content of the card done flag is sensed, and if it contains a 1 (indicating that the card has passed the read station) the content of the PC is incremented to skip the next sequential instruction.

Symbol: If Card Done Flag = 1, then PC + 1 => PC

SELECT CARD READER AND SKIP IF READY (RCSE)

Octal Code: 6672

Event Time: 2

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The content of the reader-not-ready flag is sensed and if it contains a 1 (indicating that the card reader is ready for programmed operation) the PC is incremented to skip the next sequential instruction; a card is started towards the read station from the feed hopper; and the card done flag is cleared. If the reader-not-ready flag contains a 0 (indicating power is off or no card is in the read station) card selection (motion) does not occur and the skip does not occur.

Symbol: If Reader-Not-Ready Flag = 1, then PC + 1 => PC
0 => Card Done Flag

CLEAR CARD DONE FLAG (RCRD)

Octal Code: 6674

Event Time: 3

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The card done flag is cleared. This command allows a program to stop reading at any point in a card deck.

Symbol: 0 => Card Done Flag

A logical instruction sequence to read cards is:

```
START,      RCSE            /START CARD MOTION AND SKIP IF READY
            JMP NOT RDY     /JUMP TO SUBROUTINE THAT TYPES OUT
                            /"CARD READER MANUAL INTERVENTION
                            /REQUIRED" OR HALTS
NEXT,       RCSF            /DATA READY?
            JMP .-1         /NO, KEEP WAITING
            RCRA or RCRB    /YES, READ ONE CHARACTER OR ONE
                            /COLUMN
            DCA I  STR      /STORE DATA
            RCSD            /END OF CARD?
            JMP NEXT        /NO, READ NEXT COLUMN
            JMP OUT         /YES, JUMP TO SUBROUTINE THAT CHECKS
                            /CARD COUNT OR REPEATS AT START FOR
                            /NEXT CARD
```

No validity or registration checking is performed by the CR01C. A programmed validity check can be made by reading each card column in both the alphanumeric and the binary mode (within the 1.5 millisecond time limitation), then performing a comparison check.

Before commencing a card reading program energize the reader, load the feed hopper with cards, and manually feed the first card to the read station. The function of the manual controls and indicators are as follows (as they appear from right to left on the card reader):

Control or Indicator	Function
ON/OFF switch	Controls the application of primary power to the reader. When power is applied, the reader is ready to respond to operation of the other keys or programmed commands.
AUTO/MAN switch	Controls card reading. In the manual position this switch disables the card feed mechanism so that cards must be manually placed on the read table and registered by pressing the REG key. In the automatic position card motion from the feed hopper through the read station is under program control.
REG switch	When the AUTO/MAN switch is in the AUTO position the REG key is used to feed the first card to the read station. When the AUTO/MAN switch is in the MAN position the REG key is used to feed a card manually placed on the read table.
SKIP switch	This key is not connected on the CR01C and has no effect on equipment operation.
CHECK READER indicator	This lamp is not connected on the CR01C.
READY indicator	Lights when the reader is energized and cards are present in the feed hopper. The plastic card cover should always be used on top of a deck of cards to assure that the ready switch and indicator are activated.
CARD RELEASE pushbutton	When pressed, this pushbutton (adjacent to the read station) releases a card already in the read station.

CHAPTER 11

CARD READER AND CONTROL TYPE 451

The Card Reader and Control Type 451A operates at a rate of 200 cards per minute, and the Type 451B operates at a rate of 800 cards per minute. Cards are read column by column. Column information is read in either alphanumeric or binary mode. The alphanumeric mode converts the 12-bit Hollerith code of one column into the 6-bit binary-coded decimal-code with code validity checking. The binary mode reads a 12-bit column directly into the PDP-8. Approximately one percent of the computer program running time is required to execute a routine that reads the 80 columns of information at the 200 cards per minute rate.

The control of the card reader differs from the control of other input devices, in that the timing of the read-in sequence is dictated by the device. When the command to fetch a card is given, the card reader reads all 80 columns of information in sequence. To read a column, the program must respond to a flag set as each new column is started. The instruction to read the column must come within 2.2 milliseconds of the flag at 200 cards per minute, or must come within 400 microseconds at 800 cards per minute. The commands for either card reader are:

SKIP ON CARD READER FLAG (CRSF)

Octal Code: 6632

Event Time: 2

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The content of the card reader flag is sensed, and if it contains a 1 (indicating that a card column is present for reading) the content of the PC is incremented by one so that the next sequential instruction is skipped.

Symbol: If Card Reader Flag = 1, then PC +1 => PC

READ CARD EQUIPMENT STATUS (CERS)

Octal Code: 6634

Event Time: 3

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The content of the card reader flag and status levels are transferred into the content of bits AC6 through AC9. The AC bit assignments are:

AC6 = Flag is set to 1 (the flag rises after reading each of the 80 rows).

AC7 = Card done.

AC6 = Not ready (covers not in place, power is off, START pushbutton has not been pressed, hopper is empty, stacker is full, a card is jammed, a validity check error has been detected, or the read circuit is defective).

AC9 = End of the file (EOF) (hopper is empty and operator has pushed EOF pushbutton).

Symbol: Status => AC6-9

READ CARD READER BUFFER (CRRB)

Octal Code: 6671

Event Time: 1

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The content of the card column buffer (CCB) is transferred into the AC and the card reader flag is cleared. One CRRB command reads either alphanumeric or binary information.

Symbol: CCB => AC

SELECT ALPHANUMERIC (CRSA)

Octal Code: 6672

Event Time: 2

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The card reader alphanumeric mode is selected and a card is started moving past the read heads. Information read into the CCB is in 6-bit alphanumeric form (the Hollerith code representing the decoded 12 row character in one column).

Symbol: None

SELECT BINARY (CRSB)

Octal Code: 6674

Event Time: 3

Indicators: IOT, FETCH, PAUSE

Operation: The card reader alphanumeric mode is selected and a card is started moving past the read heads. Information read into the CCB is in 12-bit binary form.

Symbols: None

Upon instruction to read the card reader buffer, the content of the 12-bit CCB is transferred into the AC. In the alphanumeric mode a 6-bit Hollerith code is transferred into AC6 through AC11 and AC0 through AC5 are cleared. In the binary mode the binary content of the 12 bits (or rows) in a card column are transferred into the AC so that row X is read into AC0, row Y into AC1, row 0 into AC2 . . . . and row 9 into AC11. The mode is specified by either the CRSA or CRSB command and can be changed while the card is being read.

CHAPTER 12
CARD PUNCH CONTROL TYPE 450

The Card Punch Control Type 450 permits operation of a standard IBM Type 523 Summary Punch with the PDP-8. Punching can occur at a rate of 100 cards per minute. Cards are punched one row at a time at 40-millisecond intervals.

The card punch dictates the timing of a read-out sequence, much as the card reader controls the read-in timing. When a card leaves the hopper, all 12 rows are punched at intervals of 40 milliseconds. Punching time for each row is 24 milliseconds, leaving 16 milliseconds to load the buffer for the next row. A card punch flag indicates that the buffer is ready to be loaded. The commands for the card punch control are:

SKIP ON CARD PUNCH FLAG (CPSF)

Octal Code: 6631

Event Time: 1

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The card punch flag is sensed, and if it contains a binary 1 (indicating that the punch buffer is available and can be loaded) the content of the PC is incremented by one so the next sequential instruction is skipped.

Symbol: If Card Punch Flag = 1, then PC + 1 => PC

CARD EQUIPMENT READ STATUS (CERS)

Octal Code: 6634

Event Time: 3

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The content of the card punch flag and the status of the Card Punch Error signal in the control are transferred into bits 10 and 11 of the AC, respectively.

Symbol: Card Punch Flag => AC10
Card Punch Error => AC11

CLEAR PUNCH FLAG (CPCF)

Octal Code: 6641

Event Time: 1

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The card punch flag is cleared in preparation for giving a selector punch command.

Symbol: 0 => Card Punch Flag

SELECT CARD PUNCH (CPSE)

Octal Code: 6642

Event Time: 2

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The card punch is selected and a card is transported to the 80-column punch die from the hopper.

Symbol: None

LOAD CARD PUNCH (CPLB)

Octal Code: 6644

Event Time: 3

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The content of the AC is transferred into a portion of the 80-bit card punch row buffer (CPB). Seven CPLB commands are required to fill the CPB.

Symbol: AC => CPB

Since 12 bits are transmitted with each IOT instruction, seven IOT instructions must be issued to load the 80-bit row buffer. The first six loading instructions fill the first 72 bits (or columns); the seventh loads the remaining 8 bits of the buffer from AC bits 4 through 11. After the last row of punching is complete, 28 milliseconds are available to select the next card for continuous punching. If the next card is not requested in this interval, the card punch will stop. The maximum rate of the punch is 100 cards per minute in continuous operation. A delay of 1308 milliseconds follows the command to select the first card; a delay of 108 milliseconds separates the punching of cards in continuous operation.

The card punch flag is connected to the program interrupt and to bit 10 of the CERS instruction. Faults occurring in the punch are detected by status bit 11 of the CERS and signify the punch is disabled, the stacker is full, or the hopper is empty.

A program sequence to punch 12 rows of data on a card can be written as follows, assuming the data to be punched in each row is stored in seven consecutive core memory locations beginning in location 100. The program begins in register PNCH.

```
PNCH,     CLA
          CERS              /READ CARD STATUS
          RAR               /ROTATE PUNCH ERROR BIT (AC11)
                            /INTO LINK
          SZL               /PUNCH ERROR?
          JMP CPERR         /YES, JUMP TO PUNCH ERROR SEQUENCE
          CPSE              /NO, SELECT CARD PUNCH
          CLA
          TAD LOC           /INITIALIZE CARD IMAGE
          DAC 10
          TAD RCNT
          DCA TEM1          /INITIALIZE 12 ROW COUNTS
LP1,      CLA
          TAD GPCT          /INITIALIZE 7 GROUPS PER ROW
          DCA TEM2
          CPSF              /SENSE PUNCH LOAD AVAILABILITY
          JMP .-1
LP2,      CLA
          TAD I 10          /7 GROUPS OF 12 BITS PER ROW
          CPLB              /LOAD BUFFER
          ISZ TEM2
          JMP LP2
          ISZ TEM1          /TEST FOR 12 ROWS
          JMP LP1
          HLT               /END PUNCHING OF 1 CARD
LOC,      77                /LOCATION OF CARD IMAGE
RCNT,     -14               /12 ROWS PER CARD
GPCT,     -7                /7 GROUPS PER ROW
TEM1,     0                 /ROW COUNTER
TEM2,     0                 /GROUP COUNTER
```

CHAPTER 13
AUTOMATIC LINE PRINTER AND CONTROL TYPE 645

The line printer can print 300 lines of 120 characters per minute. Each character is selected from a set of 64 available, by a 6-bit binary code (Appendix 2 lists the ASCII character specified for each code). Each 6-bit code is loaded separately into a core storage printing buffer (LPB) from bits 6 through 11 of the AC. The LPB is divided into two 120-character sections. To load one section of the LPB requires 120 load instructions. A print command causes the characters specified by the last-loaded section of the LPB to be printed on one line. As printing of one section of the LPB is in progress, the other section can be reloaded. After the last character in a line is printed, the section of the LPB from which characters were just printed is cleared automatically. The section of the LPB that is loaded and printed is alternated automatically within the printer and is not program specified.

The line printer can load characters into the LPB at a 10-microsecond rate, clears one section of the LPB in 3 to 6 milliseconds, and moves paper at the rate of one line every 18 milliseconds. When transfer of one code into the LPB is completed, the line printer done flag rises to indicate that the printer is ready to receive another code. When printing of the last character of a section of the LPB is completed, the line printer done flag rises and causes a program interrupt to request reloading of that section of the LPB. A line printer error flag rises and causes a program interrupt if the line printer detects an inoperative condition (printer power off, control circuits not reset, paper supply low, etc.).

A 3-bit format register (FR) in the printer is loaded from bits 9 through 11 of the AC during a print command. This register selects one of eight channels of a perforated tape in the printer to control spacing of the paper. The tape moves in synchronism with the paper until a hole is sensed in the selected channel to halt paper advance. A recommended tape has the following characteristics:

FR Code (Octal)	Paper Spacing	Tape Track
0	1 line	2
1	2 lines	3
2	3 lines	4
3	6 lines (1/4 page)	5
4	11 lines (1/2 page)	6
5	22 lines (3/4 page)	7
6	33 lines (line feed)	8
7	top of form	1

The IOT microinstructions which command the line printer are:

SKIP ON LINE PRINTER ERROR (LSE)

Octal Code: 6651

Event Time: 1

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The content of line printer error flag is sensed, and if it contains a binary 1, indicating that an error has been detected, the content of the PC is incremented by one so that the next sequential instruction is skipped.

Symbol: If Line Printer Error Flag = 1, then PC + 1 => PC

CLEAR PRINTER BUFFER (LCB)

Octal Code: 6652

Event Time: 2

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: Both sections of the line printer buffer are cleared in preparation for receiving new character information.

Symbol: 0 => LPB

LOAD PRINTER BUFFER (LLB)

Octal Code: 6654

Event Time: 3

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: A section of the printer buffer is loaded from the content of bits 6 through 11 of the AC, then the AC is cleared.

Symbol: AC6 − 11 => LPB, then 0 => AC

SKIP ON LINE PRINTER DONE FLAG (LSD)

Octal Code: 6661

Event Time: 1

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The content of the line printer done flag is sensed and if it contains a binary 1 the content of the PC is incremented by one so that the next sequential instruction is skipped.

Symbol: If Line Printer Done Flag = 1, then PC + 1 => PC

CLEAR LINE PRINTER FLAGS (LCF)

Octal Code: 6662

Event Time: 2

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The line printer done and error flags are cleared

Symbol: 0 => Line Printer Done Flag
0 => Line Printer Error Flag

CLEAR FORMAT REGISTER (LPR)

Octal Code: 6654

Event Time: 3

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The line printer format register (FR) is cleared then loaded from the content of bits 9 through 11 of the AC, and the AC is cleared. The line contained in the section of the printer buffer (LPB) loaded last is printed. Paper is advanced in accordance with the selected channel of the format tape if the content of AC8 is a 1. If AC8 is a 0 paper advance is inhibited.

Symbol: 0 => FR
AC9 − 11 => FR
0 => AC
The content of half of the LPB is printed
If AC8 = 1, then advance paper according to format tape channel FR

The following routine demonstrates the use of these commands in a sequence which prints an unspecified number of 120-character lines. This sequence assumes that the printer is not in operation, that the paper is manually positioned for the first line of print, and that one-character words are stored in sequential core memory locations beginning at 2000. The PRINT location starts the routine.

```
PRINT,      LCB          /INITIALIZE PRINTER BUFFER
            CLA
            TAD LOC      /LOAD INITIAL CHARACTER ADDRESS
            DCA 10       /STORE IN AUTO-INDEX REGISTER
LRPT,       TAD CNT      /INITIALIZE CHARACTER COUNTER
            DCA TEMP
LOOP,       LSD          /WAIT UNTIL PRINTING BUFFER READY
            JMP LOOP
            LCF          /CLEAR LINE PRINTER FLAG
            TAD I 10     /LOAD AC FROM CURRENT CHARACTER
                         /ADDRESS
            LLB          /LOAD PRINTING BUFFER
            ISZ TEMP     /TEST FOR 120 CHARACTERS LOADED
            JMP LOOP
```

```
          TAD FRM      /LOAD SPACING CONTROL AND
          LPR          /PRINT A LINE
          JMP LRPT     /JUMP TO PRINT ANOTHER LINE
LOC,      1777         /INITIAL CHARACTER ADDRESS −1
CNT,      −170         /CHARACTER COUNTER = 120 DECIMAL
TEMP,     0            /CURRENT CHARACTER ADDRESS
FRM,      10           /SPACING CONTROL AND FORMAT
```

CHAPTER 14
SERIAL MAGNETIC DRUM SYSTEM TYPE 251

The Type 251 Serial Magnetic Drum System is a standard option that serves as an auxiliary data storage device. Information in the PDP-8 can be stored (written) in the drum system and retrieved (read) in sectors of 128 computer words. After program initialization, sectors are transferred automatically between the computer core memory and the drum system, transfer of each word being interleaved with the running computer program under control of the computer data break facility. A word is transferred in parallel (12 bits at a time) and is read or written around the surface of the drum serially (one bit at a time). Within the drum system words consist of 12 information bits and a parity bit. Parity bits are generated internally during writing, and are read and checked during reading. Each word is transferred in about 66 microseconds; a sector transfer is completed in 8.2 milliseconds. Average access time is 8.65 milliseconds (17.3 milliseconds maximum). Track and sector format on the drum surface is such that all transfers require the same amount of time, so track and sector are specified together as an 11-bit address for 128 words.

Drum systems are available with 8, 16, 32, 64, 128, 192, or 256 tracks; each track holds 8 sectors of 128 13-bit words. The various drum system capacities are designated by a letter suffix to the system type number as follows: Type 251A, 8K words; 251B, 16K words; 251C, 32K words; 251D, 65K words; 251E, 131K words; 251F, 196K words; and 251G, 262K words.

Indicator lamps on a front panel usually display the content of the four major registers and the status of control flip-flops. The major registers are:

Drum Core Location Counter (DCL): A 15-bit register which addresses the next core memory location to or from which a word is to be transferred. As a word is transferred, DCL is incremented by one.

Drum Address Register (DAR): An 11-bit register which addresses the drum track and sector which is currently transferring data. The eight most significant bits of the DAR specify the track and the least significant three bits specify a sector on that track. At the completion of a successful sector transfer (error flag is 0) DAR is incremented by one.

Drum Final Buffer (DFB): A 12-bit register under control of the data break facility which is a buffer between the memory buffer register and the drum serial buffer. During writing, the DFB holds the next word to be written. During reading, the DFB stores the word just read from the drum until it is transferred to the PDP-8.

Drum Serial Buffer (DSB): A 14-bit register which contains a data word and two control bits. It is a serial-to-parallel converter during drum reading, and a parallel-to-serial converter during drum writing. Information is read from the drum into DSB serially and transferred to DFB in parallel. During drum writing, a word is transferred in parallel from DFB into DSB and written serially around the drum.

Instructions

The commands for the drum system are as follows:

LOAD DRUM CORE LOCATION COUNTER AND READ (DRCR)

Octal Code: 6603

Event Time: 1, 2

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The core memory location information in the AC is transferred into the DCL and the drum is prepared to read one sector of information for transfer to the specified core memory location.*

Symbol: AC => DCL
1 => Read Control

LOAD DRUM CORE LOCATION COUNTER AND WRITE (DRCW)

Octal Code: 6605

Event Time: 1, 3

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The core memory location information in the AC is transferred into the DCL and the drum is prepared to write on one sector the information beginning at the specified core memory address.*

Symbol: AC => DCL
1 => Write Control

CLEAR DRUM FLAGS (DRCF)

Octal Code: 6611

Event Time: 1

Indicators: IOT, FETCH, PAUSE on computer and COMPLETION FLAG and ERROR FLAG on the drum system become dark.

Execution Time: 3.75 microseconds

Operation: Both completion flag and error flag are cleared

Symbol: 0 => Completion Flag
0 => Error Flag

LOAD PARITY AND DATA ERROR (DREF)

Octal Code: 6612

Event Time: 2

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The content of both the parity error and data timing error flip-flops of the drum control is transferred into bits AC0 and AC1, respectively. The command allows the program to evaluate the cause of an error flag setting.

Symbol: Parity Error => AC0
Data Timing Error => AC1

*The sector, track, and core memory address are suitably incremented and allow transfer of the next sequential sector without respecifying addresses. The DRCN instruction must be given within 50 microseconds after the completion flag is set to 1 during the previous sector.

LOAD THE TRACK AND SECTOR (DRTS)

Octal Code: 6615

Event Time: 1, 3

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: Track and sector information in bits 1-11 of the AC is transferred into the DAR, the completion and error flags are cleared, and a transfer (reading or writing) is begun.

Symbol: AC1-11 = > DAR
0 = > Completion Flag
0 = > Error Flag

SKIP ON DRUM ERROR (DRSE)

Octal Code: 6621

Event Time: 1

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The error flag is sampled and if it contains a 0 (indicating no error has been detected) the PC is incremented to skip the next instruction.

Symbol: If Error Flag = 0, then PC + 1 => PC

SKIP ON DRUM COMPLETION (DRSC)

Octal Code: 6622

Event Time: 2

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The completion flag is sampled and if it contains a 1 (indicating a sector transfer is complete) the PC is incremented to skip the next instruction.

Symbol: If completion Flag = 1, then PC +1 => PC

INITIATE NEXT TRANSFER (DRCN)

Octal Code: 6624

Event Time: 3

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: Both the error and completion flags are cleared, then the transfer of the next sector is initiated.

Symbol: 0 => Error Flag
0 => Completion Flag
then start transfer

Programming

Two instructions cause the transfer of a 128-word sector. The first (DRCR or DRCW) specifies the initial core memory location of the transfer and the direction of the transfer (drum-to-core or core-to-drum). The second instruction (DRTS) specifies the track and sector address and initiates the transfer. Transfer of each word is under control of the computer data break facility and completion of a sector transfer is indicated by a completion flag that causes a program interrupt.

The eight most significant bits of a drum address select one of 256 tracks; the three least significant bits select one of eight sectors on the track. A 300-microsecond gap identifies the beginning of a track (sector 0). Even numbered sectors (0, 2, 4, and 6) are recorded consecutively following the 300-microsecond gap. A 50-microsecond gap identifies the beginning of the odd number sectors (sector 1). Odd numbered sectors (1, 3, 5, and 7) are recorded consecutively following the 50-microsecond gap. This format allows the transfer of two consecutively numbered sectors in one drum revolution, provided the continuation instruction (DRCN) is issued in the first 50 microseconds after the drum flag rises to indicate completion of a sector transfer. The program interrupt subroutine can easily determine that the drum system caused an interrupt, check the number of sectors transferred, check for drum errors by sampling the error flag, and issue the continuation instruction in 50 microseconds.

Because the selection of a track read-write head requires 200 microseconds stabilization time, a new track must be specified during the first 200 microseconds of the 300-microsecond gap for continuous transferring. If selected tracks and sectors are consecutive, uninterrupted transferring may be programmed merely by specifying continuation, since the drum system address and the core memory address are automatically incremented. However, if a data timing or parity error occurs, the track and sector number is not advanced and operations stop at the conclusion of a sector transfer. This feature allows the program to sense for error conditions and to locate the track and sector at which transmission fails.

The drum completion flag is set to 1 upon completion of a sector transfer, causing a program interrupt. The flag is cleared either by a clear flag instruction (DRCF) or automatically when one of two transfer instructions (DRTS, DRCN) is given.

The error flag, which should be checked at the completion of each transfer, indicates either of the following conditions:

(1) That a parity error has been detected after reading from drum to core.

(2) That the Break Request signal from the drum system was not answered within the required 66-microsecond period. This condition occurs either because other devices with higher priority are being serviced by the data break facility, or because an instruction requiring longer than 66 microseconds for completion is in progress when the break request is made. In reading from the drum, a data word is incorrect in core memory. In writing on the drum, the next word has not been received from the computer.

The following program examples indicate the operation of the drum system in single and multiple sector transfers.

SUBROUTINE TO TRANSFER (READ) ONE SECTOR

```
        CLA         /CALLING SEQUENCE
        TAD ADDR    /INITIAL CORE MEMORY ADDRESS
        JMS READ
        0           /TRACK AND SECTOR ADDRESS
        0           /RETURN
READ,   0
        DRCR        /DRCW TO WRITE
        TAD I READ  /LOAD AC WITH TRACK AND SECTOR ADDRESS
        DRTS
        DRSC        /DONE?
        JMP .-1     /NO
        DRSE        /ERRORS?
        JMP ERR     /JUMP TO ERROR CHECK ROUTINE
        ISZ READ
        JMP I READ  /RETURN
```

SUBROUTINE TO TRANSFER SUCCESSIVE (TWO) SECTORS

```
        CLA         /CALLING SEQUENCE
        TAD ADDR    /INITIAL CORE MEMORY ADDRESS
        JMS READ
        0           /TRACK AND SECTOR ADDRESS
        0           /RETURN
READ,   0
        DRCR        /DRCW TO WRITE
        TAD I READ  /LOAD AC WITH TRACK AND SECTOR ADDRESS
        DRTS
        DRSC        /DONE?
        JMP .-1     /NO
        DRSE        /ERRORS?
        JMP ERR     /JUMP TO ERROR CHECK ROUTINE
        DRCN        /CLEAR FLAGS, CONTINUE TRANSFER
                    /OF NEXT SECTOR
        DRSC
        JMP .-1
        DRSE
        JMP ERR
        ISZ READ
        JMP I READ  /RETURN
```

CHAPTER 15
DECTAPE SYSTEMS

The DECtape system is a standard option for the PDP-8 which serves as an auxiliary magnetic tape data storage facility. The DECtape system stores information at fixed positions on magnetic tape as in magnetic disk or drum storage devices, rather than at unknown or variable positions as is the case in conventional magnetic tape systems. This feature allows replacement of blocks of data on tape in a random fashion without disturbing other previously recorded information. In particular, during the writing of information on tape, the system reads format (mark) and timing information from the tape and uses this information to determine the exact position at which to record the information to be written. Similarly, in reading the same mark and timing information is used to locate data to be played back from the tape.

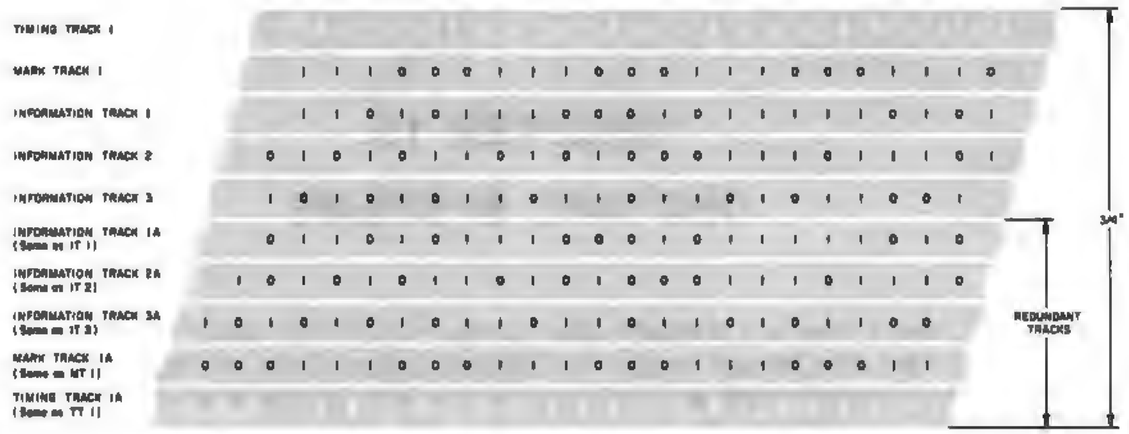
This system has a number of features to improve its reliability and make it exceptionally useful for program updating and program editing applications. These features are: phase or polarity sensed recording on redundant tracks, bidirectional reading and writing, and a simple mechanical mechanism utilizing hydrodynamically lubricated tape guiding (the tape floats on air and does not touch any metal surfaces).

DECtape Format

DECtape utilizes a 10-track read/write head. Tracks are arranged in five nonadjacent redundant channels: a timing channel, a mark channel, and three information channels. Redundant recording of each character bit on nonadjacent tracks materially reduces bit drop outs and minimizes the effect of skew. Series connection of corresponding track heads within a channel and the use of Manchester phase recording techniques, rather than amplitude sensing techniques, virtually eliminate drop outs.

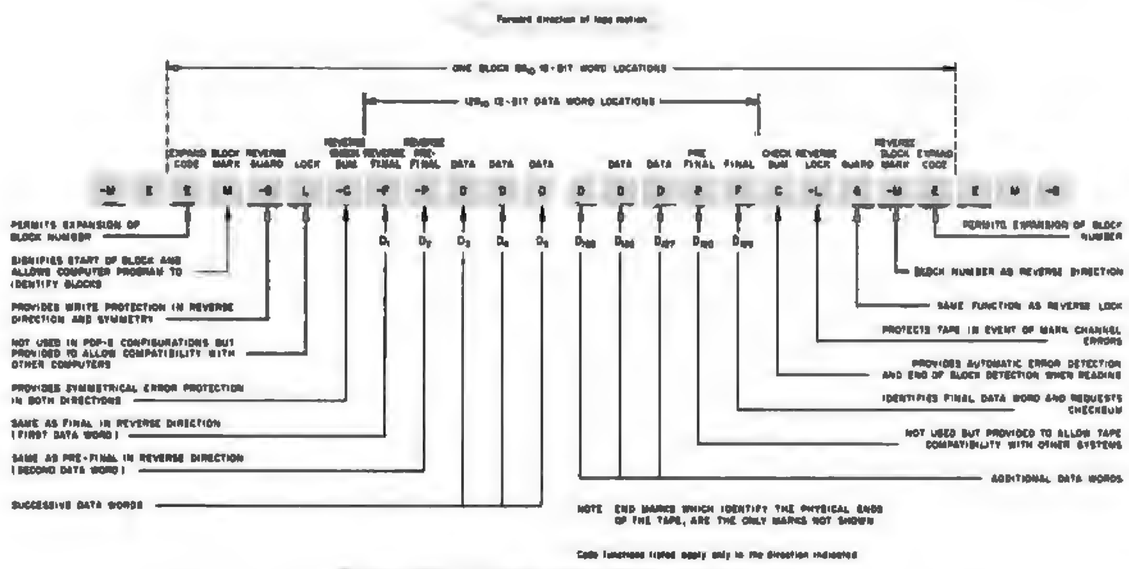
The timing and mark channels control the timing of operations within the control unit and establish the format of data contained on the information channels. The timing and mark channels are recorded prior to all normal data reading and writing on the information channels. The timing of operations performed by the tape drive and some control functions are determined by the information on the timing channel. Therefore, wide variations in the speed of tape motion do not affect system performance. Information read from the mark channel is used during reading and writing data, to indicate the beginning and end of data blocks and to determine the functions performed by the system in each control mode.

During normal data reading, the control assembles 12-bit computer length words from four successive lines read from the information channels of the tape. During normal data writing, the control disassembles 12-bit words and distributes the bits so they are recorded on four successive lines on the information channels. A mark channel error check circuit assures that one of the permissible marks is read in every six lines on the tape. This 6-line mark channel sensing requires that data be recorded in 12-line segments (12 being the lowest common multiple of 6-line marks and 4-line data words) which correspond to three 12-bit words.



TIMING TRACK 1
MARK TRACK 1
INFORMATION TRACK 1
INFORMATION TRACK 2
INFORMATION TRACK 3
INFORMATION TRACK 1A (Same as IT 1)
INFORMATION TRACK 2A (Same as IT 2)
INFORMATION TRACK 3A (Same as IT 3)
MARK TRACK 1A (Same as MT 1)
TIMING TRACK 1A (Same as TT 1)
3/4"
REDUNDANT TRACKS

Figure 9 DECtape Track Allocations



Forward direction of tape motion
ONE BLOCK 86₁₀ 18-BIT WORD LOCATIONS
128₁₀ 12-BIT DATA WORD LOCATIONS
EXPAND CODE
BLOCK MARK
REVERSE GUARD
LOCK
REVERSE CHECK SUM
REVERSE FINAL
REVERSE PRE-FINAL
DATA
PRE FINAL
FINAL
CHECK SUM
REVERSE LOCK
GUARD
REVERSE BLOCK MARK
EXPAND CODE
PERMITS EXPANSION OF BLOCK NUMBER
SIGNIFIES START OF BLOCK AND ALLOWS COMPUTER PROGRAM TO IDENTIFY BLOCKS
PROVIDES WRITE PROTECTION IN REVERSE DIRECTION AND SYMMETRY
NOT USED IN PDP-8 CONFIGURATIONS BUT PROVIDED TO ALLOW COMPATIBILITY WITH OTHER COMPUTERS
PROVIDES SYMMETRICAL ERROR PROTECTION IN BOTH DIRECTIONS
SAME AS FINAL IN REVERSE DIRECTION (FIRST DATA WORD)
SAME AS PRE-FINAL IN REVERSE DIRECTION (SECOND DATA WORD)
SUCCESSIVE DATA WORDS
PERMITS EXPANSION OF BLOCK NUMBER
BLOCK NUMBER AS REVERSE DIRECTION
SAME FUNCTION AS REVERSE LOCK
PROTECTS TAPE IN EVENT OF MARK CHANNEL ERRORS
PROVIDES AUTOMATIC ERROR DETECTION AND END OF BLOCK DETECTION WHEN READING
IDENTIFIES FINAL DATA WORD AND REQUESTS CHECKSUM
NOT USED BUT PROVIDED TO ALLOW TAPE COMPATIBILITY WITH OTHER SYSTEMS
ADDITIONAL DATA WORDS
NOTE END MARKS WHICH IDENTIFY THE PHYSICAL ENDS OF THE TAPE, ARE THE ONLY MARKS NOT SHOWN
Code functions listed apply only in the direction indicated

Figure 10 DECtape Mark Channel Format

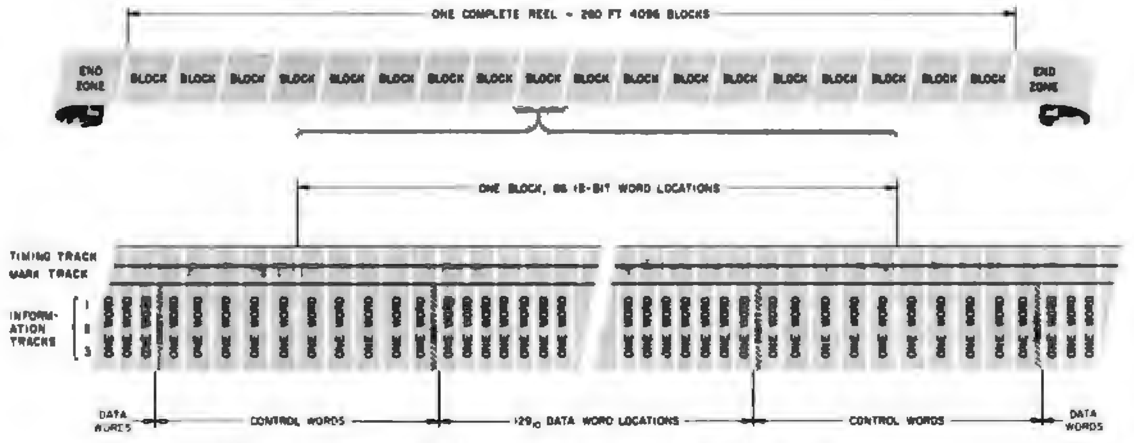


Figure 11 DECtape Control Word and Data Word Assignments

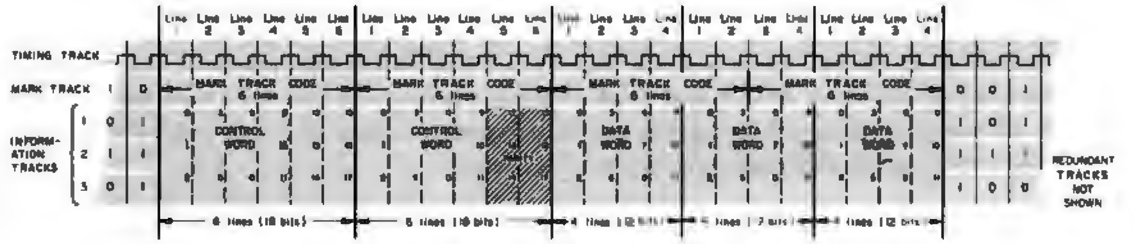


Figure 12 DECtape Format Details

A tape contains a series of data blocks that can be of any length which is a multiple of three 12-bit words. Block length is determined by information on the mark channel. Usually a uniform block length is established over the entire length of a reel of tape by a program which writes mark and timing information at specific locations. The ability to write variable-length blocks is useful for certain data formats. For example, small blocks containing index or tag information can be alternated with large blocks of data. (Software supplied with DECtape allows writing for fixed block lengths only.)

Between the blocks of data are areas called interblock zones. The interblock zones consist of 30 lines on tape before and after a block of data. Each of these 30 lines is divided into five 6-line control words. These 6-line control words allow compatibility between DECtape written on any of DEC's 12-, 18-, or 36-bit computers. As used on the PDP-8, only the last four lines of each control word are used.

Block numbers normally occur in sequence from 1 to N. There is one block numbered 0 and one block N+1. Programs are entered with a statement of the first block number to be used and the total number of blocks to be read or written. The total length of the tape is equivalent to 849,036 lines which can be divided into any number of blocks up to 4096 by prerecording of the mark track. The maximum number of blocks is determined by the following equation in which N_B = number of blocks and N_W = number of words per block (N_W must be divisible by 3).

$$N_B = \frac{212112}{(N_W + 15)} - 2$$

DECtape format is illustrated in Figures 9 through 12.

DECtape Dual Transport Type 555 and DECtape Control Type 552

The Type 555 Dual DECtape Transport consists of two logically independent tape drives, capable of handling 3.5-inch reels of 0.75-inch magnetic tape. Bits are recorded at a density of 350 ±55 bits per track inch at a speed of over 80 inches per second on the 260-foot length of a reel. Each line on the tape is read or written in approximately 33⅓ microseconds. Simultaneous writing occurs in three channels (three pairs of redundant information tracks), while reading occurs in the mark and timing channels (two pairs of redundant tracks).

The Type 552 DECtape Control operates up to four 555 transports (8 drives) to transfer binary information read from the tape into 12-bit computer words approximately every 133⅓ microseconds. In writing, the control disassembles 12-bit computer words so that they are written at four successive lines on tape. Transfers between the computer and the control always occur in parallel for a 12-bit word. Data transfers use the data break facilities of the computer. As the start and end of each block are detected by the mark track detection circuits, the control raises a DECtape (DT) flag which causes a computer program interrupt. The program interrupt is used by the computer program to determine the block number and when it determines that the forthcoming block is the one selected for a data transfer, it selects the read or write control mode. Each time a word is assembled or the DECtape is ready to receive a word from the computer, the control raises a data flag. This flag is connected to the computer data break facility to signify a break request. Therefore, when the desired block is detected, the data flag causes a data break and initiates a transfer. By using the mark track decoding circuits and data break facility in this manner, computation in the main computer program can continue during tape operations.

Data Buffer (DB): This 12-bit register serves as a storage buffer for data to be transferred between DECtape and the computer memory buffer register. During a read operation information sensed from the tape is transferred into the DB from the read/write buffer and is transferred to the computer during a data break cycle. During a write operation the DB received information from the computer and transfers it to the read/write buffer for disassembly and recording on tape. In this manner the DB synchronizes data transfers by allowing transfers between itself and the read/write buffer as a function of the tape timing.

Read/Write Buffer (R/WB): This 12-bit register is composed of three 4-bit shift registers. During reading, one bit from each information track is read into a separate segment of the R/WB and shifted right or left as a function of the direction of tape movement. When four tape positions have been read, the content of the R/WB is set into the DB as an assembled 12-bit computer word. During writing, the contents of each segment of the R/WB is shifted serially to the write register (one bit from each of the three segments of the R/WB is transferred into the write register at a time to provide the data to be written at one line) for recording on tape.

Write Register: A 3-bit register which is alternately loaded from the R/WB and complemented to write the phase-coded information on tape.

Select Register: This 4-bit register is loaded under program control to specify the tape drive selected for operation from the control unit. A single Type 522 DECtape Control can select the drives of four Type 555 Dual DECtape Transports (eight tape drives).

Motion Register: This 2-bit register contains a go/stop flip-flop and a forward/reverse flip-flop which control the motion of the selected tape drive. The register is set under program control.

Longitudinal Parity Buffer (LPB): This 6-bit register performs a parity check of the information in the three information tracks. The check essentially reads the number of binary zeros in each half of a 12-bit data word and forms a parity bit to be recorded in the checksum control word at the end of the data block. This is effected by setting the information read from two consecutive tape positions into the LPB and then complementing a bit of the LPB if the corresponding bit of the R/WB contains a 0. After reading a block of data the LPB holds a number which indicates the parity of bits 0 and 6, 1 and 7, etc. A 1 in the LPB at this time indicates odd parity and a 0 indicates even parity. When a block of data has been read correctly the LPB contains all binary ones. If the LPB does not contain all ones when a block of data has been read, the parity or mark track error flip-flop is set to 1.

Memory Address Counter (MAC): This 12-bit register specifies an address in computer core memory to be used for each word transfer. During program initialization, the starting address of a transfer is set into MAC from the computer accumulator. During the transfer, the address contained in MAC is transferred into the computer memory address register for each data word. The contents of MAC is incremented by 1 at the conclusion of each word transfer so that the transfers occur between successive addresses of computer core memory and tape, regardless of tape direction.

Window (W): This 9-bit register serves as a control signal generator for the DECtape system. The mark track data is stored in the W and control signals are generated as a function of the mode of operation in progress and the contents of the W. For example, in the search mode when the W detects a block mark, control signals are generated to raise the DECtape (DT) flag to indicate the presence of a block number in the DB and signals the start of data block to the computer.

Device Selector (DS): The device selector is a gating circuit which produces the IOT pulses necessary to initiate operation of the DECtape system and strobe information into the computer.

Dectape Flag (DT): This flip-flop serves as an indicator of DECtape system operation to the computer and is connected to the computer program interrupt facility. The function of the DT flag is determined by the control mode in operation at the time, as follows:

a. In the search mode the DT flag rises each time a block mark (block number) is read to indicate the beginning of a new block and to allow programmed determination of the block number which just passed the read/write head.

b. In the read data or write data modes the DT flag rises at the end of each block to indicate the end of a data block. Under these conditions the computer program can sense for this flag to determine when the transfer is complete.

c. In the read all bits or write all bits modes the DT flag rises to indicate completion of each 12-bit word transfer. Since block marks are not observed in these modes, this flag can be used by the computer program to count the number of words transferred as a means of determining tape location.

Error Flag: This flag is raised by four error conditions. When the flag rises it initiates a program interrupt to allow the computer interrupt subroutine to determine the condition of the 552 control by means of a read status command. The four error conditions indicated are:

a. *End:* The tape of the selected transport is in the end zone and tape motion is stopped automatically. Under these conditions end is an error if it is not expected by the program in process or is a legitimate signal used to indicate the end of a normal operation (such as rewind) if it is anticipated by the program. If the transport is not selected when the tape enters the end zone this signal is not given, tape motion is not stopped automatically, and the tape can run off the end of the reel.

b. *Timing Error:* The program was not able to keep pace with the tape transfer rate or a new motion or select command was issued before the previous command was completely executed.

c. *Parity or Mark Track Error:* Indicates that during the course of the previous block transfer a data parity error was detected, or one or more bits have been picked up or dropped out from either the timing track or the mark track.

d. *Select Error:* Signifies that a tape transport unit select error has occurred such that more than one transport in the system have been assigned the same select code or that no transport has been assigned the programmed select code.

Therefore, a select error indicates an error by the operator, a timing error is a program error, and a parity or mark channel error indicates an equipment malfunction. Under certain conditions the end may also be an indication of equipment malfunction.

Data Flag: This flag is raised each time the DECtape system is ready to transfer a 12-bit word with the computer. When raised, the flag produces a computer data break.

INSTRUCTIONS

DECtape system commands are microinstructions of the PDP-8 input-output transfer (IOT) instruction, as listed in the following paragraphs:

LOAD SELECT REGISTER (MMLS)

Octal Code: 6751
Event Time: 1
Indicators: IOT, FETCH, PAUSE
Execution Time: 3.75 microseconds

Operation: The unit select register (USR) is loaded from the content of bits 2 through 5 of the AC, the DECtape flag (DT) is cleared, and a delay is initiated. Loading of the USR involves relay switching which takes approximately 70 milliseconds. When the delay expires the DT flag is set as an indication that loading of the USR is complete and the next DECtape instruction can be given.

Symbol: AC2-5 $=>$ USR
1 $=>$ DT when done

LOAD MOTION REGISTER (MMLM)

Octal Code: 6752
Event Time: 2
Indicators: IOT, FETCH, PAUSE
Execution Time: 3.75 microseconds

Operation: The motion register (MR) is loaded from the content of bits 7 and 8 of the AC, the DECtape flag is cleared, and a 70 millisecond delay is initiated. When the delay expires the DT flag is set, indicating that loading of the MR is completed and the next DECtape instruction can be given.

Symbol: AC7-8 $=>$ MR
1 $=>$ DT when done

LOAD FUNCTION REGISTER (MMLF)

Octal Code: 6754
Event Time: 3
Indicators: IOT, FETCH, PAUSE
Execution Time: 3.75 microseconds

Operation: The function register (FR) is loaded from the content of bits 9 through 11 of the AC, then the AC is cleared. Octal decoding of these three bits establish the following DECtape control modes:

0 = Move	4 = Write data
1 = Search	5 = Write all bits
2 = Read data	6 = Write mark and timing
3 = Read all bits	

Symbol: AC9-11 $=>$ FR,
then 0 $=>$ AC

SKIP ON DECTAPE FLAG (MMSF)

Octal Code: 6761
Event Time: 1
Indicators: IOT, FETCH, PAUSE
Execution Time: 3.75 microseconds

Operation: The content of the DECtape flag is sensed, and if it contains a binary 1, the content of the PC is incremented by one so the next instruction is skipped.

Symbol: If DT = 1, then PC + 1 $=>$ PC

CLEAR MEMORY ADDRESS COUNTER (MMCC)

Octal Code: 6762
Event Time: 2
Indicators: IOT, FETCH, PAUSE
Execution Time: 3.75 microseconds

Operation: The memory address counter (MAC) is cleared.

Symbol: 0 => MAC

LOAD MEMORY ADDRESS COUNTER (MMLC)

Octal Code: 6764
Event Time: 3
Indicators: IOT, FETCH, PAUSE
Execution Time: 3.75 microseconds

Operation: A transfer of binary ones is performed from the content of the AC into the memory address counter, then the AC is cleared.

Symbol: AC V MAC = > MAC
then 0 = > AC

SKIP ON ERROR FLAG (MMSC)

Octal Code: 6771
Event Time: 1
Indicators: IOT, FETCH, PAUSE
Execution Time: 3.75 microseconds

Operation: The content of the error flag is sensed, and if it contains a 1 (signifying that an error has been detected) the content of the PC is incremented by one so the next sequential instruction is skipped.

Symbol: If Error Flag = 1, then PC + 1 => PC

CLEAR FLAGS (MMCF)

Octal Code: 6772
Event Time: 2
Indicators: IOT, FETCH, PAUSE
Execution Time: 3.75 microseconds

Operation: Both the DECtape and error flags are cleared.

Symbol: 0 => DT, Error Flag

READ STATUS (MMRS)

Octal Code: 6774
Event Time: 3
Indicators: IOT, FETCH, PAUSE
Execution Time: 3.75 microseconds

Operation: The condition of the status levels is transferred into bits 0 through 7 of the AC. The AC bit assignments are:

AC0 =	DT flag
AC1 =	Error flag
AC2 =	End (selected tape at end point)
AC3 =	Timing error
AC4 =	Reverse tape direction
AC5 =	Go
AC6 =	Parity or mark track error
AC7 =	Select error

Symbol: Status Levels V AC0-7 => AC0-7

CONTROL MODES

The seven modes of operation loaded into the function register during the MMLF command are used as follows:

Move: Initiates movement of the selected transport tape in either direction. Mark channel errors are inhibited in this mode.

Search: As the tape is moved in either direction, sensing a block mark causes both the data flag and the DT flag to rise. The data flag causes a computer data break to deposit the block number in core memory at the address held in MAC. The DT flag initiates a program interrupt to cause the program to jump to a subroutine which is responsible for checking the block numbers by using either the block number stored during this operation or by counting the number of times the DT flag rises.

Read Data: A block of data is read in either direction, the data flag rises to cause a data break each time a 12-bit word is to be transferred, and the DT flag is raised to initiate a program interrupt at the end of the data block. The program is responsible for controlling tape motion at the end of a block transfer and must stop motion or change the content of the function register when the DT flag rises. The transport continues reading until taken out of the read data mode.

Read All Bits: In this mode of operation the three information channels in data blocks and interblock zones are continuously read and transferred to the computer. This mode is similar to the read data mode except that the DT flag rises each time the data flag rises. The read all bits mode is used to read an unusual tape format which is not compatible with the read data mode. The DT flag is inhibited from causing a program interrupt in this mode

Write Data: A block of data is written on tape in either direction, the data flag is raised to effect each transfer, and the DT flag is raised at the end of the block as in the read data mode.

Write All Bits: This special mode of operation is used to write information at all positions, disregarding blocks (such as in writing block numbers). The mode is similar to the read all bits mode for writing. The DT flag does not cause a program interrupt in this mode.

Write Mark and Timing: Ths mode is used to write on the timing and mark channels to establish or change block length.

PROGRAMMED OPERATION

Prerecording of a reel of DECtape, prior to its use for data storage, is accomplished in two passes. During the first pass, the timing and mark channels are placed on the tape. During the second pass, forward and reverse block mark numbers, the standard data pattern, and the automatic parity checks are written. Since part of the data word must be reserved to produce the mark channel, it is impossible to write intelligent data in the information channels at the same time. For this reason, the two passes are required. Prerecording utilizes the write timing and mark channel control mode and a manual switch in the control which permits writing on the timing and mark channels, activates a clock which produces the timing channel recording pattern, and enables flags for program control. Unless both this control mode and switch are used simultaneously, it is physically impossible to write on the mark or timing channels. A red indicator lights on all transports associated with the control when the manual switch is in the "on" position. Under these conditions only, the write register and write amplifier used to write on information channel 0 (bits 0, 3, 6, and 9) is used to write on the mark channel.

Two PDP-8 IOT microinstructions initiate operation of the DECtape system: the first (MMMM) loads the select register, motion register, and function register by means of instruction 6757 (combining MMLS, MMLM, and MMLF) and the second command (MMML is 6766, combining MMCC and MMLC) loads the MAC with the core memory address to be used to store the block number during searching. After initiating operation of the DECtape system, the program should always check for errors immediately by means of the MMSC instruction. This instruction should also be used at the conclusion of each transfer. A program should always start the DECtape system in the search mode to locate the block number selected for a transfer, then when the block number has been located the transfer is accomplished by loading the function register with the read data or write data mode.

In searching, each block number is read by the transport and is transferred to the control. The control raises the DT flag upon receipt of each block number and stores the number in the computer core memory at the address contained in MAC. The computer program, then samples the DT flag and either counts the number of blocks passed or reads the block number from core memory and compares it with the number it is seeking. The results of the data obtained in this way are used to further control the search operation. Upon determining that the forthcoming block is the one selected for a data transfer, the program loads the function register with either the read data or write data mode. Entering another mode discontinues the search mode. The starting address to be used for the first core memory address of the transfer is then set into the MAC by the computer.

When the start of the data position of the block is detected the data flag is raised to initiate a data break each time the DECtape system is ready to transfer a 12-bit word. Therefore, the main computer program continues running but is interrupted approximately every 133⅓ microseconds during a data break for the transfer of a word. Transfers occur between DECtape and successive core memory locations, commencing at the address previously set into MAC. The number of words transferred is determined by the size of the selected tape block. At the conclusion of the block transfer the DT flag is raised and a program interrupt occurs. The interrupt subroutine checks the DECtape error flag to determine the validity of the transfer and either initiates a search for the next information to be transferred or returns to the main program.

During all normal writing transfers, a checksum (the 6-bit exclusive OR of the words in the data block) is computed automatically by the control and is automatically recorded as one of the control words in the interblock zone immediately following the end of the data block. This same checksum is used during reading to determine that the data playback and recognition takes place without error.

Any one of the eight tape drives may be selected for use by the program. After using a particular drive, the program can stop the drive currently being used and select a new drive, or can select another drive while permitting the original selection to continue running. This is a particularly useful feature when rapid searching is desired, since several transports may be used simultaneously. Caution must be exercised however, for although the earlier drive continues to run, no tape end detection or other sensing takes place. Automatic end sensing that stops tape motion occurs in all modes, but only in the selected tape drive.

Whenever either the motion or select code is changed, the program must wait until the DT flag is set to 1 before giving another motion or selection command. In other words, to prevent a timing error all operations of the currently selected drive must be completed before issuing a new select code.

DECtape Transport Type TU55 and DECtape Control Type TC01

A DECtape system configuration contains up to eight TU55 transports operated from one TC01 control. All data transfers occur between the computer and the control and are effected by the three-cycle data break facility. A 12-bit data buffer in the control synchronizes transfers between the TC01 and the PDP-8 data break facility. Data read from four consecutive lines on tape by the transport are assembled into 12-bit words by a read/write buffer in the control for transfer to the computer. Data loaded into the control from the computer is disassembled by the read/write buffer and supplied to the transport for writing on four lines of tape.

Transfer of command and control signals between the computer and the control is effected by normal IOT instructions. Small registers and control flip-flops in the TC01 are joined to serve as two status registers for the transfer of command and control information with the PDP-8 accumulator. Bit assignments of these registers is indicated in Figure 13 and Figure 14 .

DECTAPE TRANSPORT TYPE TU55

The TU55 is a bidirectional magnetic-tape transport consisting of a read/write head for recording and playback of information on five channels of the tape. Connections from the read/write head are made directly to the external control which contains the read and write amplifiers.

The logic circuits of the TU55 control tape movement in either direction over the read/write head. Tape drive motor control is exercised completely through the use of solid state switching circuits to provide fast reliable operation. These switching circuits contain silcon controlled rectifiers which are controlled by normal DEC diode and transistor logic circuits. These circuits control the torque of the two motors which transport the tape across the head according to the established function of the device, i.e., go, stop, forward, or reverse. In normal tape movement, full torque is applied to the forward or leading motor and a reduced torque is applied to the reverse or trailing motor to keep proper tension on the tape. Since tape motion is bidirectional, each motor serves as either the leading or trailing drive for the tape, depending upon the forward or reverse control status of the TU55. A positive stop is achieved by an electromagnetic brake mounted on each motor shaft. When a stop command is given, the trailing motor brake latches to stop tape motion. Enough torque is then applied to the leading motor to take up slack in the tape.

Tape movement can be controlled by commands originating in the computer and applied to the TU55 through the TC01 DECtape Control, or can be controlled by commands generated by manual operation of rocker switches on the front panel of the transport. Manual control is used to mount new reels of tape on the transport, or as a quick maintenance check for proper operation of the control logic in moving the tape.

DECTAPE CONTROL TYPE TC01

The TC01 DECtape Control operates up to eight TU55 DECtape Transports. Binary information is transferred between the tape and the computer in 12-bit computer words approximately every 133⅓ microseconds. In writing, the control disassembles 12-bit computer words so that they are written at four successive lines on tape. Transfers between the computer and the control always occur in parallel for a 12-bit word. Data transfers use the three-cycle data break facility of the computer. As the start and end of each block of data are detected by the mark track detection circuits, the control raises a DECtape control flag (DTCF) which requests a computer program interrupt. The program interrupt is used by the computer program to determine the block number. When it determines that the forthcoming block is the one selected for a data transfer it establishes the appropriate read or write function. Each time a word is assembled or the DECtape system is ready to receive a word from the computer, the control raises a data flag (DF). This flag is connected to the computer data break facility to request a data break. Therefore, when each 12-bit computer word is assembled, the data flag causes a transfer via the three-cycle data break. By using the mark channel decoding circuits and the data break in this manner, computation in the main computer program can continue during DECtape operations.

Four program flags in the control serve as condition indicators and request originators.

DECtape Flag (DT): This flag indicates the active/done status of the current function and is connected to the instruction skip facility.

Data Flag (DF): This flag requests a data break to transfer a block number into the computer during a search function, or when a data word transfer is required during a read or write function.

DECtape Control Flag (DTCF): This flag, when enabled by a binary 1 in bit 9 of status register A, requests a program interrupt if either the DECtape flag or the error flag is set.

Error Flag (EF): Detection of any non-operative condition by the control sets this flag in status register B and stops the selected transport. The error conditions indicated by this flag are:

a. Mark Track Error: This error occurs any time the information read from the mark channel is erroneously decoded.

b. End of Tape: The end zone on either end of the tape is over the read head.

c. Select Error: This error occurs a few microseconds after loading status register A to indicate any one of the following conditions:

    1. Specifying a unit select code which does not correspond to any transport select number, or which is set to multiple transports.

    2. Specifying a write function with the WRITE ENABLED/WRITE LOCK switch in the WRITE LOCK position on the selected transport.

3. Specifying an unused function code (i.e. AC6-8 = 111).
4. Specifying any function except read all with the NORMAL/WRTM/RDMK switch in the RDMK position.
5. Specifying any function except write timing and mark track with the NORMAL/WRTM/RDMK switch in the WRTM position.
6. Specifying the write timing and mark track function with the NORMAL/WRTM/RDMK switch in a position other than WRTM.

d. Parity Error: This error occurs during a read data function if the longitudinal parity over the entire data word, the reverse checksum, and the checksum is not equal to 1.

e. Timing Error: This error indicates a program fault caused by one of the following conditions:
1. A data break did not occur within 66 microseconds of the data break request.
2. The DT flag was not cleared by the program before the control attempted to set it.
3. The read data or write data function was specified while a data block was passing the read head.

INSTRUCTIONS

Instructions for a TC01/TU55 system are microprogrammed commands of the PDP-8 IOT instruction and are defined as follows:

READ STATUS REGISTER A (DTRA)

Octal Code: 6761

Event Time: 1

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The content of status register A is loaded into the accumulator by an OR transfer. The AC bit assignments are:

AC0-2 = Transport unit select number
AC3(0) = Forward
AC3(1) = Reverse
AC4(0) = Stop
AC4(1) = Go
AC5(0) = Normal mode
AC5(1) = Continuous mode
AC6-8 = 0 = Move function
AC6-8 = 1 = Search function
AC6-8 = 2 = Read data function
AC6-8 = 3 = Read all function
AC6-8 = 4 = Write data function
AC6-8 = 5 = Write all function
AC6-8 = 6 = Write timing and mark tracks function
AC6-8 = 7 = Unused (causes a select error if issued)
AC9(0) = DECtape control flag and error flag disabled from causing a program interrupt

AC9(1) = DECtape control flag and error flag enabled to cause a program interrupt

Symbol: AC0-9 V Status Register A => AC0-9

CLEAR STATUS REGISTER A (DTCA)

Octal Code: 6762

Event Time: 2

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: Status register A is cleared. The DECtape control and error flags are undisturbed.

Symbol: 0 => Status Register A

LOAD STATUS REGISTER A (DTXA)

Octal Code: 6764

Event Time: 3

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The exclusive OR of the content of bits 0 through 9 of the accumulator is loaded into status register A, and bits 10 and 11 of the accumulator are sampled to control clearing of the error and DECtape flags, respectively. Loading status register A from AC0-9 establishes the transport unit select code, motion control, function, and enables or disables the DECtape control flag to request a program interrupt as described in the DTRA instruction. The sampling of AC10 and AC11 is as follows:

AC10(0) = Clear all error flags
AC10(1) = All error flags undisturbed
AC11(0) = Clear DECtape flag
AC11(1) = DECtape flag undisturbed

Symbol: AC0-9 V Status Register A => Status Register A

If AC10 = 0, then 0 => EF Flag
If AC11 = 0, then 0 => DT Flag

SKIP ON FLAGS (DTSF)

Octal Code: 6771

Event Time: 1

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The content of both the error flags and the DECtape control is sampled, and if any flag contains a binary 1, the content of the program counter is incremented by one to skip the next sequential instruction.

Symbol: If EF Flag = 1 V DTCF Flag = 1, then PC + 1 => PC

READ STATUS REGISTUS B (DTRB)

Octal Code: 6772

Event Time: 2

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The content of status register B is loaded into the accumulator by an OR transfer. The AC bit assignments are:

AC0 = Error flag (Error flag = mark track error V end of tape V select error V parity error V timing error)
AC1 = Mark track error
AC2 = End of tape
AC3 = Select error
AC4 = Parity error
AC5 = Timing error
AC6-8 = Memory field
AC9-10 = Unused
AC11 = DECtape flag

Symbol: AC V Status Register B => AC

LOAD STATUS REGISTER B (DTLB)

Octal Code: 6774

Event Time: 3

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The memory field register portion of status register B is loaded by an OR transfer from the content of bits 6 through 9 of the accumulator.

Symbol: AC6-8 V Memory Field => Memory Field

CONTROL MODES

The DECtape system operates in either the normal or continuous mode, as determined by bit 5 of status register A during a DTXA command. Operation in each mode is as follows:

Normal (NM): Data transfers and flag settings are controlled by the format of information on the tape.

Continuous (CM): Data transfers and flag settings are controlled by a word count (WC) read from core memory during the first cycle of each three-cycle data break; and by the tape format.

FUNCTIONS

The DECtape system performs one of seven functions, as determined by the octal digit loaded into status register A during a DTXA command. These functions are:

Move: Initiates movement of the selected transport tape in either direction. Mark channel decoding is inhibited in this mode except for end of tape.

Search: As the tape is moved in either direction, sensing a block mark causes a data transfer of the block number. If the word count overflows (WCO) in either NM or CM, the DT flag is set and causes a program interrupt. After finding the first block number, the CM can be used to avoid all intermediate interrupts between the current and the desired block number. This makes a virtually automatic search possible.

Read Data: This function is used to transfer blocks of data into core memory with the transfer controlled by the tape format. In NM the DT flag is set at the end of a block and causes a program interrupt. In CM transfers stop when the word count overflows, the remainder of the block is read for parity checking, and then the DT flag is set.

Read All: Read all is used to read tape in an unusual format, since it causes all lines to be read. In NM the DT flag is set at each data transfer. In CM the DT flag is set when WCO occurs. In either case the DT flag causes a program interrupt.

Write Data: This function is used to write blocks of data with the transfer controlled by the standard tape format. After WCO occurs, zeros are written in all lines of the tape to the end of the current block. Then the parity checksum for the block is written. The DT flag rises as the in the read data function.

Write All: The write all function is used to write an unusual tape format (e. g., block numbers). The DT flag raisings are similar to the read all function.

Write Timing and Mark Track: This function is used to write on the timing and mark tracks. This permits blocks to be established or block lengths to be changed. The DT flag raisings are also similar to the read all function. This function is illegal unless a manual switch in the control is on.

PROGRAMMED OPERATION

Prerecording of a reel of DECtape, prior to its use for data storage, is accomplished in two passes. During the first pass, the timing and mark channels are placed on the tape. During the second pass, forward and reverse block mark numbers, the standard data pattern, and the automatic parity checks are written. These functions are performed by the DECTOG program. Prerecording utilizes the write timing and mark channel function and a manual switch on the control which permits writing on the timing and mark channels, activates a clock which produces the timing channel recording pattern, and enables flags for program control. Unless both this control function and switch are used simultaneously, it is physically impossible to write on the mark or timing channels. A red indicator lamp on the control lights when the manual NORMAL/WRTM/RDMK switch is in the WRTM position. Under these conditions only, the write register and write amplifier used to write on information channel 1 (bits 0, 3, 6, and 9) is used to write on the mark channel. This operation of prerecording need only be performed once for each reel of DECtape.

There are two registers in the TC01 DECtape Control that govern tape operation and provide status information to the operating program. Status register A (see Figure 13) contains three unit selection bits, two motion bits, the continuous mode/normal mode bit, three function bits, and three bits that control the flags. Status register B (see Figure 14) contains the three memory field bits and the error status bits. PDP-8 IOT microinstructions are used to clear, read, and load these registers. In addition, there is an IOT skip instruction to test control status.

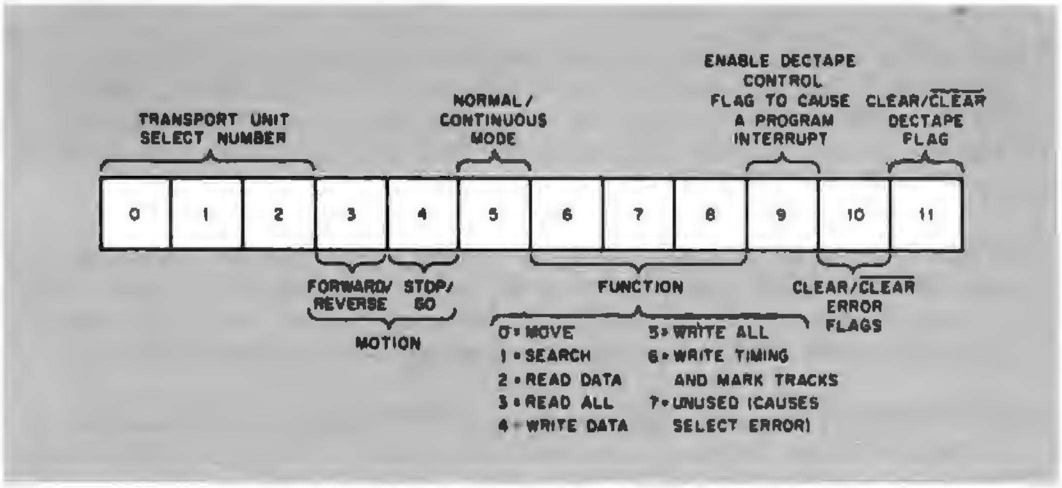


Figure 13 DECtape Status Register A Bit Assignments

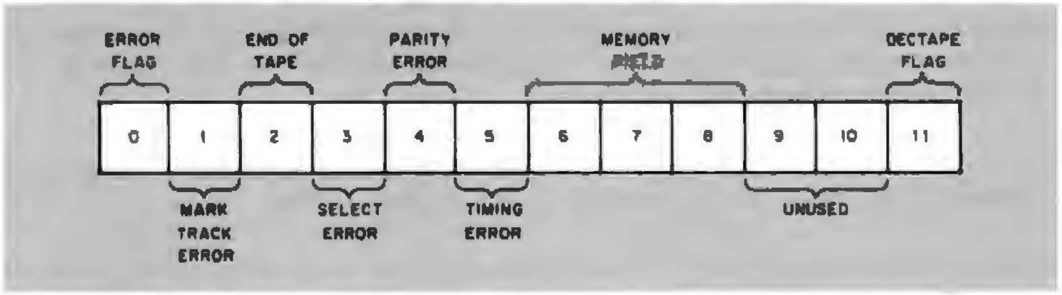


Figure 14 DECtape Status Register B Bit Assignments

Since all data transfers between DECtape and PDP-8 memory are controlled by the data break facility, the program must set the word count (WC) and current address (CA) registers (locations 7754 and 7755 respectively) before a data break. After initiating a DECtape operation, the program should always check for error conditions (a program interrupt would be initiated if the error flag is enabled and if the program interrupt system is enabled). The DECtape system should be started in the search function to locate the block number selected for transfer and then, when the correct block is found, the transfer is accomplished by programmed setting of the WC, CA, and status register A.

When searching, the DECtape control reads only block numbers. These are used by the operating program to locate the correct block number. In NM, the DECtape flag is raised at each block number. In CM, the DECtape flag is raised only after the word count reaches zero. The current address is not incremented during searching and the block number is placed in core memory at the location specified by the content of the CA. Data is transferred to or from the PDP-8 core memory from locations specified by the CA register; which is incremented by one before each transfer.

When the start of the data position of the block is detected, the data flag is raised to initiate a data break request to the data break facility each time the DECtape system is ready to transfer a 12-bit word. Therefore, the main computer program continues running but is interrupted approximately every 133⅓ microseconds for a data break to transfer a word. Transfers occur between DECtape and successive core memory locations specified by the CA. The initial transfer address -1 is stored in the CA by an initializing routine. The number of words transferred is determined only by tape format in NM, or by tape format and the word count in CM. At the conclusion of the data transfer the DT flag is raised and a program interrupt occurs. The interrupt subroutine checks the DECtape error bits to determine the validity of the transfer and either initiates a search for the next information to be transferred or returns to the main program.

During all normal writing transfers, a checksum (the 6-bit logical equivalence of the words in the data block) is computed automatically by the control and is automatically recorded as one of the control words immediately following the data portion of the block. This same checksum is used during reading to determine that the data playback and recognition take place without error.

Any one of the eight tape transports may be selected for use by the program. After using a particular transport, the program can stop the transport currently being used and select another transport, or can select another transport while permitting the original selection to continue running. This is a particularly useful feature when rapid searching is desired, since several transports may be used simultaneously. Caution must be exercised however, for although the original transport continues to run, no tape end detection or other sensing takes place. Automatic end sensing that stops tape motion occurs in all functions, but only in the selected tape transport.

The following is a list of timing considerations for programmed operations. (N_S = the number of block numbers to be read in the search function and continuous mode, counting through the one causing the WCO. Only the block number causing the WCO requests a program interrupt. N_D = number of words transferred ÷ the number of words per block. If the remainder ≠ 0, use the next larger whole number. N_A = number of words transferred.)

Operation	Timing
Answer a data break request	Up to 66 microseconds, ±30%
Word transfer rate	One 12-bit word every 133 microseconds, ±30%
Block transfer rate	One 129-word block every 18.2 milliseconds, ±30%
Start time	200 milliseconds, ±20%
Stop time	150 milliseconds, ±20%
Turn around time	200 milliseconds, ±20%
Change function from search to read data for the current block after DT flag from block number	400 microseconds, ±30%
Change function from search to write data for current block after DT flag from block number	400 microseconds, ±30%
Change function from read data to search for the next block after DT flag from transfer completion	1133 microseconds, ±30%
Change function from write data to search for next block after DT flag from transfer completion	1133 microseconds, ±30%
DECtape flag rises	
In continuous mode	
Move function	Never
Search function	(N_S) x (18.2 milliseconds, ±30%)
Read data function	(N_D) x (18.2 milliseconds, ±30%)
Read all function	(N_A) x (133 microseconds, ±30%)
Write data function	(N_D) x (18.2 milliseconds, ±30%)
Write all function	(N_A) x (133 microseconds, ±30%)
Write T & M function	(N_A) x (133 microseconds, ±30%)

Operation	Timing
In normal mode	
Move function	Never
Search function	Every 18.2 milliseconds, ±30%
Read data function	Every 18.2 milliseconds, ±30%
Read all function	Every 133 microseconds, ±30%
Write data function	Every 18.2 milliseconds, ±30%
Write all function	Every 133 microseconds, ±30%
Write T & M function	Every 133 microseconds, ±30%

Software

Three types of programs have been developed as DECtape software for the PDP-8:

a. Subroutines which the programmer may easily incorporate into a program for data storage, logging, data acquisition, data buffering (queing), etc.

b. A library calling system for storing named programs on DECtape and a means of calling them with a minimal size loader.

c. Programs for preformatting tapes controlled by the content of the switch register to write the timing and mark channels, to writ block formats, to exercise the tape and check for errors, and to provide ease of maintenance.

Program development has resulted in a series of subroutines which read or write any number of DECtape blocks, read any number of 129-word blocks as 128 words (one memory page), or search for any block (used by read and write, or to position the tape). These programs are assembled with the user's program and are called by a JMS instruction. The program interrupt is used to detect the setting of the DECtape flag, thus allowing the main program to proceed while the DECtape operation is being completed. A program flag is set when the operation has been completed. Thus, the program effectively allows concurrent operation of several input/output devices along with operation of the DECtape system. These programs occupy two memory pages ($400_8 = 256_{10}$ words).

The library system has the following features: First and perhaps foremost, the system leaves the state of the computer unchanged when it exits. Second, it calls programs by name from the keyboard and allows for expansion of the program file stored on the tape. Finally, it conforms to existing system conventions, namely, that all of memory, except for the last memory page (7600_8-7777_8), is available to the programmer. This convention ensures that the Binary Loader program (paper tape), and/or future versions of this loader, can reside in memory at all times.

The PDP-8 DECtape library system is loaded by a 17_{10}-instruction bootstrap routine that starts at address 7600_8. This loader calls a larger program into the last memory page, whose function is to preserve on the tape, the content of memory from 6000_8 through 7577_8, and then load the INDEX program and the directory into those same locations. Since the information in this area of memory has been preserved, it can be restored when operations have been completed. The basic system tape contains the following programs:

a. INDEX: Typing this word causes the names of all programs currently on file to be typed out.

b. UPDATE: Allows the user to add a new program to the files. Update queries the operator about the program's name, its starting address, and its location in core memory.

c. GETSYS: Generates a skeleton library tape on a specified DECtape unit.

d. DELETE: Causes a named file to be deleted from the tape.

Starting with the basic library tape, the user can build a complete file of his active programs and continuously update it. One of the uses of the library tape may be illustrated as follows:

A program is written in PDP-8 FORTRAN that is to be used repeatedly. The programmer may call the FORTRAN compiler from the library tape and with it, compile the program, obtaining the object program. The FORTRAN operating system may then be called from the library tape and used to load the object program. At this time the library program UPDATE is called, the operator defines a new program file (consisting of the FORTRAN operating system and the object program), and adds it to the library tape. As a result, the entire operating program and the object program are now available on the DECtape library tape.

The last group of programs, called DECTOG, is a collection of short routines controlled by the content of the switch register. It provides for the recording of timing and mark channels and permits block formats to be recorded for any block length. Patterns may be written in these blocks and then read and checked. Writing and reading is done in both directions and checked. Specified areas of tape may be "rocked" for specified periods of time. A given reel of tape may thus be thoroughly checked before it is used for data storage. These programs may also be used for maintenance and checkout purposes.

CHAPTER 16
AUTOMATIC MAGNETIC TAPE CONTROL TYPE 57A

Functional Description

This control buffers, compiles, synchronizes, and controls data transfers between up to eight magnetic tape transports and the PDP-8, using program interrupts and data breaks. Each transport requires a small interface circuit for connection to the control. The interface required and the characteristics of the transports that can be connected to the 57A are:

Transport	Tape Speed (ips)	Densities (bpi)	Interface
DEC Type 50	75	200/556	Type 520
DEC Type 545	45	200/556/800	Type 521
IBM Model 729II, IV	75	200/556	Type 552
IBM Model 7330*	36	200/556	Type 552
IBM Model 729V, VI	112.5	200/556/800	Type 552

The following functions are controlled by various combinations of IOT instructions:

Write	Space Backward
Write End of File	Rewind
Write Blank Tape	Rewind/Unload
Read	Write Continuous
Read Compare	Read Continuous
Space Forward	

Tape transport motion is governed by one of two control modes: normal, in which tape motion starts upon command and stops automatically at the end of the record; and continuous, in which tape motion starts on command and continues until stopped by the program as a function of synchronizing flags if status conditions appear.

All data transfers are under control of the PDP-8 data break facility; and commands issued during a transfer control,operate, and monitor Type 57A functions by means of the PDP-8 program interrupt facility. Assembled 12-bit PDP-8 data words pass between the computer MB and the control final data buffer register. The core memory address of each word transferred is specified to the computer MA by the control current address register. Use of the program interrupt facility allows the main computer program to continue during long tape operations without running in a loop which waits for tape flags. The program interrupt subroutine for Type 57A loads the AC with numbers, then issues IOT instructions to the control. Specific tape control modes are interpreted from the content of the AC during some IOT instructions. In addition, the current address (CA) register and the word count (WC) registers of the control are loaded from the AC.

Tape functions can be monitored by the program either during or at the end of an operation. They can be altered during operation to a limited degree. The control senses for several types of possible error condition throughout an operation. The results of this sensing can be interrogated by the subroutine at any time.

Two crystal clocks are used to generate one of three character writing rates, depending

*With restrictions

on the density (200, 556, 800) specified by the program. In writing or reading, a composite 12-bit binary word passes between the computer and the control; that is, bits 0 through 5 constitute one tape character, and bits 6 through 11 constitute a second tape character.

In normal operation, six IOT commands initiate reading or writing of one record. When the word count exceeds the number stored in the WC, the transport is stopped and the control is free for another command. In continuous operation, any number of records is written or read without the need for further transport commands except stop.

The following automatic safeguards are inherent in the design of Type 57A:

END POINT

If the end point is reached during reading or writing, the control ignores the end point and finishes the operation (ample tape is allowed). Beyond the end point tape commands specifying forward direction are illegal and the tape will not respond to such commands. If the end point is passed during spacing, the transport is shut down regardless of word count.

LOAD POINT

If the load point is reached during back spacing the transport is stopped regardless of word count. At load point, a space back command is legal, and the tape may be unloaded. When the write command is given at load point, the tape is erased 3 inches beyond the load point before writing the first record. After giving a read command at the load point, the read logic is disabled until the load point marker passes the read head, then the read logic is enabled.

WRITE LOCK RING

Without the write lock ring in the tape reel, writing is illegal and the transport will not respond to a write command.

FORMAT CONTROL

If the PDP-8 halt command is given during normal reading or read comparing, the tape proceeds to the end of record, and the control shuts down the transport. If a halt is given in continuous reading or read comparing the transport will proceed to end of tape and shut down. If a halt command is given in normal spacing, the transport will proceed to EOR and shut down. If halt is given during continuous spacing, the transport will proceed until WC overflows or until it senses a file marker, load point, or end point, then shut down.

If halt is given during writing in the normal mode, the last word to be transferred is written, the rest of the record is written as zeros, and the transport is shut down. If halt is given during writing in the continuous mode, the record is completed; then zeros are written to the end of the tape. If a WC overflow occurs during a normal read or read compare, the transport proceeds to EOR before shutting down.

Instructions

The functions of Type 57A Automatic Magnetic Tape Control are controlled by combinations of the following IOT instructions:

SKIP ON TAPE CONTROL READY (MSCR)

Octal Code: 6701

Event Time: 1

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The Tape Control Ready signal level is sampled, and it is in the binary 1 status, indicating the tape control is free to accept commands, the content of the PC is incremented by one so that the next sequential instruction is skipped.

Symbol: If Tape Control Ready = 1, then PC + 1 => PC

CLEAR AND DISABLE (MCD)

Octal Code: 6702

Event Time: 2

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The job done flag, command register, word count overflow (WCO) flag, and end of record (EOR) flag are cleared. This instruction should be immediately preceded by the two instructions CLA and TAD (4000) to obtain the operation indicated. Both the WCO and EOR flags are connected to the program interrupt facility.

Symbol: Inhibit Job Done Flag
0 => Command Register, WCO, EOR

TAPE SELECT (MTS)

Octal Code: 6706

Event Time: 2, 3

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The job done flag is inhibited from causing a program interrupt and clear the WCO and EOR flags are cleared. Then select the transport unit, the mode of parity, and the bit density from the content of the AC. The AC bit assignments are:

AC1(0) = High sense level
AC1(1) = Low sense level
AC2(0) = 200 or 556 bpi density
AC2(1) = 800 or 550 bpi density
AC8(0) = 200 bpi density
AC8(1) = 556 bpi density

AC2	AC8	Density
0	0	200
0	1	556
1	0	800
1	1	556

AC7(0) =	Even parity (BCD)
AC7(1) =	Odd parity (binary)
AC9-11 =	These three bits select one of eight tape units, addresses 0 through 7.

Symbol: Inhibit Job Done Flag
0 => WCO, EOR
AC1-11 => Select Status

SKIP ON TAPE UNIT READY (MSUR)

Octal Code: 6711

Event Time: 1

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The selected tape transport ready (TTR) status is sampled, and if the transport is ready the content of the PC is incremented by one and the next instruction is skipped. The selected unit must be free before the following MTC command is given.

Symbol: If TTR = 1, then PC + 1 => YPC

TERMINATE CONTINUOUS MODE (MNC)

Octal Code: 6712

Event Time: 2

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The CONTINUE flip-flop in the control is cleared to establish the Normal mode of operation, then the AC is cleared. This command should be immediately preceded by CLA and TAD commands to load the AC with the number 4000 to obtain the operation indicated.

Symbol: 0 => CONTINUE, AC

LOAD TAPE COMMAND (MTC)

Octal Code: 6716

Event Time: 2, 3

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The tape command register is loaded from the content of bits 3 through 6 of the AC. Bit 6 selects the motion mode and bits 3 through 5 are decoded as follows:

AC6(0) =	Normal	
AC6(1) =	Continuous	
AC3-5 =	0 =	No operation
	1 =	Rewind
	2 =	Write
	3 =	Write end of file (EOF)
	4 =	Read compare
	5 =	Read
	6 =	Space forward
	7 =	Space backward

Symbol: AC3-6 => Tape Command Register

SKIP ON WCO FLAG (MSWF)

Octal Code: 6721

Event Time: 1

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The content of the word count overflow (WCO) flag is sensed, and if it contains a 1 the content of the PC is incremented by one so the next instruction is skipped.

Symbol: If WCO = 0, the PC + 1 => PC

DISABLE THE WCO FLAG (MDWF)

Octal Code: 6722

Event Time: 2

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The WCO flag is inhibited from causing a program interrupt.

Symbol: None

CLEAR WCO FLAG (MCWF)

Octal Code: 6722

Event Time: 2

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The WCO flag is cleared. This instruction should be immediately preceded by commands that load the number 2000 into the AC to obtain the indicated operation.

Symbol: 0 => WCO

ENABLE WCO FLAG (MEWF)

Octal Code: 6722

Event Time: 2

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The WCO flag is enabled to cause a program interrupt upon word count overflow. To obtain this operation the MEWF command must be immediately preceded by a sequence that loads the number 4000 into the AC.

Symbol: None

INITIALIZE WCO FLAG (MIWF)

Octal Code: 6722

Event Time: 2

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The WCO flag is initialized. To obtain the operation the MIWF command must be immediately preceded by commands that load the number 6000 into the AC.

Symbol: None

SKIP ON EOR FLAG (MSEF)

Octal Code: 6731

Event Time: 1

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The content of the EOR flag is sensed, and if it contains a 1 the content of the PC is incremented by one so the next instruction is skipped.

Symbol: If EOR = 1, then PC + 1 => PC

DISABLE ERF FLAG (MDEF)

Octal Code: 6732

Event Time: 2

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: Operation of the ERF flag is inhibited.

Symbol: None

CLEAR THE ERF FLAG (MCED)

Octal Code: 6732

Event Time: 2

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The ERF flag is cleared to 0 in preparation for returning to the main program from the program interrupt subroutine. This command must be immediately preceded by commands that load the number 2000 into the AC to obtain the indicated operation.

Symbol: 0 => ERF

ENABLE ERF FLAG (MEEF)

Octal Code: 6732

Event Time: 2

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The ERF flag is enabled. This command must be immediately preceded by commands that load the number 4000 into the AC to obtain the indicated operation.

Symbol: None

INITIALIZE ERF FLAG (MIEF)

Octal Code: 6732

Event Time: 2

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The ERF flag is initialized by clearing and enabling. This command must be immediately preceded by commands that load the number 6000 into the AC to obtain the operation indicated.

Symbol: None

READ TAPE STATUS (MTRS)

Octal Code: 6734

Event Time: 3

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The condition of tape status levels is loaded into the AC. This command must be immediately preceded by a command that clears the AC to obtain a valid information transfer. The AC bit assignments are:

AC0	=	Data request late
AC1	=	Tape parity error
AC2	=	Read compare error
AC3	=	End of file flag set
AC4	=	Write lock ring out
AC5	=	Tape at load point
AC6	=	Tape at end point
AC7	=	Tape near end point (Type 520)
AC7	=	Last operation write (Type 521 and 522 interfaces)
AC8	=	Tape near load point (Type 520)
AC8	=	Write echo (Type 522 interface)
AC8	=	B control using transporting (Type 521 interface with multiplex transport)
AC9	=	Transport rewinding
AC10	=	Tape miss character
AC11	=	Job done flag interrupt

Symbol: Status Levels => AC0-11

CLEAR CURRENT COUNT (MCC)

Octal Code: 6741

Event Time: 1

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The current address register (CA) and word count register (WC) are both cleared.

Symbol: 0 => CA, WC

READ WORD COUNT (MRWC)

Octal Code: 6742

Event Time: 2

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The content of the AC is transferred into the word count register.

Symbol: AC = > WC

READ CURRENT ADDRESS (MRCA)

Octal Code: 6744

Event Time: 3

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The content of the current address register is transferred into the AC. This command does not clear the AC and must be preceded by a command that clears the AC to obtain a valid information transfer.

Symbol: CA V AC => AC

LOAD CURRENT ADDRESS (MCA)

Octal Code: 6745

Event Time: 1, 3

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The current address register and word count register are both cleared, then the content of the AC is transferred into the current address register.

Symbol: 0 => CA, WC
AC => CA

Programming

The following seven control modes are programmed in use of the Type 57A:

WRITE NORMAL

One of two characters, N words and one or two characters, or N words can be written in BCD mode. When writing BCD, convert all characters (00_8) to (12_8). The WCO flag is set during the writing of the next to last word in a record. In a one-word transfer only, the WCO flag is set before the data transfer begins. The ERF flag is set when the EOR (check character) is written. Parity is read and compared while writing.

The data request late bit will be set if the PDP-8 does not transfer a new word to or from the control before another data request is given. When a 522 interface is being used, a write echo status appears if the character zero (00_8) is written BCD.

WRITE END OF FILE (EOF)

The end of the marker is written 17, BCD. It is automatically detected during reading or spacing. One instruction, MTC, initiates this operation, carries it out, and stops the transport WCO does not occur. The ERF flag is set when the EOR (check character) is detected. CA and WC are not modified.

WRITE BLANK TAPE

To write three inches of blank tape, the program gives a write EOF command and then a space backward command. In either case CA and WC are not modified.

READ NORMAL

One or two characters, N words and one or two characters, or N words can be read in either parity mode. The WCO flag is set during the record when the specified word count is exceeded. The ERF flag is set when the EOR (check character) is detected. Parity errors may be read by examining the appropriate tape status bit.

When reading in BCD mode, convert all 12_8 to 00_8. When readin in binary mode and an EOF is detected, the parity error status bit will be set. If while reading, a character does not appear within the allotted time, the miss character status bit will be set.

READ COMPARE

Words from tape may be compared against consecutive on non-consecutive locations in core memory for equality. An inequality sets the read compare error flag and the CA holds the location of the inequality. Read compare is like read, except that WCO occurs before the last word is compared. The ERF is always set at EOR. Should WCO occur before EOR, the ERF will be set upon comparison of the last word and at EOR.

SPACE

Spacing forward or backward one record is automatic and does not modify the CA or WC. Spacing N records in either direction can be done on the Continuous mode, and continues until a WCO occurs or EFF is encountered, whichever comes first. If CA is cleared initially, it will contain the record count and may be examined by the program. The program may command stop prematurely with MNC, after which the tape stops as soon as EOR is seen. The parity error flag will be set if a parity error is detected.

REWIND, REWIND/UNLOAD

Rewind and rewind/unload do not require the use of CA, WC, data interrupt mode, or program interrupt mode. Rewind/unload is selected by specifying rewind and Continuous mode. The transport will not respond to a forward command for 12 milliseconds after the tape has been rewound and stopped at load point.

All operations begin with the program events indicated in the following basic program sequence. When the main program branches to this sequence (having received for example, a high priority data break request from the tape control), the control and transport are interrogated for availability (MSCR, MSUR) and if ready are instructed to carry out the specified task (MTS, MTC). If the task is one of the eight listed in the instruction list under MTC, the MSCR instruction completes the program sequence; if not, the program branches at BEGIN to another routine (write, read, etc.), returning afterwards to WAIT in the basic program.

```
BEGIN,    MSCR             /SKIP IF TAPE CONTROL FREE
          JMP.-1           /TAPE CONTROL NOT FREE, JUMP BACK TO
                           /MSCR INSTRUCTION
          CLA
          TAD  (1A - 1     /LOAD AC WITH INITIAL ADDRESS MINUS ONE
          MCA              /TRANSFER AC TO CA
          CLA
          TAD (- N + 1     /LOAD AC WITH COMPLEMENT OF NUMBER OF
                           /WORDS TO BE TRANSFERRED PLUS ONE
          MRWC             /TRANSFER AC TO WC
          CLA
          TAD (            /LOAD AC WITH SELECTED INFORMATION*
          MTS              /TRANSFER AC TO CONTROL WITH PARITY
                           /DENSITY AND UNIT NUMBER
          MSUR             /SKIP IF TAPE TRANSPORT READY
          JMP.-1           /TRANSPORT NOT READY, JUMP BACK TO
                           /MSUR INSTRUCTION
          MTC              /TRANSFER AC TO CONTROL WITH COMMAND
                           /AND TAPE MOTION MODE
WAIT,     MSCR             /WAIT FOR TAPE FUNCTION TO COMPLETE
          JMP.-1           /TAPE FUNCTION NOT COMPLETE, JUMP
                           /BACK TO MSCR
          HLT              /OPERATION COMPLETION
```

When programming in the interrupt mode, the TCR flag causes an interrupt in the operating program and the flag may be tested by using the MSCR instruction. The TCR flag must be cleared with the MCD command before dismissing the interrupt. WCO and ERF flags must be disabled before dismissing the interrupt with the option of clearing or not clearing the flags.

CHAPTER 17

MAGNETIC TAPE SYSTEM TYPE 580

The Magnetic Tape System Type 580 is a semi-automatic data storage system that can be used with the PDP-8. One tape control and one magnetic tape transport constitute the Type 580 system. Data transmission is under program control, while the timing of motion delays, end-of-record delays, write clock pulses, etc., is automatic. Densities are 200 and 556 bits per inch (selected by program), and maximum transfer rate is 25,000 characters per second. Format is compatible with IBM NRZI in either binary or BCD parity mode.

The control contains a 12-bit data buffer, which accumulates the data word in both reading and writing, and a 9-bit command register. All commands, data, and status indications are transferred to or from the computer accumulator.

The system performs the following functions:

Write
Read forward
Read reverse
Space forward (one or N records)
Space reverse (one or N records)
Rewind
Write real time (one word at a time)
Read real time (one word at a time)

These functions are specified by the presence or absence of ones in the accumulator as shown in the following list:

AC1(0)	=	Sets the SPACE flip-flop
AC3(1)	=	Sets the GO flip-flop
AC4(1)	=	Establish write function
AC5(0)	=	Establish even parity (BCD)
AC5(1)	=	Establish odd parity (binary)
AC6(1)	=	Establish read function
AC7(0)	=	Establish the reverse direction
AC7(1)	=	Establish the forward direction
AC8(0)	=	Select density of 200 BPI
AC8(1)	=	Select density of 556 BPI
AC10(1)	=	Set rewind
AC11(1)	=	Set the REAL TIME flip-flop

When the desired function has been encoded in the accumulator, an IOT instruction is given to initiate the pulses that carry out the function. There are nine IOT microinstructions for the Type 580 system, as follows:

TAPE SYSTEM INITIALIZE FUNCTION AND MOTION (TIFM)

Octal Code: 6707

Event Time: 1, 2, 3

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: All tape control registers are cleared, the command register is loaded from bits 1 through 11 of the AC, and motion delays are initiated. The bit assignments of the command register are:

AC1 = Space
AC3 = Go
AC4 = Write
AC5 = Parity mode (0 = even, 1 = odd)
AC6 = Read
AC7 = Direction (0 = reverse, 1 = forward)
AC8 = Density (0 = 200 BPI, 1 = 556 BPI)
AC10 = Rewind
AC11 = Real time

Symbol: 0 => All Control Registers
AC1-11 => Command Registers

TAPE SYSTEM READ (TSRD)

Octal Code: 6715

Event Time: 1,3

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: Clear the AC, then load the AC from the content of the data buffer (DB) and clear the data flag.

Symbol: 0 => AC
DB => AC
0 => Data Flag

TAPE SYSTEM WRITE (TSWR)

Octal Code: 6716

Event Time: 2, 3

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: Clear the data buffer, then load the data buffer from the content of the AC and clear the data flag.

Symbol: 0 => DB
AC => DB
0 => Data Flag

SKIP ON TAPE SYSTEM DATA FLAG (TSDF)

Octal Code: 6721

Event Time: 1

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The content of the data flag is sampled, and if it contains a 1 the content of the PC is incremented by one so the next instruction is skipped.

Symbol: If Data Flag = 1, then PC + 1 => PC

SKIP ON TAPE SYSTEM END OF RECORD FLAG (TSSR)

Octal Code: 6722

Event Time: 2

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The end of record (EOR) flag is sensed, and if it contains a binary 0 the content of the PC is incremented by one so the next instruction is skipped. The data flag is also sensed, and if it contains a binary 1 the next instruction is skipped.

Symbol: If EOR = 0 or if the Data Flag = 1, then PC + 1 = > PC

TAPE SYSTEM STOP DATA TRANSFER (TSST)

Octal Code: 6724

Event Time: 3

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: This instruction is issued following transmission of the last character in a record. It initiates tape shut down procedures such as writing the longitudinal parity bit, end of record mark, and the 0.75-inch inter-record gap. It also clears the SPACE flip-flop, when the correct number of records to be spaced has been reached.

Symbol: None

TAPE SYSTEM READ STATUS (TSRS)

Octal Code: 6734

Event Time: 3

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The content of the status register is transferred into the AC. The bit assignments are:

AC0(1)	= Parity error
AC1(1)	= Motion delay set
AC2(1)	= Transport is ready
AC3(1)	= Clock delays set
AC4(1)	= End of tape
AC5(1)	= Tape at load point

Symbol: Status Register => AC0-5

TAPE SYSTEM WRITE REAL TIME (TWRT)

Octal Code: 6731

Event Time: 1

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: One character is written on tape. This instruction can be used at any frequency and therefore determines the density of information written on tape.

Symbol: None

TAPE SYSTEM CLEAR PROGRAM INTERRUPT (TCPI)

Octal Code: 6732

Event Time: 2

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The status of the program interrupt flag in the control is sampled, and if it is a 1 (indicating that the 580 system caused a program interrupt) the content of the PC is incremented by one so that the next sequential instruction is skipped. This command clears the program interrupt request flag during a space operation and clears the STOP flip-flop.

Symbol: If Program Interrupt Request Flag = 1, then PC + 1 => PC
0 = > Program Interrupt Request Flag, STOP flip-flop

The following instruction sequence is an example of a routine to write data, and assumes that a previous portion of the program has tested the status of the 580 and that it is ready, etc.

```
/CENTRAL LOOP OF A WRITE DATA ROUTINE
  WRT,      CLA                     /GET FIRST WORD
            TAD I AUTO              /VIA AUTO INDEX REGISTER
            TMP TW2                 /GO TO A WRITE INSTRUCTION

  TW1,      CLA
            TAD I AUTO
            TSDF                    /WAIT FOR THE DATA FLAG
            JMP .- 1

  TW2,      TSWR                    /WRITE
            TSZ CNTR                /COUNT THE NUMBER OF
            JMP TW1                 /WORDS TO BE WRITTEN

  TW3,      CLA
            TSRS                    /READ STATUS
            TSDS                    /WAIT FOR LAST
            JMP .- 1                /WORD TO BE WRITTEN
            TSST                    /STOP DATA
```

The following instruction sequence indicates the core of a subroutine to read data from the Type 580 system. As such, this routine is unencumbered with initializing and testing operations, and presents the basic commands used with the tape system.

```
/CENTRAL LOOP OF A READ DATA ROUTINE

   RED,       CLL CML              /SET THE LINK
              TSDF                 /WAIT FOR FIRST CHAR. OR
              JMP .-1              /WORD TO ENTER BUFFER
              JMP TR2              /GO TO A READ INSTRUCTION

   TR1,       TSSR                 /SKIP IF END OF RECORD
              JMP .-1              /OR A DATA FLAG
              TSDF                 /SKIP IF DATA FLAG = 1
              JMP TR3              /END OF THE RECORD

   TR2,       TSRD                 /READ
              SZL                  /IF LINK SET
              DCA I AUTO           /STORE IN MEMORY VIA AUTO INDEX
              ISZ CNTR             /COUNT THE WORDS STORED
              JMP TR1
              CLL                  /CLEAR LINK TO INHIBIT
              JMP TR1              /STORAGE

   TR3,       CLA
              TSRS                 /READ STATUS
```

CHAPTER 18

DATA COMMUNICATION SYSTEMS TYPE 680

A data communication system consists of a PDP-8 computer with a Data Line Interface Type 681 option, a Serial Line Multiplexer Type 685, and other equipment connected to form a message switching system or to form a data link between serial data transmission equipment and a larger computer. As a message switching system, the 680 system transmits and receives data with up to 128 local or distant Teletype units. As a data link, the 680 system is an economical device for buffering, formatting, and transferring information between a computer and Teletype or other serial processing equipment operating at one or more data speeds. Assuming only minor data handling before transmission to the larger computer, a 680 system can handle up to 128 5-bit Teletype lines at 50 baud. Although the 680 system programming has provision for handling only Teletype lines, programs to pack and unpack massages for other equipment are easily written.

Software for the 680 system is designed to concentrate Teletype data in serial bit format. Although Teletype format is assumed, other data transmission formats that present information in serial format can be used. Subroutines, as presently written, are designed for the 8-bit Teletype code, the 5-bit Teletype code, or a combination of both codes. They also handle mixed speeds on either 8-bit or 5-bit lines with minor changes. Full duplex lines are assumed, but the subroutines operate with half duplex lines, providing the user handles the expected echo.

A Data Communication System Type 680 hardware configuration varies according to the number, type, and distribution of the Teletype units it contains, and upon the use made of the system. Figure 15 shows the basic 680 system configuration, assuming 15 lines: eight local lines, and seven remote lines.

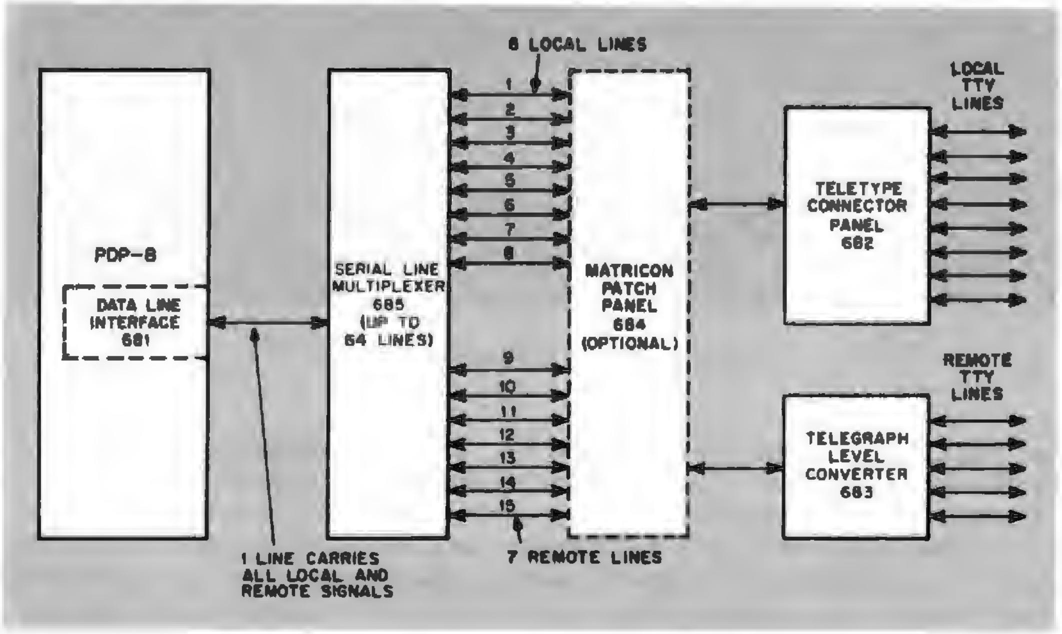

8 LOCAL LINES
LOCAL TTY LINES
PDP-8
DATA LINE INTERFACE 681
SERIAL LINE MULTIPLEXER 685 (UP TO 64 LINES)
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
MATRICON PATCH PANEL 684 (OPTIONAL)
TELETYPE CONNECTOR PANEL 682
REMOTE TTY LINES
TELEGRAPH LEVEL CONVERTER 683
1 LINE CARRIES ALL LOCAL AND REMOTE SIGNALS
7 REMOTE LINES

Figure 15 Data Communication System Block Diagram

Teletype signals from remote stations are transmitted and received by a Telegraph Level Converter Type 683. Interface for local units is provided by a Teletype Connector Panel Type 682. Teletype signals for each station run from the 682 or 683 to a Serial Line Multiplexer Type 685. A Matricon Patchboard Type 684 also provides manual selection of channel connections between the 683 and 685. The 685 consists of a multiplexer for Teletype lines and a clock that causes a program interrupt at a rate eight times the line baud frequency. Single line connections are made between the 685 and the Data Line Interface Type 681, and between the 685 and the normal computer interface. The Type 681 option provides an output instruction to transfer Teletype information from the accumulator to the 685 and provides an input instruction to read Teletype information directly into the computer core memory from the 685. All Teletype information transfers occur serially, one bit at a time.

In any serial data transmission system a word consists of an indication that character transmission is about to start, several bits that specify a character code, and an indication that the character is done. Figure 16 shows the format of 11-unit code Teletype words as a typical word format used in serial data transmission. In such a system, the device receiving the word signal must determine the bit sampling time so that information is transferred reliably, even though the digital information signal is severely integrated (pulse rise and fall times increased and pulses rounded) due to transmission path impedance, and even though no synchronization is provided between sending and receiving units or between information on different lines. In addition, jitter (time displacement) of the information signal caused by the mechanical contact nature of the equipment originating the signal, must be considered when determining the strobe time.

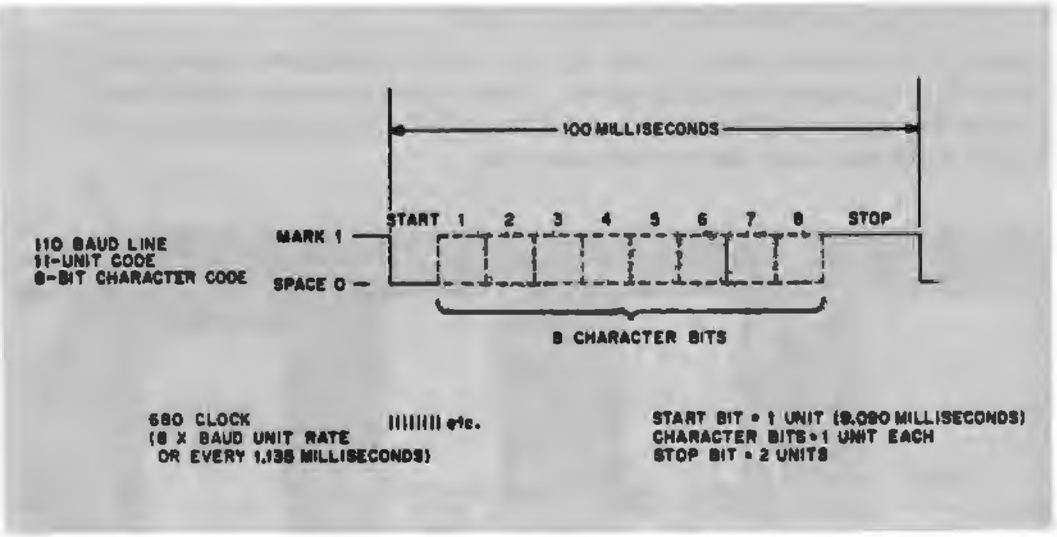


Figure 16 Typical Teletype Line Timing

The Data Communication System Type 680 uses a clock which operates at eight times the bit rate of information on the signal line to determine the sampling time. By counting pulses from this clock, strobe time in receiving and bit timing in transmitting can be controlled within 12.5 percent. Character transmission and reception in the 680 system is controlled by a combination of the hardware and software, providing the most flexibility and economy. This clock in the Serial Line Multiplexer Type 685 requests a program interrupt eight times during each character bit. The program interrupt subroutine counts the clock pulses and strobes a received bit after four clock pulses have occurred since the line became active, thus assuring that the bit is sampled after the middle of the pulse and within 12.5 percent of the center of the pulse. In like manner, clock pulses are counted by the program interrupt subroutine to transmit a bit after eight clock pulses have occurred.

Data Line Interface Type 681

The Type 681 option of the PDP-8 enables use of the computer with a Data Communication System Type 680. The Type 681 option controls and executes transmission and reception of Teletype information between the computer and the Type 680 system. Installation of a Type 681 option in a PDP-8 system adds a Teletype In (TTI) and a Teletype Out (TTO) instruction to the instruction repertoire, and adds two major states to the processor major state generator. The Status (S) and Character (C) states are entered in executing the complex Teletype In instruction.

The TTI and TTO instructions transfer one bit of a Teletype character between the computer and the Serial Line Multiplexer Type 685. These instructions are executed in subroutines entered through the program interrupt subroutine. These subroutines are responsible for determining when a character is completely assembled in the character assembly word (CAW), and for any relocation or translation of assembled characters. Characters are always assembled so that the last bit transmitted shifts into the most significant bit of the CAW and preceding bits are loaded into less significant bits of the CAW, regardless of the Teletype code or transmission path being used.

Teletype In is a complex memory reference instruction which deals with the incoming Teletype line and with two core memory locations. The two locations addressed by the TTI instruction are the next two successive locations following it. These locations contain a line status word (LSW) and a character assembly word (CAW), respectively, so the following sequence is established:

Address	Content
Y	TTI
Y+1	LSW
Y+2	CAW

Bits in the LSW are assigned to record the active/inactive status of the line and serve as a real time clock which determines when line sampling should take place. The format of the LSW is:

$$\underbrace{0}_{\text{Active}}\ \overbrace{1\ \ 2\ \ 3\ \ 4\ \ 5\ \ 6\ \ 7\ \ 8}^{\text{Not Used}}\ \underbrace{9\ \ 10\ \ 11}_{\text{Count}}$$

The CAW stores partially assembled characters. Individual bits on the incoming line enter the most significant bit (bit 0) and are shifted towards the less significant bit (to the right) during the assembly process.

The TTI instruction is normally executed following a program interrupt caused by the clock in the multiplexer. Figure 17 shows the flow diagram of the TTI instruction.

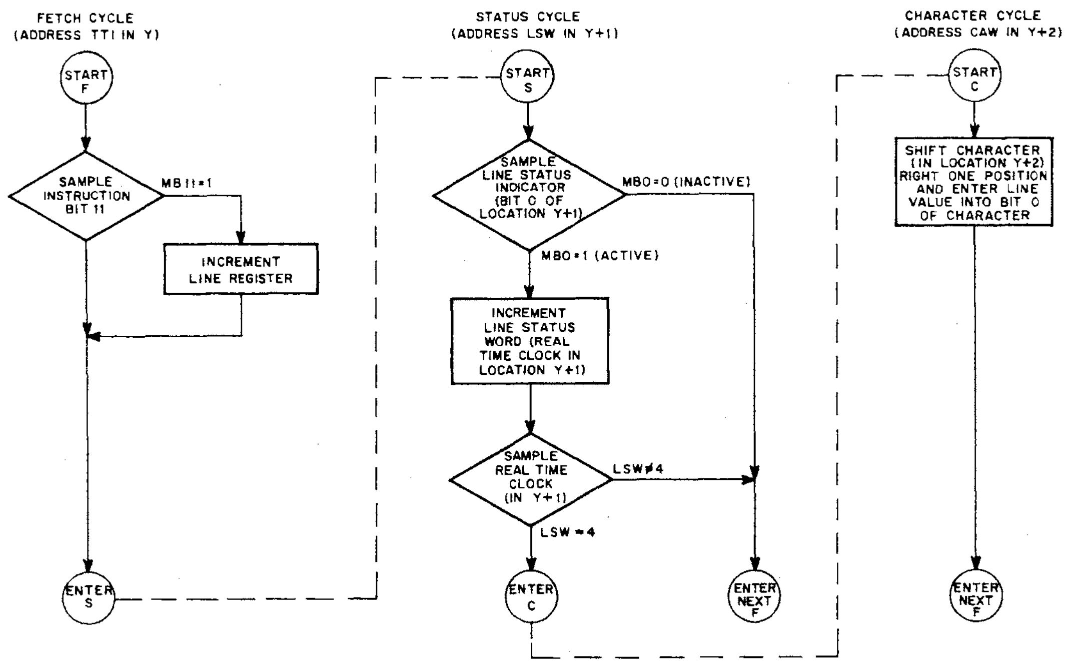

FETCH CYCLE (ADDRESS TTI IN Y)
STATUS CYCLE (ADDRESS LSW IN Y+1)
CHARACTER CYCLE (ADDRESS CAW IN Y+2)
START F
SAMPLE INSTRUCTION BIT 11
MB11=1
INCREMENT LINE REGISTER
ENTER S
START S
SAMPLE LINE STATUS INDICATOR (BIT 0 OF LOCATION Y+1)
MB0=0 (INACTIVE)
MB0=1 (ACTIVE)
INCREMENT LINE STATUS WORD (REAL TIME CLOCK IN LOCATION Y+1)
SAMPLE REAL TIME CLOCK (IN Y+1)
LSW≠4
LSW=4
ENTER C
ENTER NEXT F
START C
SHIFT CHARACTER (IN LOCATION Y+2) RIGHT ONE POSITION AND ENTER LINE VALUE INTO BIT 0 OF CHARACTER
ENTER NEXT F

Figure 17 Teletype In Instruction Flow Diagram

Execution of the TTI instruction causes the next location in memory to be read and examined. This location contains the LSW. The first bit examined is the active bit (bit 0). If bit 0 contains a 0, indicating that the line was inactive when last tested, the current content of the line will be set into the active bit. That is, if a start bit is currently being received, the active indicator bit will be set to 1. If no start bit is being received it remains a 0. In either event the character assembly word is skipped over and the next instruction will be executed. Should examination of the active bit indicate that the line is already active, the count portion of the LSW is incremented by one and, unless the resulting count equals 4, the CAW is skipped. When the count becomes equal to 4, indicating that four clock interrupts have been received since the line first became active or that 4/8 of a bit time has elapsed so the center of the bit has been reached and it should be sampled. Thus the character assembly word is read, its content is shifted right one position, and the bit presently being received on the line is set into the leftmost position of the CAW. After the first bit has been received eight clock periods occur before the count is again equal to 4 and thus each bit in the serial train is sampled within 12.5% of the center of the bit.

The Teletype Out instruction affects only the content of the accumulator and the outgoing line. It is executed during a single memory cycle. It shifts the content of the accumulator one position to the right and transmits the least significant bit on the outgoing line. Since a bit is transmitted every time this instruction is executed it should be programmed to occur only after eight clock interrupts have been received since the last output.

These instructions are used only in subroutines and are not used in the main program. The following explanation of their use is for description only.

The Teletype In command brings the bits coming over the line into memory and assembles the bits into one Teletype character. The TTI command uses three memory locations as follows:

```
TTI
0         /status and counter word (LSW)
2000      /character assembly word (for 8-bit code) (CAW)
```

The program then returns to TTI + 3

The character assembly word is preset so that 1 appears in bit 11 when the entire character, including one stop bit, has been shifted in. The subroutines, finding a 1 in bit 11, assume that an entire character has been read, place the character in its own internal buffer together with the line number it came from, and reinitializes the TTI command by resetting the line status word to 0 and the character assembly word to the proper number. At each clock pulse the program only checks 1/8th of the lines (1/4 for 5-bit codes) for completion.

Unlike the TTO command, the TTI command is executed for all lines at each clock-produced program interrupt. However, once the incoming character is started (i.e., bit 0 of the LSW = 1) the first bit (the start code) is read at the fourth pulse and each succeeding bit is read at the eighth pulse thereafter, thus guaranteeing that the bit is read at the optimum time.

The Teletype Out command shifts the content of the accumulator right one position, sends the previous content of bit 11 to the Teletype line specified by the line select register, and brings a 0 into bit 0 of the accumulator. The program sequence to transmit a word from core memory to a Teletype unit might be as follows:

```
TAD   CHAR     /GET CHARACTER TO TRANSMIT
TTO            /SHIFT AND TRANSMIT ONE BIT
DCA   CHAR     /SAVE REMAINDER OF CHARACTER
```

This sequence assumes that the line select register has been loaded with the correct line number using commands for the Serial Line Multiplexer Type 685.

Serial Line Multiplexer Type 685

The 685 is simply a switch which allows the 681 to be connected to any one of 64 Teletype lines. To select a line, the accumulator is set to the number of the desired line, and its content is then transferred into the line select register of the 685 by an IOT command. The line select register (LSR) may be loaded at any time with a program-selected address, or can be incremented by a command which may be microprogrammed with the TTI or TTO instructions to scan all lines in numbered sequence. Incrementing is used for high-speed sequential scans. This unit also contains a flip-flop for each outgoing line. This flip-flop is set or cleared by a TTO instruction and holds the line in the proper state until the next TTO instruction is executed.

Instructions

All instructions for the 680 system contain an operation code of 6, indicating that they are IOT commands. Commands which are associated with the 681 transfer one bit of a character with the computer and have a select code of 40. These commands are functionally memory reference instructions used to perform an input/output transfer operation. Commands associated with the line select register of the 685 use select codes 40 and 41. Commands associated with the clocks of the 685 use select code 42 through 45. All commands using select codes of 41 and 42 use the IOP pulses and are true IOT instructions. The instructions for the 680 system are:

TELETYPE INCREMENT (TTINCR)

Octal Code: 6401

Event Time: Not applicable in the normal sense of IOT event times. However it can be considered event time 1, since it is executed before all other operations in the TTI or TTO commands with which it can be combined.

Indicators: IOT, FETCH

Execution Time: 1.5 microseconds when performed individually, or equal to the execution time of other commands when microprogrammed.

Operation: The content of the line select register (LSR) in the Serial Line Multiplexer is incremented by one to address the next sequentially numbered line unit. This operation occurs at T1 time of the Fetch cycle.

Symbol: LSR + 1 => LSR

TELETYPE IN (TTI)

Octal Code: 6402

Event Time: Not applicable

Indicators: IOT, FETCH

Execution Time: 3.0 or 4.5 microseconds

Operation: Three core memory locations are required by the TTI instruction. The first location contains the TTI instruction, and the two succeeding locations contain a line status word (LSW) and a character assembly word (CAW), respectively. Bit 0 of the LSW records the active/inactive status of the selected Teletype line, and bits 9 through 11 of the LSW serve as a real time clock to determine the bit assembly time for the CAW. Both of these words should be cleared prior to the first use of the TTI instruction in a subroutine. The TTI instruction checks the status of the selected line and the number in the real time clock. If the line is active and the clock indicates the center of a bit has passed, one bit of the Teletype line is shifted into the CAW.

The TTI instruction is executed in two or three computer cyles. The first cycle is in the Fetch state to read the instruction from core memory and to establish the next sequential core memory location as the address to be read during the next cycle. By placing a 1 in bit 11, this instruction can be microprogrammed to increment the content of the flip-flop line register of the Serial Line Multiplexer Type 685 during the Fetch cycle.

The second cycle is a Status state in which the LSW is read, the active/inactive status of the line is checked, the timing of the current bit is checked, and (based on these conditions) the inactive status of the line is recorded in MB0 and the program advances to the next instruction, the real time clock count is incremented in the LSW and the program advances to the next instruction, or the real time clock count is incremented and the third cycle is initiated.

The active/inactive status of the Teletype line is checked by sampling the condition of bit 0 of the LSW. If MB0(0), indicating that the line is inactive (not transmitting a character) the LSW is shifted one position to the right in the MB, and the complement of the Teletype line is set into MB0. Therefore, if the line is now active, a 1 is set into MB0 and will be read during the Status cycle of the next TTI instruction. The program count is then incremented by one to skip over the CAW, the LSW is restored to core memory, the MB is cleared, and (providing no break request had been received) the Fetch state is entered to fetch the next instruction.

If the MB0(1) at the beginning of the Status cycle, the LSW is incremented by one to advance the real time clock and the LSW number is sampled. If LSW≠3 it is too early to sample the active line so the program count is incremented to skip over the CAW, the LSW is restored to core memory, the MB is cleared, and the program advances to the Fetch state for the next instruction. If LSW = 4 after incrementation, the LSW is rewritten in memory and the major state generator (MSG) is set to the Character state to strobe the line into the CAW during the next cycle.

The third cycle is a Character state in which the CAW is read into the MB from core memory, the character is shifted right one position with the line bit being shifted into MB0, then the CAW is rewritten in memory. The program then advances to the Fetch state for the next instruction.

Symbol: Status state
If MB0(0), then line shifted into LSW and F=> MSG for next instruction.
If MB0(1), and MB≠3, then LSW + 1 => LSW and F => MSG for next instruction.
If MB0(1) and MB = 3, then LSW + 1 => LSW and C => MSG to continue TTI instruction in next cycle.

Character state
Line shifted into CAW and F = > MSG for next instruction.

TELETYPE OUT (TTO)

Octal Code: 6404

Event Time: Not applicable

Indicators: IOT, FETCH

Execution Time: 1.5 microseconds

Operation: This instruction must be preceded by a command sequence (such as CLA and TAD) that loads the AC with the character to be (or being) transferred to the external Teletype equipment. The TTO instruction clears the L, shifts the content of the AC and the L one position to the right, then transfers the bit contained in AC11 to the selected Teletype line.

Symbol: 0 => L
L => AC0 and ACj => ACj + 1, then
AC11 => Selected Line

CLEAR LINE SELECT REGISTER (TTCL)

Octal Code: 6411

Event Time: 1

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The line select register is cleared, so line 0 is addressed.

Symbol: 0 => LSR

LOAD LINE SELECT REGISTER (TTSL)

Octal Code: 6412

Event Time: 2

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The line select register is set by an OR transfer from the content of bits 5 through 11 of the accumulator, then the accumulator is cleared.

Symbol: AC5-11 V LSR => LSR, then
0 => AC

READ LINE SELECT REGISTER (TTRL)

Octal Code: 6414

Event Time: 3

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The content of the line select register is loaded into bits 5 through 11 of the accumulator by an OR transfer.

Symbol: LSR V AC5-11 => AC5-11

SKIP ON CLOCK 1 FLAG (TTSKP)

Octal Code: 6421

Event Time: 1

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The content of clock 1 flag of the Serial Line Multiplexer is sampled, and if it contains a 1 (indicating that a clock pulse has occurred and the flag has been enabled to request a program interrupt) the content of the program counter is incremented by 1 to skip the next sequential instruction. If the skip occurs clock 1 caused a program interrupt if the interrupt system was enabled when the clock pulse occurred.

Symbol: If Clock 1 Flag = 1, then PC + 1 => PC

TURN ON CLOCK 1 (TTXON)

Octal Code: 6422

Event Time: 2

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The CLOCK 1 ENABLE flip-flop is set and the clock 1 flag is cleared. When the CLOCK 1 ENABLE flip-flop is set the next clock pulse sets the clock 1 flag and requests a program interrupt.

Symbol: 1 = > Clock 1 Enable
0 = > Clock 1 Flag

TURN OFF CLOCK 1 (TTXOFF)

Octal Code: 6424

Event Time: 3

Indicators: IOT, FETCH, PAUSE

Execution Time: 3.75 microseconds

Operation: The CLOCK 1 ENABLE flip-flop is cleared and the clock 1 flag is cleared. When the CLOCK 1 ENABLE flip-flop is cleared the clock 1 flag can not be set by the clock, and can not request a program interrupt or be skipped upon. The clock is unaffected and continues to run, but all operations caused by clock pulses are disabled.

Symbol: 0 => Clock 1 Enable
0 => Clock 1 Flag

When the system handles multiple-baud frequencies additional clocks and instructions are provided. Instructions similar to TTSKP, TTXON, and TTXOF use select code 43 for clock 2 and use select code 44 for clock 3.

Software

Subroutines for the 680 system, as presently coded, occupy 400_8 core memory locations plus locations for internal buffering of the input and output characters and for thet TTI instructions. In addition, autoindex registers and core memory locations in page 0 are required as specified in the following list:

8-Bit	5-Bit	5-Bit (2nd speed)	Meaning
TT8BGN	TT5BGN	TT4BGN	Beginning of subroutine
T8AX1	T5AX1	T4AX1	Autoindex register
T8AX2	T5AX2	T4AX2	Autoindex register
T8AX3			
T8AX3	T5AX3	T4AX3	Autoindex register
	T5AX4	T4AX4	Autoindex register (5-bit only)
TT8PG0	TT5PG0	TT4PG0	Start of area in page 0
T8O8F2	T5OBF2	T4OBF2	Start of 2nd output buffer (length = N)
T81BF	T51BF	T51BF	Start of input buffer (length = 2N)
T81N	T51N	T51N	Start of TT1 area (length = 3N + 1)
TTCHAR	TTCHAR	TTCHAR	Character area (appears only once)

The total amount of core memory used by 680 subroutines, including the tags and autoindex registers in page 0, is as follows:

$$422_8 + 7N \text{ (for 8-bit)}$$
or
$$438_8 + 7N \text{ (for 5-bit)}$$

where N is the number of lines specified to the subroutines. Within limits, the programs can be stored anywhere in the PDP-8 core memory.

If the 5-bit subroutines are being used all of the tags mentioned should substitute 5s for 8s shown. If both 8-bit and 5-bit systems are being used, both sets of subroutines are necessary and all tags and memory requirements must be duplicated for the second system. At present, coding is available for a single 8-bit system and for two different 5-bit systems to allow the programmer to assemble all of the necessary components with a main program at one time.

Percentages of machine time used in the average case for various types of systems are presented in the following list. Any additional features which may be required for the Teletype handling must be added to these times. The formulas for calculating these times are included so that times for systems with an intermediate number of lines or with combinations of lines can be calculated. For combined systems, add the percentages for each component.

Number of lines	8-Bit 110 Baud*	5-Bit 50 Baud**	5-Bit 75 Baud***
32	34.1%	20.0%	30.0%
64	57.7%	35.1%	52.7%
96	81.3%	50.3%	75.5%
128	104.9%	65.5%	98.3%

*Formula Used: Where N = the number of lines, the 8-bit subroutines require an average of 8.38N +119.5 microseconds.

**Formula Used: Where N = the number of lines, the 5-bit subroutines require an average time of 11.85N +120 microseconds. Clock flags (at 50 baud) occur every 2500 microseconds.

***Formula Used: The percentages for 75 baud are merely 1.5 x 50 baud rate. Clock flags occur every 1667 microseconds.

For further information, refer to DEC Program Library documents DEC-35-S-A and DEC-35-S-B.

SECTION C

OPERATION

CHAPTER 1

STANDARD PDP-8 OPERATION

Controls and Indicators

Manual control of the PDP-8 is exercised by means of keys and switches on the operator console. Visual indications of the machine status and the content of major registers and control flip-flops is also given on this console. Indicator lamps light to denote the presence of a binary 1 in specific register bits and in control flip-flops. The function of these controls and indicators is listed in Table 3, and their location is shown in Figure 18. The functions of all controls and indicators of the Model 33 ASR Teletype unit are described in Table 4, as they apply to operation of the computer. The Teletype console is shown in Figure 19.

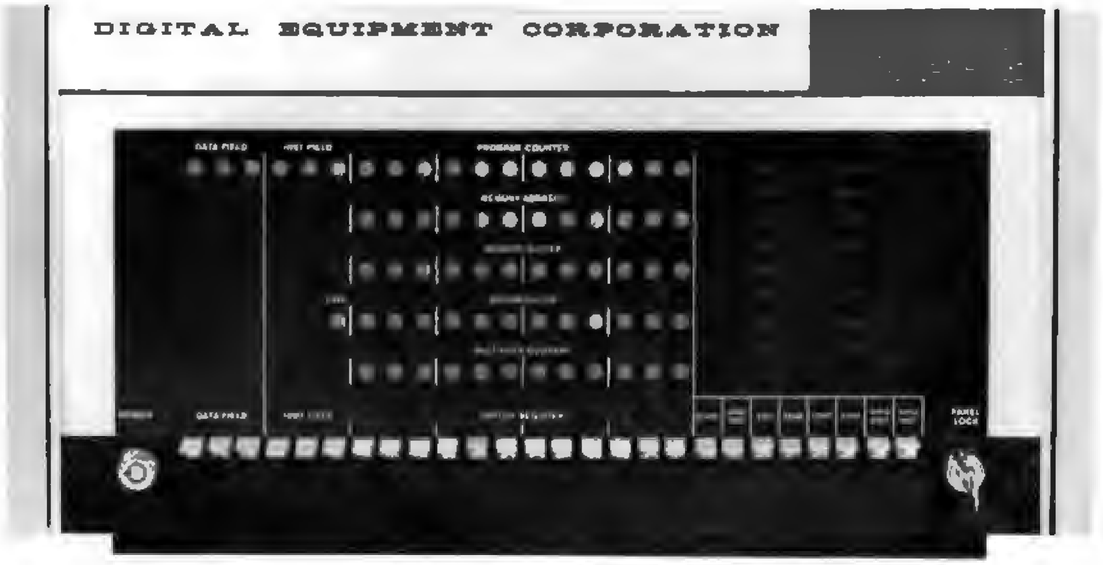


Figure 18 PDP-8 Operator Console

TABLE 3 OPERATOR CONSOLE CONTROLS AND INDICATORS

Control or Indicator	Function
PANEL LOCK switch	With this key-operated switch turned clockwise, all keys and switches except the SWITCH REGISTER switches on the operator console are disabled. In this condition the program can not be disturbed by inadvertent key operation. The program can, however, monitor the content of the SR by execution of the OSR instruction. With this switch turned counterclockwise all operator console keys and switches function normally.
POWER switch	In the counterclockwise position this key-operated switch removes primary power from the computer, and in the clockwise position it applies power.

TABLE 3 OPERATOR CONSOLE CONTROLS AND INDICATORS (continued)

Control or Indicator	Function
START key	Starts the computer program by turning off the program interrupt circuits; clearing the AC, L, MB, and IR; setting the Fetch state, transferring the content of the PC into the MA; and setting the RUN flip-flop. Therefore, the word stored at the address currently held by the PC is taken as the first instruction.
LOAD ADDRESS key	Pressing this key sets the content of the SR into the PC, sets the content of the INST FIELD switches into the IF, and sets the content of the DATA FIELD switches into the DF.
DEPOSIT key	Lifting this key sets the content of the SR into the MB and core memory at the address specified by the current content of the PC. This operation is performed by setting the Execute state and forcing a DCA instruction. The content of the PC is then incremented by one, to allow storing of information in sequential memory addresses by repeated operation of the DEPOSIT key.
EXAMINE key	Pressing this key sets the content of core memory at the address specified by the content of the PC into the MB and AC. This operation is performed by clearing the AC, setting the Execute state, and forcing a TAD instruction. The content of the PC is then incremented by one to allow examination of the content of sequential core memory addresses by repeated operation of the EXAMINE key.
CONTINUE key	Pressing this key sets the RUN flip-flop to continue the program in the state and instruction designated by the lighted console indicators, at the address currently specified by the PC.
STOP key	Causes the RUN flip-flop to be cleared at the end of the cycle in progress at the time the key is pressed.
SINGLE STEP switch	The switch is off in the down position. In the up position the switch causes the RUN flip-flop to be cleared to disable the timing circuits at the end of one cycle of operation. Thereafter, repeated operation of the CONTINUE key steps the program one cycle at a time so that the content of registers can be observed in each state.

TABLE 3 OPERATOR CONSOLE CONTROLS AND INDICATORS (continued)

Control or Indicator	Function
SINGLE INSTRUCTION switch	The switch is off in the down position. In the up position the switch causes the RUN flip-flop to be cleared at the end of the next instruction execution. When the computer is started by means of the START or CONTINUE key, this switch causes the RUN flip-flop to be cleared at the end of the last cycle of the current instruction. Therefore, repeated operation of the CONTINUE key steps the program one instruction at a time.
SWITCH REGISTER switches	Provide a means of manually setting a 12-bit word into the machine. Switches in the up position; corresponds to binary ones, down to zeros. The content of this register is loaded into the PC by the LOAD ADDRESS key or into the MB and core memory by the DEPOSIT key. The content of the SR can be set into the AC under program control by means of the OSR instruction.
DATA FIELD indicators and switches*	The indicators denote the content of the data field register (DF) and the switches serve as an extension of the SR to load the DF by means of the LOAD ADDRESS key. The DF determines the core memory field of data storage and retrieval.
INST FIELD indicators and switches*	The indicators denote the content of the instruction field register (IF) and the switches serve as an extension of the SR to load the IF by means of the LOAD ADDRESS key. The IF determines the core memory field from which instructions are to be taken.
PROGRAM COUNTER indicators	Indicate the content of the PC. When the machine is stopped the content of the PC indicates the core memory address of the first instruction to be executed when the START or CONTINUE key is operated. When the machine is running the content of the PC indicates the core memory address of the next instruction.
MEMORY ADDRESS indicators	Indicate the content of the MA. Usually the content of the MA denotes the core memory address of the word currently or previously read or written. After operation of either the DEPOSIT or EXAMINE key, the content of the MA indicates the core memory address at which information was just written or read.

* Activated only on systems containing the Type 183 Memory Extension Control option.

TABLE 3 OPERATOR CONSOLE CONTROLS AND INDICATORS (continued)

Control or Indicator	Function
MEMORY BUFFER indicators	Indicate the content of the MB. Usually the content of the MB designates the word just read or written at the core memory address held in the MA.
ACCUMULATOR indicators	Indicates the content of the AC.
LINK indicator	Indicates the content of the L.
MULTIPLIER QUOTIENT indicators*	Indicate the content of the multiplier quotient (MQ). The MQ holds the multiplier at the beginning of a multiplication and holds the least significant half of the product at the conclusion. It holds the least significant half of the dividend at the start of a division and at the end holds the quotient.
Instruction indicators (AND, TAD, ISZ, DCA, JMS, JMP, IOT, OPR)	Indicate the decoded output of the IR as the instruction currently in progress.
FETCH, EXECUTE DEFER, BREAK indicators	Indicate the primary control state of the machine and that the current memory cycle is a Fetch, Execute, Defer or Break cycle, respectively.
ION indicator	Indicates the 1 status of the INT. ENABLE flip-flop. When lit, the program in progress can be interrupted by receipt of a Program Interrupt Request signal from an I/O device.
PAUSE indicator	Indicates the 1 status of the PAUSE flip-flop when lit. An IOT instruction sets the PAUSE flip-flop at T1 time to initiate operation of the IOP generator and to inhibit advance of the normal timing generator. When IOP generator operation is completed (approximately 2.5 microseconds later), a T2 pulse is generated and the PAUSE flip-flop is cleared to enable advance of the timing generator in synchronism with the basic computer clock.
RUN indicator	Indicates the 1 status of the RUN flip-flop. When lit, the internal timing circuits are enabled and the machine performs instructions.

*Activated only on systems containing the Type 182 Extended Arithmetic Element option.

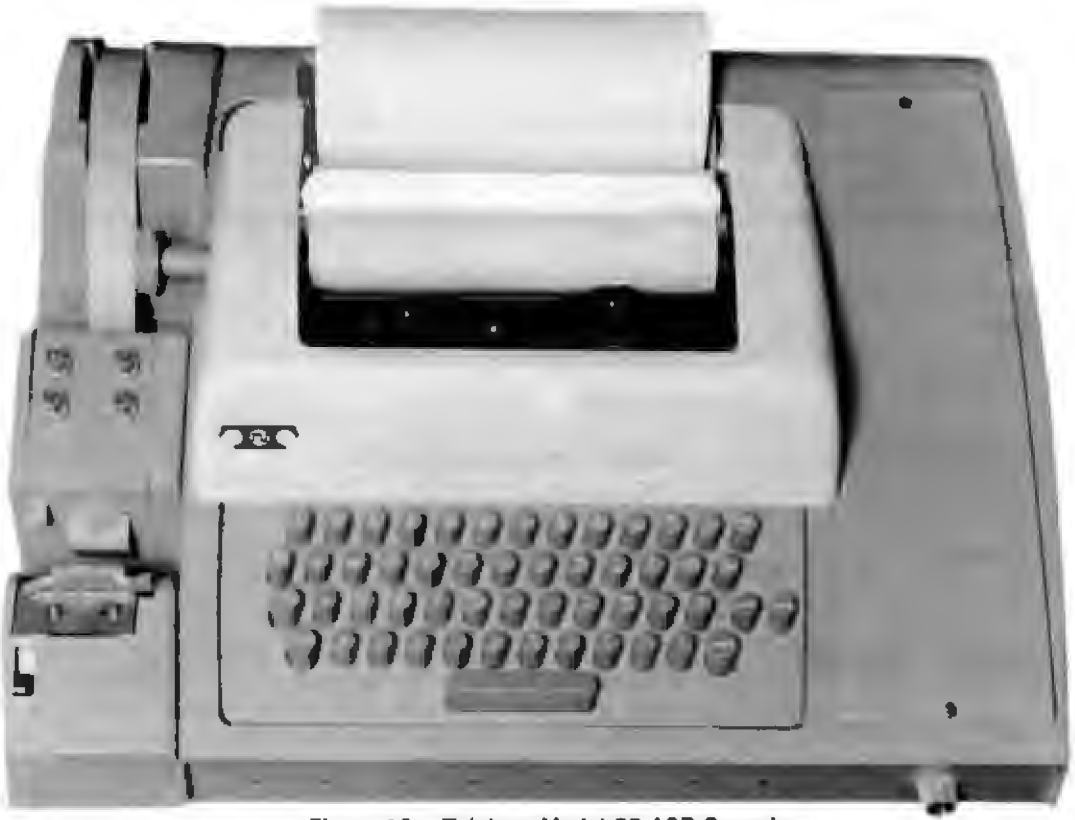


Figure 19 Teletype Model 33 ASR Console

TABLE 4 TELETYPE CONTROLS AND INDICATORS

Control or Indicator	Function
REL. pushbutton	Disengages the tape in the punch to allow tape removal or tape loading.
B. SP. pushbutton	Backspaces the tape in the punch by one space, allowing manual correction or rub out of the character just punched.
OFF and ON pushbuttons	Control use of the tape punch with operation of the Teletype keyboard/printer.
START/STOP/FREE switch	Controls use of the tape reader with operation of the Teletype. In the lower FREE position the reader is disengaged and can be loaded or unloaded. In the center STOP position the reader mechanism is engaged but de-energized. In the upper START position the reader is engaged and operated under program control.
Keyboard	Provides a means of printing on paper in use as a typewriter and punching tape when the punch ON pushbutton is pressed, and provides a means of supplying input data to the computer when the LINE/OFF/LOCAL switch is in the LINE position.
LINE/OFF/LOCAL switch	Controls application of primary power in the Teletype and controls data connection to the processor. In the LINE position the Teletype is energized and connected as an I/O device of the computer. In the OFF position the Teletype is de-energized. In the LOCAL position the Teletype is energized for off-line operation, and signal connections to the processor are broken. Both line and local use of the Teletype require that the computer be energized through the POWER switch.

Operating Procedures

Many means are available for loading and unloading PDP-8 information. The means used are, of course, dependent upon the form of the information, time limitations, and the peripheral equipment connected to the computer. The following procedures are basic to any use of the PDP-8, and although they may be used infrequently as the programming and use of the computer become more sophisticated, they are valuable in preparing the initial programs and learning the function of machine input and output transfers.

MANUAL DATA STORAGE AND MODIFICATION

Programs and data can be stored or modified manually by means of the facilities on the operator console. Chief use of manual data storage is made to load the readin mode loader program into the computer core memory. The readin mode (RIM) loader is a program used to automatically load programs into PDP-8 from perforated tape in RIM format. This program and the RIM tape format are described in Appendix 6 of this handbook and in Digital Program Library descriptions. The RIM program listed in the Appendix can be used as an exercise in manual data storage. To store data manually in the PDP-8 core memory:

1. Turn the PANEL LOCK switch counterclockwise and turn the POWER switch clockwise.
2. Set the bit switches of the SWITCH REGISTER (SR) to correspond with the address bits of the first word to be stored. Press the LOAD ADDRESS key and observe that the address set by the SR is held in the PC, as designated by lighted PROGRAM COUNTER indicators corresponding to switches in the 1 (up) position and unlighted indicators corresponding to switches in the 0 (down) position.
3. Set the SR to correspond with the data or instruction word to be stored at the address just set into the PC. Lift the DEPOSIT key and observe that the MB, and hence the core memory, hold the word set by the SR.

   Also, observe that the PC has been incremented by one so that additional data can be stored at sequential addresses by repeated SR setting and DEPOSIT key operation.

To check the content of an address in core memory, set the address into the PC as in step 2, then press the EXAMINE key. The content of the address is then designed by the MEMORY BUFFER and ACCUMULATOR indicators. The content of the PC is incremented by one with operation of the EXAMINE key, so the content of sequential addresses can be examined by repeated operation after the original (or starting) address is loaded. The content of any address can be modified by repeating both steps 2 and 3.

LOADING DATA UNDER PROGRAM CONTROL

Information can be stored or modified in the computer automatically only by enacting programs previously stored in core memory. For example, having the RIM loader stored in core memory allows RIM format tapes to be loaded as follows:

1. Turn the PANEL LOCK switch counterclockwise and turn the POWER switch clockwise.
2. Set the Teletype LINE/OFF/LOCAL switch to the LINE position.

3. Load the tape in the Teletype reader by setting the START/STOP/FREE switch to the FREE position, releasing the cover guard by means of the latch at the right, loading the tape so that the sprocket wheel teeth engage the feed holes in the tape, closing the cover guard, and setting the switch to the STOP position. Tape is loaded in the back of the reader so that it moves toward the front as it is read. Proper positioning of the tape in the reader finds three bit positions being sensed to the left of the sprocket wheel and five bit positions being sensed to the right of the sprocket wheel.

4. Load the starting address of the RIM loader program (not the address of the program to be loaded) into the PC by means of the SR and the LOAD ADDRESS key.

5. Press the computer START key and set the 3-position Teletype reader switch to the START position. The tape will be read automatically.

Automatic storing of the binary loader (BIN) program is performed by means of the RIM loader program as previously described. With the BIN loader stored in core memory, program tapes assembled in the program assembly language (PAL III) binary format can be stored as described in the previous procedure except that the starting address of the BIN loader (usually 7777) is used in step 4. When storing a program in this manner, the computer stops and the AC should contain all zeros if the program is stored properly. If the computer stops with a number other than zero in the AC, a checksum error has been detected. When the program has been stored, it can be initiated by loading the program starting address (usually designated on the leader of the tape) into the PC by means of the SR and LOAD ADDRESS key, then pressing the START key.

OFF-LINE TELETYPE OPERATION

The Teletype can be used separately from the PDP-8 for typing, punching tape, or duplicating tapes. To use the Teletype in this manner:

1. Assure that the computer PANEL LOCK switch is turned counterclockwise and turn the POWER switch clockwise.

2. Set the Teletype LINE/OFF/LOCAL switch to the LOCAL position.

3. If the punch is to be used, load it by raising the cover, manually feeding the tape from the top of the roll into the guide at the back of the punch, advancing the tape through the punch by manually turning the friction wheel, and then closing the cover. Energize the punch by pressing the ON pushbutton, and produce about two feet of leader. The leader-trailer can be code 200 or 377. To produce the code 200 leader, simultaneously press and hold the CTRL and SHIFT keys with the left hand; press and hold the REPT key; press and release the @ key. When the required amount of leader has been punched release all keys. To produce the 377 code, simultaneously press and hold both the REPT and RUB OUT keys until a sufficient amount of leader has been punched.

If an incorrect key is struck while punching a tape, the tape can be corrected as follows: if the error is noticed after typing and punching N characters, press the punch B. SP. (backspace) pushbutton N + 1 times and strike the keyboard RUB OUT key N + 1 times. Then continue typing and punching with the character which was in error.

To duplicate and obtain a listing of an existing tape: Perform the procedure under the current heading. Then load the tape to be duplicated as described in step 2 of the procedure under Loading Data Under Program Control. Initiate tape duplication by setting the reader START/STOP/FREE switch in the START position. The punch and teleprinter stop when the tape being duplicated is completely read.

Corrections to insert or delete information on a perforated tape can be made by duplicating the correct portion of the tape, and manually punching additional information or inhibiting punching of information to be deleted. This is accomplished by duplicating the tape and carefully observing the information being typed as the tape is read. In this manner the reader START/STOP/FREE switch can be set to the STOP position just before the point of the correction is typed. Information to be inserted can then be punched manually by means of the keyboard. Information can be deleted by pressing the punch OFF pushbutton and operating the reader until the portion of the tape to be deleted has been typed. It may be necessary to backspace and rub out one or two characters on the new tape if the reader is not stopped precisely on time. The number of characters to be rubbed out can be determined exactly by the typed copy. Be sure to count spaces when counting typed characters. Continue duplicating the tape in the normal manner after making the corrections.

New, duplicated, or corrected perforated tapes should be verified by reading them off line and carefully proofreading the typed copy.

SECTION D

INTERFACE AND INSTALLATION

CHAPTER 1

PDP-8 INPUT/OUTPUT FACILITIES

Since the processing power of a computer system depends largely upon the range and number of peripheral devices that can be connected to it, the PDP-8 has been designed to interface readily with a broad variety of external equipment. Section D of this handbook defines the interface characteristics of the computer to allow the reader to design and implement any electrical interfaces required to connect devices to the PDP-8. Chapters 2 and 3 functionally describe the logic circuit elements involved in programmed data transfers and data break transfers, respectively. Chapter 4 gives detailed circuit information on the modules used in the computer interface and that are available for use in special device interfaces. Chapter 5 lists connection point, module type, module location, etc., for each interface signal; gives detailed loading and driving characteristics for each module in the computer interface; then presents some general rules and characteristics to be considered in selecting or designing electrical circuits to be connected to the PDP-8. Chapter 6 presents information for planning the installation of a basic PDP-8 and the available standard optional equipment.

The simple I/O techniques of the PDP-8, the availability of DEC's FLIP CHIP logic circuit modules, and DEC's policy of giving assistance wherever possible allow inexpensive, straight-forward device interfaces to be realized. Should questions arise relative to the computer interface characteristics, the design of device interfaces using DEC modules, or installation planning, customers are invited to telephone the main plant in Maynard, Massachusetts, or any of the sales offices. Digital Equipment Corporation makes no representation that the interconnection of its circuit modules in the manner described herein will not infringe on existing or future patent rights. Nor do the descriptions contained herein imply the granting of license to use, manufacture, or sell equipment constructed in accordance therewith.

The basic PDP-8 contains a processor and core memory composed of FLIP CHIP circuit modules. These hybrid silicon circuits have an operating temperature range exceeding the limits of 32° to 130°F, so no air-conditioning is required at the computer site. Standard 115v, 60-cps power operates an internal solid-state power supply that produces all required voltages and currents.

High-capacity, high-speed I/O capabilities of the PDP-8 allow it to operate a variety of peripheral devices in addition to the standard Teletype keyboard/printer, tape reader, and tape punch. DEC options, consisting of an interface and normal data processing equipment, are available for connecting into the computer system. These options include card equipment, line printers, magnetic tape transports, magnetic drums, analog-to-digital converters, CRT displays, and digital plotters. The PDP-8 system can also accept other types of instruments or hardware devices that have an appropriate interface. Up to 61 devices requiring three programmed command pulses, or up to 193 devices requiring one programmed command pulse can be connected to the computer. One machine using the data break facility can be connected directly to the PDP-8, or up to seven such machines can be connected through a Data Multiplexer Type DM01. Interfacing of any devices to the computer requires no modifications to the processor and can be achieved in the field.

Control of some kind is needed to determine when an information exchange is to take place between the PDP-8 and peripheral equipment and to indicate the location(s) in the computer memory which will accept or yield the data. Either the computer program or the device external to the computer can exercise this control. Transfers controlled by the computer, hence under control of its stored program, are called programmed data transfers. Transfers made at times controlled by the external devices through the data break facility are called data break transfers.

Programmed Data Transfers

The majority of I/O transfers occur under control of the computer program. To transfer and store information under program control requires about six times as much computer time as under data break control. In terms of real time, the duration of a programmed transfer is rather small, due to the high speed of the computer, and is well beyond that required for laboratory or process control instrumentation.

To realize full benefit of the built-in control features of the PDP-8 programmed I/O transfers should be used in most cases. Controls for devices using programmed data transfers are usually simpler and less expensive than controls for devices using data break transfers. Using programmed data transfer facilities, simultaneous operation of devices is limited only by the relative speed of the computer with respect to the device speeds, and the search time required to determine the device requiring service. Analog-to-digital converters, digital-to-analog converters, digital plotters, line printers, message switching equipment, and realy control systems typify equipment using only programmed data transfers.

Data Break Transfers

Devices which operate at very high speed or which require very rapid response from the computer use the data break facilities. Use of these facilities permits an external device, almost arbitrarily, to insert or extract words from the computer core memory, bypassing all program control logic. Because the computer program has no cognizance of transfers made in this manner, programmed checks of input data are made prior to use of information received in this manner. The data break is particularly well-suited for devices that transfer large amounts of data in block form, e.g., high-speed magnetic tape systems, high-speed drum memories, or CRT display systems containing memory elements.

Logic Symbols

Figure 20 defines the symbols used in Section D of this handbook to express signals and digital logic circuits.

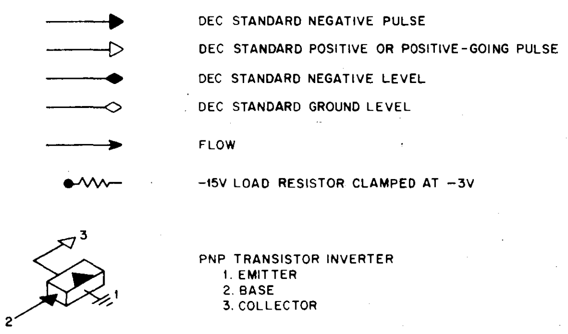
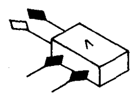
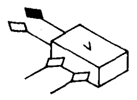

DEC STANDARD NEGATIVE PULSE
DEC STANDARD POSITIVE OR POSITIVE-GOING PULSE
DEC STANDARD NEGATIVE LEVEL
DEC STANDARD GROUND LEVEL
FLOW
-15V LOAD RESISTOR CLAMPED AT -3V
PNP TRANSISTOR INVERTER
1. EMITTER
2. BASE
3. COLLECTOR
1
2
3

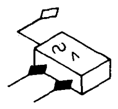
Figure 20 Logic Symbols



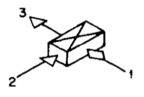
LOGIC AND GATE FOR
NEGATIVE SIGNALS
WITH COMPLEMENTARY
OUTPUT SIGNALS



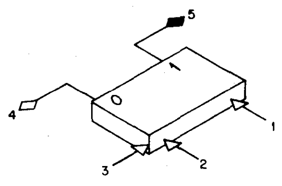
LOGIC OR GATE FOR
GROUND LEVEL SIGNALS
WITH COMPLEMENTARY
OUTPUT SIGNALS



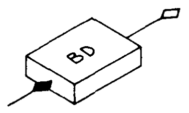
LOGIC NAND GATE FOR
NEGATIVE SIGNALS



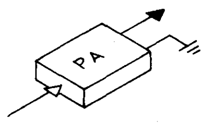
DIODE-CAPACITOR-DIODE GATE
1. CONDITIONING LEVEL INPUT
2. TRIGGERING PULSE INPUT
3. PULSE OUTPUT



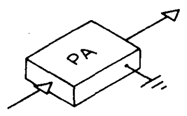
FLIP-FLOP (BISTABLE MULTIVIBRATOR)
1. GATED SET-TO-1 INPUT
2. GATED CLEAR-TO-0 INPUT
3. DIRECT CLEAR-TO-0 INPUT
4,5 OUTPUTS



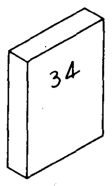
INVERTING BUS DRIVER



B OR W SERIES
PULSE AMPLIFIER. OUTPUT
CAN BE MADE POSITIVE OR
NEGATIVE BY REVERSING
GROUND AND SIGNAL OUTPUT
TERMINALS



R OR S SERIES PULSE AMPLIFIER
OUTPUT ALWAYS POSITIVE,
REFERENCED TO -3V.



DEVICE SELECTOR
LOGIC AS USED FOR ONE
SELECT CODE

Figure 20 Logic Symbols (continued)

CHAPTER 2
PROGRAMMED DATA TRANSFERS

The majority of I/O transfers take place under control of the PDP-8 program, taking advantage of control elements built into the computer. Although programmed transfers take more computer and actual time than do data break transfers, the timing discrepancy is insignificant, considering the high speed of the computer with respect to most peripheral devices. The maximum data transfer rate for programmed operations of 12-bit words is 148 kc when no status checking, end transfer check, etc., is done. This speed is well beyond the normal rate required for typical laboratory or process control instrumentation.

The PDP-8 is a parallel-transfer machine that distributes and collects data in bytes of up to twelve bits. All programmed data transfers take place trhough the accumulator, the 12-bit arithmetic register of the computer. The computer program controls the loading of information into the accumulator (AC) for an output transfer, and for storing information in core memory from the AC for an input transfer. Output information in the AC is power amplified and supplied to the interface connectors for bussed connection to many peripheral devices. Then the program-selected device can sample these signal lines to strobe AC information into a control or information register. Input data arrives at the AC as pulses received at the interface connectors from bussed outputs of many devices. Gating circuits of the program-selected device produce these pulses. Command pulses generated by the device flow to the input/output skip facility (IOS) to sample the condition of I/O device flags. The IOS allows branching of the program based upon the condition or availability of peripheral equipment, effectively making programmed decisions to continue the current program or jump to another part of the program, such as a subroutine that services an I/O device.

The bussed system of input/output data transfers imposes the following requirements on peripheral equipment:

a. The ability of each device to sample the select code generated by the computer during IOT instructions and, when selected, to be capable of producing sequential IOT command pulses in accordance with computer-generated IOP pulses. Circuits which perform these functions in the peripheral device are called the device selector (DS).

b. Each device receiving output data from the computer must contain gating circuits at the input of a receiving register capable of strobing the AC signal information into the register when triggered by a command pulse from the DS.

c. Each device which supplies input data to the computer must contain gating circuits at the output of the transmitting register capable of sampling the information in the output register and supplying a pulse to the computer input bus when triggered by a command pulse from the DS.

d. Each device should contain a busy/done flag (flip-flop) and gating circuits which can pulse the computer input/output skip bus upon command from the DS when the flag is set in the binary 1 state to indicate that the device is ready to transfer another byte of information.

Figure 21 shows the information flow within the computer which effects a programmed data transfer with input/output equipment. All instructions stored in core memory as a program sequence are read into the memory buffer register (MB) for execution. The transfer of the operation code in the three most significant bits (bits 0, 1, and 2) of the instruction into the instruction register (IR) takes place and is decoded to produce appropriate control signals. The computer, upon recognition of the operation code as an IOT instruction, enters a 3.75 μsec expanded computer cycle and enables the IOP generator to produce time sequenced IOP pulses as determined by the three least significant bits of the instruction (bits 9, 10, and 11 in the MB). These IOP pulses and the buffered output of the select code from bits 3-8 of the instruction word in the MB are bussed to device selectors in all peripheral equipment. Figure 22 indicates the timing of programmed data transfers and Figure 23 shows the decoding of the IOT instruction.

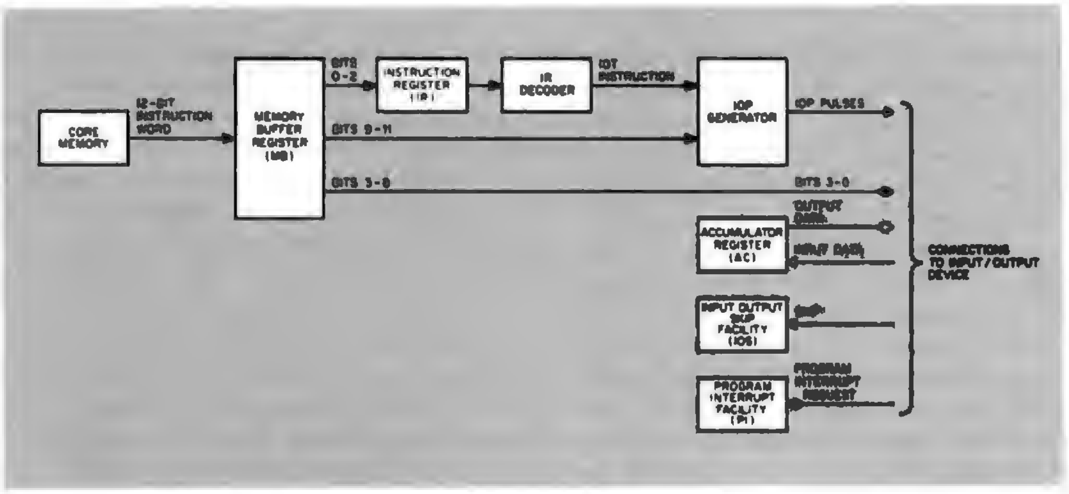

CORE MEMORY
12-BIT INSTRUCTION WORD
MEMORY BUFFER REGISTER (MB)
BITS 0-2
INSTRUCTION REGISTER (IR)
IR DECODER
IOT INSTRUCTION
IOP GENERATOR
IOP PULSES
BITS 9-11
BITS 3-8
BITS 3-8
OUTPUT DATA
ACCUMULATOR REGISTER (AC)
INPUT DATA
CONNECTIONS TO INPUT/OUTPUT DEVICE
INPUT OUTPUT SKIP FACILITY (IOS)
SKIP
PROGRAM INTERRUPT REQUEST
PROGRAM INTERRUPT FACILITY (PI)

Figure 21 Programmed Data Transfer Interface Block Diagram

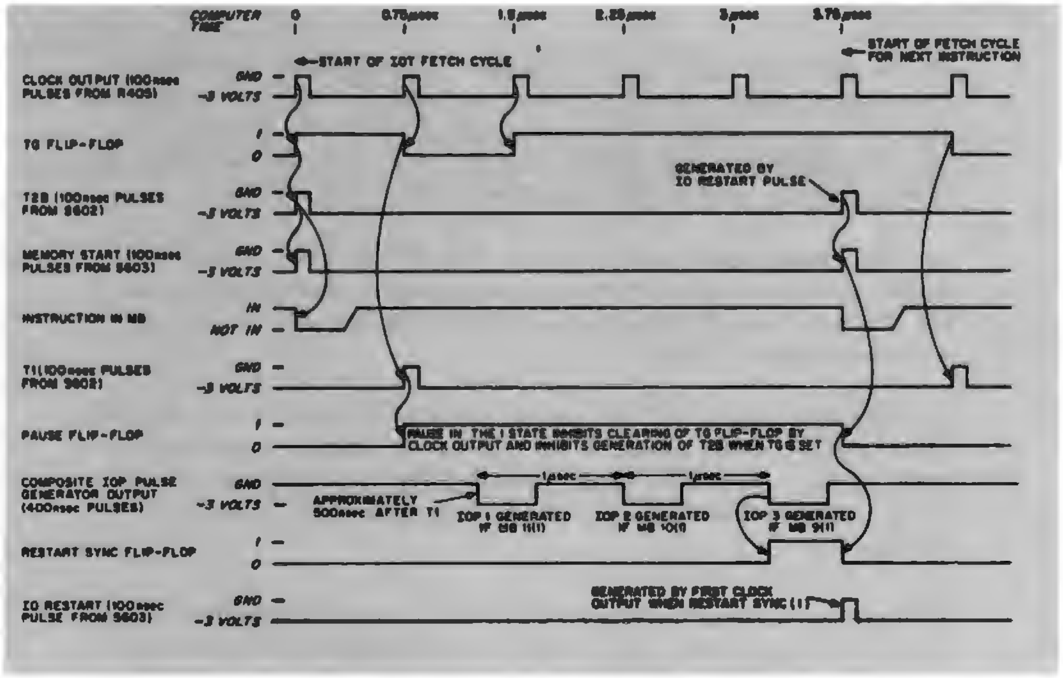

COMPUTER TIME
0
0.75μsec
1.5μsec
2.25μsec
3μsec
3.75μsec
START OF IOT FETCH CYCLE
START OF FETCH CYCLE FOR NEXT INSTRUCTION
CLOCK OUTPUT (100nsec PULSES FROM R405)
GND
-3 VOLTS
TG FLIP-FLOP
1
0
T2B (100nsec PULSES FROM S602)
GENERATED BY IO RESTART PULSE
MEMORY START (100nsec PULSES FROM S603)
INSTRUCTION IN MB
IN
NOT IN
T1 (100nsec PULSES FROM S602)
PAUSE FLIP-FLOP
PAUSE IN THE 1 STATE INHIBITS CLEARING OF TG FLIP-FLOP BY CLOCK OUTPUT AND INHIBITS GENERATION OF T2B WHEN TG IS SET
COMPOSITE IOP PULSE GENERATOR OUTPUT (400nsec PULSES)
APPROXIMATELY 500nsec AFTER T1
1μsec
1μsec
IOP 1 GENERATED IF MB 11(1)
IOP 2 GENERATED IF MB 10(1)
IOP 3 GENERATED IF MB 9(1)
RESTART SYNC FLIP-FLOP
IO RESTART (100nsec PULSE FROM S603)
GENERATED BY FIRST CLOCK OUTPUT WHEN RESTART SYNC (1)

Figure 22 Programmed Data Transfer Timing

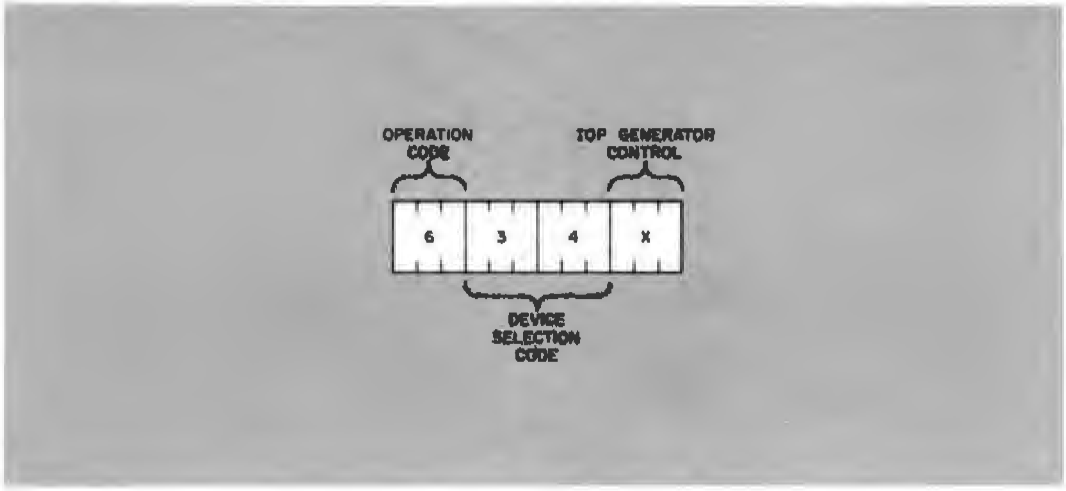


Figure 23 Typical IOT Instruction Decoding

Devices which require immediate service from the computer program, or which take an exorbitant amount of computer time to discontinue the main program until transfer needs are met, can use the program interrupt (PI) facility. In this mode of operation, the computer can initiate operation of I/O equipment and continue the main program until the device requests servicing. A signal input to the PI requesting a program interrupt causes storing of the conditions of the main program and initiates a subroutine to service the device. At the conclusion of this subroutine, the main program is reinstated until another interrupt request occurs.

Timing and IOP Generator

When the IR decoder detects an operation code of 6_8, it identifies an IOT instruction and the computer generates a $1 \longrightarrow$ Pause pulse. This pulse disables the normal timing generators of the processor and initiates operation of the IOP generator. The logic circuits of the IOP generator are shown in Figure 24 to consist of three similar channels; each channel consisting of a gated delay, a gated pulse amplifier, and an output pulse amplifier. Operation of the first channel is triggered by the $1 \longrightarrow$ Pause pulse and operation of the other two channels is triggered by the pulse output of the delay in the previous channel. Series connection of the delays produces sequential operation of the three channels. The pulse output of the third channel delay restarts the normal timing generators of the processor. Since the time delays are 0.5, 1.0, and 1.0 μsec, the cycle time of an IOT instruction is 3.75 μsec. (IOT instructions associated with enabling and disabling the program interrupt facility, and those for the Analog-to-Digital Converter Type 189, the Memory Extension Control Type 183, and the Data Line Interface Type 681 inhibit generation of the $1 \longrightarrow$ Pause pulse and so occur in the normal computer cycle time of 1.5 μsec. In these commands the IOP generator is inhibited so the normal timing pulses of the processor and special device selectors execute these instructions.)

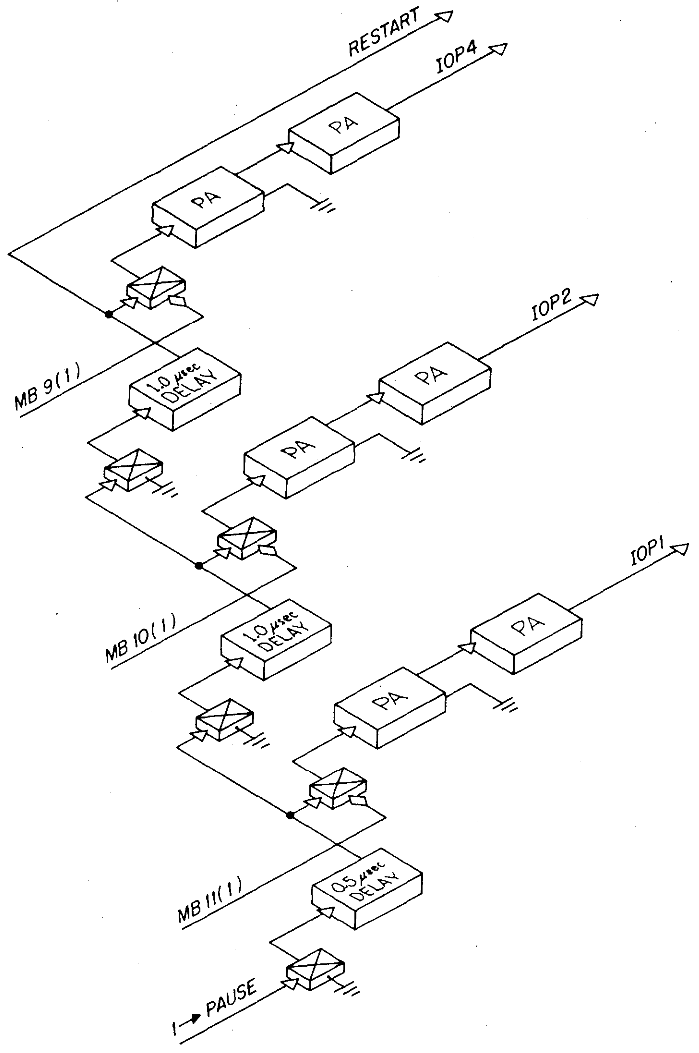


Figure 24 IOP Generator Logic

The gated pulse amplifier of each channel samples the content of one bit of the instruction when the delay output pulse occurs. If the sampled bit contains a binary 1, the gated pulse amplifier is triggered and the output pulse amplifier is operated to produce an IOP pulse. A diode-capacitor-diode (DCD) gate at the input of each gated pulse amplifier serves as a 2-input AND gate. The binary 1 status of one of the least significant bits of the instruction in the MB supplies the conditioning level of each of these gates. The output of the gated pulse amplifier initiates operation of the output pulse amplifier to generate an IOP pulse which is available at the interface connector as a DEC standard 0.4 μsec negative pulse. This configuration allows each IOP pulse to be individually programmed, permits a sequence of up to three events to occur within each instruction, and provides 1 μsec between events for normal device circuit set-up times. The instruction bit that enables or disables generation of each IOP pulse, the correspong-ing number of the IOT pulse produced in the DS from the IOP pulse, and the event time for each IOP pulse is:

Instruction Bit	IOP Pulse	IOT Pulse	Event Time
11	IOP 1	IOT 1	1
10	IOP 2	IOT 2	2
9	IOP 4	IOT 4	3

Device Selector (DS)

Bits 3 through 8 of an IOT instruction serve as a device or subdevice select code. Bus drivers in the processor buffer both the binary 1 and 0 output signals of MB3-8 and distribute them to the interface connectors for bussed connection to all device selectors. Each DS is assigned a select code and is enabled only when the assigned code is present in the MB. When enabled, a DS regenerates IOP pulses as IOT command pulses and transmits these pulses to skip, input, or output gates within the device and/or to the processor to clear the AC.

Each group of three command pulses requires a separate DS channel (W103 module), and each DS channel requires a different select code (or I/O device address). One I/O device can, therefore, use several DS channels. Note that the processor produces the pulses identified as IOP 1, IOP 2, and IOP 4 and supplies them to all device selectors. The device selector produces pulses IOT 1, IOT 2, and IOT 4 which initiate a transfer or effect some control. Figure 25 shows generation of command pulses by several DC channels.

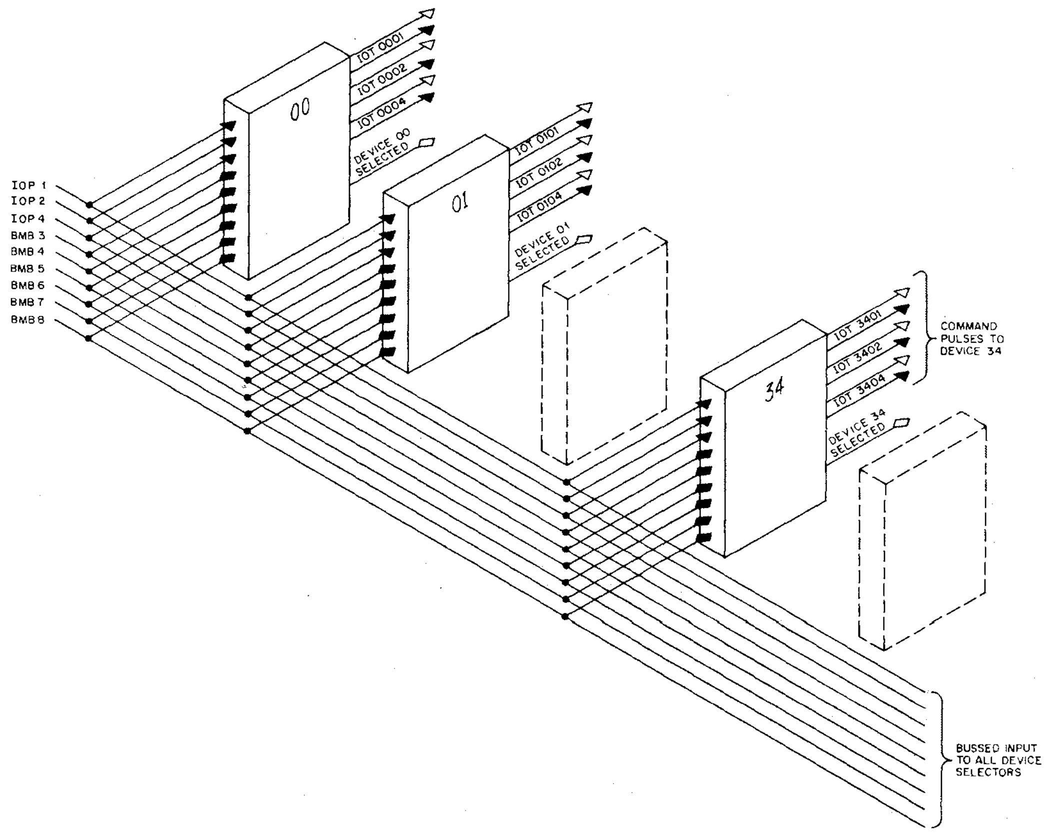


Figure 25 Generation of IOT Command Pulses by Device Selectors

The logical representation for a typical channel of the DS, using channel 34, is shown in Figure 26. A 6-input NAND gate wired to receive the appropriate signal outputs from MB3-8 for select code 34 activates the channel. In the DS module, the NAND gate contains 14 diode input terminals; 12 of these connect to the complementary outputs of MB3-8, and 2 are open to receive subdevice or control condition signals as needed. Either the 1 or the 0 signal from each MB bit is disconnected by removing the appropriate diode from the NAND gate when establishing the select code. The ground level output of the NAND gate indicates when the IOT instruction selects the device, and can therefore enable circuit operations within the device. This output also enables three gating inverters, allowing them to trigger a pulse amplifier if an IOP pulse occurs. The positive output from each pulse amplifier is an IOT command pulse identified by the select code and the number of the initiating IOP pulse. Three inverters receive the positive IOT pulses to produce complementary IOT output pulses. A pulse amplifier module can be connected in each channel of the DS to provide greater output drive or to produce pulses of a specific duration required by the selected device.

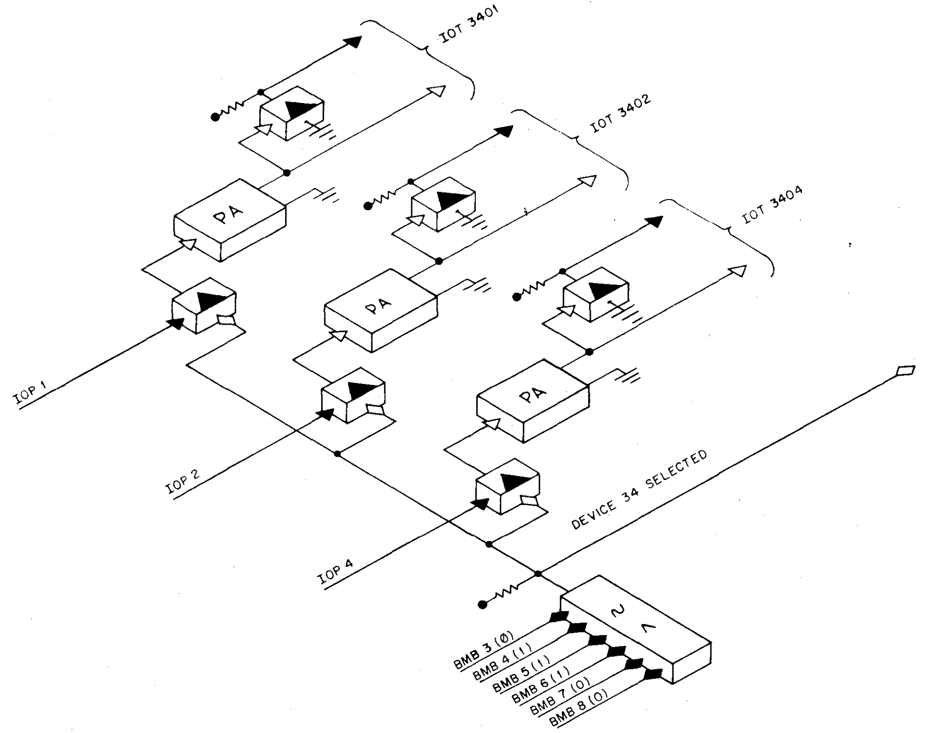
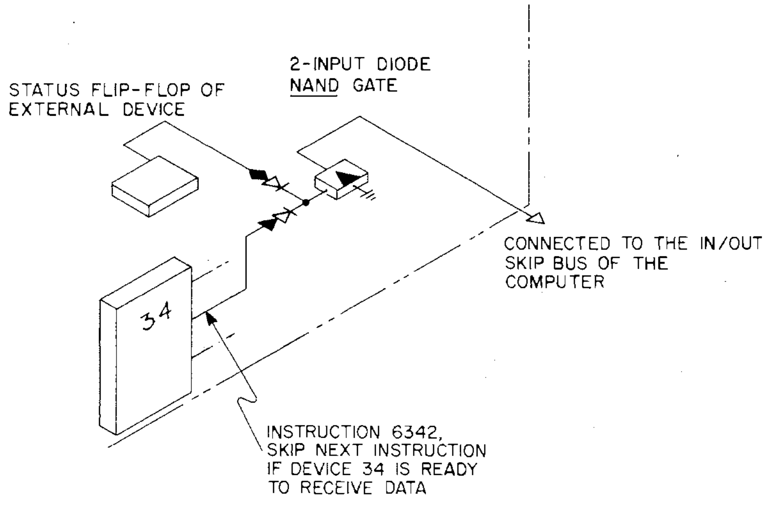


Figure 26 Typical Device Selector (Device 34)

Input/Output Skip (IOS)

Generation of an IOT pulse can be used to test the condition or status of a device flag, and to continue to or skip the next sequential instruction based upon the results of this test. This operation is performed by a 2-input AND gate in the device connected as shown in Figure 27. One input of the skip gate receives the status level (flag output signal), the second input receives an IOT pulse, and the output drives the computer IOS bus to ground when the skip conditions are fulfilled. When the IOS bus is driven to ground, the content of the program counter is incremented by 1 to advance the program count without executing the instruction at the current program count. In this manner an IOT instruction can check the status of an I/O device flag and skip the next instruction if the device requires servicing. Programmed testing in this manner allows the routine to jump out of sequence to a subroutine that services the device tested.



2-INPUT DIODE NAND GATE
STATUS FLIP-FLOP OF EXTERNAL DEVICE
CONNECTED TO THE IN/OUT SKIP BUS OF THE COMPUTER
34
INSTRUCTION 6342, SKIP NEXT INSTRUCTION IF DEVICE 34 IS READY TO RECEIVE DATA

Figure 27 Use of IOS to Test the Status of an External Device

Assuming that a device is already operating, a possible program sequence to test its availability follows:

```
Address          Instruction          Remarks

   .
   .
   .
   .
100,             6342                 /SKIP IF DEVICE 34 IS READY
101,             5100                 /JUMP .-1
102,             5XXX                 /ENTER SERVICE ROUTINE FOR
                                      /DEVICE 34
   .
   .
   .
   .
```

When the program reaches address 100, it executes an instruction skip with 6342. The skip occurs only if device 34 is ready when the IOT 6342 command is given. If device 34 is not ready, the flag signal disqualifies the skip gate, and the Skip pulse does not occur. Therefore, the program continues to the next instruction which is a jump back to the skip instruction. In this example, the program stays in this waiting loop until the device is ready to transfer data, at which time the skip gate in the device is enabled and the Skip pulse is sent to the computer IOS facility. When the skip occurs, the instruction in location 102 transfers program control to a subroutine to service device 34. This subroutine can load the AC with data and transfer it to device 34, or can load the AC from a register in device 34 and store it in some known core memory address.

Accumulator

The binary 1 output signal of each flip-flop of the AC, buffered by a bus driver, is available at the interface connectors. These computer data output lines are bus connected to all peripheral equipment receiving programmed data output information from the PDP-8. A direct-set terminal on each flip-flop of the AC is connected to the interface connectors for bussing to all peripheral equipment supplying programmed data input to the PDP-8. A pulse that drives the direct-set terminal to ground causes setting of an AC flip-flop to the binary 1 state. Output and input connections to the accumulator appear in Figure 28.

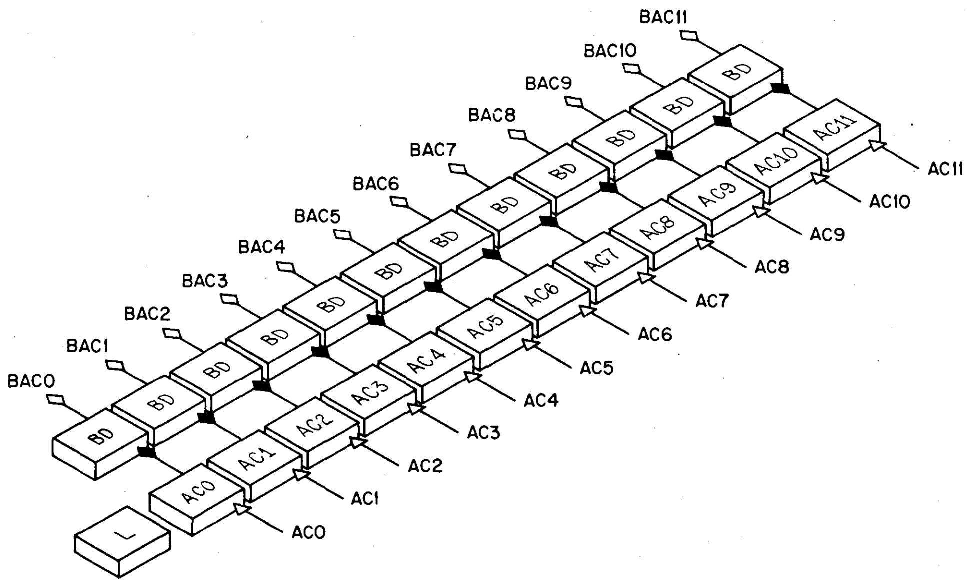


Figure 28 Accumulator Input and Output

Figure 28 illustrates the twelve bits of the accumulator and the link bit. The status of the link bit is not available to enter into transfers with peripheral equipment (unless it is rotated into the AC). A noninverting bus driver continuously buffers the output signal from each AC flip-flop. These buffered accumulator (BAC) signals are available at the interface connectors.

Input Data Transfers

When ready to transfer data into the PDP-8 accumulator, the device sets a flag connected to the IOS. The program senses the ready status of the flag and issues an IOT instruction to read the content of the external device buffer register into the AC. If the AC is not cleared before the transfer, the resultant word in the AC is the inclusive OR of the previous word in the AC and the word transferred from the device buffer register.

The illustration in Figure 29 shows that the accumulator has an input bus for each bit flip-flop. Setting a 1 into a particular bit of the accumulator necessitates grounding of the interface input bus by the standard DEC inverter. In the illustration, the 2-input AND gates set various bits of the accumulator. In this case an IOT pulse is AND combined with the flip-flop state of the external device to conditionally set 1's into the accumulator. (The program must include a clear AC command prior to loading in this manner; otherwise an inclusive OR takes place between the previous content of the accumulator and the content of the data register being read.)

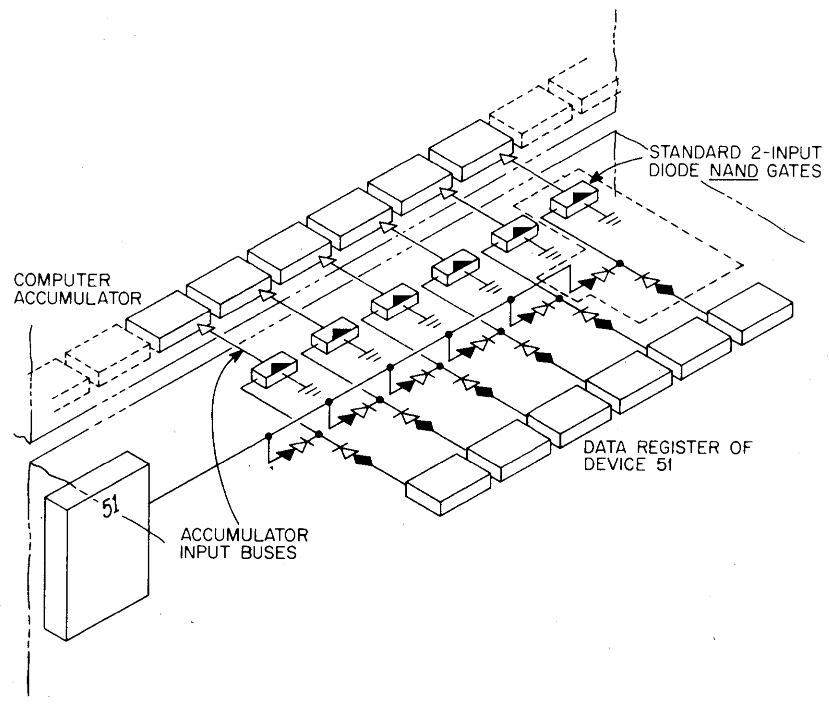


Figure 29 Loading Data into the Accumulator from an External Device

Following the transfer (possibly in the same instruction) the program can issue a command pulse to initiate further operation of the device and/or clear the device flag.

Output Data Transfers

The AC is loaded with a word (e.g., by a CLA TAD instruction sequence); then the IOT instruction is issued to transfer the word into the control or data register of the device by an IOT pulse (e.g., IOP 2), and operation of the device is initiated by another IOT pulse (e.g., IOP 4). The data word transferred in this manner can be a character to be operated upon, or can be a control word sampled by a status register to establish a control mode.

Since the BAC interface bus lines continually represent the status of the AC flip-flops, the receiving device can strobe them to sense the value in the accumulator. In Figure 30 a strobe pulse samples six bits of the accumulator to conditionally set an external 6-bit data register. Since this is not a jam transfer, it is necessary first to clear the external data register before setting 1's into it. The readin gates driving the external data register are part of the external device and are not supplied by the computer. The data register can contain any number of flip-flops up to a maximum of twelve. (If more than twelve flip-flops are involved, two or more transfers must take place.) Obviously the clear pulse and the strobe pulse shown in Figure 30 must occur when the data to be placed in the external data register is held in the accumulator. These pulses therefore must be under computer control to effect synchronization with the operation or program of the computer.

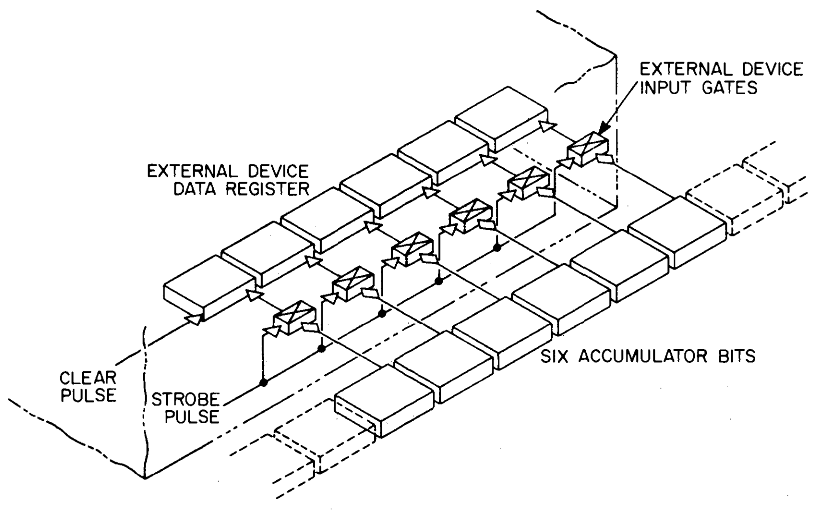


Figure 30 Loading a 6-Bit Word into an External Device from the Accumulator

Figure 31 illustrates the use of two of the pulses being gated by the device selector coded for "34." Pulse IOT 1 clears the data register and IOT 4 strobes the data from the accumulator into the data register. Note that the processor produces the IOP 1, IOP 2, and IOP 4 pulses and supplies them to all device selectors. The program-selected DS produces IOT 1, IOT 2, and IOT 4 pulses which initiate a transfer or effect some control. As indicated in Figure 31 this particular system adds two new microinstructions to the PDP-8 repertoire. One generates a pulse to clear the data register of device number 34. The other microinstruction produces a pulse to load the data register of device number 34 with the content of the accumulator.

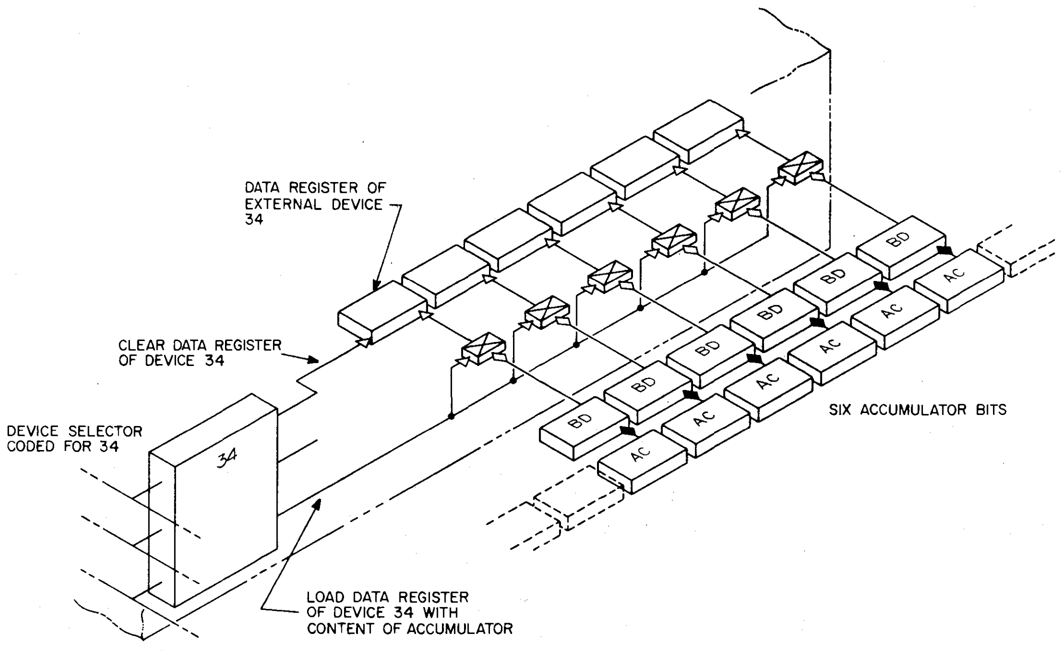

DATA REGISTER OF EXTERNAL DEVICE 34
CLEAR DATA REGISTER OF DEVICE 34
DEVICE SELECTOR CODED FOR 34
34
BD
AC
SIX ACCUMULATOR BITS
LOAD DATA REGISTER OF DEVICE 34 WITH CONTENT OF ACCUMULATOR

Figure 31 Use of a Device Selector for Activating and Controlling an External Device

The timing of the IOT cycle is shown in Figure 22. Note that the AC bus drivers are quiescent 400 nsec before the IOP 1 pulse occurs. Since FLIP CHIP DCD gates require a 400-nsec set-up time, the IOP 1 pulse cannot be used to load the content of the AC into an external buffer register having input DCD gates. If the device register has DCD gates, IOP 1 should be used to reset or clear registers, controls, or flags. The IOP 1 pulse can be used to read the content of the AC into an external device register that is equipped with input diode gates. IOP 2 or IOP 4 can be used to strobe the content of the AC through DCD gates if the lead lengths of the BAC lines and the pulse lines provide equivalent transmission delays. Only IOP 1 or IOP 2 (not IOP 4) can be used with the IOS facility.

Program Interrupt (PI)

When a large amount of computing is required, the program should initiate operation of an I/O device then continue the main program, rather than wait for the device to become ready to transfer data. The program interrupt facility, when enabled by the program, relieves the main program of the need for repeated flag checks by allowing the ready status of I/O device flags to automatically cause a program interrupt. When the program interrupt occurs, program control transfers to a subroutine that determines which device requested the interrupt and initiates an appropriate service routine.

In the example shown in Figure 32, a flag signal from a status flip-flop operates a standard inverter with no collector load. When the status flip-flop indicates the need for device service, the inverter drives the Program Interrupt Request bus to ground to request a program interrupt.

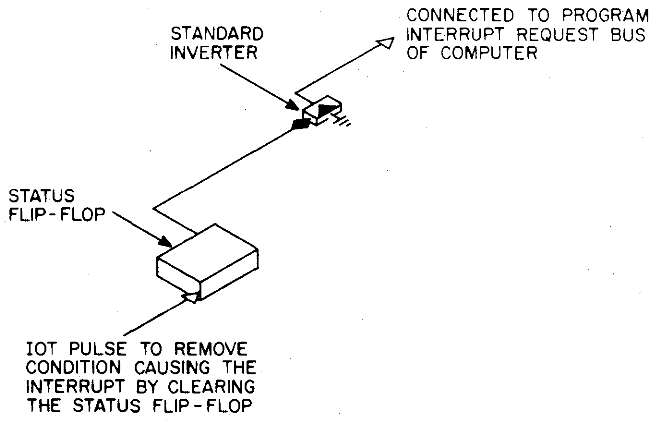

STANDARD INVERTER
CONNECTED TO PROGRAM INTERRUPT REQUEST BUS OF COMPUTER
STATUS FLIP-FLOP
IOT PULSE TO REMOVE CONDITION CAUSING THE INTERRUPT BY CLEARING THE STATUS FLIP-FLOP

Figure 32 Program Interrupt Request Signal Origin

If only one device is connected to the PI facility, program control can be transferred directly to a routine that services the device when an interrupt occurs. This operation occurs as follows:

```
Tag   Address   Instruction           Remarks
      1000          .         /MAIN PROGRAM
      1001          .         /MAIN PROGRAM CONTINUES
      1002          .         /INTERRUPT REQUEST OCCURS
                 INTERRUPT OCCURS
      0000                    /PROGRAM COUNT (PC=1003) IS STORED IN 0000
      0001      JMP SR        /ENTER SERVICE ROUTINE
SR    2000          .         /SERVICE SUBROUTINE FOR INTERRUPTING
                    .         /DEVICE AND SEQUENCE TO RESTORE
      3001          .         /AC, AND RESTORE L IF REQUIRED
      3002      ION           /TURN ON INTERRUPT
      3003      JMP I 0000    /RETURN TO MAIN PROGRAM
      1003          .         /MAIN PROGRAM CONTINUES
      1004          .
                    .
                    .
```

In most PDP-8 systems numerous devices are connected to the PI facility, so the routine beginning in core memory address 0001 must determine which device requested an interrupt. The interrupt routine determines the device requiring service by checking the flags of all equipment connected to the PI and transfers program control to a service routine for the first device encountered that has its flag in the state required to request a program interrupt. In other words, when program interrupt requests can originate in numerous devices, each device flag connected to the PI must also be connected to the IOS.

Multiple Use of IOS and PI

In common practice, more than one device is connected to the PI facility. Therefore, since the computer receives a request that is the inclusive OR of requests from all devices connected to the PI, the IOS must identify the device making the request. When a program interrupt occurs, a routine is entered from address 0001 to sequentially check the status of each flag connected to the PI and to transfer program control to an appropriate service routine for the device whose flag is requesting a program interrupt. Figure 33 shows IOS and PI connections for three typical devices.

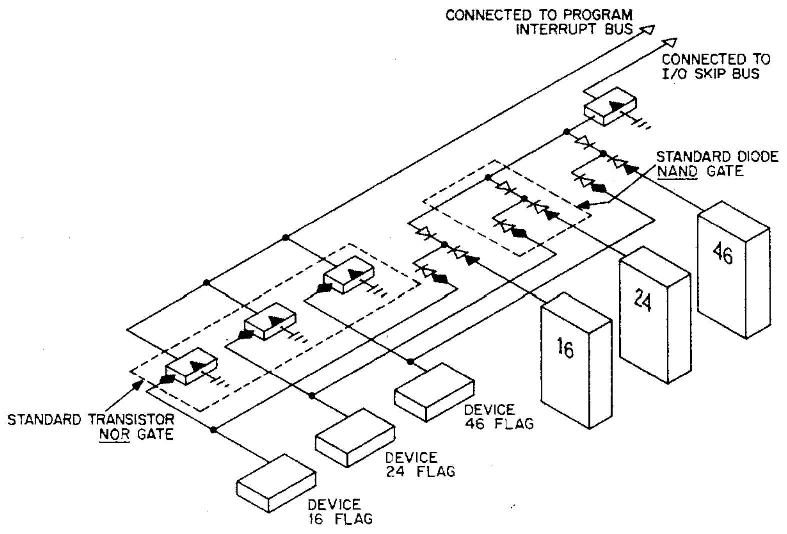

CONNECTED TO PROGRAM INTERRUPT BUS
CONNECTED TO I/O SKIP BUS
STANDARD DIODE NAND GATE
46
24
16
STANDARD TRANSISTOR NOR GATE
DEVICE 46 FLAG
DEVICE 24 FLAG
DEVICE 16 FLAG

Figure 33 Multiple Inputs to IOS and PI Facilities

The following program example illustrates how the program interrupt routine determines the device requesting service:

```
Tag      Address   Instruction               Remarks

         1000          .         /MAIN PROGRAM
         1001          .         /MAIN PROGRAM CONTINUES
         1002          .         /INTERRUPT REQUEST OCCURS
                         INTERRUPT OCCURS
         0000                    /STORE PC (PC = 1003)
         0001      JMP FLG CK    /ENTER ROUTINE TO DETERMINE WHICH DEVICE
                                 /CAUSED INTERRUPT
FLG CK             IOT 6341      /SKIP IF DEVICE 34 IS REQUESTING
                   SKP           /NO - TEST NEXT DEVICE
                   JMP SR34      /ENTER SERVICE ROUTINE 34
                   IOT 6441      /SKIP IF DEVICE 44 IS REQUESTING
                   SKP           /NO - TEST NEXT DEVICE
                   JMP SR44      /ENTER SERVICE ROUTINE 44
                   IOT 6541      /SKIP IF DEVICE 54 IS REQUESTING
                   SKP           /NO - TEST NEXT DEVICE
                   JMP SR54      /ENTER SERVICE ROUTINE 54
                       .
                       .
                       .
```

Assume that the device that caused the interrupt is an input device (e.g., tape reader). The following example of a device service routine might apply:

```
Tag                Instruction               Remarks

SR                 DAC TEMP      /SAVE AC
                   IOT XX        /TRANSFER DATA FROM DEVICE BUFFER TO AC
                   DAC I 10      /STORE IN MEMORY LIST
                   ISZ COUNT     /CHECK FOR END
                   SKP           /NOT END
                   JMP END       /END. JUMP TO ROUTINE TO HANDLE END OF
                                 /LIST CONDITION
                       .
                       .
                       .
                                 /RESTORE L AND EPC IF REQUIRED
                   TAD TEMP      /RELOAD AC
                   ION           /TURN ON INTERRUPT
                   JMP I 0       /RETURN TO PROGRAM
```

If the device that caused the interrupt was essentially an output device (receiving data from computer), the IOT - then - DAC I 10 sequence might be replaced by a TAD I 10 - then - IOT sequence.

CHAPTER 3

DATA BREAK TRANSFERS

The data break facility allows one I/O device to transfer information directly with the PDP-8 core memory on a cycle-stealing basis. Up to seven devices can connect to the data break facility through the optional Data Multiplexer Type DM01. The data break is particularly well-suited for devices which transfer large amounts of information in block form.

Peripheral I/O equipment operating at high speeds can transfer information with the computer through the data break facility more efficiently than through programmed means. The combined maximum transfer rate of the data break facility is over 7.8 million bits per second. Information flow to effect a data break transfer with an I/O device appears in Figure 34.

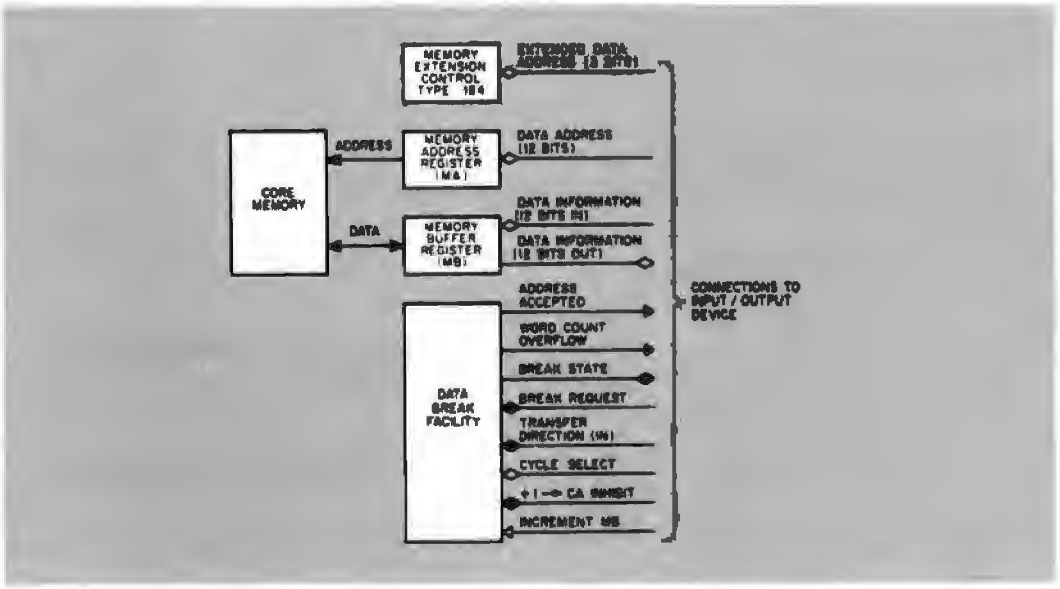

MEMORY EXTENSION CONTROL TYPE 184
EXTENDED DATA ADDRESS (3 BITS)
CORE MEMORY
ADDRESS
MEMORY ADDRESS REGISTER (MA)
DATA ADDRESS (12 BITS)
DATA
MEMORY BUFFER REGISTER (MB)
DATA INFORMATION (12 BITS IN)
DATA INFORMATION (12 BITS OUT)
DATA BREAK FACILITY
ADDRESS ACCEPTED
WORD COUNT OVERFLOW
BREAK STATE
BREAK REQUEST
TRANSFER DIRECTION (IN)
CYCLE SELECT
+1 → CA INHIBIT
INCREMENT MB
CONNECTIONS TO INPUT / OUTPUT DEVICE

Figure 34 Data Break Transfer Interface Block Diagram

In contrast to programmed operations, the data break facilities permit an external device to control information transfers. Therefore, data-break device interfaces require more control logic circuits, causing a higher cost than programmed-transfer interfaces.

Data breaks are of two basic types: single-cycle and three-cycle. In a single-cycle data break, registers in the device (or device interface) specify the core memory address of each transfer and count the number of transfers to determine the end of data blocks. In the three-cycle data break two computer core memory locations perform these functions, simplifying the device interface by omitting two hardware registers.

In general terms, to initiate a data break transfer of information, the interface control must do the following:

a. Specify the affected address in core memory.

b. Provide the data word by establishing the proper logic levels at the computer interface (assuming an input data transfer), or provide readin gates and storage for the word (assuming an output data transfer).

c. Provide a logical signal to indicate direction of data word transfer.

d. Provide a logical signal to indicate single-cycle or three-cycle break operation.

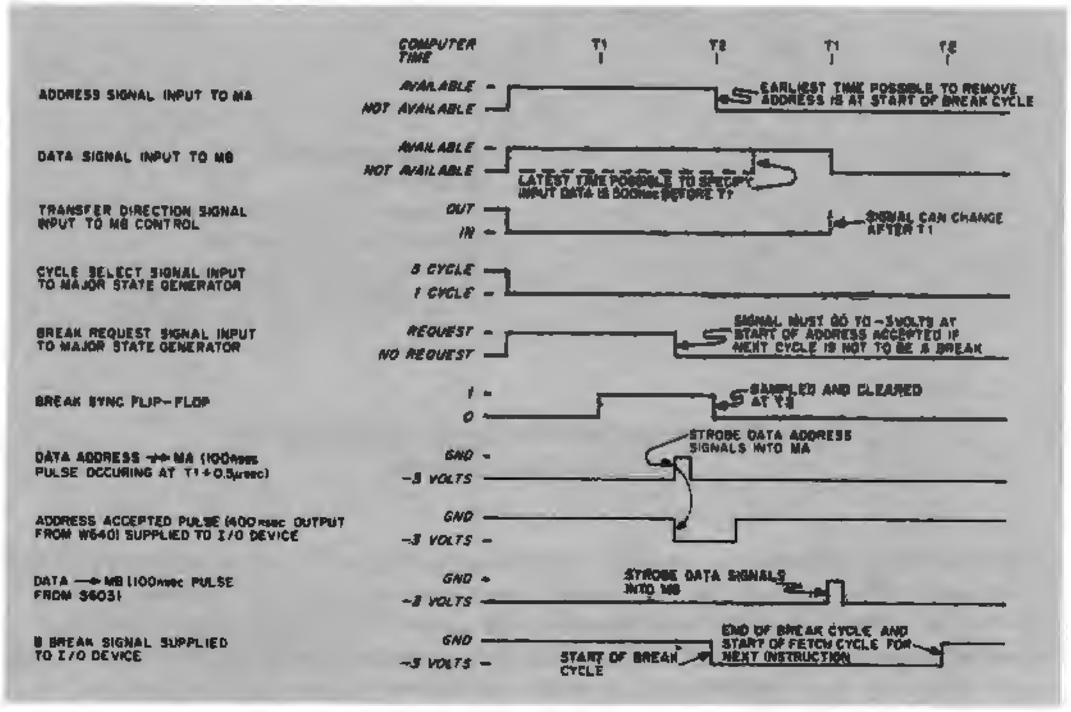
e. Request a data break by supplying a proper signal to the computer data break facility.

Single-Cycle Data Breaks

Single-cycle breaks are used for input data transfers to the computer, output data transfers from the computer, and memory increment data breaks. Memory increment is a special output data break in which the content of a memory address is read, incremented by 1, and rewritten at the same address. It is useful for counting iterations or external events without disturbing the computer program counter (PC) or AC registers.

INPUT DATA TRANSFERS

Figure 35 illustrates timing of an input transfer data break. The address to be affected in core is normally provided in the device interface in the form of a 12-bit flip-flop register (data break address register) which has been preset by the interface control by programmed transfer from the computer.



COMPUTER TIME
T1
T2
T1
T2
ADDRESS SIGNAL INPUT TO MA
AVAILABLE
NOT AVAILABLE
EARLIEST TIME POSSIBLE TO REMOVE ADDRESS IS AT START OF BREAK CYCLE
DATA SIGNAL INPUT TO MB
AVAILABLE
NOT AVAILABLE
LATEST TIME POSSIBLE TO SPECIFY INPUT DATA IS 300nsec BEFORE T1
TRANSFER DIRECTION SIGNAL INPUT TO MB CONTROL
OUT
IN
SIGNAL CAN CHANGE AFTER T1
CYCLE SELECT SIGNAL INPUT TO MAJOR STATE GENERATOR
3 CYCLE
1 CYCLE
BREAK REQUEST SIGNAL INPUT TO MAJOR STATE GENERATOR
REQUEST
NO REQUEST
SIGNAL MUST GO TO -3VOLTS AT START OF ADDRESS ACCEPTED IF NEXT CYCLE IS NOT TO BE A BREAK
BREAK SYNC FLIP-FLOP
1
0
SAMPLED AND CLEARED AT T2
DATA ADDRESS → MA (100nsec PULSE OCCURING AT T1+0.5μsec)
GND
-3 VOLTS
STROBE DATA ADDRESS SIGNALS INTO MA
ADDRESS ACCEPTED PULSE (400nsec OUTPUT FROM W640) SUPPLIED TO I/O DEVICE
GND
-3 VOLTS
DATA → MB (100nsec PULSE FROM S603)
GND
-3 VOLTS
STROBE DATA SIGNALS INTO MB
B BREAK SIGNAL SUPPLIED TO I/O DEVICE
GND
-3 VOLTS
START OF BREAK CYCLE
END OF BREAK CYCLE AND START OF FETCH CYCLE FOR NEXT INSTRUCTION

Figure 35 Single-Cycle Data Break Input Transfer Timing Diagram

External registers and control flip-flops supplying information and control signals to the data break facility and other PDP-8 interface elements are shown in Figure 36. The input buffer register (IB in Figure 36) holds the 12-bit data word to be written into the computer core memory location specified by the address contained in the address register (AR in Figure 36). Appropriate output terminals of these registers are connected to the computer to supply ground potential to designate binary 1's. Since most devices that transfer data through the data break facility are designed to use either single-cycle or three-cycle breaks, but not both, the Cycle Select signal can usually be supplied from a stable source (such as a ground connection or a −3v clamped load resistor) rather than from a bistable device as shown in Figure 36.

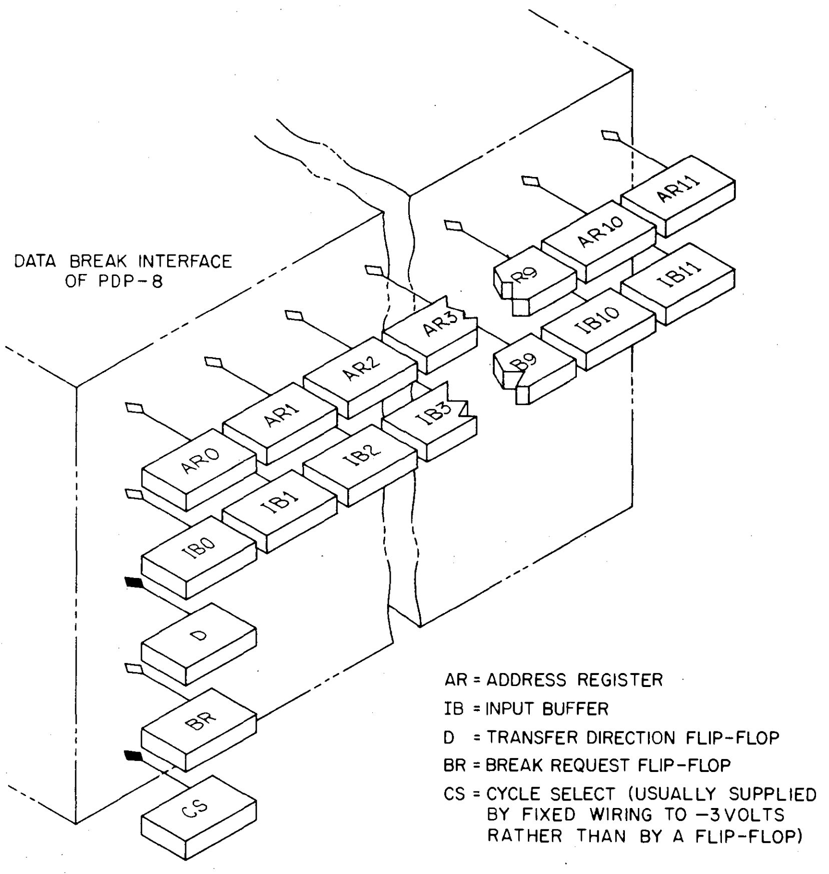

DATA BREAK INTERFACE OF PDP-8
AR11
AR10
AR9
IB11
IB10
IB9
AR3
AR2
AR1
AR0
IB3
IB2
IB1
IB0
D
BR
CS
AR = ADDRESS REGISTER
IB = INPUT BUFFER
D = TRANSFER DIRECTION FLIP-FLOP
BR = BREAK REQUEST FLIP-FLOP
CS = CYCLE SELECT (USUALLY SUPPLIED BY FIXED WIRING TO −3VOLTS RATHER THAN BY A FLIP-FLOP)

Figure 36 Device Interface Logic for Single-Cycle Data Break Input Transfer

Other portions of the device interface, not shown in Figure 36, establish the data word in the input buffer register, set the address into the address register, set the direction flip-flop to indicate an input data transfer, and control the break request flip-flop. These operations can be performed simultaneously or sequentially, but all transients should occur before the data break request is made. Note that the device interface need supply only static levels to the computer, minimizing the synchronizing logic circuits necessary in the device interface.

When the data break request arrives, the computer completes the current instruction, generates an Address Accepted pulse (at T1 time of the cycle preceding the data break) to acknowledge receipt of the request, then enters the Break state to effect the transfer (see Section A, Chapter 5 of this handbook for more details on data break operations performed by the computer). The Address Accepted pulse can be used in the device interface to clear the break request flip-flop, increment the content of the address register, etc. If the Break Request signal is removed before T1 time of the data break cycle, the computer performs the transfer in one 1.5-μsec cycle and returns to programmed operation.

OUTPUT DATA TRANSFERS

Timing of operations occurring in a single-cycle output data break is shown in Figure 37. Basic logic circuits for the device interface used in this type of transfer are shown in Figure 38. Address and control signal generators are similar to those discussed previously for input data transfers, except that the Transfer Direction signal must be at ground potential to specify the output transfer of computer information. An output data register (OB in Figure 38) is usually required in the device interface to receive the computer information. The device, and not the PDP-8, controls strobing of data into this register. The device must supply strobe pulses for all data transfers out of the computer (programmed or data break) since circuit configuration and timing characteristics differ in each device.

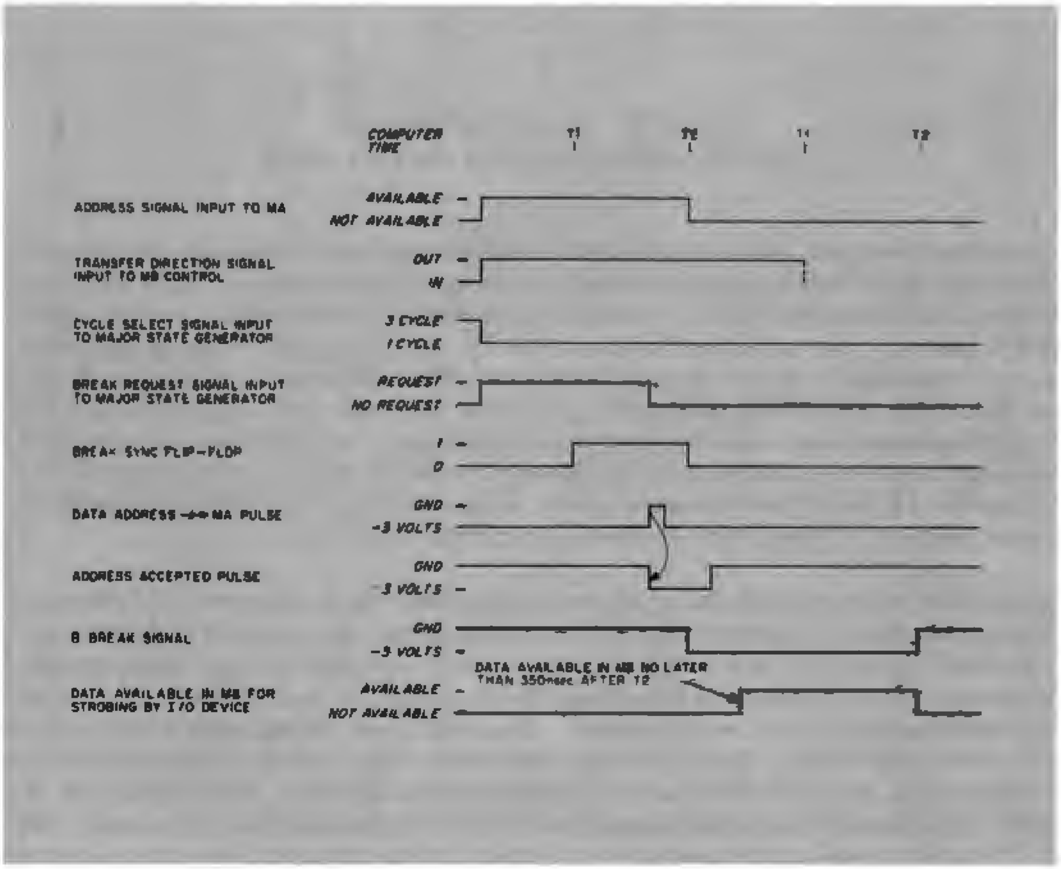

COMPUTER TIME
T1
T2
T1
T2
ADDRESS SIGNAL INPUT TO MA
AVAILABLE
NOT AVAILABLE
TRANSFER DIRECTION SIGNAL INPUT TO MB CONTROL
OUT
IN
CYCLE SELECT SIGNAL INPUT TO MAJOR STATE GENERATOR
3 CYCLE
1 CYCLE
BREAK REQUEST SIGNAL INPUT TO MAJOR STATE GENERATOR
REQUEST
NO REQUEST
BREAK SYNC FLIP-FLOP
1
0
DATA ADDRESS → MA PULSE
GND
-3 VOLTS
ADDRESS ACCEPTED PULSE
GND
-3 VOLTS
B BREAK SIGNAL
GND
-3 VOLTS
DATA AVAILABLE IN MB NO LATER THAN 350nsec AFTER T2
DATA AVAILABLE IN MB FOR STROBING BY I/O DEVICE
AVAILABLE
NOT AVAILABLE

Figure 37 Single-Cycle Data Break Output Transfer Timing Diagram

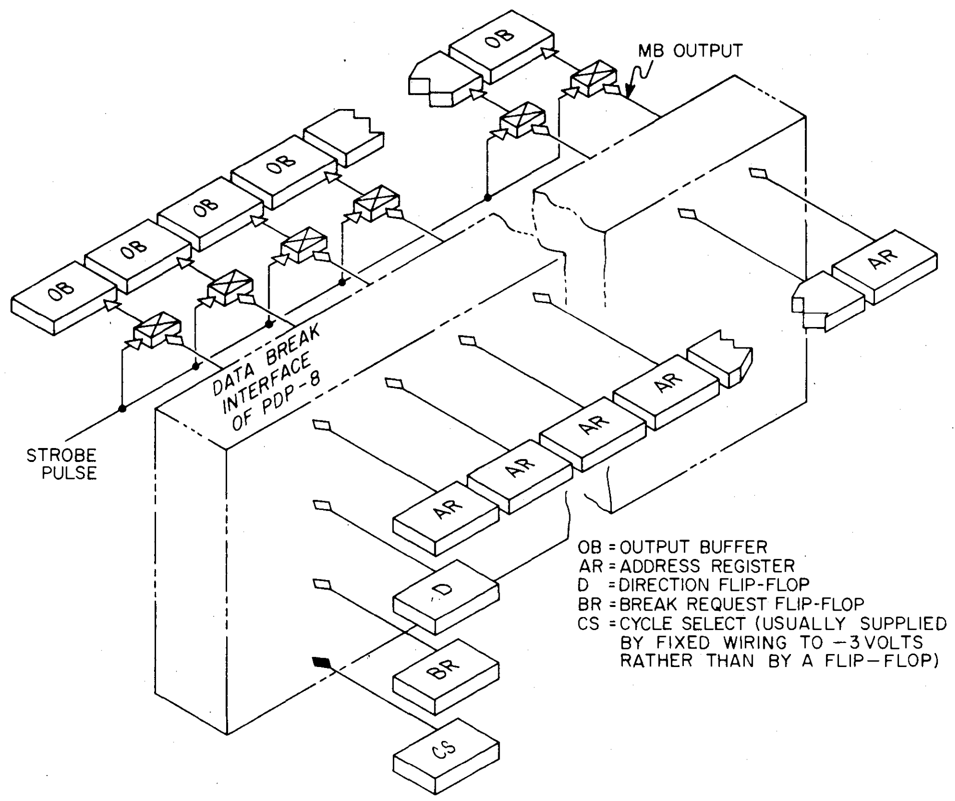

MB OUTPUT
OB
AR
D
BR
CS
DATA BREAK INTERFACE OF PDP-8
STROBE PULSE
OB = OUTPUT BUFFER
AR = ADDRESS REGISTER
D = DIRECTION FLIP-FLOP
BR = BREAK REQUEST FLIP-FLOP
CS = CYCLE SELECT (USUALLY SUPPLIED BY FIXED WIRING TO -3VOLTS RATHER THAN BY A FLIP-FLOP)

Figure 38 Device Interface Logic
for Single-Cycle Data Break Output Transfer

When the data break request arrives, the computer completes the current instruction and generates an Address Accepted pulse as in input data break transfers. At T2 time the address supplied to the PDP-8 is loaded into the MA, the Break state is entered, and the MB is cleared. Not more than 350 nsec after T2, the content of the device-specified core memory address is read and available in the MB. (This word is automatically re-written at the same address during the last half of the Break cycle and is available for programmed operations when the data break is finished.) Data Bit signals are available as static levels of ground potential for binary 1's and −3v for binary 0's. The MB is cleared at T2 time of each computer cycle, so the data word is available in the MB for approximately 1.15 μsec to be strobed by the device interface.

Generation of the strobe pulse by the device interface can be synchronized with computer timing through use of timing pulses B T1 or B T2A, which are available at the computer interface. In addition to a timing pulse (delayed or used directly from the computer), generation of this strobe pulse should be gated by condition signals that occur only during the Break cycle of an output transfer. Figure 39 shows typical logic circuits to effect an output data transfer. In this example the B Break signal and an inverted Transfer Direction signal are combined in a diode NAND gate to condition a diode-capacitor-diode gate. A buffered B T2A pulse triggers the DCD gate to produce the strobe pulse. The B T2A pulse determines the timing of the transfer in this example, since the input of the output buffer register has DCD gates. Conventional DCD gates require a minimum set-up time of 400 nsec, which is adequately provided between the time when data is available in the MB and T2 time. Although the MB is cleared and the major state generator is changed at T2 time, the B T2A pulse can effect this transfer because the delay built into FLIP CHIP flip-flops allows the output to be sampled while the input is being pulsed. If diode gates or other devices with a set-up time of less than 400 nsec are used at the input of the output buffer register, the B T1 pulse, or some other pulse generated by the device interface before T2 time, can trigger strobe pulse generation.

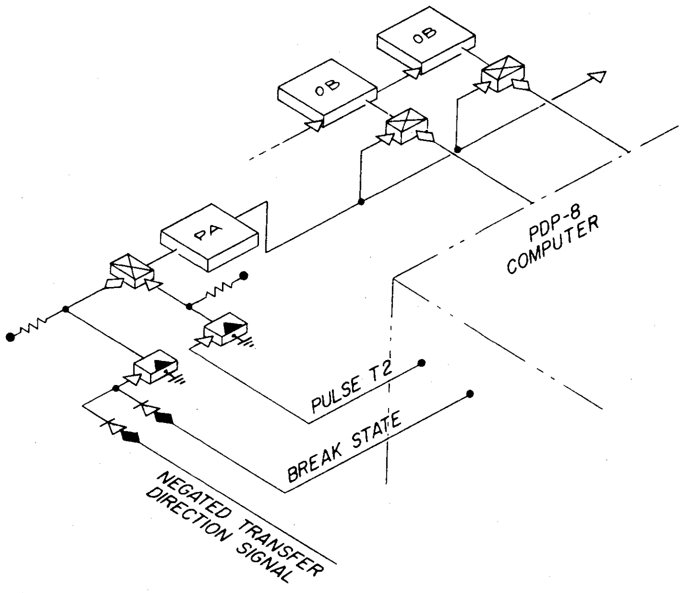


Figure 39 Device Interface for Strobing Output Data

By careful design of the input and output gating, one register can serve as both the input and output buffer register. Most DEC options using the data break facility have only one data buffer register with appropriate gating to allow it to serve as an output buffer when the Transfer Direction signal is at ground potential or as an input buffer when the Transfer Direction signal is −3v.

MEMORY INCREMENT

In this type of data break the content of core memory at a device-specified address is read into the MB, is incremented by 1, and is rewritten at the same address within one 1.5-μsec cycle. This feature is particularly useful in building a histogram of a series of measurements, such as in pulse-height analysis applications. For example, in a computer-controlled experiment that counts the number of times each value of a parameter is measured, a data break can be requested for each measurement, and the measured value can be used as the core memory address to be incremented (counted).

Signal interface for a memory increment data break is similar to an output transfer data break except that the device interface generates an Increment MB signal and does not generate a strobe pulse (no data transfer occurs between the PDP-8 and the device). Timing of memory increment operations appear in Figure 40, and an example of the logic circuits used by a device interface appears in Figure 41.

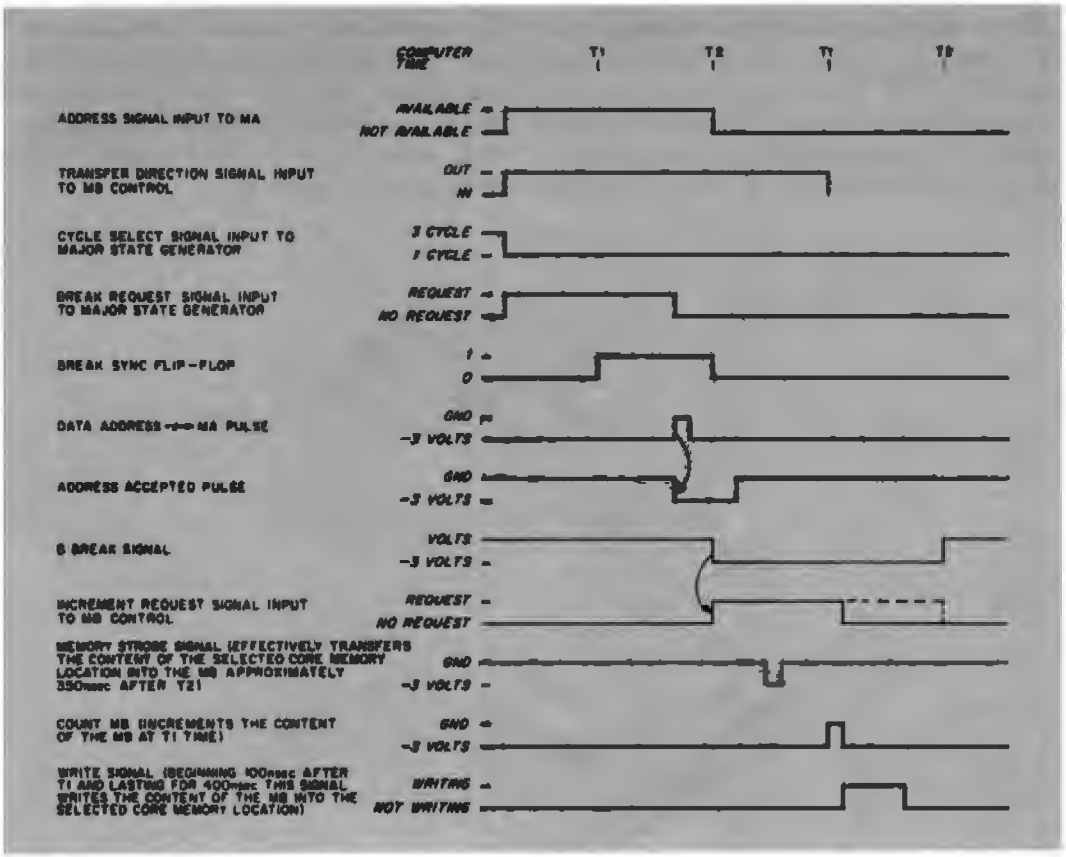

COMPUTER TIME
T1
T2
T1
T2
ADDRESS SIGNAL INPUT TO MA
AVAILABLE
NOT AVAILABLE
TRANSFER DIRECTION SIGNAL INPUT TO MB CONTROL
OUT
IN
CYCLE SELECT SIGNAL INPUT TO MAJOR STATE GENERATOR
3 CYCLE
1 CYCLE
BREAK REQUEST SIGNAL INPUT TO MAJOR STATE GENERATOR
REQUEST
NO REQUEST
BREAK SYNC FLIP-FLOP
1
0
DATA ADDRESS → MA PULSE
GND
-3 VOLTS
ADDRESS ACCEPTED PULSE
GND
-3 VOLTS
B BREAK SIGNAL
VOLTS
-3 VOLTS
INCREMENT REQUEST SIGNAL INPUT TO MB CONTROL
REQUEST
NO REQUEST
MEMORY STROBE SIGNAL (EFFECTIVELY TRANSFERS THE CONTENT OF THE SELECTED CORE MEMORY LOCATION INTO THE MB APPROXIMATELY 350nsec AFTER T2)
GND
-3 VOLTS
COUNT MB (INCREMENTS THE CONTENT OF THE MB AT T1 TIME)
GND
-3 VOLTS
WRITE SIGNAL (BEGINNING 100nsec AFTER T1 AND LASTING FOR 400nsec THIS SIGNAL WRITES THE CONTENT OF THE MB INTO THE SELECTED CORE MEMORY LOCATION)
WRITING
NOT WRITING

Figure 40 Memory Increment Data Break Timing Diagram

An interface for a device using memory increment data breaks must supply twelve Data Address signals, a Transfer Direction signal, a Cycle Select signal, and a Break Request signal to the computer data break facility as in an output transfer data break. In addition, a ground potential Increment MB signal must be provided at least 400 nsec before T1 time of the Break cycle. This signal can be generated in the device interface by AND combining the B Break computer output signal, the output transfer condition of the Transfer Direction signal, and the condition signal in the device that indicates that an increment operation should take place. When the computer receives this Increment MB signal, it forces the MB control element to generate a Count MB pulse at T1 time to increment the content of the MB.

The device interface logic shown in Figure 41 samples the normal Data Bit output signal for the most significant bit of the data word (BMB0) to determine if it overflows when incremented. If MB0 changes from the 0 to the 1 state when the data word is incremented, this logic requests a program interrupt to allow the program to take some appropriate action, such as incrementing a core memory counter for numbers above 4096, stopping the test to compile the data gathered to the current point in the operation, reinitializing the addressing, etc. The logic in the figure uses the select code of programmed data transfer operation to skip on the overflow condition to determine the cause of a program interrupt, to clear the overflow flip-flop, and to clear the device flag. Note that the devices that use data break transfers almost always use programmed data transfers to start and stop operation of the device, to initialize registers, etc., and do not rely on data break facilities alone to control their operations.

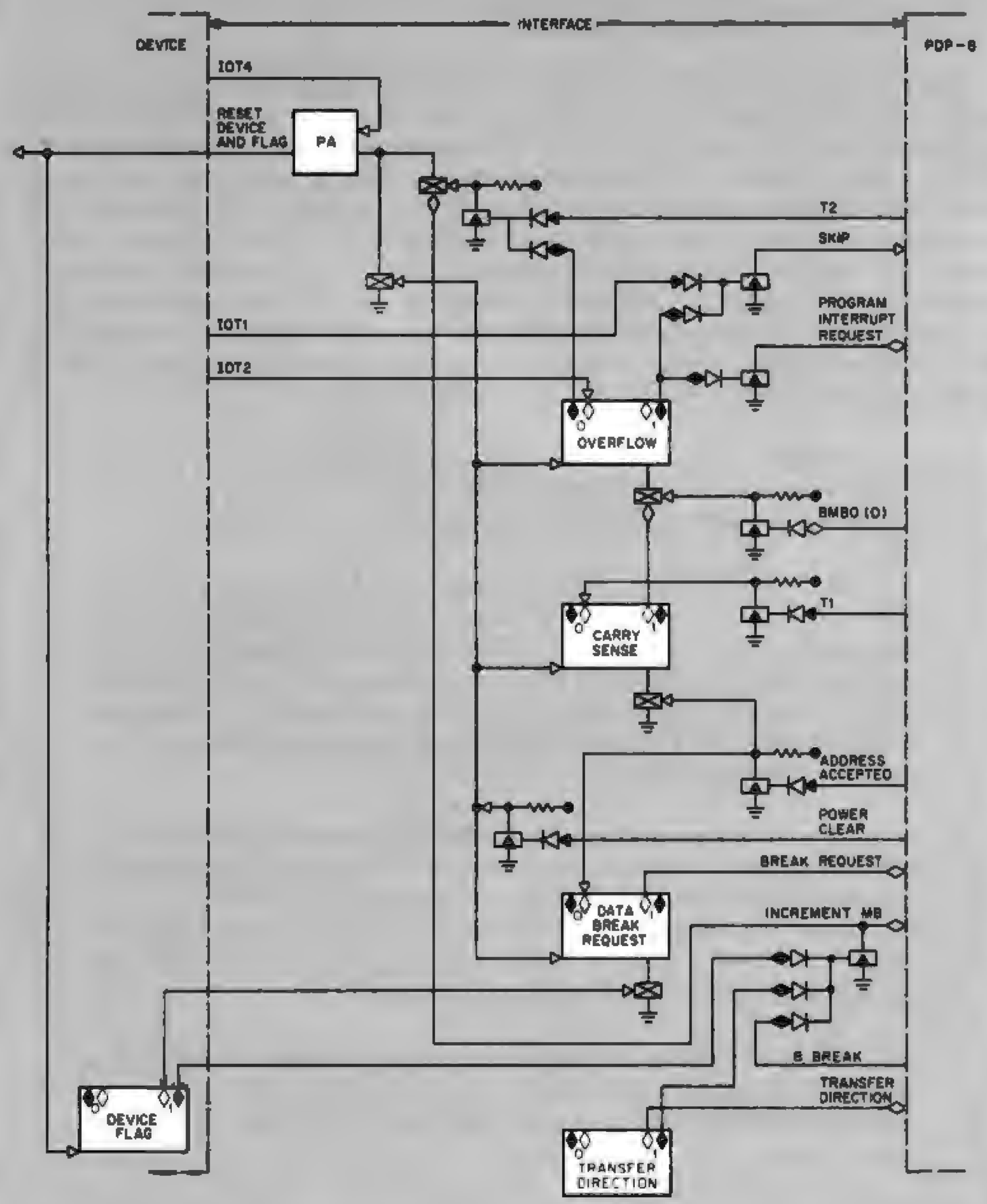


Figure 41 Device Interface Logic for Memory Increment Data Break

Three-Cycle Data Breaks

Timing of input or output 3-cycle data breaks is shown in Figure 42. The 3-cycle break uses the block transfer control circuits of the computer. The block transfer control provides an economical method of controlling the flow of data at high speeds between PDP-8 core memory and fast peripheral devices, e.g., drum, disc, magnetic tape and line printers, allowing transfer rates in excess of 220 kc.

The three-cycle data break facility provides separate current address and word count registers in core memory for the connected device, thus eliminating the necessity for flip-flop registers in the device control. When several devices are connected to this facility, each is assigned a different set of core locations for word count and current address, allowing interlaced operations of all devices as long as their combined rate does not exceed 220 kc. The device specifies the location of these registers in core memory, and thus the software remains the same regardless of what other equipment is connected to the machine. Since these registers are located in standard memory, they may be loaded and unloaded directly without the use of IOT pulses. In a procedure where a device requests to transfer data to or from core memory, the three-cycle data break facility performs the following sequence of operations:

a. An address is read from the device to indicate the location of the word count register. This address is always the same for a given device; thus it can be wired in and does not require a flip-flop register.

b. The content of the specified address is read from memory and 1 is added to it before rewriting. If the content of this register becomes 0 as a result of the addition, a WC Overflow pulse will be transmitted to the device. To transfer a block of N words, this register is loaded with − N during programmed initialization of the device. After the block has been fully transferred this pulse is generated to signify completion of the operation.

c. The next sequential location is read from memory as the current address register. Although the content of this register is normally incremented before being rewritten, an Increment CA Inhibit (+1 $\longrightarrow$ CA Inhibit) signal from the device may inhibit incrementation. To transfer a block of data beginning at location A, this register is program initialized by loading with A-1.

d. The content of the previously read current address is transferred to the MA to serve as the address for the data transfer. This transfer may go in either direction in a manner identical to the single-cycle data break system.

The three-cycle data break facility uses many of the gates and transfer paths of the single-cycle data break system, but does not preclude the use of standard data break devices. Any combination of three-cycle and single-cycle data break devices can be used in one system, as long as a multiplexer channel is available for each. Two additional control lines are provided with the three-cycle data break. These are:

a. Word Count Overflow. A standard 0.4-μsec negative computer output pulse is transmitted to the device when the word count becomes equal to zero.

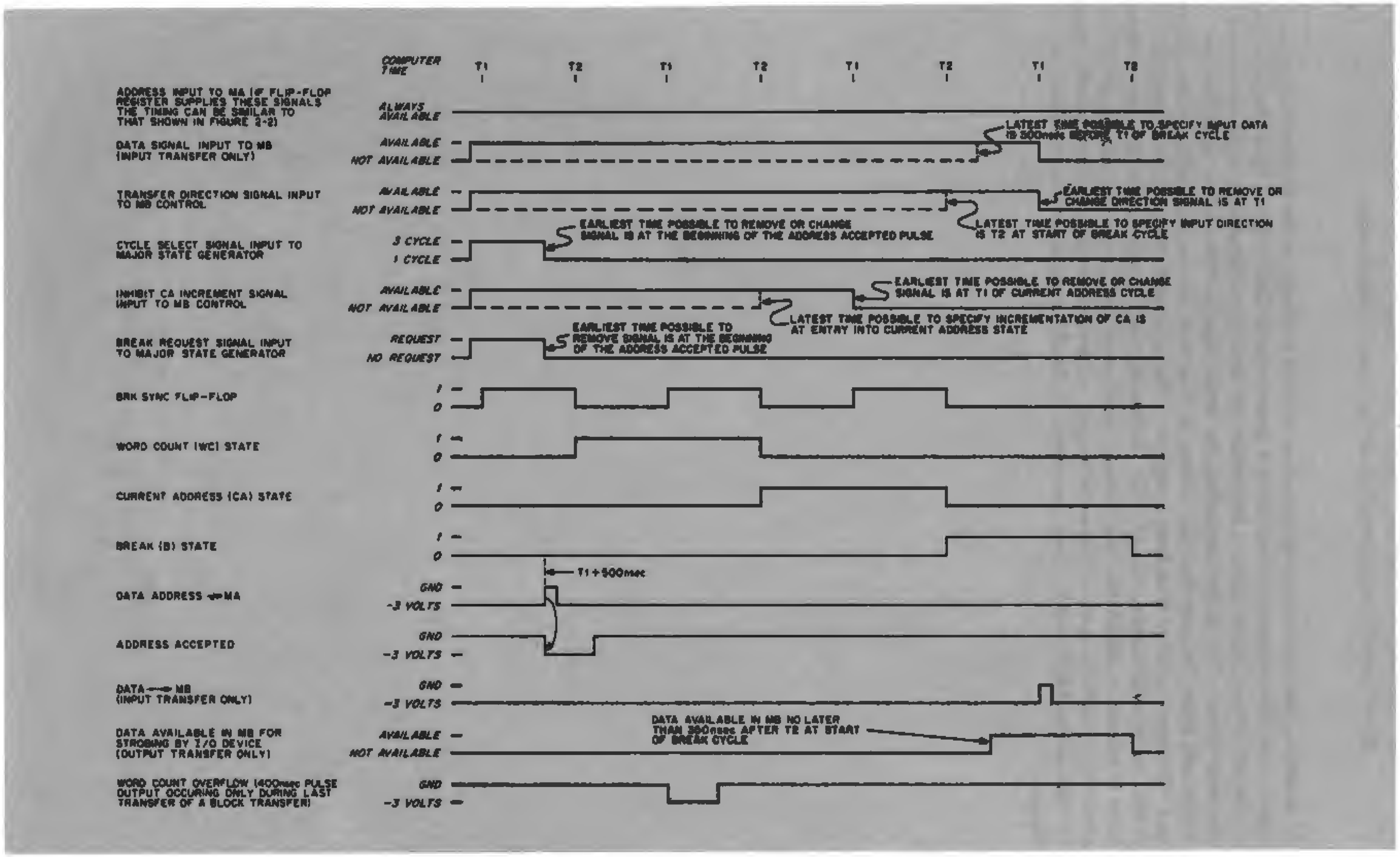

COMPUTER TIME
T1
T2
T1
T2
T1
T2
T1
T2
ADDRESS INPUT TO MA (IF FLIP-FLOP REGISTER SUPPLIES THESE SIGNALS THE TIMING CAN BE SIMILAR TO THAT SHOWN IN FIGURE 2-2)
ALWAYS AVAILABLE
DATA SIGNAL INPUT TO MB (INPUT TRANSFER ONLY)
AVAILABLE
NOT AVAILABLE
LATEST TIME POSSIBLE TO SPECIFY INPUT DATA IS 300nsec BEFORE T1 OF BREAK CYCLE
TRANSFER DIRECTION SIGNAL INPUT TO MB CONTROL
AVAILABLE
NOT AVAILABLE
EARLIEST TIME POSSIBLE TO REMOVE OR CHANGE DIRECTION SIGNAL IS AT T1
LATEST TIME POSSIBLE TO SPECIFY INPUT DIRECTION IS T2 AT START OF BREAK CYCLE
CYCLE SELECT SIGNAL INPUT TO MAJOR STATE GENERATOR
3 CYCLE
1 CYCLE
EARLIEST TIME POSSIBLE TO REMOVE OR CHANGE SIGNAL IS AT THE BEGINNING OF THE ADDRESS ACCEPTED PULSE
INHIBIT CA INCREMENT SIGNAL INPUT TO MB CONTROL
AVAILABLE
NOT AVAILABLE
EARLIEST TIME POSSIBLE TO REMOVE OR CHANGE SIGNAL IS AT T1 OF CURRENT ADDRESS CYCLE
LATEST TIME POSSIBLE TO SPECIFY INCREMENTATION OF CA IS AT ENTRY INTO CURRENT ADDRESS STATE
BREAK REQUEST SIGNAL INPUT TO MAJOR STATE GENERATOR
REQUEST
NO REQUEST
EARLIEST TIME POSSIBLE TO REMOVE SIGNAL IS AT THE BEGINNING OF THE ADDRESS ACCEPTED PULSE
BRK SYNC FLIP-FLOP
1
0
WORD COUNT (WC) STATE
1
0
CURRENT ADDRESS (CA) STATE
1
0
BREAK (B) STATE
1
0
T1 + 500nsec
DATA ADDRESS → MA
GND
-3 VOLTS
ADDRESS ACCEPTED
GND
-3 VOLTS
DATA → MB (INPUT TRANSFER ONLY)
GND
-3 VOLTS
DATA AVAILABLE IN MB FOR STROBING BY I/O DEVICE (OUTPUT TRANSFER ONLY)
AVAILABLE
NOT AVAILABLE
DATA AVAILABLE IN MB NO LATER THAN 300nsec AFTER T2 AT START OF BREAK CYCLE
WORD COUNT OVERFLOW (400nsec PULSE OUTPUT OCCURING ONLY DURING LAST TRANSFER OF A BLOCK TRANSFER)
GND
-3 VOLTS

Figure 42 Three-Cycle Data Break Timing Diagram

b. Increment CA Inhibit. When ground potential, this device-supplied signal inhibits incrementation of the current address word.

In summary, the three-cycle data break is entered similarly to the single-cycle data break, with the exception of supplying a ground-level Cycle Select signal to allow entry of the WC (Word Count) state to increment the fixed core memory location containing the word count. The device requesting the break supplies this address as in the one-cycle data break, except that this address is fixed and can be supplied by wired ground and −3v signals, rather than from a register. The sole restriction on this address is that it must be an even number (bit 11 = 0). Following the WC state a CA (Current Address) state is entered in which the core memory location following the WC address (bit 11 = 1 after PC +1 ⇒ PC) is read, incremented by one, restored to memory, and used as the transfer address (by MB ⇒ MA). Then the normal B (Break) state is entered to effect the transfer.

CHAPTER 4
DIGITAL LOGIC CIRCUITS

PDP-8 is constructed of Digital FLIP CHIP modules. The Digital Logic Handbook, C-105, describes more than 100 of these modules, all their component circuits, and the associated accessories, i.e., power supplies and mounting panels. The user should study this catalog carefully before beginning the design of a special interface.

Basic Digital Circuits

The basic component circuits used in the interface of the PDP-8 as well as in most FLIP CHIP modules are inverters, diode gates, diode-capacitor-diode (DCD) gates, pulse amplifiers, and bus drivers.

INVERTERS

An inverter circuit is analogous to a switch. Figure 43 shows the basic inverter circuit. If the inverter base is at −3v and the emitter is at ground (most transistors in FLIP CHIP modules have a permanent connection from the emitter to ground), the PNP transistor is saturated and a conduction path is established between the emitter and collector output. Conversely, when the input at the base of the inverter is at ground, the transistor is cut off and the output at the collector goes negative. Collector points can connect to a load within the module or a remote load at the driven circuit. Internal collector loads are clamped at −3v. In series-R modules the load resistor is 7.5K to draw 2 ma, and in series-S modules the load resistor is 3K to draw 5 ma.

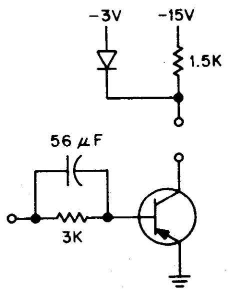


Figure 43 Inverter Circuit Schematic Diagram

Flip-flops are cross-coupled inverters using the same circuit. The state of a flip-flop is changed by driving the base of the conducting transistor to ground, thereby turning it off. Figure 44 shows the direct-set input circuit of the Type R210 PDP-8 Accumulator module.

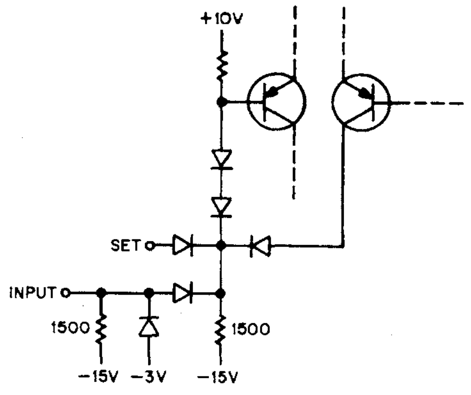


Figure 44 Direct-Set Input Circuit Schematic of the R210 PDP-8 Accumulator

DIODE GATES

The diode gate is used in the R and S series to combine, amplify, invert, and standardize the signals which represent various logic functions. Figure 45 is a circuit schematic diagram of a simple diode gate with one input.

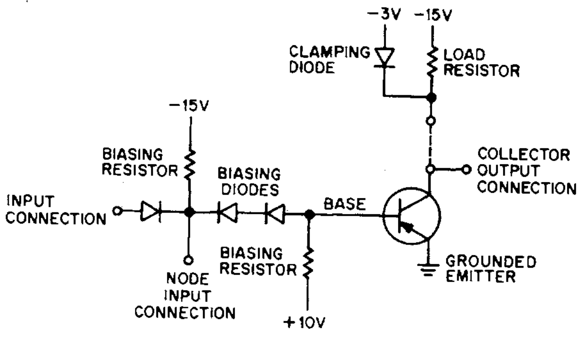


Figure 45 Single-Input Diode Gate Circuit Schematic

When the input is negative, the node point is also negative and current flows from the transistor emitter through the biasing diodes and the biasing resistor to −15v. As a result, the PNP transistor is turned on forming a short circuit between the collector and the emitter. Thus, when the input voltage is negative, the output voltage is ground potential. Since the output is from a saturated transistor, it has a low output impedance and good driving power.

When the diode gate input voltage is ground, the biasing diodes and the resistor, which is connected to the +10v supply, hold the transistor base more positive than the emitter, and the transistor is turned off. The output is then an open circuit, and it follows the voltage of any other circuit connected to it.

If the load resistor and clamp diode are attached to the transistor collector, they serve as a voltage source and hold the output at −3v while the transistor is off. When the transistor is on, the diode is cut off and the load resistor follows the output to ground.

The single-input diode gate therefore has three functions:

a. It inverts the input signal.

b. It standardizes the output voltage to −3v or ground (if the clamped load diode and resistor are connected).

c. Since the output current available from the transistor is much greater than the required input current, the diode gate amplifies.

A fourth function, gating, obtained by adding more diode inputs to the node point, is illustrated in Figure 46.

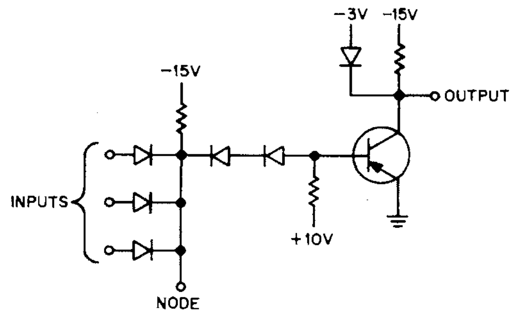


Figure 46 Multiple-Input Diode Gate Circuit Schematic

The node terminal is at approximately the same voltage as the most positive input. Thus, when any input terminal is grounded, the node terminal is also at ground and the circuit output is at −3v. If all of the inputs are negative, the node terminal is negative and the circuit output is at ground.

Figure 47 shows how gating functions can be performed by wiring together two or more diode gate outputs and one load resistor. When any input is negative, it saturates the corresponding transistor and forces the output line to ground. If all inputs are at ground, all of the transistors are open circuits and the output voltage, determined by the clamped load resistor, is −3v.

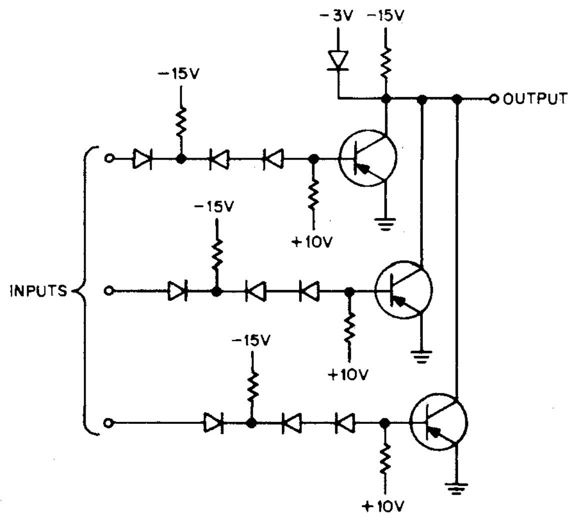


Figure 47 Parallel-Connected Diode Gate Circuit Schematic

It is possible to use the basic diode gate to construct very complex logical functions. A drawing showing all of the circuit components, however, would be difficult both to draw and read, so for this reason logic diagrams use a shorthand notation, representing one or more components as a single functional unit. Figure 48 shows a diode gate in the conventional way. The transistor circuit, including the biasing resistors and diodes, appears as a simple rectangle with an arrowhead indicating the direction of the transistor emitter. The load resistor appears as a resistor with a large dot at the top indicating that it is diode clamped to −3v.

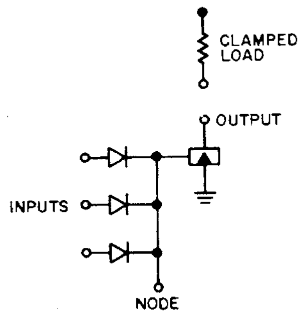


Figure 48 Diode Gate Logic Symbol

Diamonds show assertion input and output voltage levels. A solid diamond indicates a −3v level, and an open diamond indicates a ground level. In the 2-input diode gate of Figure 49, for example, if input A and input B are both negative, the output is at ground. If either A or B is at ground, the output is negative.

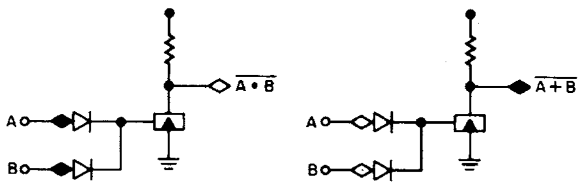


Figure 49 Logic Operations Performed by Diode Gates

DIODE-CAPACITOR-DIODE GATES

The diode-capacitor-diode (DCD) gate is used to standardize the input to various units such as flip-flops, delays, and pulse amplifiers. It provides logical isolation between pulse and level inputs and produces a logical delay which is essential for sampling flip-flops at the same time they are being changed. It also acts as a logical AND gate since both pulse and level inputs must meet certain requirements for a signal to appear at the output. Either positive pulses or positive-going level changes (both −3v to ground) may be used as the pulse input.

A schematic drawing of a DCD gate is shown in Figure 50. If the level input is held at ground and the pulse input is held at −3v, the capacitor becomes charged after the set-up time has passed. If the pulse input then suddenly goes to ground, a positive-going pulse appears at the output. There is delay at the level input, but the pulse input goes to the output without delay. Even if the level input changes simultaneously with a positive transition at the pulse input, the delay acts as a temporary memory: the pulse input is gated according to the level input that existed during the interval before the pulse.

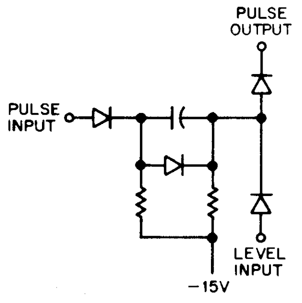


Figure 50 Diode-Capacitor-Diode Gate Circuit Schematic

An X in the rectangle distinguishes the symbol for the DCD gate (Figure 51) from the diode gate. The output is at the top, the delayed (level) input is at the bottom, and the differentiating (level change or pulse) input is on the side. An arrowhead rather than a diamond indicates the input signal to be differentiated, whether a level change or a pulse. The pulse symbols are hollow when positive-going and solid when negative-going. In the DCD gate, the pulse input must be positive-going.

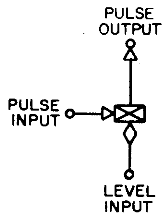


Figure 51 Diode-Capacitor-Diode Gate Logic Symbol

Since the same pulse may drive many DCD gates, the side of the rectangle opposite the pulse may be used to show a continuation of the same line, as in Figure 52. The illustration on the left below is a simplified version of the identical logical configuration on the right.

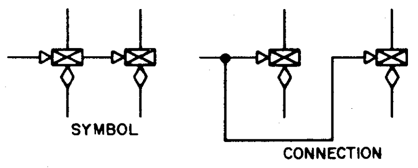


Figure 52 Parallel-Connected Trigger Pulse to DCD Gates

PULSE AMPLIFIERS

The PDP-8 uses two types of pulse amplifier circuits. Modules such as the Type S603 contain monostable multivibrators (one-shots) to produce a standard 100-nsec negative output pulse. Modules such as the Type W640 use a transformer-coupled pulse-forming circuit to produce standard 400-nsec or 1-μsec pulses. The time required to saturate the inter-stage coupling transformer determines output pulse duration, and grounding the appropriate side of the output pulse transformer determines output polarity. Figure 53 shows schematically the final stages of this type of pulse amplifier circuit.

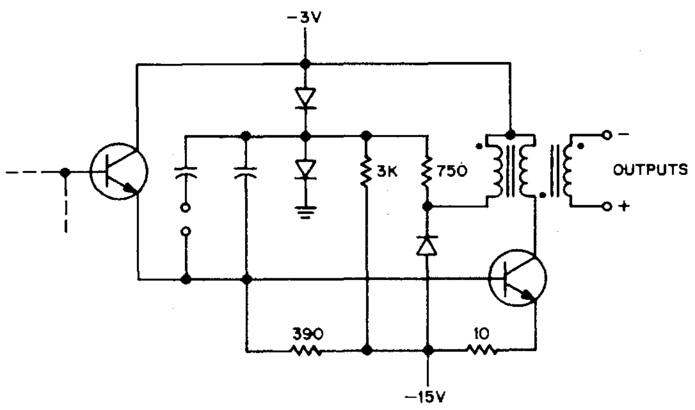


Figure 53 Pulse Amplifier Output Circuit Schematic

BUS DRIVERS

Bus driver circuits that drive a heavy load contain a push-pull output stage, as shown in Figure 54. The Type R650 module uses a dc, inverting, amplifier circuit with a timing capacitor to control rise and fall times. With the capacitor shunted to ground, typical rise and fall time is 700 nsec; with this capacitor floating, typical rise and fall time is 50 nsec. A resistor output terminal is provided for driving coaxial cable.

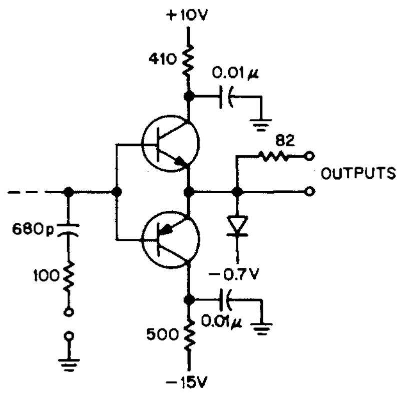


Figure 54 Bus Driver Output Circuit Schematic

Interface Circuits of the Computer

Circuit modules of the PDP-8 receiving input signals and supplying output signals are as follows:

Input		Output	
Type	Name	Type	Name
A502	Difference Amplifier	R650	Bus Driver
R123	Diode Gate	S107	Inverter
R210	PDP-8 Accumulator	W640	Pulse Amplifier
R211	MA, MB, and PC		
S107	Inverter		
S111	Diode Gate		
S151	Binary-to-Octal Decoder		
S203	Triple Flip-Flop		
S603	Pulse Amplifier		

A502 DIFFERENCE AMPLIFIER

Only the PDP-8 containing a Analog-to-Digital Converter Type 189 option uses the A502 Difference Amplifier. The A502 is a high-speed difference amplifier which compares two input voltages and indicates which of the two is the more negative. The comparator has a resolution of 1 mv, and an input range of 0 to −10v. The maximum combined error due to a change in the common input voltage from 0 to −10v and a 20°C temperature change is 5 mv equivalent input offset. Two potentiometers allow adjustment of the zero set and common balance.

The comparator switching time is less than 250 nsec for a +10 mv square wave. The switching time is also less than 250 nsec when one input is at −5.00v and the other is switched from groundto −5.02v. For finer resolution, the switching time is increased. When the comparator is driven from a high impedance, fast switching source, such as a digital-to-analog converter, time should also be allowed for transients to settle.

The 0 to −10v input draws up to 1µa depending on the relative polarity of the two voltage inputs. The maximum current difference between positive and negative input voltages is 1 µa. The difference input capacitance is 75 pf.

R123 DIODE GATE

The Data Line Interface Type 681 option uses the R123 as an interface module only where it receives the Line(1) Teletype signal. The R123 contains six 2-input negative NAND gates, with no load resistors for the gates since they usually drive the input of a flip-flop register. Standard ground and −3v levels with a duration of at least 100 nsec drive the input. Input load is 1 ma shared among the inputs that are at ground.

Standard ground and −3v levels are produced at the output. Each output can drive 20 ma of load at ground. The output terminals of diode gates may be connected in parallel. One clamped load is sufficient for parallel outputs when using less than 2 ft of wire. If the wire exceeds this length, additional clamped loads may be necessary for a sufficiently fast fall time in higher frequency applications. Two gates in parallel, driven by the same signal, can drive 38 ma at ground (20 ma each, less the 2-ma clamped load). Gates in parallel, not driven by the same signal, can drive 20 ma at ground minus 2 ma for each clamped load used.

R210 PDP-8 ACCUMULATOR

The R210 is a double-height module that serves as one bit of the accumulator register, with all of the necessary input gates. Bus driver modules from this module apply outputs to the interface. Interface input connections consist of the direct-set connection used as the input bus for programmed data transfers. This input accepts standard levels of ground and −3v. A ground level of 400-nsec minimum duration activates the input. Input load is 11 ma at ground, and when not in use, the direct-set input terminal must be at −3v.

R211 MA, MB, AND PC

The R211 is a double-height module that contains one bit of the memory address, memory buffer, and program counter registers of the PDP-8. Connections from this module to the interface receive the Data Address and Data Bit signals supplied by external equipment that uses the data break facility. These signals are received as the level input to three DCD gates on each module. Two of these gates serve as a complementary input to the MA for the Data Address signal. An inverter in the module precedes the DCD gate on the clear side of the MA flip-flop; so the single address signal input either sets or clears the flip-flop. The third DCD gate connects to the 1 side of the MB flip-flop to receive the Data Bit signal directly. Control circuits within the computer provide trigger pulses to all these gates. Input signals must be at ground level to designate a binary 1 or −3v to designate a binary 0. These signals must precede the trigger pulse by at least 400 nsec. Each Data Address input represents 12 ma of load. Each Data Bit input represents 11 ma of load.

R650 BUS DRIVER

The AC, MB, Break, and Run(1) output signals are buffered by bus driver circuits of Type R650 modules before connection to the interface.

The R650 contains two inverting bus drivers for driving heavy current loads to either ground or negative voltages. The bus drivers operate at frequencies up to 2 mc with typical rise and fall times of 25 nsec. The typical total transition times are 60 nsec for output rise and 65 nsec for output fall.

By grounding pin H or S the rise and fall time can be increased to avoid ringing on exceptionally long lines. The driver then operates at frequencies up to 500 kc with typical rise delay of 50 nsec, fall delay of 50 nsec, and total transition time of 800 nsec for output rise and 700 nsec for output fall. Terminal K or U can be used for driving coaxial cable.

The direct output (terminals J and T) drives 20 ma of external load at either ground or −3v. The resistor output (terminals K and U) drives 90-ohm coaxial cable such as RG-62-U. This output drives 5 ma of external load at either ground or −3v. The direct output connects to the interface connectors of the PDP-8.

S107 INVERTER

In the basic computer, Type S107 Inverter modules receive the Increment MB and Cycle Select signals used with the data break. In the Memory Extension Control Type 183 option the S107 Inverter receives the Address Extension 1-3 signals, and in the Data Line Interface option Type 681 it receives the Teletype Line(1) signal. Within these two options the Type S107 supplies the Data Field and Teletype Instruction (TT Inst) output signals.

The S107 Inverter contains seven inverter circuits with single-input diode gates. Six of the circuits are used for single-input inversion; the seventh circuit can be used for gating by tying additional diode input networks to its node terminal. Clamped load resistors of 5 ma are a permanent part of each inverter. Typical output total transition times are 60 nsec for rise and 50 nsec for fall.

The diode input accepts standard level inputs of ground and −3v that are a minimum of 100 nsec in duration. This 1 ma input load is shared among the inputs at ground. The node terminal input accepts only R001 or R002 Diode Network output connections, or their equivalents. The combined length of all leads attached to the node terminal must not exceed 6 inches. Input signal and load characteristics for diode networks are the same as those given for the diode input above.

Output signals from the inverters are standard levels of ground and −3v. Each inverter drives 15 ma of load at ground. Output terminals of inverters may be connected in parallel. Only one clamped load resistor is needed at the output when less than 2 feet of wire is used. If the wire exceeds this length, additional clamped load resistors may be necessary for a fast enough fall time in high frequency applications.

S111 DIODE GATE

Type S111 modules in the computer receive the Program Interrupt Request and Transfer Direction (Data In) signals.

The S111 Diode Gate contains three diode gates, each connected to a transistor inverter. The gate operates as a NAND for negative inputs and as a NOR for ground inputs. Each gate has three input terminals: two are connected to diodes; a third is connected directly to the node point of the diode gate. The third terminal allows the number of input diodes to be increased by adding external diode networks such as the R001 or R002 modules. External diodes must be connected in the same direction as the diodes in the S111. Typical output total transition times are 60 nsec for rise and 50 nsec for fall.

Input signals to the diode terminals must be standard levels of ground or −3v, and must have a minimum duration of 100 nsec. Input load is 1 ma shared among the inputs at ground. Input signals to the node terminals accept only connections from Type R001 or R002 Diode Network modules, or their equivalents. The maximum combined length of all leads attached to a node terminal is 6 inches. Input signal load is similar to the diode input.

S151 BINARY-TO-OCTAL DECODER

A Type S151 module (in parallel with a Type S107) receives Address Extension 1-3 signals supplied to the Memory Extension control option Type 183.

This S151 decodes binary information from three flip-flops into octal form. When the enable input is at ground, the selected output line is at ground and the other seven outputs are at −3v. When the enable input is at −3v, all outputs are at −3v. The internal gates are similar to those in the S111. The enable input is the common emitter connection of the output inverters. Typical total transition times are 75 nsec for output rise and 60 nsec for output fall.

Standard input levels are −3v and ground, 100 nsec minimum duration. The 2.3 ma input load is shared among the inputs at ground.

S203 TRIPLE FLIP-FLOP

The S203 contains three identical flip-flops. Each flip-flop has a direct clear and a DCD gate for conditional readin. The level input to one of these gates connects to the interface to receive the Data Break Request signal. This input receives standard ground and −3v levels. The conditioning level must be ground level to condition the gate and to request a break. This conditioning must occur at least 400 nsec before an internal computer pulse triggers the gate. The level input represents a 2-ma load at ground.

S603 PULSE AMPLIFIER

Pulse amplifiers of Type S603 modules receive the Clear AC and Skip inputs of the PDP-8. An S603 module contains three pulse amplifier circuits for power amplification and standardizing pulses in amplitude and width. Each amplifier produces standard 100-nsec negative pulses each time the input triggers from the diode or DCD gate inputs. Both interface input signals flow to the diode input of a pulse amplifier. The pulse amplifier can accommodate input pulses at any frequency up to 2 mc. Delay through the pulse amplifier is approximately 50 nsec. The diode input receives standard 100-nsec pulses (−3v to ground) or positive-going level changes (−3v to ground) with a rise time no longer than 60 nsec. The input level must be returned to −3v for at least 400 nsec before another input may occur at either the diode or DCD gate input. The diode input represents a 1-ma load at ground.

W640 PULSE AMPLIFIER

The IOP pulses, BT1 and BT2 timing pulses, and the B Power Clear pulses are supplied to the interface as outputs from Type W640 modules. The W640 module contains three ungated pulse amplifiers capable of producing either 1-μsec or 400-nsec pulses. In the normal PDP-8, these outputs are standard negative 400-nsec pulses. Each output drives 10 ma of load (equivalent to 10 inverter bases such as the B104 or S111). These amplifiers should not be used without a terminating resistor; typical values are 47 to 150 ohms.

Interface Circuits of Peripheral Equipment

Several FLIP CHIP circuit modules are of particular interest in the design of equipment to interface with the PDP-8. Chief among these are the W103 Device Selector and the R123 Diode Gate. In addition to these, the following modules serve special interface applications:

Type	Name	Application
B681	Power Inverter	General purpose digital logic to supply or operate devices requiring up to 32 ma.
B684	Bus Driver	Provides output driving current of up to 40 ma.
W040	Solenoid Driver	Provides driving capability for electromechanical devices such as counters, clutches, and other solenoid-actuated equipment requiring currents of up to 600 ma at −2.5 to −70v.
W051	Driver	Provides driving capability for devices that require up to 100 ma at 0 to −15v.
W501	Schmitt Trigger	Provides level conversion from voltages of ±10v to the standard levels required by DEC modules.
W510	Positive Input Converter	Converts positive voltage signal levels to ground and −3v levels for DEC modules.

Type	Name	Application
W600	Negative Output Converter	Converts standard DEC logic levels of ground and −3v to levels of ground and a negative voltage between −1 and −15v determined by the level of an external supply.
W601	Positive Output Converter	Converts standard DEC logic levels of ground and −3v to levels of ground and some positive voltage between +1 and +20v as determined by the level of an external supply.

W103 DEVICE SELECTOR

The W103 selects an input/output device according to the code in the instruction word (being held in the memory buffer during the IOT cycle). Figure 55 shows the logic circuits of the W103 module.

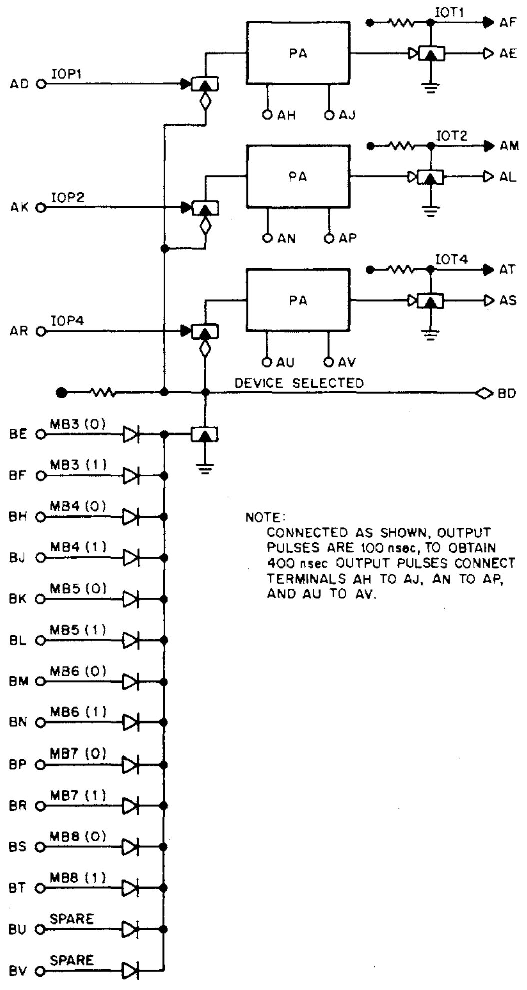


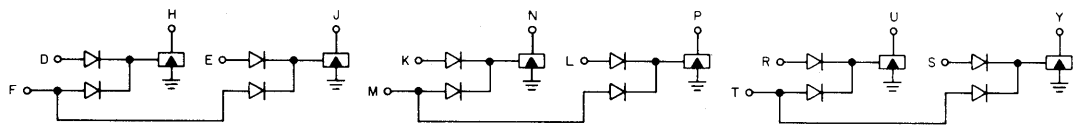
Figure 55 Device Selector W103 Logic Circuit

The twelve input diodes permit selection of any arbitrary 6-bit code, and decode the number held in MB bits 3 through 8. When the proper enabling code arrives at the diode gate, the three input gates driving the three pulse amplifiers are enabled and permit passage of the programmed IOP pulses. To establish a code on the module, the six unnecessary diodes are disabled by snipping one of their leads or removing them altogether. If MB bit 3 is a binary 1 to set up the correct code, the diode going to the binary 0 side of MB bit 3 is disabled. Two spare diodes are included for additional gating flexibility.

Three pulse amplifiers produce R-series 100-nsec positive-going output pulses. Inverted pulse amplifier output pulses are provided for gates (such as the Type R111 Diode Gate) which require negative or negative-going pulses. Jumper terminals on each pulse amplifier establish a pulse duration of 400 nsec for the output pulse. It is recommended that the 400-nsec pulse duration be used when transmitting the pulse over long distances. The 400-nsec pulse also clears R-series flip-flops whose carry gates are permanently enabled. The positive pulse output of each pulse amplifier is rated 65 ma of external load at ground; the negative output is rated 15 ma at ground (when driving a load connected to -15v). These outputs are not designed to drive loads when at -3v (loads connected to ground). To drive this type of load, a clamped load resistor must be connected to the pulse amplifier output terminal to supply the current.

R123 DIODE GATE

This module contains six 2-input NAND gates for negative levels and is useful for transferring data into or out of the PDP-8 accumulator. Standard DEC negative levels or 0.4 microsecond negative pulses such as those from the W103 Device Selector can be used as input signals. Input load per gate is 1 ma shared among the inputs at ground.



NOTES:
1. STROBE PULSE INPUT TO TERMINALS F, M, AND T WHICH ARE CONNECTED IN COMMON WHEN USED AS A BUS GATE
2. DATA BIT INPUTS TO TERMINALS D, E, K, L, R, AND S
3. TWO MODULES ARE REQUIRED TO STROBE A 12-BIT WORD

Figure 56 Diode Gate R123 Logic Circuit

Two R123 modules provide sufficient gating to transfer one 12-bit word into the accumulator. If more gates are needed to load the AC from several sources, the output terminals can be OR connected by bussing together additional gate collectors.

CHAPTER 5
INTERFACE CONNECTIONS

All interface connections to the PDP-8 are made at assigned module receptacle connectors in the left (memory-M) or right (processor-P) mounting frame (door). Capital letters designate horizontal rows of modules within a mounting frame from top to bottom. Module receptacles are numbered from left to right as viewed from the wiring side (right to left from the module side). Terminals of a connector or module are assigned capital letters from top to bottom, omitting G, I, O, and Q. Therefore, terminal PE2H is in the right mounting frame (P), the fifth row from the top (E), the second module from the left (2), and the seventh terminal from the top of the connector (H).

The module receptacles and assigned assigned use for interface signal connections are:

Receptacle	Signal Use
PE2	AC 0-8 inputs
PE3	Data Address 0-8 inputs
PE4	Data Bit 0-8 inputs
PF2	AC 9-11, Skip, Clear AC inputs and Run output
PF3	Data Address 9-11 inputs, and Address Accepted and B Break outputs
PF4	Data Bit 9-11 inputs
ME30	Address Extend 1, 2, 3 inputs and Data Field 0-2 outputs
ME34	BAC 0-8 outputs
ME35	BMB 0-5 outputs
MF34	BAC 9-11, IOTs, BT1, BT2A, and B Power Clear outputs
MF35	BMB 6-11 outputs

Terminals C, F, J, L, N, R, and U of these receptacles are grounded within the computer and terminals D, E, H, K, M, P, S, T, and V carry signals. These terminals mate with Type W011 Signal Cable Connectors at each end of 93-ohm coaxial cable.

Interface connection to the PDP-8 can be established for all peripheral equipment by making series cable connections between devices. In this manner only one set of cables is connected to the computer and two sets are connected to each device: one receiving the computer connection from the computer itself or the previous device; and one passing the connection to the next device. Where physical location of equipment does not make series bus connections feasible, or when cable length becomes excessive, additional interface connectors can be provided near the computer.

All logic signals passing between the PDP-8 and the input/output equipment are standard DEC levels or standard DEC pulses. Logic signals have mnemonic names that indicate the condition represented by assertion of the signal. Standard levels are either ground potential (0.0 to −0.3v), designated by an open diamond (——◇) or are −3v (−3.0 to −4.0v), designated by a solid diamond (——◆). Standard pulses in the positive direction are designated by an open triangle (——▷) and negative pulses are designated by a solid triangle (——▶). Pulses originating in R or S series modules are positive-going pulses which start at −3v, go to ground for 100 nsec, then return to −3v. Pulses originating in W series modules are always negative, are always referenced to ground, are 2.5v in amplitude (2.3 to 3.0v) with a 2v overshoot, and are of 400-nsec duration.

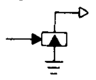
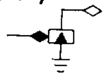
The following tables present cable connections and logic circuit identification information for PDP-8 interface signals. Computer input signals that must drive the interface bus to ground (data inputs to the AC, Clear AC, Skip, and Interrupt Request) must be connected to the collector of a grounded-emitter transistor, and so can be considered transistor-gated negative pulses () or levels ().

TABLE 5 PROGRAMMED DATA TRANSFER INPUT SIGNALS

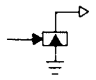
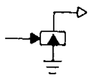
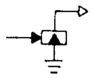
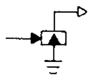
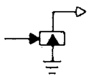
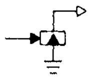
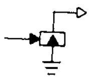
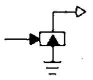
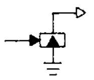
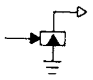
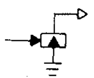
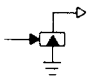
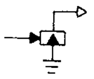
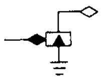
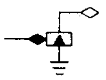
Signal	Symbol	Interface Connection	Module Terminal	Module Type
AC 0		PE2D	PA7E	R210
AC 1		PE2E	PA8E	R210
AC2		PE2H	PA9E	R210
AC 3		PE2K	PA10E	R210
AC 4		PE2M	PA11E	R210
AC 5		PE2P	PA12E	R210
AC 6		PE2S	PA13E	R210
AC 7		PE2T	PA14E	R210
AC 8		PE2V	PA15E	R210
AC 9		PF2D	PA16E	R210
AC 10		PF2E	PA17E	R210
AC 11		PF2H	PA18E	R210
Clear AC		PF2P	PA19J	S603
Interrupt Request		PF2M	PD36K	S111
Skip		PF2K	PB21V	S603

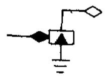
TABLE 6 PROGRAMMED DATA TRANSFER OUTPUT SIGNALS

Signal	Symbol	Interface Connection	Module Terminal	Module Type
BAC 0 (1)	—◇	ME34D	ME26J	R650
BAC 1 (1)	—◇	ME34E	ME26T	R650
BAC 2 (1)	—◇	ME34H	ME27J	R650
BAC 3 (1)	—◇	ME34K	ME27T	R650
BAC 4 (1)	—◇	ME34M	ME28J	R650
BAC 5 (1)	—◇	ME34P	ME28T	R650
BAC 6 (1)	—◇	ME34S	MF26J	R650
BAC 7 (1)	—◇	ME34T	MF26T	R650
BAC 8 (1)	—◇	ME34V	MF27J	R650
BAC 9 (1)	—◇	MF34D	MF27T	R650
BAC 10 (1)	—◇	MF34E	MF28J	R650
BAC 11 (1)	—◇	MF34H	MF28T	R650
IOP 1	—▶	MF34K	MC31H	W640
IOP 2	—▶	MF34M	MC31N	W640
IOP 4	—▶	MF34P	MC31U	W640
BMB 3 (0)	—◇	ME35K	MC27T	R650
BMB 3 (1)	—◇	ME35M	MC28J	R650
BMB 4 (0)	—◇	ME35P	MC28T	R650
BMB 4 (1)	—◇	ME35S	MC29J	R650
BMB 5 (0)	—◇	ME35T	MC29T	R650
BMB 5 (1)	—◇	ME35V	MD25J	R650
BMB 6 (0)	—◇	MF35D	MD25T	R650
BMB 6 (1)	—◇	MF35E	MD26J	R650
BMB 7 (0)	—◇	MF35H	MD26T	R650
BMB 7 (1)	—◇	MF35K	MD27J	R650
BMB 8 (0)	—◇	MF35M	MD27T	R650
BMB 8 (1)	—◇	MF35P	MD28J	R650

TABLE 7 DATA BREAK TRANSFER INPUT SIGNALS

Signal	Symbol	Interface Connection	Module Terminal	Module Type
Data Address 0 (1)	—◇	PE3D	PC7R	R211
Data Address 1 (1)	—◇	PE3E	PC8R	R211
Data Address 2 (1)	—◇	PE3H	PC9R	R211
Data Address 3 (1)	—◇	PE3K	PC10R	R211
Data Address 4 (1)	—◇	PE3M	PC11R	R211
Data Address 5 (1)	—◇	PE3P	PC12R	R211
Data Address 6 (1)	—◇	PE3S	PC13R	R211
Data Address 7 (1)	—◇	PE3T	PC14R	R211
Data Address 8 (1)	—◇	PE3V	PC15R	R211
Data Address 9 (1)	—◇	PF3D	PC16R	R211
Data Address 10 (1)	—◇	PF3E	PC17R	R211
Data Address 11 (1)	—◇	PF3H	PC18R	R211
Data Bit 0 (1)	—◇	PE4D	PD7M	R211
Data Bit 1 (1)	—◇	PE4E	PD8M	R211
Data Bit 2 (1)	—◇	PE4H	PD9M	R211
Data Bit 3 (1)	—◇	PE4K	PD10M	R211
Data Bit 4 (1)	—◇	PE4M	PD11M	R211
Data Bit 5 (1)	—◇	PE4P	PD12M	R211
Data Bit 6 (1)	—◇	PE4S	PD13M	R211
Data Bit 7 (1)	—◇	PE4T	PD14M	R211

TABLE 7 DATA BREAK TRANSFER INPUT SIGNALS (continued)

Signal	Symbol	Interface Connection	Module Terminal	Module Type
Data Bit 8 (1)	—◇	PE4V	PD15M	R211
Data Bit 9 (1)	—◇	PF4D	PD16M	R211
Data Bit 10 (1)	—◇	PF4E	PD17M	R211
Data Bit 11 (1)	—◇	PF4H	PD18M	R211
Break Request		PF3K	PC32J	S203
Transfer Direction	—◆*	PF3M	PD23E	S111
Increment MB	—◆**	PF3T	PD31M	S107
Cycle Select	—◇	PF4K	PE7S	S107
Increment CA	—◆	PF4M	PE10F	R121

*Direction is into PDP-8 when signal is −3v, out of PDP-8 when ground potential.

**The Increment MB input to the PDP-8 must be the output of a gating circuit that enables generation of the ground level signal only when the B Break signal is present.

TABLE 8 DATA BREAK TRANSFER OUTPUT SIGNALS

Signal	Symbol	Interface Connection	Module Terminal	Module Type
BMB 0 (1)	—◇	ME35D	MC26J	R650
BMB 1 (1)	—◇	ME35E	MC26T	R650
BMB 2 (1)	—◇	ME35H	MC27J	R650
BMB 3 (1)	—◇	ME35M	MC28J	R650
BMB 4 (1)	—◇	ME35S	MC29J	R650
BMB 5 (1)	—◇	ME35V	MD25J	R650
BMB 6 (1)	—◇	MF35E	MD26J	R650

TABLE 8 DATA BREAK TRANSFER OUTPUT SIGNALS (continued)

Signal	Symbol	Interface Connection	Module Terminal	Module Type
BMB 7 (1)	—◇	MF35K	MD27J	R650
BMB 8 (1)	—◇	MF35P	MD28J	R650
BMB 9 (1)	—◇	MF35S	MD28T	R650
BMB 10 (1)	—◇	MF35T	MD29J	R650
BMB 11 (1)	—◇	MF35V	MD29T	R650
B Break	—◆	PF3P	PE8T	R650
Address Accepted	—▷	PF3S	PF10H	W640
WC Overflow	—▶	PF4P	PF10N	W640

TABLE 9 MISCELLANEOUS INPUT SIGNALS

Signal	Symbol	Interface Connection	Module Terminal	Module Type
ADDR Extension 1	—◇	ME30D	ME8K, MC3K	S107, S151
ADDR Extension 2	—◇	ME30E	ME8H, MC3E	S107, S151
ADDR Extension 3	—◇	ME30H	ME8E, MC3J	S107, S151
Analog In*	—▶	Special BNC	PE11N	A502

*Input to Analog-to-Digital Converter Type 189 is made to BNC connector on back of processor fan mounting.

TABLE 10 MISCELLANEOUS OUTPUT SIGNALS

Signal	Symbol	Interface Connection	Module Terminal	Module Type
B Run (1)	—◆	PF2S	PE8J	R650
Data Field 0 (1)	—◇	ME30K	ME7L	S107
Data Field 1 (1)	—◇	ME30M	ME7N	S107
Data Field 2 (1)	—◇	ME30P	ME7R	S107
BT1	—▶	MF34S	MD30H	W640
BT2A	—▶	MF34T	MD30U	W640
B Power Clear	—▶	MF34V	MD30N	W640

Miscellaneous Interface Signals

The following input and output signal connections are available for use with DEC equipment options or for use in special interface equipment designed by the customer.

ADDRESS EXTENSION INPUTS AND DATA FIELD OUTPUTS

When the Memory Extension Control Type 183 is in the computer system, devices using the data break facility must supply a 12-bit data address and a 3-bit address extension. Conversely, the programmed transfer of an address to a register in a device that uses the data break occurs as a 12-bit word from the accumulator and a 3-bit data field extension from the 183.

The Address Extension 1-3 signals must be ground potential to designate a binary 1 and −3v to designate a binary 0. Each of these signals supplies an input to both an inverter of a Type S107 module and a Type S151 Binary-to-Octal Decoder module. Each signal at ground potential is loaded by 2 ma and each signal at −3v receives no load.

The Data Field 0-2 signals are constantly available at the interface connectors. They are flip-flop output signals buffered by an inverter of a Type S107 module. Each signal can drive 15 ma at ground potential, specifying a binary 1.

ANALOG INPUT SIGNAL

The Analog-to-Digital Converter Type 189 option receives an analog input signal between 0 and −10v. A BNC connector mounted on the outside of the processor fan housing at the back of the computer provides connection for this signal. Internal wiring cables this connector to the input of a Type A502 Comparator module. This module compares the analog input signal with a 0 to −10v analog signal produced in Type A601 and A604 Digital-Analog Converter modules. The input draws up to 1 μa depending on the relative polarity of the two voltage inputs of the A502 module. The maximum current difference between positive and negative input voltages is 1 μa. The difference input capacitance is 75 pf.

B RUN OUTPUT SIGNAL

The binary 1 output of the RUN flip-flop flows to external equipment through the interface circuits. This signal is at −3v when the computer is performing instructions and is at ground potential when the program halts. Magnetic tape and DECtape equipment use this signal to stop transport motion when the PDP-8 halts, preventing the tape from running off the end of the reel. The B Run signal is routed to the interface connector through a Type R650 Bus Driver module which can drive a 20-ma load.

BT1 AND BT2A OUTPUT PULSES

Two buffered timing pulse signals, designated BT1 and BT2A, are supplied to I/O devices. These signals can synchronize operations in external equipment with those in the computer. The BT1 and BT2A pulse signals are derived from the T1 and T2A pulse signals generated by the timing signal generator of the PDP-8. The Type W640 Pulse Amplifier module standardizes the T1 and T2A pulses as negative 400-nsec pulses. The resulting (buffered) BT1 and BT2A pulses are supplied to the interface connections. Interface cable connections for each of the pulse outputs can drive a 10-ma load.

B POWER CLEAR OUTPUT PULSES

The Power Clear pulses generated and used within the PDP-8 are made available at the interface connections. External equipment uses these pulses to clear registers and control logic during the power turn-on period. Use of Power Clear pulses in this manner is valid only when the logic circuits cleared by the pulses are energized before or at the same time the PDP-8 POWER switch is turned on.

Loading and Driving Considerations

All PDP-8 circuits providing output or receiving input interface signals are R-, S, or W-series FLIP CHIP modules. Therefore the PDP-8 interface is defined entirely in terms of current driving or draining characteristics.

All R- and S-series modules are capable of driving currents in the direction from ground to −15v, assuming Ben Franklin's definition of current flow. In no cases are R- or S-series modules designed to drive loads which are essentially base loads. If such loads are to be driven, extra clamped load resistors must be added to make up the necessary differential in current. Therefore, R- and S-series loads are defined in terms of milliamperes at ground and 0 ma at −3v. In general, the output of any R- or S-series inverter, including the flip-flop, can drive 20 ma. However, a 2-ma load in R or 5-ma load in S modules is included within most flip-flops and this current must be deducted from the available driving capability.

Inputs are also defined in terms of milliamperes. Level inputs to level gates are defined as 1 ma at ground. Level inputs to diode-capacitor-diode (DCD) gates are 2 ma at ground, and pulse inputs to DCD gates are 3 ma at ground. Since capacitive loading presents problems with R-series modules, where long lines are being driven, the user should add extra clamped loads to sufficiently discharge cable capacitance. Approximately an extra 2 ma of clamped load current should be added for every foot of wire beyond 1-1/2 ft. Inputs to the Type R650 Bus Driver module are exempted from this rule since this module is designed to drive coaxial cable of 93-ohms characteristic impedance.

The Type R650 Bus Driver module has two types of outputs, the fast and the slow (or ramp) output. Using the fast output, the bus driver operates as a fast amplifier. In using the ramp output, an integrating capacitor between input of the bus driver and the output stage, causes the output lines to move from ground to −3v or in the reverse direction in approximately 500 nsec. This connection on the AC lines reduces cross-talk between lines. All other R650 module outputs are fast.

The Type W640 Pulse Amplifier modules should be carefully terminated. If sufficient noise is generated at the output of the W640, it may cause the pulse amplifier to regenerate; hence it is also recommended that output lines of W640 modules be well shielded. The outputs of W640 modules may be either 400 nsec or 1 μsec in width. All connections on the standard PDP-8 use the 400-nsec pulse width.

Input signals to the PDP-8 are, in many cases, a clamped load resistor of 10 ma and a direct input to a flip-flop or pulse amplifier. The input load is, therefore, 10 ma for the clamped load and 1 ma for the flip-flop or the pulse amplifier. Capacitive loads must be at −3v before the pulse amplifier or flip-flop is used for the next machine cycle. There are some exceptions to this statement. First, the Data Bit inputs to the MB require 2 ma at ground and no current at −3v. The Transfer Direction signal requires 1 ma at ground. The Break Request signal input has a 10-ma clamped load plus 2 ma for the internal circuitry or 12 ma at ground. The output of all DEC inverters in the system module series drives 15 ma. In general, the clamped load which is normally used absorbs 10 ma. However, on the input signal interface, no clamped load is used; hence, the above numbers may be used.

Timing is, in general, determined by the machine itself. However, a few statements can be made about module timing. The Type S111 Diode Gates set up in approximately 50 nsec in either direction under normal load conditions. Fall times are faster with heavier loads. The diode-capacitor-diode gates set up in 400 nsec. This 400 nsec is determined from the end of the preceding 100-nsec pulse, and both the level and pulse must return to −3v for 400 nsec before the next pulse arrives. Pulses originating in R- or S-series modules are 100 nsec in width, measured from the 10% point of the leading edge to the 90% point of the trailing edge. Fall time is not critical on these pulses provided the pulse has returned to −3v in time to come up for the next pulse.

The following definitions and rules serve as a useful guide in determining the driving capability of output signals and the load presented to input signals by B-series FLIP CHIP modules or DEC System Modules used in peripheral equipment connected to the PDP-8.

BASE LOAD

Base load is the current which must be drawn from the base of a dc inverter to keep it saturated. In this condition the inverter circuit input terminal is at −3v, the emitter is at ground, and a nominal 1 ma of current flows through the 3000-ohm base resistor from ground. A 1500-ohm load resistor clamped at −3v can nominally accept 8 ma, but tolerance considerations limit this number to 7 ma. Thus, an inverter collector with a 1500-ohm clamped load can drive a maximum of seven base loads.

PULSE LOAD

Pulse load is the load presented to the output of a pulse source by an inverter base in the same speed series, or by the direct set or clear input of a flip-flop. Pulse amplifiers are usually limited to driving 16 pulse loads. This number should be decreased if the bases are widely separated physically, and can be increased to 18 if the bases are physically close together. The series inductance and shunt capacity of connecting wires make pulses at the end of a series of bases either large or small. Consequently, when driving nearly the maximum number of bases, the pulse amplitude should be carefully checked after installation. A terminating resistor in the 100- to 300-ohm range is desirable to reduce ringing on a heavily loaded pulse line. The loading on a pulse source is approximately the same when driving a base as a direct input to a flip-flop. One pulse source, of course, cannot drive both direct input of flip-flops and inverter bases because the direct inputs require DEC standard positive pulses, and base inputs require DEC standard negative pulses. A pulse load is largely determined by the value of the speed-up capacitor connected in parallel with the 3000-ohm base resistor. In the 4000-series 500-kc modules this capacitor is 680 pf; in 1000-series 5-mc modules it is 82 pf; and in 6000-series 10-kc modules it is 56 pf.

PULSED EMITTER LOAD

Pulsed emitter load is the load applied to the collector of an inverter which drives the pulse input to a flip-flop, pulse amplifier, or delay. The pulse current passes through all of the inverters in series with the pulse input, and it should be assumed to be the load on each of the series inverters.

DC EMITTER LOAD

The load applied to the collector of an inverter driving a clamped load resistor is the dc emitter load. This load is also presented by the collector of an inverter which drives an emitter in an inverter network terminated by a clamped load resistor. Under these conditions, the collector of an inverter driving an emitter in a transistor gating network must also supply the base current leaving the succeeding inverters which are saturated. This current is small, but in complex networks it must be considered. An inverter in the DEC 1000- or 6000-series modules can supply 15 ma, and in the 4000-series modules can supply 20 ma.

An inverter network can always be analyzed by assuming:

1. A short circuit exists between the emitter and collector when −3v is applied to the base.

2. Base current of 1 ma will flow if either the collector or emitter is held at ground potential.

3. The maximum dc collector current through an inverter is 20 ma for 4000-series 500-kc modules and is 15 ma for all other DEC series modules.

A capacitor-diode gate level input does not present any dc load. A transient load occurs when the input level changes. Note that all capacitor-diode gates require that the level input precede the initiating pulse input by at least 1 μsec.

CHAPTER 6

INSTALLATION PLANNING

Space Requirements

Space must be provided at the installation site to accommodate the PDP-8 and peripheral equipment and to allow access to all doors and panels for maintenance.

Installation dimensions for a table-mounted and a rack-mounted PDP-8 are shown in Figures 57 and 58, respectively. Dimensions of a rack-mounted PDP-8 in an optional DEC computer cabinet and of a table-mounted PDP-8 on an optional DEC winged table are shown in Figure 59. Floor space for a basic optional computer cabinet is 22-1/4 inches wide (with two end panels) and 27-1/16 inches deep, plus additional space for a table. Figure 59 can be used in planning the installation of all I/O equipment mounted in standard computer cabinets noting that other cabinets may be equipped with a table, and that cabinets bolted together are 19-3/4 inches wide with 1-1/4 inch end panels mounted on the outer ends. Minimum service clearance on all standard DEC computer cabinets is 8-3/4 inches at the front and 14-7/8 inches at the back. A standard DEC computer cabinet contains space for one mounting panel (two rows) of FLIP CHIP modules or an indicator panel above the computer, and for three mounting panels below the operator console. The memory frame and the processor frame are hinged to provide access to the wiring side of the mounting panels. Both of these frames extend beyond the back of the power supply in the table model to allow entrance of interface cables. Cables enter the cabinet model through a port in the bottom of standard cabinets. Wheels and leveling devices on the cabinets allow cable clearance so that subflooring is not required.

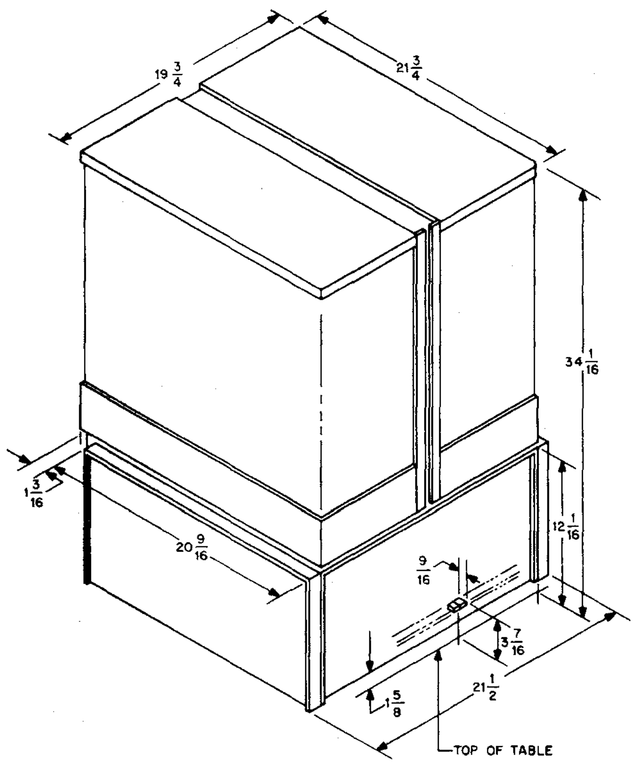


Figure 57 Table Mounted PDP-8 Installation Dimensions

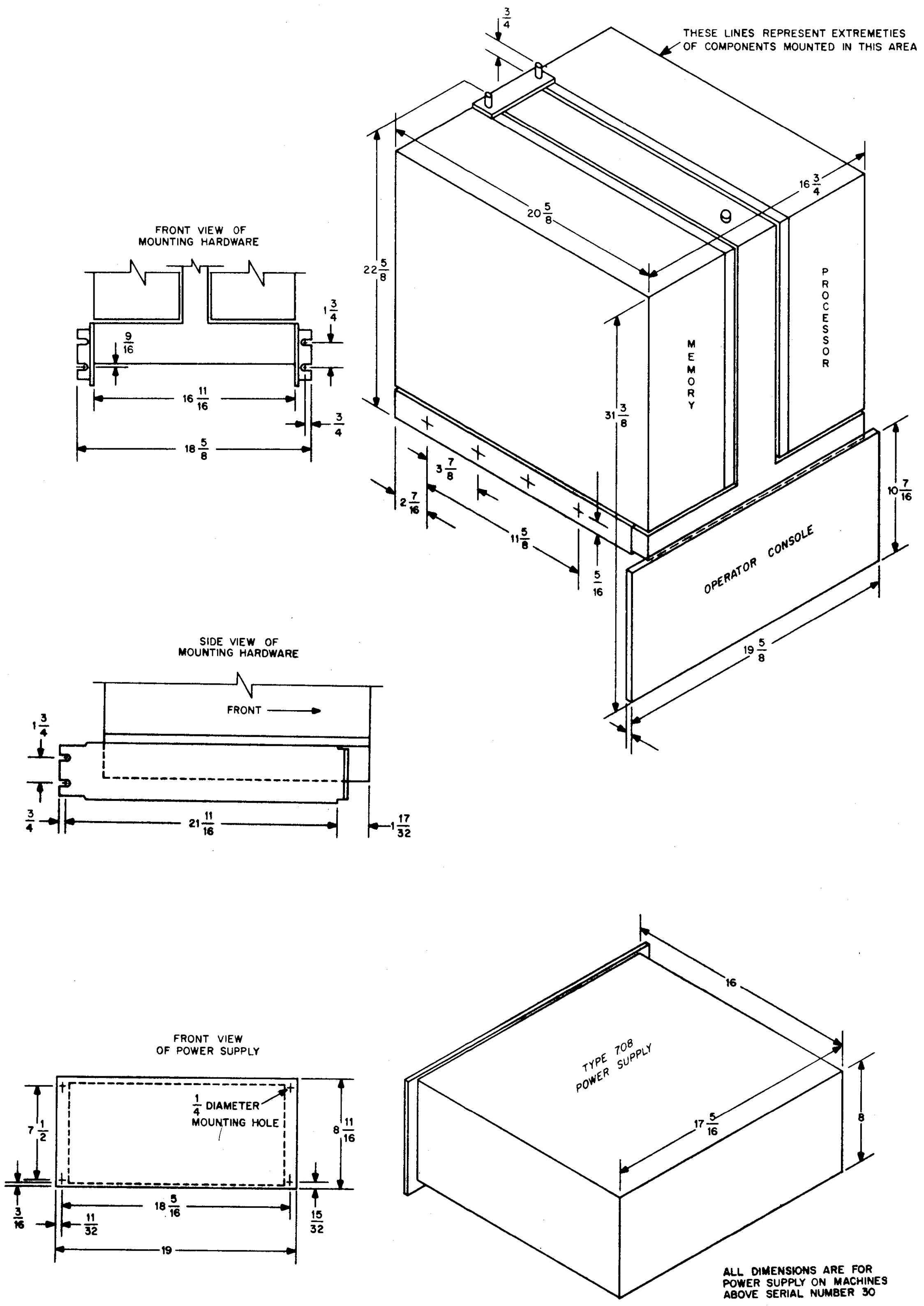

THESE LINES REPRESENT EXTREMETIES OF COMPONENTS MOUNTED IN THIS AREA
3/4
20 5/8
16 3/4
22 5/8
31 3/8
PROCESSOR
MEMORY
OPERATOR CONSOLE
10 7/16
19 5/8
3 7/8
2 7/16
11 5/8
5/16
FRONT VIEW OF MOUNTING HARDWARE
1 3/4
9/16
16 11/16
3/4
18 5/8
SIDE VIEW OF MOUNTING HARDWARE
FRONT
1 3/4
3/4
21 11/16
1 17/32
FRONT VIEW OF POWER SUPPLY
1/4 DIAMETER MOUNTING HOLE
7 1/2
8 11/16
3/16
18 5/16
11/32
15/32
19
TYPE 708 POWER SUPPLY
16
17 5/16
8
ALL DIMENSIONS ARE FOR POWER SUPPLY ON MACHINES ABOVE SERIAL NUMBER 30

Figure 58 Cabinet Mounted PDP-8 Installation Dimensions

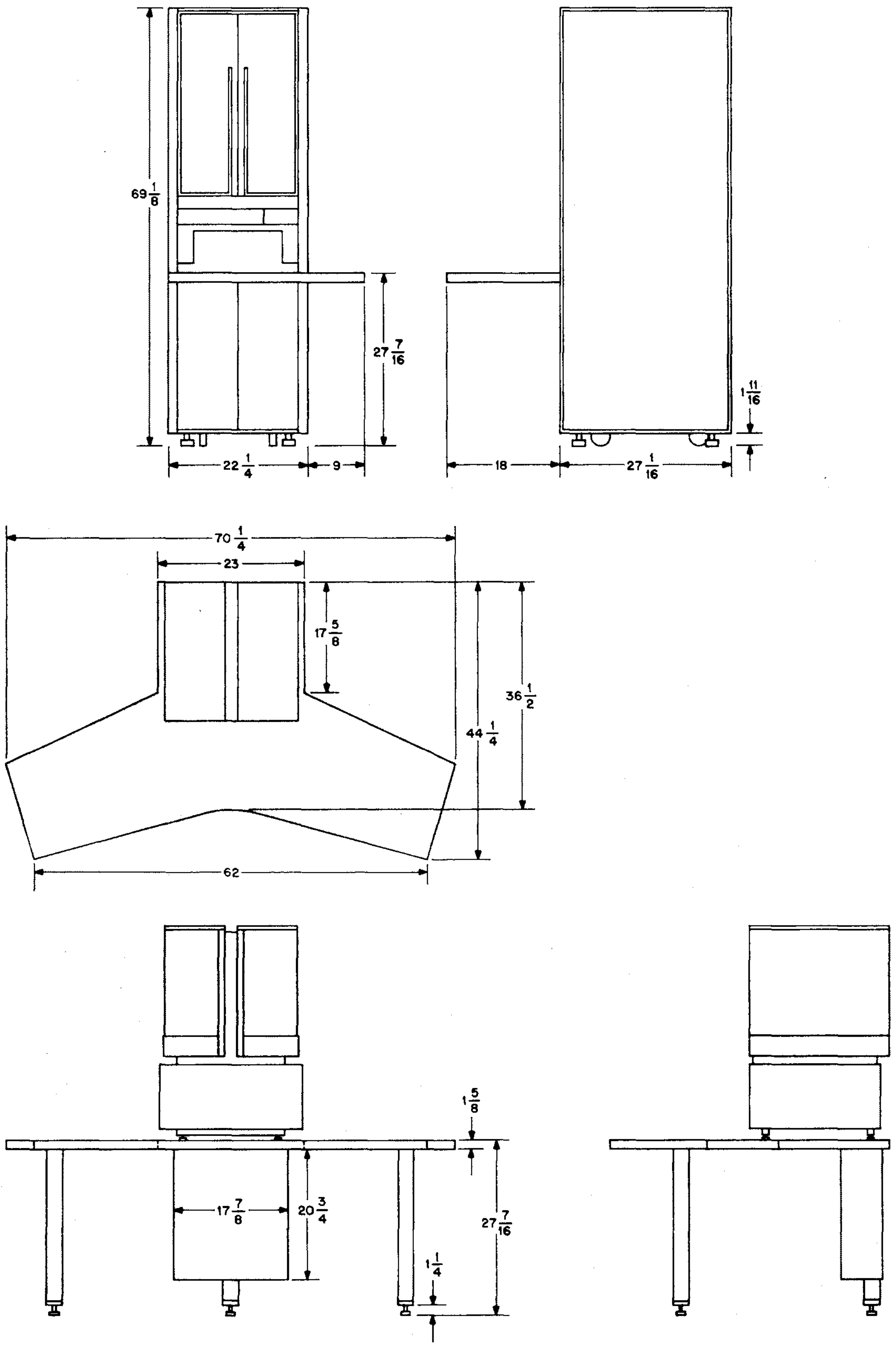


Figure 59 Optional Cabinet and Table Installation Dimensions

The standard Teletype automatic send receive set requires floor space approximately 22-1/4 inches wide by 18-1/2 inches deep. Signal cable length restricts the location of the Teletype to within 18 inches of the side of the computer.

Environmental Requirements

Ambient temperature at the installation site can vary between 32 and 130 F (between 0 and 55 C) with no adverse effect on computer operation. However, to extend the life expectancy of the system, it is recommended that the ambient temperature at the installation site be maintained between 70 and 85 F (between 21 and 30 C).

During shipping or storing of the system, the ambient temperature may vary between 32 and 130 F (between 0 and 55 C). Although all exposed surfaces of all DEC cabinets and hardware are treated to prevent corrosion, exposure of systems to extreme humidity for long periods of time should be avoided.

Power Requirements

A source of 115v (±17v), 60-cps (±0.5 cps), single-phase power capable of supplying at least 15 amp must be provided to operate a standard PDP-8. To allow connection to the power cable of the computer, this source should be provided with a Hubbell 3-terminal, except for the basic table top PDP-8 grounded-neutral flush receptacle (or its equivalent). A table-mounted PDP-8 is provided with a 15-amp power plug; a rack-mounted PDP-8 has a 20 amp twist-lock plug; and systems that draw more than 20 amps use a 30-amp twist-lock plug.

Power dissipation of a standard PDP-8 is approximately 780w, and the heat dissipation is approximately 2370 Btu/hr. Upon special request, a PDP-8 can be constructed to operate from a 220v (±33v), 60-cps (±0.5 cps), single-phase power source or from a 100v (±15v), 50-cps (±0.5 cps), single-phase power source.

Cable Requirements

Nine-conductor coaxial cables with Type W011 Cable Connectors provide signal connection between the computer and optional equipment in free-standing cabinets. These cables are connected by plugging the W011 connectors into standard FLIP CHIP module receptacles.

All free standing cabinets require independent 115v receptacles. However, these units may be turned on or off or controlled from the PDP-8 operator console.

Cables connect to cabinets through a drop panel in the bottom of cabinets. Subflooring is not necessary because casters elevate the cabinets from the floor to afford sufficient cable clearance.

Installation Procedure

During system check-out, customers are invited to visit the Maynard manufacturing facility to inspect and become familiar with their equipment. Computer customers may also send personnel to instruction courses on computer operation, programming, and maintenance conducted regularly in Maynard, Massachusetts.

DEC's engineers are available during installation and test for assistance or consultation. Further technical assistance in the field is provided by home office design engineers or branch office application engineers.

Table 11 gices installation data to be considered when installing a PDP-8. Table 12 lists space requirements. Figure 60, a typical PDP-8 system configuration, can be used as a guide for planning layout.

TABLE 11 INSTALLATION DATA

	Weight (lbs)	Dimensions			Service Clearance		Heat Dissipation Btu/hr	Current (amps)		Power Dissipation (kw)
		Height	Width	Depth	Front	Rear		Nom	Surge	
PDP-8 Table Top	250	32	21-1/2	21-1/4	-	-	2660	7.5	-	0.78
PDP-8 [3] Rack Mount	250	31-1/4	[1] 19-5/8	[2] 21-7/8	22	-	2660	7.5	-	0.78
Standard Cabinet CAB8B (Empty)	225	69-1/8	22-1/4	27-1/16	22	15	-	-	-	-
Teletype ASR-33	40	45	23	19	-	-	(Included in Standard PDP-8)			
Serial Drum 251	500	70	23	28	9	15	1540	5	8	0.45
Card Reader CR01C	25	8-1/4	18	10	-	-	270	0.57	1	0.06
Card Reader 451A	225	42	30	18	-	7	450	1.3	7	0.2
Magnetic Tape Transport 50	600	69-1/8	22-1/4	27-1/16	18-5/8	15	2114	8	12	0.96
Magnetic Tape Transport 570	850	68	32-1/8	32-3/8	19	17	9900	25	40	2.9
Magnetic Tape Transport 545	400	69-1/8	22-1/4	27-1/16	9	15	2870	7.3	8	0.9
Magnetic Tape System 580	400	69-1/8	22-1/4	27-1/16	9	15	2870	7.3	8	0.9
DECtape Transport TU55	35	10-1/2	19-1/2	9-3/4	9	-	410	1	2	0.11
Precision Display 30N	350	49	53	39	-	15	3140	8	8	0.9

NOTES:

[1] 19 inch console panel available on request

[2] Overall depth is 24-3/8 inches from front of console to back of chassis track slices.

[3] When PDP-8 is mounted in CAB8 (A or B), there is additional room for:

a. One standard (5-1/4 inch high) DEC logic mounting panel above the computer in front.

b. Three standard DEC logic mounting panels below the comp·ter table level in front.

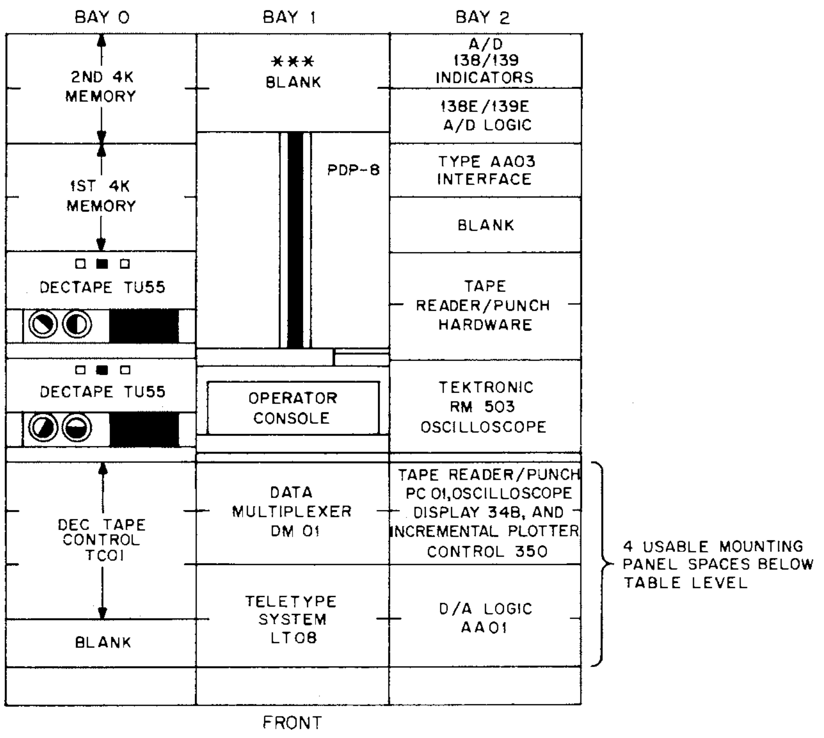
c. On the rear plenum door, space is available for 1 short mounting panel at the top and three short paneis below table level.

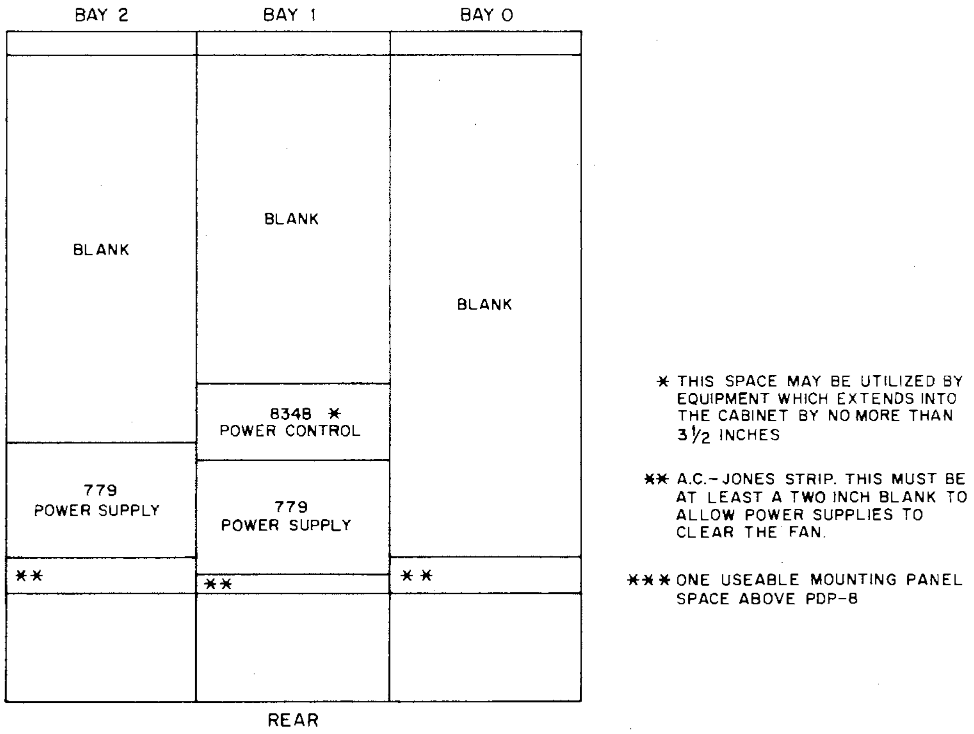
TABLE 12 SPACE REQUIREMENTS

Option	MOUNTING PANELS*		Remarks
	Logic	Other	
Memory Extension Control 183	-	-	Mounts within standard PDP-8 package
Memory Module 184	2	-	Should be mounted in expander cabinet next to PDP-8
Automatic Multiply-Divide 182	-	-	Mounts within standard PDP-8 package
Memory Parity 188	-	-	Mounts within standard PDP-8 package
Automatic Restart KR01	-	-	Mounts within standard PDP-8 package
Data Channel Multiplexer DM01	-	2	
Perforated Tape Reader 750C	2	10 in. front panel	These 5 options share same two mounting panels for control logic.
Perforated Tape Punch 75E	2	15-3/4 in. vertical clearance in cabinet	
Oscilloscope Display 34 D	2	10 in. front panel	
Incremental Plotter 350B	2	Table space needed for plotter	
Card Reader CR01C	2	Table space needed for reader	
A/D Converter 189	-	-	Mounts within standard PDP-8 package
A/D Converter and Multiplexer 138E/139E	2	5-1/4 in. for indicator panel	No additional space needed for 64 channel multiplexer expansion
Light Pen 370	-	-	Logic and power supply included in 34D or 30N
DCS 680 681 685 682	- - 2 2	- - - -	Mounted within standard PDP-8 package for up to 64 full duplex channels connectors for up to 64 Teletypes
Teletype System LT08	2	-	Handles up to 5 Teletypes
Magnetic Tape Control 57A	7	5-1/4 in. connector panel	Controls up to 8 IBM-compatible tape units (570, 50, or 545)
DECtape Control TC01	3	-	Controls up to 8 TU55 DECtape Transports

*Mounting Panels are standard DEC 5-1/4 in. high logic panels.

NOTE: Power supplies for option logic are normally mounted on rear door of DEC cabinets. Customers using their own cabinets should allow additional space for power supplies.





* THIS SPACE MAY BE UTILIZED BY EQUIPMENT WHICH EXTENDS INTO THE CABINET BY NO MORE THAN 3 1/2 INCHES

** A.C.-JONES STRIP. THIS MUST BE AT LEAST A TWO INCH BLANK TO ALLOW POWER SUPPLIES TO CLEAR THE FAN.

*** ONE USEABLE MOUNTING PANEL SPACE ABOVE PDP-8

Figure 60 Typical PDP-8 System Configuration and Layoff Planning

APPENDIX 1 INSTRUCTIONS

MEMORY REFERENCE INSTRUCTIONS

Mnemonic Symbol	Operation Code	Direct Addr.		Indirect Addr.		Operation
		States Entered	Execution Time (μsec)	States Entered	Execution Time (μsec)	
AND Y	0	F, E	3.0	F, D, E	4.5	Logical AND. The AND operation is performed between the content of memory location Y and the content of the AC. The result is left in the AC, the original content of the AC is lost, and the content of Y is restored. Corresponding bits of the AC and Y are operated upon independently. $ACj \wedge Yj => ACj$
TAD Y	1	F, E	3.0	F, D, E	4.5	Two's complement add. The content of memory location Y is added to the content of the AC in two's complement arithmetic. The result of this addition is held in the AC, the original content of the AC is lost, and the content of Y is restored. If there is a carry from AC0, the link is complemented. $AC + Y => AC$
ISZ Y	2	F, E	3.0	F, D, E	4.5	Increment and skip if zero. The content of memory location Y is incremented by one. If the resultant content of Y equals zero, the content of the PC is incremented and the next instruction is skipped. If the resultant content of Y does not equal zero, the program proceeds to the next instruction. The incremented content of Y is restored to memory. If resultant $Y = 0$, $PC + 1 => PC$

MEMORY REFERENCE INSTRUCTIONS (continued)

Mnemonic Symbol	Operation Code	Direct Addr. States Entered	Direct Addr. Execution Time (μsec)	Indirect Addr. States Entered	Indirect Addr. Execution Time (μsec)	Operation
DCA Y	3	F, E	3.0	F, D, E	4.5	Deposit and clear AC. The content of the AC is deposited in core memory at address Y and the AC is cleared. The previous content of memory location Y is lost. AC => Y 0 => AC
JMS Y	4	F, E	3.0	F, D, E	4.5	Jump to subroutine. The content of the PC is deposited in core memory location Y and the next instruction is taken from core memory location Y + 1. PC + 1 => Y Y + 1 => PC
JMP Y	5	F	1.5	F, D	3.0	Jump to Y. Address Y is set into the PC so that the next instruction is taken from core memory address Y. The original content of the PC is lost. Y = > PC

BASIC IOT MICROINSTRUCTIONS

Mnemonic	Octal	Operation
		PROGRAM INTERRUPT
ION	6001	Turn interrupt on and enable the computer to respond to an interrupt request. When this instruction is given, the computer executes the next instruction, then enables the interrupt. The additional instruction allows exit from the interrupt subroutine before allowing another interrupt to occur.
IOF	6002	Turn interrupt off i.e. disable the interrupt.
		ANALOG-TO-DIGITAL CONVERTER TYPE 189
ADC	6004	Convert the analog input signal to a digital value.
		HIGH SPEED PERFORATED TAPE READER AND CONTROL TYPE 750C
RSF	6011	Skip if reader flag is a 1.
RRB	6012	Read the content of the reader buffer and clear the reader flag. (This instruction does not clear the AC.) RB V AC 4-11 => AC 4-11
RFC	6014	Clear reader flag and reader buffer, fetch one character from tape and load it into the reader buffer, and set the reader flag when done.
		HIGH SPEED PERFORATED TAPE PUNCH AND CONTROL TYPE 75E
PSF	6021	Skip if punch flag is a 1.
PCF	6022	Clear punch flag and punch buffer.
PPC	6024	Load the punch buffer from bits 4 through 11 of the AC and punch the character. This instruction does not clear the punch flag or punch buffer.) AC 4-11 V PB => PB
PLS	6026	Clear the punch flag, clear the punch buffer, load the punch buffer from the content of bits 4 through 11 of the accumulator, punch the character, and set the punch flag to 1 when done.
		TELETYPE KEYBOARD/READER
KSF	6031	Skip if keyboard flag is a 1.
KCC	6032	Clear AC and clear keyboard flag.
KRS	6034	Read keyboard buffer static. (This is a static comman in that neither the AC nor the keyboard flag is cleared.) TTI V AC 4-11 => AC 4 -11
KRB	6036	Clear AC, clear keyboard flag, and read the content of the keyboard buffer into the content of AC 4-11.

BASIC IOT MICROINSTRUCTIONS (continued)

Mnemonic	Octal	Operation
		TELETYPE TELEPRINTER/PUNCH
TSF	6041	Skip if teleprinter flag is a 1.
TCF	6042	Clear teleprinter flag.
TPC	6044	Load the TTO from the content of AC 4-11 and print and/or punch the character.
TLS	6046	Load the TTO from the content of AC 4-11, clear the teleprinter flag, and print and /or punch the character.
		OSCILLOSCOPE DISPLAY TYPE 34D AND PRECISION CRT DISPLAY TYPE 30N
DCX	6051	Clear X coordinate buffer.
DXL	6053	Clear and load X coordinate buffer. AC 2-11 => XB
DIX	6054	Intensify the point defined by the content of the X and Y coordinate buffers.
DXS	6057	Executes the combined functions of DXL followed by DIX.
DCY	6061	Clear Y coordinate buffer.
DYL	6063	Clear and load Y coordinate buffer. AC 2-11 => YB
DIY	6064	Intensify the point defined by the content of the X and Y coordinate buffers.
DYS	6067	Executes the combined functions of DYL followed by DIY.
		OSCILLOSCOPE DISPLAY TYPE 34D
DSB	6075	Set minimum brightness.
DSB	6076	Set medium brightness.
DSB	6077	Set maximum brightness.
		PRECISION CRT DISPLAY TYPE 30N
DLB	6074	Load brightness register (BR) from bits 9 through 11 of the AC. AC 9-11 => BR
		LIGHT PEN TYPE 370
DSF	6071	Skip if display flag is a 1.
DCF	6072	Clear the display flag.
		MEMORY PARITY TYPE 188
SMP	6101	Skip if memory parity error flag = 0.
CMP	6104	Clear memory parity error flag.
		AUTOMATIC RESTART TYPE KR01
SPL	6102	Skip if power is low.

BASIC IOT MICROINSTRUCTIONS (continued)

Mnemonic	Octal	Operation
		MEMORY EXTENSION CONTROL TYPE 183
CDF	62N1	Change to data field N. The data field register is loaded with the selected field number (0 to 7). All subsequent memory requests for operands are automatically switched to that data field until the data field number is changed by a new CDF command.
CIF	62N2	Prepare to change to instruction field N. The instruction buffer register is loaded with the selected field number (0 to 7). The next JMP or JMS instruction causes the new field to be entered.
RDF	6214	Read data field into AC 6-8. Bits 0-5 and 9-11 of the AC are not affected.
RIF	6224	Same as RDF except reads the instruction field.
RIB	6234	Read interrupt buffer. The instruction field and data field stored during an interrupt are read into AC 6-8 and 9-11 respectively.
RMF	6244	Restore memory field. Used to exit from a program interrupt.
		DATA COMMUNICATION SYSTEMS TYPE 630
TTINCR	6401	The content of the line select register is incremented by one.
TTI	6402	The line status word is read and sampled. If the line is active for the fourth time, the line bit is shifted into the character assembly word. If the line is active for a number of times less than four, the count is incremented. If the line is not active, the active/inactive status of the line is recorded.
TTO	6404	The character in the AC is shifted right one position, zeros are shifted into vacated positions, and the original content of AC11 is transferred out of the computer on the Teletype line.
TTCL	6411	The line select register is cleared.
TTSL	6412	The line select register is loaded by an OR transfer from the content of AC5-11, then the AC is cleared.
TTRL	6414	The content of the line select register is read into AC5-11 by an OR transfer.
TTSKP	6421	Skip if clock 1 flag is a 1.
TTXON	6422	Clock 1 is enabled to request a program interrupt and clock 1 flag is cleared.
TTXOF	6424	Clock 1 is disabled from causing a program interrupt and clock 1 flag is cleared.

BASIC IOT MICROINSTRUCTIONS (continued)

Mnemonic	Octal	Operation
		INCREMENTAL PLOTTER AND CONTROL TYPE 350B
PLSF	6501	Skip if plotter flag is a 1.
PLCF	6502	Clear plotter flag.
PLPU	6504	Plotter pen up. Raise pen off of paper.
PLPR	6511	Plotter pen right.
PLDU	6512	Plotter drum (paper) upward.
PLDD	6514	Plotter drum (paper) downward.
PLPL	6521	Plotter pen left.
PLUD	6522	Plotter drum (paper) upward. (Same as 6512.)
PLPD	6524	Plotter pen down. Lower pen on to paper.
		GENERAL PURPOSE CONVERTER TYPE 138E AND MULTIPLEXER CONTROL TYPE 139E
ADSF	6531	Skip if A/D converter flag is a 1.
ADCV	6532	Clear A/D converter flag and convert input voltage to a digital number, flag will set to 1 at end of conversion. Number of bits in converted number determined by switch setting, 11 bits maximum.
ADRB	6534	Read A/D converter buffer into AC, left justified, and clear flag.
ADCC	6541	Clear multiplexer channel address register.
ADSC	6542	Set up multiplexer channel as per AC 6-11. Maximum of 64 single ended or 32 differential input channels.
ADIC	6544	Index multiplexer channel address (present address + 1). Upon reaching address limit, increment will cause channel 00 to be selected.
		SERIAL MAGNETIC DRUM SYSTEM TYPE 251
DRCR	6603	Load the drum core location counter with the core memory location information in the accumulator. Prepare to read one sector of information from the drum into the specified core location. Then clear the AC.
DRCW	6605	Load the drum core location counter with the core memory location information in the accumulator. Prepare to write one sector of information into the drum from the specified core location. Then clear the AC.
DRCF	6611	Clear completion flag and error flag.
DREF	6612	Clear the AC then load the condition of the parity error and data timing error flip-flops of the drum control into accumulator bits 0 and 1 respectively to allow programmed evaluation of an error flag.

Mnemonic	Octal	Operation
		SERIAL MAGNETIC DRUM SYSTEM TYPE 251 (continued)
DRTS	6615	Load the drum address register with the track and sector address held in the accumulator. Clear the completion and error flags, and begin a transfer (reading or writing). Then clear the AC.
DRSE	6621	Skip next instruction if the error flag is a 0 (no error).
DRSC	6622	Skip next instruction if the completion flag is a 1 (sector transfer is complete).
DRCN	6624	Clear error flag and completion flag, then initiate transfer of next sector.
		CARD READER AND CONTROL TYPE CR01C
RCSF	6631	Skip if card reader data ready flag is a 1.
RCRA	6632	The alphanumeric code for the column is read into AC6-11, and the data ready flag is cleared.
RCRB	6634	The binary data in a card column is transferred into AC0-11, and the data ready flag is cleared.
RCSP	6671	Skip if card reader card done flag is a 1.
RCSE	6672	Clear the card done flag, select the card reader and start card motion towards the read station, and skip if the reader-not-ready flag is a 1.
RCRD	6674	Clear card done flag.
		CARD READER AND CONTROL TYPE 451
CRSF	6632	Skip if card reader flag is a 1. If a card column is present for reading, the next instruction is skipped.
CERS	6634	Card equipment read status. Reads the status of the card reader flag and status levels into bits 6 through 9 of the AC. The AC bit assignments are: AC6 = Flag is set to 1 (the flag rises after reading each of the 80 rows). AC7 = Card done. AC8 = Not ready (covers not in place, power is off, START button has not been pressed, hopper is empty, stacker is full, a card is jammed, a validity check error has been detected, or the read circuit is defective). AC9 = End of file (EOF) (hopper is empty and operator has pushed START button).
CRRB	6671	Read the card column buffer information into the AC and clear the card reader flag. One CRRB command reads alphanumeric information. Two CRRB instructions read the upper and lower column binary information.

BASIC IOT MICROINSTRUCTIONS (continued)

Mnemonic	Octal	Operation
		CARD READER AND CONTROL TYPE 451 (continued)
CRSA	6672	Select a card in alphanumeric mode. Select the card reader and start a card moving. Information appears in alphanumeric form.
CRSB	6674	Select a card in binary mode. Select the card reader and start a card moving. Information appears in binary form.
		CARD PUNCH AND CONTROL TYPE 450
CPSF	6631	Skip if card punch flag is a 1. The card punch flag indicates the punch buffer is available, and should be loaded.
CERS	6634	Card equipment read status. Reads the status of the card punch flag and the card punch error level into bits 10 and 11 of the AC, respectively.
CPCF	6641	Clear card punch flag.
CPSE	6642	Select the card punch. Transmit a card to the 80-column punch die from the hopper.
CPLB	6644	Load the card punch buffer from the content of the AC. Seven load instructions must be given to fill the buffer.
		AUTOMATIC LINE PRINTER AND CONTROL TYPE 645
LSE	6651	Skip if line printer error flag is a 1.
LCB	6652	Clear both sections of the printing buffer.
LLB	6654	Load printing buffer from the content of AC 6-11 and clear the AC.
LSD	6661	Skip if the printer done flag is a 1.
LCF	6662	Clear line printer done and error flags.
LPR	6664	Clear the format register, load the format register from the content of AC 9-11, print the line contained in the section of the printer buffer loaded last, clear the AC, and advance the paper in accordance with the selected channel of the format tape if the content of AC 8 = 1. If the content of AC 8 = 0, the line is printed and paper advance is inhibited.
		AUTOMATIC MAGNETIC TAPE CONTROL TYPE 57A
MSCR	6701	Skip if the tape control ready (TCR) level is 1. A 1 is added to the content of the program counter if the tape control is free to accept a command.

BASIC IOT MICROINSTRUCTIONS (continued)

Mnemonic	Octal	Operation
		AUTOMATIC MAGNETIC TAPE CONTROL TYPE 57A (continued)
MCD	6702	Clear the job done flag, clear command register, clear word count overflow (WCO) flag, and clear end of record (EOR) flag. This instruction should be immediately preceded by the two instructions CLA and TAD (4000) to obtain the operation indicated. The job done flag is connected to the program interrupt facility.
MTS	6706	Disable the job done flag from the program interrupt, turn on the WCO flag and EOR flag and select the unit, the mode of parity, and the density from the content of the AC. The AC bit assignments are: (Type 521 and 522 interface only) AC1(0) = High sense level AC1(1) = Low sense level AC2(0) = 200 or 556 density AC2(1) = 800 or 556 density AC8(0) = 200 density AC8(1) = 556 density AC2 / AC8 / Density: 0 / 0 / 200 0 / 1 / 556 1 / 0 / 800 1 / 1 / 556 AC7(0) = Even parity (BCD) AC7(1) = Odd parity (binary) AC9-11 These three bits select one of eight tape units, addresses 0-7.
MSUR	6711	Skip if the tape transport is ready (TTR). The selected tape unit is checked, using this command, and must be free before the following MTC command is given.
MNC	6712	Terminate the continuous mode. This instruction clears the AC at completion. It should be immediately preceded by the two instructions CLA and TAD (4000) to obtain the operation indicated.
MTC	6716	Place the content of AC 3-6 in the tape control command register and start tape motion. Bit 6 selects motion mode. AC6(0) = Normal AC6(1) = Continuous

BASIC IOT MICROINSTRUCTIONS (continued)

Mnemonic	Octal	Operation
		AUTOMATIC MAGNETIC TAPE CONTROL TYPE 57A (continued)
MTC	6716 (cont.)	AC3-5 are decoded as follows: 0 = No operation 1 = Rewind 2 = Write 3 = Write end of file (EOF) 4 = Read compare 5 = Read 6 = Space forward 7 = Space backward
MSWF	6721	Skip if the WCO flag is a 1. The WCO flag is connected to the program interrupt.
MDWF	6722	Disable WCO flag.
MCWF	6722	Clear WCO flag. This instruction should be immediately preceded by the two instructions CLA and TAD (2000) to obtain the operation indicated.
MEWF	6722	Enable WCO flag. This instruction should be immediately preceded by the two instructions CLA and TAD (4000) to obtain the operation indicated.
MIWF	6722	Initialize WCO flag. This instruction should be immediately preceded by the two instructions CLA and TAD (6000) to obtain the operation indicated.
MSEF	6731	Skip if the EOR flag is a 1. This flag is connected to the program interrupt.
MDEF	6732	Disable ERF.
MCED	6732	Clear ERF. This instruction should be immediately preceded by the two instructions CLA and TAD (2000) to obtain the operation indicated.
MEEF	6732	Enable ERF. This instruction should be immediately preceded by the two instructions CLA and TAD (4000) to obtain the operation indicated.
MIEF	6732	Initialize ERF, clear and enable. This instruction should be immediately preceded by the two instructions CLA and TAD (6000) to obtain the operation indicated.
MTRS	6734	Read tape status bits into the content of the AC. This instruction should be immediately preceded by a CLA instruction to obtain the operation indicated. The bit assignments are: 0 = Data request late 1 = Tape parity error 2 = Read compare error 3 = End of file flag set 4 = Write lock ring out 5 = Tape at load point

BASIC IOT MICROINSTRUCTIONS (continued)

Mnemonic	Octal	Operation
		AUTOMATIC MAGNETIC TAPE CONTROL TYPE 57A (continued)
MTRS	6734 (cont.)	6 = Tape at end point 7 = Tape near end point (Type 520) 7 = Last operation write (Type 521 and 522 interface) 8 = Tape near load point (Type 520) 8 = Write echo (Type 522 interface) 8 = B control using transporting (Type 521 interface with multiplex transport) 9 = Transport rewinding 10 = Tape miss character 11 = Job done flag interrupt
MCC	6741	Clear CA and WC.
MRWC	6742	Transfer the content of AC into the WC.
MRCA	6744	Transfer the content of the CA into the AC. This instruction should be immediately preceded by a CLA command to obtain the operation indicated.
MCA	6745	Clear CA and WC, and transfer the content of the AC into the CA.
		MAGNETIC TAPE SYSTEM TYPE 580

TIFM	6707	Tape initialize function and motion. Clears all tape control registers, loads the command register from the content of the AC, and initiates motion delay. The bit assignments of the command register are:
		AC1 = Space
		AC3 = Go
		AC4 = Write
		AC5 = Parity mode (0 = even, 1 = odd)
		AC6 = Read
		AC7 = Direction (0 = reverse, 1 = forward)
		AC8 = Density (0 = 200 BPI, 1 = 556 BPI)
		AC10 = Rewind
		AC11 = Real time

TSRD	6715	Tape system read. Clear the AC then load the AC from the content of the data buffer and clear the data flag.
TSWR	6716	Tape system write. Clear the data buffer, then load the data buffer from the content of the AC and clear the data flag.
TSDF	6721	Tape skip on data flag. If the data flag is a 1, the next instruction is skipped.

Mnemonic	Octal	Operation
		MAGNETIC TAPE SYSTEM TYPE 580 (continued)
TSSR	6722	Tape skip on end of record. If the end of record delay is a 0 or if the data flag is a 1, the next instruction is skipped.
TSST	6724	Tape system stop data transfer. This instruction is issued following transmission of the last character in a record. It initiates tape shut down procedures such as writing the longitudinal parity bit, end of record mark, and the 0.75-inch inter-record gap. It also clears the SPACE flip-flop.
TWRT	6731	Tape system write real time. One character is written on tape. This instruction can be used at any frequency and therefore determines the density of information written on tape.
TCPI	6732	Tape clear program interrupt. The program interrupt request flag is sampled and if it is a 1 the next instruction is skipped. This command also clears the program interrupt request flag in the control during a space operation and clears the STOP flip-flop.
TSRS	6734	Tape system read status. The content of the status register is transferred into the AC. The bit assignments are: AC0(1) = Parity error AC1(1) = Motion delay set AC2(1) = Transport is ready AC3(1) = Clock delays set AC4(1) = End of tape AC5(1) = Tape at load point
		DECTAPE DUAL TRANSPORT TYPE 555 AND DECTAPE CONTROL TYPE 552
MMLS	6751	Load unit select register from the content of AC 2-5 and set DECtape (DT) flag when done.
MMLM	6752	Load motion register from the content of AC7-8 and set DT flag when done.
MMLF	6754	Load function register from the content of AC 9-11 then clear the AC. The octal code of these three bits establishes the following DECtape control modes: 0 = Move; 4 = Write data 1 = Search; 5 = Write all bits 2 = Read data; 6 = Write mark and timing 3 = Read all bits
MMMF	6756	Load motion register from AC7-8, load function register from AC9-11, clear the AC, and set the DT flag when done.

Mnemonic	Octal	Operation
		DECTAPE DUAL TRANSPORT TYPE 555 AND DECTAPE CONTROL TYPE 552 (continued)
MMMM	6757	Load the unit select register, motion register, and function register from AC2-11; clear the AC, and set the DT flag when done.
MMSF	6761	Skip if DT flag is a 1.
MMCC	6762	Clear memory address counter (MAC).
MMLC	6764	Load MAC from the content of AC0-11 and then clear the AC.
MMML	6766	Clear MAC, load MAC from AC0-11, and clear AC.
MMSC	6771	Skip if error flag is a 1.
MMCF	6772	Clear error flag and DT flag.
MMRS	6774	Read status bits into the content of AC 0-7. The bit assignments are: AC0 = DT flag; AC1 = Error flag; AC2 = End (selected tape at end point); AC3 = Timing error; AC4 = Reverse tape direction; AC5 = Go; AC6 = Parity or mark track error; AC7 = Select error
		DECTAPE TRANSPORT TYPE TU55 AND DECTAPE CONTROL TYPE TC01
DTRA	6761	The content of status register A is read into AC0-9 by an OR transfer. The bit assignments are: AC0-2 = Transport unit select number; AC3-4 = Motion; AC5 = Mode; AC6-8 = Function; AC9 = Enable/disable DECtape control flag
DTCA	6762	Clear status register A. All flags undisturbed.
DTXA	6764	Status register A is loaded by an exclusive OR transfer from the content of the AC, and AC10 and AC11 are sampled. If AC10 = 0, the error flags are cleared. If AC11 = 0, the DECtape control flag is cleared.
DTSF	6771	Skip if error flag is a 1 or if DECtape control flag is a 1.

BASIC IOT MICROINSTRUCTIONS (continued)

Mnemonic	Octal	Operation
		DECTAPE TRANSPORT TYPE TU55 AND DECTAPE CONTROL TYPE TC01 (continued)

DTRB	6772	The content of status register B is read into the AC by an OR transfer. The bit assignments are:
		AC0 = Error flag
		AC1 = Mark track error
		AC2 = End of tape
		AC3 = Select error
		AC4 = Parity error
		AC5 = Timing error
		AC6-8 = Memory field
		AC9-10 = Unused
		AC11 = DECtape flag

DTLB	6774	The memory field portion of status register B is loaded by an OR transfer from the content of AC6-8.

GROUP 1 OPERATE MICROINSTRUCTIONS

Mnemonic Symbol	Octal Code	Event Time	Operation
NOP	7000	—	No operation. Causes a 1.5 μsec program delay.
IAC	7001	2	Increment AC. The content of the AC is incremented by one in two's complement arithmetic.
RAL	7004	2	Rotate AC and L left. The content of the AC and the L are rotated left one place.
RTL	7006	2	Rotate two places to the left. Equivalent to two successive RAL operations.
RAR	7010	2	Rotate AC and L right. The content of the AC and L are rotated right one place.
RTR	7012	2	Rotate two places to the right. Equivalent to two successive RAR operations.
CML	7020	1	Complement L.
CMA	7040	1	Complement AC. The content of the AC is set to the one's complement of its current content.
CIA	7041	1, 2	Complement and increment accumulator. Used to form two's complement.
CLL	7100	1	Clear L.
CLL RAL	7104	1, 2	Shift positive number one left.
CLL RTL	7106	1, 2	Clear link, rotate two left.
CLL RAR	7110	1, 2	Shift positive number one right.
CLL RTR	7112	1, 2	Clear link, rotate two right.
STL	7120	1	Set link. The L is set to contain a binary 1.
CLA	7200	1	Clear AC. To be used alone or in OPR 1 combinations.
CLA IAC	7201	1, 2	Set AC = 1
GLK	7204	1, 2	Get link. Transfer L into AC 11.
CLA CLL	7300	1	Clear AC and L.
STA	7240	1	Set AC = − 1. Each bit of the AC is set to contain a 1.

GROUP 2 OPERATE MICROINSTRUCTIONS

Mnemonic Symbol	Octal Code	Event Time	Operation
HLT	7402	1	Halt. Stops the program after completion of the cycle in process. If this instruction is combined with others in the OPR 2 group the other operations are completed before the end of the cycle.
OSR	7404	2	OR with switch register. The OR function is performed between the content of the SR and the content of the AC, with the result left in the AC.
SKP	7410	1	Skip, unconditional. The next instruction is skipped.
SNL	7420	1	Skip if $L \neq 0$.
SZL	7430	1	Skip if $L = 0$.
SZA	7440	1	Skip if $AC = 0$.
SNA	7450	1	Skip if $AC \neq 0$.
SZA SNL	7460	1	Skip if $AC = 0$, or $L = 1$, or both.
SNA SZL	7470	1	Skip if $AC \neq 0$ and $L = 0$.
SMA	7500	1	Skip on minus AC. If the content of the AC is a negative number, the next instruction is skipped.
SPA	7510	1	Skip on positive AC. If the content of the AC is a positive number, the next instruction is skipped.
SMA SNL	7520	1	Skip if $AC < 0$, or $L = 1$, or both.
SPA SZL	7530	1	Skip if $AC \overline{>} 0$ and if $L = 0$.
SMA SZA	7540	1	Skip if $AC \overline{>} 0$.
SPA SNA	7550	1	Skip if $AC > 0$.
CLA	7600	2	Clear AC. To be used alone or in OPR 2 combinations.
LAS	7604	1	Load AC with SR.
SZA CLA	7640	1	Skip if $AC = 0$, then clear AC.
SNA CLA	7650	1	Skip if $AC \neq 0$, then clear AC.
SMA CLA	7700	1	Skip if $AC < 0$, then clear AC.
SPA CLA	7710	1	Skip if $AC \geq 0$, then clear AC.

EXTENDED ARITHMETIC ELEMENT MICROINSTRUCTIONS

Mnemonic Symbol	Octal Code	Event Time	Operation
MUY	7405	2	Multiply. The number held in the MQ is multiplied by the number held in core memory location PC + 1 (or the next successive core memory location after the MUY command). At the conclusion of this command the most significant 12 bits of the product are contained in the AC and the least significant 12 bits of the product are contained in the MQ. Y x MQ => AC, MQ
DVI	7407	2	Divide. The 24-bit dividend held in the AC (most significant 12 bits) and the MQ (least significant 12 bits) is divided by the number held in core memory location PC + 1 (or the next successive core memory location following the DVI command). At the conclusion of this command the quotient is held in the MQ, the remainder is in the AC, and the L contains a 0. If the L contains a 1, divide overflow occurred so the operation was concluded after the first cycle of the division. AC, MQ $\div$ Y => MQ

NMI	7411	2	Normalize. This instruction is used as part of the conversion of a binary number to a fraction and an exponent for use in floating-point arithmetic. The combined content of the AC and the MQ is shifted left by this one command until the content of AC0 is not equal to the content of AC1, to form the fraction. Zeros are shifted into vacated MQ11 positions for each shift. At the conclusion of this operation, the step counter contains a number equal to the number of shifts performed. The content of L is lost.
			ACj => ACj − 1
			AC0 => L
			MQ 0 => AC11
			MQj = MQj −1
			0 => MQ11 until AC0 $\neq$ AC1

SHL	7413	2	Shift arithmetic left. This instruction shifts the combined content of the AC and MQ to the left one position more than the number of positions indicated by the content of core memory at address PC + 1 (or the next successive core memory location following the SHL command). During the shifting, zeros are shifted into vacated MQ11 positions.
			Shift Y + 1 positions as follows:
			ACj => ACj −1
			AC0 => L
			MQ0 => AC11
			MQj => MQj −1
			0 => MQ11

EXTENDED ARITHMETIC ELEMENT MICROINSTRUCTIONS
(continued)

Mnemonic Symbol	Octal Code	Event Time	Operation

ASR	7415	2	Arithmetic shift right. The combined content of the AC and the MQ is shifted right one position more than the number contained in memory location PC + 1 (or the next successive core memory location following the ASR command). The sign bit, contained in AC0, enters vacated positions, the sign bit is preserved, information shifted out of MQ11 is lost, and the L is undisturbed during this operation. Shift Y + 1 positions as follows:
			AC0 => AC0
			ACj => ACj + 1
			AC11 => MQ0
			MQj => MQj + 1

LSR	7417	2	Logical shift right. The combined content of the AC and MQ is shifted left one position more than the number contained in memory location PC + 1 (or the next successive core memory location following the LSR command). This command is similar to the ASR command except that zeros enter vacated positions instead of the sign bit entering these locations. Information shifted out of MQ11 is lost and the L is undisturbed during this operation.
			Shift Y + 1 positions as follows:
			0 => AC0
			ACj => ACj + 1
			AC11 => MQ0
			MQj => MQj + 1

MQL	7421	2	Load multiplier quotient. This command clears the MQ, loads the content of the AC into the MQ, then clears the AC. 0 => MQ AC => MQ 0 => AC
SCA	7441	2	Step counter load into accumulator. The content of the step counter is transferred into the AC. The AC should be cleared prior to issuing this command or the CLA command can be combined with the SCA to clear the AC,then effect the transfer. SC V AC => AC
MQA	7501	2	Multiplier quotient load into accumulator. The content of the MQ is transferred into the AC. This command is given to load the 12 least significant bits of the product into the AC following a multiplication or to load the quotient into the AC following a division. The AC should be cleared prior to issuing this command or the CLA command can be combined with the MQA to clear the AC then effect the transfer. MQ V AC = > AC

EXTENDED ARITHMETIC ELEMENT MICROINSTRUCTIONS
(continued)

Mnemonic Symbol	Octal Code	Event Time	Operation
CLA	7601	1	Clear accumulator. The AC is cleared during event time 1, allowing this command to be combined with the other EAE commands that load the AC during event time 2 (such as SCA and MQA). 0 => AC
CAM	7621	1, 2	Clear accumulator and multiplier quotient, CAM = CLA LMQ.

APPENDIX 2
CODES

MODEL 33 ASR/KSR TELETYPE CODE (ASCII) IN OCTAL FORM

Character	8-Bit Code (in octal)	Character	8-Bit Code (in octal)
A	301	!	241
B	302	“	242
C	303	#	243
D	304	$	244
E	305	%	245
F	306	&	246
G	307	‘	247
H	310	(	250
I	311	)	251
J	312	*	252
K	313	+	253
L	314	,	254
M	315	–	255
N	316	.	256
O	317	/	257
P	320	:	272
Q	321	;	273
R	322	<	274
S	323	=	275
T	324	>	276
U	325	?	277
V	326	@	300
W	327	[	333
X	330	\	334
Y	331	]	335
Z	332	↑	336
		←	337
0	260		
1	261	Leader/Trailer	200
2	262	Line-Feed	212
3	263	Carriage-Return	215
4	264	Space	240
5	265	Rub-out	377
6	266	Blank	000
7	267		
8	270		
9	271		

MODEL 33 ASR/KSR TELETYPE CODE (ASCII) IN BINARY FORM

1 = HOLE PUNCHED = MARK
0 = NO HOLE PUNCHED = SPACE

MOST SIGNIFICANT BIT
LEAST SIGNIFICANT BIT

				8	7	6	5	4	S	3	2	1
	@	SPACE	NULL/IDLE				0	0		0	0	0
	A	!	START OF MESSAGE				0	0		0	0	1
	B	"	END OF ADDRESS				0	0		0	1	0
	C	#	END OF MESSAGE				0	0		0	1	1
	D	$	END OF TRANSMISSION				0	0		1	0	0
	E	%	WHO ARE YOU				0	0		1	0	1
	F	&	ARE YOU				0	0		1	1	0
	G	'	BELL				0	0		1	1	1
	H	(	FORMAT EFFECTOR				0	1		0	0	0
	I	)	HORIZONTAL TAB				0	1		0	0	1
	J	*	LINE FEED				0	1		0	1	0
	K	+	VERTICAL TAB				0	1		0	1	1
	L	,	FORM FEED				0	1		1	0	0
	M	—	CARRIAGE RETURN				0	1		1	0	1
	N	.	SHIFT OUT				0	1		1	1	0
	O	/	SHIFT IN				0	1		1	1	1
	P	0	DCO				1	0		0	0	0
	Q	1	READER ON				1	0		0	0	1
	R	2	TAPE (AUX ON)				1	0		0	1	0
	S	3	READER OFF				1	0		0	1	1
	T	4	(AUX OFF)				1	0		1	0	0
	U	5	ERROR				1	0		1	0	1
	V	6	SYNCHRONOUS IDLE				1	0		1	1	0
	W	7	LOGICAL END OF MEDIA				1	0		1	1	1
	X	8	S 0				1	1		0	0	0
	Y	9	S 1				1	1		0	0	1
	Z	:	S 2				1	1		0	1	0
	[	;	S 3				1	1		0	1	1
	\	<	S 4				1	1		1	0	0
	]	=	S 5				1	1		1	0	1
	↑	>	S 6				1	1		1	1	0
RUB OUT	←	?	S 7				1	1		1	1	1

NULL/IDLE … S 7 →	1	0	0	SAME
SPACE … ? →	1	0	1	SAME
@ … ← →	1	1	0	SAME
… RUB OUT →	1	1	1	SAME

CARD READER CODE

Card Code		Internal Code	Character
Zone	Num.		
—	—	01 0000	Blank
12	8-3	11 1011	.
12	8-4	11 1100	)
12	8-5	11 1101	]
12	8-6	11 1110	<
12	8-7	11 1111	←
12	—	11 0000	+
11	8-3	10 1011	$
11	8-4	10 1100	*
11	8-5	10 1101	[
11	8-6	10 1110	>
11	8-7	10 1111	&
11	—	10 0000	—
0	1	01 0001	/
0	8-3	01 1011	,
0	8-4	01 1100	(
0	8-5	01 1101	"
0	8-6	01 1110	#
0	8-7	01 1111	%
—	8-3	00 1011	=
—	8-4	00 1100	@
—	8-5	00 1101	↑
—	8-6	00 1110	'
—	8-7	00 1111	\
12	0	11 1010	?
12	1	11 0001	A
12	2	11 0010	B
12	3	11 0011	C
12	4	11 0100	D
12	5	11 0101	E
12	6	11 0110	F
12	7	11 0111	G
12	8	11 1000	H
12	9	11 1001	I
11	0	10 1010	↑
11	1	10 0001	J
11	2	10 0010	K
11	3	10 0011	L
11	4	10 0100	M
11	5	10 0101	N
11	6	10 0110	O
11	7	10 0111	P
11	8	10 1000	Q
11	9	10 1001	R
0	8-2	01 1010	;
0	2	01 0010	S
0	3	01 0011	T
0	4	01 0100	U
0	5	01 0101	V
0	6	01 0110	W
0	7	01 0111	X
0	8	01 1000	Y
0	9	01 1001	Z
—	0	00 1010	0
—	1	00 0001	1
—	2	00 0010	2
—	3	00 0011	3
—	4	00 0100	4
—	5	00 0101	5
—	6	00 0110	6
—	7	00 0111	7
—	8	00 1000	8
—	9	00 1001	9
All other codes		00 0000	←

AUTOMATIC LINE PRINTER CODE

Character (ASCII)	6-Bit Code (in octal)	Character (ASCII)	6-Bit Code (in octal)
@	0	□	40
A	1	!	41
B	2	"	42
C	3	#	43
D	4	$	44
E	5	%	45
F	6	&	46
G	7	'	47
H	10	(	50
I	11	)	51
J	12	*	52
K	13	+	53
L	14	,	54
M	15	−	55
N	16	.	56
O	17	/	57
P	20	Ø	60
Q	21	1	61
R	22	2	62
S	23	3	63
T	24	4	64
U	25	5	65
V	26	6	66
W	27	7	67
X	30	8	70
Y	31	9	71
Z	32	:	72
[	33	;	73
\	34	<	74
]	35	=	75
↑	36	>	76
←	37	?	77

APPENDIX 3

SCALES OF NOTATION

2^x IN DECIMAL

x	2^x	x	2^x	x	2^x
0.001	1.00069 33874 62581	0.01	1.00695 55500 56719	0.1	1.07177 34625 36293
0.002	1.00138 72557 11335	0.02	1.01395 94797 90029	0.2	1.14869 83549 97035
0.003	1.00208 16050 79633	0.03	1.02101 21257 07193	0.3	1.23114 44133 44916
0.004	1.00277 64359 01078	0.04	1.02811 38266 56067	0.4	1.31950 79107 72894
0.005	1.00347 17485 09503	0.05	1.03526 49238 41377	0.5	1.41421 35623 73095
0.006	1.00416 75432 38973	0.06	1.04246 57608 41121	0.6	1.51571 65665 10398
0.007	1.00486 38204 23785	0.07	1.04971 66836 23067	0.7	1.62450 47927 12471
0.008	1.00556 05803 98468	0.08	1.05701 80405 61380	0.8	1.74110 11265 92248
0.009	1.00625 78234 97782	0.09	1.06437 01824 53360	0.9	1.86606 59830 73615

$10^{\pm n}$ IN OCTAL

10^n	n	10^{-n}	10^n	n	10^{-n}
1	0	1.000 000 000 000 000 000 00	112 402 762 000	10	0.000 000 000 006 676 337 66
12	1	0.063 146 314 631 463 146 31	1 351 035 564 000	11	0.000 000 000 000 537 657 77
144	2	0.005 075 341 217 270 243 66	16 432 451 210 000	12	0.000 000 000 000 043 136 32
1 750	3	0.000 406 111 564 570 651 77	221 411 634 520 000	13	0.000 000 000 000 003 411 35
23 420	4	0.000 032 155 613 530 704 15	2 657 142 036 440 000	14	0.000 000 000 000 000 264 11
303 240	5	0.000 002 476 132 610 706 64	34 327 724 461 500 000	15	0.000 000 000 000 000 022 01
3 641 100	6	0.000 000 206 157 364 055 37	434 157 115 760 200 000	16	0.000 000 000 000 000 001 63
46 113 200	7	0.000 000 015 327 745 152 75	5 432 127 413 542 400 000	17	0.000 000 000 000 000 000 14
575 360 400	8	0.000 000 001 257 143 561 06	67 405 553 164 731 000 000	18	0.000 000 000 000 000 000 01
7 346 545 000	9	0.000 000 000 104 560 276 41			

$n \log_{10} 2$, $n \log_2 10$ IN DECIMAL

n	$n \log_{10} 2$	$n \log_2 10$	n	$n \log_{10} 2$	$n \log_2 10$
1	0.30102 99957	3.32192 80949	6	1.80617 99740	19.93156 85693
2	0.60205 99913	6.64385 61898	7	2.10720 99696	23.25349 66642
3	0.90308 99870	9.96578 42847	8	2.40823 99653	26.57542 47591
4	1.20411 99827	13.28771 23795	9	2.70926 99610	29.89735 28540
5	1.50514 99783	16.60964 04744	10	3.01029 99566	33.21928 09489

ADDITION AND MULTIPLICATION TABLES

Addition — Multiplication

Binary Scale

$$0 + 0 = 0$$
$$0 + 1 = 1 + 0 = 1$$
$$1 + 1 = 10$$

$$0 \times 0 = 0$$
$$0 \times 1 = 1 \times 0 = 0$$
$$1 \times 1 = 1$$

Octal Scale

0	01	02	03	04	05	06	07
1	02	03	04	05	06	07	10
2	03	04	05	06	07	10	11
3	04	05	06	07	10	11	12
4	05	06	07	10	11	12	13
5	06	07	10	11	12	13	14
6	07	10	11	12	13	14	15
7	10	11	12	13	14	15	16

1	02	03	04	05	06	07
2	04	06	10	12	14	16
3	06	11	14	17	22	25
4	10	14	20	24	30	34
5	12	17	24	31	36	43
6	14	22	30	36	44	52
7	16	25	34	43	52	61

MATHEMATICAL CONSTANTS IN OCTAL SCALE

$\pi = 3.11037\ 552421_8$	$e = 2.55760\ 521305_8$	$\gamma = 0.44742\ 147707_8$
$\pi^{-1} = 0.24276\ 301556_8$	$e^{-1} = 0.27426\ 530661_8$	$\ln \gamma = -0.43127\ 233602_8$
$\sqrt{\pi} = 1.61337\ 611067_8$	$\sqrt{e} = 1.51411\ 230704_8$	$\log_2 \gamma = -0.62573\ 030645_8$
$\ln \pi = 1.11206\ 404435_8$	$\log_{10} e = 0.33626\ 754251_8$	$\sqrt{2} = 1.32404\ 746320_8$
$\log_2 \pi = 1.51544\ 163223_8$	$\log_2 e = 1.34252\ 166245_8$	$\ln 2 = 0.54271\ 027760_8$
$\sqrt{10} = 3.12305\ 407267_8$	$\log_2 10 = 3.24464\ 741136_8$	$\ln 10 = 2.23273\ 067355_8$

APPENDIX 4
POWERS OF TWO

2^n	n	2^{-n}
1	0	1.0
2	1	0.5
4	2	0.25
8	3	0.125
16	4	0.062 5
32	5	0.031 25
64	6	0.015 625
128	7	0.007 812 5
256	8	0.003 906 25
512	9	0.001 953 125
1 024	10	0.000 976 562 5
2 048	11	0.000 488 281 25
4 096	12	0.000 244 140 625
8 192	13	0.000 122 070 312 5
16 384	14	0.000 061 035 156 25
32 768	15	0.000 030 517 578 125
65 536	16	0.000 015 258 789 062 5
131 072	17	0.000 007 629 394 531 25
262 144	18	0.000 003 814 697 265 625
524 288	19	0.000 001 907 348 632 812 5
1 048 576	20	0.000 000 953 674 316 406 25
2 097 152	21	0.000 000 476 837 158 203 125
4 194 304	22	0.000 000 238 418 579 101 562 5
8 388 608	23	0.000 000 119 209 289 550 781 25
16 777 216	24	0.000 000 059 604 644 775 390 625
33 554 432	25	0.000 000 029 802 322 387 695 312 5
67 108 864	26	0.000 000 014 901 161 193 847 656 25
134 217 728	27	0.000 000 007 450 580 596 923 828 125
268 435 456	28	0.000 000 003 725 290 298 461 914 062 5
536 870 912	29	0.000 000 001 862 645 149 230 957 031 25
1 073 741 824	30	0.000 000 000 931 322 574 615 478 515 625
2 147 483 648	31	0.000 000 000 465 661 287 307 739 257 812 5
4 294 967 296	32	0.000 000 000 232 830 643 653 869 628 906 25
8 589 934 592	33	0.000 000 000 116 415 321 826 934 814 453 125
17 179 869 184	34	0.000 000 000 058 207 660 913 467 407 226 562 5
34 359 738 368	35	0.000 000 000 029 103 830 456 733 703 613 281 25
68 719 476 736	36	0.000 000 000 014 551 915 228 366 851 806 640 625
137 438 953 472	37	0.000 000 000 007 275 957 614 183 425 903 320 312 5
274 877 906 944	38	0.000 000 000 003 637 978 807 091 712 951 660 156 25
549 755 813 888	39	0.000 000 000 001 818 989 403 545 856 475 830 078 125
1 099 511 627 776	40	0.000 000 000 000 909 494 701 772 928 237 915 039 062 5
2 199 023 255 552	41	0.000 000 000 000 454 747 350 886 464 118 957 519 531 25
4 398 046 511 104	42	0.000 000 000 000 227 373 675 443 232 059 478 759 765 625
8 796 093 022 208	43	0.000 000 000 000 113 686 837 721 616 029 739 379 882 812 5
17 592 186 044 416	44	0.000 000 000 000 056 843 418 860 808 014 869 689 941 406 25
35 184 372 088 832	45	0.000 000 000 000 028 421 709 430 404 007 434 844 970 703 125
70 368 744 177 664	46	0.000 000 000 000 014 210 854 715 202 003 717 422 485 351 562 5
140 737 488 355 328	47	0.000 000 000 000 007 105 427 357 601 001 858 711 242 675 781 25
281 474 976 710 656	48	0.000 000 000 000 003 552 713 678 800 500 929 355 621 337 890 625
562 949 953 421 312	49	0.000 000 000 000 001 776 356 839 400 250 464 677 810 668 945 312 5
1 125 899 906 842 624	50	0.000 000 000 000 000 888 178 419 700 125 232 338 905 334 472 656 25
2 251 799 813 685 248	51	0.000 000 000 000 000 444 089 209 850 062 616 169 452 667 236 328 125
4 503 599 627 370 496	52	0.000 000 000 000 000 222 044 604 925 031 308 084 726 333 618 164 062 5
9 007 199 254 740 992	53	0.000 000 000 000 000 111 022 302 462 515 654 042 363 166 809 082 031 25
18 014 398 509 481 984	54	0.000 000 000 000 000 055 511 151 231 257 827 021 181 583 404 541 015 625
36 028 797 018 963 968	55	0.000 000 000 000 000 027 755 575 615 628 913 510 590 791 702 270 507 812 5
72 057 594 037 927 936	56	0.000 000 000 000 000 013 877 787 807 814 456 755 295 395 851 135 253 906 25
144 115 188 075 855 872	57	0.000 000 000 000 000 006 938 893 903 907 228 377 647 697 925 567 626 953 125
288 230 376 151 711 744	58	0.000 000 000 000 000 003 469 446 951 953 614 188 823 848 962 783 813 476 562 5
576 460 752 303 423 488	59	0.000 000 000 000 000 001 734 723 475 976 807 094 411 924 481 391 906 738 281 25
1 152 921 504 606 846 976	60	0.000 000 000 000 000 000 867 361 737 988 403 547 205 962 240 695 953 369 140 625
2 305 843 009 213 693 952	61	0.000 000 000 000 000 000 433 680 868 994 201 773 602 981 120 347 976 684 570 312 5
4 611 686 018 427 387 904	62	0.000 000 000 000 000 000 216 840 434 497 100 886 801 490 560 173 988 342 285 156 25
9 223 372 036 854 775 808	63	0.000 000 000 000 000 000 108 420 217 248 550 443 400 745 280 086 994 171 142 578 125
18 446 744 073 709 551 616	64	0.000 000 000 000 000 000 054 210 108 624 275 221 700 372 640 043 497 085 571 289 062 5
36 893 488 147 419 103 232	65	0.000 000 000 000 000 000 027 105 054 312 137 610 850 186 320 021 748 542 785 644 531 25
73 786 976 294 838 206 464	66	0.000 000 000 000 000 000 013 552 527 156 068 805 425 093 160 010 874 271 392 822 265 625
147 573 952 589 676 412 928	67	0.000 000 000 000 000 000 006 776 263 578 034 402 712 546 580 005 437 135 696 411 132 812 5
295 147 905 179 352 825 856	68	0.000 000 000 000 000 000 003 388 131 789 017 201 356 273 290 002 718 567 848 205 566 406 25
590 295 810 358 705 651 712	69	0.000 000 000 000 000 000 001 694 065 894 508 600 678 136 645 001 359 283 924 102 783 203 125
1 180 591 620 717 411 303 424	70	0.000 000 000 000 000 000 000 847 032 947 254 300 339 068 322 500 679 641 962 051 391 601 562 5
2 361 183 241 434 822 606 848	71	0.000 000 000 000 000 000 000 423 516 473 627 150 169 534 161 250 339 820 981 025 695 800 781 25
4 722 366 482 869 645 213 696	72	0.000 000 000 000 000 000 000 211 758 236 813 575 084 767 080 625 169 910 490 512 847 900 390 625

APPENDIX 5
OCTAL-DECIMAL CONVERSION
OCTAL-DECIMAL INTEGER CONVERSION TABLE

0000 to 0777 (Octal)	0000 to 0511 (Decimal)

Octal	Decimal
10000 -	4096
20000 -	8192
30000 -	12288
40000 -	16384
50000 -	20480
60000 -	24576
70000 -	28672

	0	1	2	3	4	5	6	7
0000	0000	0001	0002	0003	0004	0005	0006	0007
0010	0008	0009	0010	0011	0012	0013	0014	0015
0020	0016	0017	0018	0019	0020	0021	0022	0023
0030	0024	0025	0026	0027	0028	0029	0030	0031
0040	0032	0033	0034	0035	0036	0037	0038	0039
0050	0040	0041	0042	0043	0044	0045	0046	0047
0060	0048	0049	0050	0051	0052	0053	0054	0055
0070	0056	0057	0058	0059	0060	0061	0062	0063
0100	0064	0065	0066	0067	0068	0069	0070	0071
0110	0072	0073	0074	0075	0076	0077	0078	0079
0120	0080	0081	0082	0083	0084	0085	0086	0087
0130	0088	0089	0090	0091	0092	0093	0094	0095
0140	0096	0097	0098	0099	0100	0101	0102	0103
0150	0104	0105	0106	0107	0108	0109	0110	0111
0160	0112	0113	0114	0115	0116	0117	0118	0119
0170	0120	0121	0122	0123	0124	0125	0126	0127
0200	0128	0129	0130	0131	0132	0133	0134	0135
0210	0136	0137	0138	0139	0140	0141	0142	0143
0220	0144	0145	0146	0147	0148	0149	0150	0151
0230	0152	0153	0154	0155	0156	0157	0158	0159
0240	0160	0161	0162	0163	0164	0165	0166	0167
0250	0168	0169	0170	0171	0172	0173	0174	0175
0260	0176	0177	0178	0179	0180	0181	0182	0183
0270	0184	0185	0186	0187	0188	0189	0190	0191
0300	0192	0193	0194	0195	0196	0197	0198	0199
0310	0200	0201	0202	0203	0204	0205	0206	0207
0320	0208	0209	0210	0211	0212	0213	0214	0215
0330	0216	0217	0218	0219	0220	0221	0222	0223
0340	0224	0225	0226	0227	0228	0229	0230	0231
0350	0232	0233	0234	0235	0236	0237	0238	0239
0360	0240	0241	0242	0243	0244	0245	0246	0247
0370	0248	0249	0250	0251	0252	0253	0254	0255
0400	0256	0257	0258	0259	0260	0261	0262	0263
0410	0264	0265	0266	0267	0268	0269	0270	0271
0420	0272	0273	0274	0275	0276	0277	0278	0279
0430	0280	0281	0282	0283	0284	0285	0286	0287
0440	0288	0289	0290	0291	0292	0293	0294	0295
0450	0296	0297	0298	0299	0300	0301	0302	0303
0460	0304	0305	0306	0307	0308	0309	0310	0311
0470	0312	0313	0314	0315	0316	0317	0318	0319
0500	0320	0321	0322	0323	0324	0325	0326	0327
0510	0328	0329	0330	0331	0332	0333	0334	0335
0520	0336	0337	0338	0339	0340	0341	0342	0343
0530	0344	0345	0346	0347	0348	0349	0350	0351
0540	0352	0353	0354	0355	0356	0357	0358	0359
0550	0360	0361	0362	0363	0364	0365	0366	0367
0560	0368	0369	0370	0371	0372	0373	0374	0375
0570	0376	0377	0378	0379	0380	0381	0382	0383
0600	0384	0385	0386	0387	0388	0389	0390	0391
0610	0392	0393	0394	0395	0396	0397	0398	0399
0620	0400	0401	0402	0403	0404	0405	0406	0407
0630	0408	0409	0410	0411	0412	0413	0414	0415
0640	0416	0417	0418	0419	0420	0421	0422	0423
0650	0424	0425	0426	0427	0428	0429	0430	0431
0660	0432	0433	0434	0435	0436	0437	0438	0439
0670	0440	0441	0442	0443	0444	0445	0446	0447
0700	0448	0449	0450	0451	0452	0453	0454	0455
0710	0456	0457	0458	0459	0460	0461	0462	0463
0720	0464	0465	0466	0467	0468	0469	0470	0471
0730	0472	0473	0474	0475	0476	0477	0478	0479
0740	0480	0481	0482	0483	0484	0485	0486	0487
0750	0488	0489	0490	0491	0492	0493	0494	0495
0760	0496	0497	0498	0499	0500	0501	0502	0503
0770	0504	0505	0506	0507	0508	0509	0510	0511

1000 to 1777 (Octal)	0512 to 1023 (Decimal)

	0	1	2	3	4	5	6	7
1000	0512	0513	0514	0515	0516	0517	0518	0519
1010	0520	0521	0522	0523	0524	0525	0526	0527
1020	0528	0529	0530	0531	0532	0533	0534	0535
1030	0536	0537	0538	0539	0540	0541	0542	0543
1040	0544	0545	0546	0547	0548	0549	0550	0551
1050	0552	0553	0554	0555	0556	0557	0558	0559
1060	0560	0561	0562	0563	0564	0565	0566	0567
1070	0568	0569	0570	0571	0572	0573	0574	0575
1100	0576	0577	0578	0579	0580	0581	0582	0583
1110	0584	0585	0586	0587	0588	0589	0590	0591
1120	0592	0593	0594	0595	0596	0597	0598	0599
1130	0600	0601	0602	0603	0604	0605	0606	0607
1140	0608	0609	0610	0611	0612	0613	0614	0615
1150	0616	0617	0618	0619	0620	0621	0622	0623
1160	0624	0625	0626	0627	0628	0629	0630	0631
1170	0632	0633	0634	0635	0636	0637	0638	0639
1200	0640	0641	0642	0643	0644	0645	0646	0647
1210	0648	0649	0650	0651	0652	0653	0654	0655
1220	0656	0657	0658	0659	0660	0661	0662	0663
1230	0664	0665	0666	0667	0668	0669	0670	0671
1240	0672	0673	0674	0675	0676	0677	0678	0679
1250	0680	0681	0682	0683	0684	0685	0686	0687
1260	0688	0689	0690	0691	0692	0693	0694	0695
1270	0696	0697	0698	0699	0700	0701	0702	0703
1300	0704	0705	0706	0707	0708	0709	0710	0711
1310	0712	0713	0714	0715	0716	0717	0718	0719
1320	0720	0721	0722	0723	0724	0725	0726	0727
1330	0728	0729	0730	0731	0732	0733	0734	0735
1340	0736	0737	0738	0739	0740	0741	0742	0743
1350	0744	0745	0746	0747	0748	0749	0750	0751
1360	0752	0753	0754	0755	0756	0757	0758	0759
1370	0760	0761	0762	0763	0764	0765	0766	0767
1400	0768	0769	0770	0771	0772	0773	0774	0775
1410	0776	0777	0778	0779	0780	0781	0782	0783
1420	0784	0785	0786	0787	0788	0789	0790	0791
1430	0792	0793	0794	0795	0796	0797	0798	0799
1440	0800	0801	0802	0803	0804	0805	0806	0807
1450	0808	0809	0810	0811	0812	0813	0814	0815
1460	0816	0817	0818	0819	0820	0821	0822	0823
1470	0824	0825	0826	0827	0828	0829	0830	0831
1500	0832	0833	0834	0835	0836	0837	0838	0839
1510	0840	0841	0842	0843	0844	0845	0846	0847
1520	0848	0849	0850	0851	0852	0853	0854	0855
1530	0856	0857	0858	0859	0860	0861	0862	0863
1540	0864	0865	0866	0867	0868	0869	0870	0871
1550	0872	0873	0874	0875	0876	0877	0878	0879
1560	0880	0881	0882	0883	0884	0885	0886	0887
1570	0888	0889	0890	0891	0892	0893	0894	0895
1600	0896	0897	0898	0899	0900	0901	0902	0903
1610	0904	0905	0906	0907	0908	0909	0910	0911
1620	0912	0913	0914	0915	0916	0917	0918	0919
1630	0920	0921	0922	0923	0924	0925	0926	0927
1640	0928	0929	0930	0931	0932	0933	0934	0935
1650	0936	0937	0938	0939	0940	0941	0942	0943
1660	0944	0945	0946	0947	0948	0949	0950	0951
1670	0952	0953	0954	0955	0956	0957	0958	0959
1700	0960	0961	0962	0963	0964	0965	0966	0967
1710	0968	0969	0970	0971	0972	0973	0974	0975
1720	0976	0977	0978	0979	0980	0981	0982	0983
1730	0984	0985	0986	0987	0988	0989	0990	0991
1740	0992	0993	0994	0995	0996	0997	0998	0999
1750	1000	1001	1002	1003	1004	1005	1006	1007
1760	1008	1009	1010	1011	1012	1013	1014	1015
1770	1016	1017	1018	1019	1020	1021	1022	1023

OCTAL-DECIMAL INTEGER CONVERSION TABLE (continued)

	0	1	2	3	4	5	6	7
2000	1024	1025	1026	1027	1028	1029	1030	1031
2010	1032	1033	1034	1035	1036	1037	1038	1039
2020	1040	1041	1042	1043	1044	1045	1046	1047
2030	1048	1049	1050	1051	1052	1053	1054	1055
2040	1056	1057	1058	1059	1060	1061	1062	1063
2050	1064	1065	1066	1067	1068	1069	1070	1071
2060	1072	1073	1074	1075	1076	1077	1078	1079
2070	1080	1081	1082	1083	1084	1085	1086	1087
2100	1088	1089	1090	1091	1092	1093	1094	1095
2110	1096	1097	1098	1099	1100	1101	1102	1103
2120	1104	1105	1106	1107	1108	1109	1110	1111
2130	1112	1113	1114	1115	1116	1117	1118	1119
2140	1120	1121	1122	1123	1124	1125	1126	1127
2150	1128	1129	1130	1131	1132	1133	1134	1135
2160	1136	1137	1138	1139	1140	1141	1142	1143
2170	1144	1145	1146	1147	1148	1149	1150	1151
2200	1152	1153	1154	1155	1156	1157	1158	1159
2210	1160	1161	1162	1163	1164	1165	1166	1167
2220	1168	1169	1170	1171	1172	1173	1174	1175
2230	1176	1177	1178	1179	1180	1181	1182	1183
2240	1184	1185	1186	1187	1188	1189	1190	1191
2250	1192	1193	1194	1195	1196	1197	1198	1199
2260	1200	1201	1202	1203	1204	1205	1206	1207
2270	1208	1209	1210	1211	1212	1213	1214	1215
2300	1216	1217	1218	1219	1220	1221	1222	1223
2310	1224	1225	1226	1227	1228	1229	1230	1231
2320	1232	1233	1234	1235	1236	1237	1238	1239
2330	1240	1241	1242	1243	1244	1245	1246	1247
2340	1248	1249	1250	1251	1252	1253	1254	1255
2350	1256	1257	1258	1259	1260	1261	1262	1263
2360	1264	1265	1266	1267	1268	1269	1270	1271
2370	1272	1273	1274	1275	1276	1277	1278	1279

	0	1	2	3	4	5	6	7
2400	1280	1281	1282	1283	1284	1285	1286	1287
2410	1288	1289	1290	1291	1292	1293	1294	1295
2420	1296	1297	1298	1299	1300	1301	1302	1303
2430	1304	1305	1306	1307	1308	1309	1310	1311
2440	1312	1313	1314	1315	1316	1317	1318	1319
2450	1320	1321	1322	1323	1324	1325	1326	1327
2460	1328	1329	1330	1331	1332	1333	1334	1335
2470	1336	1337	1338	1339	1340	1341	1342	1343
2500	1344	1345	1346	1347	1348	1349	1350	1351
2510	1352	1353	1354	1355	1356	1357	1358	1359
2520	1360	1361	1362	1363	1364	1365	1366	1367
2530	1368	1369	1370	1371	1372	1373	1374	1375
2540	1376	1377	1378	1379	1380	1381	1382	1383
2550	1384	1385	1386	1387	1388	1389	1390	1391
2560	1392	1393	1394	1395	1396	1397	1398	1399
2570	1400	1401	1402	1403	1404	1405	1406	1407
2600	1408	1409	1410	1411	1412	1413	1414	1415
2610	1416	1417	1418	1419	1420	1421	1422	1423
2620	1424	1425	1426	1427	1428	1429	1430	1431
2630	1432	1433	1434	1435	1436	1437	1438	1439
2640	1440	1441	1442	1443	1444	1445	1446	1447
2650	1448	1449	1450	1451	1452	1453	1454	1455
2660	1456	1457	1458	1459	1460	1461	1462	1463
2670	1464	1465	1466	1467	1468	1469	1470	1471
2700	1472	1473	1474	1475	1476	1477	1478	1479
2710	1480	1481	1482	1483	1484	1485	1486	1487
2720	1488	1489	1490	1491	1492	1493	1494	1495
2730	1496	1497	1498	1499	1500	1501	1502	1503
2740	1504	1505	1506	1507	1508	1509	1510	1511
2750	1512	1513	1514	1515	1516	1517	1518	1519
2760	1520	1521	1522	1523	1524	1525	1526	1527
2770	1528	1529	1530	1531	1532	1533	1534	1535

2000 to 2777 (Octal)	1024 to 1535 (Decimal)

Octal	Decimal
10000 -	4096
20000 -	8192
30000 -	12288
40000 -	16384
50000 -	20480
60000 -	24576
70000 -	28672

	0	1	2	3	4	5	6	7
3000	1536	1537	1538	1539	1540	1541	1542	1543
3010	1544	1545	1546	1547	1548	1549	1550	1551
3020	1552	1553	1554	1555	1556	1557	1558	1559
3030	1560	1561	1562	1563	1564	1565	1566	1567
3040	1568	1569	1570	1571	1572	1573	1574	1575
3050	1576	1577	1578	1579	1580	1581	1582	1583
3060	1584	1585	1586	1587	1588	1589	1590	1591
3070	1592	1593	1594	1595	1596	1597	1598	1599
3100	1600	1601	1602	1603	1604	1605	1606	1607
3110	1608	1609	1610	1611	1612	1613	1614	1615
3120	1616	1617	1618	1619	1620	1621	1622	1623
3130	1624	1625	1626	1627	1628	1629	1630	1631
3140	1632	1633	1634	1635	1636	1637	1638	1639
3150	1640	1641	1642	1643	1644	1645	1646	1647
3160	1648	1649	1650	1651	1652	1653	1654	1655
3170	1656	1657	1658	1659	1660	1661	1662	1663
3200	1664	1665	1666	1667	1668	1669	1670	1671
3210	1672	1673	1674	1675	1676	1677	1678	1679
3220	1680	1681	1682	1683	1684	1685	1686	1687
3230	1688	1689	1690	1691	1692	1693	1694	1695
3240	1696	1697	1698	1699	1700	1701	1702	1703
3250	1704	1705	1706	1707	1708	1709	1710	1711
3260	1712	1713	1714	1715	1716	1717	1718	1719
3270	1720	1721	1722	1723	1724	1725	1726	1727
3300	1728	1729	1730	1731	1732	1733	1734	1735
3310	1736	1737	1738	1739	1740	1741	1742	1743
3320	1744	1745	1746	1747	1748	1749	1750	1751
3330	1752	1753	1754	1755	1756	1757	1758	1759
3340	1760	1761	1762	1763	1764	1765	1766	1767
3350	1768	1769	1770	1771	1772	1773	1774	1775
3360	1776	1777	1778	1779	1780	1781	1782	1783
3370	1784	1785	1786	1787	1788	1789	1790	1791

	0	1	2	3	4	5	6	7
3400	1792	1793	1794	1795	1796	1797	1798	1799
3410	1800	1801	1802	1803	1804	1805	1806	1807
3420	1808	1809	1810	1811	1812	1813	1814	1815
3430	1816	1817	1818	1819	1820	1821	1822	1823
3440	1824	1825	1826	1827	1828	1829	1830	1831
3450	1832	1833	1834	1835	1836	1837	1838	1839
3460	1840	1841	1842	1843	1844	1845	1846	1847
3470	1848	1849	1850	1851	1852	1853	1854	1855
3500	1856	1857	1858	1859	1860	1861	1862	1863
3510	1864	1865	1866	1867	1868	1869	1870	1871
3520	1872	1873	1874	1875	1876	1877	1878	1879
3530	1880	1881	1882	1883	1884	1885	1886	1887
3540	1888	1889	1890	1891	1892	1893	1894	1895
3550	1896	1897	1898	1899	1900	1901	1902	1903
3560	1904	1905	1906	1907	1908	1909	1910	1911
3570	1912	1913	1914	1915	1916	1917	1918	1919
3600	1920	1921	1922	1923	1924	1925	1926	1927
3610	1928	1929	1930	1931	1932	1933	1934	1935
3620	1936	1937	1938	1939	1940	1941	1942	1943
3630	1944	1945	1946	1947	1948	1949	1950	1951
3640	1952	1953	1954	1955	1956	1957	1958	1959
3650	1960	1961	1962	1963	1964	1965	1966	1967
3660	1968	1969	1970	1971	1972	1973	1974	1975
3670	1976	1977	1978	1979	1980	1981	1982	1983
3700	1984	1985	1986	1987	1988	1989	1990	1991
3710	1992	1993	1994	1995	1996	1997	1998	1999
3720	2000	2001	2002	2003	2004	2005	2006	2007
3730	2008	2009	2010	2011	2012	2013	2014	2015
3740	2016	2017	2018	2019	2020	2021	2022	2023
3750	2024	2025	2026	2027	2028	2029	2030	2031
3760	2032	2033	2034	2035	2036	2037	2038	2039
3770	2040	2041	2042	2043	2044	2045	2046	2047

3000 to 3777 (Octal)	1536 to 2047 (Decimal)

OCTAL-DECIMAL INTEGER CONVERSION TABLE (continued)

4000	2048
to	to
4777	2559
(Octal)	(Decimal)

Octal	Decimal
10000 -	4096
20000 -	8192
30000 -	12288
40000 -	16384
50000 -	20480
60000 -	24576
70000 -	28672

	0	1	2	3	4	5	6	7
4000	2048	2049	2050	2051	2052	2053	2054	2055
4010	2056	2057	2058	2059	2060	2061	2062	2063
4020	2064	2065	2066	2067	2068	2069	2070	2071
4030	2072	2073	2074	2075	2076	2077	2078	2079
4040	2080	2081	2082	2083	2084	2085	2086	2087
4050	2088	2089	2090	2091	2092	2093	2094	2095
4060	2096	2097	2098	2099	2100	2101	2102	2103
4070	2104	2105	2106	2107	2108	2109	2110	2111
4100	2112	2113	2114	2115	2116	2117	2118	2119
4110	2120	2121	2122	2123	2124	2125	2126	2127
4120	2128	2129	2130	2131	2132	2133	2134	2135
4130	2136	2137	2138	2139	2140	2141	2142	2143
4140	2144	2145	2146	2147	2148	2149	2150	2151
4150	2152	2153	2154	2155	2156	2157	2158	2159
4160	2160	2161	2162	2163	2164	2165	2166	2167
4170	2168	2169	2170	2171	2172	2173	2174	2175
4200	2176	2177	2178	2179	2180	2181	2182	2183
4210	2184	2185	2186	2187	2188	2189	2190	2191
4220	2192	2193	2194	2195	2196	2197	2198	2199
4230	2200	2201	2202	2203	2204	2205	2206	2207
4240	2208	2209	2210	2211	2212	2213	2214	2215
4250	2216	2217	2218	2219	2220	2221	2222	2223
4260	2224	2225	2226	2227	2228	2229	2230	2231
4270	2232	2233	2234	2235	2236	2237	2238	2239
4300	2240	2241	2242	2243	2244	2245	2246	2247
4310	2248	2249	2250	2251	2252	2253	2254	2255
4320	2256	2257	2258	2259	2260	2261	2262	2263
4330	2264	2265	2266	2267	2268	2269	2270	2271
4340	2272	2273	2274	2275	2276	2277	2278	2279
4350	2280	2281	2282	2283	2284	2285	2286	2287
4360	2288	2289	2290	2291	2292	2293	2294	2295
4370	2296	2297	2298	2299	2300	2301	2302	2303

	0	1	2	3	4	5	6	7
4400	2304	2305	2306	2307	2308	2309	2310	2311
4410	2312	2313	2314	2315	2316	2317	2318	2319
4420	2320	2321	2322	2323	2324	2325	2326	2327
4430	2328	2329	2330	2331	2332	2333	2334	2335
4440	2336	2337	2338	2339	2340	2341	2342	2343
4450	2344	2345	2346	2347	2348	2349	2350	2351
4460	2352	2353	2354	2355	2356	2357	2358	2359
4470	2360	2361	2362	2363	2364	2365	2366	2367
4500	2368	2369	2370	2371	2372	2373	2374	2375
4510	2376	2377	2378	2379	2380	2381	2382	2383
4520	2384	2385	2386	2387	2388	2389	2390	2391
4530	2392	2393	2394	2395	2396	2397	2398	2399
4540	2400	2401	2402	2403	2404	2405	2406	2407
4550	2408	2409	2410	2411	2412	2413	2414	2415
4560	2416	2417	2418	2419	2420	2421	2422	2423
4570	2424	2425	2426	2427	2428	2429	2430	2431
4600	2432	2433	2434	2435	2436	2437	2438	2439
4610	2440	2441	2442	2443	2444	2445	2446	2447
4620	2448	2449	2450	2451	2452	2453	2454	2455
4630	2456	2457	2458	2459	2460	2461	2462	2463
4640	2464	2465	2466	2467	2468	2469	2470	2471
4650	2472	2473	2474	2475	2476	2477	2478	2479
4660	2480	2481	2482	2483	2484	2485	2486	2487
4670	2488	2489	2490	2491	2492	2493	2494	2495
4700	2496	2497	2498	2499	2500	2501	2502	2503
4710	2504	2505	2506	2507	2508	2509	2510	2511
4720	2512	2513	2514	2515	2516	2517	2518	2519
4730	2520	2521	2522	2523	2524	2525	2526	2527
4740	2528	2529	2530	2531	2532	2533	2534	2535
4750	2536	2537	2538	2539	2540	2541	2542	2543
4760	2544	2545	2546	2547	2548	2549	2550	2551
4770	2552	2553	2554	2555	2556	2557	2558	2559

5000	2560
to	to
5777	3071
(Octal)	(Decimal)

	0	1	2	3	4	5	6	7
5000	2560	2561	2562	2563	2564	2565	2566	2567
5010	2568	2569	2570	2571	2572	2573	2574	2575
5020	2576	2577	2578	2579	2580	2581	2582	2583
5030	2584	2585	2586	2587	2588	2589	2590	2591
5040	2592	2593	2594	2595	2596	2597	2598	2599
5050	2600	2601	2602	2603	2604	2605	2606	2607
5060	2608	2609	2610	2611	2612	2613	2614	2615
5070	2616	2617	2618	2619	2620	2621	2622	2623
5100	2624	2625	2626	2627	2628	2629	2630	2631
5110	2632	2633	2634	2635	2636	2637	2638	2639
5120	2640	2641	2642	2643	2644	2645	2646	2647
5130	2648	2649	2650	2651	2652	2653	2654	2655
5140	2656	2657	2658	2659	2660	2661	2662	2663
5150	2664	2665	2666	2667	2668	2669	2670	2671
5160	2672	2673	2674	2675	2676	2677	2678	2679
5170	2680	2681	2682	2683	2684	2685	2686	2687
5200	2688	2689	2690	2691	2692	2693	2694	2695
5210	2696	2697	2698	2699	2700	2701	2702	2703
5220	2704	2705	2706	2707	2708	2709	2710	2711
5230	2712	2713	2714	2715	2716	2717	2718	2719
5240	2720	2721	2722	2723	2724	2725	2726	2727
5250	2728	2729	2730	2731	2732	2733	2734	2735
5260	2736	2737	2738	2739	2740	2741	2742	2743
5270	2744	2745	2746	2747	2748	2749	2750	2751
5300	2752	2753	2754	2755	2756	2757	2758	2759
5310	2760	2761	2762	2763	2764	2765	2766	2767
5320	2768	2769	2770	2771	2772	2773	2774	2775
5330	2776	2777	2778	2779	2780	2781	2782	2783
5340	2784	2785	2786	2787	2788	2789	2790	2791
5350	2792	2793	2794	2795	2796	2797	2798	2799
5360	2800	2801	2802	2803	2804	2805	2806	2807
5370	2808	2809	2810	2811	2812	2813	2814	2815

	0	1	2	3	4	5	6	7
5400	2816	2817	2818	2819	2820	2821	2822	2823
5410	2824	2825	2826	2827	2828	2829	2830	2831
5420	2832	2833	2834	2835	2836	2837	2838	2839
5430	2840	2841	2842	2843	2844	2845	2846	2847
5440	2848	2849	2850	2851	2852	2853	2854	2855
5450	2856	2857	2858	2859	2860	2861	2862	2863
5460	2864	2865	2866	2867	2868	2869	2870	2871
5470	2872	2873	2874	2875	2876	2877	2878	2879
5500	2880	2881	2882	2883	2884	2885	2886	2887
5510	2888	2889	2890	2891	2892	2893	2894	2895
5520	2896	2897	2898	2899	2900	2901	2902	2903
5530	2904	2905	2906	2907	2908	2909	2910	2911
5540	2912	2913	2914	2915	2916	2917	2918	2919
5550	2920	2921	2922	2923	2924	2925	2926	2927
5560	2928	2929	2930	2931	2932	2933	2934	2935
5570	2936	2937	2938	2939	2940	2941	2942	2943
5600	2944	2945	2946	2947	2948	2949	2950	2951
5610	2952	2953	2954	2955	2956	2957	2958	2959
5620	2960	2961	2962	2963	2964	2965	2966	2967
5630	2968	2969	2970	2971	2972	2973	2974	2975
5640	2976	2977	2978	2979	2980	2981	2982	2983
5650	2984	2985	2986	2987	2988	2989	2990	2991
5660	2992	2993	2994	2995	2996	2997	2998	2999
5670	3000	3001	3002	3003	3004	3005	3006	3007
5700	3008	3009	3010	3011	3012	3013	3014	3015
5710	3016	3017	3018	3019	3020	3021	3022	3023
5720	3024	3025	3026	3027	3028	3029	3030	3031
5730	3032	3033	3034	3035	3036	3037	3038	3039
5740	3040	3041	3042	3043	3044	3045	3046	3047
5750	3048	3049	3050	3051	3052	3053	3054	3055
5760	3056	3057	3058	3059	3060	3061	3062	3063
5770	3064	3065	3066	3067	3068	3069	3070	3071

OCTAL-DECIMAL INTEGER CONVERSION TABLE (continued)

	0	1	2	3	4	5	6	7
6000	3072	3073	3074	3075	3076	3077	3078	3079
6010	3080	3081	3082	3083	3084	3085	3086	3087
6020	3088	3089	3090	3091	3092	3093	3094	3095
6030	3096	3097	3098	3099	3100	3101	3102	3103
6040	3104	3105	3106	3107	3108	3109	3110	3111
6050	3112	3113	3114	3115	3116	3117	3118	3119
6060	3120	3121	3122	3123	3124	3125	3126	3127
6070	3128	3129	3130	3131	3132	3133	3134	3135
6100	3136	3137	3138	3139	3140	3141	3142	3143
6110	3144	3145	3146	3147	3148	3149	3150	3151
6120	3152	3153	3154	3155	3156	3157	3158	3159
6130	3160	3161	3162	3163	3164	3165	3166	3167
6140	3168	3169	3170	3171	3172	3173	3174	3175
6150	3176	3177	3178	3179	3180	3181	3182	3183
6160	3184	3185	3186	3187	3188	3189	3190	3191
6170	3192	3193	3194	3195	3196	3197	3198	3199
6200	3200	3201	3202	3203	3204	3205	3206	3207
6210	3208	3209	3210	3211	3212	3213	3214	3215
6220	3216	3217	3218	3219	3220	3221	3222	3223
6230	3224	3225	3226	3227	3228	3229	3230	3231
6240	3232	3233	3234	3235	3236	3237	3238	3239
6250	3240	3241	3242	3243	3244	3245	3246	3247
6260	3248	3249	3250	3251	3252	3253	3254	3255
6270	3256	3257	3258	3259	3260	3261	3262	3263
6300	3264	3265	3266	3267	3268	3269	3270	3271
6310	3272	3273	3274	3275	3276	3277	3278	3279
6320	3280	3281	3282	3283	3284	3285	3286	3287
6330	3288	3289	3290	3291	3292	3293	3294	3295
6340	3296	3297	3298	3299	3300	3301	3302	3303
6350	3304	3305	3306	3307	3308	3309	3310	3311
6360	3312	3313	3314	3315	3316	3317	3318	3319
6370	3320	3321	3322	3323	3324	3325	3326	3327

	0	1	2	3	4	5	6	7
6400	3328	3329	3330	3331	3332	3333	3334	3335
6410	3336	3337	3338	3339	3340	3341	3342	3343
6420	3344	3345	3346	3347	3348	3349	3350	3351
6430	3352	3353	3354	3355	3356	3357	3358	3359
6440	3360	3361	3362	3363	3364	3365	3366	3367
6450	3368	3369	3370	3371	3372	3373	3374	3375
6460	3376	3377	3378	3379	3380	3381	3382	3383
6470	3384	3385	3386	3387	3388	3389	3390	3391
6500	3392	3393	3394	3395	3396	3397	3398	3399
6510	3400	3401	3402	3403	3404	3405	3406	3407
6520	3408	3409	3410	3411	3412	3413	3414	3415
6530	3416	3417	3418	3419	3420	3421	3422	3423
6540	3424	3425	3426	3427	3428	3429	3430	3431
6550	3432	3433	3434	3435	3436	3437	3438	3439
6560	3440	3441	3442	3443	3444	3445	3446	3447
6570	3448	3449	3450	3451	3452	3453	3454	3455
6600	3456	3457	3458	3459	3460	3461	3462	3463
6610	3464	3465	3466	3467	3468	3469	3470	3471
6620	3472	3473	3474	3475	3476	3477	3478	3479
6630	3480	3481	3482	3483	3484	3485	3486	3487
6640	3488	3489	3490	3491	3492	3493	3494	3495
6650	3496	3497	3498	3499	3500	3501	3502	3503
6660	3504	3505	3506	3507	3508	3509	3510	3511
6670	3512	3513	3514	3515	3516	3517	3518	3519
6700	3520	3521	3522	3523	3524	3525	3526	3527
6710	3528	3529	3530	3531	3532	3533	3534	3535
6720	3536	3537	3538	3539	3540	3541	3542	3543
6730	3544	3545	3546	3547	3548	3549	3550	3551
6740	3552	3553	3554	3555	3556	3557	3558	3559
6750	3560	3561	3562	3563	3564	3565	3566	3567
6760	3568	3569	3570	3571	3572	3573	3574	3575
6770	3576	3577	3578	3579	3580	3581	3582	3583

6000 to 6777 (Octal)	3072 to 3583 (Decimal)

Octal	Decimal
10000 -	4096
20000 -	8192
30000 -	12288
40000 -	16384
50000 -	20480
60000 -	24576
70000 -	28672

	0	1	2	3	4	5	6	7
7000	3584	3585	3586	3587	3588	3589	3590	3591
7010	3592	3593	3594	3595	3596	3597	3598	3599
7020	3600	3601	3602	3603	3604	3605	3606	3607
7030	3608	3609	3610	3611	3612	3613	3614	3615
7040	3616	3617	3618	3619	3620	3621	3622	3623
7050	3624	3625	3626	3627	3628	3629	3630	3631
7060	3632	3633	3634	3635	3636	3637	3638	3639
7070	3640	3641	3642	3643	3644	3645	3646	3647
7100	3648	3649	3650	3651	3652	3653	3654	3655
7110	3656	3657	3658	3659	3660	3661	3662	3663
7120	3664	3665	3666	3667	3668	3669	3670	3671
7130	3672	3673	3674	3675	3676	3677	3678	3679
7140	3680	3681	3682	3683	3684	3685	3686	3687
7150	3688	3689	3690	3691	3692	3693	3694	3695
7160	3696	3697	3698	3699	3700	3701	3702	3703
7170	3704	3705	3706	3707	3708	3709	3710	3711
7200	3712	3713	3714	3715	3716	3717	3718	3719
7210	3720	3721	3722	3723	3724	3725	3726	3727
7220	3728	3729	3730	3731	3732	3733	3734	3735
7230	3736	3737	3738	3739	3740	3741	3742	3743
7240	3744	3745	3746	3747	3748	3749	3750	3751
7250	3752	3753	3754	3755	3756	3757	3758	3759
7260	3760	3761	3762	3763	3764	3765	3766	3767
7270	3768	3769	3770	3771	3772	3773	3774	3775
7300	3776	3777	3778	3779	3780	3781	3782	3783
7310	3784	3785	3786	3787	3788	3789	3790	3791
7320	3792	3793	3794	3795	3796	3797	3798	3799
7330	3800	3801	3802	3803	3804	3805	3806	3807
7340	3808	3809	3810	3811	3812	3813	3814	3815
7350	3816	3817	3818	3819	3820	3821	3822	3823
7360	3824	3825	3826	3827	3828	3829	3830	3831
7370	3832	3833	3834	3835	3836	3837	3838	3839

	0	1	2	3	4	5	6	7
7400	3840	3841	3842	3843	3844	3845	3846	3847
7410	3848	3849	3850	3851	3852	3853	3854	3855
7420	3856	3857	3858	3859	3860	3861	3862	3863
7430	3864	3865	3866	3867	3868	3869	3870	3871
7440	3872	3873	3874	3875	3876	3877	3878	3879
7450	3880	3881	3882	3883	3884	3885	3886	3887
7460	3888	3889	3890	3891	3892	3893	3894	3895
7470	3896	3897	3898	3899	3900	3901	3902	3903
7500	3904	3905	3906	3907	3908	3909	3910	3911
7510	3912	3913	3914	3915	3916	3917	3918	3919
7520	3920	3921	3922	3923	3924	3925	3926	3927
7530	3928	3929	3930	3931	3932	3933	3934	3935
7540	3936	3937	3938	3939	3940	3941	3942	3943
7550	3944	3945	3946	3947	3948	3949	3950	3951
7560	3952	3953	3954	3955	3956	3957	3958	3959
7570	3960	3961	3962	3963	3964	3965	3966	3967
7600	3968	3969	3970	3971	3972	3973	3974	3975
7610	3976	3977	3978	3979	3980	3981	3982	3983
7620	3984	3985	3986	3987	3988	3989	3990	3991
7630	3992	3993	3994	3995	3996	3997	3998	3999
7640	4000	4001	4002	4003	4004	4005	4006	4007
7650	4008	4009	4010	4011	4012	4013	4014	4015
7660	4016	4017	4018	4019	4020	4021	1022	4023
7670	4024	4025	4026	4027	4028	4029	4030	4031
7700	4032	4033	4034	4035	4036	4037	4038	4039
7710	4040	4041	4042	4043	4044	4045	4046	4047
7720	4048	4049	4050	4051	4052	4053	4054	4055
7730	4056	4057	4058	4059	4060	4061	4062	4063
7740	4064	4065	4066	4067	4068	4069	4070	4071
7750	4072	4073	4074	4075	4076	4077	4078	4079
7760	4080	4081	4082	4083	4084	4085	4086	4087
7770	4088	4089	4090	4091	4092	4093	4094	4095

7000 to 7777 (Octal)	3584 to 4095 (Decimal)

OCTAL-DECIMAL FRACTION CONVERSION TABLE

Octal	Decimal	Octal	Decimal	Octal	Decimal	Octal	Decimal
.000	.000000	.100	.125000	.200	.250000	.300	.375000
.001	.001953	.101	.126953	.201	.251953	.301	.376953
.002	.003906	.102	.128906	.202	.253906	.302	.378906
.003	.005859	.103	.130859	.203	.255859	.303	.380859
.004	.007812	.104	.132812	.204	.257812	.304	.382812
.005	.009765	.105	.134765	.205	.259765	.305	.384765
.006	.011718	.106	.136718	.206	.261718	.306	.386718
.007	.013671	.107	.138671	.207	.263671	.307	.388671
.010	.015625	.110	.140625	.210	.265625	.310	.390625
.011	.017578	.111	.142578	.211	.267578	.311	.392578
.012	.019531	.112	.144531	.212	.269531	.312	.394531
.013	.021484	.113	.146484	.213	.271484	.313	.396484
.014	.023437	.114	.148437	.214	.273437	.314	.398437
.015	.025390	.115	.150390	.215	.275390	.315	.400390
.016	.027343	.116	.152343	.216	.277343	.316	.402343
.017	.029296	.117	.154296	.217	.279296	.317	.404296
.020	.031250	.120	.156250	.220	.281250	.320	.406250
.021	.033203	.121	.158203	.221	.283203	.321	.408203
.022	.035156	.122	.160156	.222	.285156	.322	.410156
.023	.037109	.123	.162109	.223	.287109	.323	.412109
.024	.039062	.124	.164062	.224	.289062	.324	.414062
.025	.041015	.125	.166015	.225	.291015	.325	.416015
.026	.042968	.126	.167968	.226	.292968	.326	.417968
.027	.044921	.127	.169921	.227	.294921	.327	.419921
.030	.046875	.130	.171875	.230	.296875	.330	.421875
.031	.048828	.131	.173828	.231	.298828	.331	.423828
.032	.050781	.132	.175781	.232	.300781	.332	.425781
.033	.052734	.133	.177734	.233	.302734	.333	.427734
.034	.054687	.134	.179687	.234	.304687	.334	.429687
.035	.056640	.135	.181640	.235	.306640	.335	.431640
.036	.058593	.136	.183593	.236	.308593	.336	.433593
.037	.060546	.137	.185546	.237	.310546	.337	.435546
.040	.062500	.140	.187500	.240	.312500	.340	.437500
.041	.064453	.141	.189453	.241	.314453	.341	.439453
.042	.066406	.142	.191406	.242	.316406	.342	.441406
.043	.068359	.143	.193359	.243	.318359	.343	.443359
.044	.070312	.144	.195312	.244	.320312	.344	.445312
.045	.072265	.145	.197265	.245	.322265	.345	.447265
.046	.074218	.146	.199218	.246	.324218	.346	.449218
.047	.076171	.147	.201171	.247	.326171	.347	.451171
.050	.078125	.150	.203125	.250	.328125	.350	.453125
.051	.080078	.151	.205078	.251	.330078	.351	.455078
.052	.082031	.152	.207031	.252	.332031	.352	.457031
.053	.083984	.153	.208984	.253	.333984	.353	.458984
.054	.085937	.154	.210937	.254	.335937	.354	.460937
.055	.087890	.155	.212890	.255	.337890	.355	.462890
.056	.089843	.156	.214843	.256	.339843	.356	.464843
.057	.091796	.157	.216796	.257	.341796	.357	.466796
.060	.093750	.160	.218750	.260	.343750	.360	.468750
.061	.095703	.161	.220703	.261	.345703	.361	.470703
.062	.097656	.162	.222656	.262	.347656	.362	.472656
.063	.099609	.163	.224609	.263	.349609	.363	.474609
.064	.101562	.164	.226562	.264	.351562	.364	.476562
.065	.103515	.165	.228515	.265	.353515	.365	.478515
.066	.105468	.166	.230468	.266	.355468	.366	.480468
.067	.107421	.167	.232421	.267	.357421	.367	.482421
.070	.109375	.170	.234375	.270	.359375	.370	.484375
.071	.111328	.171	.236328	.271	.361328	.371	.486328
.072	.113281	.172	.238281	.272	.363281	.372	.488281
.073	.115234	.173	.240234	.273	.365234	.373	.490234
.074	.117187	.174	.242187	.274	.367187	.374	.492187
.075	.119140	.175	.244140	.275	.369140	.375	.494140
.076	.121093	.176	.246093	.276	.371093	.376	.496093
.077	.123046	.177	.248046	.277	.373046	.377	.498046

OCTAL-DECIMAL FRACTION CONVERSION TABLE (continued)

Octal	Decimal	Octal	Decimal	Octal	Decimal	Octal	Decimal
.000000	.000000	.000100	.000244	.000200	.000488	.000300	.000732
.000001	.000003	.000101	.000247	.000201	.000492	.000301	.000736
.000002	.000007	.000102	.000251	.000202	.000495	.000302	.000740
.000003	.000011	.000103	.000255	.000203	.000499	.000303	.000743
.000004	.000015	.000104	.000259	.000204	.000503	.000304	.000747
.000005	.000019	.000105	.000263	.000205	.000507	.000305	.000751
.000006	.000022	.000106	.000267	.000206	.000511	.000306	.000755
.000007	.000026	.000107	.000270	.000207	.000514	.000307	.000759
.000010	.000030	.000110	.000274	.000210	.000518	.000310	.000762
.000011	.000034	.000111	.000278	.000211	.000522	.000311	.000766
.000012	.000038	.000112	.000282	.000212	.000526	.000312	.000770
.000013	.000041	.000113	.000286	.000213	.000530	.000313	.000774
.000014	.000045	.000114	.000289	.000214	.000534	.000314	.000778
.000015	.000049	.000115	.000293	.000215	.000537	.000315	.000782
.000016	.000053	.000116	.000297	.000216	.000541	.000316	.000785
.000017	.000057	.000117	.000301	.000217	.000545	.000317	.000789
.000020	.000061	.000120	.000305	.000220	.000549	.000320	.000793
.000021	.000064	.000121	.000308	.000221	.000553	.000321	.000797
.000022	.000068	.000122	.000312	.000222	.000556	.000322	.000801
.000023	.000072	.000123	.000316	.000223	.000560	.000323	.000805
.000024	.000076	.000124	.000320	.000224	.000564	.000324	.000808
.000025	.000080	.000125	.000324	.000225	.000568	.000325	.000812
.000026	.000083	.000126	.000328	.000226	.000572	.000326	.000816
.000027	.000087	.000127	.000331	.000227	.000576	.000327	.000820
.000030	.000091	.000130	.000335	.000230	.000579	.000330	.000823
.000031	.000095	.000131	.000339	.000231	.000583	.000331	.000827
.000032	.000099	.000132	.000343	.000232	.000587	.000332	.000831
.000033	.000102	.000133	.000347	.000233	.000591	.000333	.000835
.000034	.000106	.000134	.000350	.000234	.000595	.000334	.000839
.000035	.000110	.000135	.000354	.000235	.000598	.000335	.000843
.000036	.000114	.000136	.000358	.000236	.000602	.000336	.000846
.000037	.000118	.000137	.000362	.000237	.000606	.000337	.000850
.000040	.000122	.000140	.000366	.000240	.000610	.000340	.000854
.000041	.000125	.000141	.000370	.000241	.000614	.000341	.000858
.000042	.000129	.000142	.000373	.000242	.000617	.000342	.000862
.000043	.000133	.000143	.000377	.000243	.000621	.000343	.000865
.000044	.000137	.000144	.000381	.000244	.000625	.000344	.000869
.000045	.000141	.000145	.000385	.000245	.000629	.000345	.000873
.000046	.000144	.000146	.000389	.000246	.000633	.000346	.000877
.000047	.000148	.000147	.000392	.000247	.000637	.000347	.000881
.000050	.000152	.000150	.000396	.000250	.000640	.000350	.000885
.000051	.000156	.000151	.000400	.000251	.000644	.000351	.000888
.000052	.000160	.000152	.000404	.000252	.000648	.000352	.000892
.000053	.000164	.000153	.000408	.000253	.000652	.000353	.000896
.000054	.000167	.000154	.000411	.000254	.000656	.000354	.000900
.000055	.000171	.000155	.000415	.000255	.000659	.000355	.000904
.000056	.000175	.000156	.000419	.000256	.000663	.000356	.000907
.000057	.000179	.000157	.000423	.000257	.000667	.000357	.000911
.000060	.000183	.000160	.000427	.000260	.000671	.000360	.000915
.000061	.000186	.000161	.000431	.000261	.000675	.000361	.000919
.000062	.000190	.000162	.000434	.000262	.000679	.000362	.000923
.000063	.000194	.000163	.000438	.000263	.000682	.000363	.000926
.000064	.000198	.000164	.000442	.000264	.000686	.000364	.000930
.000065	.000202	.000165	.000446	.000265	.000690	.000365	.000934
.000066	.000205	.000166	.000450	.000266	.000694	.000366	.000938
.000067	.000209	.000167	.000453	.000267	.000698	.000367	.000942
.000070	.000213	.000170	.000457	.000270	.000701	.000370	.000946
.000071	.000217	.000171	.000461	.000271	.000705	.000371	.000949
.000072	.000221	.000172	.000465	.000272	.000709	.000372	.000953
.000073	.000225	.000173	.000469	.000273	.000713	.000373	.000957
.000074	.000228	.000174	.000473	.000274	.000717	.000374	.000961
.000075	.000232	.000175	.000476	.000275	.000720	.000375	.000965
.000076	.000236	.000176	.000480	.000276	.000724	.000376	.000968
.000077	.000240	.000177	.000484	.000277	.000728	.000377	.000972

OCTAL-DECIMAL FRACTION CONVERSION TABLE (continued)

Octal	Decimal	Octal	Decimal	Octal	Decimal	Octal	Decimal
.000400	.000976	.000500	.001220	.000600	.001464	.000700	.001708
.000401	.000980	.000501	.001224	.000601	.001468	.000701	.001712
.000402	.000984	.000502	.001228	.000602	.001472	.000702	.001716
.000403	.000988	.000503	.001232	.000603	.001476	.000703	.001720
.000404	.000991	.000504	.001235	.000604	.001480	.000704	.001724
.000405	.000995	.000505	.001239	.000605	.001483	.000705	.001728
.000406	.000999	.000506	.001243	.000606	.001487	.000706	.001731
.000407	.001003	.000507	.001247	.000607	.001491	.000707	.001735
.000410	.001007	.000510	.001251	.000610	.001495	.000710	.001739
.000411	.001010	.000511	.001255	.000611	.001499	.000711	.001743
.000412	.001014	.000512	.001258	.000612	.001502	.000712	.001747
.000413	.001018	.000513	.001262	.000613	.001506	.000713	.001750
.000414	.001022	.000514	.001266	.000614	.001510	.000714	.001754
.000415	.001026	.000515	.001270	.000615	.001514	.000715	.001758
.000416	.001029	.000516	.001274	.000616	.001518	.000716	.001762
.000417	.001033	.000517	.001277	.000617	.001522	.000717	.001766
.000420	.001037	.000520	.001281	.000620	.001525	.000720	.001770
.000421	.001041	.000521	.001285	.000621	.001529	.000721	.001773
.000422	.001045	.000522	.001289	.000622	.001533	.000722	.001777
.000423	.001049	.000523	.001293	.000623	.001537	.000723	.001781
.000424	.001052	.000524	.001296	.000624	.001541	.000724	.001785
.000425	.001056	.000525	.001300	.000625	.001544	.000725	.001789
.000426	.001060	.000526	.001304	.000626	.001548	.000726	.001792
.000427	.001064	.000527	.001308	.000627	.001552	.000727	.001796
.000430	.001068	.000530	.001312	.000630	.001556	.000730	.001800
.000431	.001071	.000531	.001316	.000631	.001560	.000731	.001804
.000432	.001075	.000532	.001319	.000632	.001564	.000732	.001808
.000433	.001079	.000533	.001323	.000633	.001567	.000733	.001811
.000434	.001083	.000534	.001327	.000634	.001571	.000734	.001815
.000435	.001087	.000535	.001331	.000635	.001575	.000735	.001819
.000436	.001091	.000536	.001335	.000636	.001579	.000736	.001823
.000437	.001094	.000537	.001338	.000637	.001583	.000737	.001827
.000440	.001098	.000540	.001342	.000640	.001586	.000740	.001831
.000441	.001102	.000541	.001346	.000641	.001590	.000741	.001834
.000442	.001106	.000542	.001350	.000642	.001594	.000742	.001838
.000443	.001110	.000543	.001354	.000643	.001598	.000743	.001842
.000444	.001113	.000544	.001358	.000644	.001602	.000744	.001846
.000445	.001117	.000545	.001361	.000645	.001605	.000745	.001850
.000446	.001121	.000546	.001365	.000646	.001609	.000746	.001853
.000447	.001125	.000547	.001369	.000647	.001613	.000747	.001857
.000450	.001129	.000550	.001373	.000650	.001617	.000750	.001861
.000451	.001132	.000551	.001377	.000651	.001621	.000751	.001865
.000452	.001136	.000552	.001380	.000652	.001625	.000752	.001869
.000453	.001140	.000553	.001384	.000653	.001628	.000753	.001873
.000454	.001144	.000554	.001388	.000654	.001632	.000754	.001876
.000455	.001148	.000555	.001392	.000655	.001636	.000755	.001880
.000456	.001152	.000556	.001396	.000656	.001640	.000756	.001884
.000457	.001155	.000557	.001399	.000657	.001644	.000757	.001888
.000460	.001159	.000560	.001403	.000660	.001647	.000760	.001892
.000461	.001163	.000561	.001407	.000661	.001651	.000761	.001895
.000462	.001167	.000562	.001411	.000662	.001655	.000762	.001899
.000463	.001171	.000563	.001415	.000663	.001659	.000763	.001903
.000464	.001174	.000564	.001419	000664	.001663	.000764	.001907
.000465	.001178	.000565	.001422	.000665	.001667	.000765	.001911
.000466	.001182	.000566	.001426	.000666	.001670	.000766	.001914
.000467	.001186	.000567	.001430	.000667	.001674	.000767	.001918
.000470	.001190	.000570	.001434	.000670	.001678	.000770	.001922
.000471	.001194	.000571	.001438	.000671	.001682	.000771	.001926
.000472	.001197	.000572	.001441	.000672	.001686	.000772	.001930
.000473	.001201	.000573	.001445	.000673	.001689	.000773	.001934
.000474	.001205	.000574	.001449	.000674	.001693	.000774	.001937
.000475	.001209	.000575	.001453	.000675	.001697	.000775	.001941
.000476	.001213	.000576	.001457	.000676	.001701	.000776	.001945
.000477	.001216	.000577	.001461	.000677	.001705	.000777	.001949

APPENDIX 6
PERFORATED-TAPE LOADER SEQUENCES

READIN MODE LOADER

The readin mode (RIM) loader is a minimum length, basic, perforated-tape reader program for the 33 ASR. It is initially stored in memory by manual use of the operator console keys and switches. The loader is permanently stored in 18 locations of page 37.

A perforated tape to be read by the RIM loader must be in RIM format:

Tape Channel 8 7 6 5 4 S 3 2 1	Format
1 0 0 0 0 . 0 0 0	Leader-trailer code
0 1 A1 . A2 0 0 A3 . A4	Absolute address to contain next 4 digits
0 0 X1 . X2 0 0 X3 . X4	Content of previous 4-digit address
0 1 A1 . A2 0 0 A3 . A4	Address
0 0 X1 . X2 0 0 X3 . X4	Content
(Etc.)	(Etc.)
1 0 0 0 0 . 0 0 0	Leader-trailer code

The RIM loader can only be used in conjunction with the 33 ASR reader (not the high-speed perforated-tape reader). Because a tape in RIM format is, in effect, twice as long as it need be, it is suggested that the RIM loader be used only to read the binary loader when using the 33 ASR. (Note that PDP-8 diagnostic program tapes are in RIM format.)

The complete PDP-8 RIM loader (SA = 7756) is as follows:

```
Absolute  Octal
Address   Content         Tag    Instruction I Z   Comments
 7756,    6032            BEG,     KCC             /CLEAR AC AND FLAG
 7757,    6031                     KSF             /SKIP IF FLAG = 1
 7760,    5357                     JMP .-1         /LOOKING FOR CHARACTER
 7761,    6036                     KRB             /READ BUFFER
 7762,    7106                     CLL RTL
 7763,    7006                     RTL             /CHANNEL 8 IN AC0
 7764,    7510                     SPA             /CHECKING FOR LEADER
 7765,    5357                     JMP BEG+1       /FOUND LEADER
 7766,    7006                     RTL             /OK, CHANNEL 7 IN LINK
 7767,    6031                     KSF
 7770,    5367                     JMP .-1
 7771,    6034                     KRS
 7772,    7420                     SNL             /READ, DO NOT CLEAR
 7773,    3776                     DCA I TEMP      /CHECKING FOR ADDRESS
 7774,    3376                     DCA TEMP        /STORE CONTENT
 7775,    5356                     JMP BEG         /STORE ADDRESS
 7776,    0              TEMP          0           /NEXT WORD
 7777,    JMP START OF BIN LOADER      0           /TEMP STORAGE
```

Placing the RIM loader in core memory by way of the operator console keys and switches is accomplished as follows:

1. Set the starting address 7756 in the switch register (SR).
2. Press LOAD ADDRESS key.
3. Set the first instruction (6032) in the SR.
4. Press the DEPOSIT key.
5. Set the next instruction (6031) in the SR.
6. Press DEPOSIT key.
7. Repeat steps 5 and 6 until all 16 instructions have been deposited.

To load a tape in RIM format, place the tape in the reader, set the SR to the starting address 7756 of the RIM loader (not of the program being read), press the LOAD ADDRESS key, press the START key, and start the Teletype reader.

Refer to Digital Program Library document Digital-8-1-U for additional information on the Readin Mode Loader program.

BINARY LOADER

The binary loader (BIN) is used to read machine language tapes (in binary format) produced by the program assembly language (PAL). A tape in binary format is about one half the length of the comparable RIM format tape. It can, therefore, be read about twice as fast as a RIM tape and is, for this reason, the more desirable format to use with the 10 cps 33 ASR reader or the Type 750C High Speed Perforated Tape Reader.

The format of a binary tape is as follows:

LEADER: about 2 feet of leader-trailer codes.

BODY: characters representing the absolute, machine language program in easy-to-read binary (or octal) form. The section of tape may contain characters representing instructions (channels 8 and 7 not punched) or origin resettings (channel 8 not punched, channel 7 punched) and is concluded by 2 characters (channels 8 and 7 not punched) that represent a checksum for the entire section.
TRAILER: same as leader

Example of the format of a binary tape:

Tape Channel 8 7 6 5 4 S 3 2 1	Memory Location	Content	Comments
1 0 0 0 0 . 0 0 0			leader-trailer code
0 1 0 0 0 . 0 1 0			
0 0 0 0 0 . 0 0 0		0200	
0 0 1 1 1 . 0 1 0			
0 0 0 0 0 . 0 0 0	0200	CLA	origin setting
0 0 0 0 1 . 0 1 0			
0 0 1 1 1 . 1 1 1	0201	TAD 277	
0 0 0 1 1 . 0 1 0			
0 0 1 1 1 . 1 1 0	0202	DCA 276	
0 0 1 1 1 . 1 0 0			
0 0 0 0 0 . 0 1 0	0203	HLT	
0 1 0 0 0 . 0 1 0			
0 0 1 1 1 . 1 1 1		0277	origin setting
0 0 0 0 0 . 0 0 0			
0 0 1 0 1 . 0 1 1	0277	0053	
0 0 0 0 1 . 0 0 0			
0 0 0 0 0 . 1 1 1		1007	sum check
1 0 0 0 0 . 0 0 0			leader-trailer code

After a BIN tape has been read in, one of the two following conditions exists:

a. No checksum error: halt with AC = 0
b. Checksum error: halt with AC = (computed checksum) — (tape checksum)

Operation of the BIN loader in no way depends upon or uses the RIM loader. To load a tape in BIN format place the tape in the reader, set the SR to 7777 (the starting address of the BIN loader), press the LOAD ADDRESS key, set SR switch 0 up for loading via the Teletype unit or down for loading via the high speed reader, then press the START key, and start the tape reader.

Refer to Digital Program Library document Digital-8-2-U-RIM for additional information on the Binary Loader program.

APPENDIX 7
PROGRAMMING SYSTEM

FEATURED PROGRAMS

The programming system for the PDP-8 consists of the MACRO-8 Symbolic Assembler, FORTRAN System compiler, Symbolic On-Line Debugging Program, Symbolic Tape Editor, Floating Point Package, mathematical function subroutines, and utility and maintenance programs. All operate with the basic computer. The programming system was designed to simplify and accelerate the process of learning to program. At the same time, experienced programmers will find that it incorporates many advanced features. The system is intended to make immediately available to each user the full, general-purpose data processing capability of the PDP-8 and to serve as the operating nucleus for a growing library of programs and routines to be made available to all installations. New techniques, routines, and programs are constantly being developed, field-tested, and documented in the Digital Program Library for incorporation in users' systems.

MACRO-8 Symbolic Assembler

The use of an assembly program has become standard practice in programming digital computers. This process allows the programmer to code his instructions in a symbolic language, one he can work with more conveniently than the 12-bit binary numbers which actually operate the computer. The assembly program then translates the symbolic language program into its machine code equivalent. The advantages are significant: the symbolic language is more meaningful and convenient to a programmer than a numeric code; instructions or data can be referred to by symbolic names without concern for, or even knowledge of, their actual addresses in core memory; decimal and alphabetical data can be expressed in a form more convenient than binary numbers; programs can be altered without extensive changes; and debugging is considerably simplified.

The MACRO-8 Symbolic Assembler accepts source programs written in the symbolic language and converts core memory locations, computer instructions, and operand addresses from the symbolic to the binary form. It produces an object program tape, a symbol table defining memory allocations, and useful diagnostic messages.

FORTRAN System Compiler

The FORTRAN (for FORmula TRANslation) System compiler for the PDP-8 lets the user express the problem he is trying to solve in a mixture of English words and mathematical statements that is close to the language of mathematics and is also intelligible to the computer. In addition to reducing the time needed for program preparation, the compiler enables users with little or no knowledge of the computer's organization and operating language to write effective programs for it. The FORTRAN Compiler contains the instructions the computer requires to perform the clerical work of translating the FORTRAN version of the problem statement into an object program in machine language. It also produces diagnostic messages. After compilation, the object program, the operating system and the data it will work with, are loaded into the computer for solution of the problem.

The FORTRAN language consists of four general types of statements: arithmetic, logic, control, and input/output. FORTRAN functions include addition, subtraction, multiplication, division, sine, cosine, arctangent, square root, natural logarithm, and exponential.

Symbolic On-Line Debugging Program

On-line debugging with DDT-8 gives the user dynamic printed program status information. It gives him close control over program execution, preventing errors ("bugs") from destroying other portions of his program. He can monitor the execution of single instructions or subsections, change instructions or data in any format, and output a corrected program at the end of the debugging session.

Using the standard Teletype keyboard/reader and teleprinter/punch, the user can communicate conveniently with the PDP-8 in the symbols of his source language. He can control the execution of any portion of his object program by inserting breaks, or traps, in it. When the computer reaches a break, it transfers control of the object program to DDT. The user can then examine and modify the content of individual core memory registers to correct and improve his object program.

Symbolic Tape Editor

The Symbolic Tape Editor program is used to edit, correct, and update symbolic program tapes using the PDP-8 and the Teletype unit. With the editor in core memory, the user reads in portions of his symbolic tape, removes, changes, or adds instructions or operands, and gets back a complete new symbolic tape with errors removed. He can work through the program instruction by instruction, spot-check it, or concentrate on new sections.

Floating Point Package

The Floating Point Package permits the PDP-8 to perform arithmetic operations that many other computers can perform only after the addition of costly optional hardware. Floating point operations automatically align the binary points of operands, retaining the maximum precision available by discarding leading zeros. In addition to increasing accuracy, floating point operations relieve the programmer of scaling problems common in fixed point operations. This is of particular advantage to the inexperienced programmer.

Mathematical Function Routines

The programming system also includes a set of mathematical function routines to perform the following operations in both single and double precision: addition, subtraction, multiplication, division, square root, sine, cosine, arctangent, natural logarithm, and exponential.

Utility and Maintenance Programs

PDP-8 utility programs provide printouts or punchouts of core memory content in octal, decimal, or binary form, as specified by the user. Subroutines are provided for octal or decimal data transfer and binary-to-decimal, decimal-to-binary, and Teletype tape conversion.

A complete set of standard diagnostic programs is provided to simplify and expedite system maintenance. Program descriptions and manuals permit the user to effectively test the operation of the computer for proper core memory functioning and proper execution of instructions. In addition, diagnostic programs to check the performance of standard and optional peripheral devices are provided with the devices.

ABSTRACTS OF PROGRAMS

The PDP-8 is delivered to the user complete with an extensive selection of system programs and routines making the full data processing capability of the new computer immediately available to each user, eliminating many commonly experienced initial programming delays.

The programs described in these abstracts come from two sources, past programming effort on the PDP-5 computer, and present and continuing programming effort on the PDP-8. Thus the PDP-8 programming system takes advantage of the many man-years of program development and field testing by PDP-5 users.

Although in many cases PDP-8 programs originated as PDP-5 programs, all utility and functional program documentation is issued in a new, recursive format introduced with the PDP-8. Programs written by users of either the PDP-5 or the PDP-8 and submitted to the DECUS library (DECUS - Digital Equipment Corporation Users' Society) are immediately available to PDP-8 users. Consequently, users of either computer can take advantage of the continuing program developments for the other.

System Programs

Digital-8-1-S

Symbolic Editor

The Symbolic Editor program is used to generate, edit, correct, and update symbolic program tapes using the tape teleprinter. With the Editor in memory, the user reads in portions of his symbolic tape, removes, changes, or adds instructions or operands, and gets back a new, complete, symbolic tape with errors removed. He can work through the program instruction by instruction, spot check it, or concentrate on new sections. The tape can contain either symbolic machine language, FORTRAN source statement, data, or text information. This program is available for use with either the 33ASR reader/punch or the high speed reader/punch.

Digital-8-2-S

FORTRAN System

One-pass FORTRAN compiler and operating system compiles FORTRAN source language statements into an object program tape. The operating system executes the program. This system contains the interpreter, arithmetic function subroutines, and input/output packages.

Digital-8-3-S

PAL III (Program Assembler Language)

Symbolic machine language assembler. Converts programs coded in symbolic machine language to binary machine language. The basic process performed by the Assembler is the substitution of numeric values for symbols, according to associations defined in the symbol table. In addition, the user may request that the Assembler itself assign values to the user's own symbols at assembly time. These symbols are normally used to name memory locations, which may then be referenced by name. An assembly listing may be produced.

Digital-8-4-S

DDT-8

Dynamic Debugging Tape provides a means for on-line program debugging at the symbolic or mnemonic level. By typing commands on the console teleprinter, memory locations can be examined and changed, program tapes can be inserted, selected portions of the program can be run, and the updated program can be punched.

Digital-8-5-S

Floating-Point System

A Basic System
B Interpreter, I/O, I/O Controller
C Interpreter, I/O, Functions
D Interpreter, I/O, I/O Controller, Functions

Includes Floating-Point Interpreter and I/O subsystems. Allows the programmer to code his problem in floating-point machine language.

Floating-point operations automatically align the binary points of operands, retaining the maximum precision available by discarding leading zeros. In addition to increasing accuracy, floating-point operations relieve the programmer of the scaling problems common in fixed-point operations. This system includes elementary function subroutines programmed in floating-point. These subroutines are sine, cosine, square root, logarithm, arctan, and exponential functions. Data being processed in floating-point is maintained in three words of memory (12-bit exponent, 24-bit mantissa). An accuracy of seven decimal places is maintained.

Digital-8-6-S

Symbol Print

Loaded over the FORTRAN Compiler, this program lists the variables used and where they will be located in core. It also indicates the section of core not used by the compiled program and data.

Digital-8-7-S

DECtape Library System

The PDP-8 DECtape Library System is loaded by a 17_{10} instruction bootstrap routine that starts at 7600_8. This loader calls a larger program into the last memory page, whose function is to preserve on tape the contents of memory from 6000_8-7577_8, and then to load the INDEX program and the directory into those same locations. Since the information in this area of memory has been preserved, it can be restored when operations have been completed. The skeleton system tape contains the following programs:

INDEX	Typing this causes the names of all programs currently on file to be typed out.

UPDATE	Allows the user to add a new program to the files. UPDATE queries the operator about the program's name, its starting address, and its location in core memory.
GETSYS	Generates a skeleton library tape on a specified DECtape unit.
DELETE	Causes a named file to be deleted from the tape.

Starting with the skeleton library tape, the user can build up a complete file of his active programs and continuously update it.

Digital-8-8-S

MACRO-8

The MACRO-8 Symbolic Assembler accepts source programs written in symbolic language and translates them into binary form in two passes. MACRO-8 produces an object program tape (binary), a symbol table (for use with DDT), and octal symbolic assembly listing, and useful diagnostic messages. MACRO-8 is compatible with PAL III, and has the following additional features: user-defined macros; double precision integers, floating-point constants, arithmetic and Boolean operators, literals, text facilities, and automatic Link generation.

Digital-8-10-S

CALCULATOR

CALCULATOR is an equation evaluation routine. It differs from FORTRAN in that the function to be evaluated is entered via keyboard and calculated immediately upon termination of entry. Format control is provided so that computed results may be conveniently tabulated. Expressions causing the calling of common function subroutines are included.

Digital-8-11-S

DATAK

The DATAK system permits a complex, program-controlled data acquisition system to be adapted to a particular experimental environment through the use of a sophisticated and concise pseudo code. In addition to data-acquisition applications, DATAK furnishes the experimenter with a means of calibrating transducers and is a powerful aid in troubleshooting a complex data-gathering system. Paper tape output produced is acceptable as FORTRAN input.

Digital-8-12-S

ODT-II

ODT-II (Octal Debugging Tape) aids in debugging a PDP-8 program by facilitating communication with the program being run via the ASR 33 Teletypewriter. ODT-II features include register examinations and modification, control transfer, word searching, octal dumping, and instruction traps.

Digital-8-13-S

One-Dimensional Display and Analysis

The one-dimensional pulse-height analysis program is used to read in and analyze 1024-channel energy spectra data. The program receives and executes commands from the keyboard. These commands start and stop data taking and determine into which data region it goes, displays the data with markers, allows areas of interest on the display screen to be expanded, integrates between markers, writes out data, punches out data, and controls background subtraction.

Digital-8-14-S

Multiparameter Display and Analysis

The two-dimensional pulse-height analysis program is used to read in and analyze two-parameter energy and spectra data. The program receives and executes commands from the keyboard. These commands start and stop data taking, control the displays, and control writing and punching of the data. The displays available are: isometric, vertical and horizontal slicing, differential and integral contours, and "twinkle box." The program is flexible with respect to the dimensions of the data matrix.

Digital-8-15-S

Oceanographic Analysis

This program represents the basic accepted physical oceanography method for the reduction of data concerning depth, temperature, and salinity measurements of the water column.

This program has been designed to allow the field oceanographer a rapid means of immediately calculating Sigma-T, anomaly of specific volume, and sound velocity following a Nansen cast whereby he may examine in detail the results of his endeavor, to determine not only the structure of the environment he has just sampled but also to check the validity of his measurements.

In addition to the above, an interpolation routine is incorporated into the program as well as a depth integration of the anomaly of specific volume.

Digital-8-16-S

Master Tape Duplicator

The tape duplicator for the PDP-8 is a single-buffered read and punch program, utilizing the program interrupt. It computes a character count and checksum for each tape and compares with checks at the end of the tape.

Digital-8-35-S

A 680 5-Bit Character Assembly Subroutines
B 680 8-Bit Character Assembly Subroutines

These subroutines concentrate Teletype data by assembling serial-bit data into 5-bit (8-35-S-A) or 8-bit (8-35-S-B) characters and presenting the user with line number and character data. They also add start and stop bits and transmit characters serially. Full-duplex lines are assumed, but the subroutines will work with half-duplex if the user handles the expected echo.

Elementary Function Routines

Digital-8-9-F

Square Root Subroutine - Single Precision

Forms the square root of a single-precision number. An attempt to take the square root of a negative number will give 0 for a result.

Digital-8-11-F

Signed Multiply Subroutine - Single Precision

Forms a 22-bit signed product from 11-bit signed multiplier and multiplicand.

Digital-8-12-F

Signed Divide Subroutine - Single Precision

This routine divides a signed 11-bit divisor into a signed 23-bit dividend giving a signed 11-bit quotient and a remainder of 11 bits with the sign of the dividend.

Digital-8-13-F

Double-Precision Multiply Subroutine - Signed

This subroutine multiplies a 23-bit signed multiplicand by a 23-bit signed multiplier and returns with a 46-bit signed product.

Digital-8-14-F

Double-Precision Divide Subroutine - Signed

This routine divides a 23-bit signed divisor into a 47-bit signed dividend and returns with a 23-bit signed quotient and a remainder of 23 bits with the sign of the dividend.

Digital-8-16-F

Sine Routine - Double Precision

The Double-Precision Sine Subroutine evaluates the function Sin(X) for $-4 < X < 4$ (X is in radians). The argument is a double-precision word, 2 bits representing the integer part and 21 bits representing the fractional part. The result is a 23-bit signed fraction $-1 < \mathrm{Sin}(X) < 1$.

Digital-8-18-F

Cosine Routine - Double Precision

This subroutine forms the cosine of a double-precision argument (in radians). The input range is $-4 < X < 4$.

Digital-8-20-F

Four-Word Floating-Point Package

This is a basic floating-point package that carries data as three words of mantissa and one word of exponent. Common arithmetic operations are included as well as basic input/output control. No functions are included.

Digital-8-21-F

Signed Multiply (Uses EAE Type 182) Single Precision

This subroutine forms a 22-bit signed product from an 11-bit signed multiplier and multiplicand using the Extended Arithmetic Element Type 182. It occupies less storage and takes less time to execute than its non-EAE counterpart (Digital-8-11-F-Sym), and it has the same calling sequence.

Digital-8-22-F

Signed Divide (Uses EAE Type 182) Single Precision

This subroutine divides a double-precision signed 22-bit dividend by a signed 11-bit divisor, producing a signed 11-bit quotient and a remainder of 11 bits having the sign of the dividend.

It makes use of the Extended Arithmetic Element Type 182 instruction set and occupies less storage and takes less time to execute than its non-EAE counterpart Digital-8-12-F. It has the same calling sequence except that the subroutine name is changed from DIVIDE to SPDIV.

Digital-8-23-F

Signed Multiply (Uses EAE Type 182) Double Precision

This subroutine multiplies a 23-bit, signed 2's complement binary number by a 23-bit signed 2's complement binary number, giving a 46-bit product with two signs on the higher order end. It makes use of the Extended Arithmetic Element Type 182 instruction set and, because of this, occupies less storage and takes less time to execute than its non-EAE counterpart (Digital-8-13-F). Its calling sequence is compatible with the non-EAE version.

Digital-8-25-F

EAE Floating-Point Package

These packages perform the same tasks as the Floating-Point Packages (Digital-8-5-S A, B, C, D) except that certain routines have been speeded up by the use of the Extended Arithmetic Element Type 182.

For a detailed description of PDP-8 floating-point arithmetic and the Interpretive Floating-Point Packages, the reader is referred to Digital-8-5-S.

Utility Programs

Digital-8-0

Format for PDP-8 Program Documentation

With the advent of the PDP-8, Digital Equipment Corporation introduced a new, recursive format for program documentation. This format is used for routines and subroutines, such as utility and functional, but not necessarily for system programs.

This format and its use are described in this document.

Digital-8-1-U

Read-In-Mode Loader

The RIM Loader is a minimum-sized routine for reading and storing the information in Read-In-Mode coded tapes via the ASR 33 Perforated Tape Reader.

Digital-8-2-U

Binary Loader (33ASR, 750, 183 Memory Extension)

The Binary Loader is a short routine for reading and storing the information in binary-coded tapes via the ASR 33 Perforated Tape Reader or by means of the Type 750 High-Speed Perforated Tape Reader.

The Binary Loader will accept tapes prepared by the use of PAL (Program Assembly Language; see Digital-8-3-S) or MACRO-8 (see Digital-8-8-S). Diagnostic messages may be included on tapes produced when using either PAL or MACRO. The Binary Loader will ignore all diagnostic messages.

Digital-8-3-U

DECtape Library System Loader

The use of the DECtape Library System Loader is discussed. Certain conventions with respect to last page storage are established for this loader as well as for the Read-In-Mode and Binary Loaders.

Digital-8-4-U-RIM

Read-In-Mode Punch ASR 33

This program provides a means of punching out the information in selected blocks of core memory as RIM-coded tape via the ASR 33 Perforated Tape Reader.

Digital-8-5-U

Binary Punch 33/75A

This program provides a means of punching out the information in selected blocks of core memory as binary-coded tape via the ASR 33 Perforated Tape Punch or via the High-Speed Punch 75A.

Digital-8-6-U

Octal Memory Dump

This routine reads the console switches to obtain the upper and lower limits of an area of memory, then types on the Teletype an absolute address plus the octal contents of the first four words specified and repeats this until the block is exhausted, at which time the user may repeat the operation.

Digital-8-7-U

Logical Subroutines

Subroutines for performing the logical operations of inclusive and exclusive OR are presented as a package.

Digital-8-8-U

Shift Right, Shift Left Subroutines (Single and Double Precision)

Four basic subroutines, shift right and shift left, each at both single and double precision, are presented as a package.

Digital-8-9-U

Logical Shift Subroutines

Two basic subroutines, shift right at both single and double precision, are presented as a package. The shifts are logical in nature.

Digital-8-10-U

Binary-Coded-Decimal to Binary Conversion Subroutine

This basic subroutine converts unsigned binary-coded-decimal numbers to their equivalent binary values.

Digital-8-11-U

Double Precision BCD-to-Binary by Radix Deflation

This subroutine converts a 6-digit BCD number to its equivalent binary value contained in two computer words.

Digital-8-12-U

Incremental Plotter Subroutine

This subroutine moves the pen of an incremental plotter to a new position along the best straight line. The pen may be raised or lowered during the mction.

Digital-8-14-U

Binary to Binary-Coded-Decimal Conversion

This subroutine provides the basic means of converting binary data to binary-coded-decimal (BCD) data for typeout, magnetic tape recording, etc.

Digital-8-15-U

Binary-to-Binary-Coded-Decimal Conversion (Four Digit)

This subroutine extends the method used in Digital-8-14-U so that binary integers from 0 to 4095 in a single computer word may be converted to four binary-coded-decimal characters packed in two computer words.

Digital-8-17-U

EAE (Type 182) Instruction Set Simulator

This routine permits the automatic multiply-divide hardware option to be simulated on a basic PDP-8.

Digital-8-18-U

Subroutine for Alphanumeric Message Typeout

This is a basic subroutine to type messages packed in computer words. Two 6-bit characters are packed internally in a single word. All ASR 33 codes from 301 to 337 and from 240 to 277 (excepting 243 and 245) can be typed. The typing of line feed (code 212) and carriage return (code 215) are made possible by arbitrarily assigning internal codes of 43 and 45, respectively, to represent these characters, thus preventing the output of ASCII codes 243 (#) and 245 (%).

Digital-8-19-U

Teletype Output Subroutines

A group of subroutines useful in controlling ASR 33 output is presented as a package. Provision is made for simulation of tabulation stops. The distance "tabbed" may be controlled by the user. Characrers whose ASR 33 codes are in the groups 241 through 277, inclusive, and 300 through 337, inclusive, are legal. Space, carriage return then line feed, and tabulation are provided via subroutines.

Digital-8-20-U

Character String Typeout Subroutine

This basic subroutine types messages stored internally as a "string" of coded characters. All ASR 33 characters are legal.

Digital-8-21-U

Symbolic Tape Format Generator

The Format generator allows the user to create PDP-8 symbolic tapes with Formatting. It may be used to condense tapes with spaces by inserting tabs, or merely to align tabs, instructions, and comments.

Digital-8-22-U

Unsigned Decimal Print

This subroutine permits the typeout of the contents of a computer word as a 4-digit, positive, decimal integer.

Digital-8-27-U

DECtape Subroutines

Allows the programmer to read, write, or search DECtape using prewritten and tested subroutines. A series of subroutines which will read or write any number of DECtape blocks, read any number of 129-word blocks as 128 words (or one memory page), or search for any block (used by read and write, or to position the tape). These programs are assembled with the user program and are called by a jump to subroutine instruction. The program interrupt detects the setting of the DECtape (DT) flag, allowing the main program to proceed while the DECtape operation is being completed. A program flag is set when the operation is completed. The program thus effectively allows concurrent operation of several input/output devices with the DECtape.

Digital-8-32-U

Binary Punch (6 Channel)

This program provides a means of punching out the information in selected blocks of core memory as binary-coded tape via the 6-channel high-speed punch.

Digital-8-33-U

5/8 TOG (DECtape Formatter)

This program is designed to write timing tracks, mark tracks, and block numbers onto a reel of DECtape providing the tape with the basic skeletal format necessary for its inclusion in any programmed DECtape system. The Formatter program also performs preliminary read-data and write-data checks to assure the user that the tape produced can be reliably included in such an environment.

Digital-8-34-U

DECEX DECtape Exerciser

This program provides complete certification of the DECtape format produced.

Maintenance Programs

Maindec 801-1

PDP-8 Instruction Test Part 1

This program is a minimal test of memory reference instructions, operate instructions, interrupt mode, and the keyboard printer. This test should be used when the state of the processor prevents read-in of more advanced diagnostic programs. It is simply a "go-no go" test of the instructions and is not intended to be diagnostic.

Maindec 801-2A

PDP-8 Instruction Test Part 2A

This program is a test of memory reference instructions, operate instructions, and interrupt mode. An attempt is made to detect and isolate errors to their most basic faults and to the minimum number of logic cards.

Maindec 801-2B

PDP-8 Instruction Test Part 2B

This program is a test of TWOS ADD (TAD) and ROTATE logic, (RAL, RTL, RAR, RTR). Random numbers are used in the TWOS ADD portion of the test and sequential numbers are used in the ROTATE portion. Program control is dependent upon operator manipulation of four switches in the SWITCH REGISTER (bits 0, 1, 2, 3). Error information is normally printed out on the keyboard printer.

Maindec 801-2C

PDP-8 JMS and JMP Test

This program tests the JMP and JMS instructions by doing a JMP and JMS to locations 177-4000. The program also tests the JMS return address for accuracy.

Maindec 801-3A

PDP-8 Instruction Test (EAE Type 182) Part 3A

This program is a test of the Extended Arithmetic Element Type 182. The following instructions are tested: MQL, MQA, SHL, LSR, ASR, NMI, SCA. An attempt is made to detect and isolate errors to their most basic faults and to the minimum number of logic cards. Multiply and divide are tested by Maindec 801-3B.

Maindec 801-3B

PDP-8 Instruction Test (EAE Type 182) Part 3B

Divide overflow detection hardware and divide and multiply hardware are tested by using a pseudo random-number generator to produce the parameters for each test. A software simultated divide and multiply are used to test the results of the hardware divide and multiply.

Maindec 802

Memory Checkerboard Test

Maindec 802 tests memory for core failure on half-selected lines under the worst possible conditions for reading and writing. It is used primarily for testing the operation of memory at marginal voltages.

There are two versions of Maindec 802. The Low End program occupies registers 0003-0111 octal and tests memory from 0112-7777 octal. The High End program occupies registers 7450-7555 octal and tests memory from 0000-7447 octal.

Maindec 803

PDP-8 Memory Address Test

Maindec 803 is designed to provide rough inspection of the performance of the Memory Address register and the decoder network which selects a given memory cell. It is used primarily to detect errors arising from open or shorted selection lines.

Maindec 810

PDP-8 Teletype Reader Test

Maindec 810 tests performance of the Teletype Model 33 Perforated Tape Reader using the reader to scan a closed-loop test tape punched with alternating groups of character codes 000 and 377.

Each character is tested for bits dropped or gained while reading; each group of characters is checked for characters missed entirely or read more than once.

Maindec 811

PDP-8 High Speed Reader Test

This program tests performance of the Type 750 High Speed Perforated Tape Reader and control by scanning a closed-loop test tape for transmission accuracy. The reader control is tested for correct operation with the PDP-8 interrupt system.

Maindec 812

PDP-8 Teletype Punch Test

Maindec 812 punches a test tape in a predetermined pattern. The tape passes directly from the Teletype punch to the Teletype reader, which checks the pattern for accuracy.

Maindec 814

PDP-8 Teleprinter Test

The PDP-8 Teleprinter Test tests performance of the Teletype 33 Keyboard Printer. There are two parts to the test, selectable by the operator. The first part tests keyboard input by immediately causing the character typed to be printed for comparison. The second part tests continuous operation of the teleprinter by causing a line consisting of the ASCII character set to be repeatedly printed. The latter also tests for correct functioning of the interrupt after a character has been printed.

Maindec 817

PDP-8 High Speed Punch Test

This program consists of two separate tests. The first causes the High Speed Punch Type 75E to produce a tape containing a sequence of "pseudo-random" character codes. This tape is checked for accuracy using either the high-speed reader or the Teletype reader.

In the second test, the character code represented by the setting of SR_{4-11} is punched repeatedly. The switch setting may be changed while the test is running.

Maindec 820-1

Extended Memory Control Part 1

This program exercises and tests Extended Memory Type 183 instructions CDF, CIF, RDF, RIF, RMF, and RIB, for proper operation. Basically, this program tests the control section of the Type 183 memory. Data is tested by tests Maindec 802 and Maindec 820-2.

Maindec 820-2

Extended Memory Checkerboard Part 2

Maindec 820-2 is a preliminary test for core memory failures on half-selected lines under worst-case conditions of reading and writing. It is used to test memory module X while running the program in memory module Y.

Maindec 825

680 Static Test

The 680 Static Test verifies correct operation of the 681 and 685 circuits associated with the 680 Data Communications System, in a static state. That is, the program does not actually transmit characters, but tests only the logical operation of the hardware. Hardware malfunctions detected by the program result in a processor halt.

Maindec 826

A 680 8-Bit Character Exerciser

B 680 5-Bit Character Exerciser

The 680 Character Exerciser Program further verifies correct operation of the 680 Data Communications System. This test assumes that the Teletype lines are full duplex. However, if the line outputs are jumpered to the line inputs, the test does verify that the input characters are received as transmitted.

Maindec 827

580 Utility Routines and Compiler

This program is designed to exercise the 580 Tape System. The test routines are called from a small compiler, and are under control of a pseudo language, which may be stored on paper tape for daily maintenance, or typed on-line for debugging and observing malfunctions of the 580 Tape System.

Maindec 827-U

Magnetic Tape Type 580 Utility Routines

These subroutines allow the user to operate the 580 Magnetic Tape System by providing most commands associated with a more sophisticated (hardware) tape system.

Maindec 828

PDP-8 LT08 Teleprinter Test

The LT08 Teleprinter Test verifies correct operation of the LT08 Control Line hardware and any configuration of from one to five teleprinters. Hardware malfunctions detected by the program result in a processor halt. The test includes a Concurrent Output Routine, a Concurrent Input Routine, an Output Scope Loop, and a WRU Test that verifies that none of the teleprinters associated with the LT08 respond to a WRU (who are you) code.

Maindec 829

PDP-8 Memory Power On/Off Test

This program tests memory for bit drop out and pick up after a simulated power failure.

Maindec 830

Type 30G Symbol Generator Exerciser

This program exercises symbol generator logic by using selected character patterns.

Maindec 831

PDP-5/8 DECtape Maintenance Package

The PDP-5/8 DECtape Maintenance Package is a collection of routines designed to be used by maintenance personnel as aids in debugging hardware troubles and as periodic confidence checks on correct operation of the device. Routines are provided to test IOT instructions, delays, control registers, timing, and basic modes of operation. Other routines are included which allow the operator to adjust the device more efficiently and exercise different modes of operation pursuant to "scoping" machine functions. Routines to obtain octal dumps of core memory and routines to write varying bit patterns in core are also available.

Maindec 832

Real-Time Clock Test

This program tests the real-time clock IOT logic, and crystal oscillator specifications.

Maindec 833

Lots of Little Pictures on the Eight

This program contains 12 individual 338 buffered display routines. The routines were selected to enable adjustment and validation of CRT Analog/Digital hardware.

Maindec 834

Type 338 Display PJMP Test

This program is a test of the PJMP instruction. On the 338 display analyses of the Pushdown Pointer, Display Address Counter, Status, Push Jump Destination and Return Addresses are printed upon detection of an error in these areas.

Maindec 835

Type 338 Display POP Test

This program is a test of the POP instruction. On the 338 Display analyses of the Display Address Counter and Pushdown Pointer are printed upon error detection.

Maindec 839

PDP-8 Memory Parity Option

This program is designed to exercise and detect memory parity control and data errors on the PDP-8.

DECUS No. 5-1 and 5-2

Service and Debugging Subroutines for the PDP-5

Two basic subroutine packages for use in communicating with the PDP-5 for service and debugging functions have been written at Bell Telephone Laboratories. The first package, called the Binary Package, is a completely independent one-page subroutine for handling paper tape input-output. The second, called the Octal Package, is a two-page set of subroutines which facilitates exchange of information between the operator and the computer via the Teletype 33ASR unit. As an addition to the Octal Package, there is a symbolic instruction dump subroutine package which occupies three pages of storage.

1. The Binary Package contains a BIN format loader, and provisions for dumping of bracketed locations in BIN or RIM format, punching leader trailer, punching a check sum on BIN tapes, and punching a starting address for self-starting RIM tapes. If entered as a subroutine, the appropriate return can be made. Control of the package is made by the switch register and CONTINUE switch on the PDP-5 control panel.

2. The Octal Package contains provisions for an eight to the line octal dump of bracketed locations, moving of blocks of information in the memory, and loading of memory from the Teletype. Links to the Binary Package and the symbolic dump are also included. Control for this package is through the Teletype and the switch register.

   The symbolic dump portion of the Octal Package provides for a printing on the Teletype of up to four pieces of information for each location of a bracketed group of locations. These include the address, the octal contents of that address, the interpretation as two trimmed Teletype code characters of the octal contents and the symbolic instruction decoding of the contents. Any combination of these pieces of information can be selected through use of the switch register.

DECUS No. 5-3

BRL - A Binary Relocatable Loader with Transfer Vector Options for the PDP-5 Computer

BRL is a binary loader program occupying 4640_8 to 6177_8 registers: also 160 to 177. It has two main functions:

1. It allows a PDP-5 operator to read a suitably prepared binary program into any page location in memory except the registers occupied by BRL. Thus, a program need not be reassembled from symbolic to binary tape whenever it is desired to relocate it.

2. It greatly simplifies the calling of programmed subroutines by allowing the programmer to use an arbitrary subroutine calling sequence when writing his program, instead of having to remember the location of the subroutines. For example, if a programmer wishes to call "Square Root," he does not have to know where Square Root is stored in memory; BRL will instead keep track of this for him. This feature is available to the IBM 7094 programmer and is known as a "transfer vector" option.

DECUS No. 5-4

Octal Typeout of Memory Area with Format Option

Write-up and Listing Only

DECUS No. 5-5

Expanded Adding Machine

Expanded Adding Machine is a minimum-space version of Expensive Adding Machine (DEC-5-43-D) using a table lookup method and including an error space facility.

This is a basic version to which additional control functions can easily be added: Optional vertical or horizontal format, optional storage of intermediate result without reentry, fixed-point output of results within reason, and other features that can be had in little additional space under switch register control.

Write-up and Listing Only

DECUS No. 5-6

BDC to Binary Conversion of 3-Digit Numbers

This program is based on DEC-5-4 and is intended to illustrate the use of alternative models in program construction.

While not the fastest possible, this program has one or two interesting features. It converts any 3-digit BCD-coded decimal number, $D_1D_2D_3$ into binary in the invariant time of 372 microseconds. Efficient use is made of BCD positional logic to work the conversion formula $(10D_1 + D_2)\ 10 + D_3$ by right shifts in the accumulator. In special situations, it could be profitable to insert and initial test/exit on zero, adding 12 microseconds to the time for non-zero numbers.

Write-up and Listing Only

DECUS No. 5/8-7

Decimal to Binary Conversion by Radix Deflation on PDP-8

Write-up and Listing Only

DECUS No. 5-8

PDP-5 Floating Point Routines

Consists of:

1. Square Root - Tape and Symbolic Listing
2. Sine-Cosine - Tape
3. Exponential - Tape

DECUS No. 5/8-9

Analysis of Variance PDP-5/8

The analysis of variance program was written for the standard PDP-5/8 configuration (i.e. 4K memory, ASR33 teletype). The output consists of:

A. For each sample:
1. Sample number
2. Sample size
3. Sample mean
4. Sample variance
5. Sample standard deviation

B. The grand mean

C. Analysis of Variance Table:
1. The grand mean.
2. The weighted sum of squares of class means about the grand mean.
3. The degrees of freedom between samples.
4. The variance between samples.
5. The pooled sum of squares of individual value about the means of their respective classes.

6. The degrees of freedom within samples.
7. The variance within samples.
8. The total sum of squares of deviations from the grand mean.
9. The degrees of freedom.
10. The total variance.
11. The ratio of the variance between samples to the variance with samples.

This is the standard analysis of variance table that can be used with the F test to determine the significance, if any, of the differences between sample means. The output is also useful as a first description of the data.

All arithmetic calculations are carried out by the Floating Point Interpretive Package (Digital-8-5-S).

DECUS No. 5-10

Paper Tape Reader Test

A test tape can be produced and will be continuously read as an endless tape. Five kinds of errors will be detected and printed out. The Read routine is in 6033-6040. Specifications: Binary with Parity Format - Length: registers in locations (octal): 10, 11, 40 through 67 (save 63, 64), and 6000-7777.

DECUS No. 5-11

PDP-5 Debug System

Purpose of this program is to provide a system capable of:

1. Octal dump 1 word per line.
2. Octal dump 10_8 words per line.
3. Modifying memory using the typewriter keyboard.
4. Clearing to zero parts of memory.
5. Setting to HALT codes part of memory.
6. Entering breakpoints into a program.
7. Initiating jumps to any part of memory.
8. Punching leader on tape.
9. Punching memory on tape in RIM format.
10. Punching memory on tape in PARITY format.
11. Load memory from tape in PARITY format.

DECUS No. 5-12

Pack-Punch Processor and Reader for the PDP-5

The processor converts a standard binary-format tape into a more compressed format, with two twelve-bit words contained on every three lines of tape. Checksums are punched at frequent intervals, with each origin setting or at least every 200 words.

The reader, which occupies locations 7421 to 7577 in the memory, will load a program which is punched in the compressed format. A test for checksum error is made for each group of 200 or less words, and the program will halt on detection of an error. Only the most recent group of words will have to be reloaded. Read-in time is about ten percent less than for conventional binary format, but the principal advantage is that little time is lost when a checksum error is detected, no matter how long the tape.

DECUS No. 5-13

PDP-5 Assembler

This program accepts symbolic programs punched on cards and assembles them for the PDP-5. An assembly listing is produced, and a magnetic tape is generated containing the program. This magnetic tape can be converted to paper tape and then read into the PDP-5, or it can be read directly into a PDP-5 with an IBM compatible tape unit. Cards are available.

DECUS No. 5/8-14

Dice Game for the PDP-5, PAL

Binary tape and write-up only. Program uses program interrupt facility.

DECUS No. 5-15

ATEPO (Auto Test in Elementary Programming and Operation of a PDP-5 Computer)

The program will type questions or instructions to be performed by the operator of the PDP-5 (4K) computer. The program will check to see if the operator has followed the instructions or has answered the questions correctly. If this is the case, it will type the next question or instruction.

The program itself uses the locations 200_8 to 500_8, and the messages to be typed are in 600_8 -3410_8. The area 500_8 -577_8 is used to store the RIM and BIN loaders while the programming is running. Page 0 and the locations above 4000_8 are a "work area" in which all the instructions are going to be executed, and in which the operator can make practically all kinds of mistakes, because the contents of the locations in that area are reset after typing any message. The program requires the RIM and BIN loaders to be in locations 7700_8-7777_8 in order to transfer to the "safe area."

After using the program, the loaders can be returned to their original position by just starting the computer in location 377; it will jump to a small subroutine in 3500_8-3515_8 that will make the transfer. The possibility of mistakes that would interfere with the program itself is reduced to practically zero, if the bit 0 is permanently kept (except when answering question 4) in the position 1. With the exception mentioned above, all of the other solutions can be accomplished with the bit 0 in position 1. For that reason, we strongly recommend that a piece of tape, or something similar, be put over the switch corresponding to that bit when being used by an operator without experience, for whom the program was designed.

DECUS No. 5/8-16

Tape Duplicator for the PDP-5/8

The tape duplicator for the PDP-5/8 is a single buffered read and punch program utilizing the program interrupt. It computes a character count and checksum for each tape and compares with checks at the end of the tape.

Checks are also computed and compared during punching. There are three models of operation:

A. SWITCH Ø ON - MAKE MASTER TAPE

B. SWITCH 1 ON - DUPLICATE MASTER TAPE

C. SWITCH 2 ON - VERIFY DUPLICATION

During duplication, the program will notify the operator whether or not more copies can be made without re-reading the master.

Binary and symbolic tapes are available.

DECUS No. 5/8-17

Type 250 Drum Transfer Routine For Use on PDP-5/8

Transfer data from drum to core (Read) or core to drum (Write) via ASR-33 Keyboard Control.

DECUS No. 5-18

Bin Tape Disassembler for the PDP-5*

This program disassembles a PDP-5 program, in Bin format, on punched paper tape. The

*Work performed under the auspices of the U. S. Atomic Energy Commission tape is read by a high-speed reader, but the program may be modified to use the ASR-33 reader. The margin setting, address, octal contents, mnemonic interpretation (PAL), and the effective address are printed on the ASR-33 Teletype.

DECUS No. 5-19

DDT-5-2 Octal-Symbolic Debugging Program

DDT-5-2 is an octal-symbolic debugging program for the PDP-5 which occupies locations 5600 through 7677. It is able to merge a symbol table punched by PAL II and stores symbols, 4 locations per symbol, from 5577 down towards 0000. The mnemonics for the eight basic instructions and various OPR and IOT group instructions are initially defined (see DEC-5-1-S Attachment II, p. 21), and the highest available location for the user is initially 5373.

From the teletype, the user can symbolically examine and modify the contents of any memory location. DDT-5 allows the user to punch a corrected program in BIN format.

DDT-5 has a breakpoint facility to help the user run sections of his program. When this facility is used, the debugger also uses location 0005.

This program has nearly all the features of DDT for the PDP-1. The meaning of the control characters of ODT (DEC-5-5-S) are the same in DDT-5.

DECUS No. 5/8-20

Remote Operator FORTRAN System

Program modifications and instructions to make the FCRTRAN OTS version dated 2/12/65 operated from remote stations.

DECUS No. 5/8-21

Triple Precision Arithmetic Package for the PDP-5 and the PDP-8

This is an arithmetic package to operate on 36-bit signed integers. The operations are add, subtract, multiply, divide, input conversion, and output conversion. Triple precision routines have a higher level of accuracy for work such as accounting. The largest integer which may be represented is $2^{35}-1$ or 10 decimal digits. The routines simulate a 36-bit (3 word) accumulator in core locations 40, 41, and 42 and a 36-bit multiplier quotient register in core locations 43, 44, and 45.

Aside from the few locations in page 0, the routines use less core storage space than the equivalent double-precision routines.

DECUS No. 5/8-22

DECtape Dupl icate

This is a DECtape routine to transfer all of one reel (transport 1) to another (transport 2). This program occupies one page of memory beginning at 7400. The last page of memory is not used during the operation of the program, however, the memory from 1 to 7436 is used to set the DECtape reels in the proper starting attitude and is then destroyed during duplication. Duplication will commence after which both reels will rewind. Parity error will cause the program to halt with 0040 in the accumulator.

DECUS No. 5/8-23

PDP-5/8 Oscilloscope Symbol Generator

The subroutine may be called to write a string of characters, a pair of characters, or a single character on an oscilloscope. Seventy (octal) symbols in ASCII Trimmed Code and four special "format" commands are acceptable to this routine. The program is operated in a fashion similar to the DEC Teletype Output Package.

Binary tape with parity format, PAL binary tape, Assembler listing, and cards for an LRL Assembly are available.

Specifications: 1. BIN with parity format or PAL BIN
2. Length - registers 200-577 (octal)
3. Oscilloscope display unit

DECUS No. 5-24

Vector Input/Edit

This program accepts Teletype and effects editing options by implementing a man-machine dialogue. Development of the program was supported, in part, by the Air Force Office of Scientific Research and the Army Research Office.

DECUS No. 5-25

A Pseudo Random Number Generator for the PDP-5 Computer

The random number generator subroutine, when called repeatedly, will return a sequence of 12-bit numbers which, though deterministic, appears to be drawn from a random sequence uniform over the interval 0000_8 to 7777_8. Successive numbers will be found to be statistically uncorrelated. The sequence will not repeat itself until it has been called over 4 billion times.

The program tape is prefixed with a text for a relocatable loader, used at NYU, but this may be bypassed and the binary section will then read directly into 3000-3077.

DECUS No. 5-26

Compressed Binary Loader (CBL) Package

PDP-5 Installations using an ASR-33 Teletype for reading in binary tapes can save significant time (approximately 25%) by taking advantage of all eight channels of the tape. The CBL loader only occupies locations 7700 through 7777. The tape formatted into individual blocks, each with a checksum.

On detection of an error, the loader halts so the tape may be repositioned in the leader area of the block which caused the error.

PAL II has been modified to punch in CBL format, and a DDT-5-3 (comparable to DDT-5-5, DECUS No. 5-19) has been written.

The following programs are included in the package:

1. CBL Loader
2. CBC Converter (BIN to CBL)
3. CONV Converter (CBL to BIN)
4. PAL IIC (punches CBL format)
5. DDT-5-3 (reads and punches CBL format)

DECUS No. 5/8-27

ERC Boot

The ERC Boot is a bootstrap routine somewhat simpler than the one presently available for the PDP-8. This routine restores the entire last page, consisting of:

1. Clear Memory Routine
2. RIM Loader
3. Modified Binary Loader.

The Clear Memory routine is entered at 7600 (octal). It clears (to 0000) the lower 31 pages of memory, then branches to the Binary Loader.

The modified Binary Loader halts after reading tape with the checksum in the accumulator. If the binary tape is properly terminated, pressing CONTINUE takes a branch to the beginning location of the program. PAL compiled programs may be properly terminated by ending the PAL symbolic tape in the following manner:

```
P
 A
  L
S
 Y
  M
P  B
 R  O
  O  L
   G  I
    R  C
     A
      M
```

```
*START (any named starting address)
$
```

The Binary Loader stores the octal value for START in the location labeled ORIGIN in BIN. The instruction following HLT in BIN is replaced by JMP I ORIGIN (5616), causing a branch to START.

DECUS No. 8-28

PAL III Modifications for PDP-8 and ASR-33

This modification of the PAL III Assembler speeds up assembly on the ASR-33/35 and operates only with this I/O device. The symbolic tape is read only once, on pass 0, and stored in the machine. This pass initiates as does pass 1, except that both switches 0 and 1 must be down. Other passes of the assembly initiate normally, but do not end the symbolic tape.

If the program is too large for the buffer, the Teletype punches 50-200 codes before halting. If the program size is doubtful, it is advisable to leave the punch on during pass 0 and not continue to pass 1.

The user and symbol table starts at 2736 and the buffer begins at 3103, allowing room for 25 user symbols only. The highest location of the buffer is 7440. This leaves a buffer size of 4336_8 or 2270_{10}, which is sufficient for a large program or a small program with many comments, owing one tape character per location. To increase symbol table size, constants at 1413 and 1414 may be adjusted. For instance, if 50 symbols are desired:

```
1413    BUF, 3246
1414    BUFCNT, -4172
```

A few other modifications have been made to aid in circumventing the slow speed of the Teletype. The length of the leader-trailer tape has been shortened; there is no pass III leader punched; and the symbol table has no leader or trailer on either pass, making the symbol tape incompatible with DDT (an appropriate leader can be punched manually). To restore any of these characteristics, the appropriate statements in the modifications tape may be deleted.

Symbolic tape may be modified as above and a new binary tape produced. The binary tape must then read in after loading the assembler for modification. This process can be done each time the assembler is loaded, or the modified assembler can be punched as a complete separate tape.

DECUS No. 8-29

BCD to Binary Conversion Subroutines

These two subroutines improve upon the DEC supplies conversion routine. Comparison cannot be made to the DECUS-supplied fixed-time conversion, DECUS No. 5-6, because it is specified only for the PDP-5. One routine is designed for minimal storage, the other for minimal time. Both are fixed-time conversions; time specified is for a 1.6-μsec machine.

Minimal time routine: 73.6 μsec/32 locations

Minimal storage routine: 85 μsec/29 locations

DEC routine: 64-237 μsec/37 locations

Time for number D_1, D_2, and D_3 is 64 + (D_1 + D_2) 9.6 μsec.

DECUS No. 5-30

GENPLOT - General Plotting Subroutine

This self-contained subroutine is for the PDP-5 with a 4K memory and a CALCOMP incremental plotter. The subroutine can move (with the pen in the up position) to location (x, y), make an "X" at this location, draw a line from this present position to location (x, y), and initialize the program location counters. A binary, relocatable tape is available.

If the subroutine is to be relocatable, the titles are: MOve, PLot, DRaw, and INit. The readin procedure is the same as for other relocatable subroutines.

DECUS No. 5-31

FORPLOT - FORTRAN Plotting Program for PDP-5

FORPLOT is a general-purpose plotting program for the PDP-5 computer in conjunction with the CALCOMP 560 Plotter. It is self-contained and occupies memory locations 0000_8 to 4177_8. FORPLOT accepts decimal data inputted on paper tape in either fixed or floating point formats. Formats can be mixed at will. PDP-5 FORTRAN output tapes are acceptable directly and any comments on these are filtered out.

FORPLOT scales input data. The operator informs the computer, in advance, of the data values to be assigned to the top, bottom, right, and left plot boundaries. There are no restrictions on these data values. It is not necessary that any of them be zero, nor is it necessary that the top and right boundaries correspond to more positive data values than the bottom and left boundaries, respectively.

All plotted graphs are 10 inches in the ordinate direction. The operator controls the length of the abscissa which may reach a maximum of 39.99 inches.

A number of plot format options are available to the user. The program is capable of drawing an abscissa and ordinate axis, each ticked at intervals of 1 inch, at the right and bottom boundaries of the plot, respectively. If the user chooses to omit these axes, a circled point is placed at the starting point to indicate the bottom-right plot boundary. Points are left unconnected unless connected through user control. Finally, at the option of the user, FORPLOT can locate either a small "x" or a small octagon (approximately 1/16 inch across), or a superposition of both at each plotted point to make the point more plainly visible.

A column selection feature is provided enabling the operator to select data columns from tapes containing several of them. In this bulletin "column" refers to a column in the arrangement of the data when it is printed on the ASR-33. If desired, the operator can supply only the ordinates and the program will space the plotted points uniformly in the abscissa direction according to a preset constant.

DECUS No. 5/8-32

Program to Relocate and Pack Programs in Binary Format

This relocation program allows automatic transfer of a program in binary format into any portion of memory, no matter what starting address it has been allocated. Each program is given a fixed starting address, allowing it to be loaded in the normal manner without using the relocation program. This includes moving a program to an entirely different starting address on a different page of memory. The program is now in use at CRNL and is proving a most effective tool in assembling a combination of existing programs and in amending and fault-finding new programs.

This package includes three programs to aid in relocating and retrieving subroutines:

1. A Clear Memory routine which clears registers 200_8 through 3777_8. Length is 12 locations.

2. A programmed display routine which displays the entire memory. Empty registers appear on the base line while occupied addresses appears as 4096 counts on the display. The length is 15 locations.

3. Because it is often necessary to retrieve relocated programs on paper tape, a punch routine is included which punches those areas of the first half of memory which contain the program. The punch program senses and deletes gaps in memory.

   This program saves much time since starting and ending addresses need not be remembered as with ODT and other types of binary punch programs.

   The tape is punched in binary format complete with checksum, and is preceded by and followed with a length of leader. The length is 90 locations.

The relocation program and associated programs are themselves relocatable.

DECUS No. 5/8-33

Tape to Memory Comparator

Tape to Memory Comparator is debugging program which allows comparison of the computer memory with a binary tape. It is particularly useful for detecting reader problems, or during stages of debugging a new program.

A typeout occurs whenever the memory disagrees with the contents of the binary program tape. The typeout consists of the memory location, contents of memory, and contents of the tape on one line. The checksum is typed out as an error at one location greater than the last address on the tape.

Presently, the program uses a high-speed reader; however it may be modified for the TTY reader. The program occupies 165 octal locations on a single page. It does not use page 0 or autoindex registers.

DECUS No. 5-34

Memory Halt - A PDP-5 Program to Store Halt in Most of Memory

With Memory Halt and OPAK, (DECUS No. 5-2.1), in memory, it is possible to store halt (7402) in the following memory locations:

0000	to	0005
0007	to	6177
7402	and	7403

Memory Halt occupies locations 0200 to 0237 with a starting address of 0200. When started, it stores 7402 in locations 0001 to 0005 and locations 7402 and 7403. It then sets up some memory location so that OPAK can store 7402 in locations 0007 to 6177.

Halts in memory are useful when a program transfers control to an area of memory not occupied by the program itself. Upon executing the JMP or JMS instruction, the computer halts. With careful investigation, the programmer can determine why the transfer of control took place.

DECUS No. 5/8-35

Binary Coded Decimal to Binary Conversion Subroutine and Binary to Binary Coded Decimal Subroutine (Double Precision)

This program consists of a pair of relatively simple and straightforward double-precision conversions. They make no claim to speed or brevity. A double entry has been used which is:

```
TAD  HIGH
JMS  BCD BIN
TAD  LOW
JMS  BCD BIN+3
```

DECUS No. 5-36

Octal Memory Revised

The Octal Memory Dump on Teletype is a DEC routine (DEC-5-8-U) which dumps memory by reading the switch register twice; once for a lower limit and again for an upper limit. It then types an address, the contents of the program and the next three locations, issues a CR/LF, then repeats the process for the next four locations. This leaves the right two-thirds of the Teletype page unused. The 78_{10} instructions occupy two pages.

This revised routine uses the complete width of the Teletype page and occupies only one memory page, using less paper and two less instructions. Now an address and the contents of 15 locations are typed out before a carriage return.

Octal Memory Dump Revised has proved its value as a subroutine and/or a self-contained dump program when it is necessary to dump large sections of DECtape, magnetic tape (IBM compatible), or a binary formatted paper tape.

DECUS No. 5-37

Transfer II

For users who have more than one memory bank attached to the PDP-5/8, Transfer II may prove valuable in moving information from one field to another. Often areas designated for loaders are being used for other reasons, only to find the loaders necessary a few minutes later. When debugging, Transfer II enables a programmer to make a few changes in a new program and test it without reading in the original program again, especially if his corrections did not work. In short, Transfer II enables more extensive use of memory banks.

1. IDENTIFICATION

1.1 F-85

1.2 PDP-8 Users Handbook Change Notice

1.3 October 7, 1966

Change Notice for PDP-8 Users Handbook - F-85

Page 4; line 13-14

Change FROM: is read from a memory location in 0.8 microsecond and rewritten in the same location in another 0.8 microsecond of one 1.6-microsecond memory cycle.

TO: is read from a memory location in 0.75 microsecond and rewritten in the same location in another 0.75 microsecond of one 1.5-microsecond memory cycle.

Page 13; Figure 6

Change FROM: Format diagram incorrect as shown.

TO: Format diagram shown as Figure 7 on page 17.

Page 17; Figure 7

Change FROM: Format diagram incorrect as shown.

TO: Format diagram shown as Figure 6 on page 13.

Page 20; line 9

Change FROM: Event Time: 1

TO: Event Time: 2

Page 39; line 24

Change FROM: Octal Code: 6104

TO: Octal Code: 6101

Page 39; line 33

Change FROM: Octal Code: 6102

TO: Octal Code: 6104

Page 45; line 10

Change FROM: 3. Give the MUL command.

TO: 3. Give the MUY command.

Page 68; line 26

Change FROM: Table 2 Analog-to-Digital Converter Type 139E Characteristics

TO: Table 2 Analog-to-Digital Converter Type 138E Characteristics

Page 95; line 5

Add after line 5: Clears the AC after execution.

Page 95; line 15

Add after line 15: Clears the AC after execution.

Page 128; line 24

Change FROM: AC1(0) = Sets the SPACE flip-flop

TO: AC1(0) = Sets the SPACE flip-flop, also enables the interrupt

Page 131; line 27

```
Change FROM:  TMP TW2          /GO TO A WRITE INSTRUCTION

         TO:  JMP TW2          /GO TO A WRITE INSTRUCTION
```

Page 131; line 33

```
Change FROM:  TSZ CNTR         /COUNT THE NUMBER OF

         TO:  ISZ CNTR         /COUNT THE NUMBER OF
```

Page 131; line 37

```
Change FROM:  TSDS             /WAIT FOR LAST

              TSDF             /WAIT FOR LAST
```

Page 198; line 10

```
Change FROM:  Address Accepted  ——▷   PF3S     PF10H     W640

         TO:  Address Accepted  ——▶   PF3S     PF10H     W640
```

Queue of jobs

- I/O Sub bit processing
  Bit

  Display processor –

  memory size

  Microprogram implementation – have pattern
  available to [illegible]
  shell

Think out whole system -- mapping
- multiprocessor (sync methods)
- multicompiler

PDP-X

Description

Communication processor
a job done [illegible]

revision date

5/1/67

5/12/67

Index

PDP-X is a modern, very high performance, third generation computer family designed for the small computer market. Upward and downward program compatibility permits easy system growth and enhances application programming. Standard IO and Memory interfaces are used for all processor models and all perhipheral devices. The architecture lends itself to fourth generation hardware implementation and the development of multiprocessor systems.

The system architecture of the PDP-X computer family is described below. In addition to specifying the organization of the entire family, details of a particular implementation have been included. The major design objective has been significantly increaded performance in order to meet increasingly more sophisticated user demands.

N

1.1 The performance of current products

Although there is no magic formula into which parameters of vastly different machines may be substituted to achieve an absolute measure of performance, the relative performance of past and current DEC small computer central processors may be estimated since their architectures are so closely related. Factors relating to word length, order code, memory speed possible, etc., have been evaluated and are given in figure 1. PDP-6/10 has been somewhat arbitrarily estimated to be an order magnitude more powerful than PDP-7/9. System performance dependence upon available software and optional perhipherals has been specifically omitted. Note that the PDP-8I ~~odes~~ does not appear, its performance is identical to that of the PDP-8; 9+ and 1+ represent versions which include optional multiply/divide and priority interrupt hardware.

The PDP-X, designed to be, initially, a replacement for the PDP-9, has a minimum performance at least equal to the 9+ and possibly several times better. This performance extends upwards with the addition of processor options. Other implementations of the same architecture span markets currently held by PDP-8 and the currently nonexistant 24 bit machine. Selling prices as a criterion for the machine would shift the set of curves for PDP-X left. The PDP-9 replacement has approximately the same amount of hardware and based upon estimates of integrated circuit costs derived from PDP-8I, its manufacturing cost should be half PDP-9. Similarly, the very smallest model, ru~~ff~~nning with a slow memory, should cost less to manufacture than the PDP-8S.

The performance/price ratio of PDP-X to PDP-9 is, conservatively, 3 to 1. Verification of this ratio is difficult without more cost estimation and a considerable programming test. Perhaps the best measure will be the relative performance and size of the Fortran IV compiled programs and the effort required to write the compiler itself. — True. — what are the indications so far.?

how so.

N

are you design[illegible] computers or designing the [illegible] Sales liter[illegible]

1.2 Models

Three basic models are worth singling out of the possible implementations. As shown in figure 1, they cover a performance range from PDP-8 to smaller versions of PDP-10 and may be aimed at, respectively, the PDP-8, PDP-9, and a currently non extant 24 bit processor. The two larger models would compare in performance to the SDS Sigma 2 and Sigma 5.

The smallest processor, the PDP-X / 14, has only the basic instruction set implemented and all its registers are located in main core memory; a typical ADD instruction takes 4 memory cycles at less than a microsecond each. The longer word length, lower price, and superior instruction set make this machine superior to the -8. The processor may also be implemented using an even less expensive memory system to achieve the minimum cost true ^is complement computer. Expansion to a system with hardware general registers is not possible since the flow chart must differ to optimize each processor.

why

The set is minimal, almost like PDP.8. as such an interpretation PDP.8 make the two =.

The medium processor, PDP-X / 16, implements its registers in a fast memory array and is provided with an expanded instruction set. Additional instructions as well as a number of interrupt channels with corresponding general register sets may be optionally installed. As in / 14 the memory width is 16 bits but some double word instructions are implemented.

The largest processor, PDP-X / 32, is an expanded version of the above. Although the word length is still 16 bits, many double word instructions, including floating point, are implemented and the basic memory width is 32 bits to speed instruction processing. The economics of building this machine, especially for markets which do little if any serious arithmetic computation, needs careful scrutiny.

this is a pretty nice machine if the extended order code idea is adhered to and well defined.

Cite larger,
1. faster memorie
2. LSI registe an
3. Perhaps small assoc. stor
4. ROS.

1.3 Design goals

a. Advanced concepts- The system architecture should make superior use of currently available and anticipated technology. In particular, program core storage requirements must be reduced to minimize the relatively expensive memory's contribution to total system cost and the architecture should be amenable to the use of internal scratch pad memories, gate arrays, and other forms of large scale integration.

b. Implementations- The architecture should be implementable in several processor models whose price and performance span the entire small computer market and include a model small enough to use as part of an IO device controller or selector channel. Smooth evolution and reimplementation should be possible over the next several years as the architecture leads to many new models.

c. Software- Although major hardware improvements are possible, even more significant gains can be achieved through further development of software systems. The hardware necessary for dynamic memory allocation/protection, privileged instruction traps, and other features of complex software systems must be imbedded into the basic design. A true real-time compiler, especially one that permits dynamic memory assignment, seems a necessity. There are many special application packages that would make the system far more useful in many new market areas.

d. Standard interfaces- Standard memory and IO interfaces must be shared by all processor models, memories, and perhipherals in order to unify the set of options and to facilitate field expansion of systems.

e. Goals of the implementations- To the normal goal of lowest possible manufactureing costs for any model may be added the requiremnnt of automated production and production test facilities. System selling prices should be reduced by making it possible to produce useful results using less of the more powerful hardware and software.

N

f. Specific IO goals- Control signals available at the IO interface should permit channel control of all basic perhipherals; device hardware requirements should be minimized, any special timing, for example, should be done in the processor IO logic; the system should respond extremely rapidly to interrupts, even those requiring the full processor computational ability; communication with devices physically far from the processor should be possible; the IO bus should be mechanically simple.

g. Specific processor goals- the order code should be as concisely organized as possible, placing as few arbitrary restriction on the program as possible; a single instruction should be able to directly address anywhere in memory as well as call upon an immediate operand; the most common instructions must be available in compressed form to conserve memory requirements; recursive, reëntrant, and pure code should be possible.

h. The system architecture should in no way limit system expansion as a multiprocessor. Future implementations should include a dynamically restructureable multiprocessor which exhibits fail soft features.

i. Multi computer organzation,

1.4 Design Decisions

a The basic word length has been choosen to be 16 rather than 18 bits in order to maintain compatibility with the majority of the newer computers, especially IBM. The byte and character are 8 bits long; a double word consists of 4 bytes; a floating point word, with hexadecimal radix, contains either 4 or 8 data bytes.

b The word, 16 bits of data, has been choosen to be the basic addressable unit although instructions are available which reference bits, bytes, words, doublewords, etc. as data. Doubleword instructions need not fall on doubleword boundaries although double data words must.

exactly what will be the ~~set~~ instrs. (bits, bytes, 2-words, 4-words) then indexing should be done right.

c The basic structure contains multiple accumulators/index registers. The general register structure simplifies the order code and proves greater programming power over more conventional single accumulator organizations. The floating point registers, more of a programming convention than hardware feature on the two smaller processors, are distinct from the general registers. ?

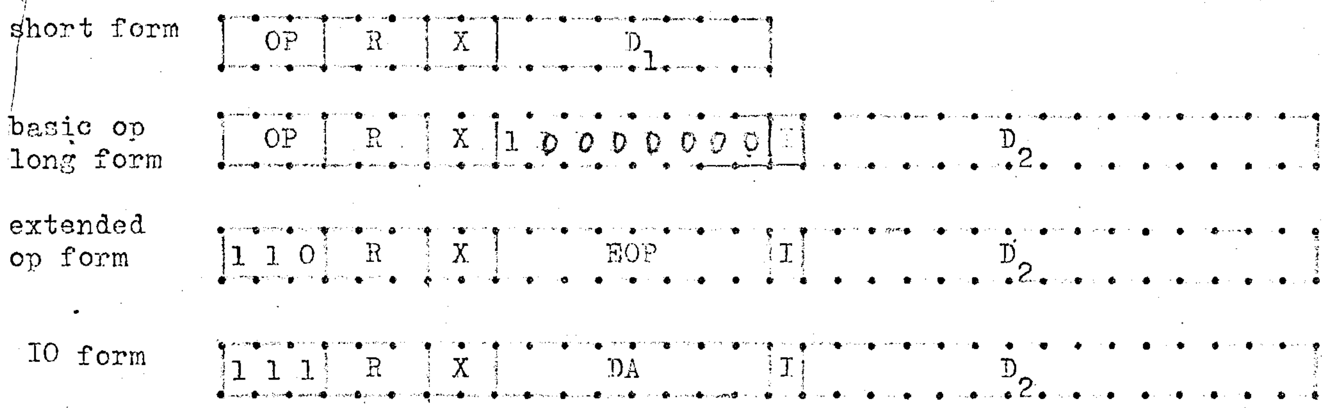
d No base registers are used in addressing, instructions are capable of addressing relatively, indexed, and to page 0 in the short format. A long format permits direct specification of any word anywhere in the entire memory system. The most common instructions are available in short form, all are available in long form.

e The basic unit of IO data is the byte. This unit is natural for paper tape perhipherals, the most common types, as well as the teletype. The bus organization permits the transmission of a full word whenever necessary.

f A priority interrupt system which permits direct device recognition is provided as standard. Separate register sets for the interrupt levels are provided to maximize IO bandwidth.

g A standardized, unified IO structure common to all processors permits both program controlled and channel controlled transfers over the same bus with a minimum of device hardware.

2.1 Instruction Format



mnem	bits	definition
OP	3	basic operation code specifying major instruction class
R	3	general register specification or sub function selection for non accumulator reference instructions
X	2	index register and address mode selector
D_1	8	short form address and immediate operand
D_2	15	long form address
I	1	indirect addressing specification
EOP	8	extended operation code specifying instruction
DA	8	IO device address and bus selection

2.2 Instructions

Instructions may be divided into 2 groups , basic and extended. The basic instructions appear in all models and may either be in long or short format. Extended instructions are implemented in some models, they trap when executed in machines for which no hardware has been provided; all extended instructions are long format only. Instruction class is determined by the 3 Op Code bits (0,1,2) of the instruction word. EOP (extended) instructions are characterized by a 110 pattern in the Op Code and the specific operation in the D_1 bits.

/14	/16	/32
All basic instructions are implemented; the EOP class is uniformly trapped.	All basic and some EOP instructions, the remainder of the EOP class trap.	All EOP and basic class instructions, some EOP class are, by convention, still trapped

Instructions may also be classified by the type of operand they effect. These include:

arithmetic	signed words
logical	unsigned words
floating	floating point double/quadruple words
branch	address pointers
IO	IO system

class	OP	mnem	definition
load	0	L	load selected register (R) from memory; condition code remains unchanged
store	1	ST	store selected register (R) into memory; condition code remains unchanged
and	2	N	and selected register (R) with memory, place result in selected register; condition code 0 remains unchanged 1 set if negative result 2 cleared if zero result, set otherwise
add	3	A	add contents of selected register (R) to memory following the rules of two's complement arithmetic, place result in selected register; condition code bits are first cleared and then set as follows: 0 set if carry out of bit 0 1 set if negative result 2 cleared if zero result, set otherwise
branch	4		general conditional branch and subroutine linkage instruction; R bits specify particular operation. When R=7 the program counter, updated to point to the instruction following the branch, is saved in general register 2. The condition code remains unchanged for all branch instructions.

mnem	R	condition
BCZ	0	branch if condition code bit 0 set
BM	1	1
BN	2	2
B	3	unconditional branch
BNC	4	branch if condition code bit 0 not set
BP	5	1
BZ	6	2
BAL	7	branch and link

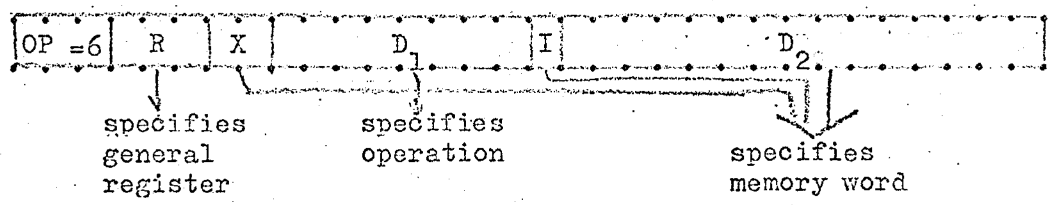
class	OP	mnem	definition
modify	5		general memory modification instruction, R bits specify particular operation. Condition code bit 0 is changed only by the two shift instructions (R=4,5). Condition code bits are set as follows for <u>all</u> modify instructions: bit 1 set if negative result, cleared if positive bit 2 cleared if zero result, set otherwise

R	operation

mnem	R	operation
T	0	no operation but condition code is set to reflect state of the memory word
C	1	logical complement, the memory word is complemented on a bit by bit basis
I	2	increment, one is added to the sepcified memory location
CI	3	arithmetic complement (two's), complement then increment, negate
SR	4	shift right, the memory word and condition code bit 0 are rotated together as a 17 bit register one place to the right, loading condition code bit 0 from bit 15
SL	5	shift left, the memory word and condition code bit 0 are rotated left together as a 17 bit register, loading condition code bit 0 from bit 0 of the memory word
SB	6	swap bytes, the left and right bytes of the memory word are interchanged
CL	7	clear, the memory word is set to all zeros

class	OP	mnem	definition
EOP	6		extended operation code class; forced long format; D_1 bits specify particular operation to be performed by selecting an entry point into a read only memory routine. The effect on the condition code is determined by the particular operation performed.

EOP class instruction doubleword



If the operation specified has not been implemented in the machine a trap occurs as follows:

location	8_{10}	receives the updated program counter
	9	EOP instruction
	10	effective address
	11	contains the entry point into the EOP handler. This word is loaded into the program counter

Since D_1 codes 0 through 31_{10} are never implemented in the machine hardware, some 32_{10} programmed operators are available.

Extended Operation Codes

The opcode is located in D.1. The first 64 of these (codes 000 - 077) are reserved for monitor calls, Fortran operating system calls, etc.

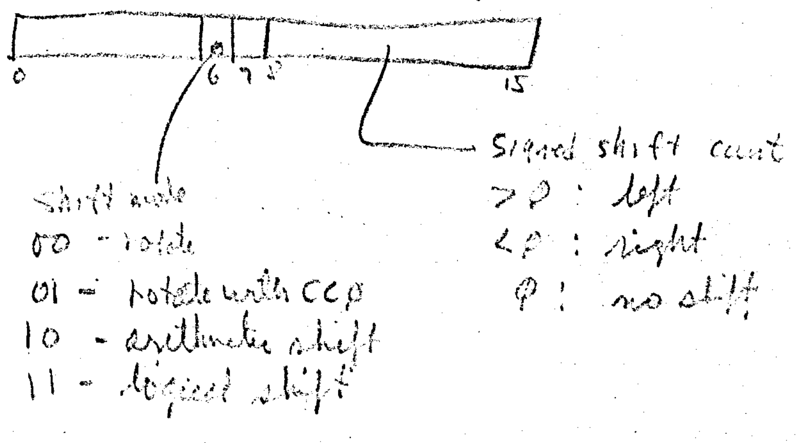
Below is a list of the first sixteen (20₈) of the implemented instructions.

Ptr. incmt

OP	mnem	definition
100	LDC	Load Character. The content of the effective address is used as a character pointer to locate an 8-bit byte. This byte is loaded into the right half of the specified R-register. The left half is cleared. The addressed memory word is left unchanged.
101	STC	Store Character. The content of the effective address is used as a character pointer. The right half of the specified R-register is stored at the indicated character position. The other half of the addressed word is unaltered. The content of R remains unchanged.
102	MUL	Multiply. The content of the effective address is multiplied by the content of the specified R-register. The high order result is stored in the specified R-register and the low-order result is stored in the next odd R-location. If R already is odd, the low order result is stored in the specified R location. C(EFA) x C(R) → C(R), C(R v 1). The arguments are treated as 16-bit logical data. The result is a 32-bit logical double word.
103	MULS	Multiply Signed. This operation is the same as MUL except that the arguments are treated as signed 2's complement data. The result is a signed 2's complement double word.

(Note: Both MUL and MULS leave CCØ unaltered. CC2 is set if the double word result is non-zero, otherwise it is cleared. CC1 is set if bit Ø of the high-order result is a neg, otherwise it's

OP	mnem	definition
104	DIV	Divide. The logical double word located by the R-field is divided by the logical word located by the effective address. The result is stored in the double word located by R. $C(R), C(R \vee 1) \div C(EFA) \rightarrow C(R)$ remainder $C(R \vee 1)$ quotient (Note: if R is odd, then is an integer divide)
105	DVS	Divide Signed. This operation is the same as DIV except that the arguments are treated as signed 2's complement data. The remainder and quotient are stored with the correct sign such that the identity: dividend = divisor x quotient + remainder holds. (Note: CC0 is left undisturbed and CC1, CC2 are set according to the quotient).
106	SUB	Subtract. The content of the effective address is subtracted from the content of the specified R-register. The result is placed in the specified R-register. The content of the effective address remains unchanged. $C(R) - C(EFA) \rightarrow C(R)$ (Note: The condition code bits 0,1,2 are set as in ADD).
107	SHFT	Shift. The content of R is shifted as indicated by the content of the effective address.



The right half byte of the content of the effective address is used as a signed step count. A positive value indicates a left shift. A negative value indicates a right shift. Bits 6,7 of the effective word indicate the type of shift to be performed:

00 rotate - bits leaving one end enter at the other end.

01 rotate with CCϕ - bits leaving one end enter Condition Code bit ϕ. CCϕ enters at the other end.

10 arithmetic shift - perform two's complement multiplication by powers of two. The sign is unchanged. When going to the right, the sign is shifted into bit 1. Ones or zeros leaving bit 15 are lost. The arithmetic error bit is set if, during shifting, the sign bit and bit 1 become unequal

11 logical shift - bits leaving one end are lost and zeroes enter the other end.

Condition code bit ϕ is undisturbed except for rotate with CCϕ. CC bits 1 and 2 are set according to the result.

LOGICAL COMPARE AND MODIFY (opcodes 110-113)

Bits of an R-field word that are masked by bits of a memory word may be tested and/or modified to determine a conditional branch. The bits to be tested and/or modified are selected by ones in the content of the effective address. Condition code bit 2 is cleared if all of the selected bits of the R-field word are zero; otherwise it is set to a one. Condition code bit 1 is set to a one if bit ϕ is selected and bit ϕ of the R-field word is a one; otherwise it is cleared. The selected bits of the R-field word are then modified or not depending upon the operation being performed.

110	TSTN	Test. Test the content of the R-field word with the content of the effective address. Set condition code bits 1 and 2 according to the result.
111	TSTZ	Test and Zero selected bits. Test the content of the R-field word with the content of the effective address. Set condition code bits 1 and 2 according to the result. Clear selected bits in the R-field word (ie, for every one in the content of the effective address, clear the corresponding bit in the R-field word).
112	TSTO	Test and set selected bits to Ones. Test the content of the R-field word with the content of the effective address. Set condition code bits 1 and 2 according to the result. Set selected bits in the R-field word (ie, for every one in the content of the effective address, set the corresponding bit in the R-field word). This performs the logical function inclusive or.
113	TSTC	Test and Complement selected bits. Test the content of the R-field word with the content of the effective address. Set condition code bits 1 and 2 according to the result. Complement selected bits in the R-field word (ie, for every one in the content of the effective address, complement the corresponding bit in the R-field word). This performs the logical function-exclusive or.
114	LCMP	Logical Compare. Compare the content of the R-field word with the content of the effective address both treated as unsigned logical words. Set condition code bits 1 and 2 according to the result.
115	ACMP	Arithmetic Compare. Compare the content of the R-field word with the content of the effective address both treated as signed arithmetic words. Set condition code bits 1 and 2 according to the result.

116 PUSH-DOWN class. The R field of the instruction is decoded to determine the function to be performed. In the following mapped location 12 is the push-down pointer which always points to the last item placed on the push-down list. Location 13 is the push-down counter. It is incremented each time an item is placed on the push-down list and is decremented each time an item is removed from the push-down list. A carry out of bit Ø indicates a push-down error.

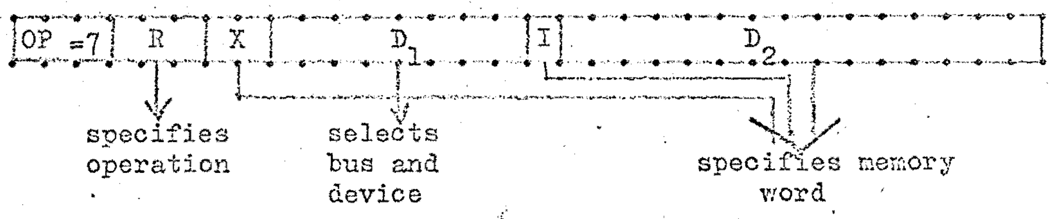
Mnem	R	Definition
PUSH	Ø	Push. The content of the effective address is placed in the next location of the push-down list.
PUSHB	1	Push and Branch. The program counter is placed in the next location of the push-down list. The effective address replaces the program counter.
PUSHBL	2	Push, Branch and Link. The subroutine linkage register is placed in the next location of the push-down list. The program counter is placed in the subroutine linkage register. The program counter is placed in the next location of the push-down list. The effective address replaces the program counter.
——	3	Unused
POP	4	Pop. The last word placed on the push-down list is moved to the content of the effective address.
POPB	5	POP and Branch. The sum of the effective address and the last word placed on the push-down list replaces the program counter. This is the return

		pair for PUSHB.
POPBL	6	Pop, Branch and Link. The sum of the effective address and the last word placed on the push-down list replaces the program counter. The last word placed on the push-down list is then placed in the subroutine linkage location. This is the return pair for PUSHBL.
—	7	Unused

class	OP	mnem	definition
IO	7		input/output instruction class; R bits specify particular operation; forced long format. D_1 is taken to specify the device, the lower order 6 bits select a device code, the high order 2 bits select one of 4 possible IO busses. Condition code bit 0 is unchanged. Condition code bit 1 is set on the read status (IOS) instruction if no device responds and on the command (IOC) instruction if the command is unacceptable. Bit 2 is cleared if the data byte or word resulting from the operation is identically zero, it is set otherwise. This has particular meaning in the IO test status (IOT) instruction. A byte is normally transmitted to the device from the right half of the effective address, some devices will automatically take the second byte also.

	R	operation
IOR	0	read device data word into selected memory word
IOS	1	read device status into selected memory word
	2	unassigned
	3	unassigned
IOW	4	write device data word from selected selected memory word
IOC	5	command, write device status from selected memory word
IOT	6	test status, the device status and the selected memory word are ANDed, a non zero result sets condition code bit 1
	7	unassigned

IO class instruction doubleword



Read instructions to a device that can write only or write instructions to a device that can only read will result in no data transfer.

2.3 General registers

Each level of priotiry contains a set of 16 general registers, 8 of which may be used by the program as accumulators, index registers, etc. The program status word (PSW) occupies registers 0 and 1. These registers occupy page 0 words 0 to 7 in the memory space as well as the R bits in the instruction, hence register to register instructions are possible. The registers may be stored, loaded, added into, etc. depending on the operation code of the particular instruction used. The second group of 8 registers contain the trap locations for unimplemented EOP instructions and the push down words. These may be modified or read as memory words but are not explicitly referenced as accumulators.

register use

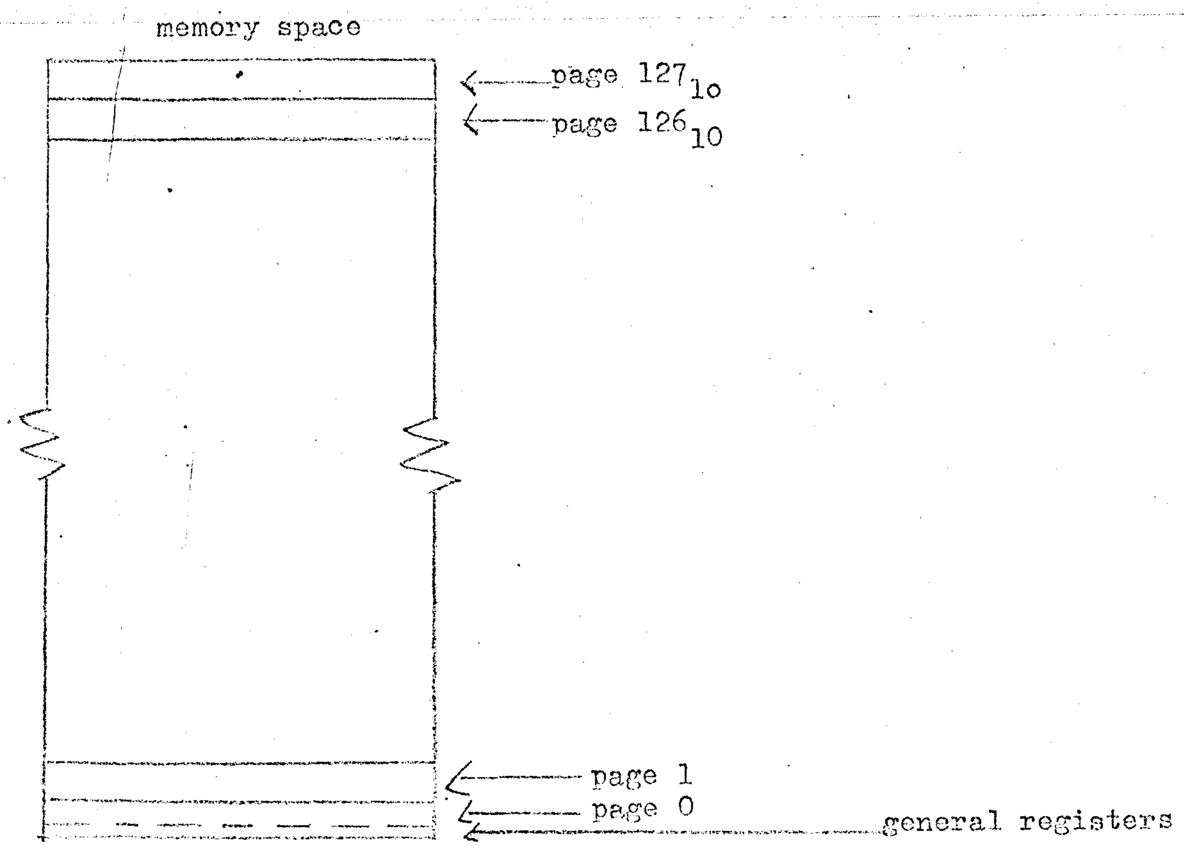
0	R_0	status word, contains condition code, etc.
1	R_1	status word, contains program counter (PC)
2	R_2	accumulator, subroutine linkage register, or secondary index
3	R_3	accumulator or main index register — ?
4	R_4	accumulator
5	R_5	"
6	R_6	"
7	R_7	"
8		EOP, receives the updated program counter ←
9		EOP, " instruction itself
10		EOP, " effective address
11		EOP, contains the entry point into the EOP handler, loaded into PC
12		contains the push down pointer
13		" " " " counter
14		reserved for use by processor
15		reserved for use by processor

For each level of machine priority, both background and IO, there exists a set of general registers; in addition, the hardware insures that the applicable set [illegible] is available at apparent locations $0\text{-}15_{10}$ in memory address space. Thus, the general registers need not be stored and restored during interrupts.

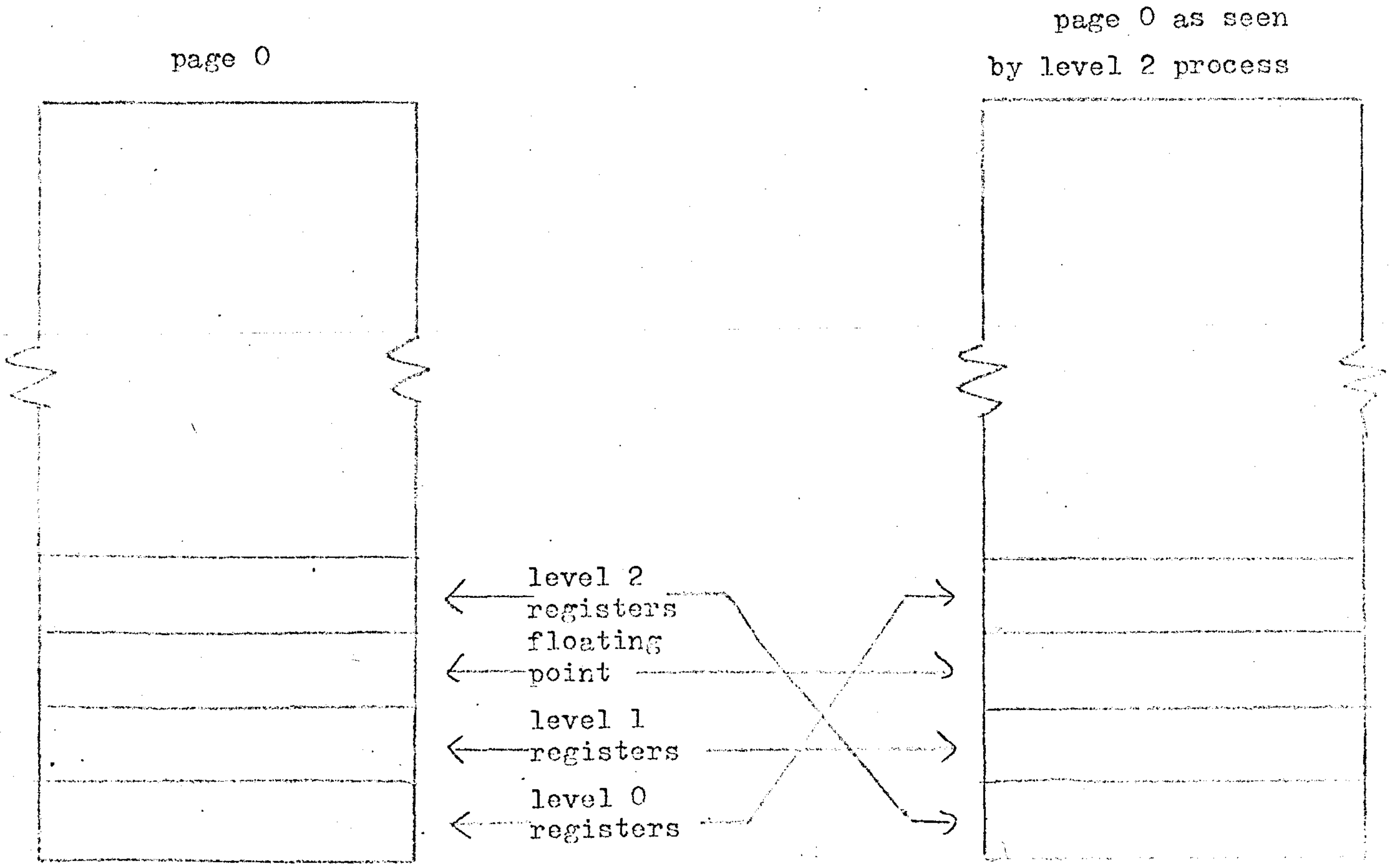
The lowest (background) priority level contains floating point registers. These registers are not available for use on the higher priority levels unless they are explicitly stored and restored under program control. Each of the 4 floating point registers is 64 bits (4 words) long, permitting multiple precision floating point instructions. In all floating operations the R bits of the instructions specify these registers, only the low order 2 bits of R are used.

The set of general registers map onto the main memory space in page 0. The figure on the left shows the entire memory space; the figure on the right is an exploded view of physical memory. Apparent memory is the memory space as seen by the running process; this differs from physical memory in the location of its general registers as is shown for a priority level 2 process in the bottom figure.

model	levels
14	2, core
16	2 minimum hardware, 4 maximum
32	4 hardware



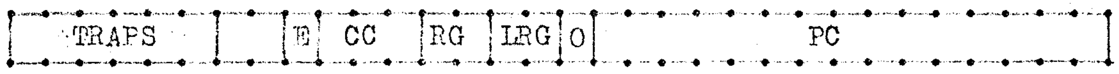
memory space
page 127_{10}
page 126_{10}
page 1
page 0
general registers



page 0
page 0 as seen
by level 2 process
level 2
registers
floating
point
level 1
registers
level 0
registers

2.4 Program Status Word

The collection of bits that constitute the state of the processor between instructions are called, collectively, the Program Status Word (PSW). This state word occupies the doubleword at memory locations 0 and 1 of the active process, corresponding to general registers R_0 and R_1.



bit(s)	definition
0	arithmetic (add, divide, floating, etc.) enabled if bit 8 = 1
1	machine check (processor or memory error)
2	nonexistant memory (reference to an address not in the memory system or to a protected area)
3	nonexistant instruction (attempt to use an instruction for which no such hardware has been provided)
4	priveleged instruction (attempt to execute a system instruction while in user mode)
5	read only violation (attempt to write into a write protected memory area)
6-7	unused
8	arithmetic trap enable
9	condition code bit 0
10	condition code bit 1
11	condition code bit 2
12-13	priority of active process (current register group)
14-15	priority of interrupted process (last register group)
16	unused, always 0
17-31	program counter of active process

2.5 Addressing

Addresses are generated by either long or short format instructions. In either case, the processor forms a 15 bit effective address (EFA) which it sends to the memory system. The left byte (high order 7 bits) of the address is called the page, the right byte is called the line; there are 128 pages of 256 words each directly addressable.

The available addressing modes are :

direct (no indexing) to any word in memory ,

relative ($\pm$ 127_{10} words from the instruction),

immediate (the next word is the operand,

linked (the subroutine linkage register is used to pick up arguments or make returns),

indexed. The short displacement (D_1) is taken to be a two's complement negative number whose sign is to be extended. Long format addresses are specified whenever $D_1 = 128_{10}$ or the instruction implicitly forces this format (all IO and extended op code instructions).

Addressing table

X	short	long	description
0	D_1	D_2	direct
1	$\pm D_1 + PC$	PC+1	relative/immediate
2	$\pm D_1 + R_2$	$D_2 + R_2$	linked
3	$\pm D_1 + R_3$	$D_2 + R_3$	indexed

The basic addressable unit is the word (two bytes, 16 bits), although certain instructions do reference bytes or doublewords. Words in storage are consecutively numbered starting with 0. The 15 bit address field accomodates a maximum of 32,768 words. When only a part of the maximum storage capacity is available in a given installation, the available storage is contiguously addressable from 0. A nonexistant memory trap occurs when any operand is located beyond the installed capacity. The invalid address is recognized when the data is accessed and a program interruption occurs.

2.7 Priority (Interrupt) Structure

The interrupt system is designed to handle 4 levels of priority including the main program (at the lowest level). The interrupt due to an internal source (trap) or external source causes the new, appropriate set of general registers to be used in place of the previously operating set. Hence, no time is lost before the interrupt service program can begin. Priorities of service are fully nested; high priority requests interrput low priority processes but even lower ones are delayed. Linkage between the interrupted process and the interrupt process is performed by the LRG bits of the program status word (PSW). These bits receive the priority number of the interrupted process; the priority of the currently active process is contained in a 2 bit register, the register group (RG) register. Priorities are assigned as follows:

Level (RG=)	use
0	main program
1	traps, lowest hardware device level
2	device hardware level
3	highest hardware device level

When an interrupt occurs, the LRG bits of the new PSW are loaded from the RG register. The RG register is then set to the new priority level. Subsequent instructions will come from the interrupt process PC and register set. The interrupt is cleared with a debreak IO instruction which loads the RG register from the LRG bits of the current PSW. Subsequent instructions will be from the originally interrupted process.

3.1 Devices and controllers

The hardware involved in IO operation is logically divided into 4 parts: IO section, IO bus, controller, and device. The IO section and bus are described in detail below. Controllers and devices are generally different for each type of IO media; from the programming point of view most controller functions merge with IO device functions.

In all cases, the controller function is to provide the locical and buffering capabilities necessary to operate the associated IO device. Each controller functions only with the IO device for which it is designed, but each controller has standard signal connections with regard to the IO bus. The teletype device (keyboard, printer), for example, connects to the IO bus through teletype controller logic (single character data buffering and interrupt logic). The detailed meaning of the command/status bits read under program control through the IO section from controller type to type but the general format remains unchanged.

3.2 Modes of data transfer

There are 3 basically different modes of data transfer available in the IO system: program controlled, multiplexor channel, and selector channel. All three use the standard IO bus interface; the third, optionally implemented, provides an additional physical bus interface and additional control logic at the processor end. Maximum data transfer rate of each mode varies with processor model, the program controlled rate is lowest and the selector channel rate highest. In all cases transfer sequences are initiated by IO instructions issued to the appropriate controller, rather than to the channel.

Program controlled transfer, while slowest, provides the greatest flexibility. Data may be modified, limit checked, or otherwise monitored as it is inputted and special control sequences required by special purpose or custom designed IO equipment may be generated. For the slower devices, especailly paper tape or teletype, direct program control of IO leads to simpler programming.

Multiplexor channels are provided in the basic processors. When a device requires channel servicing, the state of the processor is dumped, the device serviced, and the state of the processor restored. The program never realizes that the transfer took place except that it has been subjected to a short delay. The multiplexor channel is capable of sustaining concurrent IO operations with several devices. Bytes of data are interleaved together and routed to or from the selected IO devices and to or from the desired locations in main storage. The channel's single data path is time shared by the concurrently operating devices.

Selector channels are capable of operating only one device at a time, however they permit extremely high data rates, over a million bytes per second is possible. Devices such as disc files operate only with seclector channels, other devices operate in all data transfer modes. As with the multiplexor channel, the selector channel is invisible to the programmer; all instructions are directed at the device rather than the channel. Devices requiring special hardware features in the IO system, such as signal averaging, normally would add a selector channel with appropriate ROS control

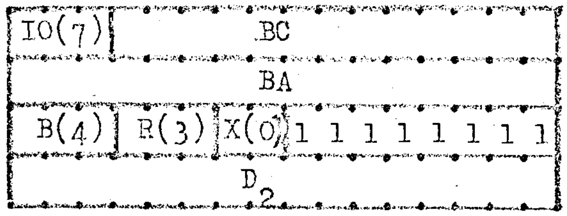
N

3.3 Operation of the multiplexor channel and interrupt

A device signals that it needs attention by requesting service at the priority level that has been assigned to the device. Devices operating on the multiplexor channel require attention for every byte (word) of information transferred; devices under program control require attention whenever they complete a specified operation. When the priority of the active process drops below the priority of the request, the interruption occurs. The state of the old process is stored in its general register set (R_0 and R_1 contain the processor state) and a new general register set is switched in. The priority of the old process is saved in the LRG bits of the new status word. Processor hardware then requests the device to transmit its address (6 bits); the address is or'ed into bit positions 8 through 15 of a word with bit position 7 set to 1 and all other positions 0. This address, called the interrupt address, lies somewhere on page 1.

Subsequent operation depends on the word found at the interrupt address. Any instruction class other than IO is executed, such an instruction is normally a branch to an IO service routine for program controlled transfer operation. An IO class instruction signifies that the device is under multiplexor channel control. A byte (word) of data is read from (written to) the device and packed into memory (unpacked from memory). The byte address pointer and byte counter are updated. If the byte counter went to zero indicating that the last byte (word) has been transferred or the device indicated an unusual condition, the instruction following the IO class instruction is also executed. This is normally a branch to an IO service routine that re-initializes the device and channel for subsequent operations. If no unusual condition was detected and the byte counter did not overflow, the device continues operating and will reinterrupt with the next data byte.

The format of the words statting at the interrupt address for multiplexor channel operation are given below. The double word at that address for program controlled transfer is a simple single or doubleword branch instruction. Branch instructions are normally unconditional, direct.



BC stands for byte counter and maintains a count of data bytes as they are transferred to and from the device. Prior to each transfer BC is incremented to determine whether or not this is the last byte. When initializing a device for multiplexor channel operation, the programmer must load BC with the two's complement of the number of bytes to be transferred. At the end of channel operation, the entire word at the interrupt address will be set to zero. Exceptional conditions which cause termination before the specified number of bytes is read leave the word non zero.

BA stands for byte address and maintains the address of the data byte next tobe transferred to and from memory. Prior to each transfer BA is incremented to form the byte address of the data byte. This byte address is shifted right before use to form a word address, the end bit determines which half of the word the data byte will be loaded into. A 1 indicates the left byte, a zero the right byte. When the program initializes the channel it must load BA with the byte address of the first byte to transferred.

Unless the word executed at interrupt address (or interrupt address + 2 when reached during channel operation) is a branch class, control immediately returns to the interrupted process. The priority level is restored from the LRC bits and the processor continues with the old program counter, status, and general registers.

3.4 IO Bus

The connection between the processor and the IO device control units is called the IO Bus. The interface consists of signal lines that connect the control units to the processor; except for the signal used to establish selection, all communications lines to and from the processor are common to all control units. At any one instant, however, only one control unit may be logically connected to the processor. The logical connection is maintained from the time it is first established by the processor until it is broken by the processor. The rise and fall of all signals transmitted over the interface are controlled by interlocked responses. This interlocking removes the dependence of the interface on circuit speed and bus length, making it applicable to a wide variety of circuits and data rates.

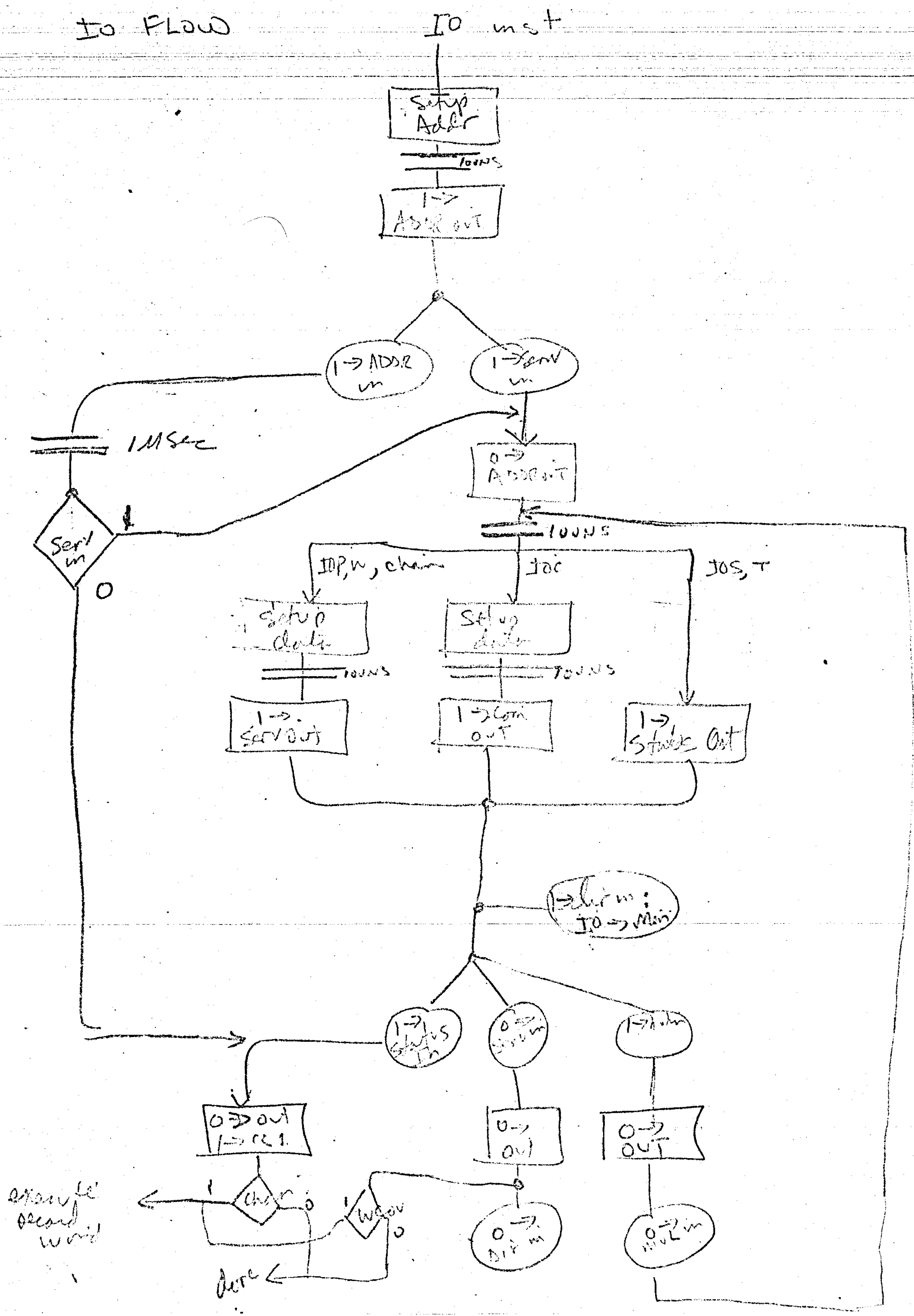
32 signals comprise the bus including 16 control signals and 16 data signals. Half of the signals transmit to the processor, the other half recieve from it. The Select Out signal is retransmitted by each device, as is Select In.

Line (direction)		function
Address	In	echo on Address Out, used to detect nonextant device; response to Select Out
Address	Out	selection code is on data lines, respond with Service In and become selected
Command	Out	command is on data lines, respond with Service In if acceptable; else, respond with Status In
Data 0	Out	data lines from processor to device controllers, also includes command specification and address
⋮		
Data 7	Out	
Data 0	In	data lines from device controller to processor, also includes status and address
⋮		
Data 7	In	
Direction	In	additional response to Service Out for to computer data transfer
Multiple	In	response to service out when additional byte is required
Operational	Out	system reset when down
Request 1	In	request to processor for attention at each of 3 priority levels (1 lowest, 3 highest)
⋮		
Request 3	In	
Service	In	drops as response to most Out signals, rises as response to Address In
Service	Out	accept or transmit data on data lines, respond by dropping Service In or raising Multiple In
Status	In	response to Command Out if unacceptable command
Status	Out	processor request for controller status, respond by sending status and raising Direction In
Select 1	Out	processor request for address from requesting device at the same priority leve. Select Out is propagated by those devices not requesting, blocked by the first device requesting, respond with Address In, send address on data lines
⋮		
Select 3	Out	

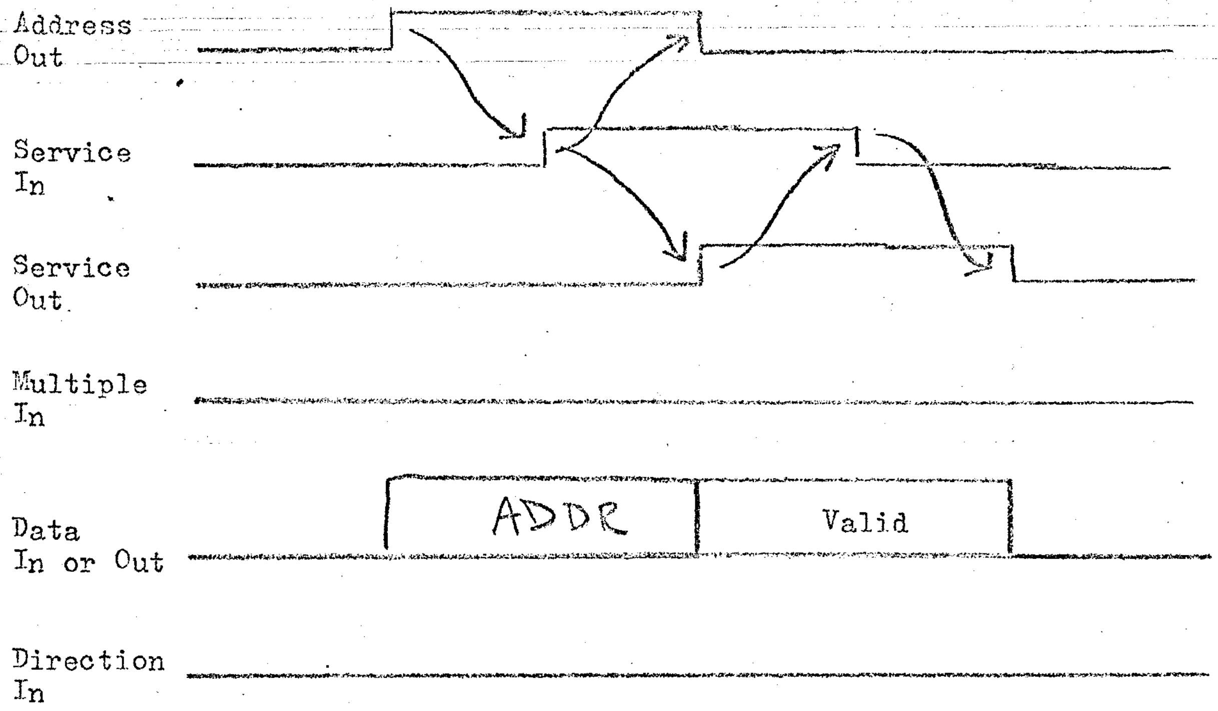
3.5 Bus flow diagrams

The following diagrams indicate signal timing relationships on the bus for various forms of transfer. Note that the only difference between read and write is the status of the Direction In bus line, that line solely determines the direction of transfer. Devices capable of both reading and writing have a status bit which determines the direction and use of the corresponding Direction In bus signal.

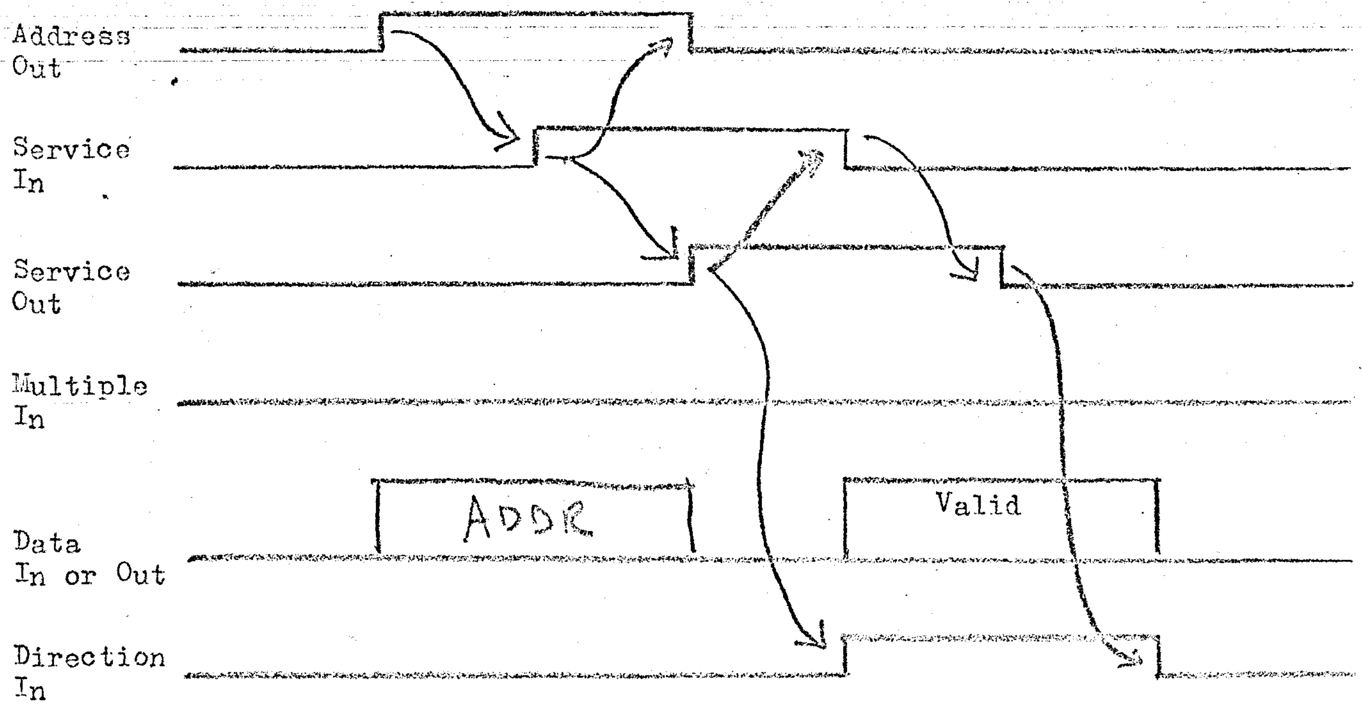
IO FLOW



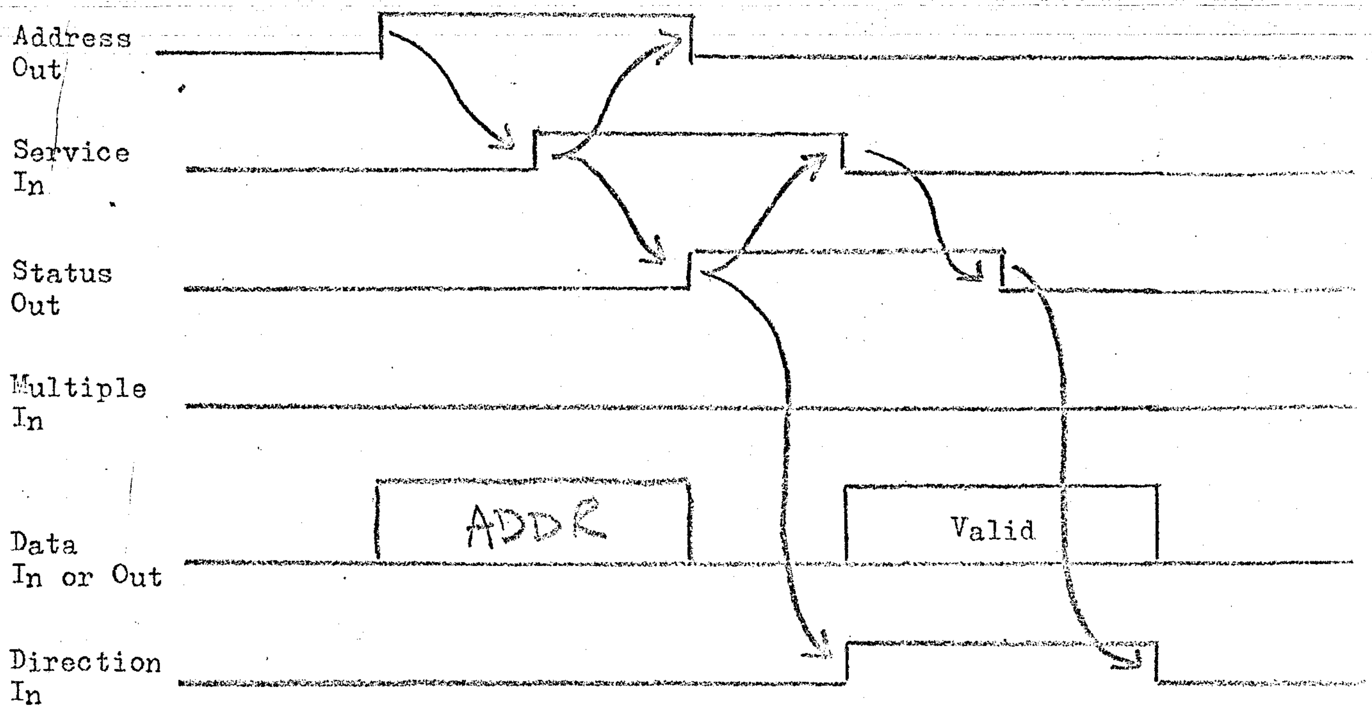
IO inst
Setup Addr
100NS
1 -> Addr out
1 -> ADDR in
1 -> Serv in
1MSec
Serv in
1
0
0 -> Addr out
100NS
IOP, w, chain
IOC
IOS, T
Setup data
Setup data
100NS
100NS
1 -> Serv out
1 -> Com out
1 -> Strobe Out
1 -> Dir in : IO -> Mem
1 -> Status In
0 -> Serv in
1 -> Hold
0 -> out 1 -> CC 1
0 -> out
0 -> OUT
Chain
1
0
Word
1
0
0 -> Dir in
0 -> Hold in
execute second word
done



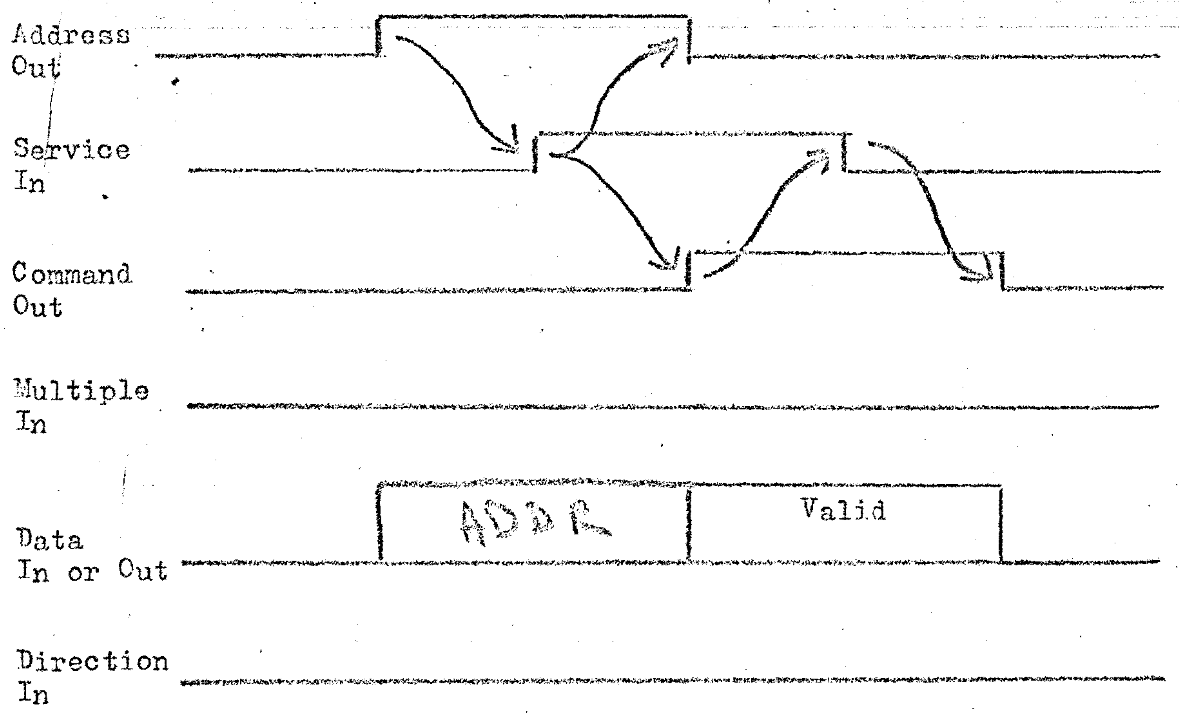
IOW Command operation sequence



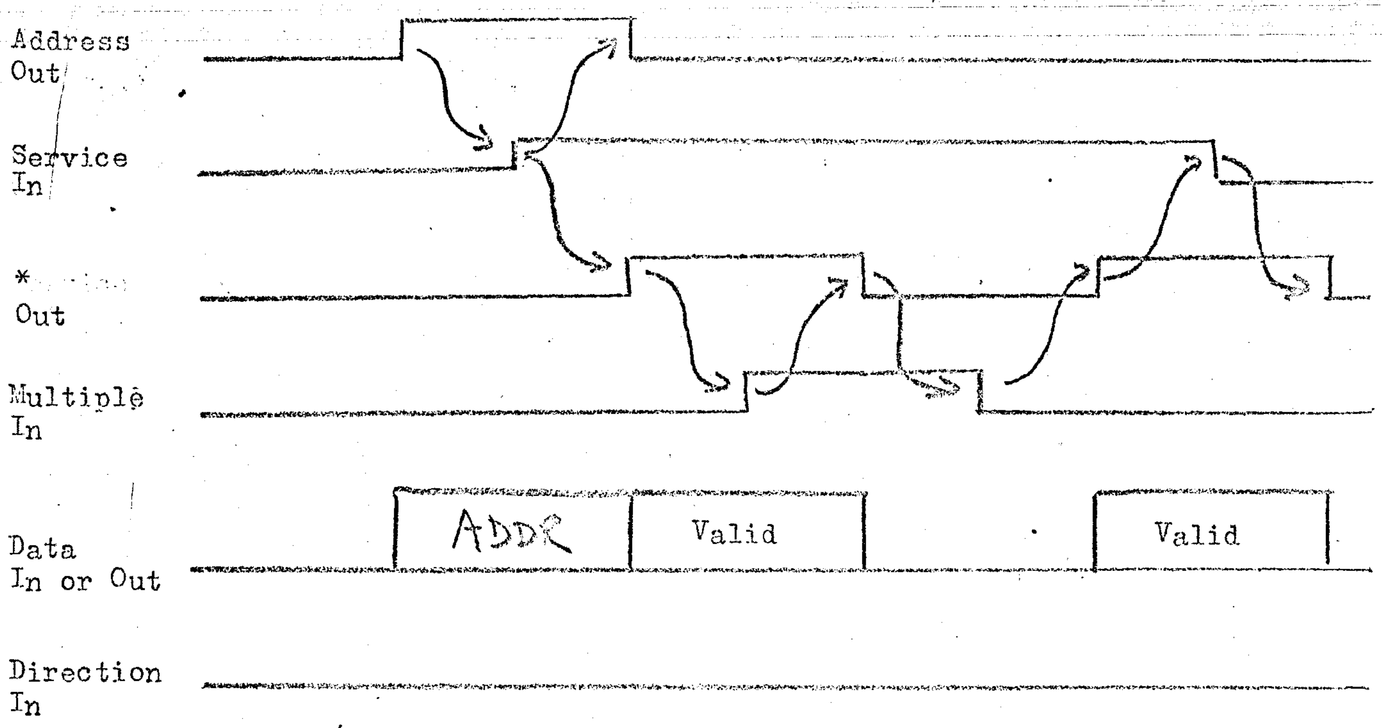
IOR Command operation sequence



IOT, IOS Command operation sequence

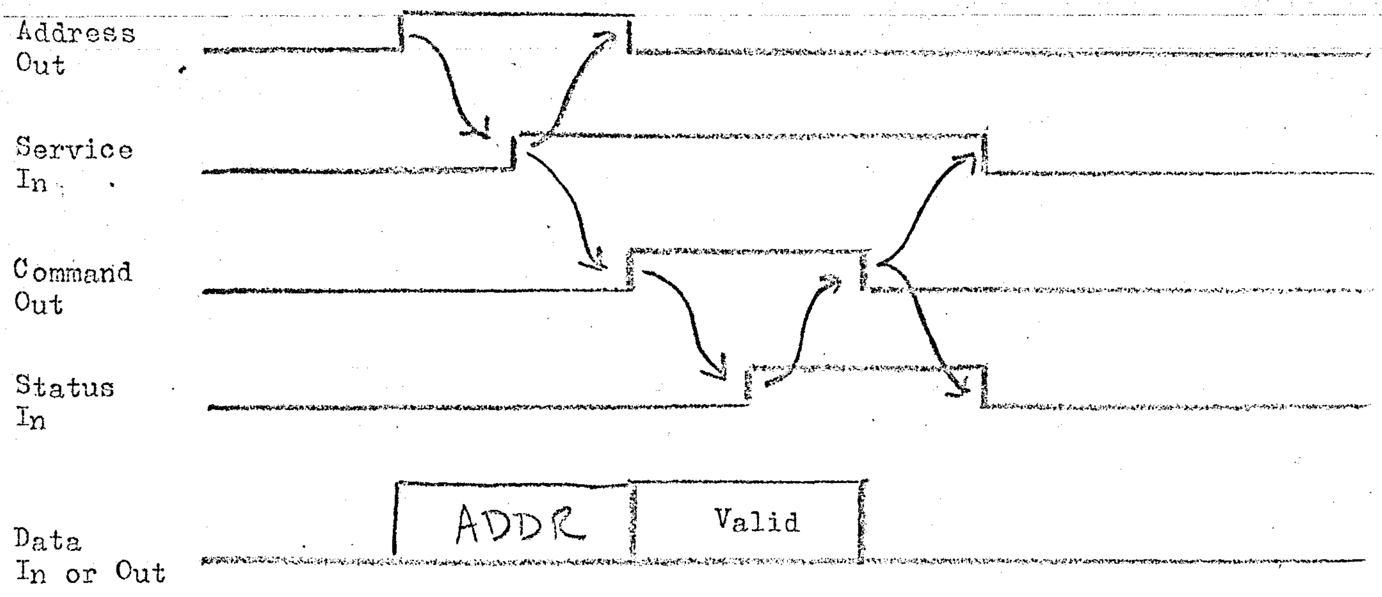


IOC Command operation sequence

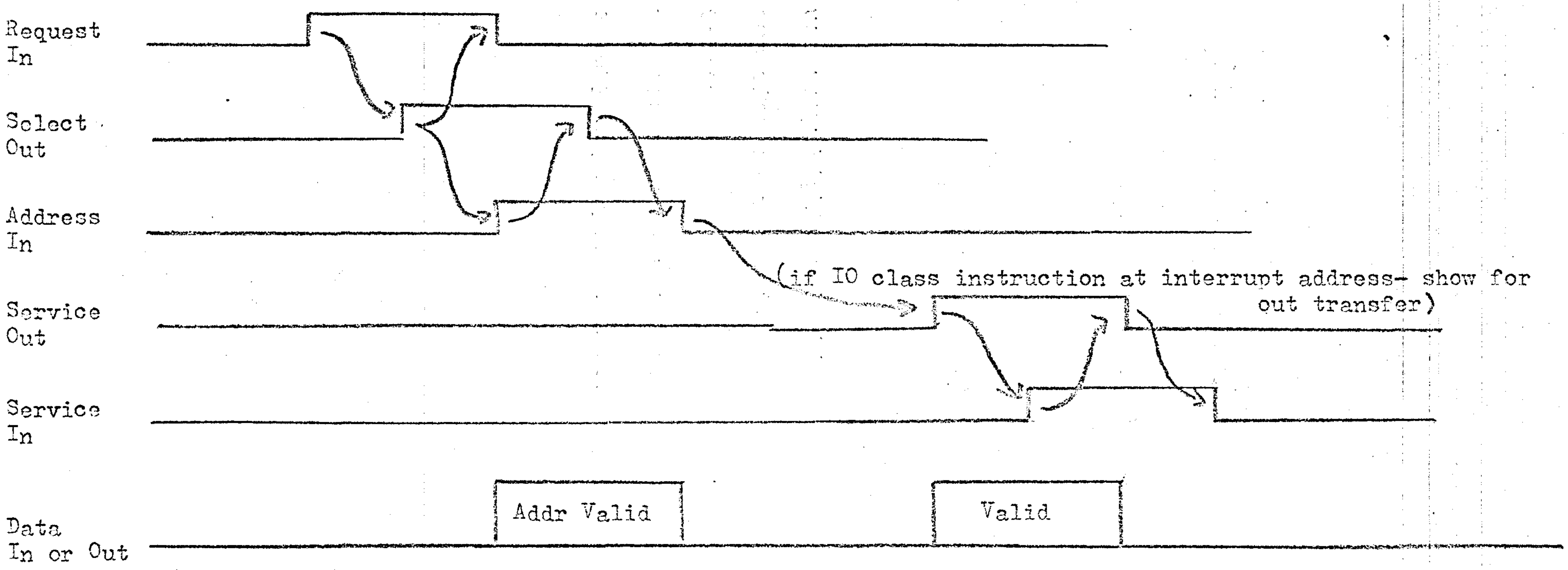


* Service, Command, Status

Multiple Byte transfer



Illegal Command operation sequence



Response to interrupt

1.0 Introduction

This document proposes a high speed, serial IO bus system in which 2 coaxial cables emanate in the processor and thread their way through all IO devices. The bus is broken (terminated, received, and retransmitted) in each device to establish priorities and to permit long lines without degrading the performance of physically close devices. All signal flow is syncronized to code blocks originating in the processor. The following are the major properties of the system:

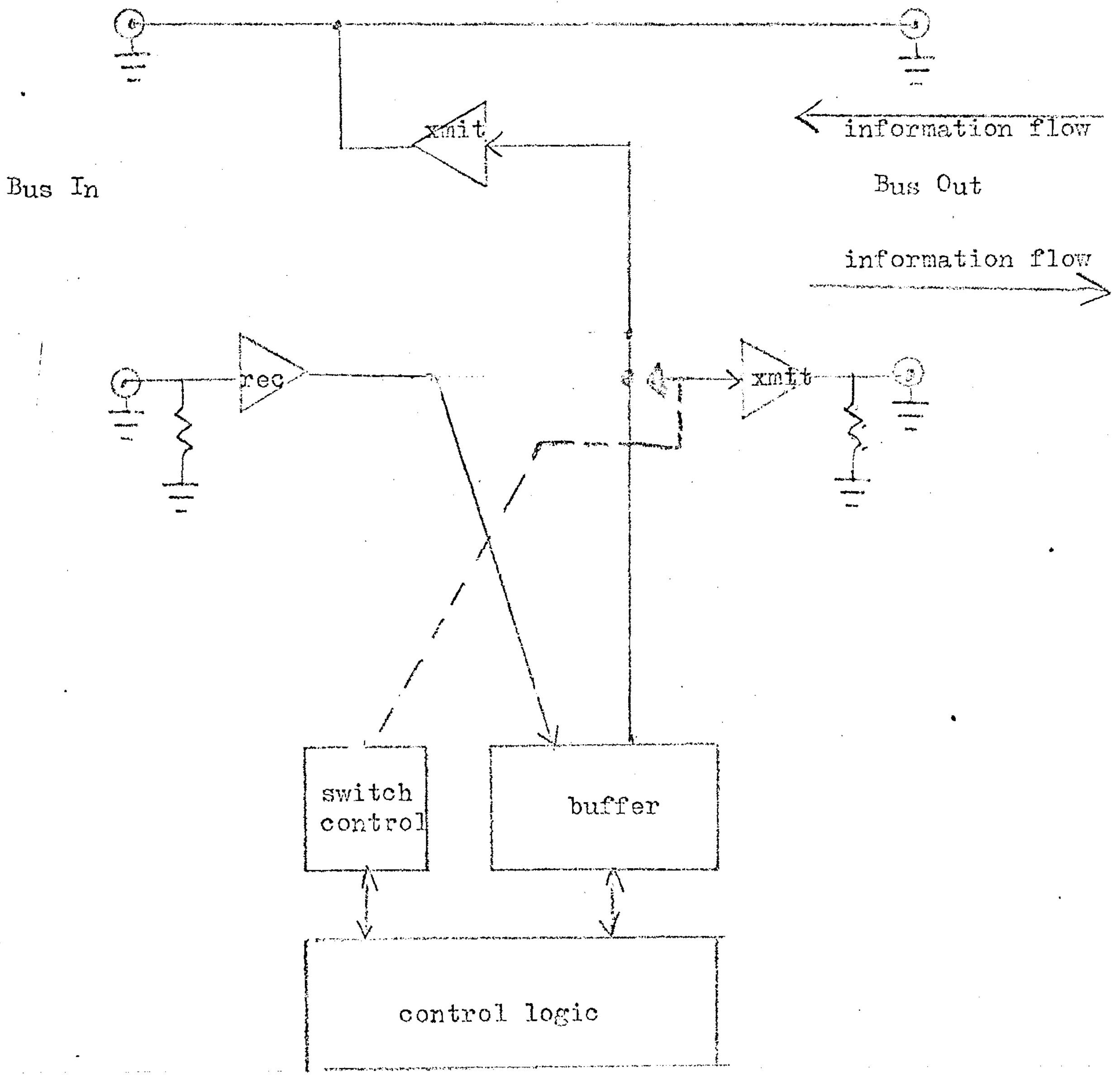
1. Simple cabling - two single, simple coaxial cables with standard connectors are used.
2. High performance circuitry - since only 1 receiver and 2 transmitters are required at each device, it is economical to use very highly reliable, very high performance circuitry.
3. Information is transmitted in code blocks much as they are by teletype.
4. IO devices require a serial to parallel conversion buffer, some very high speed control logic, and the analog circuitry used in the receivers and transmitters. Much of this logic would be required in any bus system.
5. Multibus - since the bus itself is simple, devices may easily be interfaced to several. The intent is, partially, to facilitate expansion to multiprocessor systems without expensive crossbar switching systems.

With a 50ns bit rate on the bus, a code block would require 0.8 usec for transmission. Round trip time through 20 devices and a total of 100' of cable yields a basic half cycle of 5 usec for the far devices and about 1 usec for the close ones. These are rates comparable to PDP-9.

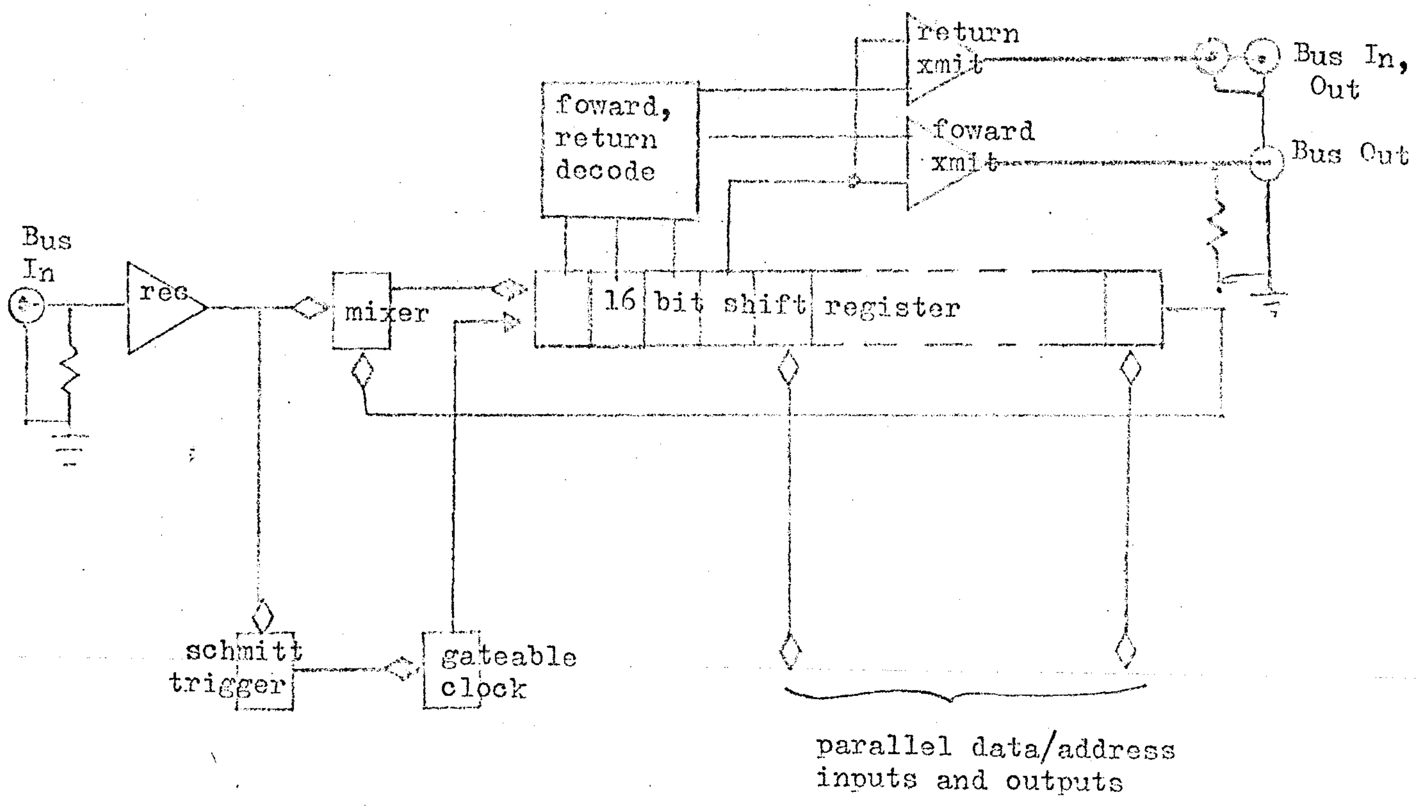
block length	.8
device delay 20x0.2	4.0
cable delay 100x0.002	.2
	5.0 usec

2.0 Basic device Hardware

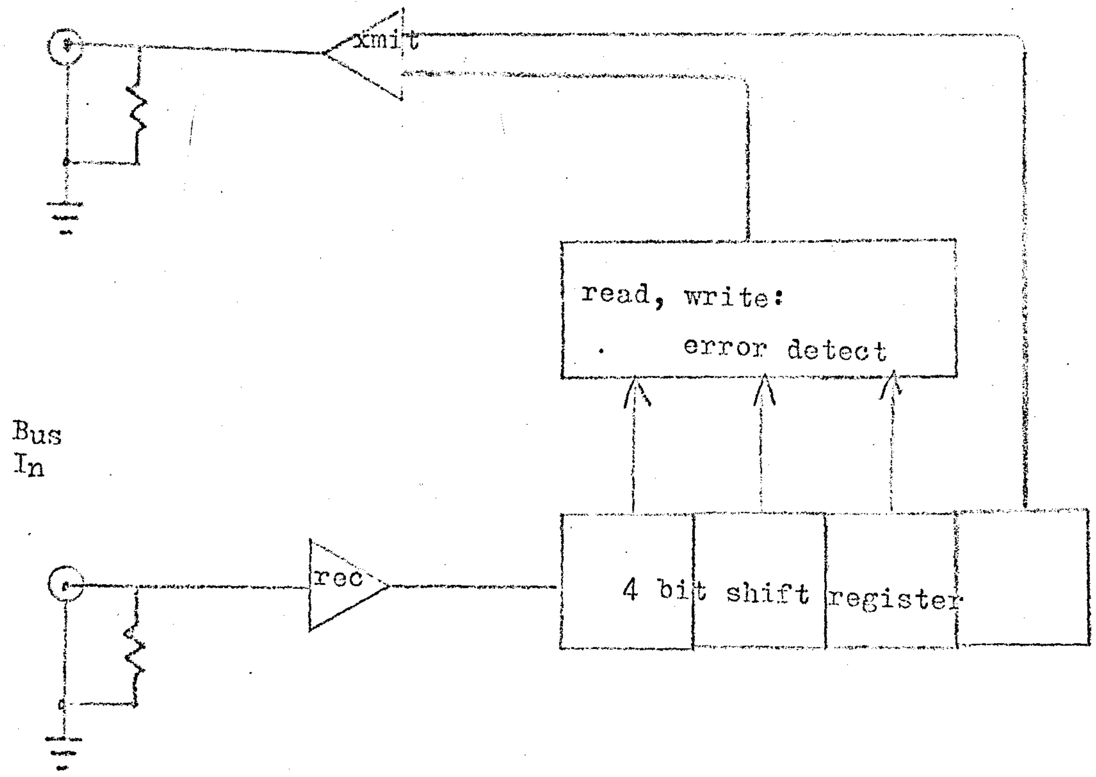
2.1 Block Diagram



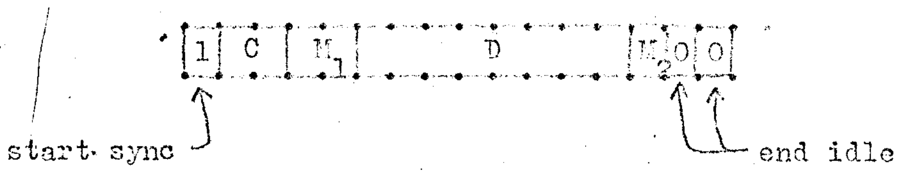
2.2 Control circuit details



2.3 Terminator



3.0 Code Format



C	command type
M_1M_2	command mode
D	data or address

3.1 Command types

0	Select	- address follows, become selected if address match
1	Scan	- processor looking for interrupt, send address
2	Read	- selected device to send data
3	Write	- selected device to accept data

3.2 Command mode

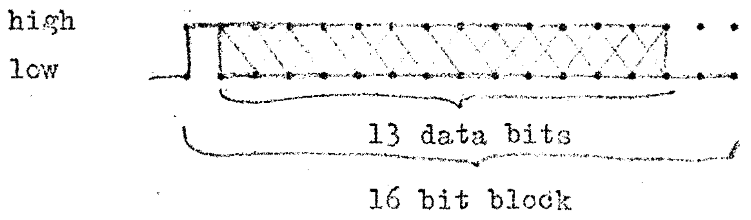
Mode is used as additional control information during read or write operations. Distinction is made between read/write status vs data, additional bytes, illegal commands, etc.

3.3 Data

Data bits transmit a byte of information or an address. On write operations they are supplied by the processor; on read they are inserted by the device.

4.o Bus signal electrical properties

The bus signal consists of a group of code bits spaced at 50ns intervals. The high state indicates a "1", the low state a "0". Rise and fall times are on the order of 10 ns.

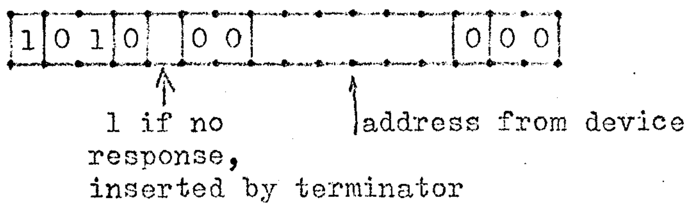


5.0 Basic operation

Devices receive a bit stream from the adjacent device, phase their internal clock to the start bit at the begining of the stream, and generally retransmit the stream to the next device after approximately 4 bits of delay time. Decoding the bit stream on Bus In may, however, result in a very different operation. In responding to a read command, for example, the selected device modifies the command string, appending its data bits, and sends it back along the second bus but does not foward it.

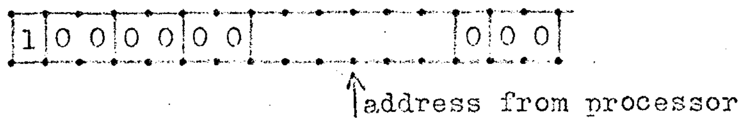
5.1 Scan operation

Whenever the computer interrupt system is on, the processor periodically issues scan sequences. When received by a requesting device, the sequence is modified to include the address of the requesting device and is transmitted back to the processor; it is not fowarded. The device address is mapped by the processor into an interrput address and the interrupt is processed. Scan sequence frequency is determined by bus length. The end of bus terminator returns the unacknowledged scan sequence. Internal request syscronization in each device is accomplished at the begining of each block.



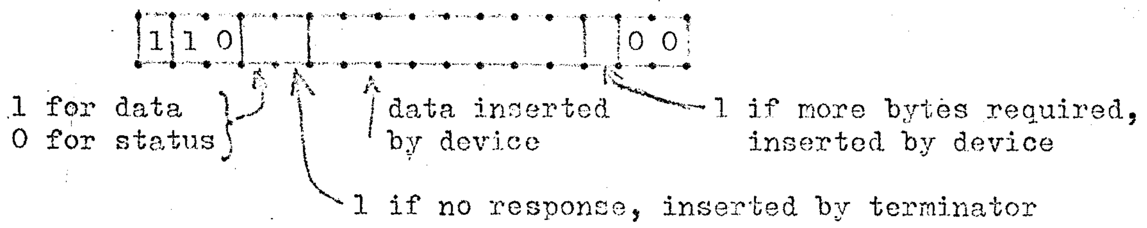
5.2 Selection Block

A selection block comprises the first part of a read or write operation. An address is transmitted to all devices, one of which responds to address match by setting an internal select flip flop. The sequence itself is retransmitted by each device to the next.



5.3 Read

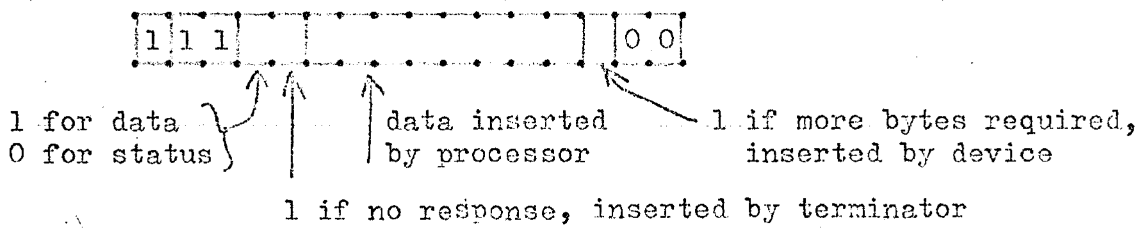
A read block is initiated by the processor when it transmits a dummy data word. All non selected devices merely re-transmit forward the incoming sequence, the selected device does not forward the block, rather it inserts its byte to be read, necessary control information, etc. and returns it to the processor. If no device has been selected, the terminator will retransmit the block with an error indication.



1 1 0
0 0
1 for data
0 for status
data inserted by device
1 if more bytes required, inserted by device
1 if no response, inserted by terminator

5.4 Write

A write data or write status (command) operation begins with a selection block. The processor then transmits a write block which contains the data byte. The block is not forwarded but is retransmitted to the processor with any necessary control information. As in read, if no device has been selected, the terminator will retransmit the block with an error indicator.



1 1 1
0 0
1 for data
0 for status
data inserted by processor
1 if more bytes required, inserted by device
1 if no response, inserted by terminator

2 levels of interrupt
For xfer and for status
interrupts to locs ad f(device#)

~~not use another channel~~ but use a set for end conditions 4.2.3

PDP-X MEMORANDUM INDEX

#9

Read or write P
[illegible] P [illegible]
M

2
~~Selector~~ channels lose

~~why address a bus?~~

like SDS
[illegible]

#13 → 3.2

multiplexor
Selector

why not use most sig bit to address high

Bad Name

P

2.6.1
chart [illegible]

→ interrupt on reader switch, punch reader,

→ End of file on card reader to denote when finished; ~~to decide to as a appl. just punted one~~

A new architecture for mini-computers— The DEC PDP-11

by G. BELL,* R. CADY, H. McFARLAND, B. DELAGI, J. O'LAUGHLIN and R. NOONAN

Digital Equipment Corporation
Maynard, Massachusetts

and

W. WULF

Carnegie-Mellon University
Pittsburgh, Pennsylvania

INTRODUCTION

The mini-computer** has a wide variety of uses: communications controller; instrument controller; large-system pre-processor; real-time data acquisition systems . . .; desk calculator. Historically, Digital Equipment Corporation's PDP-8 Family, with 6,000 installations has been the archetype of these mini-computers.

In some applications current mini-computers have limitations. These limitations show up when the scope of their initial task is increased (e.g., using a higher level language, or processing more variables). Increasing the scope of the task generally requires the use of more comprehensive executives and system control programs, hence larger memories and more processing. This larger system tends to be at the limit of current mini-computer capability, thus the user receives diminishing returns with respect to memory, speed efficiency and program development time. This limitation is not surprising since the basic architectural concepts for current mini-computers were formed in the early 1960's. First, the design was constrained by cost, resulting in rather simple processor logic and register configurations. Second, application experience was not available. For example, the early constraints often created computing designs with what we now consider weaknesses:

1. limited addressing capability, particularly of larger core sizes
2. few registers, general registers, accumulators, index registers, base registers
3. no hardware stack facilities
4. limited priority interrupt structures, and thus slow context switching among multiple programs (tasks)
5. no byte string handling
6. no read only memory facilities
7. very elementary I/O processing

* Also at Carnegie–Mellon University, Pittsburgh, Pennsylvania.

** The PDP-11 design is predicated on being a member of one (or more) of the micro, midi, mini, . . ., maxi (computer name) markets. We will define these names as belonging to computers of the third generation (integrated circuit to medium scale integrated circuit technology), having a core memory with cycle time of .5 ~ 2 microseconds, a clock rate of 5 ~ 10 *Mhz* . . ., a single processor with interrupts and usually applied to doing a particular task (e.g., controlling a memory or communications lines, pre-processing for a larger system, process control). The specialized names are defined as follows:

	maximum addressable primary memory (words)	*processor and memory cost (1970 kilodollars)*	*word length (bits)*	*processor state (words)*	*data types*
micro	8 K	~ 5	8 ~ 12	2	integers, words, boolean vectors
mini	32 K	5 ~ 10	12 ~ 16	2–4	vectors (i.e., indexing)
midi	65 ~ 128 K	10 ~ 20	16 ~ 24	4–16	double length floating point (occasionally)

8. no larger model computer, once a user outgrows a particular model
9. high programming costs because users program in machine language.

In developing a new computer the architecture should at least solve the above problems. Fortunately, in the late 1960's integrated circuit semiconductor technology became available so that newer computers could be designed which solve these problems at low cost. Also, by 1970 application experience was available to influence the design. The new architecture should thus lower programming cost while maintaining the low hardware cost of mini-computers.

The DEC PDP-11, Model 20 is the first computer of a computer family designed to span a range of functions and performance. The Model 20 is specifically discussed, although design guidelines are presented for other members of the family. The Model 20 would nominally be classified as a third generation (integrated circuits), 16-bit word, 1 central processor with eight 16-bit general registers, using two's complement arithmetic and addressing up to 2^{16} eight bit bytes of primary memory (core). Though classified as a general register processor, the operand accessing mechanism allows it to perform equally well as a 0-(stack), 1-(general register) and 2-(memory-to-memory) address computer. The computer's components (processor, memories, controls, terminals) are connected via a single switch, called the Unibus.

The machine is described using the PMS and ISP notation of Bell and Newell (1970) at different levels. The following descriptive sections correspond to the levels: external design constraints level; the PMS level—the way components are interconnected and allow information to flow; the program level or ISP (Instruction Set Processor)—the abstract machine which interprets programs; and finally, the logical design level. (We omit a discussion of the circuit level—the PDP-11 being constructed from TTL integrated circuits.)

DESIGN CONSTRAINTS

The principal design objective is yet to be tested; namely, do users like the machine? This will be tested both in the market place and by the features that are emulated in newer machines; it will indirectly be tested by the life span of the PDP-11 and any offspring.

Word length

The most critical constraint, word length (defined by IBM) was chosen to be a multiple of 8 bits. The memory word length for the Model 20 is 16 bits, although there are 32- and 48-bit instructions and 8- and 16-bit data. Other members of the family might have up to 80 bit instructions with 8-, 16-, 32-and 48-bit data. The internal, and preferred external character set was chosen to be 8-bit ASCII.

Range and performance

Performance and function range (extendability) were the main design constraints; in fact, they were the main reasons to build a new computer. DEC already has (4) computer families that span a range* but are incompatible. In addition to the range, the initial machine was constrained to fall within the small-computer product line, which means to have about the same performance as a PDP-8. The initial machine outperforms the PDP-5, LINC, and PDP-4 based families. Performance, of course, is both a function of the instruction set and the technology. Here, we're fundamentally only concerned with the instruction set performance because faster hardware will always increase performance for any family. Unlike the earlier DEC families, the PDP-11 had to be designed so that new models with significantly more performance can be added to the family.

A rather obvious goal is maximum performance for a given model. Designs were programmed using benchmarks, and the results compared with both DEC and potentially competitive machines. Although the selling price was constrained to lie in the $5,000 to $10,000 range, it was realized that the decreasing cost of logic would allow a more complex organization than earlier DEC computers. A design which could take advantage of medium- and eventually large-scale integration was an important consideration. First, it could make the computer perform well; and second, it would extend the computer family's life. For these reasons, a general registers organization was chosen.

Interrupt response

Since the PDP-11 will be used for real time control applications, it is important that devices can communicate with one another quickly (i.e., the response time of a request should be short). A multiple priority level, nested interrupt mechanism was selected; additional priority levels are provided by the physical position of a device on the Unibus. Software polling is

* PDP-4, 7, 9, 15 family; PDP-5, 8, 8/S, 8/I, 8/L family; LINC, PDP-8/LINC, PDP-12 family; and PDP-6, 10 family. The initial PDP-1 did not achieve family status.

unnecessary because each device interrupt corresponds to a unique address.

Software

The total system including software is of course the main objective of the design. Two techniques were used to aid programmability: first benchmarks gave a continuous indication as to how well the machine interpreted programs; second, systems programmer continually evaluated the design. Their evaluation considered: what code the compiler would produce; how would the loader work; ease of program relocability; the use of a debugging program; how the compiler, assembler and editor would be coded—in effect, other benchmarks; how real time monitors would be written to use the various facilities and present a clean interface to the users; finally the ease of coding a program.

Modularity

Structural flexibility (sometimes called modularity) for a particular model was desired. A flexible and straightforward method for interconnecting components had to be used because of varying user needs (among user classes and over time). Users should have the ability to configure an optimum system based on cost, performance and reliability, both by interconnection and, when necessary, constructing new components. Since users build special hardware, a computer should be easily interfaced. As a by-product of modularity, computer components can be produced and stocked, rather than tailor-made on order. The physical structure is almost identical to the PMS structure discussed in the following section; thus, reasonably large building blocks are available to the user.

Microprogramming

A note on microprogramming is in order because of current interest in the "firmware" concept. We believe microprogramming, as we understand it (Wilkes, 1951), can be a worthwhile technique as it applies to processor design. For example, microprogramming can probably be used in larger computers when floating point data operators are needed. The IBM System/360 has made use of the technique for defining processors that interpret both the System/360 instruction set and earlier family instruction sets (e.g., 1401, 1620, 7090). In the PDP-11 the basic instruction set is quite straightforward and does not necessitate microprogrammed interpretation. The processor-memory connection is asynchronous and therefore memory of any speed can be connected. The instruction set encourages the user to write reentrant programs; thus, read-only memory can be used as part of primary memory to gain the permanency and performance normally attributed to microprogramming. In fact, the Model 10 computer which will not be further discussed has a 1024-word read only memory, and a 128-word read-write memory.

Understandability

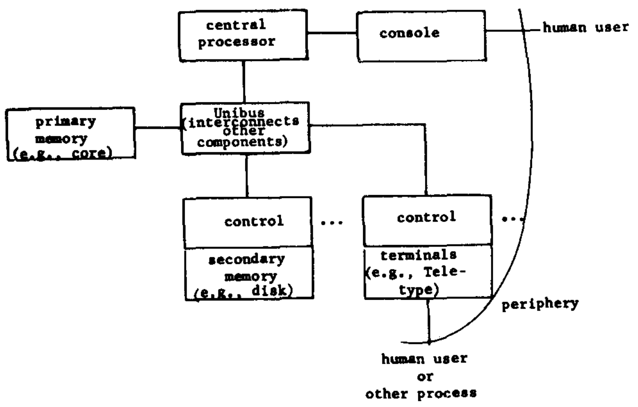
Understandability was perhaps the most fundamental constraint (or goal) although it is now somewhat less important to have a machine that can be quickly understood by a novice computer user than it was a few years ago. DEC's early success has been predicated on selling to an intelligent but inexperienced user. Understandability, though hard to measure, is an important goal because all (potential) users must understand the computer. A straightforward design should simplify the systems programming task; in the case of a compiler, it should make translation (particularly code generation) easier.

PDP-11 STRUCTURE AT THE PMS LEVEL*

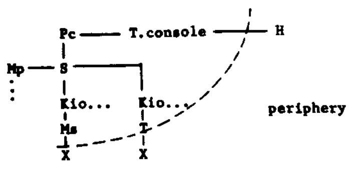
Introduction

PDP-11 has the same organizational structure as nearly all present day computers (Figure 1). The primitive PMS components are: the primary memory (Mp) which holds the programs while the central processor (Pc) interprets them; io controls (Kio) which manage data transfers between terminals (T) or secondary memories (Ms) to primary memory (Mp); the components outside the computer at periphery (X) either humans (H) or some external process (e.g., another computer); the processor console (T. console) by which humans communicate with the computer and observe its behavior and affect changes in its state; and a switch (S) with its control (K) which allows all the other components to communicate with one another. In the case of PDP-11, the central logical switch structure is implemented using a bus or chained switch (S) called the Unibus, as shown in Figure 2. Each physical component has a switch for placing messages on the bus or taking messages off the bus. The central control decides the next component to

* A descriptive (block-diagram) level (Bell and Newell, 1970) to describe the relationship of the computer components: processors memories, switches, controls, links, terminals and data operators.


central processor
console
human user
primary memory (e.g., core)
Unibus (interconnects other components)
control
control
secondary memory (e.g., disk)
terminals (e.g., Teletype)
periphery
human user or other process

Conventional block diagram



PMS diagram[1]

[1]PMS Notation

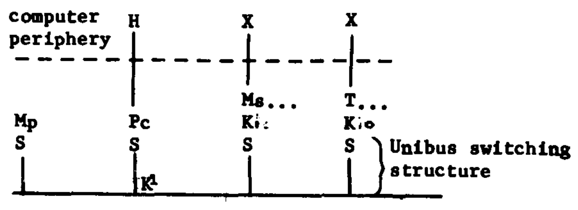
form	comment
Component/X	name X is an alias (abbreviation for a component is separated by /)
a := b	a is assigned the meaning of b
|	delimits mutually exclusive alternatives
Components := (Processor/P | Memory/M|Switch/S| Control/K|Terminal/T| Data operation/D|Link/L| Human/H)	set of primitive components and their abbreviations
X(a_1:v_1; a_2:v_2; ... a_n:v_b)	n attribute/a, value/v pairs. Attribute may be omitted if it can be inferred from dimensions of value.
index number/#	attribute giving component number
name/'	attribute giving component name
miscellaneous abbreviations := (Mp/primary memory|Ms/secondary memory|Pc/central processor| Kio/io control|Pio/io processor|s/sec/seconds|char/character| b/bit|w/word|i/information|	
———	information carrying link (bi-directional)
→ ←	uni-directional information carrying links
	delimits alternatives

Figure 1—Conventional block diagram and PMS diagram of PDP-11

use the bus for a message (call). The S (Unibus)differs from most switches because any component can communicate with any other component.

The types of messages in the PDP-11 are along the lines of the hierarchical structure common to present day computers. The single bus makes conventional and other structures possible. The message processes in the structure which utilize S(Unibus) are:

1. The central processor (Pc) requests that data be read or written from or to primary memory (Mp) for instructions and data. The processor calls a particular memory module by concurrently specifying the module's address, and the address within the modules. Depending on whether the processor requests reading or writing, data is transmitted either from the memory to the processor or vice versa.
2. The central processor (Pc) controls the initialization of secondary memory (Ms) and terminal (T) activity. The processor sets status bits in the control associated with a particular Ms or T, and the device proceeds with the specified action (e.g., reading a card, or punching a character into paper tape). Since some devices transfer data vectors directly to primary memory, the vector control information (i.e., the memory location and length) is given as initialization information.
3. Controls request the processor's attention in the form of interrupts. An interrupt request to the processor has the effect of changing the state of the processor; thus the processor begins executing a program associated with the interrupting process. Note, the interrupt process is only a signaling method, and when the processor interruption occurs, the interruptee specifies a unique address value to the processor. The address is a starting address for a program.
4. The central processor can control the transmission of data between a control (for T or Ms) and either the processor or a primary memory for program controlled data transfers. The device signals for attention using the interrupt dialogue and the central processor responds by managing the data transmission in a fashion similar to transmitting initialization information.



[1] Unibus control packaged with Pc

Figure 2—PDP-11 physical structure PMS diagram

5. Some device controls (for T or Ms) transfer data directly to/from primary memory without central processor intervention. In this mode the device behaves similar to a processor; a memory address is specified, and the data is transmitted between the device and primary memory.
6. The transfer of data between two controls, e.g., a secondary memory (disk) and say a terminal/T. display is not precluded, provided the two use compatible message formats.

As we show more detail in the structure there are, of course, more messages (and more simultaneous activity). The above does not describe the shared control and its associated switching which is typical of a magnetic tape and magnetic disk secondary memory systems. A control for a DECtape memory (Figure 3) has an S('DECtape bus) for transmitting data between

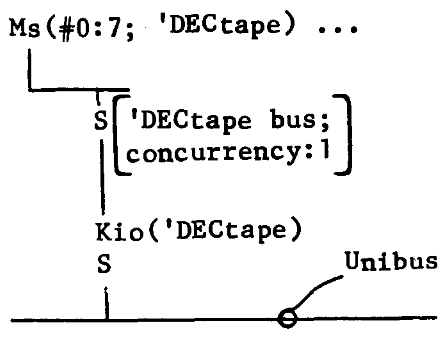


Figure 3—DECtape control switching PMS diagram

a single tape unit and the DECtape transport. The existence of this kind of structure is based on the relatively high cost of the control relative to the cost of the tape and the value of being able to run concurrently with other tapes. There is also a dialogue at the periphery between X-T and X-Ms which does not use the Unibus. (For example, the removal of a magnetic tape reel from a tape unit or a human user (H) striking a typewriter key are typical dialogues.)

All of these dialogues lead to the hierarchy of present computers (Fig. 4). In this hierarchy we can see the paths by which the above messages are passed (Pc-Mp; Pc-K; K-Pc; Kio-T and Kio-Ms; and Kio-Mp; and, at the periphery, T-X and T-Ms; and T.console-H).

Model 20 implementation

Figure 5 shows the detailed structure of a uniprocessor, Model 20 PDP-11 with its various

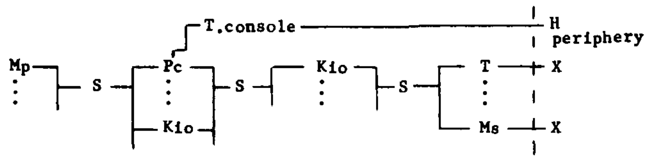
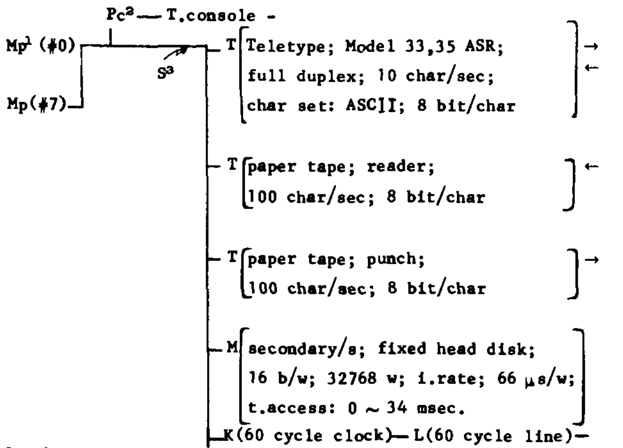


Figure 4—Conventional hierarchy computer structure

components (options). In Figure 5 the Unibus characteristics are surpressed. (The detailed properties of the switch are described in the logical design section.)

Extensions to increase performance

The reader should note (Figure 5) that the important limitations of the bus are: a concurrency of one, namely, only one dialogue can occur at a given time, and a maximum transfer rate of one 16-bit word per .75 μsec., giving a transfer rate of 21.3 megabits/second. While the bus is not a limit for a uni-processor structure, it is a limit for multiprocessor structures. The bus also imposes an artificial limit on the system performance when high speed devices (e.g., TV cameras, disks) are



Pc²— T.console -
Mp¹ (#0)
Mp(#7)
S³
T Teletype; Model 33,35 ASR; full duplex; 10 char/sec; char set: ASCII; 8 bit/char
T paper tape; reader; 100 char/sec; 8 bit/char
T paper tape; punch; 100 char/sec; 8 bit/char
M secondary/s; fixed head disk; 16 b/w; 32768 w; i.rate; 66 μs/w; t.access: 0 ~ 34 msec.
K(60 cycle clock)— L(60 cycle line)—

¹Mp(technology: core; 4096 words; t.cycle: 1.2 μs; t.access: .6 μs; 16 bits/word)

²P(central/c; Model 30; integrated circuit; general registers; 2 addresses/instruction; addresses are: register, stack, Mp; data types: bits, bytes, words, word integers, byte integers, boolean vectors; 8 bits/byte; 16 bits/word operations: (+, -, / (optional), x (optional), /2, x2, ¬, - (negate); ∨, ⊃);
M(processor state; 'general registers; 8 + 1 word; integrated circuit))

³S('Unibus; non-hierarchy; bus; concurrency:1; 1 word/.75 μs)

Figure 5—PDP-11 structure and characteristics PMS diagram

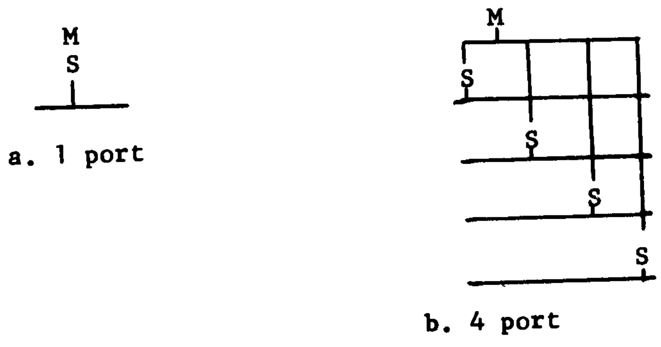


Figure 6—1 and 4 port memory modules PMS diagram

transferring data to multiple primary memories. On a larger system with multiple independent memories the supply of memory cycles is 17 megabits/second times the number of modules. Since there is such a large supply of memory cycles/second and since the central processor can only absorb approximately 16 megabits/second, the simple one Unibus structure must be modified to make the memory cycles available. Two changes are necessary: first, each of the memory modules have to be changed so that multiple units can access each module on an independent basis; and second, there must be independent control accessing mechanisms. Figure 6 shows how a single memory is modified to have more access ports (i.e., connect to 4 Unibusses).

Figure 7 shows a system with 3 independent memory modules which are accessed by 2 independent Unibusses. Note that two of the secondary memories and one of the transducers are connected to both Unibusses. It should be noted that devices which can potentially interfere with Pc-Mp accesses are constructed with two ports; for simple systems, the two ports are both connected to the same bus, but for systems with more busses, the second connection is to an independent bus.

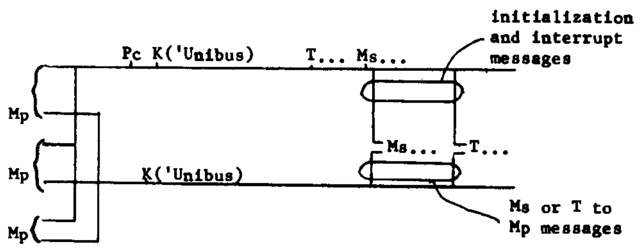


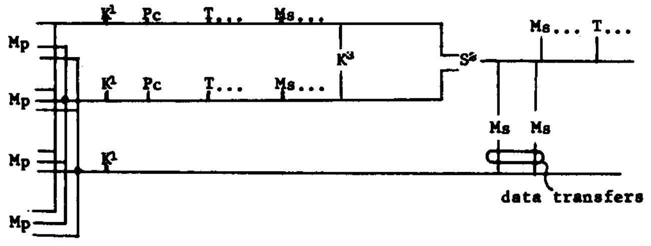
Figure 7—Three Mp, 2 S('Unibus) structure PMS diagram

Figure 8 shows a multiprocessor system with two central processors and three Unibusses. Two of the Unibus controls are included within the two processors, and the third bus is controlled by an independent control unit. The structure also has a second switch to allow either of two processors (Unibusses) to access common shared devices. The interrupt mechanism allows either processor to respond to an interrupt and similarly either processor may issue initialization information on an anonymous basis. A control unit is needed so that two processors can communicate with one another; shared primary memory is normally used to carry the body of the message. A control connected to two Pc's (see Figure 8) can be used for reliability; either processor or Unibus could fail, and the shared Ms would still be accessible.

Higher performance processors

Increasing the bus width has the greatest effect on performance. A single bus limits data transmission to 21.4 megabits/second, and though Model 20 memories are 16 megabits/second, faster (or wider) data path width modules will be limited by the bus. The Model 20 is not restricted, but for higher performance processors operating on double word (fixed point) or triple word (floating point) data two or three accesses are required for a single data type. The direct method to improve the performance is to double or triple the primary memory and central processor data path widths. Thus, the bus data rate is automatically doubled or tripled.

For 32- or 48-bit memories a coupling control unit is needed so that devices of either width appear isomorphic to one another. The coupler maps a data



[1] K('Unibus)
[2] S('Unibus Multiple bus to single bus coupler; from: 2 Unibus; to: 1 Unibus)
[3] K('Processor to processor coupler)
[4] Ms(duplex)

Figure 8—Dual Pc multiprocessor system PMS diagram

request of a given width into a higher- or lower-width request for the bus being coupled to, as shown in Figure 9. (The bus is limited to a fixed number of devices for electrical reasons; thus, to extend the bus a bus repeating unit is needed. The bus repeating control unit is almost identical to the bus coupler.) A computer with a 48-bit primary memory and processor and 16-bit secondary memory and terminals (transducers) is shown in Figure 9.

In summary, the design goal was to have a modular structure providing the final user with freedom and flexibility to match his needs. A secondary goal of the Unibus is open-endedness by providing multiple busses and defining wider path busses. Finally, and most important, the Unibus is straightforward.

THE INSTRUCTION SET PROCESSOR (ISP) LEVEL-ARCHITECTURE*

Introduction, background and design constraints

The Instruction Set Processor (ISP) is the machine defined by hardware and/or software which interprets programs. As such, an ISP is independent of technology and specific implementations.

The instruction set is one of the least understood aspects of computer design; currently it is an art. There is currently no theory of instruction sets, although there have been attempts to construct them (Maurer, 1966), and there has also been an attempt to have a computer program design an instruction set (Haney, 1968). We have used the conventional approach in this design: first a basic ISP was adopted and then incremental design modifications were made (based on the results of the benchmarks).**

* The word architecture has been operationally defined (Amdahl, Blaauw and Brooks, 1964) as "the attributes of a system as seen by a programmer, i.e., the conceptual structure and functional behavior, as distinct from the organization of the data flow and controls, the logical design and the physical implementation."

** A predecessor multiregister computer was proposed which used a similar design process. Benchmark programs were coded on each of 10 "competitive" machines, and the object of the design was to get a machine which gave the best score on the benchmarks. This approach had several fallacies: the machine had no basic character of its own; the machine was difficult to program since the multiple registers were assigned to specific functions and had inherent idiosyncrasies to score well on the benchmarks; the machine did not perform well for programs other than those used in the benchmark test; and finally, compilers which took addvantage of the machine appeared to be difficult to write. Since all "competitive machines" had been hand-coded from a common flowchart rather than separate flowcharts for each machine, the apparent high performance may have been due to the flowchart organization.

Although the approach to the design was conventional, the resulting machine is not. A common classification of processors is as zero-, one-, two-, three-, or three-plus-one-address machines. This scheme has the the form:

$$op\ l1,\ l2,\ l3,\ l4$$

where $l1$ specifies the location (address) in which to store the result of the binary operation (op) of the contents of operand locations $l2$ and $l3$, and $l4$ specifies the location of the next instruction.

The action of the instruction is of the form:

$$l1 \leftarrow l2\ op\ l3;\ goto\ l4$$

The other addressing schemes assume specific values for one or more of these locations. Thus, the one-address von Neumann (Burks, Goldstine and von Neumann, 1946) machines assume $l1 = l2 =$ the "accumulator" and $l4$ is the location following that of the current instruction. The two-address machine assumes $l1 = l2$; $l4$ is the next address.

Historically, the trend in machine design has been to move from a 1 or 2 word accumulator structure as in the von Neumann machine towards a machine with accumulator and index register(s).* As the number of registers is increased the assignment of the registers to specific functions becomes more undesirable and inflexible; thus, the general-register concept has developed. The use of an array of general registers in the processor was apparently first used in the first-generation, vacuum-tube machine, PEGASUS (Elliott et al., 1956) and appears to be an outgrowth of both 1- and 2-address structures. (Two alternative structures—the early 2- and 3-address per instruction computers may be disregarded, since they tend to always access primary memory for results as well as temporary storage and thus are wasteful of time and memory cycles, and require a long instruction.) The stack concept (zero-address) provides the most efficient

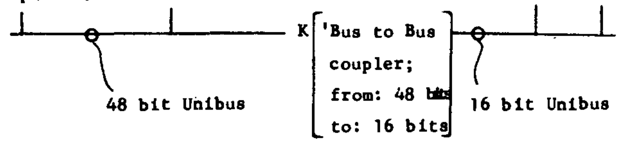


Figure 9—Computer with 48 bit Pc, Mp with 16 bit Ms, T PMS diagram

* Due in part to needs, but mainly technology which dictates how large the structure can be.

access method for specifying algorithms, since very little space, only the access addresses and the operators, needs to be given. In this scheme the operands of an operator are always assumed to be on the "top of the stack". The stack has the additional advantage that arithmetic expression evaluation and compiler statement parsing have been developed to use a stack effectively. The disadvantage of the stack is due in part to the nature of current memory technology. That is, stack memories have to be simulated with random access memories, multiple stacks are usually required, and even though small stack memories exist, as the stack overflows, the primary memory (core) has to be used.

Even though the trend has been toward the general register concept (which, of course, is similar to a two address scheme in which one of the addresses is limited to small values), it is important to recognize that any design is a compromise. There are situations for which any of these schemes can be shown to be "best". The IBM System/360 series uses a general register structure, and their designers (Amdahl, Blaauw and Brooks, 1964) claim the following advantages for the scheme:

1. Registers can be assigned to various functions: base addressing, address calculation, fixed point arithmetic and indexing.
2. Availability of technology makes the general registers structure attractive.

The System/360 designers also claim that a stack organized machine such as the English Electric KDF 9 (Allmark and Lucking, 1962) or the Burroughs B5000 (Lonegran and King, 1961) has the following disadvantages:

1. Performance is derived from fast registers, not the way they are used.
2. Stack organization is too limiting and requires many copy and swap operations.
3. The overall storage of general registers and stack machines are the same, considering point #2.
4. The stack has a bottom, and when placed in slower memory there is a performance loss.
5. Subroutine transparency is not easily realized with one stack.
6. Variable length data is awkward with a stack.

We generally concur with points 1, 2, and 4. Point 5 is an erroneous conclusion, and point 6 is irrelevant (that is, general register machines have the same problem). The general-register scheme also allows processor implementations with a high degree of parallelism since instructions of a local block all can operate on several registers concurrently. A set of truly general purpose registers should also have additional uses. For example, in the DEC PDP-10, general registers are used for address integers, indexing, floating point, boolean vectors (bits), or program flags and stack pointers. The general registers are also addressable as primary memory, and thus, short program loops can reside within them and be interpreted faster. It was observed in operation that PDP-10 stack operations were very powerful and often used ((accounting for as many as 20% of the executed instructions, in some programs, e.g., the compilers.)

The basic design decision which sets the PDP-11 apart was based on the observation that by using *truly* general registers and by suitable addressing mechanisms it was possible to consider the machine as a zero-address (stack), one-address (general register), or two-address (memory-to-memory) computer. Thus, it is possible to use whichever addressing scheme, or mixture of schemes, is most appropriate.

Another important design decision for the instruction set was to have only a few data types in the basic machine, and to have a rather complete set of operations for each data type. (Alternative designs might have more data types with few operations, or few data types with few operations.) In part, this was dictated by the machine size. The conversion between data types must be easily accomplished either automatically or with 1 or 2 instructions. The data types should also be sufficiently primitive to allow other data types to be defined by software (and by hardware in more powerful versions of the machine). The basic data type of the machine is the 16 bit integer which uses the two's complement convention for sign. This data type is also identical to an address.

PDP-11 model 20 instruction set (basic instruction set)

A formal description of the basic instruction set is given in Appendix 1 using the ISPL notation (Bell and Newell, 1970). The remainder of this section will discuss the machine in a conventional manner.

Primary memory

The primary memory (core) is addressed as either 2^{16} bytes or 2^{15} words using a 16 bit number. The linear address space is also used to access the input-output devices. The device state, data and control registers are read or written like normal memory locations.

General register

The general registers are named: R[0:7]⟨15:0⟩*; that is, there are 8 registers each with 16 bits. The naming is done starting (at the left with bit 15 (the sign bit) to the least significant bit 0. There are synonyms for R[6] and R[7]:

Stack Pointer/SP⟨15:0⟩ := R[6]⟨15:0⟩
used to access a special stack which is used to store the state of interrupts, traps and subroutine calls

Program Counter/PC⟨15:0⟩ := R[7]⟨15:0⟩
points to the current instruction being interpreted. It will be seen that the fact that PC is one of the general registers is crucial to the design.

Any general register, R[0:7], can be used as a stack pointer. The special Stack Pointer (SP) has additional properties that force it to be used for changing processor state interrupts, traps, and subroutine calls (It also can be used to control dynamic temporary storage subroutines.)

In addition to the above registers there are 8 bits used (from a possible 16) for processor status, called PS⟨15.0⟩ register. Four bits are the Condition Codes (CC) associated with arithmetic results; the T-bit controls tracing; and three bits control the priority of running programs Priority ⟨2:0⟩. Individual bits are mapped in PS as shown in Appendix 1.

Data types and primitive operations

There are two data lengths in the basic machine: bytes and words, which are 8 and 16 bits, respectively. The non-trivial data types are word length integers (w.i.); byte length integers (by .i); word length boolean vectors (w.bv), i.e., 16 independent bits (booleans) in a 1 dimensional array; and byte length boolean vectors (by.bv). The operations on byte and word boolean vectors are identical. Since a common use of a byte is to hold several flag bits (booleans), the operations can be combined to form the complete set of 16 operations. The logical operations are: "clear," "complement," "inclusive or," and "implication" ($x \supset y$ or $\neg x \vee y$).

There is a complete set of arithmetic operations for the word integers in the basic instruction set. The arithmetic operations are: add, subtract, multiply (optional), divide (optional), compare, add one, subtract one, clear, negate, and multiply and divide by powers of two (shift). Since the address integer size is 16 bits, these data types are most important. Byte length integers are operated on as words by moving them to the general registers where they take on the value of word integers. Word length integer operations are carried out and the results are returned to memory (truncated).

The floating point instructions defined by software (not part of the basic instruction set) require the definition of two additional data types (of length two and three), i.e., double word (d.w.) and triple (t.w.) words. Two additional data types, double integer (d.i.) and triple floating point (t.f. or f) are provided for arithmetic. These data types imply certain additional operations and the conversion to the more primitive data types.

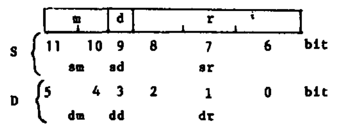
Address (operand) calculation

The general methods provided for accessing operands are the most interesting (perhaps unique) part of the machine's structure. By defining several access methods to a set of general registers, to memory, or to a stack (controlled by a general register), the computer is able to be a 0, 1 and 2 address machine. The encoding of the instruction Source (S) fields and Destination (D) fields are given in Fig. 10 together with a list of the various access modes that are possible. (Appendix 1 gives a formal description of the effective address calculation process.)

It should be noted from Figure 10 that all the common access modes are included (direct, indirect, immediate, relative, indexed, and indexed indirect) plus several relatively uncommon ones. Relative (to PC) access is used to simplify program loading, while immediate mode speeds up execution. The relatively uncommon access modes, auto-increment and auto-decrement, are used for two purposes: access to a stack under control of the registers* and access to bytes or words organized as strings or vectors. The indirect access mode allows a stack to hold addresses of data (instead of data). This mode is desirable when manipulating longer and variable-length data types (e.g., strings, double fixed and triple floating point). The register auto increment mode may be used to access a byte string; thus, for example, after each access, the register can be made to point to the next data item. This is used for moving data blocks, searching for particular elements of a vector, and byte-string operations (e.g., movement, comparisons, editing).

* A definition of the ISP notation used here may be found in Appendix 1.

*Note, by convention a stack builds toward register 0, and when the stack crosses 400_8, a stack overflow occurs.



r = register specification R[r]
d = defer (indirect) address bit
m = mode (00 = R[r]; 01 = R[r]; next R[r] +ai;[1]
    10 = R[r], R[r] -ai, next R[2]
    11 = indexed with next word)

The following access modes can be specified:

0 direct-to a register, R[r]
1 indirect-to a register, R[r] for address of data

2 auto increment via register (pop) - use register as address, then increment register
3 auto increment via register (pop) - defer
4 auto decrement via register (push) - decrement register, then use register as address
5 auto decrement indirect - decrement register, then use register as the address of the address of data

2 immediate data - next full word is the data (r=PC)
3 direct data - next full word is the address of data (r=PC)

6 direct indexed - use next full word indexed with R[r] as address of data
7 direct indexed - indirect - use next full word indexed with R[r] as the address of the address of data
6 relative access - next full word plus PC is the address (r=PC)
7 relative indirect access - next full word plus PC is the address of the address of data (r=PC)

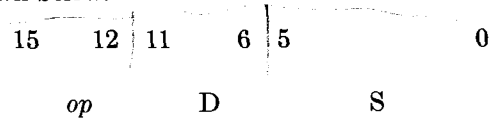
[1]address increment/ai value is 1 or 2

Figure 10—Address calculation formats

This addressing structure provides flexibility while retaining the same, or better, coding efficiency than classical machines. As an example of the flexibility possible, consider the variations possible with the most trivial word instruction MOVE (see Figure 11). The MOVE instruction is coded as it would appear in conventional 2-address, 1-address (general register) and 0-address (stack) computers. The two-address format is particularly nice for MOVE, because it provides an efficient encoding for the common operation: A ← B (note, the stack and general registers are not involved). The vector move A[I] ← B(I) is also efficiently encoded. For the general register (and 1-address format), there are about 13 MOVE operations that are commonly used. Six moves can be encoded for the stack (about the same number found in stack machines).

Instruction formats

There are several instruction decoding formats depending on whether 0, 1, or 2 operands have to be explicitly referenced. When 2 operands are required, they are identified as Source/S and Destination/D and the result is placed at Destination/D. For single operand instructions (unary operators) the instruction action is D ← u D; and for two operand instructions (binary operators) the action is D ← D b S (where u and b are unary and binary operators, e.g., ¬, − and +, −, ×, /, respectively. Instructions are specified by a 16-bit word. The most common binary operator format (that for operations requiring two addresses) is shown below.



The other instruction formats are given in Figure 12.

Instruction interpretation process

The instruction interpretation process is given in Figure 13, and follows the common fetch-execute cycle. There are three major states: (1) interrupting—the PC and PS are placed on the stack accessed by the Stack Pointer/SP, and the new state is taken from an address specified by the source requesting the trap or interrupt; (2) trace (controlled by T-bit)—essentially one instruction at a time is executed as a trace

Assembler Format	Effect	Description
Two Address Machine format:		
MOVE B,A[1]	A ← B	replace A with contents of B
MOVE #N,A	A ← N	replace A with number, N
MOVE B(RZ), A(RZ)	A[I] ← B[I]	replace element of a connector
MOVE (R_3) +, (R_4) +	A[I] ← B[I]; I ← I + 1	replace element of a vector, move to next element
General Register Machine format:		
MOVE A,R1	R1 ← A	load register
MOVE R1, A	A ← R1	store register
MOVE @A,R1	R1 ← M[A]	load or store indirect via element A
MOVE R1, R3	R1 ← R3	register to register transfer
MOVE R1, A(RZ)	A[I] ← R1	store indexed (load indexed) (or store)
MOVE @A(R0),R1	R1 ← M[A[I]]	load (or store) indexed indirect
MOVE (R1), R3	R1 ← M[R2]	load indirect via register
MOVE (R1) +, R3	R3 ← M[I]	load (or store) element indirect via register, move to next element
Stack Machine format:		
MOVE #N, -(R0)	S ← N	load stack with literal
MOVE A, -(R0)	S ← A	load stack with contents of A
MOVE @(R0)+, -(R0)	S ← M[S]	load stack with memory specified by top of stack
MOVE (R0)+, A	A ← S	store stack in A
MOVE (R0)+, @(R0)+	$M[S_2] \leftarrow S_1$	store stack top in memory addressed by stack top -1
MOVE (R0), -(R0)	S ← S	duplicate top of stack

[1] Assembler format:
() denotes contents of memory addressed by
- decrement register first
+ increment register after
@ indirect
# literal

Figure 11—Coding for the MOVE instruction to compare with conventional machines

```
Binary arithmetic and logical operations: | bop | S | D |¹
      form:  D ← S  b  D
      example:  ADD (:=bop=0010) → (CC,D ← D+S);

Unary arithmetic and logical operation: | uop | D |
      form:  D ← u D;
      examples:  NEG (:=uop=0000101100) → (CC,D ← - D) - negate
                 ASL (:=uop=0000110011) → (CC,D ← D x 2); shift left

Branch (relative) operators: | brop | offset |
      form:  if brop condition then (PC ← PC + offset);
      example:  BEQ (: = brop = 03₁₆) (Z → (PC ← PC + offset)):

Jump: | 0 000 000  001 | D |
      form:  PC ← D + Pc
Jump to subroutine: | 0 000 100 | D |
      save  R[sr] on stack, enter subroutine at D + PC

Misc. operations: | op code |
      form:  ST ← f
      example:  HALT (: = instruction = 0) → (RUN ← 0);

¹Note:  these instructions are all 1 word.  D and/or S may each require 1
      additional immediate data or address word.  Thus instructions can
      be 1, 2, or 3 words long.
```

Figure 12—PDP-11 instruction formats (simplified)

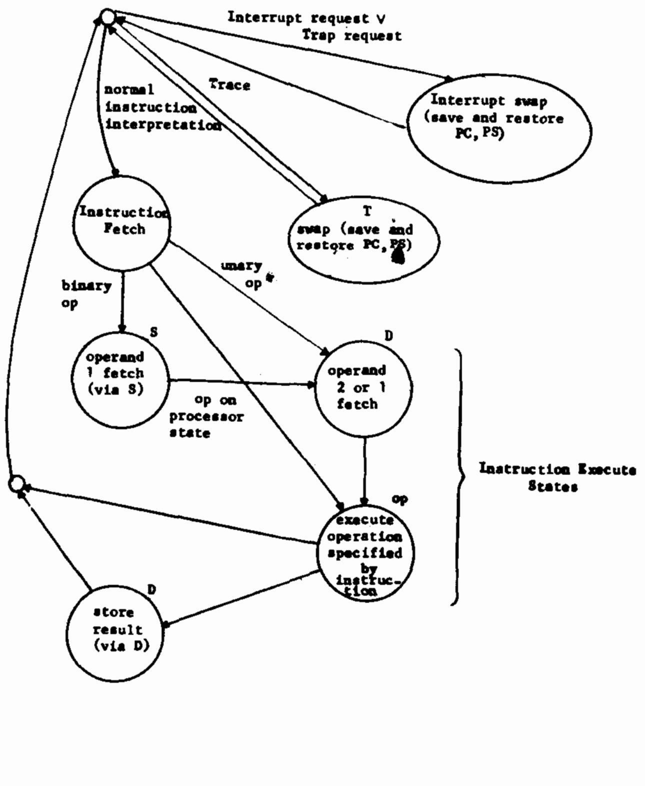

Interrupt request V
Trap request
normal instruction interpretation
Trace
Interrupt swap (save and restore PC, PS)
Instruction Fetch
T swap (save and restore PC, PS)
unary op
binary op
S
D
operand 1 fetch (via S)
operand 2 or 1 fetch
op on processor state
Instruction Execute States
op
execute operation specified by instruction
D
store result (via D)

Figure 13—PDP-11 instruction interpretation process state diagram

trap occurs after each instruction, and (3) normal instruction interpretation. The five (lower) states in the diagram are concerned with instruction fetching, operand fetching, executing the operation specified by the instruction and storing the result. The non-trivial details for fetching and storing the operands are not shown in the diagram but can be constructed from the effective address calculation process (Appendix 1). The state diagram, though simplified, is similar to 2- and 3-address computers, but is distinctly different than a 1 address (1 accumulator) computer.

The ISP description (Appendix 1) gives the operation of each of the instructions, and the more conventional diagram (Fig. 12) shows the decoding of instruction classes. The ISP description is somewhat incomplete; for example, the add instruction is defined as: ADD (:= bop = 0010) → (CC,D ← D + S); *addition* does not exactly describe the changes to the Condition Codes/CC (which means whenever a binary opcode [bop] of 0010_2 occurs the ADD instruction is executed with the above effect). In general, the CC are based on the result, that is, Z is set if the result is zero, N if negative, C if a carry occurs, and V if an overflow was detected as a result of the operation. Conditional branch instructions may thus follow the arithmetic instruction to test the results of the CC bits.

Examples of addressing schemes

Use as a stack (zero address) machine

Figure 14 lists typical zero-address machine instructions together with the PDP-11 instructions which perform the same function. It should be noted that translation (compilation) from normal infix expressions to reverse Polish is a comparatively trivial task. Thus, one of the primary reasons for using stacks is for the evaluation of expressions in reverse Polish form.

Consider an assignment statement of the form

$$D \leftarrow A + B/C$$

which has the reverse Polish form

$$DABC/+ \leftarrow$$

and would normally be encoded on a stack machine as follows

load stack address of D
load stack A
load stack B
load stack C
/
+
store

Common stack instruction:	Equivalent PDP-11 instruction:
place address value A on stack	MOVE #A, -(R0)[1]
load stack from memory address specified by stack	MOVE @(R0)+, - (R0)
load stack from memory location A	MOVE A, -(R0)
store stack at memory address specified by stack	MOVE (R0)+, @(R0)+
store stack at memory location A	MOVE (R0)+, A
duplicate top of stack	MOVE (R0), -(R0)
+ , add 2 top data of stack to stack	ADD (R0) +, @R0
-, x, /; subtract, multiply, divide	(see add)
-; negate top data of stack	NEG @R0
clear top data of stack	CLR @R0
∨; "inclusive or" 2 top data of stack "and" 2 top data of stack	BSET (R0)+, @R0
¬; complement top of stack	COM @R0
test top of stack (set branch indicators)	TST @R0
branch on indicator	BR (=, ≠, >, ≥, <, ≤)
jump unconditional	JUMP
add addressed location A to top of stack - (not common for stack machine) equivalent	ADD A, @R0
to: load stack, add swap top 2 stack data	MOVE (R0)+, R1 MOVE (R0)+, R2 MOVE R1, -(R0) MOVE R2, -(R0)
reset stack location to N	MOVE #N, R0
∧, "and" 2 top stack data	COM @R0 BCLR (R0)+, @R0

[1]Stack pointer has been arbitrarily used as register R0 for this example.

Figure 14—Stack computer instructions and equivalent PDP-11 instructions

However, with the PDP-11 there is an address method for improving the program encoding and run time, while not losing the stack concept. An encoding improvement is made by doing an operation to the top of the stack from a direct memory location (while loading). Thus the previous example could be coded as:

load stack B
divide stack by C
add A to stack
store stack D

Use as a one-address (general register) machine

The PDP-11 is a general register computer and should be judged on that basis. Benchmarks have been coded to compare the PDP-11 with the larger DEC PDP-10. A 16 bit processor performs better than the DEC PDP-10 in terms of bit efficiency, but not with time or memory cycles. A PDP-11 with a 32 bit wide memory would, however, decrease time by nearly a factor of two, making the times essentially comparable.

Use as a two-address machine

Figure 15 lists typical two-address machine instructions together with the equivalent PDP-11 instructions for performing the same operations. The most useful instruction is probably the MOVE instruction because it does not use the stack or general registers. Unary instructions which operate on and test primary memory are also useful and efficient instructions.

Extensions of the instruction set for real (floating point) arithmetic

The most significant factor that affects performance is whether a machine has operators for manipulating data in a particular format. The inherent generality of a stored program computer allows any computer by subroutine to simulate another—given enough time and memory. The biggest and perhaps only factor that separates a small computer from a large computer is whether floating point data is understood by the computer. For example, a small computer with a cycle time of 1.0 microseconds and 16 bit memory width might have the following characteristics for a floating point add, excluding data accesses:

programmed:	250 microseconds
programmed (but special normalize and differencing of exponent instructions):	75 microseconds
microprogrammed hardware:	25 microseconds
hardwired:	2 microseconds

It should be noted that the ratios between programmed and hardwired interpretation varies by roughly two orders of magnitude. The basic hardwiring scheme and the programmed scheme should allow binary program compatibility, assuming there is an interpretive program for the various operators in the Model 20. For example, consider one scheme which would add eight 48 bit registers which are addressable in the extended instruction set. The eight floating registers, F, would be mapped into eight double length

Two Address Computer	PDP-11
A ← B; transfer B to A	MOVE B,A
A ← A+B; add	ADD B,A
-, x, /	(see add)
A ← -A; negate	NEG A
A ← A ∨ B; inclusive or	BSETB,A
A ← ¬ A; not	COM
Jump unconditioned	JUMP
Test A, and transfer to B	TST A
	BR (=, ≠, >, ≥, <, ≤) B

Figure 15—Two address computer instructions and equivalent PDP-11 instructions

(32 bit) registers, D. In order to access the various parts of F or D registers, registers F0 and F1 are mapped onto registers R0 to R2 and R3 to R5.

Since the instruction set operation code is almost completely encoded already for byte and word length data, a new encoding scheme is necessary to specify the proposed additional instructions. This scheme adds two instructions: enter floating point mode and execute one floating point instruction. The instructions for floating point and double word data would be:

binary ops	op	floating point/f	and double word/d
bop′ S D	←	FMOVE	DMOVE
	+	FADD	DADD
	−	FSUB	DSUB
	×	FMUL	DMUL
	/	FDIV	DDIV
	compare	FCMP	DCMP
unary ops			
uop′ D	−	FNEG	DNEG

LOGICAL DESIGN OF S(UNIBUS) AND PC

The logical design level is concerned with the physical implementation and the constituent combinatorial and sequential logic elements which form the various computer components (e.g., processors, memories, controls). Physically, these components are separate and connected to the Unibus following the lines of the PMS structure.

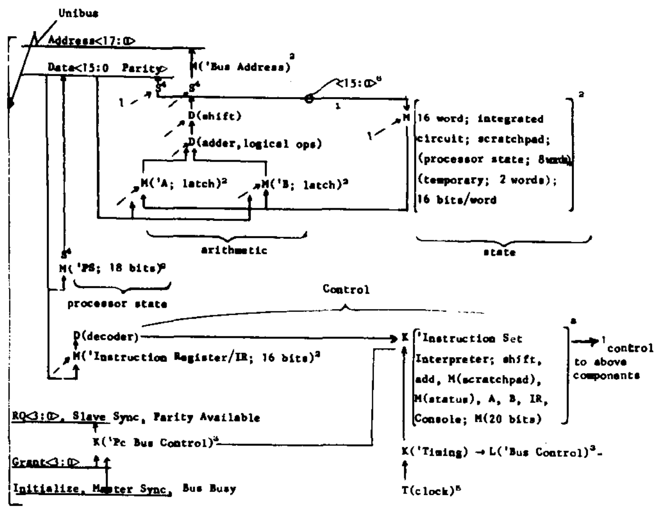
Unibus organization

Figure 16 gives a PMS diagram of the Pc and the entering signals from the Unibus. The control unit for the Unibus, housed in Pc for the Model 20, is not shown in the figure.

The PDP-11 Unibus has 56 bi-directional signals conventionally used for program-controlled data transfers (processor to control), direct-memory data transfers (processor or control to memory) and control-to-processor interrupt. The Unibus is interlocked; thus transactions operate independent of the bus length and response time of the master and slave. Since the bus is bi-directional and is used by all devices, any device can communicate with any other device. The controlling device is the master, and the device to which the master is communicating is the slave. For example, a data transfer from processor (master) to memory (always a slave) uses the Data Out dialogue facility for writing and a transfer from memory to processor uses the Data In dialogue facility for reading.

Bus control

Most of the time the processor is bus master fetching instructions and operands from memory and storing results in memory. Bus mastership is determined by the current processor priority and the priority line upon which a bus request is made and the physical placement of a requesting device on the linked bus.


Unibus
Address<17:0>
Data<15:0 Parity>
M('Bus Address)²
S⁴
<15:0>⁶
D(shift)
D(adder, logical ops)
M('A; latch)²
M('B; latch)²
M [16 word; integrated circuit; scratchpad; (processor state; 8 words) (temporary; 2 words); 16 bits/word]²
arithmetic
state
M('PS; 18 bits)²
processor state
Control
D(decoder)
M('Instruction Register/IR; 16 bits)²
K ['Instruction Set Interpreter; shift, add, M(scratchpad), M(status), A, B, IR, Console; M(20 bits)]
control to above components
RQ<3:0>, Slave Sync, Parity Available
K('Pc Bus Control)³
Grant<3:0>
Initialize, Master Sync, Bus Busy
K('Timing) → L('Bus Control)³
T(clock)⁵

¹D - data operations to perform shifting and adding - usually combinatorial
²M - memories of a word (16 bits) or several words
³K - control units; sequential switching circuits
⁴S - switch, used to gate contents of an M to a set of lines
⁵T - Transducer - to encode time into a logic (clock) signal
⁶Notation to denote 16 lines named 15,14,...,1,0

Figure 16—PDP-11 Pc structure

The assignment of bus mastership is done concurrent with normal communication (dialogues).

Unibus dialogues

Three types of dialogues use the Unibus. All the dialogues have a common protocol which first consists of obtaining the bus mastership (which is done concurrent with a previous transaction) followed by a data exchange with the requested device. The dialogues are: Interrupt; Data In and Date In Pause; and Data Out and Data Out Byte.

Interrupt

Interrupt can be initiated by a master immediately after receiving bus mastership. An address is transmitted from the master to the slave on Interrupt. Normally, subordinate control devices use this method to transmit an interrupt signal to the processor.

Data in and data in pause

These two bus operations transmit slave's data (whose address is specified by the master) to the master. For the Data In Pause operation data is read into the master and the master responds with data which is to be rewritten in the slave.

Data out and data out byte

These two operations transfer data from the master to the slave at the address specified by the master. For Data Out a word at the address specified by the address lines is transferred from master to slave. Data Out Byte allows a single data byte to be transmitted.

Processor logical design

The Pc is designed using TTL logical design components and occupies approximately eight 8″ × 12″ printed circuit boards. The organization of the logic is shown in Figure 17. The Pc is physically connected to two other components, the console and the Unibus. The control for the Unibus is housed in the Pc and occupies one of the printed circuit boards. The most regular part of the Pc, the arithmetic and state section, is shown at the top of the figure. The 16-word scratch-pad memory and combinatorial logic data operators, D(shift) and D(adder, logical ops), form the most regular part of the processor's structure. The 16-word memory holds most of the 8-word processor state found in the ISP, and the 8 bits that form the Status word are stored in an 8-bit register. The input to the adder-shift network has two latches which are either memories or gates. The output of the adder-shift network can be read to either the data or address parts of the Unibus, or back to the scratch-pad array.

The instruction decoding and arithmetic control are less regular than the above data and state and these are shown in the lower part of the figure. There are two major sections: the instruction fetching and decoding control and the instruction set interpreter (which in effect defines the ISP). The later control section operates on, hence controls, the arithmetic and state parts of the Pc. A final control is concerned with the interface to the Unibus (distinct from the Unibus control that is housed in the Pc).

CONCLUSIONS

In this paper we have endeavored to give a complete description of the PDP-11 Model 20 computer at four descriptive levels. These present an unambiguous specification at two levels (the PMS structure and the ISP), and, in addition, specify the constraints for the design at the top level, and give the reader some idea of the implementation at the bottom level logical design. We have also presented guidelines for forming additional models that would belong to the same family.

ACKNOWLEDGMENTS

The authors are grateful to Mr. Nigberg of the technical publication department at DEC and to the reviewers for their helpful criticism. We are especially grateful to Mrs. Dorothy Josephson at Carnegie-Mellon University for typing the notation-laden manuscript.

REFERENCES

1 R H ALLMARK J R LUCKING
Design of an arithmetic unit incorporationg a nesting store
Proc IFIP Congress pp 694–698 1962

2 G M AMDAHL G A BLAAUW F P BROOKS JR
Architecture of the IBM System/360
IBM Journal Research and Development Vol 8 No 2 pp 87–101 April 1964

3 C G BELL A NEWELL
Computer structures
McGraw-Hill Book Company Inc New York In press 1970

4 A W BURKS H H GOLDSTINE J VON NEUMANN
Preliminary discussion of the logical design of an electronic computing instrument, Part II
Datamation Vol 8 No 10 pp 36–41 October 1962

5 W S ELLIOTT C E OWEN C H DEVONALD B G MAUDSLEY
The design philosophy of Pegasus, a quantity-production computer
Proceedings IEEE Pt. B 103 Supp 2 pp 188–196 1956

6 F M HANEY
Using a computer to design computer instruction sets
Thesis for Doctor of Philosophy degree College of Engineering and Science Department of Computer Science Carnegie-Mellon University Pittsburgh Pennsylvania May 1968

7 W LONERGAN P KING
Design of the B5000 system
Datamation Vol 7 No 5 pp 28–32 May 1961

8 W D MAURER
A theory of computer instructions
Journal of the ACM Vol 13 No 2 pp 226–235 April 1966

9 S ROTHMAN
R/W 40 data processing system
International Conference on Information Processing and Auto-math 59 Ramo-Wooldridge (A division of Thompson Ramo Wooldridge Inc) Los Angeles California June 1959

10 M V WILKES
The best way to design an automatic calculating machine
Report of Manchester University Computer Inaugural Conference July 1951 (Manchester 1953)

APPENDIX 1

DEC PDP–11 instruction set processor Description (in ISPL*)

The following description is not a detailed description of the instructions. The description omits the trap behavior of unimplemented instructions, references to non-existent primary memory and io devices, SP (stack) overflow, and power failure.

Primary Memory State	
M/Mb/Memory$[0:2^{16}-1]\langle 7:0\rangle$	*(byte memory)*
Mw$[0:2^{15}-1]\langle 15:0\rangle := $ M$[0:2^{16}-1]\langle 7:0\rangle$	*(word memory mapping)*
Processor State (9 words)	
R/Registers$[0:7]\langle 15:0\rangle$	*(word general registers)*
SP$\langle 15:0\rangle :=$ R$[6]\langle 15:0\rangle$	*(stack pointer)*
PC$\langle 15:0\rangle :=$ R$[7]\langle 15:0\rangle$	*(program counter)*

*ISP NOTATION

Although the ISP language has not been described in publications, its syntax is similar to other languages. The language is inherently interpreted in parallel, thus to get sequential evaluation the word "next" must be used. Italics are used for comments. The following notes are in order:

$a := f(\ldots)$	equivalence or substitution process used for name and process substitution. For every occurrence of $a, f(\ldots)$ replaces it.
$a \leftarrow f(\ldots)$	Replacement operator; the contents in register a are replaced by the value of the function.
register declaration, e.g., $Q[0:1][0:4095]\langle 15:0\rangle$	an array of words of two dimensions 2 and 4096; each word has 16 bits denoted 15, 14, 13, . . ., 1, 0
$\langle a:b\rangle_n$	Denotes a range of characters $a, a+1, \ldots, b$ to base n. If n is not given, the base is 2.
$[c:d]$	Array designation $c, c+1, \ldots, d$
$a \rightarrow b;$	equivalent to ALGOL if a then b
"next"	sequential interpretation
instruction declaration, e.g., ADD (: = bop = 0010) → $(CC, D \leftarrow D + S)$	defines the "ADD" instruction, assigns it a value, and gives its operation. ADD is executed when bop = 0010_2. Equivalent to: ADD → $(CC, D \leftarrow D + S)$ where ADD: = (bop = 0010) bop has been previously declared
□	concatenation, consider the combined registers as one

operators: = (+/add | −/subtract/negate | ×/multiply | //divide | ∧/and | ∨ /or | √/not | ⊕/exclusive or | =/equal/>/greater than | ≥ | < | ≤ | ≠ | modulo | etc.)

PS⟨15:0⟩	*(processor state register)*
Priority/P⟨2:0⟩ := PS⟨7:5⟩	*(under program control; priority level of the process currently being interpreted a higher level process may interrupt or trap this process)*
CC/Condition␣Codes⟨3:0⟩ := PS⟨3:0⟩	*(under program control; when set, each instruction executed will trap; used for interpretive and breakpoint debugging)*
Carry/C := CC⟨0⟩	*(a result condition code indicating an arithmetic carry from bit 15 of the last operation)*
Negative/N := CC⟨3⟩	*(a result condition code indicating last result was negative)*
Zero/Z := CC⟨2⟩	*(a result condition code indicating last result was zero)*
Overflow/V := CC⟨1⟩	*(a result condition code indicating an arithmetic overflow of the last operation)*
Trace/T := ST⟨4⟩	*(denotes whether instruction trace trap is to occur after each instruction is executed)*
Undefined⟨7:0⟩ := PS⟨15:8⟩	*(unused)*
Run	*(denotes normal execution)*
Wait	*(denotes waiting for an interrupt)*

Instruction Format
(Bit assignments used in the various instruction formats)

i/instruction⟨15:0⟩	
bop⟨3:0⟩ := i⟨15:12⟩	*(binary operation code)*
uop⟨15:6⟩ := i⟨15:6⟩	*(unary operation code)*
brop⟨15:8⟩ := i⟨15:8⟩	*(branch operation code)*
sop⟨15:6⟩ := i⟨15:6⟩	*(shift operation code)*
s/source⟨5:0⟩ := i⟨11:6⟩	*(source control byte)*
sm⟨0:1⟩ := s⟨5:4⟩	*(source mode control)*
sd := s⟨3⟩	*(source defer bit)*
sr := s⟨2:0⟩	*(source register)*
d/destination⟨5:0⟩ := i⟨5:0⟩	*(destination control byte)*
dm⟨0:1⟩ := d⟨5:4⟩	
dd := d⟨3⟩	
dr⟨2:0⟩ := d⟨2:0⟩	
offset⟨7:0 := i⟨7:0⟩	*(signed 7 bit integer)*
address␣increment/ai	*(implicit bit derived from i to denote byte or word length operations)*

Data Types

by/byte⟨7:0⟩	
w/word⟨15:0⟩	
by.i/byte.integer⟨7:0⟩	*(signed integers)*
w.i/word.integer⟨15:0⟩	
by.bv/byte.boolean␣vector⟨7:0⟩	*(boolean vectors (bits))*
w.bv/word.boolean␣vector⟨15:0⟩	

```
  d/double␣word⟨31:0⟩                                       (*double word)
  t/triple␣word⟨47:0⟩                                       (*triple word)
  f/t.f/triple.floating␣point⟨47:0⟩                         (*triple floating point)
```

Source/S and Destination/D Calculation

```
  S/Source⟨15:0⟩ := (¬ sd → (                                                 (direct access)
     (sm = 00) → R[sr];                                                       (register)
     (sm = 01) ∧ (sr ≠ 7) → (M[R[sr]]; next R[sr] ← R[sr] + ai);              (auto increment)
     (sm = 01) ∧ (sr = 7) → (M[PC]; PC ← PC + 2);                             (immediate)
     (sm = 10) → (R[sr] ← R[sr] − ai; next M[R[sr]]);                         (auto decrement)
     (sm = 11) ∧ (sr ≠ 7) → (M[M[PC] + R[sr]]; PC ← PC + 2);                  (indexed)
     (sm = 11) ∧ (sr = 7) → (M[M[PC] + PC]; PC ← PC + 2));                    (relative)
                      sd → (                                                  (indirect access)
     (sm = 00) → M[R[sr]];                                                    (indirect via register)
     (sm = 01) ∧ (sr ≠ 7) → (M[M[R[sr]]]; next R[sr] ← R[sr] + ai);           (indirect via stack, auto decrement)
     (sm = 01) ∧ (sr = 7) → (M[M[PC]]; PC ← PC + 2);                          (direct absolute)
     (sm = 10) → (R[sr] ← R[sr] − ai; next M[R[sr]]);                         (indirect via stack, auto increments)
     (sm = 11) ∧ (sr ≠ 7) → (M[M[PC] + R[sr]]; PC ← PC + 2);                  (indirect, indexed)
     (sm = 11) ∧ (sr = 7) → (M[M[M[PC] + PC]]; PC ← PC + 2))                  (indirect relative)
```

(*The above process defines how operands are determined (accessed) from either memory or the registers. The various length operands, Db(byte), Dw(word), Dd(double) and Df(floating) are not completely defined. The Source/S and Destination/D processes are identical. In the case of jump instruction an address, D′, is used—instead of the word in location M*[*CI*].)

Instruction Interpretation Process

```
  ¬ Interrupt␣rqs ∧ Run ∧ Wait → (i ← M[PC]; PC ← PC + 2;                      (fetch)
                                  next instruction␣execution; next             (execute)
     T → (SP ← SP + 2; next                                                    (trace bit store state)
        M[SP] ← PS;
        SP ← SP + 2; next
        M[SP] ← PC;
        PC ← M[$14_8$]
        ST ← M[$16_8$]))

  Interrupt␣rq[j] ∧ (CC[j] > CC) ∧ Run → (T ← 0;                              (interrupt)
     SP ← SP + 2; next
     M[SP] ← PS;                                                               (store state and PC enter new proc-
                                                                                ess). The locations M[f(j)] are:
                                                                                reserved instruction = M[10]
                                                                                illegal instruction = M[4]
                                                                                stack overflow = M[4]
                                                                                bus errors = M[4])
     SP ← SP + 2;
     M[SP] ← PC
     PC ← M[f(j)]
     PS ← M[f(j) + 2])
```

Instruction Set and the Execution Process

(*The following instruction set will be defined briefly and is incomplete. It is intended to give the reader a simple understanding of the machine operation.*)

```
Instruction␣execution := (
  MOV(:= bop = 0001) → (CC,D ← S);                                             (move word)
  MOVB(:= bop = 1001) → (CC,Db ← Sb);                                          (move byte)
```

* not hardwired or optional

Binary Arithmetic: D ← D b S;	
ADD(: = bop = 0110) → (CC,D ← D+s);	*(add)*
SUB(: = bop = 1110) → (CC,D ← D − S);	*(subtract)*
CMP(: = bop = 0010) → (CC ← D − S);	*(word compare)*
CMPB(: = bop = 1010) → (CC ← Db − Sb);	*(byte compare)*
MUL(: = bop = 0111) → (CC,D ← D × S);	*(*multiply if D is a register then a double length operator)*
DIV(: = bop = 1111) → (CC,D ← D/S);	*(*divide, if D is a register, then a remainder is saved)*
Unary Arithmetic D ← u S;	
CLR(: = uop = 050_8) → (CC,D ← 0);	*(clear word)*
CLRB(: = uop = 1050_8) → (CC,Db ← 0);	*(clear byte)*
COM(: = uop = 051_8) → (CC,D ← ¬D);	*(complement word)*
COMB(: = uop = 1051_8) → (CC,Db ← ¬Db);	*(complement byte)*
INC(: = uop = 052_8) → (CC,D ← D + 1);	*(increment word)*
INCB(: = uop = 1052_8) → (CC,Db ← Db + 1);	*(increment byte)*
DEC(: = uop = 053_8) → (CC,D ← D − 1);	*(decrement word)*
DECB(: = uop = 1053_8) → (CC,Db ← Db − 1);	*(decrement byte)*
NEG(: = uop = 054_8) → (CC,D ← −D);	*(negate)*
NEGB(: = uop = 1054_8) → (CC,Db ← −Db)	*(negate byte)*
ADC(: = uop = 055_8) → (CC,D ← D + C);	*(add the carry)*
ADCB(: = uop = 1055_8) → (CC,Db ← Db + C);	*(add to byte the carry)*
SBC(: = uop = 056_8) → (CC,D ← D − C);	*(subtract the carry)*
SBCB(: = uop = 1056_8) → (CC,Db ← Db − C);	*(subtract from byte the carry)*
TST(: = uop = 057_8) → (CC ← D);	*(test)*
TST(: = uop = 1057_8) → (CC ← Db);	*(test byte)*
Shift operations: D ← D × 2^n;	
ROR(: = sop = 060_8) → (C□D ← C□D/2{rotate};	*(rotate right)*
RORB(: = sop = 1060_8) → (C□Db ← C□Db/2{rotate});	*(byte rotate right)*
ROL(: = sop = 061_8) → (C□D ← C□D × 2{rotate});	*(rotate left)*
ROLB(: = sop = 1061_8) → (C□Db ← C□Db × 2{rotate});	*(byte rotate left)*
ASR(: = sop = 062_8) → (CC,D ← D × 2);	*(arithmetic shift right)*
ASRB(: = sop = 1062_8) → (CC,Db ← Db/2);	*(byte arithmetic shift right)*
ASL(: = sop = 063_8) → (CC,D ← D × 2);	*(arithmetic shift left)*
ASLB(: = sop = 1063_8) → (CC,Db ← Db × 2);	*(byte arithmetic shift left)*
ROT(: = sop = 064_8) → (C□D ← D × 2^s);	*(rotate)*
ROTB(: = sop = 1064_8) → (C□Db ← D × 2^s);	*(byte rotate)*
LSH(: = sop = 065_8) → (CC,D ← D × 2^s{logical});	*(*logical shift)*
LSHB(: = sop = 1065_8) → (CC,Db ← Db × 2^s{logical});	*(*byte logical shift)*
ASH(: = sop = 066_8) → (CC,D ← D × 2^s);	*(*arithmetic shift)*
ASHB(: = sop = 1066_8) → (CC,Db ← Db × 2^s);	*(*byte arithmetic shift)*
NOR(: = sop = 067_8) → (CC,D ← normalize(D)); (R[r′] ← normalize␣exponent(D));	*(*normalize)*
NORD(: = sop = 1067_8) → (Db ← normalize(Dd); R[r′] ← normalize␣exponent(D));	*(*normalize double)*
SWAB(: = sop = 3) → (CC,D ← D⟨7:0, 15:8⟩)	*(swap bytes)*
Logical Operations	
BIC(: = bop = 0100) → (CC,D ← D ← D ∧ ¬S);	*(bit clear)*
BICB(: = bop = 1100) → (CC,Db ← Db ∨ ¬Sb);	*(byte bit clear)*
BIS(: = bop = 0101) → (CC,D ← D ∨ S);	*(bit set)*
BISB(: = bop = 1101) → (CC,Db ← Db ∨ Sb);	*(byte bit set)*
BIT(: = bop = 0011) → (CC ← D ∧ S);	*(bit test under mask)*
BITB(: = bop = 1011) → (CC ← Db ∧ Sb);	*(byte bit test under mask)*

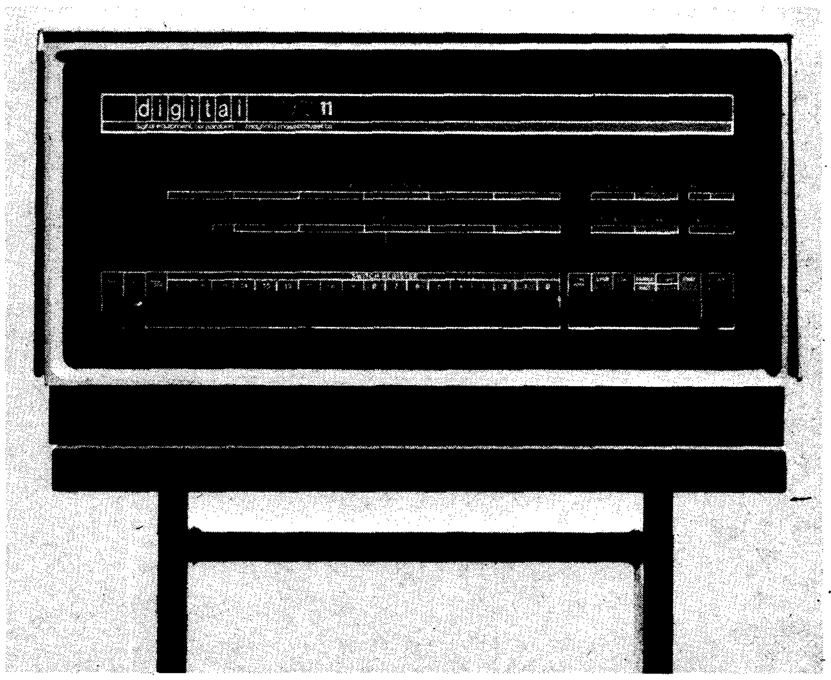
```
Branches and Subroutines Calling: PC ← f;
  JMP(:= sop = $0001_8$) → (PC ← D');                          (jump unconditional)
  BR(:= brop = $01_{16}$) → (PC ← PC + offset);                 (branch unconditional)
  BEQ(:= brop = $03_{16}$) → (Z → (PC ← PC + offset));          (equal to zero)
  BNE(:= brop = $02_{16}$) → (¬Z → (PC ← PC + offset));         (not equal to zero)
  BLT(:= brop = $05_{16}$) → (N ⊕ V → (PC ← PC + offset));      (less than (zero))
  BGE(:= brop = $04_{16}$) → (N ≡ V → (PC ← PC + offset));      (greater than or equal (zero))
  BLE(:= brop = $07_{16}$) → (Z ∨ (N ⊕ V) → (PC ← PC + offset)); (less than or equal (zero))
  BGT(:= brop = $06_{16}$) → (¬(Z ∨ (N ⊕ V)) → (PC ← PC + offset)); (less greater than (zero))
  BCS/BHIS(:= brop = $87_{16}$) → (C → (PC ← PC + offset));     (carry set; higher or same (un-
                                                                 signed))
  BCC/BLO(:= brop = $86_{16}$) → (¬C → (PC ← PC + offset));     (carry clear; lower (unsigned))
  BLOS(:= brop = $83_{16}$) → (C ∧ Z → (PC ← PC + offset));     (lower or same (unsigned))
  BHI(:= brop = $82_{16}$) → ((¬C ∨ Z) → (PC ← PC + offset));   (higher than (unsigned))
  BVS(:= brop = $85_{16}$) → (V → (PC ← PC + offset));          (overflow)
  BVC(:= brop = $84_{16}$) → (¬V → (PC ← PC + offset));         (no overflow)
  BMT(:= brop = $81_{16}$) → (N → (PC ← PC + offset));          (minus)
  BPL(:= brop = $80_{16}$) → (¬N → (PC ← PC + offset));         (plus)
  JSR(:= sop = $0040_8$) → (                                     (jump to subroutine by putting
    SP ← SP − 2; next                                            R[sr], PC on stack and loading
    M[SP] ← R[sr];                                               R[sr] with PC, and going to sub-
    R[sr] ← PC;                                                  routine at D)
      PC ← D);
  RTS(:= i = $000200_8$) → (                                     (return from subroutine)
    PC ← R[dr];
    R[dr] ← M[SP];
    SP ← SP + 2);

Miscellaneous processor state modification:
  RTI(:= i = $2_8$) → (PC ← M[SP];                               (return from interrupt)
                       SP ← SP + 2; next
                       PS ← M[SP];
                       SP ← SP + 2);
  HALT(:= i = 0) → (Run ← 0);
  WAIT(:= i = 1) → (Wait ← 1);
  TRAP(:= i = 3) → (SP ← SP + 2; next                            (trap to M[$34_8$] store status and
                    M[SP] ← PS;                                  PC)
                    SP ← SP + 2; next
                    M[SP] ← PC;                 36
                    PC ← M[$34_8$];                               (enter new process)
                    PS ← M[12]);
  EMT(:= brop − $82_{16}$) → (                                   (emulator trap)
                         SP ← SP + 2; next
                         M[SP] ← PS;
                         SP ← SP + 2; next
                         M[SP] ← PC;
                         PC ← M[$30_8$];
                         PS ← M[$32_8$]);
IOT(:= i = 4) → (see TRAP)                                       (I/O trap to M[$20_8$])
  RESET(:= i = 5) → (not described)                              (reset to external devices)
  OPERATE(:= i⟨5:15⟩ = 5) → (                                    (condition code operate)
    i⟨4⟩ → (CC ← CC ∨ i⟨3:0⟩);                                   (set codes)
    ¬i⟨4⟩ → (CC ← CC ∧ ¬ i⟨3:0⟩));                               (clear codes)
                             end Instruction␣execution
```

pdp11 handbook

digital

digital equipment corporation

TABLE OF CONTENTS



The PDP-11 is available in two versions—PDP-11/10 and PDP-11/20. The basic PDP-11/10 contains 1,024 words of read only memory in conjunction with 128 words of read/write memory and the basic PDP-11/20 includes 4,096 words of read/write memory.

CHAPTER 1

INTRODUCTION

This publication is a handbook for Digital Equipment Corporation's PDP-11. It provides a comprehensive overview of the system structure, the instruction repertoire, input/output programming, peripherals, general interfacing, software, and console operation.

PDP-11 is Digital's answer to the demand for a modular system for real-time data acquisition, analysis and control. PDP-11 systems can handle a wide variety of real-time control applications—each system being individually tailored from a comprehensive array of modular building blocks. Digital is unique among manufacturers of small-scale computers in its ability to provide not only fast and efficient processing units, but also a large family of its own compatible I/O devices including A/D and D/A converters, magnetic tape, disk storage, paper tape, and displays, as well as a wide range of general-purpose modules. This capability offers the user a new, more efficient approach to real-time systems.

The following paragraphs introduce the new PDP-11 by way of highlighting several of the important design features that set it apart from other machines in its class. Subsequent chapters of this manual place these features in their proper context and provide detailed descriptions of each.

PDP-11 SYSTEMS

The PDP-11 is available in two versions designated as PDP-11/10 and PDP-11/20. The PDP-11/10 contains a KA11 processor, 1,024 words of 16-bit read-only memory, and 128 16-bit words of read-write memory. The basic PDP-11/20 contains a KA11 processor and 4,096 words of 16-bit read-write core memory, a programmer's console, and an ASR-33 Teletype. Both versions can be similarly expanded with either read-write or read-only memory and peripheral devices.

UNIBUS

Unibus is the name given to the single bus structure of the PDP-11. The processor, memory and all peripheral devices share the same high-speed bus. The Unibus enables the processor to view peripheral devices as active memory locations which perform special functions. Peripherals can thus be addressed as memory. In other words, memory reference instructions can operate directly on control, status, or data registers in peripheral devices. Data transfers from input to output devices can bypass the processor completely.

KA11 PROCESSOR

The KA11 processor incorporates a unique combination of powerful features not previously available in 16-bit computers.

Priority Interrupts—A four-level automatic priority interrupt system permits the processor to respond automatically to conditions outside the system, or in the processor itself. Any number of separate devices can be attached to each level.

Each peripheral device in a PDP-11 system has a hardware pointer to its own unique pair of memory locations which, in turn, point to the device's service routine. This unique identification eliminates the need for polling of devices to identify an interrupt, since the interrupt servicing hardware selects and begins executing the appropriate service routine.

The device's interrupt priority and service routine priority are independent. This allows dynamic adjustment of system behavior in response to real-time conditions.

The interrupt system allows the processor continually to compare its own priority levels with the levels of any interrupting devices and to acknowledge the device with the highest level above the processor's priority level. Servicing an interrupt for a device can be interrupted for servicing a higher priority device. Service to the lower priority device can be resumed automatically upon completion of the higher level servicing. Such a process, called nested interrupt servicing, can be carried out to any level.

Reentrant Code—Both the interrupt handling hardware and the subroutine call hardware are designed to facilitate writing reentrant code for the PDP-11. This type of code allows use of a single copy of a given subroutine or program to be shared by more than one process or task. This reduces the amount of core needed for multi-task applications such as the concurrent servicing of many peripheral devices.

General Registers—The PDP-11 is equipped with eight general registers. All are program-accessible and can be used as accumulators, as pointers to memory locations, or as full-word index registers. Six registers are used for general-purpose access while the seventh and eighth registers are used as a stack pointer and program counter respectively.

Instruction Set—An important feature of the PDP-11 instruction set is the availability of double operand instructions. These instructions allow memory-to-memory processing and eliminate the need to use registers for storage of intermediate results. By using double operand instructions, every memory location can be treated as an accumulator. This significantly reduces the length of programs by eliminating load and store operations associated with single operand machines.

Addressing—Much of the power of the PDP-11 is derived from its wide range of addressing capabilities. PDP-11 addressing modes include list sequential addressing, full address indexing, full 16-bit word addressing, 8-bit byte addressing, stack addressing, and direct addressing to 32K words.
Variable length instruction formatting allows a minimum number of bits to be used for each addressing mode. This results in efficient use of program storage space.

Asynchronous Operation—The PDP-11's memory and processor operations are asynchronous. As a result, I/O devices transferring directly to or from memory may steal memory cycles during instruction operation.

PACKAGING

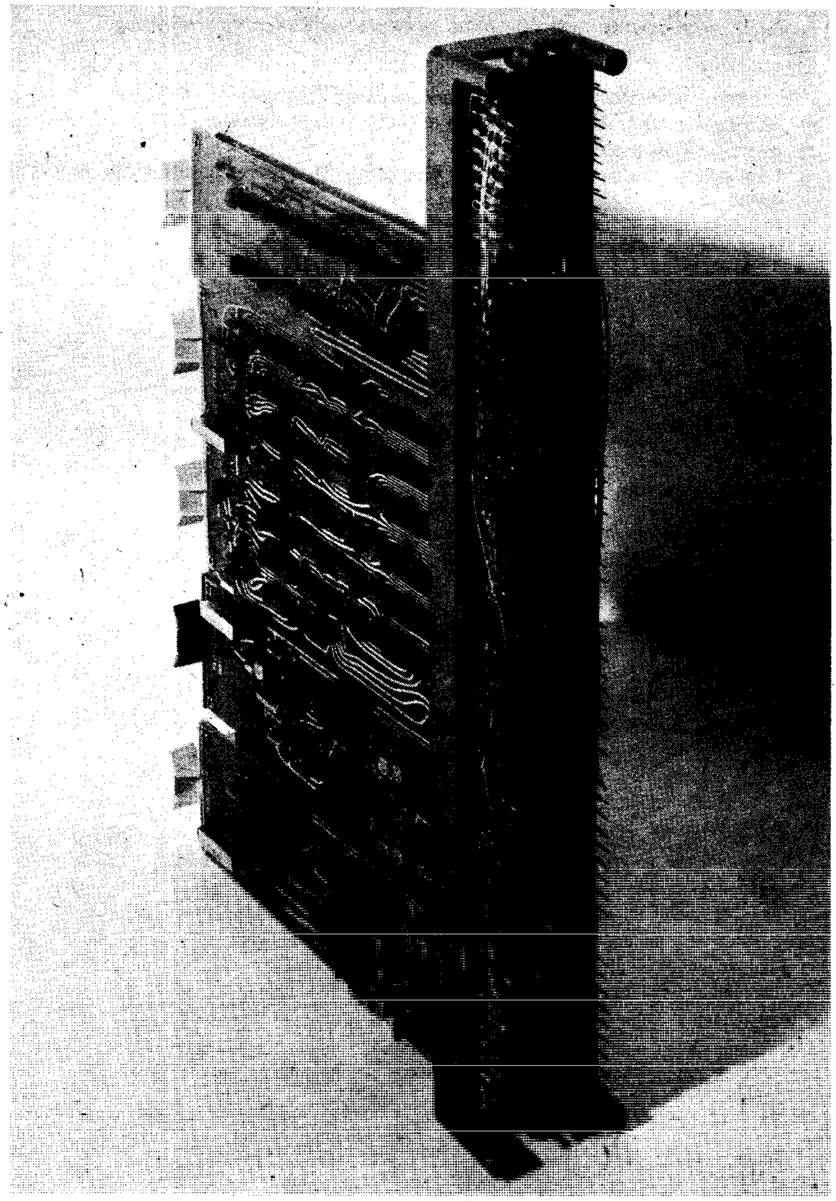
The PDP-11 has adopted a modular approach to allow custom configuring of systems, easy expansion, and easy servicing. Systems are composed of basic building blocks, called System Units, which are completely independent subsystems connected only by pluggable Unibus and power connections. There is no fixed wiring between them. An example of this type of subsystem is a 4,096-word memory module.
System Units can be mounted in many combinations within the PDP-11 hardware, since there are no fixed positions for memory or I/O device controllers. Additional units can be mounted easily and connected to the system

in the field. In case maintenance is required, defective System Units can be replaced with spares and operation resumed within a few minutes.

SOFTWARE

A complete package of user-oriented software includes:

- Absolute assembler providing object and source listings
- String-oriented editor
- Debugging routines capable of operating in a priority interrupt environment
- Input/output handlers for standard peripherals
- Relocatable integer and floating point math library



All PDP-11 processors, memories and peripherals are electrically and mechanically modular subsystems supported in System Units which are simply plugged together to form a computer tailored to user needs.

CHAPTER 2
SYSTEM INTRODUCTION

SYSTEM DEFINITION

Digital Equipment Corporation's PDP-11 is a 16-bit, general-purpose, parallel-logic computer using two's complement arithmetic. The PDP-11 is a variable word length processor which directly addresses 32,768 16-bit words or 65,536 8-bit bytes. All communication between system components is done on a single high-speed bus called a Unibus. Standard features of the system include eight general-purpose registers which can be used as accumulators, index registers, or address pointers, and a multi-level automatic priority interrupt system.

SYSTEM COMPONENTS

UNIBUS—There are five concepts that are very important for understanding both the hardware and software implications of the Unibus.

Single Bus—The Unibus is a single, common path that connects the central processor memory, and all peripherals. Addresses, data, and control information are sent along the 56 lines of the bus.

The form of communication is the same for every device on the Unibus. The processor uses the same set of signals to communicate with memory as with peripheral devices. Peripheral devices also use this set of signals when communicating with the processor, memory, or other peripheral devices.

Peripheral device registers may be manipulated as flexibly as core memory by the central processor. All the instructions that can be applied to data in core memory can be applied equally well to data in peripheral device registers. This is an especially powerful feature, considering the special capability of PDP-11 instructions to process data in any memory location as though it were an accumulator.

Bidirectional Lines—Unibus lines are bidirectional, so that the same signals which are received as input can be driven as output. This means that a peripheral device register can be either read or set by the central processor or other peripheral devices; thus, the same register can be used for both input and output functions.

Master-Slave Relation—Communication between two devices on the bus is in the form of a master-slave relationship. At any point in time, there is one device that has control of the bus. This controlling device is termed the "bus master." The master device controls the bus when communicating with another device on the bus, termed the "slave." A typical example of this relationship is the processor, as master, fetching an instruction from memory (which is always a slave). Another example is the disk, as master, transferring data to memory, as slave.

Interlocked Communication—Communication on the Unibus is interlocked so that for each control signal issued by the master device, there must be a response from the slave in order to complete the transfer. Therefore, communication is independent of the physical bus length and the response time of the master and slave devices. The maximum transfer rate on the Unibus is one 16-bit word every 750 nanoseconds, or 1.3 million 16-bit words per second.

Dynamic Master-Slave Relation—Master-slave relationships are dynamic. The processor, for example, may pass bus control to a disk. The disk, as master, could then communicate with a slave memory bank.

Since the Unibus is used by the processor and all I/O devices, there is a priority structure to determine which device gets control of the bus. Therefore, every device on the Unibus which is capable of becoming bus master has a priority assigned to it. When two devices which are capable of becoming a bus master request use of the bus simultaneously, the device with the higher priority will receive control first. Details of what conditions must be satisfied before a device will get control of the bus are given in the section on System Interaction.

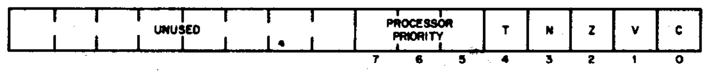
KA11 CENTRAL PROCESSOR—There are four major features which are of particular interest to the programmer: 1), the General Registers; 2), the Processor Status Word; (3), the Addressing Modes; and 4), the Instruction Set. The addressing modes and the instruction set of the PDP-11 processor will be discussed in detail in Chapters 3 and 4.

General Registers—The KA11 processor contains eight 16-bit general registers. These eight general registers (referred to as R0, R1, ... R7) may be used as accumulators, as index registers, or as stack pointers. One of these registers, R7, is reserved as a program counter (PC). Generally, the PC holds the address of the next instruction, but it may point to data or to an address of data. The register R6 has the special function of processor stack pointer.

Central Processor Status Register—The Central Processor Status Register (PS) contains information on the current priority of the processor, the result of previous operations, and an indicator for detecting the execution of an instruction to be trapped during program debugging. The priority of the central processor can be set under program control to any one of eight levels. This information is held in bits 5, 6, and 7 of the PS.
Four bits of the PS are assigned to monitoring different results of previous instructions. These bits are set as follows:

Z—if the result was zero
N—if the result was negative
C—if the operation resulted in a carry from the most significant bit
V—if the operation resulted in an arithmetic overflow

The T bit is used in program debugging and can be set or cleared under program control. If this bit is set, when an instruction is fetched from memory a processor trap will be caused by the completion of the instruction's execution.



Central Processor Status Register (PS)

CORE MEMORY—The PDP-11 allows both 16-bit word and 8-bit byte addressing. The address space may be filled by core memory and peripheral device registers. The top 4,096 words generally are reserved for peripheral device registers. The remainder of address space can be used for read-write core memory or read-only core memory.

Read-write core memory is currently available in 4,096 16-bit word segments. This memory has a cycle time of 1.2 microseconds and an access time of 500 nanoseconds. It is a standard part of a PDP-11/20 system.

Read-only core memory (ROM) is available in 1,024 16 bit-word segments. The access time of the ROM is 500 nanoseconds. Memory is also available in 128 16-bit word segments with a 2.0 microsecond cycle time. Both 1,024 words of read-only memory and 128 words of read-write memory mount in a single System Unit and are a standard part of the PDP-11/10 system.

PERIPHERAL DEVICES—The ASR-33 Teletype with low-speed paper tape reader and punch is provided in the basic PDP-11/20 system. Options for the PDP-11 include a paper tape reader capable of reading 300 characters per second, a paper tape punch with an output capacity of 50 characters per second, and additional Teletype units. Provision is made for the addition of numerous peripheral devices. These include standard DEC peripherals as well as other devices which will be unique to the PDP-11.

SYSTEM INTERACTION

At any point in time only one device can be in control of the bus, or be bus master. The master communicates with another device on the bus which is called the slave. Usually, the established master will communicate with the slave in the form of data transfers.

Full 16-bit words or 8-bit bytes of information can be transferred on the bus between the master and the slave. The information can be instructions, addresses, or data. This type of operation occurs when the processor, as master, is fetching instructions, operands, and data from memory, and restoring the results into memory after execution of instructions. Pure data transfers occur between a disk control and memory.

TRANSFER OF BUS MASTER—When a device (other than the central processor) is capable of becoming bus master and requests use of the bus, it is generally for one of two purposes: 1) to make a non-processor transfer of data directly to or from memory, or 2) to interrupt program execution and force the processor to branch to a specific address where an interrupt service routine is located.

PRIORITY STRUCTURE—When a device capable of becoming bus master requests use of the bus, the handling of that request depends on the location of that device in the priority structure. These factors must be considered to determine the priority of the request:

1. The processor's priority can be set under program control to one of eight levels using bits 7, 6, and 5 in the processor status register. These three bits set a priority level that inhibits granting of bus requests on lower levels.
2. Bus requests from external devices can be made on one of five request lines. A non-processor request (NPR) has the highest priority, and its request is honored by the processor between bus cycles of an instruction execution. Bus request 7 (BR7) is the next highest priority, and BR4 is the lowest. The four lower level priority requests are honored by the processor between instructions. When the processor's priority is set to a level, for example 6, all bus requests on BR6 and below are ignored.
3. When more than one device is connected to the same bus request (BR) line, a device nearer the central processor has a higher priority than a device farther away. Any number of devices can be connected to a given BR or NPR line.

Once a device other than the processor has control of the bus, it is for one of two types of requests: 1) NPR Request, 2) Interrupt Request.

NPR Requests—NPR data transfers can be made between any two peripheral devices without the supervision of the processor. Normally, NPR transfers are between a mass storage device, such as a disk, and core memory. The structure of the bus also permits device-to-device transfers, allowing customer-designed peripheral controllers to access other devices such as disks directly.

An NPR device has very fast access to the bus and can transfer at high data rates once it has control. The processor state is not affected by the transfer; therefore the processor can relinquish control while an instruction is in progress. This can occur at the end of any bus cycle except in between a read-modify-write sequence. (See Chapter 8 for details). In the PDP-11, an NPR device can gain bus control in 3.5 microseconds or less. An NPR device in control of the bus may transfer 16-bit words from memory at memory speed or every 1.2 microseconds in the PDP-11/20 or every 1.0 microseconds in the PDP-11/10.

Interrupt Requests—Devices that request interrupts on the bus request lines (BR7, BR6, BR5, BR4) can take advantage of the power and flexibility of the processor. The entire instruction set is available for manipulating data and status registers. When a device servicing program must be run, the task currently under way in the central processor is interrupted and the device service routine is initiated. Once the device request has been satisfied, the processor returns to the interrupted task.

In the PDP-11, the return address for the interrupted routine and the processor status word are held in a "stack." A stack is a dynamic sequential list of data with special provision for access from one end. A stack is also called a "push down" or "LIFO" (Last-In First-Out) list. Storage and retrieval from stacks is called "pushing" and "popping" respectively. These operations are illustrated in Figure 2-1.

In the PDP-11, a stack is automatically maintained by the hardware for interrupt processing. Thus, higher level requests can interrupt the processing of lower level interrupt service, and automatically return control to the lower level interrupt service routines when the higher level servicing is completed.

Here is an example of this procedure. A peripheral requires service and requests use of the bus at one of the BR levels (BR7, BR6, BR5, BR4). The operations undertaken to "service" the device are as follows:

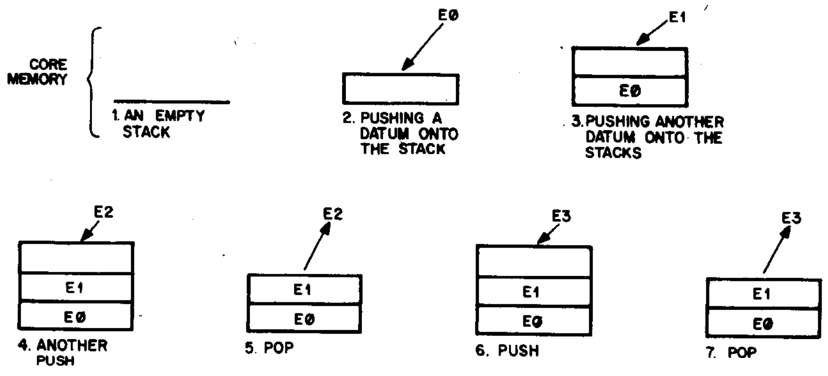
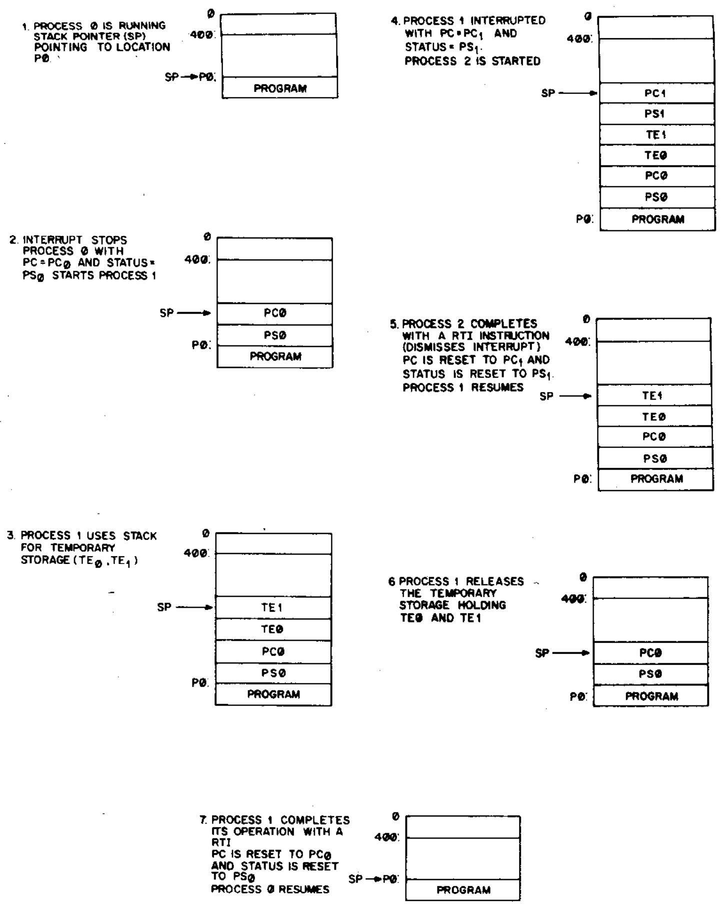


Fig 2-1 Illustration of Push and Pop Operations

1. Priorities permitting, the processor relinquishes the bus to that device.
2. When the device has control of the bus, it sends the processor an interrupt command with the address of the words in memory containing the address and status of the appropriate device service routine.
3. The processor then "pushes"—first, the current central processor status (PS) and then, the current program counter (PC) onto the processor stack.
4. The new PC and PS (the "interrupt vector") are taken from the location specified by the device and the next location, and the device



1. PROCESS Ø IS RUNNING STACK POINTER (SP) POINTING TO LOCATION PØ
Ø
400:
SP→PØ:
PROGRAM
2. INTERRUPT STOPS PROCESS Ø WITH PC=PCØ AND STATUS= PSØ STARTS PROCESS 1
SP
PCØ
PSØ
PØ:
PROGRAM
3. PROCESS 1 USES STACK FOR TEMPORARY STORAGE (TEØ, TE1)
TE1
TEØ
4. PROCESS 1 INTERRUPTED WITH PC=PC1 AND STATUS=PS1. PROCESS 2 IS STARTED
PC1
PS1
5. PROCESS 2 COMPLETES WITH A RTI INSTRUCTION (DISMISSES INTERRUPT) PC IS RESET TO PC1 AND STATUS IS RESET TO PS1. PROCESS 1 RESUMES
6 PROCESS 1 RELEASES THE TEMPORARY STORAGE HOLDING TEØ AND TE1
7. PROCESS 1 COMPLETES ITS OPERATION WITH A RTI PC IS RESET TO PCØ AND STATUS IS RESET TO PSØ PROCESS Ø RESUMES

Figure 2-2 Nested Device Servicing

service routine is begun. Note that those operations all occur automatically and that no device-polling is required to determine which service routine to execute.

5. 7.2 microseconds is the time interval between the central processor's receiving the interrupt command and the fetching of the first instruction. This assumes there were no NPR transfers during this time.
6. The device service routine can resume the interrupted process by executing the RTI (Return from Interrupt) instruction which "pops" the processor stack back into the PC and PS. This requires 4.5 microseconds if there are no intervening NPR's.
7. A device service routine can be interrupted in turn by a sufficiently high priority bus request any time after completion of its first instruction.
8. If such an interrupt occurs, the PC and PS of the device service routine are automatically pushed into the stack and the new device routine initiated as above. This "nesting" of priority interrupts can go on to any level, limited only by the core available for the stack. More commonly, this process will nest only four levels deep since there are four levels of BR signals. An example of nested device servicing is shown in Figure 2-2. A rough core map is given for each step of the process. The SP points to the top word of the stack as shown.

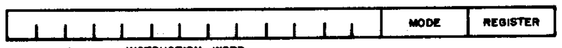
CHAPTER 3
ADDRESSING MODES

Most data in a program is structured in some way—in a table, in a stack, in a table of addresses, or perhaps in a small set of frequently-used variables local to a limited region of a program. The PDP-11 handles these common data structures with addressing modes specifically designed for each kind of access. In addition, addressing for unstructured data is general enough to permit direct random access to all of core. Memory is not broken up into pages and fields (often awkward and wasteful of core storage).

Addressing in the PDP-11 is done through the general registers. Programs requiring several stacks can use the general registers for stack pointers. Those requiring many local variables can use general registers as accumulators. The general registers can be used interchangeably as index registers or as sequential list pointers to access tabular data. Address arithmetic may be done directly in the general registers.

ADDRESS FIELDS

PDP-11 instruction words contain a 6-bit address field divided into two subfields selecting the general register and the mode generating the effective address.



INSTRUCTION WORD

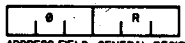
The register subfield specifies which of the eight general registers is to be used in the address calculation. The mode subfield indicates how this register is to be used in determining the operand. These modes will be described in the following paragraphs.

GENERAL REGISTER ADDRESSING—The general registers will be used as simple accumulators for operating on frequently-accessed variables. In this mode, the operand is held directly in the general register. The general registers are in fast memory, (280-nanosecond cycle time) resulting in a speed improvement for operations on these variables.

PAL-11, the PDP-11 assembler, interprets instructions of the form

OPR R

as general register operations. R has been defined as a register name and OPR is used to represent a general instruction mnemonic. The address field for general register operations is

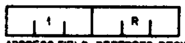


ADDRESS FIELD-GENERAL REGISTER MODE
(MODE IS INDICATED AS AN OCTAL DIGIT)

Operands that are pointed to by addresses (indirect or deferred) are denoted to PAL-11 by the @ symbol. Thus, instructions of the form

OPR @R

specify deferred register addressing and have the following address field.



ADDRESS FIELD-DEFERRED REGISTER MODE

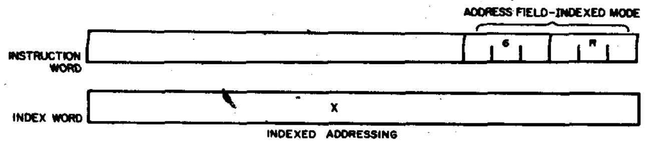
Deferred register addressing may also be selected in PAL-11 by the form OPR (R).

INDEXED ADDRESSING—The general registers may be used as index registers to permit random access of items in tables or stacks of data. Instructions of the form

OPR X(R)

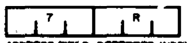
specify indexed mode addressing. The effective address is the sum of X and the contents of the specified general register R.

The index word containing X follows the instruction word.



INDEXED ADDRESSING

Index mode addressing can be deferred to permit access of data elements through tables or stacks of their addresses. The address field for index deferred mode is



ADDRESS FIELD-DEFERRED INDEXED MODE

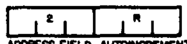
It is specified by instructions of the form

OPR @X(R)

AUTOINCREMENT ADDRESSING—Autoincrement addressing provides for automatic stepping of a pointer through sequential elements of a table of operands. In this mode, the address of the operand is taken from the general register and then the contents of the register are stepped (incremented by one or two) to address the next word or byte depending upon whether the instruction operates on byte or word data. Instructions of the form

OPR (R)+

specify autoincrement addressing. The address field for autoincrement addressing is

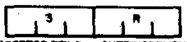


ADDRESS FIELD- AUTOINCREMENT

This mode may also be deferred. Instructions of the form

OPR @(R)+

specify deferred autoincrement addressing and assemble with the following address field.

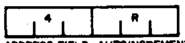


ADDRESS FIELD - AUTOINCREMENT DEFERRED MODE

AUTODECREMENT ADDRESSING—Autodecrement addressing steps the specified general register to the next lower byte (decrement by one) or word (decrement by two) address and uses the new contents of the general register as the operand address. Instructions of the form

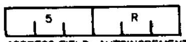
OPR —(R)

specify autodecrement addressing. The address field for autodecrement addressing is



ADDRESS FIELD- AUTOINCREMENT MODE

This mode also may be deferred and specified by instructions of the form OPR @ —(R). When deferred the address field is



ADDRESS FIELD-AUTOINCREMENT DEFERRED MODE

STACK PROCESSING

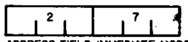
The combination of autoincrement addressing in which the general register is stepped forward *after* the operand address is determined and autodecrement addressing in which the general register is stepped backward *before* the operand address is determined is the basic requirement for convenient low overhead stack operations.

The PDP-11 has extensive stack processing capabilities. The stack pointer (SP), R6, maintains a stack for the nested handling of interrupts. All of the general registers can maintain stacks under program control. Elements in the middle of stacks may be accessed through indexed addressing. This provides for convenient access of dynamically assigned temporary storage, especially useful in nested procedures.

USE OF THE PC AS A GENERAL REGISTER

There are special implications in the use of the addressing modes already described when applied to the PC (R7). The use of the PC with the addressing modes described above generates immediate, absolute, relative, and deferred relative addressing.

IMMEDIATE ADDRESSING—Immediate addressing provides time and space improvement for access of constant operands by including the constant in the instruction. The instruction word referencing an immediate operand specifies autoincrement addressing through the program counter. The address field would be



ADDRESS FIELD-IMMEDIATE MODE

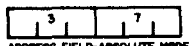
The program counter points to the word after the instruction word. The contents of this word are therefore used as the operand and the program counter is stepped to the next word. PAL-11 recognizes address expressions of the form "#n" as immediate operands and codes them with the address field shown above followed by a word of data (n).

A full word is assembled for immediate operands even in byte instructions so that instruction words are always fetched from even locations.

ABSOLUTE ADDRESSING—The contents of the location following the instruction word may be taken as the address of an operand by specifying deferral in immediate mode addressing. That is, instructions of the form

OPR @#A

refer to the operand at address A. PAL-11 assembles address expressions of this form into an address field



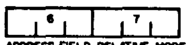
ADDRESS FIELD-ABSOLUTE MODE

followed by a word containing the operand address.

RELATIVE ADDRESSING—Relative addressing specifies the operand address relative to the instruction location. This is accomplished by using the PC as an index register. The PC is considered as a base address. The offset, the distance between the location of the operand and the PC, is held in the index word of the instruction. PAL-11 assembles instructions of the form

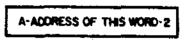
OPR A

(where A has not been assigned as a name of a general register) as an instruction word with the address field

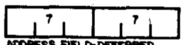


ADDRESS FIELD-RELATIVE MODE

followed by an index word of the form



DEFERRED RELATIVE ADDRESSING—Deferral of relative addressing permits access to data through memory locations holding operand addresses. The "@" character specifies deferred addressing; i.e., OPR @A. The address field for deferred relative addressing is



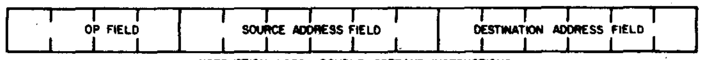
ADDRESS FIELD-DEFERRED RELATIVE MODE

USE OF THE SP AS A GENERAL REGISTER

The processor stack pointer will in most cases be the general register used in PDP-11 stack operations. Note that the content of SP, (SP), refers to the top element of the stack, that —(SP) will push data onto the stack, that (SP)+ will pop data off the stack, and that X(SP) will permit random access of items on the stack. Since the SP is used by the processor for interrupt handling, it has a special attribute: autoincrements and autodecrements are always done in steps of two. Byte operations using the SP in this way will simply leave odd addresses unmodified.

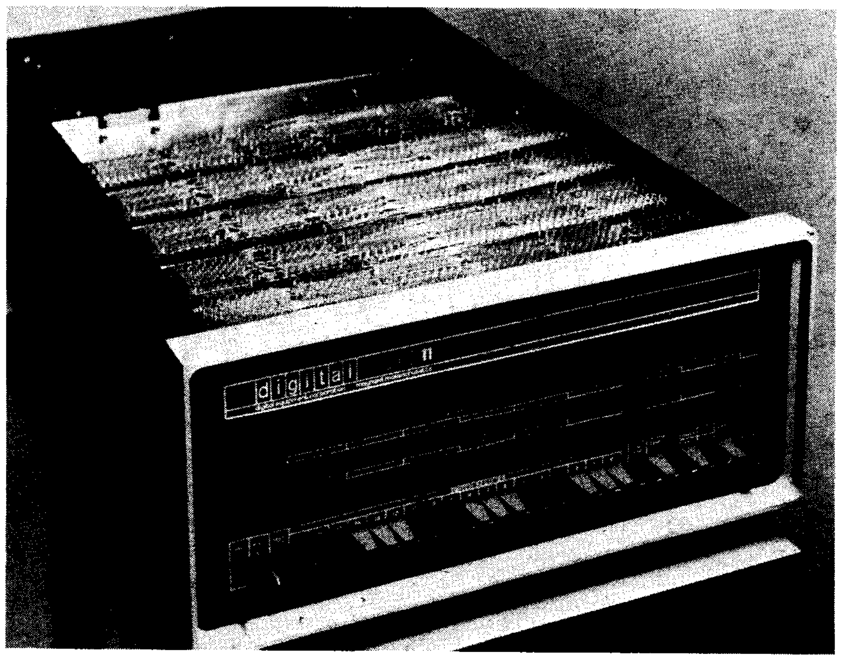
DOUBLE OPERAND ADDRESSING

Operations which imply two operands such as add, subtract and compare are presented in the PDP-11 by instructions which specify two addresses. The instruction word for such operations is of the form



INSTRUCTION WORD— DOUBLE OPERAND INSTRUCTIONS

Instruction Word—Double Operand Instructions and is followed by index words and immediate operands for the source and destination address fields as appropriate. Source address calculations are performed before destination address calculations. Since each operand may be anywhere in core storage or in the general registers, each memory location is thus effectively provided with the arithmetic capabilities of an accumulator. Further, since peripheral device registers and memory location are addressed in the same way, the contents of peripheral data buffers can be stored or loaded directly to and from memory without use of any general register. This means that interrupt routines can be executed without saving and restoring any of the general registers.



The PDP-11 is a 16-bit computer with a universal bus called a Unibus allowing networks of memories and peripherals to be used in virtually any combination.

CHAPTER 4
INSTRUCTION SET

This chapter presents the order code for the PDP-11. Each PDP-11 instruction is described in terms of five parameters: operation, effect on condition codes, base timing, assembler mnemonics, and octal representation. Special comments are included where appropriate.

NOTATION

The following notations will be used in this section:

(XXX)	: The contents of XXX
src	: The Source Address
dst	: The Destination Address
$\wedge$	: Boolean "AND" Function
$\vee$	: Boolean "OR" Function
$\forall$	: Boolean "Exclusive OR" Function
$\sim$	: Boolean 'NOT" Function (Complement)
$\rightarrow$	: "becomes"
$\uparrow$	: "is popped from the stack"
$\downarrow$	: "is pushed onto the stack"

INSTRUCTION TIMING

The PDP-11 is an asynchronous processor in which, in many cases, memory and processor operations are overlapped. The execution time for an instruction is the sum of a basic instruction time and the time to determine and fetch the source and/or destination operands. The following table shows the addressing times required for the various modes of addressing source and destination operands. The instruction time for each operation is given (throughout this chapter) for the 11/20 configuration. All times stated are subject to ±20% variation.

ADDRESSING FORM	**TIMING**	
(src or dst)	src (μs)†	dst (μs)†
R	0	0
(R) or @R	1.5	1.4*
(R) +	1.5	1.4*
−(R)	1.5	1.4*
@(R) +	2.7	2.6*
@−(R)	2.7	2.6*
BASE(R)	2.7	2.6*
@BASE(R) or @(R)	3.9	3.8*

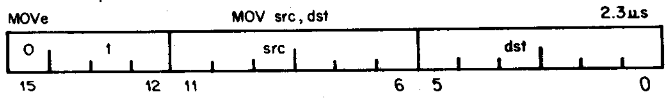
* dst time is .4 μs. less than listed time if instruction was a
CoMPare, CoMPare Byte
Bit Test, Bit Test Byte
TeST, or TeST Byte
none of which ever modify the destination word.
† referencing bytes at odd addresses adds 0.6μs to src and dst times.

DOUBLE OPERAND INSTRUCTIONS—Double Operand Instructions are represented in assembly language as:

OPR src, dst

where src and dst are the addresses of the source and destination operands respectively. The execution time for these operations is comprised of the source time, the destination time, and the instruction time. The source and destination times depend on addressing modes and are described in the preceding table.

Arithmetic Operations—



Operation: (src) → (dst)

Condition Codes:
Z: set if (src) = 0 ; cleared otherwise
N: set if (src) < 0; cleared otherwise
C: not affected
V: cleared

Description: Moves the source operand to the destination location. The previous contents of the destination are lost. The contents of the source are not affected.

The MOV instruction is a generalization of 'load," "store," "setup," 'push," "pop,' and interregister transfer operations.

General registers may be loaded with the contents of memory addresses with instructions of the form:

MOV src, R

Registers may be loaded with a counter, and pointer values with MOV instructions:

MOV #n, R
(which loads the number n into register R)

Operands may be pushed onto a stack by:

MOV src, -(R)

and may be popped off a stack by:

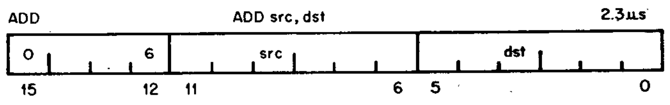
MOV (R)+, dst

Interregister transfers are simply:

MOV RA, RB
(RA and RB are general registers)

Memory-to-memory transfers may be done with the MOV instruction in the general form:

MOV src, dst



Operation: (src) + (dst) → (dst)

Condition Codes:
Z: set if result = 0; cleared otherwise
N: set if result < 0; cleared otherwise
C: set if there was a carry from the most significant bit of the result; cleared otherwise
V: set if there was arithmetic overflow as a result of the operation, that is, if both operands were of the same sign and the result was of the opposite sign; cleared otherwise

Description: Adds the source operand to the destination operand and stores the result at the destination address. The original contents of the destination are lost. The contents of the source are not affected. Two's complement addition is performed.

The ADD instruction includes as special cases the "add-to-register," "add-to-memory," and "add-register-to-register" functions:

Add-to-Register	ADD src, R
Add-to-Memory	ADD R, dst
Add Register-to-Register	ADD RA, RB

Arithmetic may also be done directly in memory by the general form ADD instruction

ADD src, dst

Use of this form saves considerable loading and storing of accumulators.

Two special cases of the ADD instruction are particularly useful in compilers, interpreters, and other stack arithmetic processes:

ADD (R)+, (R)

(where R is the stack pointer)

which replaces the top two elements of the stack with their sum; and ADD src, (R), which increases the top element of the stack by the contents of the source address.

The "Add Immediate" operation is yet another special case of this generalized ADD instruction:

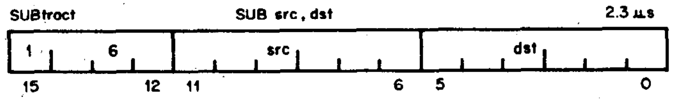
ADD #n, dst

Immediate operations are useful in dealing with constant operands. Note that:

ADD #n, R

steps the register R (which may be an index register) through n addresses eliminating the need for a special "add-to-index- register" instruction.

All these special cases of the ADD instruction apply equally well to the other double operand instruetions that follow.

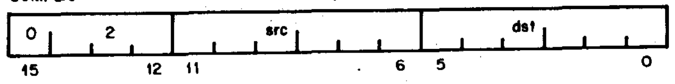


Operation: (dst) — (src) → (dst) [in detail, (dst) + ~ (src) + 1 → (dst)]

Condition Codes:
- Z: set if result = 0; cleared otherwise
- N: set if result < 0; cleared otherwise
- C: cleared if there was a carry from the most significant bit of the result; set otherwise
- V: set if there was arithmetic overflow as a result of the operation, that is, if the operands were of opposite signs and the sign of source was the same as the sign of the result; cleared otherwise.

Description: Subtracts the source operand from the destination operand and leaves the result at the destination address. The original contents of the destination are lost. The contents of the source are not affected.

CoMPare CMP src,dst 2.3µs*



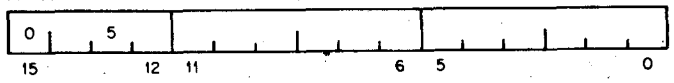
Operation: (src) — (dst) [in detail, (src) + ~ (dst) + 1]

Condition Codes:	Z: set if result = 0; cleared otherwise
	N: set if result < 0; cleared otherwise
	C: cleared if there was a carry from the most significant bit of the result; set otherwise
	V: set if there was arithmetic overflow; that is, operands were of opposite signs and the sign of the destination was the same as the sign of the result; cleared otherwise.

Description: Arithmetically compares the source and destination operands. Affects neither operand. The only action is to set the condition codes appropriately.

Boolean Instructions—These instructions have the same format as the double operand arithmetic group. They permit operations on data at the bit level.

Bit Set BIS src,dst 2.3µs

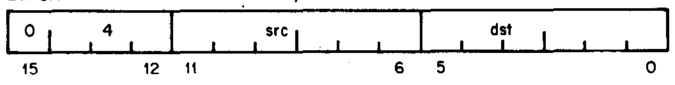


Operation: (src) V (dst) → (dst)

Condition Codes:	Z: set if result = 0; cleared otherwise
	N: set if high-order bit of result set; cleared otherwise
	C: not affected
	V: cleared

Description: Performs "Inclusive OR" transfer between the source and destination operands and leaves the result at the destination address; that is, corresponding bits set in the source are set in the destination. The original contents of the destination are lost. The source is not affected.

Bit Clear BIC src,dst 2.9µs

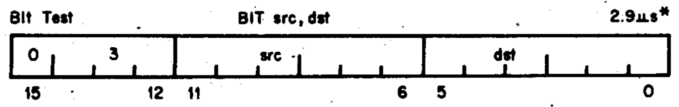


Operation: ~ (src) ∧ (dst) → (dst)

Conditions Codes:	Z: set if result = 0; cleared otherwise
	N: set if high-order bit of result set; cleared otherwise
	C: not affected
	V: cleared

Description: The BIC instruction clears each bit in the destination that corresponds to a set bit in the source. The original contents of the destination are lost. The contents of the sources are unaffected.

*There is no read/modify/write cycle in the CMP, BIT, and TST operations. This saves 0.4 µs in all destination address modes except register mode.



Operation: (src) $\wedge$ (dst)

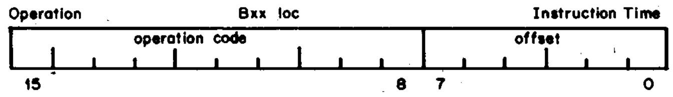
Condition Codes:	Z: set if result = 0; cleared otherwise
	N: set if high-order bit of result set; cleared otherwise
	C: not affected
	V: cleared

Description: Performs logical "and" comparison of the source and destination operands and modifies condition codes accordingly. Neither the source nor destination operands are affected.

The BIT instruction may be used to test whether **any** of the corresponding bits that are set in the destination are also set in the source or whether all corresponding bits set in the destination are clear in the source.

Note that the operations of BIS, BIC, and BIT are parallel in that the same mask may be used to set, clear and test the state of particular bits in a word.

BRANCHES—Branches have the instruction format



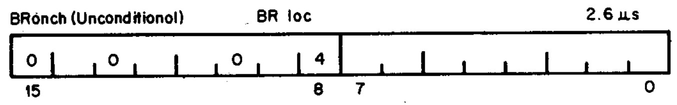
The offset is treated as a signed two's complement displacement to be multiplied by 2 and applied to the program counter. The program counter points to the next word in sequence. The effect is to cause the next instruction to be taken from an address, "loc", located up to 127 words back (—254 bytes) or 128 words ahead (+ 256 bytes) of the branch instruction. PAL-11 gives an error indication in the instruction if "loc" is outside this range.

The PDP-11 assembler handles address arithmetic for the user and computes and assembles the proper offset field for branch instructions in the form

Bxx loc

where loc is the address to which the branch is to be made. The branch instructions have no effect on condition codes.

Unconditional Branch—



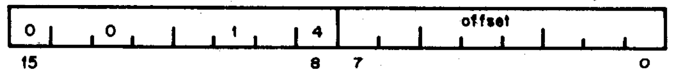
Operation: loc → (PC)

Description: Provides a way of transferring program control within a limited range with a one word instruction. The execution time is equal to the instruction time (2.6μs) for the operation.

Simple Conditional Branches—Conditioned branches combine in one instruction a conditional skip, unconditional branch sequence.

Timing for the conditional branches is shown as execution time if the condition is not met, followed by the execution time if the condition is met (and a program branch occurs).

Branch on EQual(Zero) BEQ loc 1.5 µs, 2.6 µs



Operation: loc → (PC) if Z = 1

Description: Tests the state of the Z-bit and causes a branch if Z is set. It is used to test equality following a CMP operation, to test that no bits set in the destination were also set in the source following a BIT operation, and generally, to test that the result of the previous operation was zero.

Thus the sequence

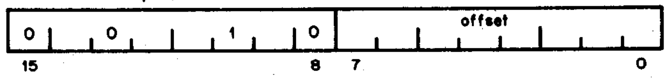
```
        CMP   A,B      ; compare A and B
        BEQ   C        ; branch if they are equal
```

will branch to C if A = B (A − B = 0)
and the sequence

```
        ADD   A,B      ; add A to B
        BEQ   C        ; branch if the result = 0
```

will branch to C if A + B = 0.

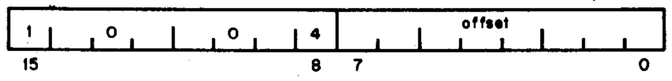
Branch on Not Equal(Zero) BNE loc 1.5µs, 2.6 µs



Operation: loc → (PC) if Z = 0

Description: Tests the state of the Z-bit and causes a branch if the Z-bit is clear. BNE is the complementary operation to BEQ. It is used to test inequality following a CMP, to test that some bits set in the destination were also set in the source, following a BIT and, generally, to test that the result of the previous operation was not zero.

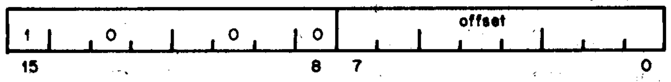
Branch on Minus BMI loc 1.5 µs, 2.6 µs



Operation: loc → (PC) if N = 1

Description: Tests the state of the N-bit and causes a branch if N is set. It is used to test the sign (most significant bit) of the result of the previous operation.

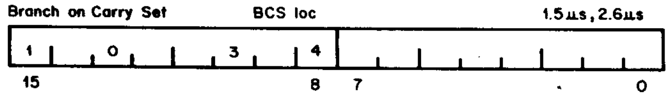
Branch on PLus BPL loc 1.5 µs, 2.6µs



Operation: loc → (PC) if N = 0.

Description: Tests the state of the N-bit and causes a branch if N is clear. BPL is the complementary operation to BMI.

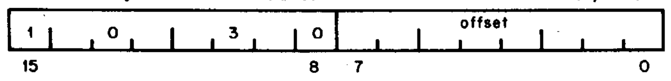
Branch on Carry Set BCS loc 1.5µs, 2.6µs



Operation: loc → (PC) if C = 1

Description: Tests the state of the C-bit and causes a branch if C is set. It is used to test for a carry in the result of a previous operation.

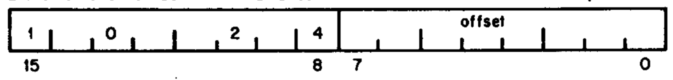
Branch on Carry Clear BCC loc 1.5µs, 2.6µs



Operation: loc → (PC) if C = 0

Description: Tests the state of the C-bit and causes a branch if C is clear. BCC is the complementary operation to BCS.

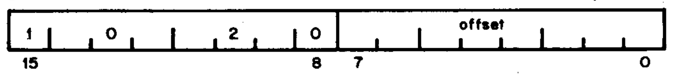
Branch on oVerflow Set BVS loc 1.5µs, 2.6µs



Operation: loc → (PC) if V = 1

Description: Tests the state of the V-bit (overflow) and causes a branch if the V-bit is set. BVS is used to detect arithmetic overflow in the previous operation.

Branch on oVerflow Clear BVC loc 1.5µs, 2.6µs



Operation: loc → (PC) if V = 0

Description: Tests the state of the V-bit and causes a branch if the V-bit is clear. BVC is the complementary operation to BVS.

Signed Conditional Branches—Particular combinations of the condition code bits are tested with the signed conditioned branches. These instructions are used to test the results of instructions in which the operands were considered as signed (two's complement) values.

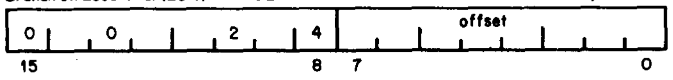
Note that the sense of signed comparisons differs from that of unsigned comparisons in that in signed 16-bit, two's complement arithmetic the sequence of values is as follows:

```
largest ................................ 077777
                                         077776
positive                                   .
                                           .
                                           .
                                         000001
         ——————————————————————————————— 000000
                                         177777
                                         177776
                                           .
negative                                   .
                                           .
                                         100001
smallest ——————————————————————————————— 100000
```

whereas in unsigned 16-bit arithmetic the sequence is considered to be

```
highest ................................ 177777
                                           .
                                           .
                                           .
                                           .
                                           .
                                           .
                                         000002
                                         000001
lowest ................................. 000000
```

Branch on Less Than(Zero) BLT loc 1.5 µs, 2.6 µs

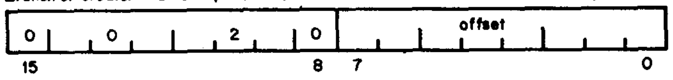


Operation: loc → (PC) if N ⊻ V = 1

Description: Causes a branch if the "Exclusive OR" of the N- and V-bits are 1. Thus BLT will always branch following an operation that added two negative numbers, even if overflow occurred.

In particular, BLT will always cause a branch if it follows a CMP instruction operating on a negative source and a positive destination (even if overflow occurred). Further, BLT will never cause a branch when it follows a CMP instruction operating on a positive source and negative destination. BLT will not cause a branch if the result of the previous operation was zero (without overflow).

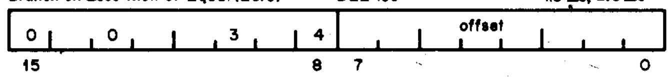
Branch on Greater than or Equal (Zero) BGE loc 1.5 µs, 2.6 µs



Operation: loc → (PC) if N ⊻ V = 0

Description: Causes a branch if N and V are either both clear or both set. BGE is the complementary operation to BLT. Thus BGE will always cause a branch when it follows an operation that caused addition to two positive numbers. BGE will also cause a branch on a zero result.

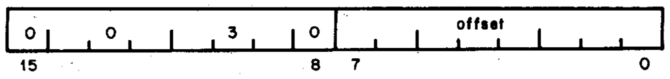
Branch on Less than or Equal (Zero) BLE loc 1.5 µs, 2.6 µs



Operation: loc → (PC) if Z v (N ∀ V) = 1

Description: Operation of BLE is similar to that of BLT but in addition will cause a branch if the result of the previous operation was zero.

Branch on Greater Than (Zero) BGT loc 1.5 µs, 2.6 µs

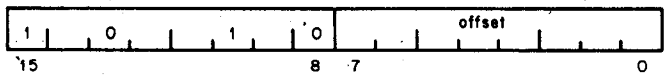


Operation: loc → (PC) if Z v (N ∀ V) = 0

Description: Operation of BGT is similar to BGE, except that BGT will not cause a branch on a zero result.

Unsigned Conditional Branches—The Unsigned Conditional Branches provide a means of testing the result of comparison operations in which the operands are considered as unsigned values.

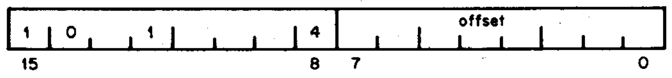
Branch on HIgher BHI 1.5 µs, 2.6 µs



Operation: loc → (PC) if both C and Z = 0

Description: Causes a branch if the previous operation caused neither a carry nor a zero result. This will happen in comparison (CMP) operations as long as the source has a higher unsigned value than the destination.

Branch on LOwer or Same BLOS loc 1.5 µs, 2.6 µs

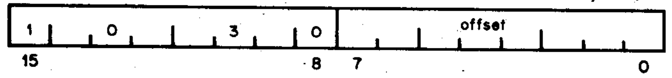


Operation: loc → (PC) if C v Z = 1

Description: Causes a branch if the previous operation caused either a carry or a zero result. BLOS is the complementary operation to BHI. The branch will occur in comparison operations as long as the source is equal to, or has a lower unsigned value than, the destination.

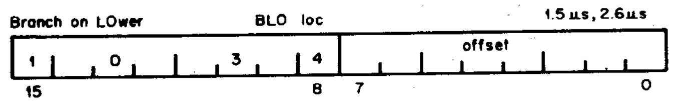
Comparison of unsigned values with the CMP instruction can be tested for "higher or same" and "higher" by a simple test of the C-bit. For convenience, the mnemonics BHIS (Branch on HIgher or Same) and BLOS (Branch on Lower Or Same) have been defined such that BHIS = BCC and BLO = BCS.

Branch on Higher or Same BHIS loc 1.5 µs, 2.6 µs



Operation: loc → (PC) if C = 0

Description: BHIS is the same instruction as BCC



Operation: loc → (PC) if C = 1

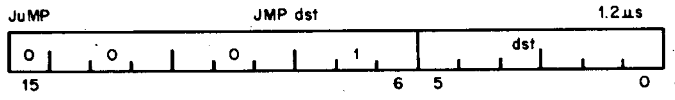
Description: BLO is the same instruction as BCS

The following example illustrates the use of some of the instructions and addressing modes described thus far. Two new instructions are used: INC (INCrement) and ASL (Arithmetic Shift Left) which respectively, add 1 (INC) and multiply an operand by 2 (ASL). Their operation is fully described later in this chapter.

This example demonstrates the generation of a table (histogram) that shows the frequency of occurrence of each value in another table (within a range of values 1-100). Histogram generation (including initialization) requires 22 words. Values outside the range 1-100 are ignored.

```
HIST:       MOV #OTABLE, R0      ;set up to clear output table
            MOV #—100., R1       ;100 entries in output table
CLOOP:      CLR (R0)+            ;clear next entry
            INC R1               ;check if done
            BNE CLOOP            ;if not, continue clearing
            MOV #ITABLE, R0      ;set up input pointer
            MOV # −1000., R1     ;length of table
            MOV #100., R2        ;max input value
HLOOP:      MOV (R0)+, R4        ;get next input value
            BLE NOCOUNT          ;ignore if less than or equal zero
            CMP R4, R2           ;check against max value
            BGT NOCOUNT          ;ignore if greater
            ASL R4               ;2 bytes per table entry
            INC OTABLE (R4)      ;increment proper element
NOCOUNT:    INC R1               ;input done?
            BNE HLOOP            ;if not, continue scanning
            HALT                 ;histogram complete
```

The JuMP Instruction—JMP (JuMP) provides more flexible program branching then is provided with the branch instructions. Control may be transferred to any location in memory (no range limitation) and can be accomplished with the full flexibility of the PDP-11 addressing modes.



Operation: dst → (PC)

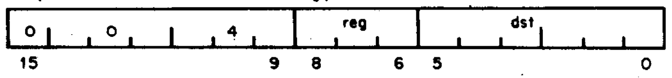
Conditioned Codes: not affected

Description: Register mode is illegal in JMP instructions and will cause an "illegal instruction" condition. (Program control cannot be transferred to a register.) Register deferred mode is legal and will cause program control to be transferred to the address held in the specified register. Note that instructions are word data and must therefore be fetched from an even-numbered address. A "boundary error" condition will result when the processor attempts to fetch an instruction from an odd address.

Deferred index mode JMP instructions permit transfer of control to the address contained in a selectable element of a table of dispatch vectors.

SUBROUTINES—The subroutine call in the PDP-11 provides for automatic nesting of subroutines, reentrancy, and multiple entry points. Subroutines may call other subroutines (or indeed themselves) to any level of nesting without making special provision for storage of return addresses at each level of subroutine call. The subroutine calling mechanism modifies no fixed location in memory and thus also provides for reentrancy. This allows one copy of a subroutine to be shared among several interrupting processes.

Jump to SubRoutine JSR reg, dst 4.2 µs



Operation: dst → (tmp) (tmp is an internal processor register)
(reg) ↓ (push reg contents onto processor stack)
(PC) → (reg) (PC holds location following JSR; this address
(tmp) → (PC) now put in reg)

Condition Codes: not affected

Description: Execution time for JSR is the sum of instruction and destination times. In execution of the JSR, the old contents of the specified register, (the "linkage pointer"), are automatically pushed onto the processor stack and new linkage information placed in the register. Thus subroutines nested within subroutines to any depth may all be called with the same linkage register. There is no need either to plan the maximum depth at which any particular subroutine will be called or to include instructions in each routine to save and restore the linkage pointer. Further, since all linkages are saved in a reentrant manner—on the processor stack—execution of a subroutine may be interrupted, the same subroutine reentered and executed by an interrupt service routine. Execution of the initial subroutine can then be resumed when other requests are satisfied. This process (called nesting) can proceed to any level.

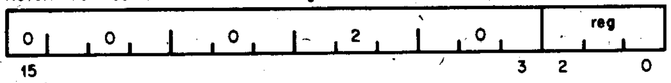
A subroutine called with a JSR reg, dst instruction can access the arugments following the call with either autoincrement addressing, (reg) +, (if arguments are accessed sequentially) or by indexed addressing, X(reg), (if accessed in random order). These addressing modes may also be deferred, @(reg)+ and @X(reg) if the parameters are operand addresses rather than the operands themselves.

JSR PC, dst is a special case of the PDP-11 subroutine call suitable for subroutine calls that transmit parameters through the general registers. No register except the program counter is modified by this call.

Another special case of the JSR instruction is JSR PC, (SP)+ which exchanges the top element of the processor stack and the contents of the program counter. Use of this instruction allows two routines to swap program control and resume operation when recalled where they left off. Such routines are called "co-routines."

Return from a subroutine is done by the RTS instruction. RTS reg loads the contents of the reg into the PC and pops the top element of the processor stack into the specified register.

ReTurn from Subroutine   RTS reg   3.5 µs



Operation: (reg) → (PC)
↑ (reg)

Condition Codes: not affected

Description: Loads content of reg into PC and pops the top element of the processor stack into the specified register. Execution time for RTS is equal *to the basic instruction time.*

Return from a subroutine is typically made through the same register that was used in its call. Thus, a subroutine called with a JSR PC, dst exists with a RTS PC and a subroutine called with a JSR R5, dst, picks up parameters with addressing modes (R5)+, X(R5), or @X(R5) and finally exists with a RTS R5.

Programming Examples of the Use of Subroutines—

1. Passing arguments in subroutine calls—The subroutine TOLER checks each element in an array of unsigned integers to determine whether any elements are outside specified limits. If all are within tolerance, the value 0 is returned in the register R0. If TOLER find an element out of tolerance, it returns the address of the bad element + 2 in R0. The calling sequence for TOLER is:

```
                JSR R5, TOLER
· WORD ARRAY                                ;address of array to be
                                            ;checked (·WORD expres-
                                            ;sion—defines a word equal
                                            ;to the value of the expres-
                                            ;sion)
· WORD —LENGTH                              ;minus # of items in array
· WORD HILIM                                ;upper limit of tolerance
· WORD LOLIM                                ;lower limit of tolerance
                                            ;subroutine returns here

;Tolerance Check-Array Elements Within Limits?
TOLER:          MOV (R5)+, R0               ;get array address
                MOV (R5)+, R1               ;get minus the length
                MOV (R5)+, R2               ;get high tolerance limit
                MOV (R5)+, R3               ;get low tolerance limit
TLOOP:          MOV (R0)+, R4               ;get next element of array
                CMP R4, R2                  ;check it against high limit
                BHI TEXIT                   ;leave routine if higher
                CMP R4, R3                  ;check it against low limit
                BLO TEXIT                   ;leave routine if lower
                INC R1                      ;increment count, check
                                            ;whether at end of array
                BNE TLOOP                   ;continue if not at end yet
                CLR R0                      ;exit with R0 = 0 if all ok
TEXIT:          RTS R5                      ;return, R0 holds pointer
                                            ;or 0
```

The instruction INC R1 increases the contents of R1 by 1 and the instruction CLR R0 zeroes the register R0

2. Saving and restoring registers on the stack—This subroutine pushes R0-R5 onto the stack. It is called by:

```
                JSR R5, SAVE
SAVE:           MOV R4, —(SP)       ;R5 was pushed by the JSR
                MOV R3, —(SP)       ;R5 will be at the bottom
                                    ;of the stack
                MOV R2, —(SP)       ;R4, R3, R2, R1, and R0
                                    ;in order
                MOV R1, —(SP)       ;will be above it
                MOV R0, —(SP)       ;R0 is at the top of the
                                    ;stack
                JMP R5              ;R5 holds the return ad-
                                    ;dress
```

The TST operation is equivalent to comparing the operand with 0, i.e.,

TST opr = CMP opr, #0

The only effect is to set the appropriate condition codes.

The following example illustrates a subroutine to restore R0-R5 from the stack.

```
REST:           TST (SP) +          ;this increments the SP by 2
                MOV (SP)+, R0       ;the registers are restored
                MOV (SP)+, R1       ;in reverse order to that in
                MOV (SP)+, R2       which
                MOV (SP)+, R3       ;they were put on the stack
                MOV (SP)+, R4       ;R5 is loaded into the PC
                RTS R5              and the old R5 restored
```

The operation TST (SP)+ removes the top element on the stack. At the time it is used, the top element holds the contents of R5 that were saved by the call to REST. Since R5 is to be loaded with the value saved on the stack by SAVE, this information is not needed.

3. Stacks, recursion, and nesting—The following subroutine converts an unsigned binary integer to a string of typed ASCII characters. In the routine, the remainders of successive divisions by 10 are saved and then typed in reverse order.

   The operation of the subroutine is to call a part of itself (beginning with DECREM) repeatedly until a zero quotient is calculated by an integer divide subroutine, IDIVR. At each iteration, the dividend is divided by 10, the resulting quotient replaces the dividend, and the remainder is pushed onto the processor stack. The processor stack thus holds interleaved data (remainders) and control information (return addresses from calls to DECPNT and DECREM) when the quotient finally comes up as 0 and the branch is made to DECTTY. The portion of the routine beginning at DECTTY then pops a remainder from the stack, converts it to an ASCII character, types it and then returns control to DECTTY (with RTS PC) until the stack is reduced finally to its state immediately after the call to DECPNT.

At this point execution of RTS PC returns control to the main program.

A character is typed in DECTY by loading the teleprinter buffer (TPB) and waiting for the teleprinter READY flag, the most significant bit of the low-order byte of the teleprinter status word (TPS), to be set.

The symbols CR and LF are assumed equal to the ASCII representations for carriage return and line feed respectively.

This subroutine types the unsigned integer in R0. It illustrates recursion and the use of stacks.

```
DECPNT:     MOV #10., R2          ;set up divisor of 10
DECREM:     JSR PC, IDIVR         ;subroutine divides (R0) by
                                  ;(R2)
            MOV R1, —(SP)         ;quotient is in R0, remain-
                                  ;der is in R1
            TST R0                ;after pushing remainder
                                  ;onto stack test quotient
            BEQ DECTTY            ;if the quotient is 0, we're
                                  ;done getting remainders
            JSR PC, DECREM        ;if not try again
DECTTY:     MOV (SP)+, R0         ;get next remainder
            ADD #60, R0           ;make an ASCII character
TTYOUT:     MOV R0, TPB           ;type the ASCII character in
                                  ;R0
TTYLUP:     TST TPS               ;wait for the teleprinter to
                                  ;be done
            BPL TTYLUP            ;TPS is negative when the
                                  ;TP is done
            CMP #CR, R0           ;was the character of a car-
                                  ;riage return
            BEQ TTYLF             ;if not: return, if so; get a
                                  ;line feed
            RTS PC                ;returns either to DECTTY
                                  ;or main program
TTYLF:      MOV #LF, TPB          ;type a line feed
            BR TTYLUP             ;and wait for it to be com-
                                  ;pleted
```

4. Multiple entry points—In the example that follows, the subroutines described above are used to type out all the entries in a table of unsigned integers that are not within specified tolerance.

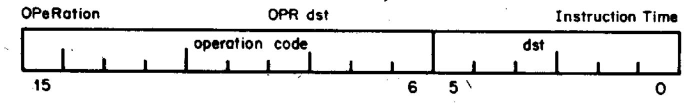
   The subroutine TOLER is entered at TOLER for initialization and at TLOOP to pick up each bad entry of the array after the first one.

   The subroutine DECPNT is entered at DECPNT to print the value of the unsigned binary number held in R0 and at TTYOUT to print the ASCII character held in R0. TTYOUT prints the carriage return, line feed sequence when it sees the carriage return character.

   This routine types all out-of-tolerance elements of an integer array. The program starts at TYPOUT.

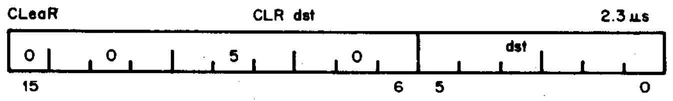
```
TYPFIN:     HALT                 ;suspend processor opera-
                                 ;tion, wait for key continue
TYPOUT:     JSR R5, TOLER        ;get address of bad item;
                                 ;initialization entry
            . WORD ARRAY         ;address of array
            . WORD —LENGTH       ;-length of array
            . WORD HILIM         ;high limit
            . WORD LOLIM         ;low limit
TYPCHK:     BEQ TYPFIN           ;Z-bit is set if no more out
                                 ;of limits
            JSR R5, SAVE         ;an element is out of limits,
                                 ;save registers
            MOV —(R0), R0        ;R0 holds address + 2, get
                                 ;operand into R0
            JSR PC, DECPNT       ;print out number
            MOV #CR, R0          ;type CR, LF
            JSR PC, TTYOUT       ;note use of second entry
                                 ;point
            JSR R5, REST         ;restore registers
            JSR R5, TLOOP        ;continue searching array,
                                 ;alternate entry
            BR TYPCHK            ;another bad element?
```

SINGLE OPERAND INSTRUCTIONS–Single Operand Instructions are represented as:



The execution time for single operand instructions is the sum of the basic instruction time and destination address time for the operation.

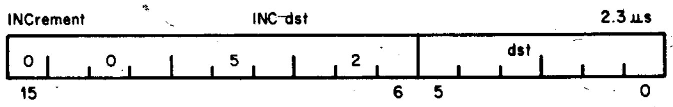
General Operations—



Operation: 0 → (dst)

Condition Codes:	Z: set
	N: cleared
	C: cleared
	V: cleared

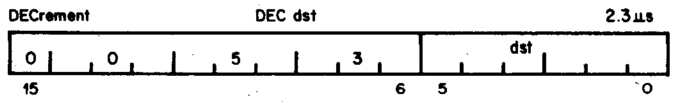
Description: Zeroes the specified destination.



Operation: (dst) + 1 → (dst)

Condition Codes:	Z: set if the result is 0; cleared otherwise
	N: set if the result is < 0; cleared otherwise
	C: not affected
	V: set if (dst) held 077777; cleared otherwise

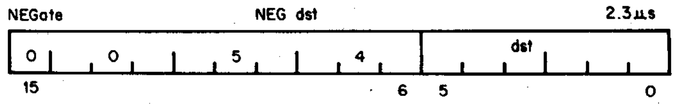
Description: Adds 1 to the contents of the destination.



Operation: (dst) — 1 → (dst)

Condition Codes	Z: set if the result is 0; cleared otherwise
	N: set if the result is < 0; cleared otherwise
	V: not affected
	C: set if (dst) was 100000; cleared otherwise

Description: Subtracts 1 from the contents of the destination.

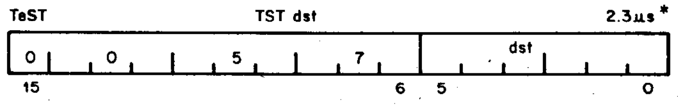


Operation: — (dst) → (dst)

Condition Codes: as in SUB dst, #0

Z:	set if the result is 0; cleared otherwise
N:	set if the result is < 0; cleared otherwise
C:	cleared if the result is 0; set otherwise
V:	set if the result is 100000; cleared otherwise

Description: Replaces the contents of the destination address by their two's complement. (However, 100000_8 is replaced by itself—in two's complement notation the most negative number has no positive counterpart.)

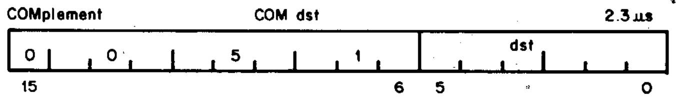


Operation: 0 — (dst)

Condition Codes: as in CMP #0, dst

Z:	set if the result is 0; cleared otherwise
N:	set if the result is < 0; cleared otherwise
C:	cleared
V:	cleared

Description: Sets the condition codes Z and N according to the contents of the destination address.



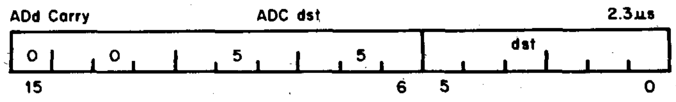
Operation: ~ (dst) → (dst)

Condition Codes:	Z: set if result is 0; cleared otherwise
	N: set if most significant bit of result set; cleared otherwise
	C: set
	V: cleared

Description: Replaces the contents of the destination address by their logical complement (each bit equal to 0 is set and each bit equal to 1 is cleared).

* See the note for the CMP instruction.

Multiple Precision Operations—It is sometimes convenient to do arithmetic on operands considered as multiple words. The PDP-11 makes special provision for such operations with the instructions ADC (ADd Carry) and SBC (SuBtract Carry).



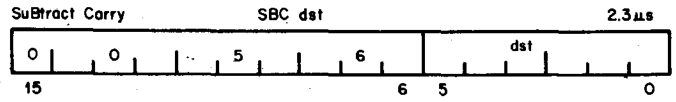
Operation: (dst) + (C) → (dst)

Condition Codes:
Z: set if result = 0; cleared otherwise
N: set if result < 0; cleared otherwise
C: set if (dst) was 177777 and (C) was 1; cleared otherwise
V: set if (dst) was 077777 and (C) was 1; cleared otherwise.

Description: Adds the contents of the C-bit into the destination. This permits the carry from the addition of the two low-order words to be carried into the high-order result.

Double precision addition may be done with the following instruction sequence:

```
ADD A0, B0   ;  add low-order parts
ADC B1       ;  add carry into high-order
ADD A1, B1   ;  add high-order parts
```



Operation: (dst) — (C) → (dst)

Condition Codes:
Z: set if the result 0; cleared otherwise
N: set if the result < 0; cleared otherwise
C: cleared if the result is 0 and C = 1; set otherwise
V: set if the result is 100000; cleared otherwise

Description: Subtracts the contents of the C-bit from the destination. This permits the carry from the subtraction of two low-order words to be subtracted from the high-order part of the result.

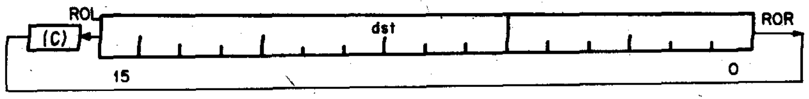
Double precision subtraction is done by:

```
SUB A0, B0
SBC B1
SUB A1, B1
```

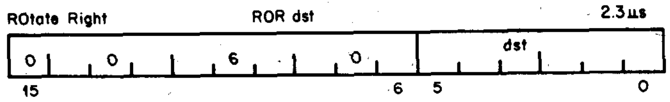
Double precision negation is accomplished with:

```
NEG B0      ;negate low-order part; sets C unless B0 = 0
SBC B1      ;makes "NEG B1" = "COMB B1" unless B0 = 0
NEG B1      ;negate high-order part
```

Rotates—Testing of sequential bits of a word and detailed bit manipulation are aided with rotate operations. The instructions ROR (ROtate Right) and ROL (ROtate Left) cause the C-bit of the status register to be effectively appended to the destination operand in circular bit shifting.



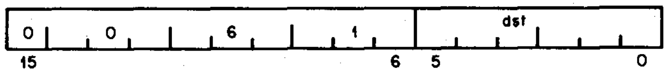
ROtate Right ROR dst 2.3µs



Condition Codes:
- Z: set if all bits of result = 0; cleared otherwise.
- N: set if the high-order bit of the result is set; cleared otherwise
- C: loaded with the low-order bit of the destination
- V: loaded with the Exclusive OR of the N-bit and C-bit (as set by the completion of the rotate operation).

Description: Rotates all bits of the destination right one place. Bit 0 is loaded into the C-bit of the status word and the previous contents of the C-bit are loaded into bit 15 of the destination.

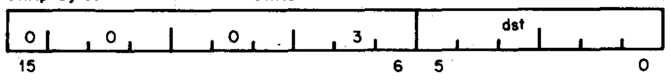
ROtate Left ROL dst 2.3µs



Condition Codes:
- Z: set if all bits of the result word = 0; cleared otherwise
- N: set if the high-order bit of the result word is set; cleared otherwise
- C: loaded with the high-order bit of the destination
- V: loaded with the Exclusive OR of the N-bit and C-bit (as set by the completion of the rotate operation)

Description: Rotates all bits of the destination left one place. Bit 15 is loaded into the C-bit of the status word and the previous contents of the C-bit are loaded into bit 0 of the destination.

SWAp Bytes SWAB dst 2.3µs



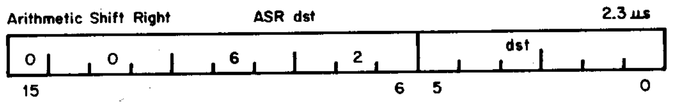
Condition Codes:
- Z: set if low-order byte of result = 0; cleared otherwise
- N: set if high-order bit of low-order byte (bit 7) of result is set; cleared otherwise
- C: cleared
- V: cleared

Description: Exchanges high-order byte and low-order byte of the destination word (dst must be a word address).

Shifts—Scaling data by factors of 2 is accomplished by the shift instructions:

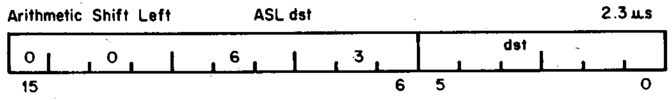
ASR—Arithmetic Shift Right
ASL—Arithmetic Shift Left

The sign bit (bit 15) of the operand is replicated in shifts to the right. The low-order bit is filled with 0 in shifts to the left. Bits shifted out of the C-bit are lost.



Condition Codes:
Z: set if the result = 0; cleared otherwise
N: set if the high-order bit of the result is set; cleared otherwise
C: loaded from the low-order bit of the destination
V: loaded from the Exclusive OR of the N-bit and C-bit (as set by the completion of the shift operation)

Description: Shifts all bits of the destination right one place. Bit 15 is replicated. The C-bit is loaded from bit 0 of the destination. ASR performs signed division of the destination by 2.



Condition Codes:
Z: set if the result = 0; cleared otherwise
N: set if the high-order bit of the result is set; cleared otherwise
C: loaded with the high-order bit of the destination
V: loaded with the Exclusive OR of the N-bit and C-bit (as set by the completion of the shift operation)

Description: Shifts all bits of the destination left one place. Bit 0 is loaded with a 0. The C-bit of the status word is loaded from the most significant bit of the destination. ASL performs a signed multiplication of the destination by 2.

Multiple precision shifting is done with a sequence of shifts and rotates.

Double Precision Right Shift:

```
ASR A1;  low-order bit of A1 to C-bit
ROR A0;  C-bit to high-order bit of A0
```

Double Precision Left Shift:

```
ASL A0;  high-order bit of A0 to C-bit
ROL A1;  C-bit to low-order bit of A1
```

Normalization of operands (scaling of the operand until the operand taken as a 15-bit fraction with sign is in the range — ½ < operand ⩽ ½) proceeds as follows:

```
NORM:     ASL   A       ; shift 0's into low-order bit
          BEQ   NFIN    ; if the result is 0, the operation is
                        ; complete
          BVC   NORM    ; if the sign did not change, continue
          ROR   A       ;restore the sign
          BR    NDONE   ; normalization complete
NFIN:     ROR   A       ; restore the sign: 000000 or 100000
          ASR   A       ; and replicate it: 000000 or 140000
NDONE:    . . .
```

Double precision normalization proceeds similarly:

```
DNORM:     ASL     A0       ; double precision left shift
           ROL     A1       ;
           BEQ     DZERO    ; high order result 0?. if so, check low
           BVC     DNORM    ; if the sign did not change, continue
           ROR     A1       ; restore the sign
           BR      DNDONE   ; normalization complete
DZERO:     TST     A0       ; low order zero, too?
           BNE     DNORM    ; if not, continue normalization
           ROR     A1       ; restore the sign; 000000 or 100000
           ASR     A1       ; and replicate it; 000000 or 140000
DNDONE:    . . .
```

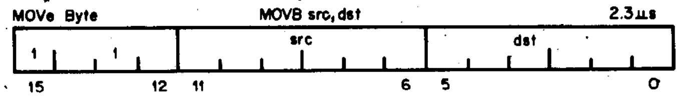
The following example illustrates the use of shifts and rotates in a 16-bit unsigned integer multiply subroutine. Access of operands through address parameters following the subroutine is also shown. The multiplication takes 115-170 μs in in-line code. The entire subroutine as shown below takes approximately 200 μs and requires 16 words. The calling sequence is JSR R5, MULT.

```
          .WORD MCAND          ; address of multiplicand
          .WORD MPLIER         ; address of multiplier
          .WORD PROD           ; address of product
MULT:     CLR R0
          MOV @ (R5) +, R1     ; get multiplier into R1
          MOV @ (R5) +, R2     ; get multiplicand into R2
          MOV #—16., R3        ; set counter
MLOOP:    ASL R0               ; double prec shift
          ROL R1               ; shift and add multiply
          BCC NOADD            ; most significant bit governs add
          ADD R2, R0           ; if set add in multiplicand
          ADC R1               ; keep 32-bit product
NOADD:    INC R3               ; done?
          BNE MLOOP            ; if not continue
          MOV (R5) +, R2       ; get address to store prod.
          MOV R0, (R2) +       ; put low-order away, move to high
          MOV R1, (R2)         ; put high-order away
          RTS R5               ; return to calling program
```

BYTE OPERATIONS—The PDP-11 processor includes a full complement of instructions that manipulate byte operands. Addressing is byte-oriented so that instructions for byte manipulation are straightforward. In addition, byte instructions with autoincrement or autodecrement direct addressing cause the specified register to be stepped by one to point to the next byte of data. Byte operations in register mode access the low-order byte of the specified register. These provisions enable the PDP-11 to perform as either a word or byte processor.

Timing of byte instructions is the same as for word instructions except that an additional 0.6 μs is required for access of bytes at odd addresses.

Double Operand Byte Instructions—

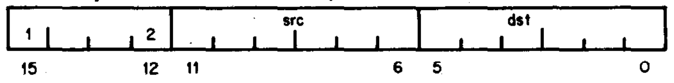


Operation: (src) → (dst)

Condition Codes: Set on the byte result as in MOV

Description: Same as MOV instruction. The MOVB instruction in register mode (unique among byte operations) extends the most significant bit of the byte register (sign extension). Otherwise MOVB operates on bytes exactly as MOV operates on words.

CoMPare Byte CMPB src,dst 2.3μs*

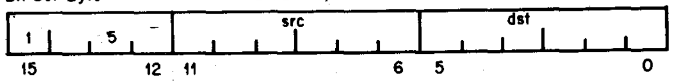


Operation: (src) — (dst) ; in detail (src) + ~ (dst) + 1

Condition Codes: Set on the byte result as in CMP

Description: Same as CMP instruction.

Bit Set Byte BISB src,dst 2.3μs

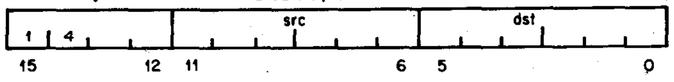


Operation: (src) V (dst) → (dst)

Condition Codes: Set on the byte result as in BIS

Description: Same as BIS.

Bit Clear Byte BICB src,dst 2.3μs

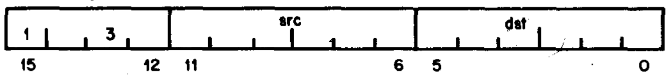


Operation: ~ (src) (dst) → (dst)

Condition Codes: set on the byte result as in BIC

Description: Same as BIC.

Bit Test Byte BITB src,dst 2.3μs*



Operation: (src) (dst)

Condition Codes: Set on the byte result as in BIT

Description: Same as BIT.

The following subroutine scans a packed character string of variable length lines, removes blanks and unpacks the string to left-justified length lines. INSTRING is the address of the INput STRING, OUTSTRING is the address of the OUTput String. EOLCHAR, SPCHAR, and EORCHAR are the End Of Line CHARacter, SPace CHARacter, and End of Record CHARacter respectively.

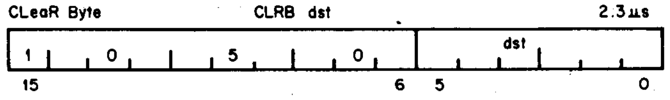
* *See the note for the CMP instruction.*

LNLINE is the Length of uNpacked LINES. The routine requires 24 words.

```
EDIT:     MOV #INSTRING, R0      ; set up input byte pointer
          MOV #OUTSTRING, R1     ; set up output byte pointer
          MOV #EOLCHAR, R2       ; put high use constant in reg.
          MOV #SPCHAR, R3        ; to save time in loop
NOLINE:   MOV #LNLINE, R4        ; R4 holds # char left in line
NXTCHR:   MOVB (R0) + ,R5        ; get next byte
          CMP R5, R2             ; end of line?
          BEQ FILINE             ; if yes, fill line
          CMP R5, R3             ; blank?
          BEQ NXTCHR             ; if yes, skip character
          DEC R4                 ; decrement # of characters left in line
          MOVB R5, (R1) +        ; move byte to output string
          BR NXTCHR              ; continue
FILINE:   CLRB (R1) +            ; put a blank byte in output
          DEC R4                 ; decrement # char left
          BNE FILINE             ; continue if not end
CHKEND:   CMPB (R0), #EORCHAR    ; end of record?
          BNE NULINE             ; if not EOR, start next line
```

Single Operand Byte Instructions—

CLeaR Byte CLRB dst 2.3µs

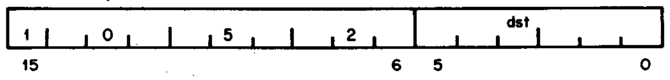


Operation: 0 → (dst)

Condition Codes: Set on the byte result as in CLR

Description: Same as CLR

INCrement Byte INCB dst 2.3µs

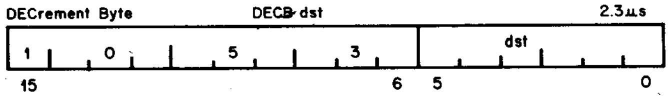


Operation: (dst) + 1 → (dst)

Condition Codes: Set on the byte result as in INC

Description: Same as INC. The carry from a byte does not affect any other byte.

DECrement Byte DECB dst 2.3µs

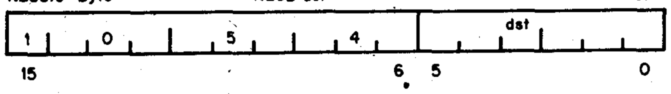


Operation: (dst) — 1 → (dst)

Condition Codes: Set on the byte result as in DEC

Description: Same as DEC.

NEGate Byte **NEGB dst** **2.3 µs**

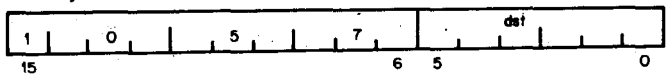


Operation: —(dst) → (dst) ; in detail, ~ (dst) + 1 → (dst)

Condition Codes: Set on the byte result as NEG

Description: Same as NEG.

TeST Byte **TSTB dst** **2.3 µs***

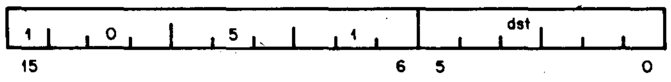


Operation: 0 — (dst)

Condition Codes: Set on the byte result as TST

Description: Same as TST.

COMplement Byte **COMB dst** **2.3 µs**

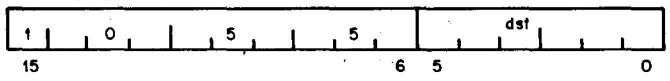


Operation: ~ (dst) → (dst)

Condition Codes: Set on the byte result as COM

Description: Same as COM.

ADd Carry Byte **ADCB dst** **2.3 µs**

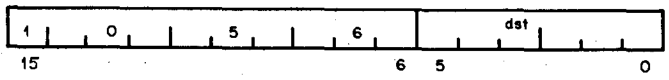


Operation: (dst) + (C) → (dst)

Condition Codes: Set on the byte result as ADC

Description: Same as ADC.

SuBtract Carry Byte **SBCB dst** **2.3 µs**

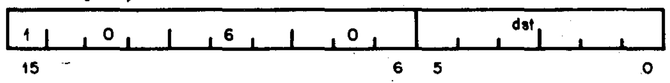


Operation: (dst) — (C) → (dst)

Condition Codes: Set on the byte result as SBC

Description: Same as SBC.

ROtate Right Byte **RORB dst** **2.3 µs***



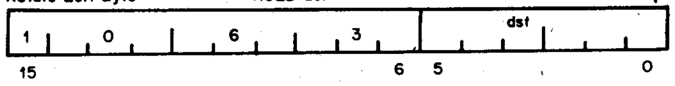
Operation: as in ROR on byte operands

Condition Codes: Set on the byte result as ROR

Description: Same as ROR

* See the note for the CMP instruction.

ROtate Left Byte ROLB dst 2.3µs†

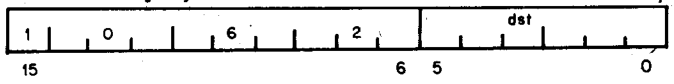


Operation: as in ROL on byte operands

Condition Codes: set on the byte results as ROL

Description: Same as ROL

Arithmetic Shift Right Byte ASRB dst 2.3µs†

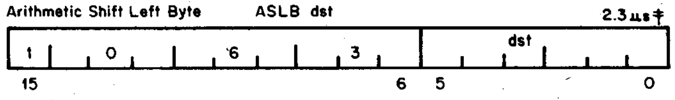


Operation: as in ASR on byte operands

Condition Codes: set on the byte result as ASR

Description: Same as ASR

Arithmetic Shift Left Byte ASLB dst 2.3µs†



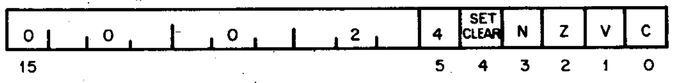
Operation: as in ASL on byte operands

Condition Codes: set on the byte results as ASL

Description: Same as ASL

CONDITION CODE OPERATORS—Condition code operators set and clear condition code bits. Selectable combinations of these bits may be cleared or set together in one instruction.

Condition Code Operators 1.5µs



Condition code bits corresponding to bits in the condition code operator (bits 3-0; N, Z, V, C) are modified according to the sense of bit 4, the set/clear bit of the operator. The following mnemonics are permanent symbols in the assembler:

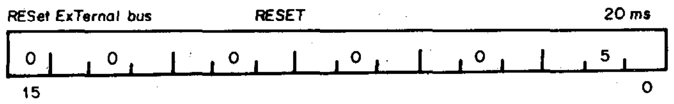
Mnemonic	Operation	Op Code	Mnemonic	Operation	Op Code
CLC	Clear C	000241	SEC	Set C	000261
CLV	Clear V	000242	SEV	Set V	000262
CLZ	Clear Z	000244	SEZ	Set Z	000264
CLN	Clear N	000250	SEN	Set N	000270

Timing for all condition code operators is the basic instruction time (1.5µs) for the operators. (The codes 000240 and 000260 are the shortest "no-operation" instructions.)

† Shift and rotate operations require an additional 0.6 µs to access bytes at odd addresses.

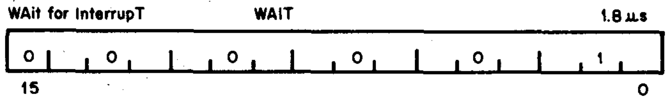
Combinations of the above set or clear operations may be ORed together to form new instruction mnemonics. For example: CLCV = CLC ! CLV. The new instruction clears C and V bits. ("!" signifies "inclusive or" in PAL-11.)

MISCELLANEOUS CONTROL INSTRUCTIONS



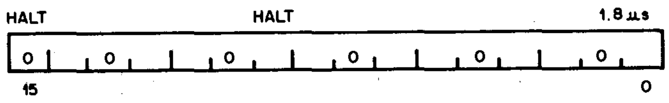
Condition Codes: not affected

Description: Sends an INIT pulse along the Unibus by the processor. All devices on the bus are reset to their state at power-up.



Condition Codes: not affected
Description: Provides a way for the processor to relinquish use of the bus while it waits for an external interrupt. Having been given a WAIT command, the processor will not compete for bus use by fetching instructions or operands from memory. This permits higher transfer rates between a device and memory, since no processor-induced latencies will be encountered by bus requests from the device. In WAIT, as in all instructions, the PC points to the next instruction following the WAIT operation.

Thus when an interrupt causes the PC and PS to be pushed onto the processor stack, the address of the next instruction following the WAIT is saved. The exit from the interrupt routine (i.e. execution of an RTI instruction) will cause resumption of the interrupted process at the instruction following the WAIT.

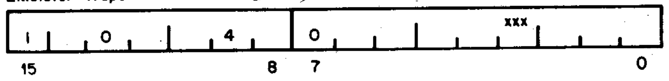


Condition Codes: not affected
Description: Causes the processor operation to cease. The console is given control of the bus. The console data lights display the contents of R0; the console address lights display the address of the halt instruction. Transfers on the Unibus are terminated immediately. The PC points to the next instruction to be executed. Pressing the continue key on the console causes processor operation to resume. No INIT signal is given.

Processor Traps —Processor Traps are internally generated interrupts. Error conditions, completion of an instruction in trace mode (i.e. T-bit of status word set), and certain instructions cause traps. As in interrupts, the current PC and PS are saved on the processor stack and a new PC and PS are loaded from the appropriate trap (interrupt) vector. See Appendix C for a summary of Trap Vector Addresses.

Trap Instructions—Trap Instructions provide for calls to emulators, I/O monitors, debugging packages, and user-defined interpreters.

EMulator Traps EMT xxx 8.9μs

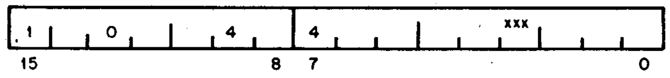


Operation: (PS) ↓ SP
(PC) ↓ SP
(30) → PC
(32) → PS

Condition Codes: loaded from trap vector.

Description: Performs a trap sequence with a trap vector address of 30. All operation codes from 104000 to 104377 are EMT calls. The low-order byte, bits 0-7 of the EMT instructions, may be used to transmit information to the emulating routine (e.g., function to be performed). The trap vector for EMT is at address 30. The new PC is taken from the word at address 30; the new central processor status (PS) is taken from the word at address 32.

TRAP TRAP xxx 8.9μs

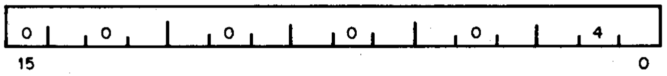


Operation: as in EMT except the trap vector is located at 34.

Condition Codes: loaded from trap vector.

Description: Performs a trap sequence with a trap vector address of 34. Operation codes from 104400 to 104777 are TRAP instructions. TRAPs and EMTs are identical in operation, except that the trap vector for TRAP is at address 34.

I/O Trap IOT 8.9μs

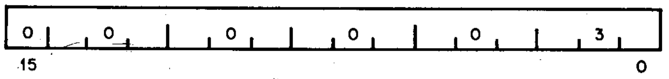


Operation: as EMT except the trap vector is located at address 20 and no information is transmitted in the low byte.

Condition Codes: loaded from trap vector.

Description: Used to call the I/O executive routine IOX.

No defined mnemonic 000003 8.9μs

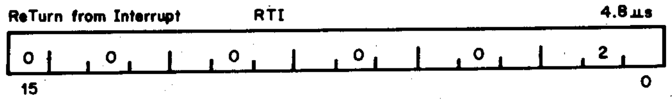


Operation: Same as IOT except that trap vector is located at address 14.

Condition Codes: loaded from trap vector.

Description: Used to call debugging aids. The user is cautioned against employing code 000003 in programs run under these debugging aids.

ReTurn from Interrupt RTI 4.8µs



Operation: SP ↑ (PC), SP ↑ (PS).

Condition Codes: loaded from processor stack.

Description: Used to exit from an interrupt or TRAP service routine. The PC and PS are restored (popped) from the processor stack.

Instruction traps are also caused by attempts to execute instruction codes reserved for future processor expansion (reserved instructions) or instructions with illegal addressing modes (illegal instructions). Order codes not corresponding to any of the instructions described above are considered to be reserved instructions. Illegal instructions are JMP and JSR with register mode destinations. Reserved and illegal instruction traps occur as described under EMT, but trap through vectors at addresses 10 and 04 respectively.

Stack Overflow Trap—Stack Overflow Trap is a processor trap through the vector at address 4. It is caused by referencing addresses below 400_8 through the processor stack pointer R6 (SP) in autodecrement or autodecrement deferred addressing. The instruction causing the overflow is completed before the trap is made.

Bus Error Traps—Bus Error Traps are:

1. Boundary Errors—attempts to reference word operands at odd addresses.

2. Time-Out Errors—attempts to reference addresses on the bus that made no response within 10 µs. In general, these are caused by attempts to reference nonexistent memory, and attempts to reference nonexistent peripheral devices.

Bus error traps cause processor traps through the trap vector address 4.

Trace Trap—Trace Trap enables bit 4 of the PS word and causes processor traps at the end of instruction executions. The instruction that is executed after the instruction that set the T-bit will proceed to completion and then cause a processor trap through the trap vector at address 14.

The following are special cases and are detailed in subsequent paragraphs.

1. The traced instruction cleared the T-bit.
2. The traced instruction set the T-bit.
3. The traced instruction caused an instruction trap.
4. The traced instruction caused a bus error trap.
5. The traced instruction caused a stack overflow trap.
6. The process was interrupted between the time the T-bit was set and the fetching of the instruction that was to be traced.
7. The traced instruction was a WAIT.
8. The traced instruction was a HALT.

An instruction that cleared the T-bit—Upon fetching the traced instruction an internal flag, the trace flag, was set. The trap will still occur at the end of execution of this instruction. The stacked status word, however, will have a clear T-bit.

An instruction that set the *T-bit*—Since the T-bit was already set, setting it again has no effect.

An instruction that caused an Instruction Trap—The instruction trap is sprung and the entire routine for the service trap is executed. If the service routine exists with an RTI or in any other way restores the stacked status word, the T-bit is set again, the instruction following the traced instruction is executed and, unless it is one of the special cases noted above, a trace trap occurs.

An instruction that caused a Bus Error—This is treated as in an Instruction Trap. The only difference is that the error service is not as likely to exit with an RTI, so that the trace trap may not occur.

An instruction that caused a stack overflow—The instruction completes execution as usual—the Stack Overflow does not cause a trap. The Trace Trap Vector is loaded into the PC and PS, and the old PC and PS are pushed onto the stack. Stack Overflow occurs again, and this time the trap is made.

An interrupt between setting of the T-bit and fetch of the traced instruction—The entire interrupt service routine is executed and then the T-bit is set again by the exiting RTI. The traced instruction is executed (if there have been no other interrupts) and, unless it is a special case noted above, causes a trace trap.

Note that no interrupts are acknowledged between the time of fetching any trapped instruction (including one that is trapped by reason of the T-bit being set) and completing execution of the first instruction of the trap service.

A *WAIT*—The trap occurred immediately. The address of the next instruction is saved on the stack.

A *HALT*—The processor halts. When the continue key on the console is pressed, the instruction following the HALT is fetched and executed. Unless it is one of the exceptions noted above, the trap occurs immediately following execution.

Trap priorities—In case multiple processor trap conditions occur simultaneously the following order of priorities is observed (from high to low):

1. Bus Errors
2. Instruction Traps
3. Trace Trap
4. Stack Overflow Trap

The details on the trace trap process have been described in the trace trap operational description which includes cases in which an instruction being traced causes a bus error, instruction trap, or a stack overflow trap.

If a bus error is caused by the trap process handling instruction traps, trace traps, stack overflow traps, or a previous bus error, the processor is halted.

If a stack overflow is caused by the trap process in handling bus errors, instruction traps, or trace traps, the process is completed and then the stack overflow trap is sprung.

CHAPTER 5
ADDRESS ALLOCATION

The PDP-11 provides for a very flexible addressing structure. Both 16-bit words and 8-bit bytes can be directly addressed. Addresses are 16-bits long allowing for direct addressing of 32,768 words or 65,536 bytes.

ADDRESS MAP

As a result of the organization of the PDP-11, bus addresses serve several functions. A map of possible PDP-11 bus address allocation is shown

BUS ADDRESS	CONTENT	
0		
	Program Counter Processor Status Word	Processor Trap Vectors and Device Interrupt Vectors
400_8	Stack Pointer Overflow Limit	
	Stacks, Program and Data Storage	
160000_8		
	Status Register and Data Buffer Register	Typical Registers for Programmed Transfer Device
	Device Address Register Word Count Register Memory Address Register Control and Status Registers	Typical Registers for a Block Transfer Device
177777_8		

Figure 5-1
Simplified Address Allocation Map

in Figure 5-1. Three areas of addresses of particular interest to the programmers are: 1) Interrupt and Trap Vectors; 2) Processor Stack and General Storage; and 3) Peripheral Device Registers.

INTERRUPT AND TRAP VECTORS—Addresses between location zero and location 400_8 are generally reserved for interrupt and trap vectors.

PROCESSOR STACK AND GENERAL STORAGE—Addresses between 400_8 and the limit of implemented core are available for the processor stack or other programs and data. The highest address in this region is 157777_8.

PERIPHERAL DEVICE REGISTERS—Addresses above 160000_8 generally are reserved for peripheral device status, control, and data registers. The general registers and the processor status can be addressed from the program console using addresses in this area.

A more detailed address allocation map can be found in Appendix D.

CORE MEMORY

The three types of core memory that can be used in a PDP-11 system are: 1) Read-Write Core Memory; 2) Read-Only Core Memory; and 3) Wordlet Memory. These memories can be located anywhere in address space provided they do not overlap. They do not have to be in continuous address locations.

MM11-E READ WRITE CORE MEMORY—The MM11-E has the following specifications:

Capacity: 4,096 16-bit words or 8,192 8-bit bytes
Cycle Time: 1.2 microseconds
Access Time: 500 nanoseconds
Configuration: Planer 3-wire, 3-D using 22 mil cores
Packaging: One standard PDP-11 System Unit
Interface: Designed to work with PDP-11 bus, TTL-compatible

MR11-A READ-ONLY CORE MEMORY (ROM)—The ROM has the following specifications:

Capacity: 1,024 16-bit words or 2,048 8-bit bytes
Access Time: 500 nanoseconds
Configuration: 2-piece core with wire braid, 256 wires, 64 cores
Packaging: 3/4 of one standard PDP-11 System Unit
Interface: Designed to work with PDP-11 bus, TTL-compatible

MW11-A WORDLET MEMORY—The wordlet memory is used with ROM systems and provides read-write memory capacity for temporary data and instruction storage.

Capacity: 128 16-bit words or 256 8-bit bytes
Cycle Time: 2.0 microseconds
Access Time: 1.0 microsecond
Configuration: 5-Wire, 3D
Packaging: 1/4 standard PDP-11 single System Unit
Interface: The wordlet memory will work with the ROM and interfaces through the ROM System Unit to the PDP-11 bus.

CHAPTER 6
PROGRAMMING OF PERIPHERALS

Programming of peripherals is extremely simple in the PDP-11—a special class of instructions to deal with input/output operations is unnecessary. The Unibus permits a unified addressing structure in which control, status, and data registers for peripheral devices are directly addressed as memory locations. Therefore all operations on these registers, such as transferring information into or out of them or manpulating data within them, are performed by normal memory reference instruction.

The ability to use all memory reference instructions on peripheral device registers greatly increases the flexibility of input/output programming. Information in a device register can be compared directly with a value and a branch made on the result.

```
CMP  #101,  PRB
BEQ  SERVICE
```

In this case the program looks for 101_8 from the paper tape reader data buffer, and branches if it finds it. There is no need to transfer the information into an intermediate register for comparison.

When the character is of interest, a memory reference instruction can transfer the character into a user buffer in core or in another peripheral device.

```
MOV  PRB,  LOC
```

This instruction transfers a character from the paper tape reader buffer into a user-defined location.

All arithmetic operations can be performed on a peripheral device register.

```
ADD  #10,  DSX
```

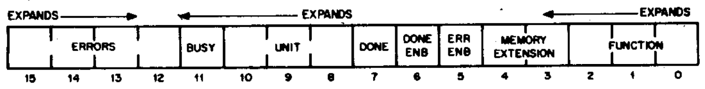
This instruction will add 10_8 to a display's x-deflection register.

All peripheral device registers can be treated as accumulators. There is no need to funnel all data transfers, arithmetic operations, and comparisons through a single or small number of accumulator registers.

DEVICE REGISTERS

All peripheral devices are specified by a set of registers which are addressed as core memory and manipulated as flexibly as an accumulator. There are two types of registers associated with each device: 1) Control and Status Registers (CSR); and 2) Data Registers.

CONTROL AND STATUS REGISTERS (CSR)—Each peripheral has one or more control and status registers which contain all the information necessary to communicate with that device. The general form of a control and status register is shown below.



General Control and Status Register

This general form does not necessarily apply to any device, but is presented as a format which could be used as a guideline for designing peripheral devices. Many devices will require less than sixteen status bits. Other devices will require more than sixteen bits and therefore will require additional status and control registers.

Device Function Bits—These three bits specify operations that a device is to perform. An example of one operation for a paper tape reader is read one character. For a disk one operation would be read a block of words from memory and store them on the disk.

Memory Extension Bits—These two bits are reserved for future expansion. They will allow devices to use a full 18 bits to specify addresses on the bus.

Done Enable and Error Enable Bits—These two bits are independently programmable. If bit 6 is set, an interrupt will occur as a result of a function done condition. If bit 5 is set, an interrupt will occur as the result of an error condition. This occurs when one or more of the error bits is set to a one. To initiate an interrupt routine to read from the paper tape reader, the instruction

```
MOV  #101,  PRS
```

could be used. This sets bit 0 and bit 6 of the paper tape reader control and status register. Setting bit 0 starts the read operation and setting bit 6 enables an interrupt to occur when the read operation is complete.

Condition Bits—The CSR contains a DONE bit, a READY bit, or a DONE-BUSY pair of bits, depending on the device. These bits are set and cleared by the hardware, but may be queried by the program to determine the availability of the device. For example, the teleprinter status register (TPS) has a READY bit (7) that is cleared on request for output and then set when output is complete. The keyboard status register (TKS) has a DONE-BUSY pair (Bits 7 and 11) that distinguishes between no input (DONE = BUSY = 0), input under way (DONE = 0, BUSY = 1), and input complete (DONE = 1, BUSY = 0).

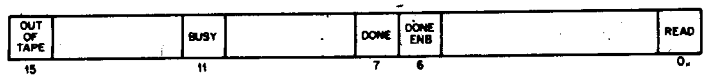
The DONE bit could be used to control an input loop for reading from the paper tape reader as follows:

```
LOOP:  TSTB   PRS     ;  test low byte of paper tape status register
       BPL    LOOP    ;  branch back if DONE bit (bit 7) is not set
```

Unit Bits—Some peripheral systems have more than one device per control. For example, a disk system can have multiple surfaces per control and an analog-to-digital converter can have multiple channels. The unit bits select the proper surface or channel.

Error Bits—Generally there is an individual bit associated with a specific error. When more bits are required for errors, they can be obtained by expanding the error section in the word or by using another status word.

Example of Control and Status Register—The high-speed paper tape reader control and status register (PRS) is as follows:



These bits may be read or set by instructions which use the appropriate effective address. Bit 0 of the PRS is the function bit for reading one character. Incrementing the PRS will set bit 0 and cause one character to be read. The instruction

```
                    INC     PRS
```

performs that function. MOV #1, PRS does the same thing but takes one more word.

DATA BUFFER REGISTERS—Each device has at least one buffer register for temporarily storing data to be transfer into or out of the computer. The number and type of data registers is a function of the device. The paper tape reader and punch use single 8-bit data buffer registers. A disk would use 16-bit data registers and some devices may use two 16-bit registers for data buffers.

PROGRAMMING EXAMPLES

PROGRAM CONTROLLED DATA TRANSFER WITH THE INTERRUPT DISABLED —Single character I/O devices (teletype, paper tape reader/punch) have an addressable register buffer through which data is transferred. For input, the data buffer register is the source operand of the instruction used to get the data; for output, it is the destination operand. For example assuming the paper tape reader interrupt is not enabled, character input could proceed as follows:

```
        MOV   R,  —(SP)          ; save R on the stack
        MOV   #BUFFER,  R        ; pointer to input buffer into register R
START:  INC   PRS                ; start up reader
LOOP:   BIT   PRS,  #100200      ; test DONE and ERROR bits
        BEQ   LOOP               ; branch back if none on yet
        BMI   ERROR              ; branch to error routine if minus
        MOVB  PRB,  (R)+         ; move byte from device buffer reg-
                                 ; ister to user's buffer and increment
                                 ; pointer
        CMP   #LIMIT  R,         ; check for end of buffer
        BGE   START              ; get next character
        MOV   (SP)+,  R          ; restore R
```

Character output to the paper tape punch might be executed as follows:

```
        MOV   R0,  —(SP)         ; save R0
        MOV   R1,  —(SP)         ; save R1
        MOV   NCHAR,  R0         ; number of characters into R0
        MOV   BUFFER,  R1        ; user buffer address into R1
LOOP:   BIT   PPS,  #100200      ; test device ready and error bits
        BEQ   LOOP               ; fall through if on
        BMI   ERROR              ; branch on error
        MOVB  (R1)+,  PPB        ; output character, increment pointer
        DEC   R0                 ; decrement character counter (and
                                 ; set condition codes)
        BGT   LOOP               ; repeat if greater than zero
        MOV   (SP)+,  R0         ; restore R0
        MOV   (SP)+,  R1         ; restore R1
```

BLOCK TRANSFER WITH THE INTERRUPT DISABLED—High-speed block transfer devices use the Unibus to make data transfers between the device and core memory. These devices are provided with addressible registers that control the flow of data.

A typical set might be:

1. Control and status register
2. Memory address register
3. Word count register
4. Device address register

Loading the device address register would in general initiate the transfer, which then proceeds without processor intervention. The device issues non-processor requests for the Unibus that, when granted, allow direct data transfer between the device and memory. These requests are interleaved with processor requests for the bus. If very fast transfer is required, the processor may execute a WAIT instruction after starting the block transfer.

The DONE or appropriate error bits are set in the CSR with completion of the transfer or when an error occurs. These may be enabled to cause an interrupt or may be tested to determine when the device needs assistance.

A block transfer could be executed as follows:

```
    MOV   #401,    DKS,          ;  read block of data (function 1)
                                 ;  from unit 1
    MOV   #BUFADR,   DKMA        ;  buffer address to memory ad-
                                 ;  dress register
    MOV   #BUFCNT,   DKWC        ;  word count to word count register
    MOV   #BLKNO,   DKDA         ;  block number to device address
                                 ;  register, which starts the trans-
                                 ;  fer
          .
          .
          .
          .
    ; when data is needed.

    LOOP:  BIT   #DKMSK,   DKS   ;  test done bit and error bits
           BEQ   LOOP            ;  branch back if none on
           BIT   #DKEMSK,   DKS  ;  test for any error bits
           BNE   ERROR           ;  branch if any on
    ; data is now in buffer at BUFADR
```

INTERRUPT STRUCTURE

If the appropriate interrupt enable bit is on, in the control and status register of a device, transition from 0 to 1 of the DONE or READY bit causes an interrupt request to be issued to the processor. Also if DONE or READY is a 1 when the interrupt enable is turned on, an interrupt request is made. If the device makes the request at a priority greater than that at which the processor is running and no other conflicts exist, the request is granted and the interrupt sequence takes place:

a. the current program counter and processor status are pushed onto the processor stack;
b. the new PC and PS are loaded from a pair of locations (the interrupt vector) in low core unique to the interrupting device.

Since each device has a unique interrupt vector which dispatches control to the appropriate interrupt handling routine immediately, no device polling is required. Furthermore, since the PS contains the processor priority, the priority at which an interrupt request is serviced can be set under program control and is independent of the priority of the interrupt request. The

ReTurn from Interrupt instruction is used to reverse the action of the interrupt sequence. The top two words on the stack are popped into the PC and PS, returning control to the interrupted sequence.

PROGRAMMING EXAMPLE

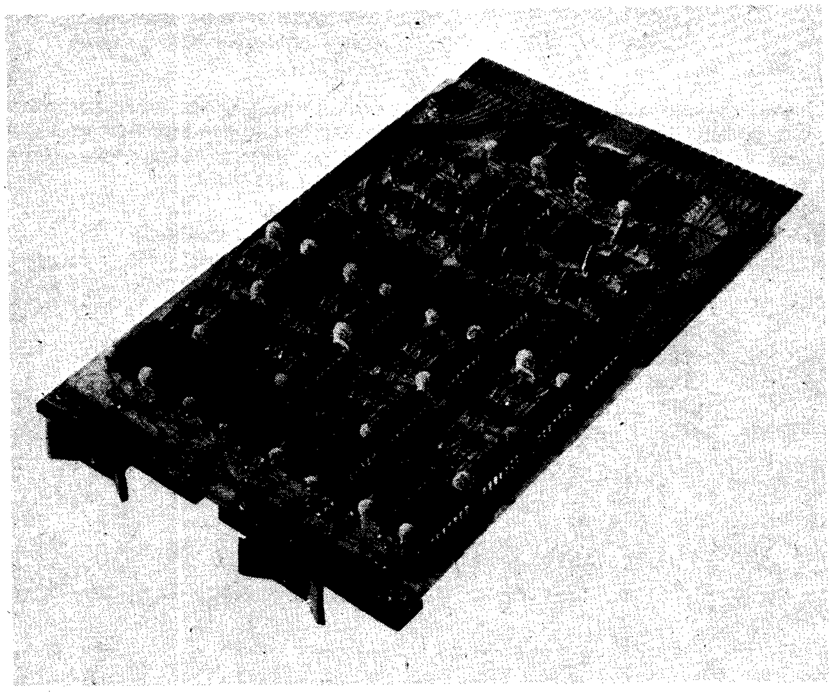
A paper tape reader interrupt service could appear as follows:

First the user must initialize the service routine by specifying an address pointer and a word count

```
INIT:     MOV #BUFADR, #0        ; set up address pointer
          POINTR = . - 2         ; in third word of MOV instruction.
          MOV #CNTR, #0          ; set up character count in
          CRCNT = . - 2          ; third word of MOV instruction.
          MOV #101, PRS          ; read a character with interrupt
                                 ; enabled.
```

When the interrupt request occurs and is acknowledged, the processor stores the current PC and PS on the stack. Next it picks up the interrupt vector or new PC and PS beginning at location 70_8. The next instruction executed is the first instruction of the device service routine at PRSER.

```
PRSER:    TST     PRS            ; test for error
          BMI     ERROR          ; branch to error routine if
                                 ; bit 15 of PRS is set.
          MOVB    PRB, @POINTR   ; move character (byte)
                                 ; from reader to buffer
          INC     POINTR         ; increment pointer
          DEC     CRCNT          ; decrement character count
          BEQ     DONE           ; branch when input done
          INC     PRS            ; start reader for next character
DONE:     RTI                    ; return from interrupt
```


The DIGITAL M225 module contains 8 high speed general-purpose registers. The M225 general registers provide program flexibility when used as accumulators, index registers, and pointers to data words.

CHAPTER 7

TELETYPE (MODEL LT33-DC/DD)

The standard Teletype Model 33 ASR (Automatic Send-Receive) can be used to type in or print out information at a rate of up to ten characters per second, or to read in or punch out perforated paper tape at a ten characters per second rate. Signals transferred between the 33 ASR and the control logic are standard serial, 11-unit code Teletype signals. The signals consist of "marks" and "spaces" which correspond to idle and bias current in the Teletype serial line, and to 0's and 1's in the control and computer. The start mark and subsequent eight bits are each one unit of time duration and are followed by the stop mark which is two units.

The 8-bit code used by the Model 33 ASR Teletype unit is the Americal Standard Code for Information Interchange (ASCII) modified. To convert the ASCII code to Teletype code, add 200 octal (ASCII + 200_8 = Teletype).

The Model 33 ASR can generate all assigned codes except 340 through 374 and 376. The Model 33 ASR can detect all characters, but does not interpret all codes that it can generate as commands. The standard number of characters printed per line is 72. The sequence for proceeding to the next line is a carriage return followed by a line feed. Punched tape format is as follows:

Tape Channel	87	654	S	321
Binary Code (Punch = 1)	10	110		100
Octal Code (S = Sprocket)	2	6		4

SIZE— Floor space approximately 22¼" wide, 18½" deep
Cable length 8 feet

MODEL	**POWER REQUIREMENTS**	
LT33-DC	115 V ±10%	60 ±0.45 Hz
LT33-DD	230 V ±10%	50 ±0.75 Hz

TELETYPE CONTROL (MODEL KL11)

TELETYPE CONTROL—Serial information read or written by a Teletype unit is assembled or disassembled by the control for parallel transfer on the Unibus. The control also provides the flags which cause a priority interrupt and indicate the availability of the teletype.

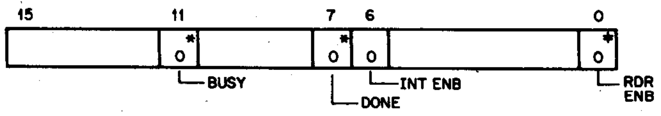
KEYBOARD/READER—The Teletype control contains an 8-bit buffer (TKB) which assembles and holds the code for the last character struck on the keyboard or read from the tape. Teletype characters from the keyboard/reader are received serially by the 8-bit shift register TKB. The code of a Teletype character is loaded into the TKB so that "spaces" correspond to binary 0's and holes, "marks," correspond to binary 1's. Upon program command, the contents of the TKB may be transferred in parallel to a memory location or a general register.

A character is read from the low-speed reader by setting the Teletype reader enable bit, (RDR ENB), to a 1. This sets the busy bit (BUSY) to a 1. When a Teletype character starts to enter, the control de-energizes a relay in the

Teletype unit to release the tape feed latch. When released, the latch mechanism stops tape motion only when a complete character has been sensed, and before sensing of the next character is started. When the character is available in buffer (TKB), the busy bit (BUSY) is cleared and the done flag (DONE) is set. If the interrupt is enabled, a request is made for the bus at level 4 (BR4). The interrupt vector is at location 60_8. The DONE bit is cleared by any instruction which reads the contents of the buffer (TKB) into the processor. If the DONE flag is cleared before the interrupt is granted, no interrupt will occur. The keyboard must be read within 18 milliseconds of DONE to ensure no loss of information.

Registers[1]

Teletype Keyboard Status (TKS)

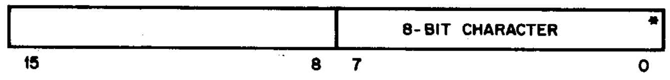


Bit

0	RDR ENB	Requests that one character be read from the reader; set from the bus: (Note: Setting RDR ENB causes tape to advance by one character which is shifted into TKB if DONE is cleared.) Receipt of START bit on the serial input line sets BUSY, clears RDR ENB and clears TKB.
6	INT ENB	0—No interrupt; 1—Attach the keyboard and reader to the priority interrupt system at bus request level 4.
7	DONE	Character available; cleared by reading the buffer (TKB).
11	BUSY	Character is being read; set by RDR ENB going to a 1. Cleared by DONE going to a 1.

1 The following notation will be used throughout this chapter for describing registers.
0 — A power clear sets this bit to 0.
1 — A power clear sets this bit to 1.
* — This bit can only be read from the bus.
‡ — This bit can only be set from the bus. If it is read, it will always appear as zero.

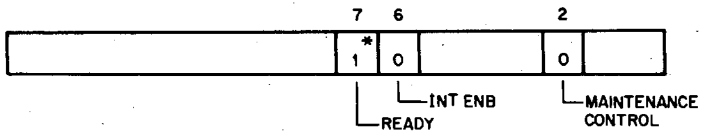
Teletype Keyboard Buffer (TKB).



TELEPRINTER/PUNCH—On program command, a character is sent in parellel from a memory location (or a general register) to the TPB shift register for transmission to the teleprinter/punch unit. The control generates the start "space," then shifts the eight bits serially into the Teletype unit, and then generates the stop "marks." This transfer of information from the TPB into the teleprinter/punch unit is accomplished at the normal Teletype rate and requires 100 milliseconds for completion. The READY flag in the teleprinter/punch indicates that the TPB is ready to receive a new character. A maintenance mode is provided which connects the TPB output to the TKB input so that the parallel serial and serial parallel shifting may be verified.

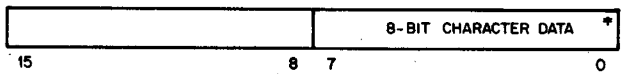
Registers

Teleprinter Status Word (TPS)



Bit		
2	MAINT	Maintenance function which connects TPB serial output to TKB serial input.
6	INT ENB	0—No interrupt; 1—attaches the Teleprinter to the priority interrupt system at BR4.
7	READY	Set by punch/printer DONE; cleared by loading the teleprinter buffer (TPB).

Teleprinter Buffer (TPB)



PROGRAMMING EXAMPLE—To read a character from tape and echo it on the printer:

```
ECHO:     INC TKS            ; set RDR ENB
          TSTB TKS           ; test for DONE set
          BPL .—2            ; test again if not set
          TSTB TPS           ; test for printer READY set
          BPL .—2            ; test again if not set
          MOVB TKB, TPB      ; put input character into output
                               buffer to be printed
          BR ECHO            ; return for another character
```

PERIPHERAL ADDRESS ASSIGNMENTS

TKS	177560
TKB	177562
TPS	177564
TPB	177566

VECTOR ADDRESS	Keyboard/Reader	60
	Teleprinter/Punch	64

PRIORITY LEVEL set to BR4—Teletype printer is lower than the Teletype keyboard

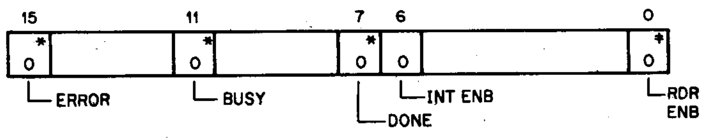
MOUNTING—Requires one small peripheral controller mounting space (¼ of a DD11 or one of two such spaces in KA11)

HIGH-SPEED PERFORATED TAPE READER PUNCH AND CONTROL (TYPE PC11)

TAPE READER—This device senses 8-hole perforated paper or Mylar tape photo-electrically at 300 characters per second. The reader control requests reader movement, transfers data from the reader into the reader buffer (PRB), and signals the computer when incoming data is present. It does this by setting a DONE bit. If the interrupt is enabled and the interrupt is granted, the processor traps to location 70_8 and may immediately begin executing the service routine for the paper tape reader.

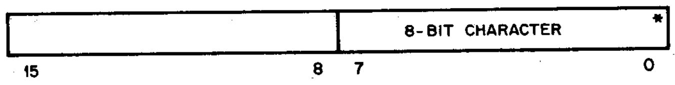
Registers

Paper Tape Reader Status Word (PRS)



Bit		
0	RDR ENB	Requests read of next character; can be set from bus only if ERROR = 0. Clears PRB, sets BUSY.
6	INT ENB	0—No interrupt; 1—attached to priority interrupt system at BR4. (Note: Interrupt occurs when INT ENB is a 1 and either the error flag, ERROR, or the done flag, DONE, becomes a 1.)
7	DONE	Set by character available; cleared by reading the paper tape reader buffer (PRB).
11	BUSY	Set by RDR ENB going to a 1; cleared by DONE going to a 1.
15	ERROR	Error Flag — Set or cleared by out-of-tape sensor or off line switch.

Paper Tape Reader Buffer (PRB)



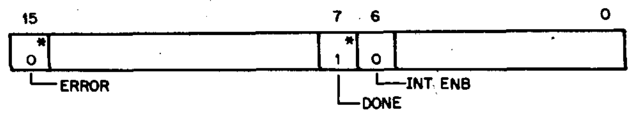
PROGRAMMING EXAMPLE—Tape reading subroutine (not using interrupt):

```
READ:    INCB PRS           ; enable reader
TEST:    BIT #100 200 PRS   ; test for error or done
         BEQ TEST           ; branch back if not done
         BMI ERROR          ; branch if error = 1
         MOVB PRB, RO       ; get character from buffer
         RTS R              ; return to caller
ERROR:   (message type out routine)
         HALT               ; wait for operator intervention
         JMP READ           ; try again when continue switch is hit.
```

TAPE PUNCH—This option of a Royal McBee paper tape punch that perforates 8-hole tape at a rate of 50 characters per second. Information to be punched on a line of tape is loaded in an 8-bit punch buffer (PPB) from a memory location or one of the general registers. The punch flag, READY, becomes a 1 at the completion of punching action, signaling new information may be transferred into the punch buffer and punching initiated.

Registers

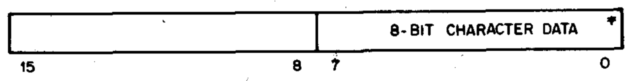
Paper Tape Punch Status Word (PPS)



Bit

6	INT ENB	0—No Interrupt; 1—Attached to priority interrupt system. (Note: An interrupt occurs when INT ENB is a 1 and either the ERROR flag or the READY flag becomes a 1.)
7	READY	Set by punch done; cleared by loading the paper tape punch buffer (PPB).
15	ERROR	Error Flag—Set by out-of-tape sensor, or unit power off switch.

Paper Tape Punch Buffer (PPB)



Loading the buffer initiates punching.

PROGRAMMING EXAMPLE

```
PUNCH:    BIT #100200, PPS   ;   test for ready or error
          BEQ PUNCH
          BMI ERROR
          MOV R0, PPB        ;
          RTS R              ;
ERROR:    (message type out)
          HALT; wait for operator to fix punch
          JMP PUNCH; try again when Continue is hit.
```

PERIPHERAL ADDRESS ASSIGNMENTS

PRS	177550
PRB	177552
PPS	177554
PPB	177556

VECTOR ADDRESSES—Reader 70
Punch 74

PRIORITY LEVEL—Set to BR4. Punch is lower than reader.

MOUNTING—Electromechanical assembly—EIA Standard 19" rack, 10½" vertical mounting space, by 17½" deep.

PC11-M Controller—One small peripheral controller mounting space (¼ of DD11 or one of two such places in KA11).

ENVIRONMENTAL

55°—100°F

20% —95% RH (without condensation)

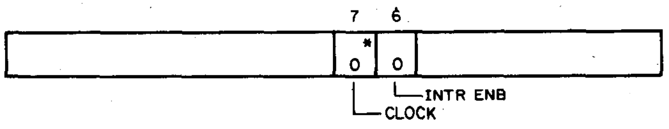
MODEL	DESCRIPTION	POWER REQUIREMENTS
PC11	Reader, Punch & Control	115±10% 60 Hz
PC11A	Reader, Punch & Control	115±10% 50 Hz
PR11	Reader & Control	115±10% 50-60 Hz

LINE FREQUENCY CLOCK (TYPE KW11-L)

The KW11-L real time clock provides a method of measuring time intervals at line frequency. This clock consists of a frequency source and control logic. When enabled the clock causes an interrupt every 16.6 or 20 milliseconds, depending upon line frequency.

Register

Line Time Clock Status Register (LKS)



Bit

6 INTR ENB	When set, an interrupt will occur every time CLOCK goes true. Cleared by program or reset or start sequence.
7 CLOCK	Set to 1 every 16.6 milliseconds (60 Hz) or 20 milliseconds (50 Hz). Cleared by reading LKS, RESET or pressing the START switch.

PERIPHERAL ADDRESS ASSIGNMENTS

LKS	177546
VECTOR ADDRESS	100
PRIORITY LEVEL	BR6

MOUNTING—This option plugs into the KA11 processor.

CHAPTER 8
DESCRIPTION OF THE UNIBUS

Communication between all system units in a PDP-11 configuration is done by a single common bus: the Unibus. All communication—both instructions and logical operations—is defined by a set of 56 signals. This set of 56 signals is used for program controlled data transfers, direct memory data transfers, priority bus control, and program interrupt.

This chapter presents the concepts of the Unibus and how they affect program software and interfacing hardware. The use of the 56 bus signals to effect data transfers and to control bus use is also described.

GENERAL CONCEPTS OF THE UNIBUS

There are five major aspects of the Unibus that affect both software and hardware considerations in the PDP-11.

SINGLE BUS—The set of 56 signals that comprise the Unibus is the one and only bus connecting all peripheral devices, memories, and the central processor. Thus, to every device there exists a single set of signals by which it can be interrogated by the processor or other devices, or be used by the device itself to transfer data to and from memory.

The processor uses this same set of signals to communicate with all memories and devices. The important point here is that the form of the communication used by processor and peripheral devices is identical. Consequently, the same set of program instructions used to reference memory is used to reference peripheral devices. (A look at the PDP-11 instruction set will reveal that there are no explicit I/O instructions.)

Peripheral devices in a PDP-11 system are designed to respond to the Unibus in the same manner as memory. Device status registers, device control registers, and device data registers are each assigned unique "memory" addresses. For example, the instruction MOVB R0, PUNCH would load the punch buffer register with an 8-bit character contained in R0. Other instructions would monitor the punch status and the program could determine when the punching operation was complete.

BIDIRECTIONAL BUS—Unibus bus signals are bidirectional—the signal received as an input can be driven as an output, as shown in Figure 8-1.

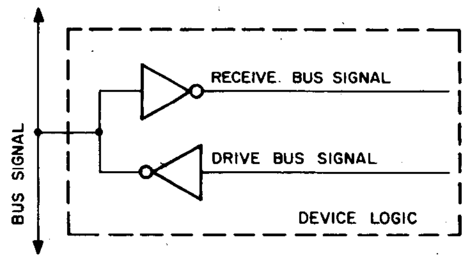


Figure 8-1 Bidirectional Nature of the Bus

MASTER-SLAVE RELATION—At any one point in time, there is one device, called the master, that has control of the bus. The master device controls the bus to communicate with other devices, called slaves, on the bus. An example of this relationship is the processor (master) fetching an instruction from memory (which is always a slave).

INTERLOCKED COMMUNICATION—*For each control signal issued by the* master device, there is a response from the slave; thus bus communication is independent of the physical bus length and the response time of the master and slave devices. Also, master-slave relationships can exist in nearly any combination between fast-responding and slow-responding devices.

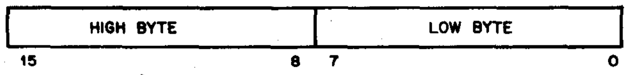
DYNAMIC MASTER-SLAVE RELATION—Master-slave relationships are dynamic. The processor, for example, can pass bus control to a disk. The disk, as master, could then communicate with a slave memory bank.

UNIBUS SIGNALS

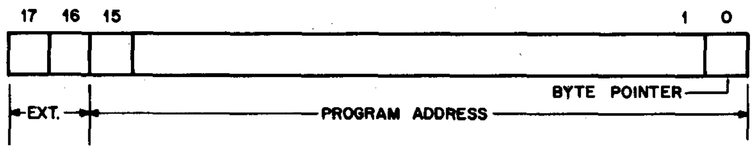
The 56 Unibus signals can be divided into two major groups—the interrupt group and the non-interrupt group. The interrupt group can then be subdivided into two classes—the request and control class and the grant class. All bus signals except the grant class are bidirectional in nature and are connected to every device (though they may not be used by every device). The grant signals, because of their special nature in priority bus control (to be explained later), are bussed through each device and are unidirectional in nature.

NON-INTERRUPT SIGNALS

Data Lines (D $<$ 15:00 $>$)—(Note that the notation A <a:b> specifies b — a + 1 signal lines which are named Aa through Ab.) The 16 data lines are used to transfer information between master and slave. This is the bit format:



Address Lines (A $<$ 17:00 $>$)—The 18 address lines are used by the master device to select the slave (a unique core memory or device register address) with which it will be communicating. This is the bit format of the 18 signals:



A $<$ 15:01 $>$ are used to specify a unique 16-bit word group. In byte operations, A00 is used to specify the byte being referenced. If a word is referenced at X (X must be even, since **words can be addressed on even boundaries only)**, the low byte can be referenced at X and the high byte at X + 1.

A $<$ 15:00 $>$ are supplied by the software as memory reference addresses. A17 and A16 are used as extended memory bits for relocation and as protection schemes in future systems. In the PDP-11/20 and the PDP-11/10, A17 and A16 are asserted or forced to 1 whenever an attempt is made to reference a memory location where A15 = A14 = A13 = 1. Thus the hardware converts the 16-bit software address to a full 18-bit bus address.

An address map is shown in Figure 8-2.

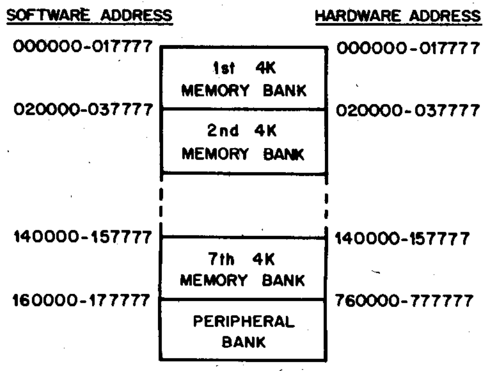


Figure 8-2 Address Map

The peripheral bank is composed of the processor's fast memory, status register, console switch register, and all device registers.

Control Lines (C < 1:0 >)—These two bus signals are coded by the master device to indicate to the slave one of four possible data transfer operations.

Master Synchronization and Slave Synchronization (MSYN, SSYN)—MYSN is a control signal used by the master to indicate to the slave that address and control information is present. SSYN is the slave's response to MSYN

Initialization (INIT)—This signal is a power clear signal asserted by the console and the processor which is used to reset peripheral devices.

PA, PB, SP1, SP2—These lines are not implemented on the PDP-11/10 or PDP-11/20.

INTERRUPT SIGNALS

Bus Request Lines (BR < 7:4 >)—These four bus signals are used by peripheral devices to request control of the bus.

Bus Grant Lines (BG < 7:4 >)—These signals are the processor's response to a BR. They will be asserted only at the end of instruction execution.

Non-Processor Request (NPR)—This is a bus request from a peripheral device to the processor.

Non-Processor Grant (NPG)—This is the processor's response to an NPR. It occurs at the end of bus cycles within the instruction execution.

Selection Acknowledge (SACK)—SACK is asserted by a bus-requesting device that has received a bus grant. Bus control will pass to this device when the current master of the bus completes its operations.

INTERRUPT (INTR)—This signal is asserted by the master to start program interruption in the processor.

Bus Busy (BBSY)—This signal denotes bus in use by a master device.

UNIBUS DATA TRANSFER OPERATIONS

Direction of data transfers on the Unibus is defined in relation to the master device. A data transfer from processor to memory (always a slave) is "data out," and a transfer from memory to processor is "data in."

TYPES OF DATA TRANSFERS—The type of data transfer being made between master and slave is determined by the C lines coded as follows:

C1	C0	
0	0	DATI- DATa In
0	1	DATIP-DATa In,Pause
1	0	DATO -DATa Out
1	1	DATOB-DATa Out, Byte

DATO AND DATOB—The DATO and DATOB operations are used to transfer data out of the master to the slave. DATO is used to transfer a word to the address specified by A < 17:01 >. The slave ignores A00 and the data appears on D < 15:00>. DATOB is used to transfer a byte of data to the address specified by A < 17:00 >. A00 = 0 indicates the low byte, and data appears on D < 07:00 >; A00 = 1 indicates the high byte, and data appears on D < 15:08 >.

DATI AND DATIP—The DATI and DATIP operations transfer data from a slave whose address is specified on A < 17:01 > into the master. Both transfers are made in words on D < 15:00 >. In destructive read-out devices, DATI commands a read-write operation, while a DATIP commands a read operation only and sets a pause flag. When the device receives the subsequent DATO or DATOB and its pause flag is set, the usual read cycle is skipped and an immediate write cycle is initiated. Thus, DATIPs are immediately followed by a DATO or DATOB to effect a read-modify-write data exchange. In non-destructive read-out devices, DATI and DATIP are treated identically.

This diagram illustrates the data flow in the four data transfers:

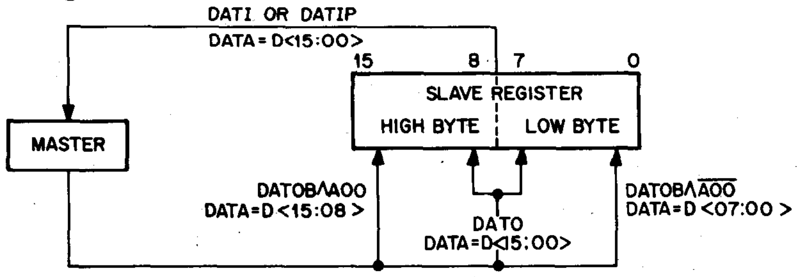


Figure 8-3 Data Flow

Note that all transfers into the master are word operations; it is up to the master to accept the appropriate byte. On a DATOB, the master must place the byte on the appropriate data lines; the slave must accept the proper byte.

DATA TRANSFER EXAMPLES—The bus operations used by the processor for a typical instruction sequence illustrates how the data transfer operations are used. The "program" starts at location 1000:

```
1000:    INCB @R0
         ADD #3, @R0
```

where R0 contains 500 and location 500 contains 10023. The result of this instruction sequence will leave 10027 in location 500. In binary form, this coding appears as:

```
1000:       105210      ;INCB @R0
1002:       062710      ;ADD (PC)+, @R0
1004:       000003      ;3
```

The following table lists the bus operations that result as a consequence of these two instructions:

Processor Cycle	Bus Operation	Bus Address	Data Transferred
1. Fetch	DATI	(PC) = 001000	105210
2. Destination	DATIP	(R0) = 000500	010023
3. Execute	DATOB	(R0) = 000500	000024
4. Fetch	DATI	(PC) = 001002	062710
5. Source	DATI	(PC) = 001004	000003
6. Destination	DATIP	(R0) = 000500	010024
7. Execute	DATO	(R0) = 000500	010027

Note that in step 3, it is inconsequential what data appears on D < 15:08 >; the slave accepts only the modified low byte.

A second example of bus operation compares the contents of the Teletype keyboard data buffer whose address is 177560 with the ASCII value for the letter "A."

```
200:    CMPB @#177560, #301
```

This instruction is assembled in three words as follows:

```
200:    123727      ;CMPB @(R7)+, (R7)+
202:    177560      ;Address of data buffer
204:    000301      ;301
```

The processor will execute this instruction with these cycles:

Processor Cycle	Bus Operation	Bus Address	Data Transferred
1. Fetch	DATI	(PC) = 200	123727
2. Source	DATI	(PC) = 202	177560
3. Source	DATI	777560	ASCII
4. Destination	DATI	(PC) = 204	000301
5. Execute	none — condition codes set internally.		

Note that in step 3, the software specified address 177560 was converted to the bus address 777560.

SIGNAL DESCRIPTION OF DATA TRANSFERS—Figure 8.4(a) shows the signal flow between master and slave during a DATO operation. (The sequence is similar for DATOB except that only a byte of information is transferred.) The master sets Control for DATO, sets Address for the unique slave address, and sets Data for the information to be transferred. The master then asserts MSYN. This signal is received by the slave that recognizes its address; it responds by accepting the data and asserting SSYN. SSYN is received by the master which then negates Control, Address, Data, and MSYN. The slave sees MSYN negated and negates SSYN. The master device continues its operation when it sees SSYN negated.

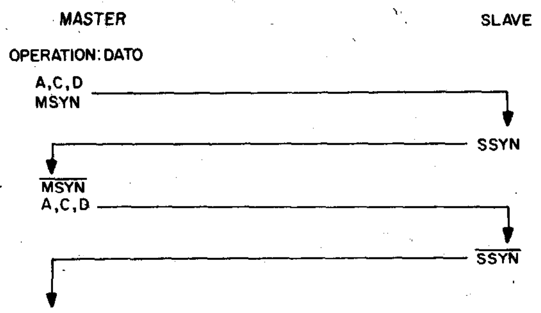
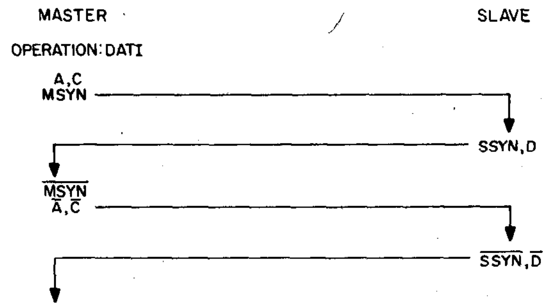


Figure 8-4(a)

The flow of signals for DATI is shown in Figure 8.4(b). (DATIP is similar except that the internal operation of the slave device is modified.) The master sets Control for DATI, sets Address for the slave to be selected, and asserts MSYN. The selected slave responds by setting Data for the information requested and asserts SSYN. The master sees SSYN, accepts the data, and then negates Control, Address, and MSYN. The slave sees MSYN negated and negates SSYN. The master continues when it sees SSYN negated.

A more detailed signal sequence for the DATI, DATIP, DATO, and DATOB bus operations can be found in Appendix D.



MASTER
SLAVE
OPERATION: DATI
A,C
MSYN
SSYN,D
$\overline{\text{MSYN}}$
$\overline{\text{A}},\overline{\text{C}}$
$\overline{\text{SSYN}},\overline{\text{D}}$

Figure 8-4(b)

UNIBUS CONTROL OPERATIONS

The following section will deal with how a device becomes master of the bus and how control of the bus is transferred from one device to another. Two additional bus operations will be presented—the PTR (Priority Transfer) and INTR (Interrupt).

In normal operation, the processor is bus master, fetching instructions and operands from memory. Other devices on the bus have the capability of becoming bus master, and use the bus for one of two purposes: 1), to gain direct memory access or 2), to interrupt program execution and force the processor to branch to a specific address.

PRIORITY ARBITRATION—Transfer of bus control from one device to another is determined by a priority scheme in which three factors must be considered.

First, the processor's priority is determined by bits 7, 6, and 5 in the processor status register. These three bits set a priority level that inhibits granting of bus requests on lower levels.

Second, bus requests from external devices can be made on one of five request lines. NPR has the highest priority, and its request is honored by the processor between bus cycles of an instruction execution. BR7 is the next highest; BR4 is the lowest. These four lower level requests are honored by the processor between instructions, except when the instruction currently being executed causes an internal trap (either an error or trap instruction). In this case, BR requests will not be honored until completion of the first instruction after the trap sequence. Thus if two requests are made to the processor for bus control, the higher of the two requests will be honored first.

Third, in response to a bus request, the processor may honor the request by asserting a bus grant (BG) corresponding to the line on which the bus request was made. This signal is passed serially through each device in the system. If a device had made a request, it would block the grant signal and prevent it from reaching the following devices. Thus, in this "pass-the-pulse" chain, the device that is closest to the processor has the highest priority on that request level.

This table lists device priorities:

Highest:	Devices on NPR
	Processor when priority = 111
	Devices on BR7
	Processor when priority = 110
	Devices on BR6
	Processor when priority = 101
	Devices on BR5
	Processor when priority = 100
	Devices on BR4
	Processor when priority = 011
	Internal options
	Processor when priority = 010
	Internal options
	Processor when priority = 001
	Internal options
Lowest:	Processor when priority = 000

When the processor's priority is set at N, all requests for bus control at level N and below are ignored.

SELECTION OF NEXT BUS MASTER—The signal sequence by which a device becomes selected as next bus master is the PTR (Priority Transfer) bus operation. Note that this operation does not actually transfer bus control; it only selects a device as **next** bus master. It takes one additional condition to complete the transfer: the current bus master must complete its bus operations. The signal that indicates this is BBSY. Thus, when a device makes an NPR or BR request to the processor for bus control, it waits until it first becomes selected as next bus master by the PTR operation and second, it no longer senses BBSY. The negation of the BBSY signal indicates that the current master has completed its bus operation. The selected device now becomes bus master and asserts BBSY itself.

INTERRUPT SEQUENCE—Once the device has bus control and is asserting BBSY itself, it is sole user of the bus until it releases its control. This release of control can be made either actively or passively. Passive release is realized by negating BBSY. Bus control will then pass to either a device that was selected in the meantime by another PTR sequence or back to the processor, which will continue where it was interrupted. Active release of bus control is realized through the INTR bus sequence.

The INTR (interrupt) operation is used by the bus master to transfer to the processor a memory address (called the interrupt vector). Two consecutive words, the starting address of an interrupt service routine and a new status word, are stored at the interrupt vector address. After the INTR sequence is complete, the processor automatically becomes bus master and begins a trap sequence in which it stores the current value of the PC and PS on the stack and fetches a new PC and PS from the location pointed to by the interrupt vector. Thus, the next instruction executed is the start of the interrupt service routine.

It is illegal to issue an INTR command after gaining control of the bus by requesting on an NPR line. NPR requests are granted during instruction execution and external bus masters must restrict their bus use to nonprocessor activities.

Interrupt Servicing Sequence Example—The following is an example of the INTR sequence.

When a peripheral requires service and requests control of the bus with a BR signal, the operations undertaken to "service" the device are as follows:

● Gain Control of the Bus—When the processor has no higher priority tasks to complete, it relinquishes the bus to that device. Higher priority items are (in order of priority):

1. Acknowledging an NPR request
2. Handling a processor error (illegal instructions, requirements for non-existent memory, etc.)
3. Completing the current instruction
4. Acknowledging a trace trap
5. Continuing a higher priority process
6. Acknowledging a higher level BR signal
7. Acknowledging same level BR signals for devices closer to the processor

● Do INTR Sequence—when the device has control of the bus, it initiates an INTR sequence, transferring to the processor the interrupt vector address which specifies two words in memory containing the address and status of the appropriate device service routine.

● Push Old Interrupt Vector Onto Stack—The processor then "pushes"—first, the current central processor status (PS) and then the current program counter (PC) onto the processor stack.

● Fetch New Interrupt Vector—The new PC and PS (the "interrupt vector") are taken from the address specified by the device, and the device service routine is begun. Note that those operations all occur automatically and that no device polling is required to determine which service routine to execute.

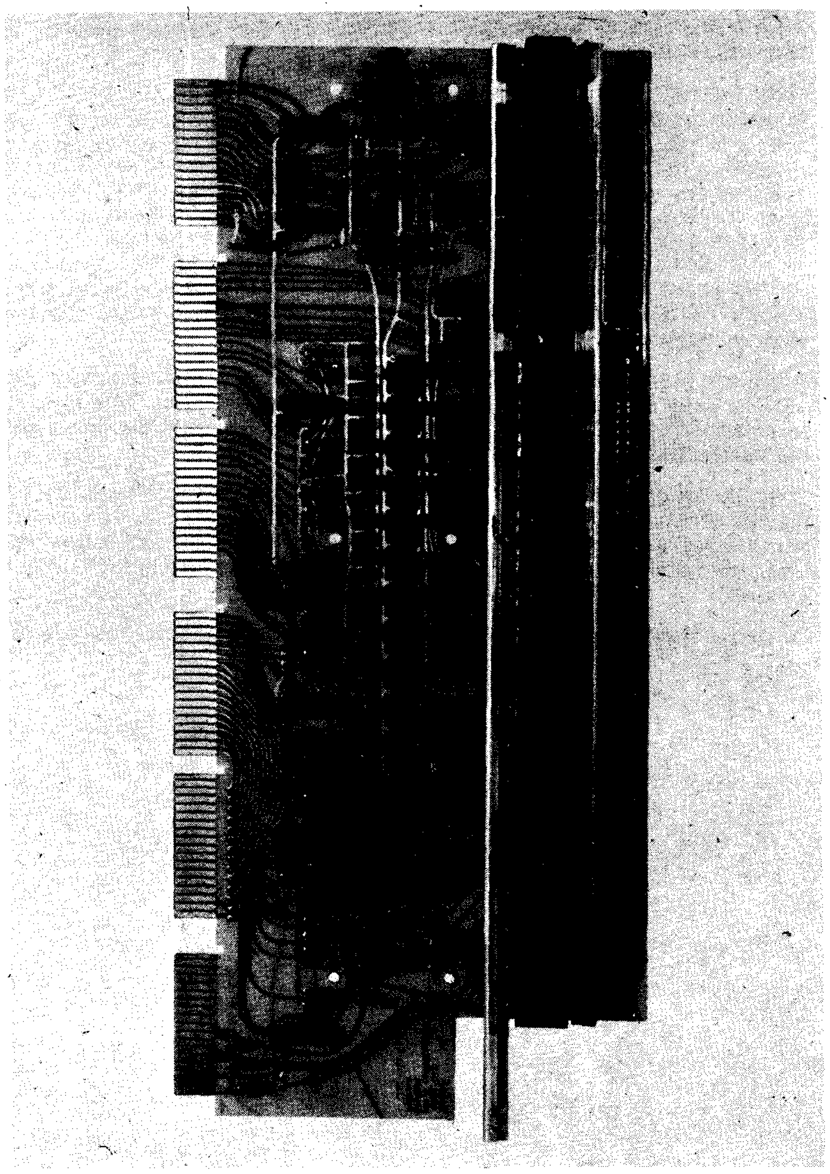
Example of NPR Operation—Disk operation gives an example of a device which uses the bus for direct memory access. Under program control, the processor would initialize registers in the disk control that specify word count (WC, number of words in block of data to be transferred), memory address (MA, the address at which the block of data is found or is loaded), and Track Address (TA, the point on the disk where the block of data starts). Also, the program would set certain function bits in the disk's command and status register that specify a read or write function. For this example, assume the disk was set to read.

Once the disk's control registers are initialized, the disk control logic starts a search for the requested data. (The processor in the meantime has continued in its program execution.) When the disk has found the data, it assembles the first 16-bit word from the disk surface into its data register. The disk now requests bus control via the NPR request line. The processor, when it has completed its current bus cycle of the current instruction and no higher NPR requests exist, grants control of the bus to the disk. The disk, as bus master, effects a DATO bus operation, transferring the contents to its data buffer to the core address held in its MA. The MA is now incremented and the WC is decremented. When the DATO operation is complete, the disk passively releases control of the bus.

When the second word has been assembled, the disk again requests bus control, does a data transfer, and then releases bus control. This cycle is repeated until the WC reaches zero. At this point, the disk has completed the transfer that was requested.

To notify the program that the transfer is finished, the disk initiates a request for bus control at the BR level, gains control when higher priority requests are satisfied, and does an immediate INTR to the processor and causes the program to branch to a specific service program (as described in the previous example).

Details of the INTR and PTR bus operations can be found in Appendix D.



The plug-in console board with modular construction is supplied in the basic 11/20 configuration. In addition to aiding programming, console contributes to ease of maintenance on the PDP-11.

CHAPTER 9
Interfacing

A typical device bus interface as shown in Figure 9-1 is composed of five major components: 1), Registers; 2), Bus Drivers and Receivers; 3), Address Selector; 4), Interrupt Control; and 5), Device Control Logic.

REGISTERS

Each device is assigned bus addresses at which the program can interrogate and/or load the device status, control, and data registers. The standardized mapping for these registers and the bit assignments of the command/status register (CSR) were given in Chapters 5 and 6.

As shown in Figure 9-1, all information flow between the device logic and the Unibus is done through the registers. In general, registers are designed to be both loadable and readable from the bus. This allows the program to use such instructions as ADD R0, REG, or INC REG. However, registers can be "one-sided," either "read-only" or "write-only." Examples of read-only bits are the DONE and BUSY flags in the device's CSR. These bits are derived from the internal state of the device logic and are not under direct program control. Write-only registers are used when it is unnecessary to read back information. Attempting to read such a register would result in an all-zero transfer. The instructions effective with this type of register are then limited to those which load the register such as MOV R0, REG, or CLR REG (as opposed to ADD REG, R0, or INC REG).

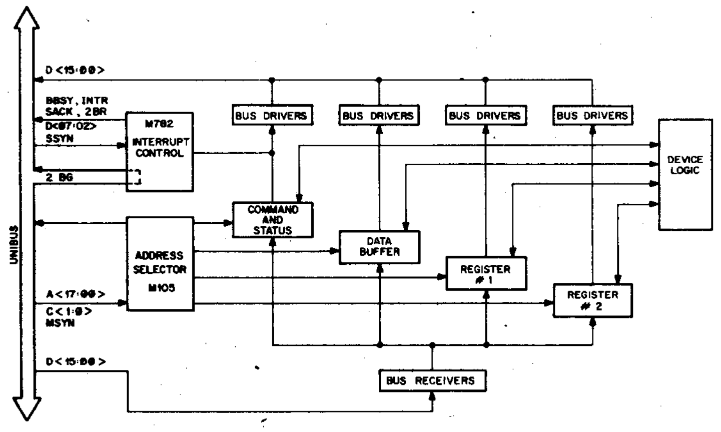
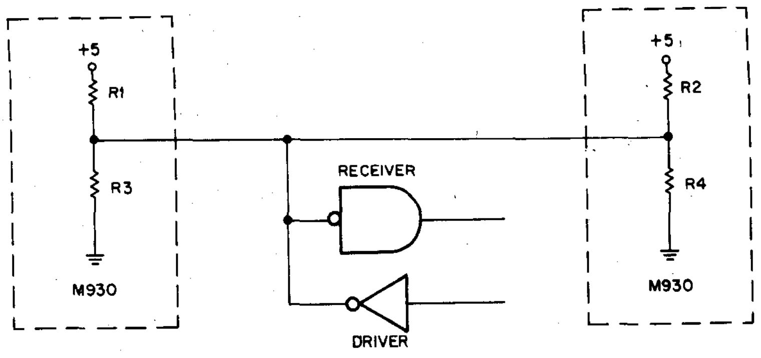


Figure 9.1 Typical Peripheral Device Interface

BUS DRIVERS AND RECEIVERS

To maintain the transmission-line characteristics of the Unibus, special circuits are required to pass signals to and from the bus. The majority of bus signals (all except the five grant lines) are received, driven and terminated as shown in Figure 9-2.



R1, R2 = 180Ω 5% 1/4W
R3, R4 = 390Ω 5% 1/4W

Figure 9.2 Typical Unibus Line

Information is received from the bus using gates which have a high input impedance and proper logic thresholds. High input levels must be greater than 2.5 V with an input current less than 160 μa. Low level input must be less than 1.4 V with an input current greater than 0 μa.

Information transmitted on the bus must be driven with open collector drivers capable of sinking 50 ma with a collector voltage of less than .8 V. Output leakage current must be less than 25 μa.

In PDP-11 systems, the bus signals are terminated at both ends by resistor dividers provided on the M930 module. Physically, an M930 is located in the processor; another is located at the last unit on the bus. A bus signal sits at logical "0" (inactive, or negated state) at a voltage of 3.4 V. A bus line is at logical "1" (active, or asserted) when it is pulled to ground.

Drivers and receivers meeting these specifications are available on the M783, M784 and M785 modules as shown in Figures 9-3, 9-4 and 9-5.

M105 ADDRESS SELECTOR

The M105 Address Selector as shown in Figure 9-6 is used to provide gating signals for up to four device registers. The selector decodes the 18-bit bus address on A < 17:00 > as follows:

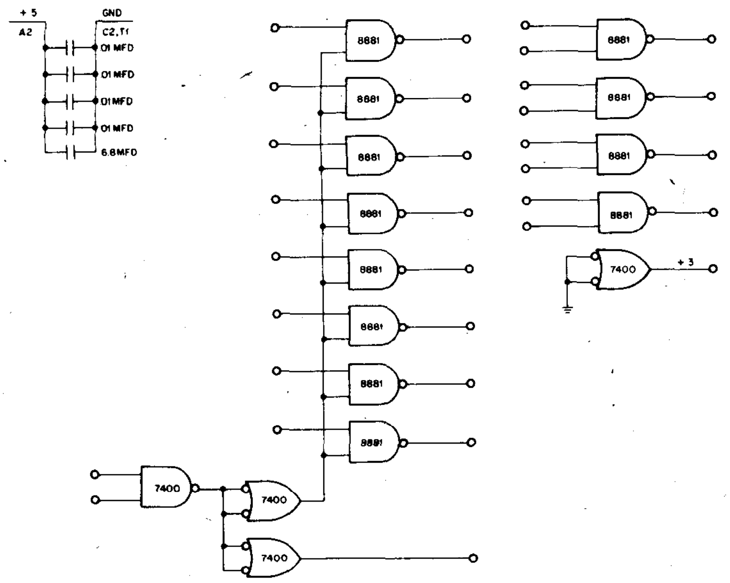


Figure 9.3 M783 Unibus Drivers

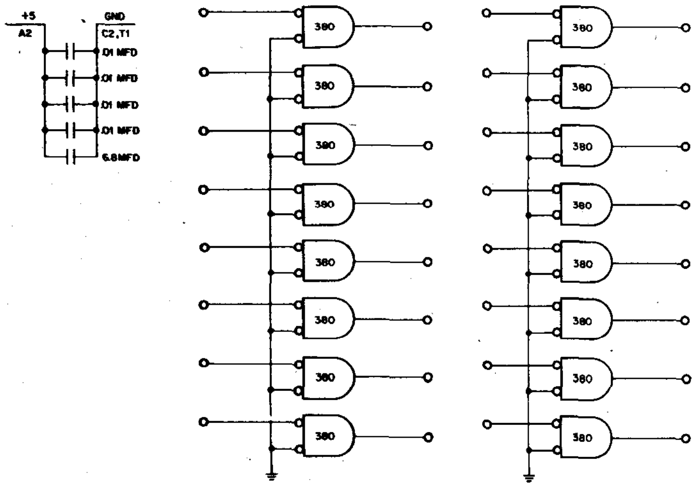


Figure 9.4 M784 Unibus Receivers

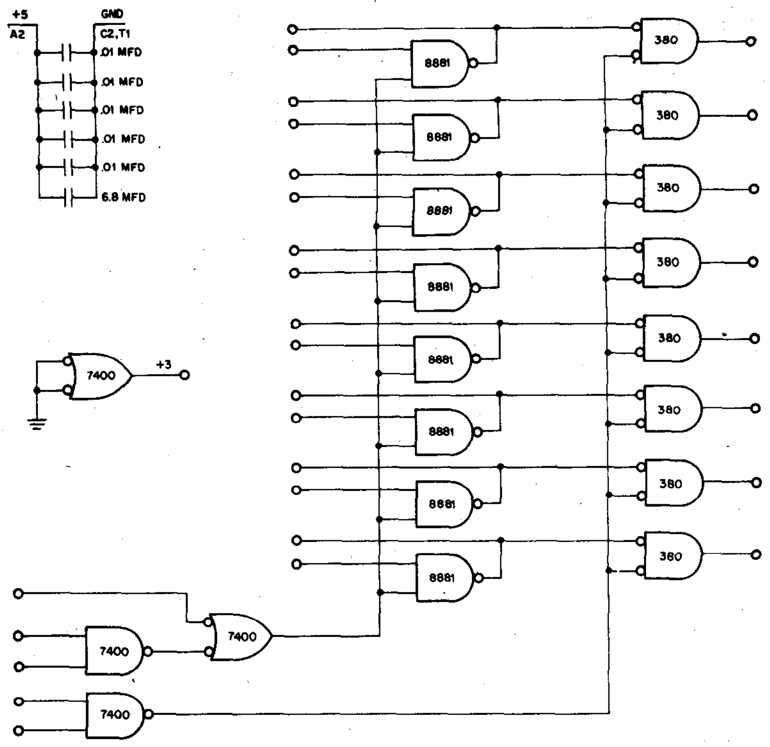


Figure 9.5 M785 Unibus Drivers and Receivers

A00 is used for byte control. A01 and A02 are decoded to provide one of four addresses. A < 12:03 > are determined by jumpers on the card. When the jumper is in, the selector will look for a 0 on that address line-A < 17:13 > must all be 1's—(this defines the external bank). Other bus inputs to the selector are C < 1:0 > and MSYN. The single bus output is SSYN. The user signals are SELECT 0, 2, 4, and 6 (corresponding to the decoding of A02 and A01, one of which is asserted when A < 17:13 > are all 1's and A < 12:03 > compare with the state of the jumpers. Other user signals are OUT HIGH (gate data into high byte), OUT LOW (gate data into low byte), and IN (gate data onto the bus). The equations for these last three signals are as follows:

$$\begin{aligned} \text{OUT HIGH} &= \text{DATO} + \text{DATOB*A00} \\ \text{OUT LOW} &= \text{DATO} + \text{DATOB*}\overline{\text{A00}} \\ \text{IN} &= \text{DATI} + \text{DATIP} \end{aligned}$$

where "+" means a logical or and "*" means a logical and.
Use of the M105, drivers, receivers and a flip-flop register is shown in Figure 9-7.

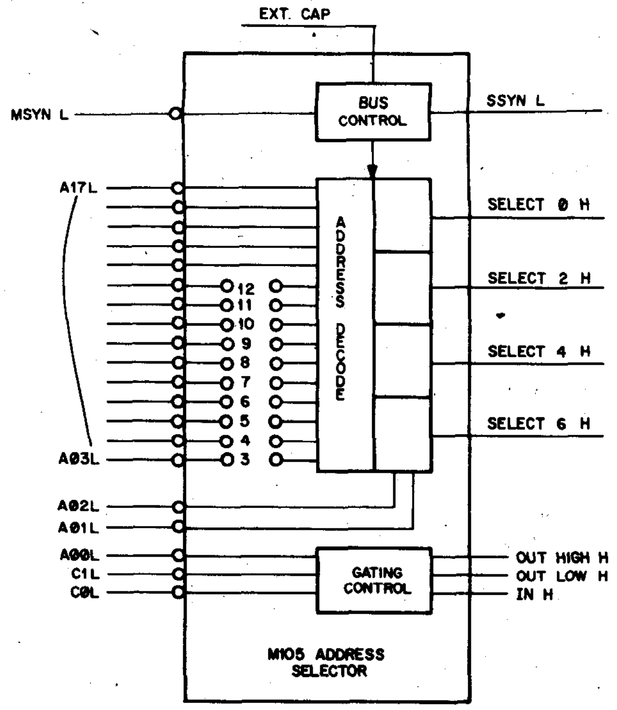


Figure 9.6 M105 Address Selector

M782 INTERRUPT CONTROL

The M782 Interrupt Control module contains the necessary logic circuits to allow a peripheral device to gain bus control and perform a program interrupt. The three circuits on this card are block diagrammed in Figure 9-8. Note that only signals relevant to the user's interface are shown; bus signals SSYN, BBSY and SACK have been omitted for clarity.

The Master Control circuit is used to gain bus control. When INT and INT ENB are asserted, a bus request is made on the request line to which BR is jumpered. When the processor issues the corresponding grant and other bus conditions are met, the MASTER signal is asserted, indicating that this device now has bus control. Note that this circuit also can be used to gain bus control on an NPR line for a device which requests the bus for direct memory access.

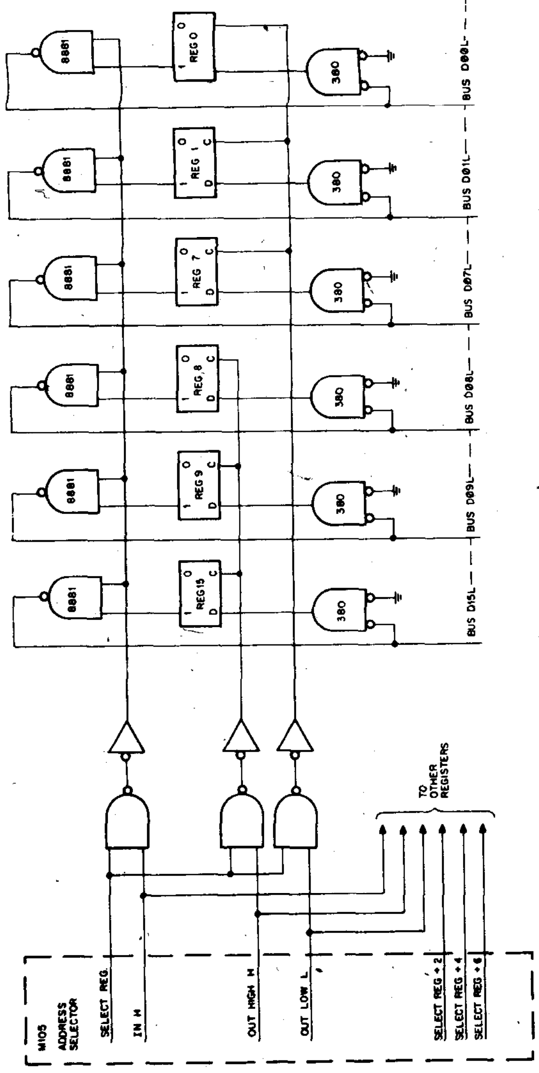


Figure 9.7 Typical Peripheral Device Register

In addition to two Master Control circuits, a third logic network provides the necessary signals and gating to perform the INTR bus operation. When either of the START INTR signals is asserted, the INTR bus signal is asserted along with a vector address on D < 07:02 >. Bits 07:03 are determined by jumpers on the card. A jumper "in" forces a 0 in that bit. Bit 2 is controlled by Vector Bit 2. When the processor responds to the INTR signal by asserting SSYN, the INTR DONE signal is asserted. This line is used to clear the condition which asserted INTR START.

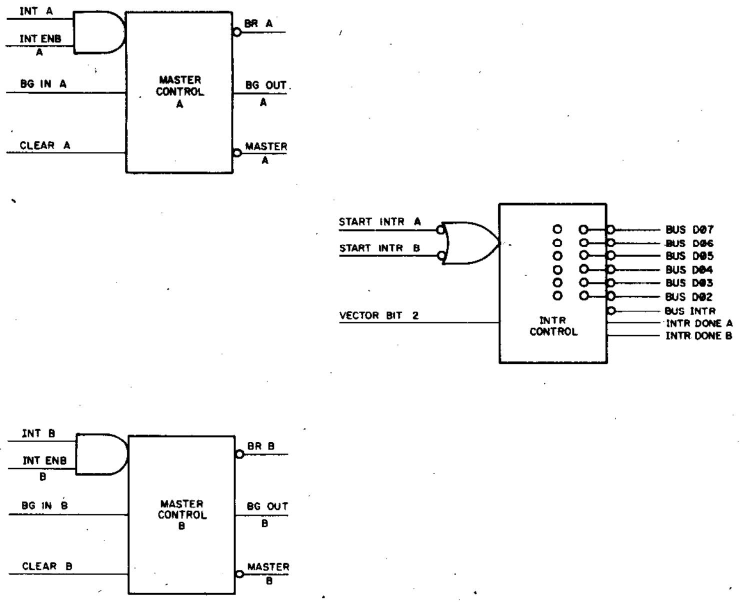


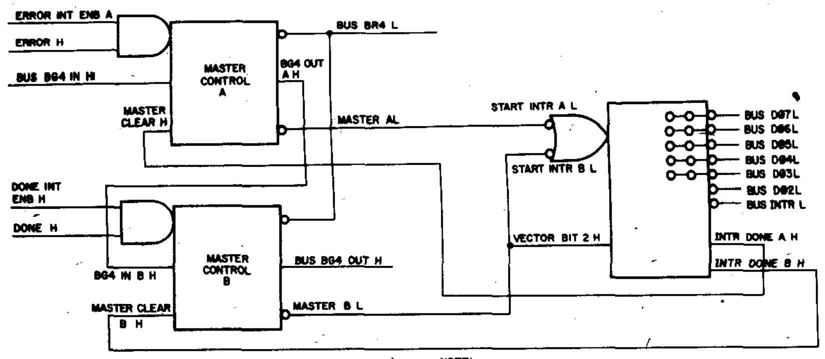
Figure 9.8 M782 Interrupt Control

Figure 9-9 shows a possible interconnection of the M782 to provide independent interrupts for two possible conditions in a device: ERROR and DONE. The ERROR and DONE signals shown in Figure 9-9 are signals from bits 15 and 7 in a device's CSR. Likewise ERROR INT ENB and DONE INT ENB are derived from the CSR. Both interrupts in this example are tied to the BR4 level; the corresponding grant line BG4 enters the ERROR Master Control and is passed on to the DONE Master Control. Thus, ERROR has a slightly higher priority interrupt level than DONE.

Both MASTER signals are tied to the INTR control. Thus, whenever either ERROR or DONE gains bus control, an INTR operation is initiated. Note that Vector Bit 2 is a 1 or 0 as a function of which master control is interrupting. Also, INTR DONE is tied to MASTER CLEAR to clear the master condition.

DEVICE CONTROL LOGIC

The type of control logic for a peripheral depends on the nature of the device. Digital offers a wide line of general-purpose logic modules for implementing control logic. These modules are described in detail in another Digital publication: The Logic Handbook.



NOTE:
1. BUS REQUEST IS MADE ON LEVEL 4
2. "ERROR" INTERRUPTS TO 104
3. "DONE" INTERRUPTS TO 100
4. "ERROR" HAS HIGHER PRIORITY THAN "DONE" BECAUSE "ERROR" RECEIVES BG4 FIRST

Figure 9.9 Typical Interconnection of M782 Interrupt Control

CHAPTER 10
CONFIGURATION AND INSTALLATION PLANNING

MODULAR CONSTRUCTION

Physically, the PDP-11 is composed of a number of System Units. Each System Unit is composed of three 8-slot connector blocks mounted end-to-end as shown in Figure 10-1. The Unibus connects to the System Unit at the lower left and at the upper left. Power also connects to the unit in the leftmost black. A System Unit is connected to other System Units only via the Unibus.

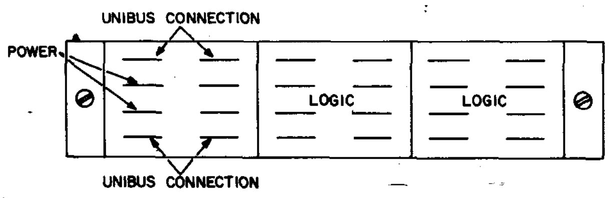


Figure 10.1 System Unit

The remainder of the System Unit contains logic for the processor, memory or an I/O device interface. This logic is composed of single height, double height, or quad height modules which are 8.5 " deep.

The use of System Units allows the PDP-11 to be optimally packaged for each individual application. Up to six System Units can be mounted into a single mounting box. For a basic PDP-11/20 system, the processor/console would fill 2½ System Unit spaces and 4096 words of core memory would fill one System Unit space. This leaves 2½ spaces for user-designated options. This would allow the user to add 8,192 words of additional core memory, a Teletype control, and a High-Speed Paper Tape Control, or 4,096 words of core memory and six Teletype interfaces. Larger systems will require a BA11-EC or BA11-ES Extension Mounting Box which contains space for six additional System Units.

The use of System Units also facilitates expansion of systems in the field and service. To add an additional option to a PDP-11 system, the proper System Unit is mounted in the Basic or Extension Mounting Box and the Unibus is extended. Servicing of the PDP-11 can be done by swapping modules or by swapping System Units.

MOUNTING BOXES AND CABINETS

The PDP-11 is available as either a tabletop or rack-mounted configuration. The rack-mounted configuration may be installed in a DEC cabinet or mounted in a customer cabinet. The PDP-11 mounts in an EIA standard 19-inch cabinet. The rack-mounted PDP-11 has tilt-slides as standard mounting hardware.

The following mounting units and cabinets are available for PDP-11 systems.

PDP-11 TABLETOP BOX AND POWER SUPPLY FOR 11/20, 11/10 SYSTEMS (BA11-CC AND H720)—This cover and box may be specified with a basic 11/20 and 11/10 system and includes:

1. H720 Power Supply
2. 15′ of power cord with ground wire

→ For 115 V standard, 3-prong, U-ground, 15-ampere connectors
→ For 230 V pigtail leads on one end
3. Cooling Fans
4. Filter
5. Programmers Console with 11/20 or Turn-Key Console with 11/10

Approximate Size—11" high, 20" wide, 24" deep. Figure 10-2 shows the layout of this unit.

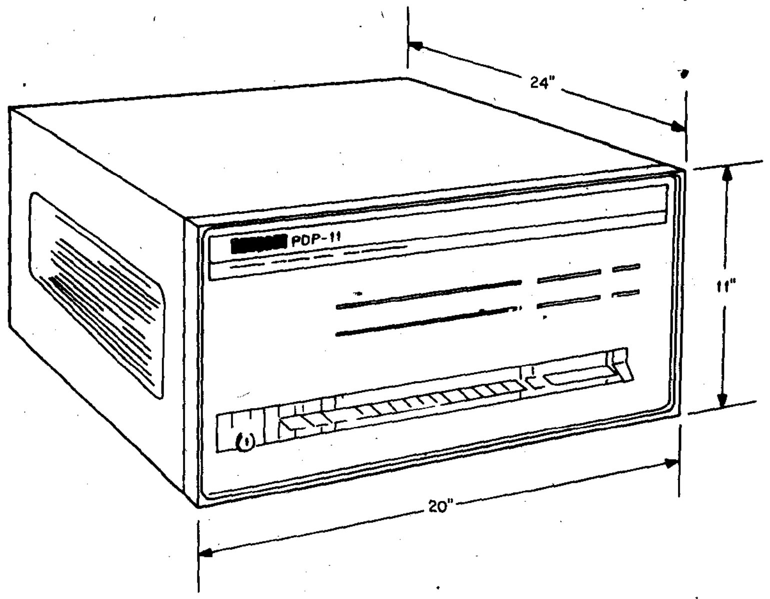


Figure 10.2 Table Top PDP-11 Dimensions

Approximate Weight—100 lbs. (including CP, console and 4K core)

Power—120 V ± 10%, 47-63 Hz	6 amps.	single phase
(BA11-CC and H720-A)		
230 V ± 10%, 47-63 Hz	3 amps.	single phase
(BA11-CC and H720-B)		

PDP-11 BASIC MOUNTING BOX AND POWER SUPPLY (BA11-CS AND H720) —This basic mounting box may be specified with a basic 11/20 or a 11/10 system and includes:

1. Tilt and Lock Chasis Slides
2. H720 Power Supply
3. 15' of power cord with ground wire
→ For 115 V standard, 3-prong, U-ground, 15-ampere connector
→ For 230 V pigtail leads on one end
4. Cooling Fans
5. Filter
6. Programmers Console with 11/20 or Turn-Key Console with 11/10

Approximate Size—10½" high, 19" wide, 23" deep. Figures 10-3, 10-4 and 10-5 show the layout of this unit and give slide dimensions.

Approximate Weight—90 lbs. (including CP, console and 4K core)

Power—120 V ± 10%, 47-63 Hz 6 amps. single phase
(BA11-C5 and H720-A)
230 V ± 10%, 47-63 Hz 3 amps. single phase
(BA11-C5 and H720-B)

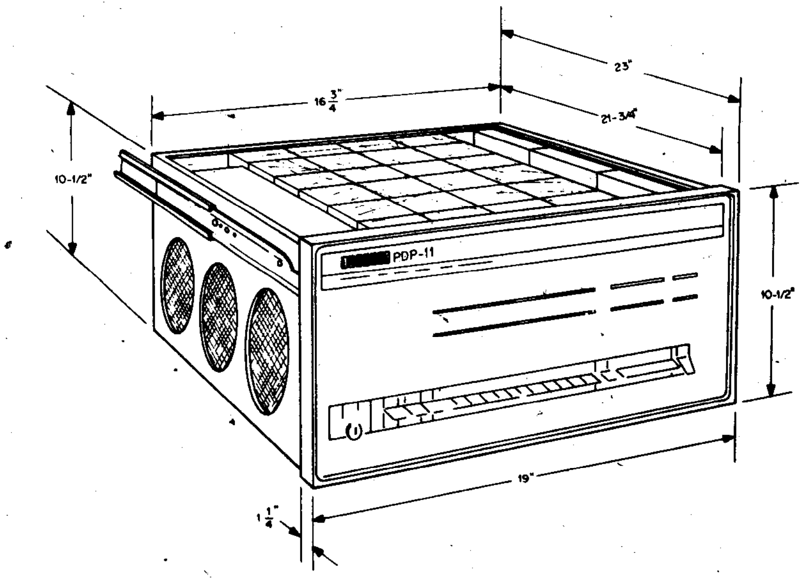


Figure 10.3 Rack Mountable PDP-11 Dimensions

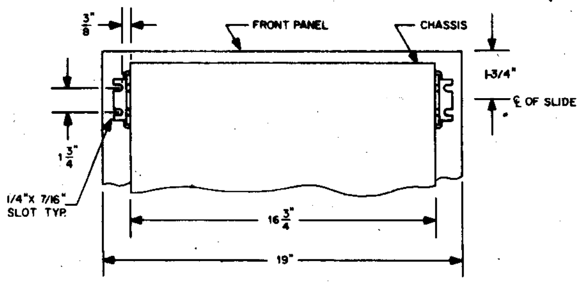


Figure 10.4 Rear View of Mounting Hardware

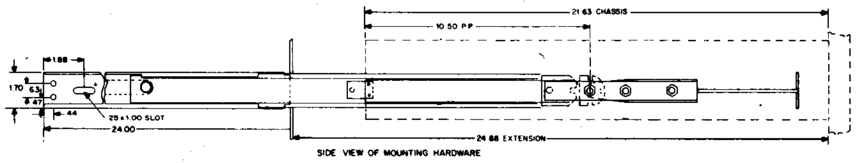


Figure 10.5 Side View of Mounting Hardware

PDP-11 TABLETOP EXTENSION MOUNTING BOX (BA11-EC)—The tabletop Extension Box is supplied, when ordered, for mounting of up to 6 additional System Units which can not be contained in the Basic Mounting Box. This unit is supplied with:

1. 15′ of power cord with ground wire
   → For 115 V standard, 3-prong, U-ground, 15-ampere connector
   → For 230 V pigtail leads on one end
2. Cooling Fans
3. Filter
4. Front Panel
5. Unibus Cable from Basic Mounting Box, 8′6″ long

Approximate Size—11″ high, 20″ wide, 24″ deep

Power—120 V ± 10%, 47-63 Hz   6 amps.   single phase
(when H720-A is added)
230 V ±10%, 47-63 Hz   3 amps.   single phase
(when H720-B is added)

PDP-11 EXTENSION MOUNTING BOX (BA11-ES)—The Extension Box is supplied, when ordered, for mounting of up to 6 additional System Units which can not be contained in the Basic Mounting Box. This unit contains:

1. Tilt and Lock chassis slides
2. 15′ of power cord with ground wire
   → For 115 V standard, 3-prong, U-ground, 15-ampere connector
   → For 230 V pigtail leads on one end
3. Cooling Fans
4. Filter
5. Front Panel
6. Bus Cable from Basic Box, 8′ 6″ long

Approximate size—10½″ high, 19″ wide, 23″ deep

Power—120 V ± 10%, 47-63 Hz   6 amps.   single phase
(when H720-A is added)
230 V ±10%, 47-63 Hz   3 amps.   single phase
(when H720-B is added)

PDP-11 FREESTANDING BASE CABINET (H960-CA)—This optional cabinet cabinet can be used to mount the BA11-CS Basic Mounting Box and a BA11-ES Extension Mounting Box supplied with Tilt and Lock chassis slides in addition to other PDP-11 equipment.

Panel capacity is six 10½″ high mounting spaces, each of which is covered with black plastic panels if equipment is not mounted—(5 panels, maximum, supplied).

Items supplied with the cabinet include:

1. H950-A Frame
2. H952-E Coasters
3. H-952-F Levelers
4. H-952-C Fan Assembly (in top of cabinet)
5. H-950-S Filter
6. PDP-11 Logo
7. H-950-B Rear Door
8. 10½″ Plastic Bezels, maximum of 5 supplied
9. Two H952-A End Panels

10. H950-D Mounting Panel Doors
11. H952-B Stabilizer Feet
12. #7406782 Kick Plate
13. #7005909 Power Distribution Panel (ac and dc, mounted on upper left side)

Approximate Size—22" wide, 39" deep (including stabilizer feet), 71½" high

Approximate Weight—150 lbs. (without computer)

Voltage—115 V 60 Hz (for fans)
230 V 50 Hz (for fans)

PDP-11 POWER SUPPLY SUBSYSTEM H720—This Power supply is used in the Basic and Extension Mounting boxes and supplies power to all devices mounted in one of these boxes. It is included in basic PDP-11 systems, but must be ordered separately with a BA11ES or BA11EC Extension Mounting Box.

Approximate Size—16½" wide, 8" high, 6" deep

Approximate Weight—25 lbs.

Voltages—(specify input voltage)

IN	108V ±10%, 47-63 Hz	6 amps	(H720A)
	120V ±10%, 47-63 Hz	6 amps	(H720A)
	216V ±10%, 47-63 Hz	3 amps	(H720B)
	228V ±10%, 47-63 Hz	3 amps	(H720B)
	240V ±10%, 47-63 Hz	3 amps	(H720B)
OUT	+5V ±5%	12 amps	
	−15V ±5%	10 amps	
	+8RMS (unregulated)	1.5 amps	
	−22V (unregulated)	1.0 amps	

FREESTANDING PROGRAMMER'S TABLE (H952-HA)—This freestanding table fits directly below the programmer's console in the Freestanding Base Cabinet and extends into the cabinet approximately 1". The surface plate is supported by its own adjustable height legs.

Approximate Size—20" extension from cabinet, 19" wide, 27" above floor

SYSTEM UNITS AND CABLES

The following items are available for mounting standard and special peripheral device logic into a PDP-11 system.

PERIPHERAL MOUNTING UNIT (DD11-A)—The DD11 is a prewired System Unit which allows standard small peripheral interfaces to be mounted in a PDP-11 system. It accepts standard small peripheral interfaces (up to 4) such as the KL11 Teletype Control or the controller portion (PC11-M) of the High Speed Reader/Punch. For mounting, it requires one-sixth (1/6) of a BA11 Mounting Box.

BLANK SYSTEM UNIT (BB11)—The BB11 consists of three 288-pin connector blocks connected end-to-end. This unit is unwired except for Unibus and power connections and allows customer-built interfaces to be integrated easily into a PDP-11 system. For mounting it requires one-sixth (1/6) of a BA11 Mounting Box.

UNIBUS MODULE (M920)—The M920 is a double module which connects the Unibus from one System Unit to the next within a Mounting Box. The printed circuit cards are separated by 1" for this purpose. A single M920 will carry all 56 Unibus signals and 14 grounds.

UNIBUS CABLE (BC11A)—The BC11A is a 120-conductor flexprint cable used to connect System Units in different mounting boxes or a peripheral device which is removed from the mounting boxes.

The 120 signals consist of the 56 Unibus lines plus 64 grounds. Signals and grounds alternate to minimize cross talk.

Type	Length
BC11A-2	2'
BC11A-5	5'
BC11A-8A	8'6"
BC11A-10	10'
BC11A-15	15'
BC11A-25	25'

CABLE REQUIREMENTS

When an Extension Mounting Box is used, an external cable, the BC11A, is the only signal connection between mounting boxes. This external bus cable may also be used to connect other peripherals to the PDP-11. The maximum combined, internal and external, bus cable length is 50'.

PDP-11/20 POWER REQUIREMENTS

Input Voltage and Current—105-125 Vac, 6 amperes, 210-260 Vac 3 amperes, (single phase)

Line Frequency—47-63 Hz

Power Dissipation—400 watts

A standard 15-foot, 3-prong, U-ground, 15-ampere, line cord is provided on the rear of the PDP-11 for connection to the power source on 120 Vac models. On 230 Vac models, a 15-foot, 3-conductor cable with pigtails is provided.

TELETYPE REQUIREMENTS

The standard Teletype requires a floor space approximately 22½ inches wide by 18½ inches deep. The Teletype cable length restricts its location to within 8 feet of the side of the computer.

Input Voltage—115 Vac ±10%, 60 Hz ±0.45 Hz, 230 Vac ±10%, 50 Hz ±0.75 Hz

Line Current Drain—2.0 amperes

Power Dissipation—150 watts

The Teletype plugs into the rear of the PDP-11 Basic Mounting Box and is turned ON and OFF by the power switch on the front panel of the PDP-11.

ENVIRONMENTAL REQUIREMENTS

The PDP-11 is designed to operate from +10 to +50°C and with a relative humidity of from 20 to 95% (without condensation).

INSTALLATION PROCEDURE

The PDP-11 is crated for shipment to the customer site to prevent damage. Installation is provided by DEC personnel at the customers site.

Computer customers may send personnel to instruction courses on computer operation, programming, and maintenance conducted regularly in Maynard, Massachusetts, Palo Alto, California, and Reading, England.

The PDP-11 has adopted a modular packaging approach to allow custom configuring of systems, easy expansion and easy servicing.

CHAPTER 11
PAPER TAPE SOFTWARE SYSTEM

PAPER TAPE SOFTWARE SYSTEM (PTS)

PTS is a compatible group of software packages designed to aid development of PDP-11 application programs. A brief description of each item with its major features is offered below with detailed programming information available in corresponding software user's manuals.

PTS FEATURES

- 4K Absolute Assembler
- Symbolic Program Editor for editing of paper tape which is string oriented
- On-Line Debugging Aid allowing rapid and accurate modification of assembled programs
- I/O Driver Routine allowing subroutine level communication with peripheral devices and double buffered input/output operation concurrent with running programs
- Floating Point Math Package using both reentrant and relocatable code
- General Utilities including loaders and dump routines

PAL-11A ASSEMBLER—This two- or three-pass assembler runs on a PDP-11 with 4K words of core memory and an ASR-33. It will also accommodate a high-speed reader/punch. Optional outputs include the absolute object code, an assembly listing containing each source statement, and an indication of any errors detected in the statement. A symbol table may be alphabetically listed.

ED11 EDITOR—The PDP-11 Editor (ED11) allows the user to type identified portions of source program on the teleprinter and to make corrections or additions. This is accomplished by typing simple commands that cause the Editor to read, print, punch out on paper tape, search, delete and/or add to the text of the program.

Use of the ED11 presupposes no special knowledge or technical skill beyond that of the operation of explicitly defined one-character commands. The commands are grouped according to function: input, positioning of the current-character location pointer, output, search (which is done by character string), insert, delete, and exchange of text portions.

ED11 uses 2,000 words of core and requires an ASR-33 unit which includes a printer, keyboard, paper tape reader and paper tape punch. Alternatively, a KSR-33 may be used in conjunction with the high-speed paper tape reader and punch.

ODT-11 ON-LINE DEBUGGING TECHNIQUE—ODT-11 is a core resident program which allows the user to debug his binary programs at the console by running them in specific segments and checking for expected results at various points. If modification of the program is needed, the user can alter the contents of the appropriate location by "opening" it and typing in new data.

Two versions of ODT are available, one being a subset of the other. The larger system uses 750 words of core and utilizes an ASR-33, or a KSR-33 and a high-speed paper tape punch and reader. The smaller version uses the same peripherals and 500 words of core. Up to eight breakpoints can be set using the larger version of ODT, while one breakpoint is allowed in the smaller version.

Debugging operations alternate between commands to ODT and the running of the program to be debugged. Breakpoints are set in the user's program by ODT commands, and a command to run starts execution of the program. When a breakpoint is encountered, the program run is suspended, and the progress of its execution can be monitored and altered. This is accomplished by using commands to open memory locations of interest, as well as special registers.

An operator may examine and change the operating priority of both ODT and the user's program, the mask and address range for searches, results of logical and arithmetic operations, the SP and PC, and the general registers. Other commands will search for values of specified bits of a word, or for references to an address within an address range, calculate 16-bit and 8-bit offsets to an address and restart the running of the user's program at any address.

IOX Input/Output Utility Peripheral Driver—IOX is a set of service routines allowing single or double buffered I/O processing on an ASR-33 and/or a high-speed paper tape reader and punch. This routine allows the user to make simple assembly language calls specifying devices and data forms to accomplish interrupt-controlled data transfer concurrent with execution of the running program. Multiple devices can be run simultaneously.

IOX frees the user from the details of dealing directly with the device and allows development of programs which may be run under the direction of a monitor with minimum modification.

IOX also provides some degree of real-time control by allowing user programs to be executed at priority levels at the completion of some device action or data transfer.

MATH PACKAGE—A number of commonly used subroutines are available to simplify programming. These routines are reentrant and relocatable to provide maximum flexibility. Arguments are treated as floating point numbers with a signed 31-bit fraction and a signed 15-bit exponent. Subroutines supplied include:

ADD
MULtiply
SUBtract
DIVide
SIN
COS
ATAN
FIX—FLOAT
FLOAT—FIX
NORmalize
(Integer MULtiply and DIVide are also supplied)

LOADERS—Two loaders are used:

- A Bootstrap loader loads the ABSolute loader and jumps to it.
- ABSolute loader loads PAL-11A output, checks for checksum errors and jumps to a user program or halts when done.

CORE DUMP ROUTINES—Routines are provided which dump specified ranges of core locations on paper tape in absolute format or on the teleprinter in octal.

CHAPTER 12
THE OPERATOR'S CONSOLE

The PDP-11 Operator's Console has been configured to achieve convenient control of the system. Through switches and keys on the console, programs or information can be manually inserted or modified. Also indicator lamps on the console face display the status of the machine, the contents of the Bus Address Register and the data at the output of the data paths.

The console is shown in Figure 12-1.

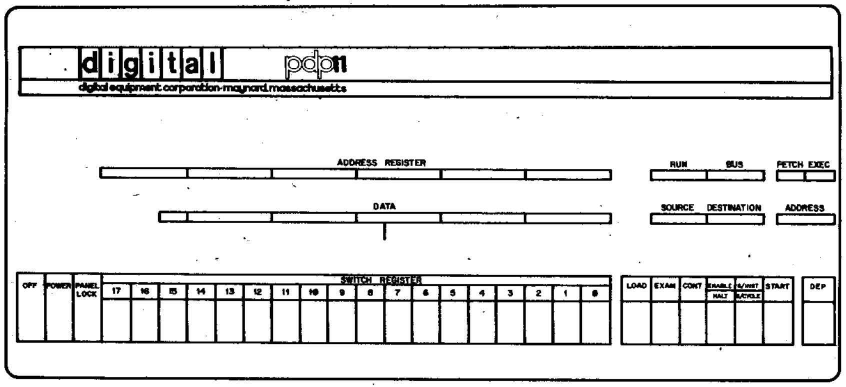


Figure 12-1

CONSOLE ELEMENTS

The console has the following indicators and switches:

1. A bank of 8 indicators, indicating the following conditions or operations: Fetch, Execute, Bus, Run, Source, Destination and Address (2 bits).
2. An 18-bit Address Register Display
3. A 16-bit Data Register Display
4. An 18-bit Switch Register
5. Control Switches:

    a. LOAD ADDR (Load Address)
    b. EXAM (Examine)
    c. CONT (Continue)
    d. ENABLE/HALT
    e. S/INST—S/CYCLE (Single Instruction/Single Cycle)
    f. START
    g. DEP (Deposit)

INDICATOR LIGHTS—The indicators signify specific machine functions, operations, or states. Each is defined below.

1. Fetch—indicates that the central processor is in the state of fetching an instruction.
2. Execute—indicates that the central processor is in the state of executing an instruction.

3. Bus—indicates that a peripheral is controlling the bus. It is lit when BBSY (Bus Busy) is asserted, unless the processor (which includes the console) is asserting BBSY.
4. Run—indicates that the processor is running. It monitors the control flip-flop for the internal clock.
5. Source—indicates that the central processor is obtaining source data except from an internal register.
6. Destination—indicates that the central processor is obtaining destination data (except from an internal register).
7. Address—identifies the source or destination address cycle of the central processor, using two lights that are decoded zero, one, two, or three. When references are made via the Unibus to the addresses, the lights tell the machine's source or destination cycle. For an internal register reference, there is a "zeroth" addressing operation.

REGISTER DISPLAYS—The Operator's Console has an 18-bit Address Register Display and a 16-bit Data Register Display. The Address Register Display is tied directly to the output of an 18-bit flip-flop register called the Bus Address Register. This register displays the address of data examined or deposited.

The 16-bit data register is divided on the face of the console by a line into two 8-bit bytes. This register is tied to the output of the processor data paths and will reflect the output of the processor adder.

SWITCH REGISTER—The PDP-11/10 and PDP-11/20 can reference 2^{16} bytes addresses. However, the Unibus has expansion capability for 2^{18} byte addresses. In order that the console can access the entire 18-bit address scheme, the switch register is 18 bits wide. These bits are assigned as 0 through 17. The highest two are used only as addresses. A switch in the "up" position is considered to have a "1" value and in the "down" position to have a "0" value. The condition of the 18 switches can be loaded into the bus address register or any memory location by using the appropriate control switches which are described below.

CONTROL SWITCHES—The switches listed in item 5 of the "Console Elements" have these specific control functions:

1. LOAD ADDR—transfers the contents of the 18-bit switch register into the bus address register.
2. EXAM—displays the contents of the location specified by the bus address register.
3. DEP—deposits the contents of the low 16 bits of the switch register into the address then displayed in the address register. (This switch is actuated by raising it.)
4. ENABLE/HALT—allows or prevents running of programs. For a program to run, the switch must be in the ENABLE position (up). Placing the switch in the HALT position (down) will halt the system.
5. START—starts executing a program when the ENABLE/HALT switch is in the ENABLE position. When the START switch is depressed, it asserts a system initialization signal; the system actually starts when the switch is released. The processor will start executing at the address which was last loaded by the LOAD ADDR key.
6. CONT—allows the machine to continue without initialization from whatever state it was in when halted.
7. S/INST-S/CYCLE—determines whether a single instruction or a single bus cycle is performed when the CONT switch is depressed while the machine is in the halt mode.

When the system is running a program, the LOAD ADDR, EXAM, and DEPOSIT functions are disabled to prevent disrupting the program. When the machine is to be halted, the ENABLE/HALT switch is thrown to the halt position. The machine will halt either at the end of the current instruction, or at the end of the current bus cycle, depending upon the position of the S/INST-S/CYCLE switch.

OPERATING THE CONTROL SWITCHES

When the PDP-11 has been halted, it is possible to examine and update bus locations. To examine a specific location, the operator sets the switches of the switch register to correspond to the location's address. The operator then presses LOAD ADDR, which will transfer the contents of the switch register into the bus address register. The location of the address to be examined is then displayed in the address register display. The operator then depresses EXAM. The data in that location will appear in the data register display.

If the operator then depresses EXAM again, the bus address register will be incremented by 2 to the next word address and the new location will be examined. In the PDP-11, the bus address register will always be pointing to the data currently displayed in the data register. The incrementation occurs when the EXAM switch is depressed, and then the location is examined.

The examine function has been designed so that if LOAD ADDR and then EXAM are depressed, the address register will not be incremented. In this case, the location reflected in the address register display is examined directly. However, on the second (and successive) depressings of EXAM, the bus address register is incremented. This will continue for successive depressings as long as another control switch is not depressed.

If the operator finds an incorrect entry in the data register, he can enter new data there by putting it in the switch register and raising the DEP key. The address register will not increment when this data is deposited. Therefore, when the operator presses the EXAM key, he can examine the data he just deposited. However, when he presses EXAM again, the system will increment.

If the operator attempts to examine data from, or deposit data into, a nonexistent memory location, the "time out" feature will cause an error flag. The data register will then reflect location 4, the trap location, for references to nonexistent locations. To verify this condition, the operator should try to deposit some number other than four in that location; if four is still indicated, this would indicate that either nothing is assigned to that location, or that whatever is assigned to that location is not working properly.

When doing consecutive examines or consecutive deposits, the address will increment by 2, to successive word locations. However, if the programmer is examining the fast registers (the "scratch pad" memory), the system only increments by 1. The reason for this is that once the switch register is set properly, the programmer can then use the four least significant bits of the switch register in examining fast memory registers from the front panel.

To start a PDP-11 program, the programmer loads the starting address of the program in the switch register, depresses LOAD ADDR, and after ensuring that the ENABLE/HALT switch is in the ENABLE position, depresses START. The program will start to run as soon as the START switch is released.

The Run indicator lamp is driven off the flip-flop that controls the clock. Normally, when the system is running, not only will this light be on, but the other lights (Fetch, Execute, Source, Destination, the Address lights, and the Address and Data registers) will be flickering. If the run light is on, and none of the other indicators are flickering, the system could be executing a "wait" instruction which waits for an interrupt.

While in the halt mode, if the operator wishes to do a single instruction, he places the S/INST-S/CYCLE switch in the S/INST position and depresses CONT. When CONT is depressed, the console momentarily passes control to the processor, allowing the machine to execute one instruction before regaining control. Each time the CONT switch is depressed, the machine will execute one instruction.

Similarly, if the operator wishes to have the machine perform a single bus cycle, he places the S/INST-S/CYCLE switch in the S/CYCLE position and presses CONT. The machine will then perform one complete bus cycle and halt. The operator cannot do an examine or deposit function at the end of a single bus cycle unless the cycle ends coincidental with the end of an instruction. This prevents altering machine flow. Only when the machine is at the end of an instruction and in the halt mode can the examine or deposit functions operate.

To start the machine running its program again, the operator places the ENABLE/HALT switch in the ENABLE position, and depresses the CONT switch.

APPENDIX A—PDP-11 INSTRUCTION REPERTOIRE

Mnemonic	Instruction Operation	OP Code	Condition Codes ZNCV	Timing
DOUBLE OPERAND GROUP: OPR scr, dst				
MOV(B)	MOVe (Byte) (src) → (dst)	·1SSDD	✓✓—0	2.3
CMP(B)	CoMPare (Byte) (src) — (dst)	·2SSDD	✓✓✓✓	2.3*
BIT(B)	BIt Test (Byte) (src) ∧ (dst)	·3SSDD	✓✓—0	2.9*
BIC(B)	BIt Clear (Byte) ~ (src) ∧ (dst) → (dst)	·4SSDD	✓✓—0	2.9
BIS(B)	BIt Set (Byte) (src) ∨ (dst) → (dst)	·5SSDD	✓✓—0	2.3
ADD	ADD (src) + (dst) → (dst)	06SSDD	✓✓✓✓	2.3
SUB	SUBtract (dst) — (src) → (dst)	16SSDD	✓✓✓✓	2.3
CONDITIONAL BRANCHES: Bxx loc				
BR	BRanch (unconditionally) loc → (PC)	0004XX	——	2.6—
BNE	Branch if Not Equal (Zero) loc → (PC) if Z = 0	0010XX	——	2.6—
BEQ	Branch if Equal (Zero) loc → (PC) if Z = 1	0014XX	——	2.6—
BGE	Branch if Greater or Equal (Zero) loc → (PC) if N ∀ V = 0)	0020XX	——	2.6—
BLT	Branch if Less Than (Zero) loc → (PC) if N ∀ V = 1	0024XX	——	2.6—
BGT	Branch if Greater Than (Zero) loc → (PC) if Z v (N ∀ V = 0)	0030XX	——	2.6—
BLE	Branch if Less Than or Equal (Zero) loc → (PC) if Z v (N ∀ V) = 1	0034XX	——	2.6—
BPL	Branch if PLus loc → (PC) if N = 0	1000XX	——	2.6—
BMI	Branch if MInus loc → (PC) if N = 1	1004XX	——	2.6—
BHI	Branch if HIgher loc → (PC) if C v Z = 0	1010XX	——	2.6—
BLOS	Branch if LOwer or Same loc → (PC) if C v Z = 1	1014XX	——	2.6—
BVC	Branch if oVerflow Clear loc → (PC) if V = 0	1020XX	——	2.6—
BVS	Branch if oVerflow Set loc → (PC) if V = 1	1024XX	——	2.6—
BCC (or BHIS)	Branch if Carry Clear loc → (PC) if C = 0	1030XX	——	2.6—
BCS (or BLO)	Branch if Carry Set loc → (PC) if C = 1	1034XX	——	2.6—

SUBROUTINE CALL: JSR reg, dst

JSR	Jump to SubRoutine (dst)→ (tmp), (reg) ↓ (PC) → (reg), (tmp) → (PC)	004RDD	——	4.2

SUBROUTINE RETURN: RTS reg

RTS	ReTurn from Subroutine (reg) → PC, ↑(reg)	00020R	——	3.5

SINGLE OPERAND GROUP: OPR dst

CLR(B)	CLeaR (Byte) 0 → (dst)	·050DD	1000	2.3
COM(B)	COMplement (Byte) ~ (dst) → (dst)	·051DD	√√00	2.3
INC(B)	INCrement (Byte) (dst) + 1 → (dst)	·052DD	√√—√	2.3
DEC(B)	DECrement (Byte) (dst) — 1 → (dst)	·053DD	√√—√	2.3
NEG(B)	NEGate (Byte) ~ (dst) + 1 → (dst)	·054DD	√√√√	2.3
ADC(B)	ADd Carry (Byte) (dst) + (C) → (dst)	·055DD	√√√√	2.3
SBC(B)	SuBtract Carry (Byte) (dst) — (C) → (dst)	·056DD	√√√√	2.3
TST(B)	TeST (Byte) 0 — (dst)	·057DD	√√00	2.3*
ROR(B)	ROtate Right (Byte) rotate right 1 place with C	·060DD	√√√√	2.3°
ROL(B)	ROtate Left (Byte) rotate left 1 place with C	·061DD	√√√√	2.3°
ASR(B)	Arithmetic Shift Right (Byte) shift right with sign extension	·062DD	√√√√	2.3°
ASL(B)	Arithmetic Shift Left (Byte) shift left with lo-order zero	·063DD	√√√√	2.3°
JMP	JuMP (dst) → (PC)	0001DD	——	1.2
SWAB	SWAp Bytes bytes of a word are exchanged	0003DD	√√00	2.3

CONDITION CODE OPERATORS: OPR 1.5

Condition Code Operators set or clear combinations of condition code bits. Selected bits are set if S = 1 and cleared otherwise. Condition code bits corresponding to bits set as marked in the word below are set or cleared.

CONDITION CODE OPERATORS:

15				5	4	3	2	1	0
0	0	0	2		4 S	N	Z	V	C

Thus SEC = 000261 sets the C bit and has no effect on the other condition code bits (CLC = 000241 clears the C Bit)

OPERATE GROUP: OPR

HALT	HALT processor stops; (RO) and the HALT address in lights	000000	——	1.8
WAIT	WAIT processor releases bus, waits for interrupt	000001	——	1.8

RTI	ReTurn from Interrupt ↑ (PC), ↑ (PS)	000002	✓✓✓✓	4.8
IOT	Input/Output Trap (PS) ↓, (PC) ↓, (20) → (PC), (22) → (PS)	000004	✓✓✓✓	8.9
RESET	RESET an INIT pulse is issued by the CP	000005	——	20 ms.
EMT	EMulator Trap (PS) ↓, (PC) ↓, (30) → (PC), (32) → (PS)	104000—104377	✓✓✓✓	8.9
TRAP	TRAP (PS) ↓, (PC) ↓, (34) → (PC), (36) → (PS)	104400—104777	✓✓✓✓	8.9

NOTATION:

1. for order codes
   word/byte bit, set for byte (+100000)
   SS—source field,
   DD—destination field
   XX—offset (8 bit)
2. for operations

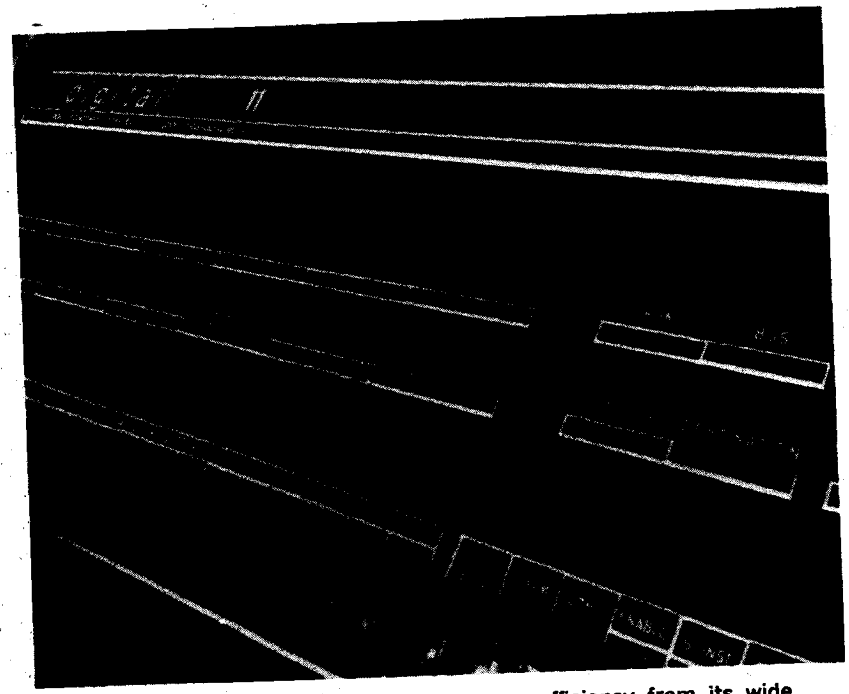
$\wedge$	and,
v	or,
~	not,
( )	contents of,
∀	XOR
↓	"is pushed onto the processor stack"
↑	"the contents of the top of the processor stack is popped and becomes"
→	"becomes"

3. for timing

*	0.4 μs less if not register mode
—	0.9 μs less if conditions for branch not met
°	1.2 μs more if addressing odd byte
′	(0.6 μs additional in addressing odd bytes otherwise)

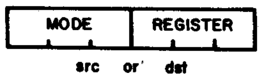
4. for condition codes

✓	set conditionally
—	not affected
0	cleared
1	set


The PDP-11 derives speed and memory efficiency from its wide range of addressing capabilities.

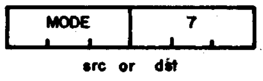
APPENDIX B—ADDRESSING SUMMARY

ADDRESSING MODES



GENERAL REGISTER ADDRESSING

Mode	Description	Symbolic	Timing (µs) src	Timing (µs) dst
0	register	R	00	00
1	register deferred	@ R or (R)	1.5	1.4
2	auto increment	(R) +	1.5	1.4
3	auto increment deferred	@ (R) +	2.7	2.6
4	auto decrement	— (R)	1.5	1.4
5	auto decrement deferred	@ — (R)	2.7	2.6
6	indexed	X (R)	2.7	2.6
7	indexed deferred	@ X (R) or @ (R)	3.9	3.8

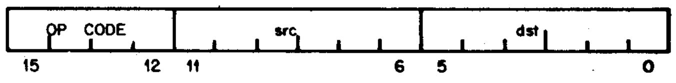


PC REGISTER ADDRESSING

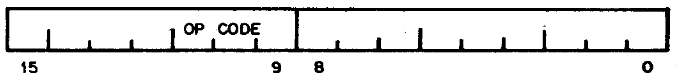
Mode	Description	Symbolic	Timing µs) src	Timing µs) dst
2	immediate	#n	1.5	1.4
3	absolute	@ #A	2.7	2.6
6	relative	A	2.7	2.6
7	relative deferred	@ A	3.9	3.8

INSTRUCTION FORMATS

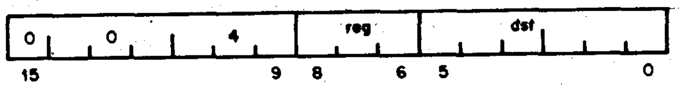
DOUBLE OPERAND GROUP: OPR src, dst



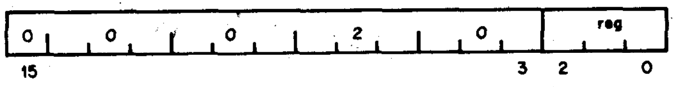
CONDITIONAL BRANCHES: Bxx loc (loc=(offset·2)+.+2)



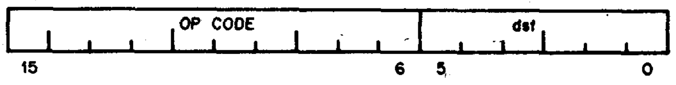
SUBROUTINE CALL: JSR reg, dst



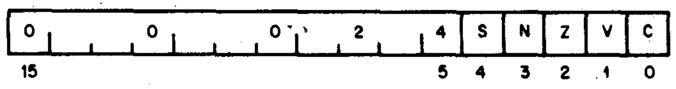
SUBROUTINE RETURN: RTS reg



SINGLE OPERAND GROUP: OPR dst



CONDITION CODE OPERATORS:



APPENDIX C—ADDRESS MAP

0	USER DEVICE INTERRUPT VECTOR
4	BUS ERROR, ILLEGAL INSTRUCTION, STACK OVERFLOW TRAP VECTOR
10	RESERVED INSTRUCTIONS TRAP VECTOR
14	CODE 000003 AND TRACE TRAP VECTOR
20	IOT INSTUCTION TRAP VECTOR
24	POWER FAIL INTERRUPT VECTOR
30	EMT INSTRUCTION TRAP VECTOR
34	TRAP INSTUCTION TRAP VECTOR
40	
44	SYSTEM SOFTWARE COMMUNICATION
50	
54	
60	TELEPRINTER INTERRUPT VECTOR
64	TELETYPE KEYBOARD AND LOW SPEED READER INTERRUPT VECTOR
70	HIGH SPEED PAPER TAPE PUNCH INTERRUPT VECTOR
74	HIGH SPEED PAPER TAPE READER INTERRUPT VECTOR
•	
•	
•	(additional interrupt vectors)
•	
•	
•	
400	
•	
•	
•	PROCESSOR STACK PROGRAM AND DATA RESIDENT SYSTEM SOFTWARE
•	
•	
•	(ABSOLUTE LOADER, BOOTSTRAP, I/O EXECUTIVE)
(end of implemented storage)	
160000	
•	
•	SMALL READ-ONLY STORAGE UNITS
•	
•	
•	
•	
•	OTHER PERIPHERAL DEVICE REGISTERS
•	
•	
•	
177550	HIGH SPEED READER AND PUNCH DEVICE STATUS AND BUFFER REGISTERS
•	
•	
•	

```
177560  TELETYPE KEYBOARD AND PUNCH DEVICE STATUS AND BUFFER
           REGISTER
   .
   .
   .

   .
   .
177576
177600 )
   .    > RESERVED FOR EXPANSION OF PROCESSOR REGISTERS
177677 )
177700 )
   .    > GENERAL REGISTERS   R0 — R7
   .    )
177776  CENTRAL PROCESSOR STATUS REGISTER (PS)
```

APPENDIX D—UNIBUS OPERATIONS

There are six bus operations: four to effect data transfers, one to transfer bus control, and one to effect a program interrupt. This appendix describes the signal interaction on the Unibus to perform these six operations.

DATA TRANSFERS

The four data transfers use the C lines coded as follows:

C1	C0	
0	0	DATI-DATa In
0	1	DATIP-DATa In, Pause
1	0	DATO-DATa Out
1	1	DATOB-DATa Out, Byte

DATI AND DATIP—These two bus operations transfer data from a slave whose address is specified by A $< 17{:}01 >$ into the master. Both transfers are made in words on D $< 15{:}00 >$. In destructive read-out devices, DATI commands a read-restore operation, while DATIP commands a read-pause operation and the setting of a pause flag. DATIPs are to be followed by a DATO or DATOB to effect a read-modify-write data exchange. In non destructive read-out devices, DATI and DATIP are treated identically. The sequence of operations is as follows:

1. Master puts address on A, 0 or 1 on C, and waits 150 nanoseconds. (75 nanoseconds for deskewing address + 75 nanoseconds for address decoding).
2. Master asserts MSYN.
3. Slave decodes address, sees 0 or 1 on C, and MSYN and begins read cycle (flip-flop register would simply gate flop outputs to bus).
4. Slave completes read cycle, outputs data to D lines, and asserts SSYN. If the slave is a destructive read-out device, it now restores data on a DATI: it sets a pause flag on a DATIP.

Figure D-1 shows the signals for a DATI operation.

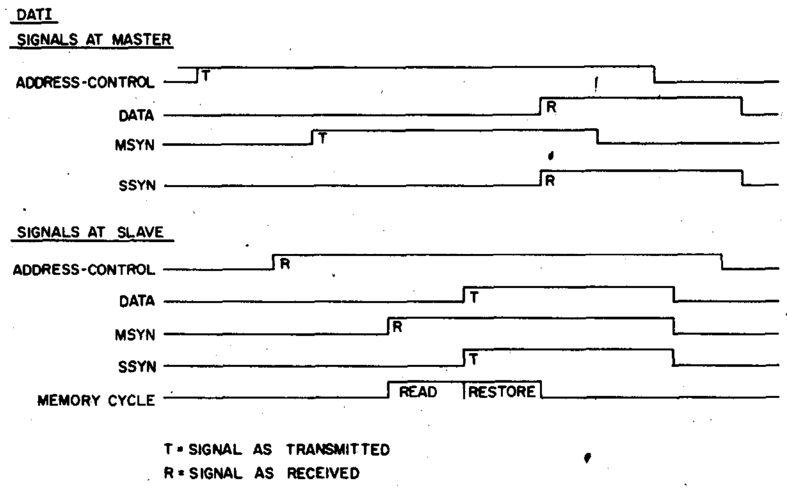


Figure D-1 DATI Operation

5. Master sees SSYN and waits 75 nanoseconds, minimum (data deskewing + internal gating deskewing).
6. Master strobes data, drops MSYN, and waits 75 nanoseconds (deskew address).
7. Master drops A and C and waits for SSYN to fall.
8. Slave sees MSYN fall and drops SSYN and D lines.
9. Master sees SSYN fall, signaling end of bus operation.

NOTES:
1. Step 1 of the next data transfer may begin at step 7 of the current DATI or DATIP.
2. Step 2 of the next data transfer may begin at step 9 of the current DATI or DATIP.

DATO AND DATOB—These two bus operations transfer data out of the master to the slave. DATO is used to transfer a word to the address specified by A < 17:01 >. The slave ignores A00 and the data appears on D < 15:00 >. DATOB is used to transfer a byte to the address specified by A < 17:00 >. A00 = 0 indicates the low byte and data appears on D < 07:00 >; A00 = 1 indicates high byte and data appears on D < 15:08 >. The sequence of operation is as follows:

1. Master puts address on A, data on D, 2 or 3 on C, and waits 150 nanoseconds (75 nanoseconds for deskewing address + 75 nanoseconds for address decoding).
2. Master asserts MSYN.
3. Slave decodes address, sees 2 or 3 on C and MSYN and strobes in word or byte. When slave has taken data, it asserts SSYN. If the slave is a destructive read-out device and its pause flag is set (by DATIP), slave begins write cycle; if not, slave must first do a read cycle to clear the memory cell and then a write.
4. Master sees SSYN and drops MSYN and waits 75 nanoseconds (deskew address).
5. Master drops A, D, and C, and waits for SSYN to fall.
6. Slave sees MSYN fall and drops SSYN.
7. Master sees SSYN fall, signaling end of bus operation.

Figure D-2 shows the signals for a DATO operation.

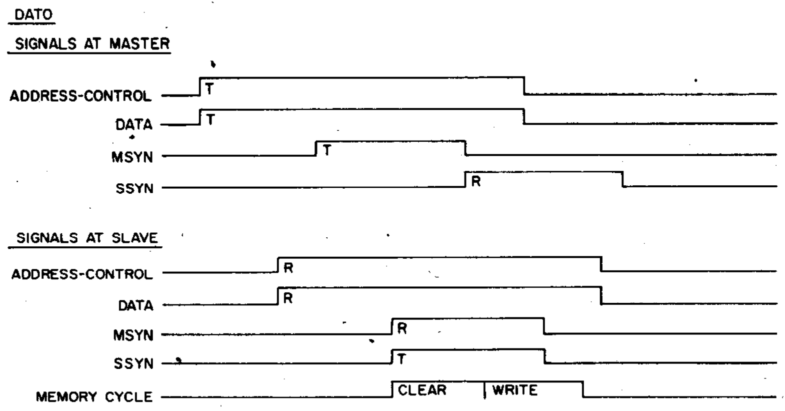


Figure D-2 DATO Operation

NOTES:

1. Step 1 of the next data transfer may begin at step 5 of the current DATO or DATOB.
2. Step 2 of the next data transfer may begin at step 7 of the current DATO or DATOB.

PTR-PRIORITY TRANSFER

This bus operation is used to pass control of the bus from one master to another. The steps which follow are performed simultaneously with the data transfers:

0. Current master device always has BBSY asserted.
1. Requesting device asserts its assigned BR line.
2. Processor sees BR asserted, determines which BR is highest, and asserts the corresponding BG line if the processor's current priority level allow that level of bus request.
3. Each device that receives the BG passes it on to the next device unless it itself is requesting.
4. The BG is propagated along the priority chain until it reaches the first requesting device. This device becomes selected as next bus master and does not allow the BG to pass to succeeding devices.
5. The selected device asserts SACK and drops its BR, and waits for BBSY, BG, and SSYN to drop.
6. The processor sees SACK and drops BG.
7. The device which is current master completes its data transfers, drops BBSY, and ceases to be bus master.
8. The selected device sees BG, BBSY, and SSYN drop, becomes bus master, asserts BBSY, drops SACK, and begins data transfers.
9. New master relinquishes bus control, either to the processor or to a requesting device, by dropping BBSY at the end of its last bus operation. This is termed a passive release of bus control.

NOTES:

1. NPR bus requests are handled as above.
2. Processor defers action on BR <7:4> until last bus cycle of an instruction execution or interrupt sequence, NPR is acted upon immediately.
3. Processor becomes bus master and asserts BBSY whenever it sees BBSY = 0 and no other device has been selected or is being selected as next bus master.
4. Processor will not execute step 2 if SACK is asserted. See note 2 under INTR.

Figure D-3 shows the signals for a PTR operation.

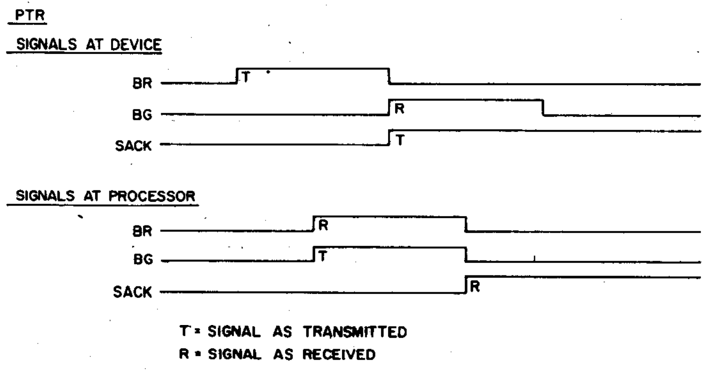


Figure D-3 PTR Operation

INTR—INTerRupt

This bus operation is initiated by a master immediately after receiving bus control to effect a program interrupt in the processor. It proceeds as follows:

0. Device has become bus master via PTR and BBSY is asserted.
1. Master puts interrupt vector address on D and asserts INTR.
2. Processor sees INTR and waits 75 nanoseconds (deskew data).
3. Processor strobes data and asserts SSYN.
4. Master sees SSYN, drops INTR, D, and BBSY. The master has now relinquised bus control directly to the processor. The INTR sequence is termed an active release of bus control.
5. Processor sees INTR drop and drops SSYN and enters interrupt sequence to update PC and PS.

NOTES:

1. Step 1 must be made simultaneously with step 8 of PTR; that is, SACK cannot be dropped until INTR is asserted.
2. When the processor sees SACK drop, it waits 75 nanoseconds (deskew). If, at that time, INTR = 1, the processor issues no BG's until the interrupt sequence is complete.

Figure D-4 shows the signals for the INTR operation.

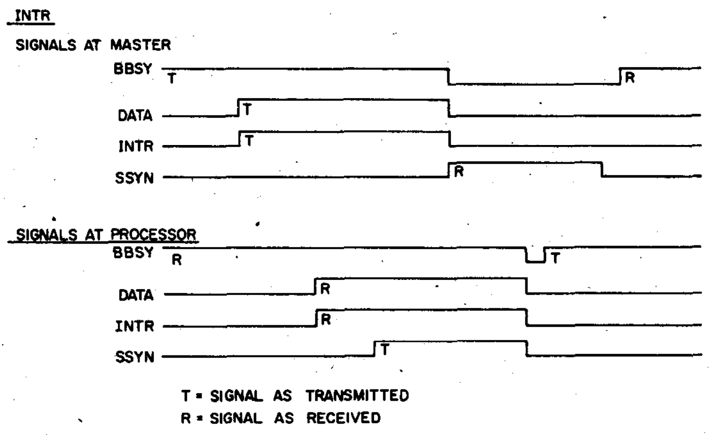


Figure D-4 INTR Operation

GENERAL NOTES ON THE BUS OPERATIONS

1. A master device doing a read-modify-write operation must keep bus control BBSY asserted for both bus transactions (both the DATIP and the DATO or DATOB). This is the one case where an NPR request will not be honored between bus transactions.
2. A device becomes master by the PTR operation. If the request for bus control was made on the NPR line, bus control must be released passively (by dropping BBSY). Bus control is then passed either back to the processor to execute the next bus cycle of the instruction or to another device requesting on the NPR line. If a device becomes master via a BR request line, control may be passed actively back to the processor by using the INTR operation or passively (by drop-

ping BBSY). If control is given up actively, only NPR requests will be honored during the interrupt sequence of updating the PC and PS. If control is given up passively, control may pass either to the processor to fetch the next instruction or to an NPR requesting device.

The PDP-11 provides Direct Device Addressing. All memory and devices on the Unibus are directly addressable and may be operated upon by all computer instructions. Direct device to device transfers are possible.

pdp11

INTRODUCTION TO RT-11

Order No. DEC-11-ORITA-A-D

digital

August 1977

This document is an introductory manual for the RT-11 V03 operating system. Its purpose is to acquaint new users with the RT-11 commands that perform common system operations. This manual presents the background material necessary to understand the system operations. It also contains a series of command examples and demonstration exercises that complement the text.

INTRODUCTION TO RT-11

Order No. DEC-11-ORITA-A-D

SUPERSESSION/UPDATE INFORMATION:	This is a new manual.
OPERATING SYSTEM AND VERSION:	RT-11 V03

digital equipment corporation · maynard. massachusetts

First Printing, August 1977

DIGITAL	DECsystem-10	MASSBUS
DEC	DECtape	OMNIBUS
PDP	DIBOL	OS/8
DECUS	EDUSYSTEM	PHA
UNIBUS	FLIP CHIP	RSTS
COMPUTER LABS	FOCAL	RSX
COMTEX	INDAC	TYPESET-8
DDT	LAB-8	TYPESET-10
DECCOMM	DECSYSTEM-20	TYPESET-11

CONTENTS

PREFACE

The RT-11 (Real Time-11) computer system is a single-user computer/operating system that serves the programming needs of both the beginning and the advanced programmer. It supports a number of programming languages, including industry-standard FORTRAN and BASIC; easily-learned FOCAL; APL; and for more advanced users, the PDP-11 assembly language, MACRO-11. In addition, it provides a comprehensive set of operating commands that programmers at all levels use to control system operations.

MANUAL INTENT

The purpose of this introductory manual is to acquaint you with a number of RT-11 operating commands that are used to perform common system operations. The manual does this by first presenting the background material that you need to understand a particular system operation; then it shows you how to apply the system operation in a series of operating commands and exercises that you recreate; finally, it provides a list of reference materials that contain more information about the operation. This approach makes it possible for you to learn quickly the major features of the system; at the same time, it eliminates many of the early learning problems encountered by new users.

This manual describes system usage fundamentals. It is not the intent of this manual to teach you to program the PDP-11 computer. You may already be proficient in one or more of the available programming languages. Likewise, no attempt has been made in this manual to cover all the possible applications for which the RT-11 computer system is suited. You will discover many applications yourself as you continue to use the system.

MANUAL DESIGN

This manual is designed specifically for three categories of RT-11 users:

- Those having little or no previous "hands-on" computer experience (including those whose experience has been limited to batch environments)

- Those who are experienced users of a computer system other than RT-11

- Those who have used previous versions of the RT-11 computer system but wish a quick introduction to the newest features of the current system (Version 3 and later releases)

The manual contains 17 chapters and 2 appendixes. The descriptions that follow and the chart at the end of this section will help you determine your own reading path.

Chapter 1, Introducing the RT-11 Computer System, discusses general system concepts. It introduces the roles of hardware and software in a computer system and describes the specific hardware and software components of the RT-11 computer system. Chapter 1 is intended for users in the first two categories.

Chapter 2, Starting the RT-11 Computer System, shows all users how to start the system.

Chapter 3, Interacting with the RT-11 Computer System, demonstrates how you use the console terminal to control system operations. Again, this chapter is most helpful to users in the first two categories.

Chapters 4 through 7 describe system operations that are useful to all categories of users. Each chapter begins with a textual explanation of a particular system operation and expands into computer demonstrations showing the operation in use. Topics covered are: Using the Monitor Command Language; Creating and Editing Text Files; Comparing Text Files; and Performing File Maintenance Operations. Experienced RT-11 users may prefer to skip the textual explanations and review only the computer exercises.

Chapter 8, Choosing a Programming Language, helps you determine which language to use. Choose BASIC-11, FORTRAN IV, MACRO-11, or a combination of these three languages to continue the exercises in the manual (BASIC-11 and FORTRAN IV capabilities are optional).

If your choice is FORTRAN IV, read Chapter 9, Running a FORTRAN IV Program.

If you wish to use BASIC-11, read Chapter 10, Running a BASIC-11 Program.

If you choose to exercise MACRO-11, read Chapter 11, Running an Assembly Language Program.

MACRO and FORTRAN users continue to Chapter 12, Linking Object Programs, and Chapter 13, Constructing Library Files.

All users should read Chapter 14, Debugging a User Program, which provides some suggestions for finding and fixing errors in user programs.

Those users who plan to exercise the foreground/background capability of the RT-11 system should read Chapter 15, Using the Foreground/Background Monitor.

Finally, all users should continue to Chapter 16, Using Indirect Files, which describes the procedure for performing operations unattended, and Chapter 17, which gives some Advice to New Users.

Two appendixes are provided for reference. Appendix A discusses system bootstrapping procedures that are not generally needed, but may be required by some system users. Appendix B provides some additional information on selected system usage.

A glossary of technical terms appears at the end of the manual for reference purposes.

The following flowchart will help you plan your reading path through the manual. Read the chart from top to bottom; answer the questions and follow the direction of the arrows to see which chapters you should read.

NOTE

The demonstration portions of this manual are for use with Version 3 and later releases of RT-11. The exercises are quite lengthy, and you may prefer not to complete them in one sitting. You may pause at the end of any individual chapter. It is important that you stop only at the end of a chapter since you will otherwise not complete an exercise and thus may introduce errors that will affect later exercises. Instructions for pausing and beginning again are given in Appendix B.

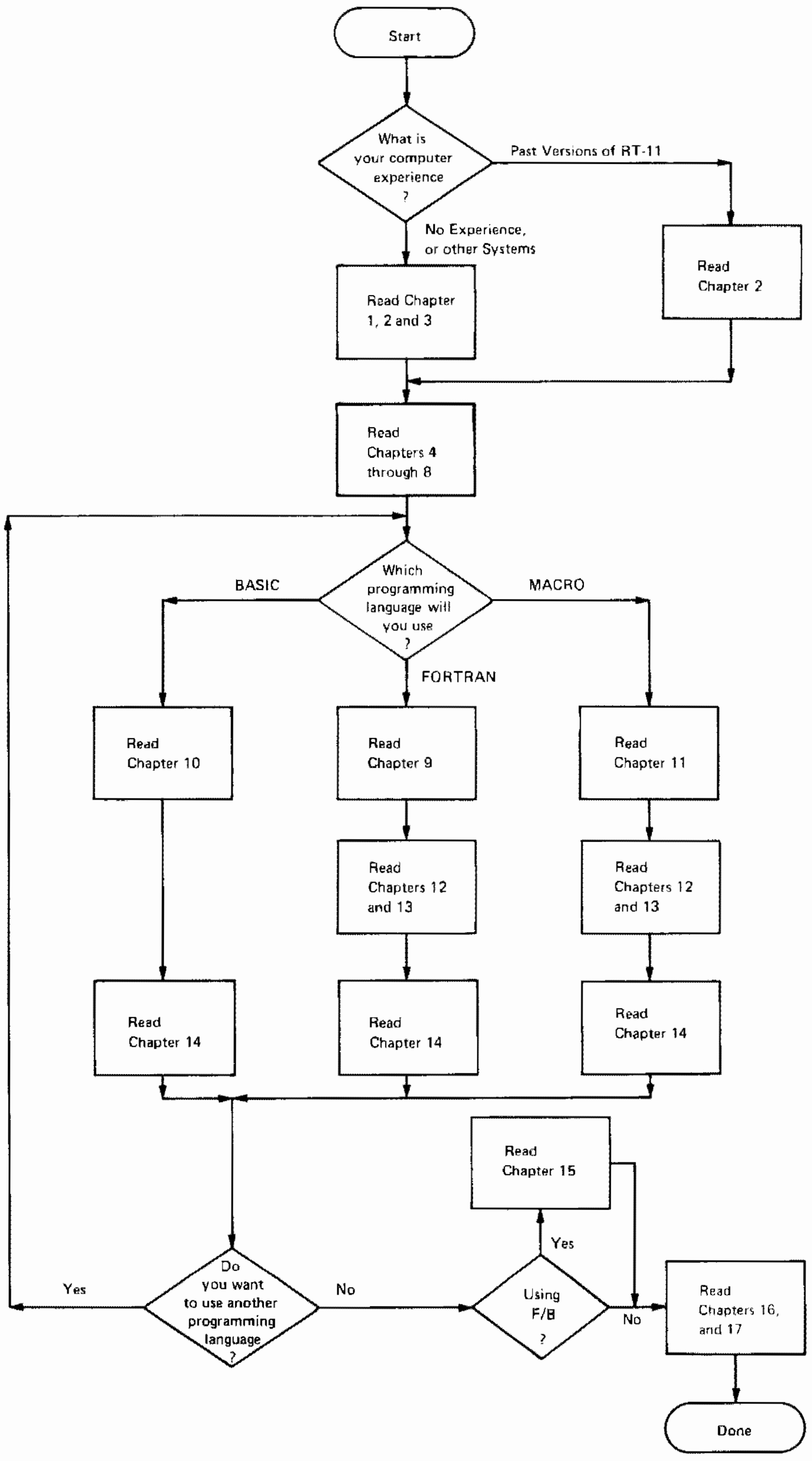


Figure PREFACE-1 Flowchart for Selective Reading

CHAPTER 1
INTRODUCING THE RT-11 COMPUTER SYSTEM

A computer system is a collection of components working together to process data. The purpose of a computer system is to make it as easy as possible for you to use a computer to solve problems. To accomplish this goal, hardware elements are combined with software elements to form a functioning unit. The hardware elements are the mechanical devices in the system, the machinery and the electronics that perform physical functions. The software elements are the programs that have been written for the system; these perform logical and mathematical operations and provide a means for you to control the system. Documentation includes the manuals and listings that tell you how to use the hardware and software. Collectively, these components provide a complete computer system that allows both layman and expert alike to use a computer.[1]

```
  SYSTEM HARDWARE
  SYSTEM SOFTWARE
+ SYSTEM DOCUMENTATION
----------------------
  COMPUTER SYSTEM
```

SYSTEM HARDWARE

The RT-11 computer system requires three basic hardware items: the computer itself, which performs all data processing; a terminal device, used like a typewriter for 2-way communication between the user and the system; and a storage medium, for storing programs and data. Figure 1-1 illustrates the hardware components of a typical RT-11 computer system.

The Computer

The computer does the real work of the system; it performs all instruction decoding and data processing. The RT-11 computer system is constructed around a DIGITAL PDP-11 computer, several of which are shown in Figure 1-2. Any model of PDP-11 can be used in an RT-11 system.

[1]This chapter attempts to build a working vocabulary that is both meaningful to the new user and consistent with standard DIGITAL terminology. Some definitions may appear inconsistent with those you have previously learned or used.

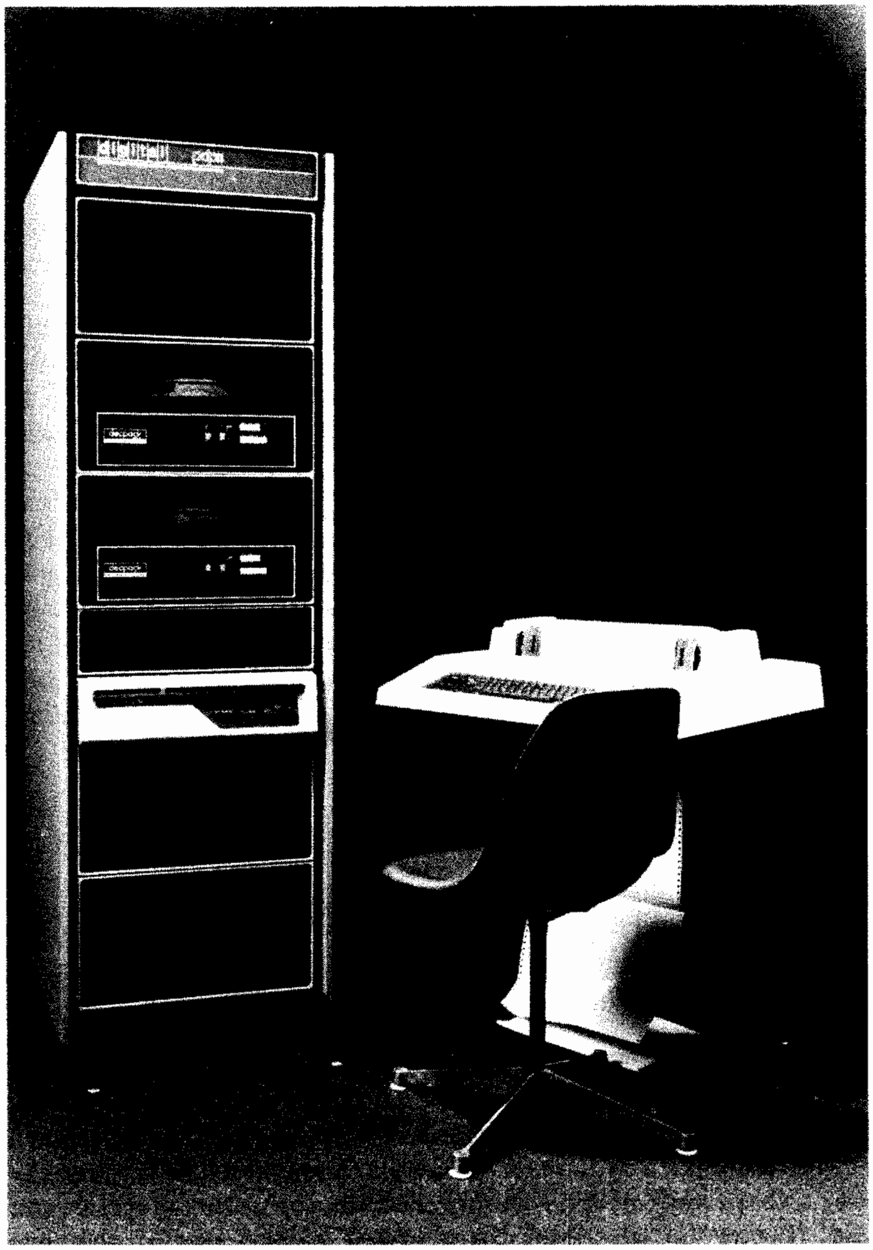


Figure 1-1 RT-11 Computer System

Notice in Figure 1-2 that the front panel, or operator's console, of each PDP-11 computer is slightly different. The switches, buttons, and lights that are on the operator's console can be used for various kinds of computer operations and applications. In the RT-11 computer system they are used only to start the system. Once the system has been started, your interaction with the computer system occurs through the terminal.

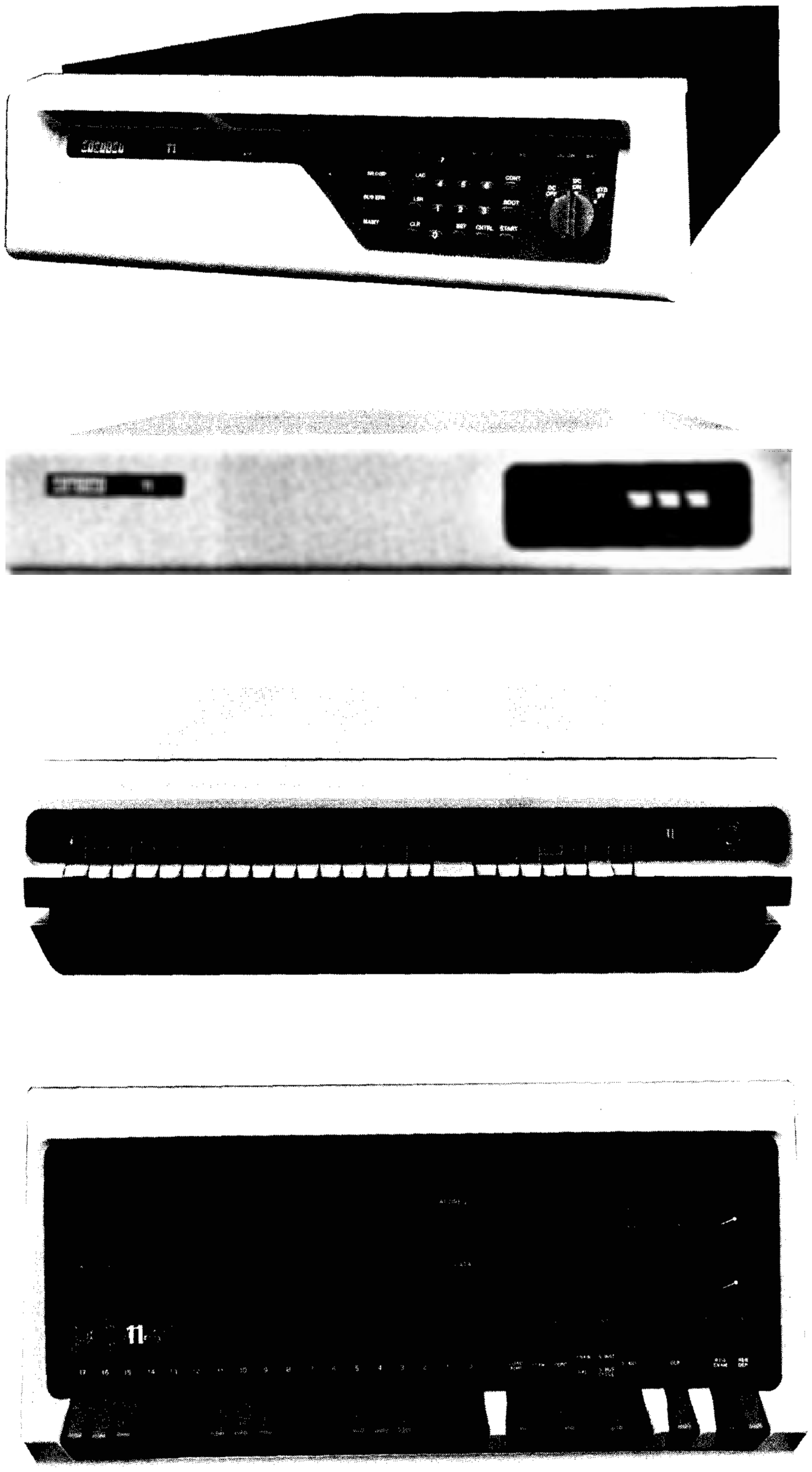
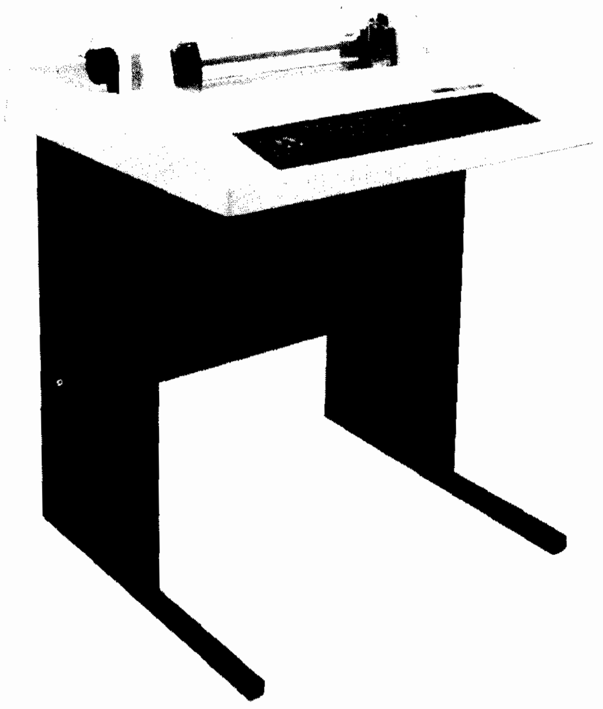


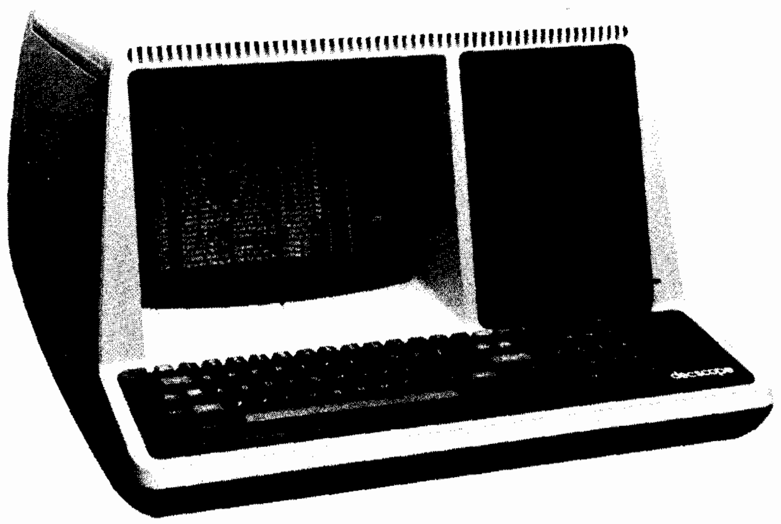
Figure 1-2 PDP-11 Computers

The Terminal

The terminal allows 2-way communication between you (the user) and the computer system. You enter information – operating commands, for example – from the terminal keyboard, which is operated much like a typewriter keyboard. The computer, in turn, prints information and messages on the terminal's printer or screen. Figure 1-3 shows many of the terminal devices that can be used in an RT-11 computer system.



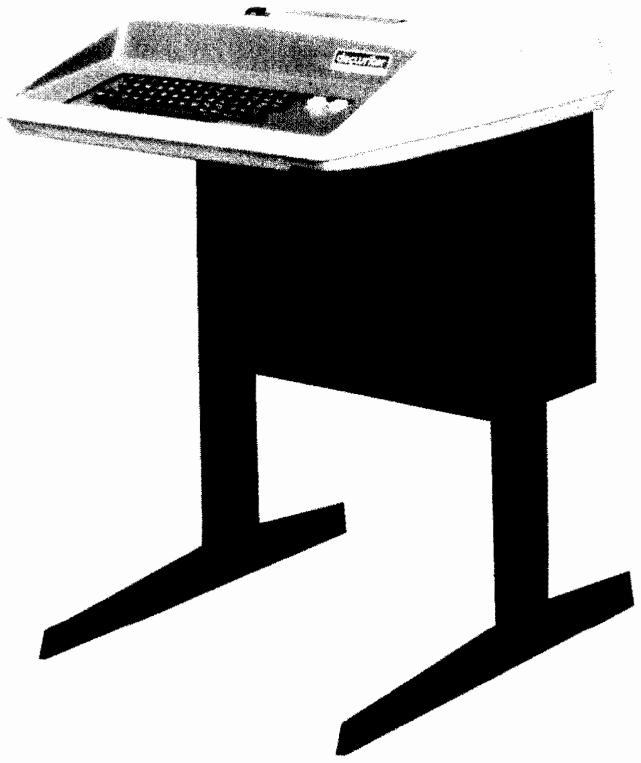
LA36



VT52

Figure 1-3 Terminal Devices

VT05



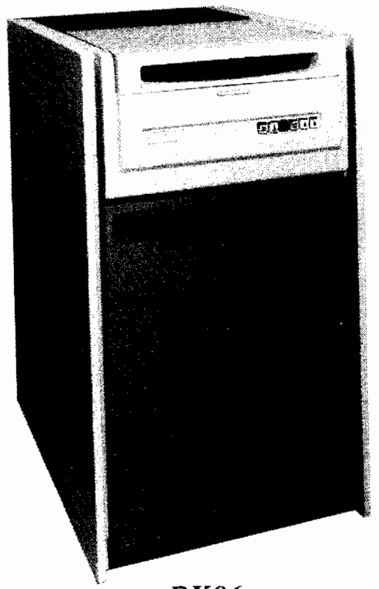
LA30

Figure 1-3 Terminal Devices (Cont.)

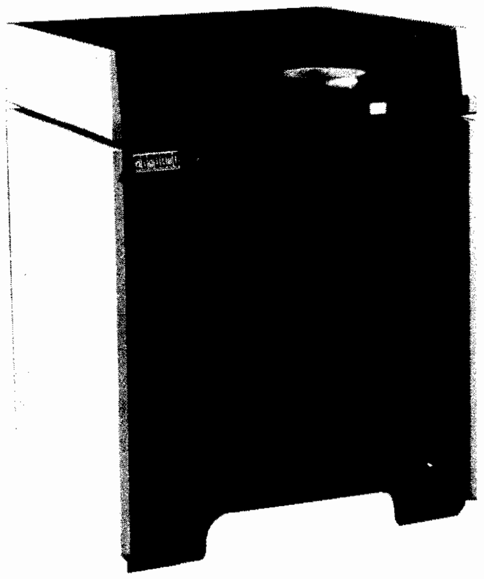
Generally, an RT-11 computer system has only one terminal through which all system/user interaction takes place. This is called the console terminal. If the system has more than one terminal, one of them is still designated the console terminal; others simply provide auxiliary message-printing capabilities.

The Storage Medium

The third important hardware device in an RT-11 computer system is the storage medium (usually a disk). It stores programs – those that make up the computer system software and those that you create. It serves as a distribution medium; system software is often packaged and distributed on a disk by the system supplier. Finally, it stores other data, information that is eventually needed for a computer operation (called input), the results of a computer operation (called output), or textual information such as a report. Figure 1-4 shows the random-access storage media (within their specific drive units) that can be used in an RT-11 computer system (random access means that access time for data is independent of the location of data. Contrast this concept with sequential access).

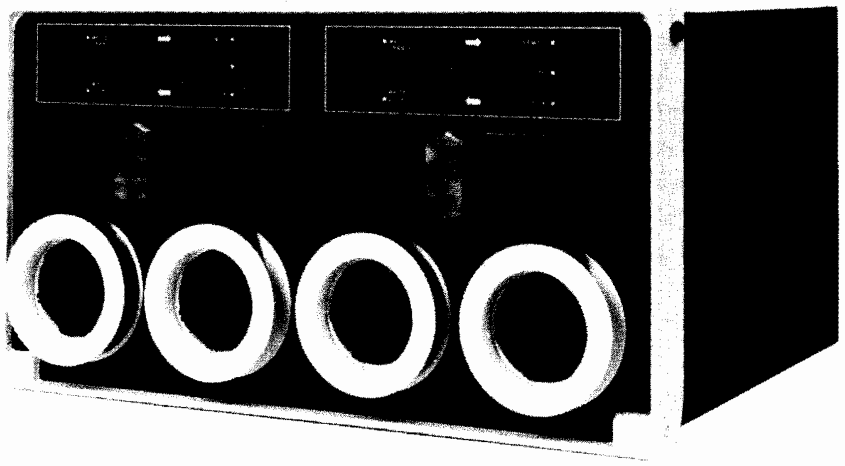


RK06

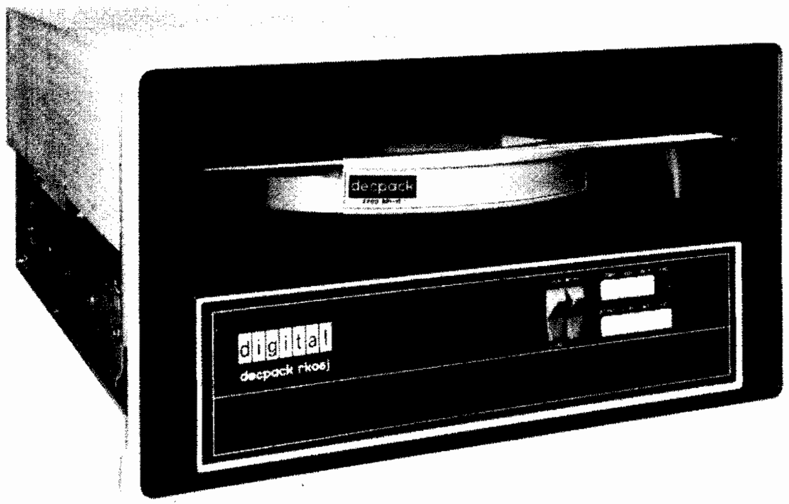


RP03

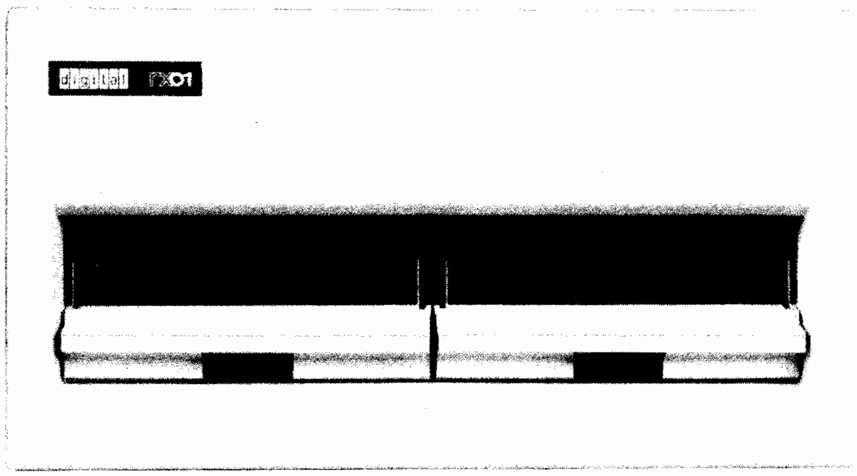
Figure 1-4 Random-access Storage Media and their Devices



DECtape



RK05



RX01 Diskette

Figure 1-4 Random-access Storage Media and their Devices (Cont.)

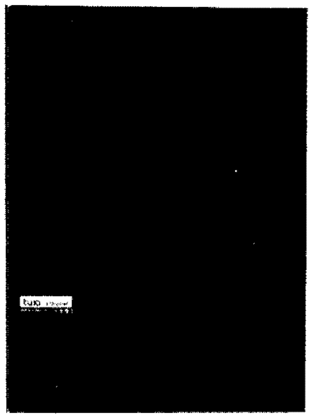
These three devices – the computer, the terminal, and the storage medium – are the required hardware components of an RT-11 computer system. With the exception of the computer, all hardware devices are called peripheral devices. Peripheral devices supplement the computer by providing external resources for operations that the computer cannot handle alone. In addition to the terminal and storage medium (which are required peripheral devices), other peripheral devices can be used in an RT-11 computer system.

Optional Devices

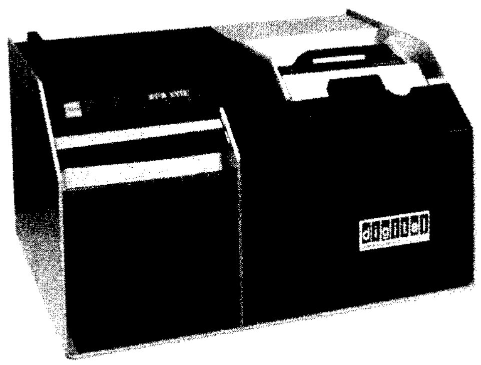
Optional peripheral devices are added to a computer system according to the specific needs of the system users. For example, computer systems that are used primarily for program development may have extra storage devices and a high-speed printing device. Computer systems used in a laboratory environment may have graphics display hardware, an oscilloscope device, and an analog-to-digital converter. Computer systems that provide (or use) information in conjunction with another kind of computer system usually have a magtape device because magtape is an industry-standard storage device.

Peripheral devices are categorized as input/output (I/O) devices since the functions they perform provide information (input) to the computer, accept information (output) from the computer, or do both. Some common input devices are card readers, paper tape readers, and programmable clocks. Output devices include line printers, paper tape punches, and plotters. Input/output devices include terminals and storage devices because they are capable of performing both input and output operations.

Figure 1-5 shows several of the optional peripheral devices that are often added to an RT-11 computer svstem.

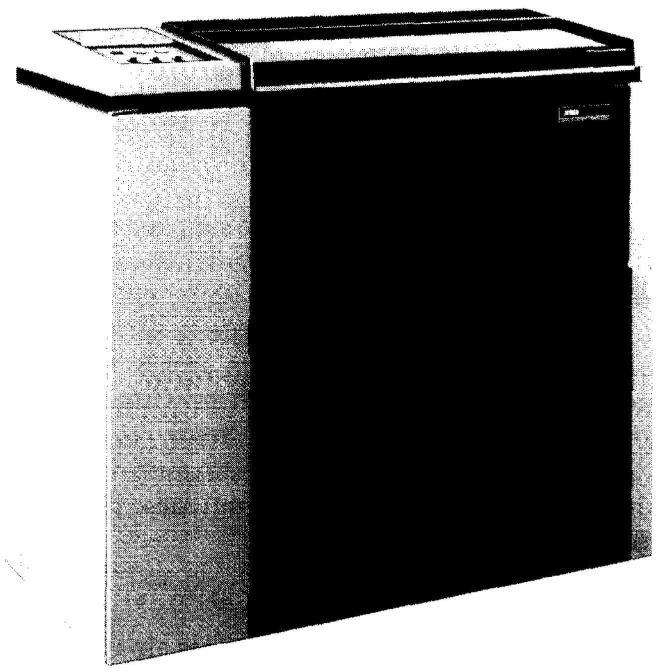


Magtape

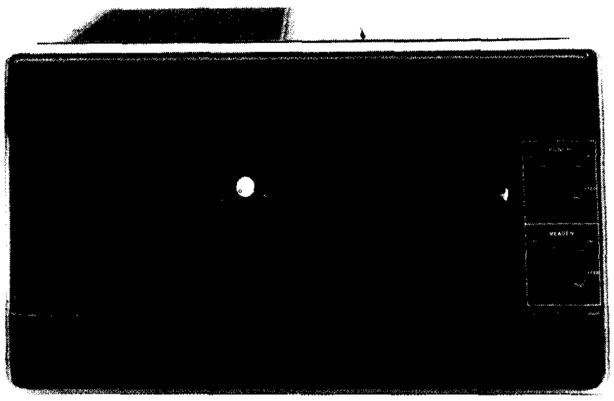


Card Reader

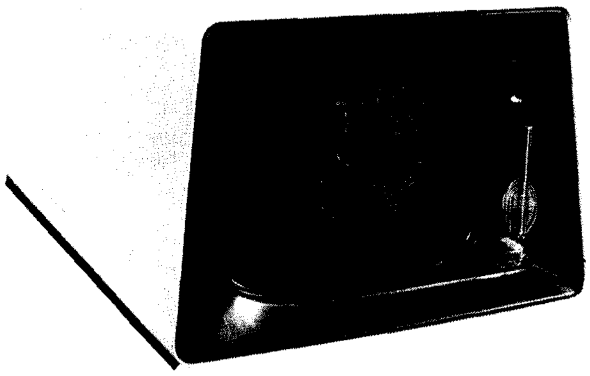
Figure 1-5 Peripheral Devices



Line Printer



Paper Tape Reader/Punch



VT-II Display

Figure 1-5 Peripheral Devices (Cont.)

The hardware configuration of your own RT-11 computer system includes the computer, the terminal, the storage medium, and any other peripheral devices you choose to add.

SYSTEM SOFTWARE

System software is an organized set of supplied programs that effectively transform the system hardware components into usable tools. These programs include operations, functions, and routines that make it easier for you to use the hardware to solve problems and produce results. For example, some system programs store and retrieve data among the various peripheral devices. Others perform difficult or lengthy mathematical calculations. Some programs allow you to create, edit, and process application programs of your own. Still others handle entire applications for you; these programs are strictly business-related or laboratory-related.

As illustrated in Figure 1-6, system software always includes an operating system, which is the "intelligence" of the computer system. Usually the system software includes one or several language processors; it sometimes also includes specific applications.

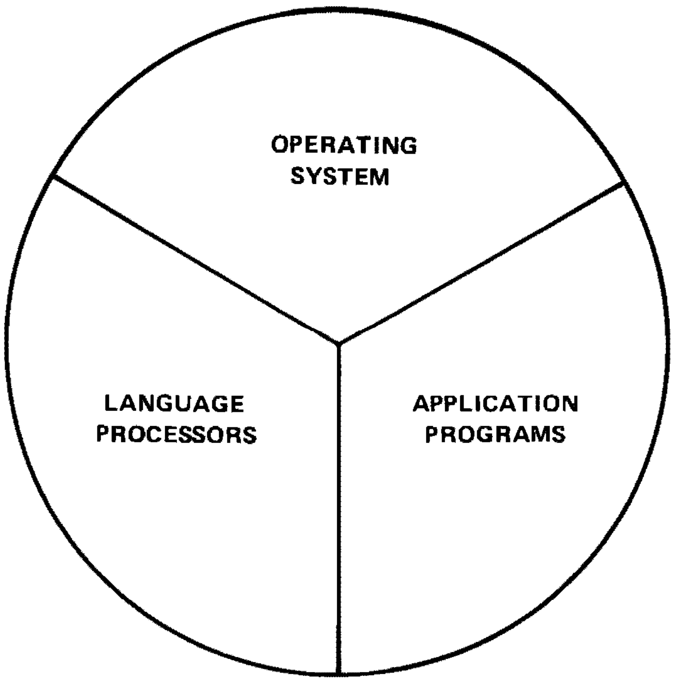


Figure 1-6 RT-11 System Software

The RT-11 Operating System

An operating system is a collection of programs that provides an environment in which you can create and run programs of your own. The operating system organizes all the hardware and software resources of the computer system into a working unit and gives you control.

The RT-11 operating system comprises a monitor/executive program for system control and supervision; several device handlers (programs), one for each of the supported hardware devices; a variety of utility programs for program/data creation and manipulation; and finally, the interfaces that are necessary to support several programming language processors. The operating system is illustrated in Figure 1-7.

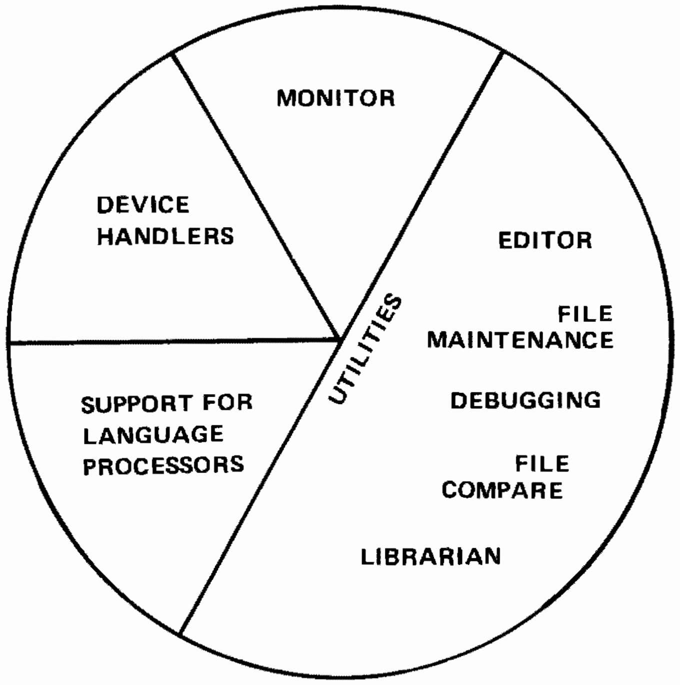


Figure 1-7 RT-11 Operating System

The monitor (executive) program is the interface between the system hardware, the system software, and you. Part of the monitor function is to accept, process, and execute your instructions for controlling the system. A comprehensive set of monitor operating commands allows you to direct, from the console terminal keyboard, those system operations that you want to occur.

Device handlers are routines that provide the interface to the various hardware devices that are part of the computer system. A handler exists for every peripheral device that the system supports.

Utility programs cover a wide range of resources; such programs allow you to create and edit text, maintain other programs, and help you locate user-programming errors. Some specific utility programs in the RT-11 operating system are the following:

- An editor, which allows you to create and modify textual material; this material could be the statements that make up a computer program, a memo, or any text you wish to create
- File maintenance utility programs, which allow you to manipulate and maintain your programs and data – to transfer them between devices, to update them, and to delete them when you are done with them
- A debugging program, which helps you uncover and correct errors in your programs
- A librarian, which makes it easy for you to store and retrieve often-used programming routines
- A linking program, which converts object modules into a format suitable for loading and execution
- A source comparison program, which is used to compare two ASCII files and to output any differences to a specified output device
- A dump program, which outputs to the console or lineprinter all or any part of a file in octal words, octal bytes, ASCII characters and/or Radix-50 characters.

The RT-11 operating system also provides support for several programming languages and their respective language processors.

Language Processors

A language processor is a translating program that you use to process a source program you have created. A language processor exists for every programming language supported by the system, whether it is a high-level language or a machine-level language.[1]

High-level languages, such as BASIC-11 and FORTRAN IV, are relatively easy languages to learn and use. Since a single language statement often performs a whole series of intricate computer operations,

[1] Language selection is discussed in Chapter 8 of this manual.

high-level languages let you direct your attention to solving the problem at hand. They do not require that you understand how the computer interprets the problem. High-level languages supported by the RT-11 operating system, in addition to FORTRAN and BASIC, include FOCAL-11, APL, and DIBOL, DEC's interactive commercial language.

Machine-level or assembly languages are available for users who prefer to work at the instruction level of the computer. At this level, you have control over such factors as program size and speed of execution. Machine-level languages do require that you be familiar with the computer and the hardware devices of the system. RT-11 provides the MACRO-11 assembly language processor for those who would rather work at this more intricate level.

Application Packages

The RT-11 operating system supports several applications packages. These include a laboratory applications package for the standard functions found in most laboratory environments. Another package called GAMMA-11 is designed specifically for the needs of a nuclear medicine laboratory. A scientific subroutine package (for FORTRAN users) provides a large selection of mathematical and statistical routines commonly required in scientific programming. And a graphics support package for BASIC and FORTRAN users provides display features such as vectors, alphanumerics, points, multi-intensities and blinks. Because of the specialized nature of these applications packages, they are not described further in this manual.

SYSTEM DOCUMENTATION

The third and final component of a computer system is its documentation. This includes manuals that tell how you use the software and hardware of the computer system, plus any source listings of actual programs that make up the operating system.

Hardware Manuals

Hardware manuals describe the devices in the computer system. RT-11 hardware documentation includes a Processor Handbook that describes the PDP-11 computer you are using, and a User's Guide or Maintenance Manual for each peripheral device in your computer system. These manuals tell you how to operate the devices and give you special programming information that you may need if you intend to write device drivers or special system software that involve the devices.

Software Manuals

Software manuals[1] describe the operating system and the language processors. RT-11 software documentation falls into three major categories: introductory or once-only manuals (intended to be used once and then stored away); computer manuals (intended to be used at the computer); and desk manuals (intended to be used at your desk for reference purposes).

Once-only manuals include this manual and others that are needed only when your system is initially installed. You may have little or no occasion to use these manuals once your computer system is in operation and you are familiar with its use.

Computer manuals are those manuals that tell you how to use the computer system. They describe in detail command usage and syntax, list summaries of system operations, and give the meanings of system messages.

Desk manuals are those manuals that you continually use for reference as you write your own application programs. These manuals include the general language reference manuals and the advanced programming manuals that contain programming information specific to the RT-11 computer system.

Source Listings

Source listings are actual listings of the assembly-language code that makes up the RT-11 operating system. These listings are very detailed and are generally needed only if you intend to modify the system software. They can be ordered on micro-fiche film from the DIGITAL Software Distribution Center.

This completes a general introduction to the RT-11 computer system. Subsequent chapters of this manual describe how you use the various system components mentioned here to perform a series of related computer operations. You begin in Chapter 2 by learning how to start the RT-11 computer system.

REFERENCES

Eckhouse, Richard H., *Minicomputer Systems: Organization and Programming (PDP-11)*. Englewood Cliffs, New Jersey: Prentice-Hall, 1975.

A guide to programming fundamentals, PDP-11 organization and structure, and programming techniques. See Chapter 1.

[1]All RT-11-related software manuals are listed in the RT-11 Documentation Directory. Many of these manuals are provided with your system; others can be ordered from the DIGITAL Software Distribution Center.

Katzan, Harry Jr., Information Technology, *The Human Use of Computers*. New York: Mason & Lipscomb Publishers, Inc., Petrocelli Books, 1974.

An introductory textbook covering basic computing concepts, programming languages, and topics in computers and society. See Chapters 1, 2, 4, 5, and 10.

PDP-11 Computer Family, Products and Services. Maynard, Mass.: Digital Equipment Corporation, 1977.

An overview of the available PDP-11 family products and services; includes capsule descriptions of the various PDP-11 computers, peripherals, and operating systems, and describes the supportive services provided by DIGITAL.

PDP-11 Peripherals Handbook. Maynard, Mass.: Digital Equipment Corporation, 1976.

A technical summary of the available PDP-11 peripheral devices; includes descriptions, specifications, programming, and interfacing information for PDP-11 peripheral devices.

PDP-11 Processor Handbook. Maynard, Mass.: Digital Equipment Corporation, 1975.

A hardware manual for the owners and users of the PDP-11 family of computers and for those who will be using the PDP-11 assembly language instruction set.

PDP-11 Software Handbook. Maynard, Mass.: Digital Equipment Corporation, 1975.

A general overview and introduction to available PDP-11 software, operating systems, and language processors.

RT-11 Documentation Directory (DEC-11-ORDDB-A-D). Maynard, Mass.: Digital Equipment Corporation, 1977.

A listing and brief summary of current RT-11-related software documentation.

Spencer, Donald D., *Fundamentals of Digital Computers*. Indianapolis, Kansas City, New York: Howard W. Sams & Co., Inc., The Bobbs-Merrill Co., Inc., 1969.

A discussion of the history and evolution of computers, computer applications, and fundamentals of computer use. See Chapters 1 through 12 and Chapter 20.

CHAPTER 2
STARTING THE RT-11 COMPUTER SYSTEM

Before you can use the RT-11 computer system to perform any operations, you must start it. Starting the system involves turning on the computer and the various hardware devices and loading the appropriate software components into computer memory.

COMPUTER MEMORY

Within every PDP-11 computer is a physical, designated storage area called memory. Computer memory is where system information and data is temporarily loaded and stored for use during the various system operations.

Each time you use the computer system, there may already be information in computer memory left there by whoever used the system last. For example, there may be the results or data of another user's program; there may be the results of a particular system operation; there may even be an entirely different operating system in memory. For your use, computer memory must contain the RT-11 operating system, and specifically the RT-11 monitor program. Thus, your first operation as a system user is to transfer the monitor program from the disk device, where it was stored during system installation, to computer memory, where you can use it. The process of transferring the RT-11 monitor is called bootstrapping the system and is the only system operation that requires you to use the operators console on the front panel of the computer (see Figure 2-1).

HARDWARE CONFIGURATION

Starting the RT-11 computer system requires that you know how to operate your system's hardware devices. Since you may not have had the opportunity to use any of the devices yet, ask an experienced user to help you the first time. Follow the instructions in the section in this chapter entitled "Bootstrap Procedure." If necessary, refer to the various hardware manuals provided with your system and to any special instructions that have been left by the DIGITAL representative who installed your system.

First read through the following material and fill in the appropriate information where requested. You should be able to determine all responses by checking the RT-11 System Generation Manual.

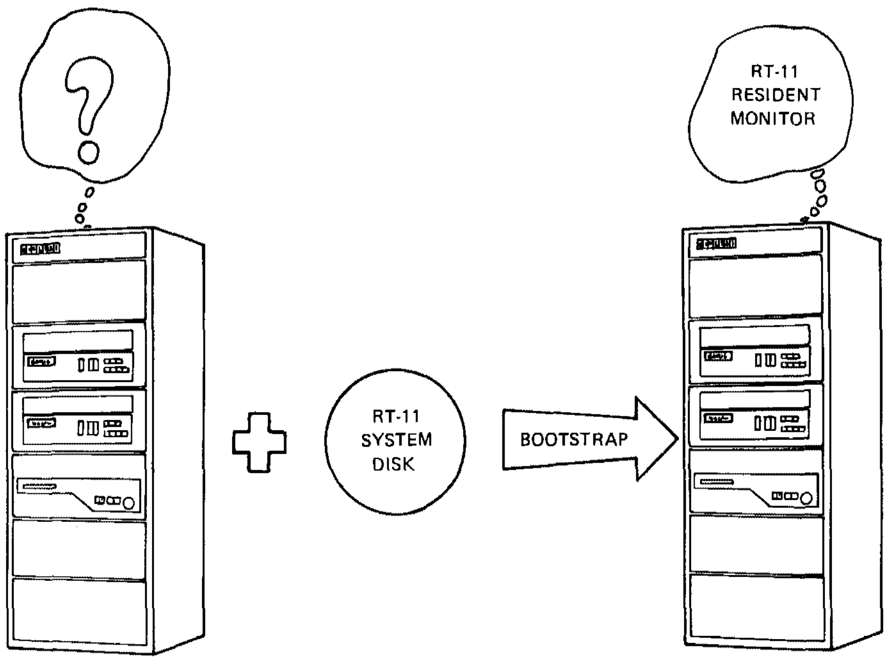


Figure 2-1 The Bootstrap/Computer Relationship

You must have the following materials to start the system and to perform the exercises in this manual:

- The disk device containing the RT-11 operating system (called the system volume); a system volume may have been created specifically for your use with this manual
- The volume containing the FORTRAN and/or BASIC language processors if these languages are not stored on the system volume (available only to FORTRAN and BASIC users)
- A volume for program storage (for example, magtape or another disk); this volume should contain no important information since all information on it will be erased during a later computer exercise
- A copy of the RT-11 System Generation Manual

NOTE

Hardware configuration information, along with instructions for starting (bootstrapping) your RT-11 system, should have been provided by the DIGITAL representative who initially installed your system. This information should appear in the RT-11 System Generation Manual and should be adequate for you to answer all the questions asked here. If you have trouble, see Appendix B, "Suggestions for Bootstrapping the System." Do not continue to any other chapter in this manual until you understand the following configuration information and can bootstrap the system yourself.

Terminal

1. What kind of terminal device are you using (for example, LA36 DECwriter II, VT52 video terminal, etc.)?

Computer

2a. Does your computer operator's console have pushbuttons or switches?

2b. How much memory does your computer have?

System Volume

3a. What kind of system volume are you using (for example, RK06 disk, RX01 diskette, etc.)?

3b. What is the 2-letter code for this volume (typical codes are given in Table 2-1; respond with the code for your own volume)?

Table 2-1 Representative System Volumes

Volume	Code
RX01 diskette	DX
RK05 disk	RK
RK06 disk	DM
RP02/03 disk	DP
RF11 disk	RF
RJS03/4 disk	DS
TC11 DECtape	DT

Storage Volume

4a. What volume are you using for program storage (for example, TM11 magtape, RK05 disk)?

4b. In which device unit will you use this volume (0, 1, etc.? choose any available device unit)?

Optional Devices and Supported Languages

5. What peripheral devices are part of your system (for example, line printer, magtape, VT11 display hardware; list all devices other than the terminal and the computer)?

6. What programming languages does your system support (MACRO-11 or BASIC-11, for example)?

BOOTSTRAP PROCEDURE

Once you have determined your hardware configuration, you are ready to bootstrap the system. The purpose of the bootstrap procedure is to load and start the RT-11 monitor in computer memory, making the RT-11 computer system available for you to use.

1. Turn the terminal to an on-line condition. If there is a baud rate switch, set it to 300.

2. Make sure the computer power is on and that the computer is not already in use. Stop the computer:

   - If your operator's console has switches, set the switches to HALT, then ENABLE

   - If your operator's console has pushbuttons, locate the button labeled CNTRL; hold it down and push the button labeled HLT/SS; then release both.

3. Place the system volume in its corresponding device unit 0. Ensure that the system volume is write-protected (for all except RX01 diskette, which is always write-enabled).

4. Place the storage volume in the device unit noted in question 4b in the Hardware Configuration section. Ensure that this volume is write-enabled.

5. Check the operator's console on your computer (refer to question 2a in the Hardware Configuration section). If your console has pushbuttons, continue. Otherwise, go to step 8.

6. Locate the pushbutton labeled CNTRL, hold it down and push the button labeled BOOT. Check the terminal printer or screen. If there is no response, read the section in Appendix A entitled "Using a Pushbutton Console to Bootstrap"; otherwise continue to step 7.

7. Your terminal printer or screen should show several numbers followed by:

   ```
   $
   ```

   Type on the terminal keyboard the 2-letter code that represents your system volume (from question 3b in the Hardware Configuration section) followed by a carriage return (the RETURN key), represented throughout the text by the characters (RET)[1]. Be sure to use the SHIFT key so that you type upper-case characters. For example, for RX01 diskette, type:

   ```
   DX (RET)
   ```

   Continue to step 11.

8. Check your switch console. If it has a 3-way dial labeled DC OFF, DC ON, and STAND BY, go to step 9. If it has three individual switches labeled DC ON/OFF, ENABLE/HALT, and LTC ON/OFF, go to step 10. If it has a long row of switches across the entire console, read the section in Appendix A entitled "Using a Switch Register Console to Bootstrap".

9. Set the 3-way dial to DC ON. Then locate the BOOT switch (to the left of the dial) and raise it. Go to step 11.

10. Put all three switches in the up position; then move the DC ON/OFF switch down and up and check the terminal response.

    - If it is:

      ```
      $
      ```

      type on the terminal keyboard the 2-letter code that represents your system volume (from question 3b in the Hardware Configuration section) followed by a carriage return (the RETURN key), represented throughout the text by the symbol (RET). Be sure to use the SHIFT key so that you type upper-case characters. For example, for RX01 diskette, type:

      ```
      DX (RET)
      ```

[1]The RK05 disk is an exception. Hardware bootstraps use "DK", not "RK", for RK05.

Continue to step 11.

- Any other response indicates that you must type the bootstrap on the terminal keyboard. Read the section in Appendix A entitled "Typing the Bootstrap on the Terminal Keyboard."

11. If your system has been correctly bootstrapped, a message prints on the console terminal. Check this message; it should read:

    RT-11SJ V03-xx (the xx's have developmental significance only and can be ignored)

    If this version number (with the exception of the xx's) does not appear, read the section in Appendix B entitled "Suggestions for Bootstrapping the System."

    The proper response indicates that the monitor component of the RT-11 operating system is active. Set the system volume to a write-enabled condition (for all except RX01 diskette, which is always write-enabled).

You should now direct your attention to the console terminal since system interaction continues on this device.

REFERENCES

DECscope User's Manual[1] (EK-OVT5X-OP-001), Cross Products, 1975.

A hardware manual for the owners and operators of the DECscope (VT50) family of video terminals and for those who will be programming computers to interact with these devices.

PDP-11 Processor Handbook, Maynard, Mass.: Digital Equipment Corporation, 1975.

A hardware manual for the owners and users of the PDP-11 family of computers and for those who will be using the PDP-11 assembly language instruction set.

RX8/RX11 Floppy Disk System Maintenance Manual[1] (EK-ORX01-MM-PRE2), Maynard, Mass.: Digital Equipment Corporation, 1975.

A hardware manual for the owners and operators of RX01 diskettes and for those who will be programming computers to interact with this device.

RT-11 System Generation Manual (DEC-11-ORGMB-A-D) and *RT-11 System Release Notes* (DEC-11-ORNRB-A-D). Maynard, Mass.: Digital Equipment Corporation, 1977.

Two RT-11-specific software manuals that contain instructions for installing, customizing, and starting the RT-11 computer system.

[1]Used as an example; consult hardware user or maintenance manuals specific to your system.

CHAPTER 3
INTERACTING WITH THE RT-11 COMPUTER SYSTEM

Interaction with the RT-11 computer system involves an exchange of information between you (the user) and the software operating system. The exchange may be active, with you dictating command information from the terminal keyboard and the system responding immediately; or it may involve the storing of information on mass storage volumes for later use.

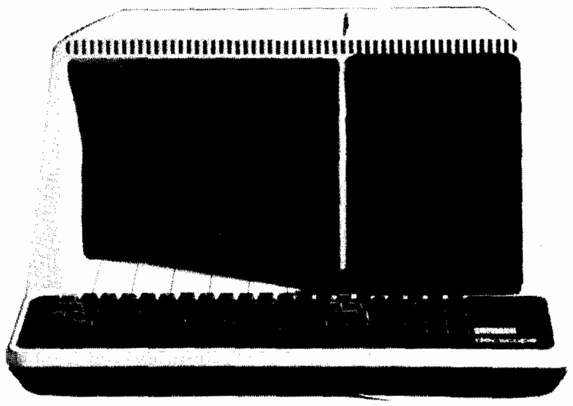
USING THE CONSOLE TERMINAL TO EXCHANGE INFORMATION

During the bootstrap procedure you activated the RT-11 computer system by loading and starting the monitor program in computer memory. One of the functions of the monitor program is to provide you with the capability to use the console terminal. Since the console terminal can perform both input and output operations, it is used to interface between the system and the user. With it, you can:

- Type the commands that control system operation
- Receive messages and responses from the system

All console terminals have a keyboard used to enter information, and a paper output device or video screen used to echo characters typed at the keyboard and to print system messages and responses. Figure 3-1 shows the two most commonly used terminals, the LA36 and the VT52.

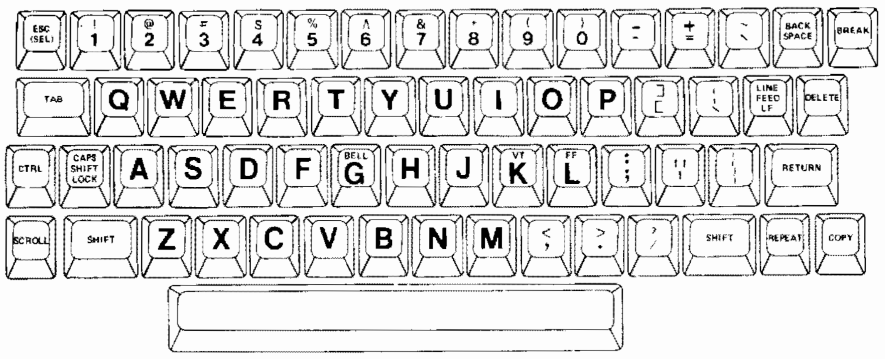
The difference between these two terminals occurs in their output mechanism. While the LA36 terminal has only a paper printer, the VT52 has a video screen. The screen and the paper printer serve the same purpose – they show user input and system responses; however, paper output can be saved for later use while screen output is temporary. The keyboards of both terminals are the same and are shown in Figure 3-2. Also shown in this figure is an LA30 (VT05) keyboard so that you can note some of the differences found in the keyboards of older terminals.



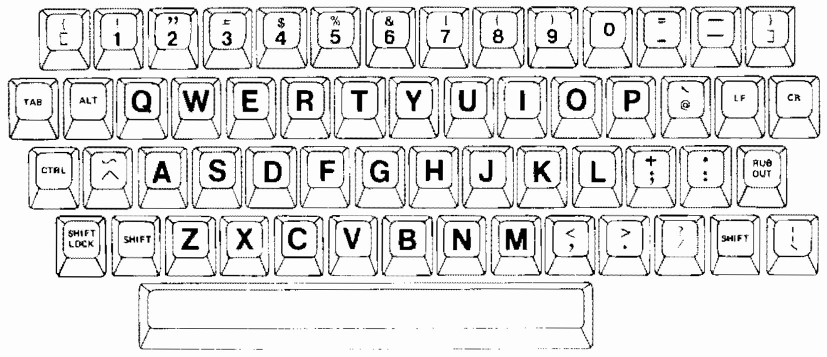
VT52

LA36

Figure 3-1 LA36/VT52 Terminals



VT52/LA36 Keyboard



VT05/LA30 Keyboard

Figure 3-2 Keyboard Layouts

Using Figure 3-2 as a guide, study your own terminal keyboard. First, notice that the keys for the alphabetic characters are positioned in the same way as on most standard typewriters. The SHIFT key allows you to select between numeric and special characters and between upper- and lower-case characters[1]. The position of the numeric and special characters varies somewhat among the different terminals so you may need to hunt for a particular key until you become familiar with your own terminal.

Locate the DELETE key (LA36/VT52 terminals) or the RUBOUT key (LA30/VT05 terminals). These keys perform the same function: they are used to correct a typing mistake. Pressing the key once cancels the last character typed. Pressing it twice cancels the last two characters, and so on, back to the beginning of the line.

[1]With the exception of system messages and one other exception explained in Chapter 5, the RT-11 computer system uses upper-case characters exclusively.

Locate the TAB key. Tab stops on a computer terminal are positioned every eight spaces across the line, beginning at column 1. Pressing the TAB key moves the character pointer (that is, the position on the line where the next character will be typed) to the beginning of the next tab stop.

The key marked RETURN (LA36/VT52 terminals) or CR (LA30/VT05 terminals) performs a carriage return; it both returns the character pointer to the beginning of the line and advances it to the next line. This key is used to terminate the line currently being typed and to terminate certain RT-11 system commands.

Locate the ESC (SEL) key and LINE FEED key (LA36/VT52 terminals) and ALT and LF keys (LA30/VT05 terminals). These are special command terminators that are described later in Chapters 5 and 14.

An important key is the CTRL key. The CTRL key is always used in conjunction with another character key to perform one of several specific system operations. CTRL commands are explained in detail when you begin to use them later in the manual.

Table 3-1 reviews the console terminal keyboard characters. Keys not specifically mentioned are not used by the RT-11 computer system and can be ignored.

You will have ample opportunity to become familiar with your terminal keyboard as you perform the demonstrations in this manual.

USING MASS STORAGE VOLUMES TO STORE INFORMATION

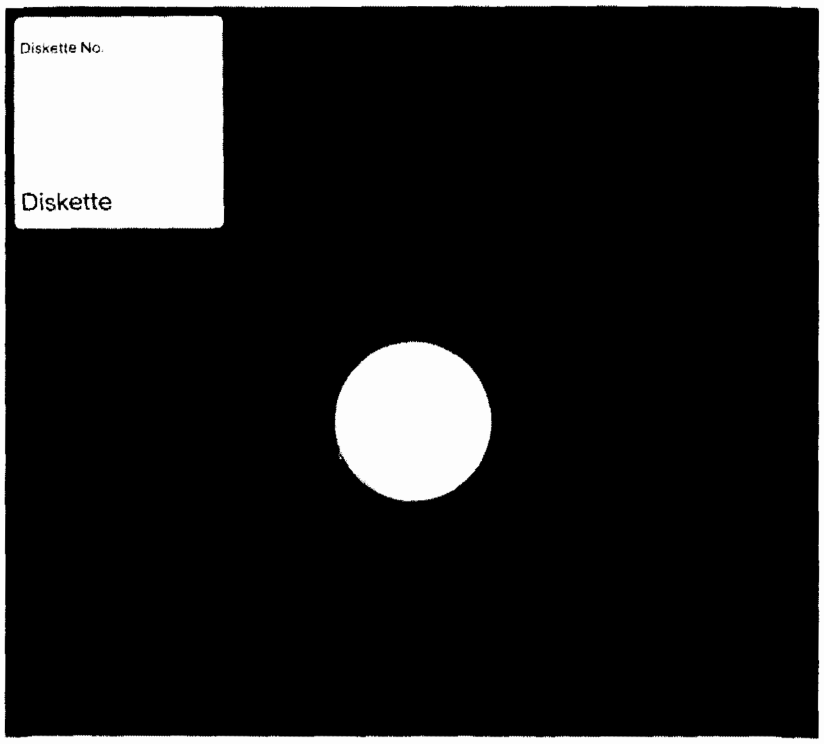
Mass storage volumes provide an area (apart from computer memory) to keep information for later use. The information may be user application programs, data needed by a program, the results of a program run, textual information, batch-type programs, and so on. As an example, the RT-11 operating system is stored on a mass storage volume called the system volume. When information is needed, as it was during bootstrapping, you can transfer the information from the storage volume into computer memory, where it can be used.

Before you can access the information stored on any storage volume, however, you must first insert the volume (the medium) into its

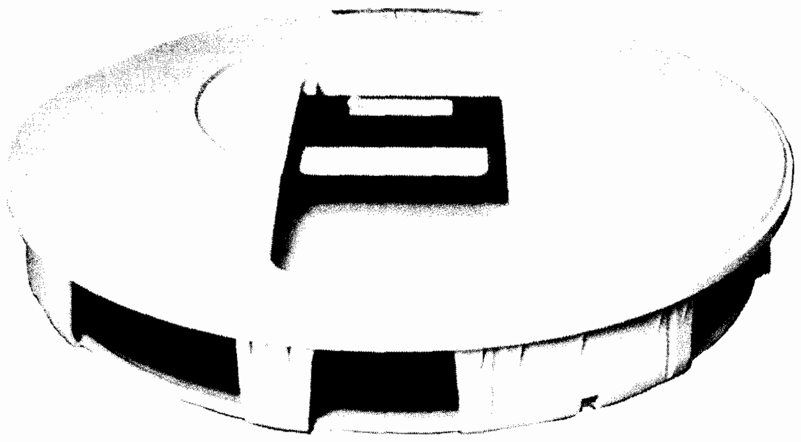
Table 3-1 Keyboard Characters

Key	Function
ALT ALTMODE	See ESC
BACK SPACE	Ignored during normal system use
BREAK	Ignored during normal system use
CR	See RETURN
CTRL	Control; part of several two-key command combinations that perform specific system functions
DELETE	Erase; cancels the last character typed
ESC	Command terminator; terminates an editing command string; typed twice, transmits the command(s) to the computer and performs a carriage return
LF LINE FEED	Command terminator; terminates certain system commands; transmits the command to the computer and performs a carriage return
NEW LINE	See LF
REPEAT	Ignored during normal system use
RETURN	Line terminator, command terminator; terminates the current line; terminates certain system commands; transmits the command to the computer and performs a carriage return
RUBOUT	See DELETE
SHIFT	Selects the uppermost of two characters appearing on a key
TAB	Moves the character pointer ahead to the beginning of the next tab stop
any other key	Transmits the alphanumeric or special character to the computer

corresponding device unit (drive) which is the hardware device connected to the computer. Once a volume has been inserted into a device unit, the unit's symbol also identifies the volume. There may be more than one device unit for any given volume, in which case each individual device unit is numbered 0, 1, 2, and so on. As you learned in the bootstrap procedure, the system volume is inserted in device unit 0 and remains in this device unit as long as you are using the system. Other storage volumes can be inserted in any available (corresponding) device units. Figure 3-3 illustrates several mass storage volumes.



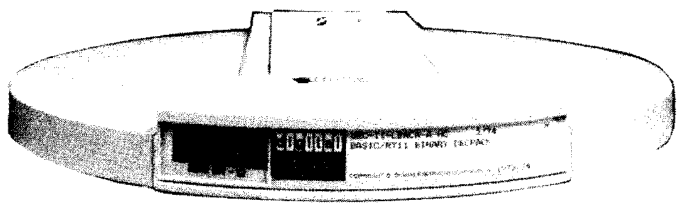
Diskette

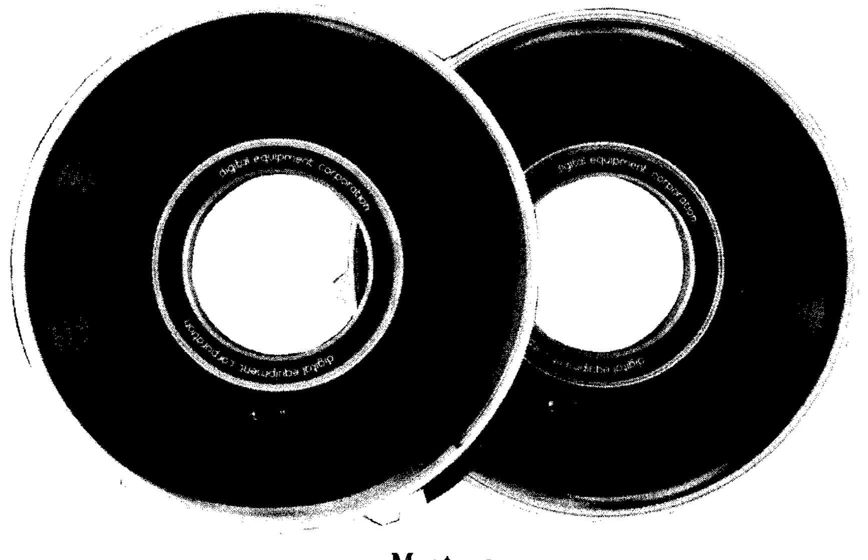


RK06

Figure 3-3 Mass Storage Volumes

RP03


RK05



Magtape

Figure 3-3 Mass Storage Volumes (Cont.)

Mass storage volumes are capable of holding large amounts of information. However, most volumes are physically small enough so that you can transport them away from the system, to your desk perhaps, or to another computer system. In addition to all disks (shown earlier in Figure 1-4), magtapes and cassettes are also mass storage volumes.

File Storage

You store information on a mass storage volume in the form of files. Each file is simply a logical collection of data. Files may be parts of programs or entire programs, program input data, or text such as a letter or report. Whatever its content, each file is treated as a unit and occupies a fixed physical area of the volume.

Every file on a mass storage volume has a unique name that is composed of a file name and file type. The file name and file type serve to identify the file and distinguish it from other files on the volume. You can instruct the system to print on your terminal the names of all files on any given volume. The resulting list is called the volume directory listing. By referring to the volume directory, you can find the name, size, and creation date of each file residing on that volume and erase old files that you no longer need. Whenever you perform an operation that affects the contents of the volume, a new volume directory reflects the change.

File Protection

Occasionally, after many files are added to a storage volume, the volume runs out of room for new information. The storage volume may also become damaged, lost, stolen, or worn through use. For these reasons it is a good idea to have several extra storage volumes on hand and to protect your more important files against accidental erasure or loss.

One way to protect a file is to make a copy of it on a second storage volume. The copy is called a backup file and is your security in case the original file (or its respective storage volume) becomes damaged or lost.

In addition, some storage volumes provide a mechanism that protects files against accidental erasure. This mechanism is generally a switch on the volume itself, or on the device unit, that you can manually set to a write-protect or write-enable condition (as you did during bootstrapping). When the volume is write-protected, information can be copied only from the volume to computer memory or to another volume that is write-enabled. A volume that is write-enabled, on the other hand, additionally allows information to be copied from memory back to the volume.

The RT-11 operating system itself also provides a protection feature. This optional feature requires that you confirm certain system commands that might otherwise erase important information. The system also issues prompting messages to ensure that you provide the proper file information when it is needed by a command.

In Chapter 4 and succeeding chapters you will use the terminal to enter command information and you will start performing file copy and other system operations. Before you continue, make sure that there is a backup copy of your system volume. If you cannot locate one, read Appendix B, Backing Up the System Volume, before going on.

REFERENCES

DECscope Users' Manual[1] (EK-VT5X-OP-001), Maynard, Mass.: Digital Equipment Corporation, 1975.

A hardware manual for the owners and operators of VT-50 and VT-52 video terminals and for those who will be programming the computer to interact with these devices.

LA36/LA35 DECwriter II User's Manual[1] (EK-LA3635-OP-001), Maynard, Mass.: Digital Equipment Corporation, 1975.

A hardware manual for the owners and operators of the LA36/LA35 DECwriter II and for those who will be programming the computer to interact with these devices.

RT-11 System Message Manual (DEC-11-ORMEB-A-D), Maynard, Mass.: Digital Equipment Corporation, 1977.

An explanation of system messages that may occur during normal system use; includes required user actions.

[1]Used as an example; consult hardware user or maintenance manuals specific to your system.

CHAPTER 4
USING THE MONITOR COMMAND LANGUAGE

During the bootstrap operation, the RT-11 monitor was copied into computer memory and started. The RT-11 monitor is actually many different components working together to supply basic system functions. For example, part of the monitor is called the resident monitor and provides console terminal service, a system volume device handler, and centrally-required program code to provide a working environment for both system and user programs. The resident monitor is so named because it always remains in computer memory regardless of other system operations that may be occurring. Other parts of the monitor are brought into memory from the system volume as needed. These include the user service routine (USR) which provides support for the RT-11 file system, and the keyboard monitor (KMON) which controls terminal keyboard interaction. From your standpoint, the keyboard monitor is the most visible part of the system software. Among other services, it supplies the monitor command language that you use to control system operations.

ENTERING COMMAND INFORMATION

The monitor command language is a set of English-like command words that you type on the terminal keyboard to initiate and control system operations. There are two general formats that you can use to type a command; one is a long format and the other a short format. The long format causes the system to print prompting messages. These messages ask you to supply specific information, such as file names and device names. The long format is helpful until you become familiar with the commands. You will then probably prefer to use the short format. This format allows you to enter all required information on a single command line; prompts are issued only if you neglect to supply necessary information. Both formats are demonstrated throughout this manual.

You terminate all monitor commands by typing a carriage return. That is, after you type the required command information, you press the carriage return key (represented in this manual by (RET)). This instructs the monitor to initiate the command and to perform the operation.

The monitor prints a period at the left margin of the terminal printer or screen whenever it is waiting for you to type a command. The period is your cue that the system is in monitor command mode and ready to accept a monitor command. Check the output on your terminal printer or screen. You should see at the left margin:

```
RT-11SJ     V03-xx
```

RT-11SJ identifies the RT-11 monitor called the single-job (SJ) monitor. Following this is the version (and update) number of the system in use, in this case, Version 3. The period on the next line indicates that the system is in monitor command mode and is waiting for you to type a monitor command.

General Command Format

Whenever you issue a monitor command, you must supply certain information needed to guide command processing. This information includes the following (square brackets indicate optional qualifiers and characters):

COMMAND[/option]	First you indicate, by command, which system operation you want initiated. Command options are available to allow you to alter the normal (default) operation.
INPUT[/option]	You next indicate, by device and file name, input information that is to be used during the operation. The system volume serves as the default input device. You must explicitly indicate other volumes that you want used for input, and you must usually indicate the file names and file types of the input files. Input file options are available to allow you to alter assumed (default) input operations.
OUTPUT[/option][1]	Finally you indicate, by device and file name, output information that is to be created as a result of the operation. The system volume serves as the

[1]OUTPUT[/option] is not always used; sometimes output must be specified as COMMAND[/option] INPUT/OUTPUT:filespec.

default output device. You must explicitly indicate other volumes that you want used for output, and you must usually indicate the file names and file types of the output files to be created. Output file options are available to allow you to alter assumed (default) output operations.

As mentioned earlier, there are two ways you can type this command information on the terminal keyboard; both formats are illustrated below:

Long Command Format (system prompts for specific information)

```
    .COMMAND[/option] (RET)
    INPUT PROMPT? INPUT[/option] (RET)
    OUTPUT PROMPT? OUTPUT[/option] (RET)
```

Short Command Format (no prompts)

```
.COMMAND[/option] INPUT[/option] OUTPUT[/option] (RET)
```

Notice that you use a slash (/) character to separate an option from the portion of the command that it qualifies, and a carriage return (RET) to terminate each individual command line. When you have supplied all the necessary information, the carriage return signals the monitor to execute the command. You may use whichever format you wish. Both command formats are demonstrated throughout the manual.

Control Commands

In addition to monitor commands, there are several special function commands, called control commands, that you type by first pressing the CTRL key on the terminal keyboard, and then (while holding it down) typing the appropriate letter key of the command. These commands require no terminator; the system performs the function as soon as you type the command.

Control commands are special function commands used to correct typing errors, to interrupt program execution, to inhibit terminal output, and other similar special system operations. They are described in the manual as you need to use them.

Recreating the Examples

During the course of this chapter, and throughout the remainder of the manual, you will use a number of monitor commands to perform some common system operations. For example, you will list the directories of device volumes, copy files between devices, create files, and execute system and user programs. You perform these operations by recreating on the terminal keyboard the examples already provided for you.

You should first read the entire explanation of a command to be aware of its format, the operation it performs, and the options that are available. Then type the command on the terminal keyboard exactly as you see it used. Characters that you type appear in the demonstrations in red print. Characters that are system responses are shown in black print.

Table 4-1 lists symbols that you will see used throughout the demonstrations. These symbols represent various keys on the terminal keyboard. When you see one of these symbols in a command line, type the appropriate key on the keyboard.

Table 4-1 Keyboard Symbols

Symbol	Type
(RET)	carriage return key
(LF)	line feed key
(SP)	space bar (once for each time the symbol is shown). Assume that you should type a single space unless you are otherwise instructed; the space symbol is used only if there is doubt as to the number of spaces to type.
(TAB)	TAB key (once for each time the symbol is shown)
(DEL)	DELETE (RUBOUT) key (once for each time the symbol is shown)
(ESC)	ESCAPE (ALTMODE) key (once for each time the symbol is shown)
(CTRL/x)	CTRL key (hold down CTRL key while typing the letter character (x))

CORRECTING TYPING MISTAKES

All commands that you give the system are typed on the terminal keyboard. If you make a mistake while typing a command, there are two easy ways that you can correct it.

One way to correct a typing error is to use the DELETE key on the keyboard. Pressing the DELETE key once cancels the character just typed; pressing it a second time cancels the next to last character typed, and so on, from right to left, until the beginning of the current line is reached. Then additional DELETEs are ignored.

CTRL/U

The second way to correct a typing error is to use a special control command, CTRL/U. Typing this command once is equivalent to typing as many DELETEs as are needed to cancel every character in the current line.

Type the following characters on the keyboard – the letters DABE, followed by two DELETEs, followed by the letters TE – and notice the system's response:

```
.DABE (DEL) (DEL) TE
```

The monitor echoes each deleted character and encloses them within backslashes. As far as the monitor is concerned, the only characters you have typed are DATE.

```
.DABE\EB\TE
```

Thus, your current line is DATE. Continue by typing a CTRL/U. Remember to first press the CTRL key and then type the U key while holding the CTRL key down; no carriage return is necessary.

```
(CTRL/U)
```

Notice that CTRL/U echoes on the terminal printer or screen as ^U.

```
.DABE\EB\TE^U
```

All characters on the line are effectively cancelled and the character pointer is moved to the beginning of a new line so that you can enter another command. You are still in monitor command mode even though no prompting period appears at the left margin.

Once the carriage return or line feed key is pressed, the previous line cannot be corrected via DELETE or CTRL/U.

These two methods are commonly used to correct typing errors made at the keyboard. You can choose whichever method seems most convenient.

INITIAL MONITOR COMMAND OPERATIONS

The kinds of command operations that you usually perform immediately after the monitor is bootstrapped are those that set up initial conditions such as the current date and time of day, and those that initialize and prepare the system for future operations such as file transfers. If your system has VT11 display hardware and you decide that you want to use it, you should also enable (turn on) the graphics display screen.

Using VT11 Display Hardware

Display hardware on an RT-11 computer system consists of a cathode ray tube that allows programs to use graphics displays. If your system has display hardware[1], which is illustrated in Figure 4-1, you can use the graphics screen in place of the terminal printer or screen if you wish.

NOTE

> Check question 5 in the Hardware Configuration section of Chapter 2 to determine if your system has display hardware. If you do not have display hardware, go on to the next section in this chapter, Entering the Date and Time-of-Day.

GT

The monitor command that enables the graphics screen is the GT command. The GT command is used to change the condition of the graphics display. In this case, you will use it to activate the graphics display hardware so that the VT11 display screen replaces the console terminal printer or screen as the terminal output device.

[1]Video terminal screens are not considered graphics display hardware.

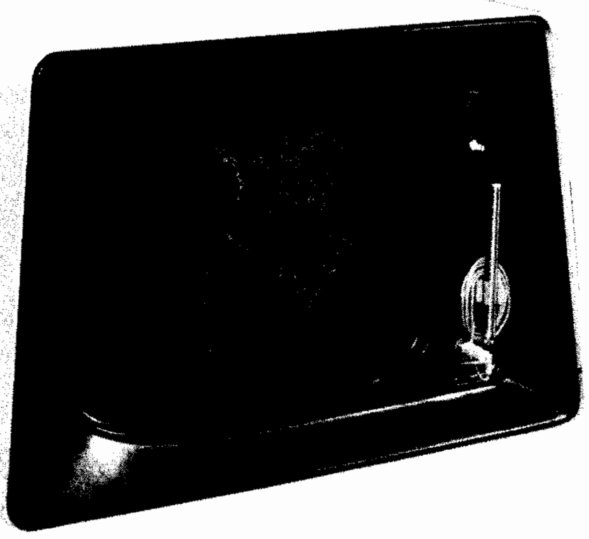


Figure 4-1 VT11 Display Hardware

Type the following on your terminal keyboard (if necessary, refer to Table 4-1 to review the special symbols):

Long and Short Command Format

```
.GT ON (RET)
```

If your system does not have display hardware, the monitor prints a message[1] on the terminal printer or screen informing you that the command is illegal for your system configuration:

```
?KMON-F-Illegal command
```

Otherwise, the command is accepted and you should notice that all character echoing and system responses now appear on the graphics screen instead of the terminal printer or screen. A period appears there, indicating that the system is waiting for another command. The character pointer is visible as a blinking rectangular- or L-shaped cursor situated after the period.

[1]The meanings of all system messages are listed in the RT-11 System Message Manual.

Like the terminal screen, output that appears on the graphics screen is temporary. Once the screen is filled, lines are rolled off the top and are lost to view. However, if your terminal has a printer, a special control command allows you to control console terminal output so that it appears on both the graphics screen and the terminal printer simultaneously. In this manner, you can direct selected portions of terminal output, directory listings for example, to be both displayed and printed at the same time. The advantage of this is that the display copy is eventually lost but you create a printed copy for later use.

The control command that provides this function is CTRL/E. It is initiated by holding the CTRL key down while typing the E key. No carriage return is necessary. When you type this command, no characters echo on the graphics screen, but you should notice that all subsequent characters (both input and output) appear on both the graphics screen and the terminal printer.

CTRL/E

Thus, if your terminal has a printer and you wish to use the printer in addition to your VT11 graphics screen, type once:

```
CTRL/E        (Remember, this command does not echo.)
```

Now type the following and notice where the characters echo:

```
.WRONG COMMAND CTRL/U
```

To disable the printer at any time so that character echoing occurs only on the graphics screen, type another CTRL/E command:

```
CTRL/E
```

Finally, to return terminal output control to the terminal, disabling the graphics screen, use the GT OFF command; this changes the terminal device handler back to its original output setting:

Long and Short Command Format

```
.GT OFF RET
```

Decide now whether to use the graphics screen for the remaining demonstrations. If so, use the GT ON command to enable the graphics screen, and remember that the CTRL/E command is available when you wish to produce simultaneous output.

Entering the Date and Time-of-Day

Entering the current date and time-of-day are record-keeping system operations; they help you later identify when other system operations were performed.

For example, by entering the current date you instruct the system to assign this date to all files you create. The date will also appear in volume directories and listings produced by the various language processors and utility programs. If your system has a clock, by specifying the current time-of-day you instruct the system to keep track of time based on the time you set. The current time is printed on listings when they are produced, and may also be used to control certain program operations.

DATE

Enter the date by typing the monitor DATE command as follows (there is only one format):

Long and Short Command Format

```
.DATE 13-JUN-77 (RET)
```

This sets the date to June 13, 1977. Since this date is not current, reenter the correct date using the same command format:

```
.DATE dd-mmm-yy (RET)
```

Typing the new date overrides the previous date entered.

TIME

The monitor TIME command is used to set the time-of-day. Time must be specified in 24-hour notation. The system then keeps track of time in hours, minutes, and seconds, based on the initial time that you enter in the command. Enter the time as follows (there is only one format):

Long and Short Command Format

```
.TIME 15:01:00 (RET)
```

If your system does not have a clock, the monitor prints a message on the terminal informing you that the command is not valid for your system configuration:

```
?KMON-F-no clock
```

Otherwise, the time is set to 3:01 p.m. If your system has a clock, reenter the correct time, using the same command format:

```
.TIME hh:mm:ss (RET)
```

Typing the new time overrides the previous time entered.

To check the time and date at any time while you are using the system, simply type either the DATE command or the TIME command, followed by a carriage return only:

Long and Short Command Format

```
.DATE (RET)
13-Jun-77

.TIME (RET)
15:06:19
```

The system responds by printing the date or the time based on the information you previously entered.

Setting the time is temporary. If you want it to be kept current, you must reenter it whenever you bootstrap the system.

Assigning Logical Names to Devices

Each hardware device in the RT-11 system is identified by a 2-character code name. These names, listed in Table 4-2, are defined in the system software and are recognized and used by the operating system. These are the device names that you generally use in command input and output lines. However, you may want to temporarily change any of these device names for a variety of reasons. The following paragraphs describe both using the physical device names shown in Table 4-2 and assigning logical (temporary) device names to devices.

Two additional logical device names are used. These special names are described in Table 4-3.

You use device names in the input and output portions of a command line to identify where input information can be found and

Table 4-2 Physical Device Names

Code	Device
CR:	Card Reader
CTn:	Cassette
DMn:	RK06 Disk
DPn:	RP02/03 Disk
DSn:	RJS03/4 Disk
DTn:	DECtape
DXn:	RX01 Diskette
LP:	Line Printer
MMn:	TJU16 Magtape
MTn:	TM11 Magtape
PC:	Paper Tape Punch/Reader
RF:	RF11 Disk
RKn:	RK11 Disk
TT:	Console Terminal Keyboard/Printer

Table 4-3 Special Logical Device Names

Code	Device
SY:	The volume from which the monitor was bootstrapped; i.e., the system volume.
DK:	The default storage volume (initially the same as SY:; i.e., the system volume).

where output information will be sent. If a file is involved, you also include its file name and file type in the following format:

```
devicename:filename.filetype
```

The device name is followed by a colon, and is always separated from any file name and file type by a colon. The device name is generally one of the codes listed in Tables 4-2 and 4-3. When you use a device name in any command, you must also include the device unit number (represented by the letter 'n' in Table 4-2) unless the number is 0. The system assumes unit 0 of the device if no unit number is given. Thus, diskette unit 0 is DX: or DX0:; diskette unit 1 is DX1:; RK: disk unit 2 is RK2:; and so on. Note from Table 4-3, that you can use the device codes SY: or DK: for your system volume in addition to its standard device name. However, since the system volume is initially the default storage volume for all operations, you do not need to use a device name for your system volume.

The names listed in Tables 4-2 and 4-3 are the device names defined within the system software. However, you can temporarily change any of these name assignments, either by reassigning existing names to different devices, or by assigning new logical names of your own choosing to devices.

There are many reasons why you might want to temporarily change a device name and assign it a logical name. You may, for example, have a program that is written for a specific device. If that particular device is not available on your system, you need only assign its name to a device that is available. The program then uses the new device instead.[1]

Since not all RT-11 users have access to the same kind of storage volume, you are instructed to assign the logical name VOL: to whatever volume you are using for storage. After you make this assignment, subsequent command lines can be the same for everyone using this manual.

Similarly, the special logical device name DK:, presently assigned to your system volume, could be assigned to any kind of storage volume. Not only would DK: signify your storage volume regardless of its physical device name, but you could also avoid typing DK: since it is the default storage volume for most commands (only the R command requires that the file specified must be on the system volume SY:).

To assign a logical name to your storage volume, first determine its physical device name. Check questions 4a and 4b in the Hardware Configuration section of Chapter 2 to see which device and which device unit you are using for your storage volume. Translate this into the appropriate name and number using Table 4-1 as a guide.

ASSIGN

Use the monitor ASSIGN command to change this physical name to a logical name. Substitute for physical-device-name in the command

[1]This is called device independence.

lines below the physical name and device unit number for your storage volume (for example, for RK05 disk unit 1, substitute RK1):

Long Command Format

```
.ASSIGN (RET)
Physical device name? physical-device-name (RET)
Logical  device name? VOL: (RET)
```

Short Command Format

```
.ASSIGN physical-device-name VOL: (RET)
```

Once the assignment is made, the system recognizes the logical name VOL: as the device name for your storage volume. This is the only logical assignment you need to make. Since you are not changing the DK: assignment, the system volume remains the default device for all I/O operations.

SHOW

As you continue to use the system, you may well make many device assignments and deassignments. To check the status of all assignments made during a computer session, you can use the monitor SHOW command to print on your terminal a list of all the logical assignments currently in effect. For example, use the SHOW command now to check the status of the assignment just made:

Long and Short Command Format

```
.SHOW DEVICES (RET)
```

Check the list printed on your terminal to make sure that the code VOL: has been assigned to your storage volume. The letters VOL: should follow the appropriate device name in the list, similar to the response shown below in which VOL: represents disk unit 1:

```
TT
RK   (RESIDENT)
  RK0     = SY , DK
  RK1     = VOL
DS
DM
RF
DP
DX
DT
MT
CT
LP
BA
NL
<FREE>

USR SWAP
```

Logical device assignments are temporary. Thus, if you want a particular device assignment to always remain in effect, you must reassign it each time the system is bootstrapped.

Listing Volume Directories

DIRECTORY

Both your system volume and your storage volume have directories, which are a compiled list of all the files stored on the volume. You can print a volume directory on your terminal, using the monitor DIRECTORY command.[1] For example, to list the directory of your system volume, type:

Long and Short Command Format

```
.DIRECTORY (RET)
```

(The system volume is the default device.)

CTRL/O

Since the directory of the system volume may be quite long, after approximately 10 lines have printed on the terminal, type:

```
(CTRL/O)
```

This special control command echoes as ^O and inhibits the remainder of the listing output from printing on the terminal. When control returns to monitor command mode, look at the directory

[1]Users of VT11 display hardware may wish to use the CTRL/E command to enable both the graphics screen and the terminal printer for the following exercises.

listing. At the top of the listing is today's date, as you entered it earlier in the DATE command. Following the date is a list of the files on the volume. Notice the 2-column format of each line in the directory:

```
 13-Jun-77
RKMNSJ.SYS    86 02-Jun-77       RKMNFB.SYS    97 02-Jun-77
RKMNXM.SYS   106 02-Jun-77       RKMNSJ.BL     83 02-Jun-77
DMMNSJ.SYS    88 02-Jun-77       DMMNFB.SYS    98 02-Jun-77
DMMNXM.SYS   108 02-Jun-77       DXMNSJ.SYS    86 02-Jun-77
DXMNFB.SYS    97 02-Jun-77       DXMNXM.SYS   107 02-Jun-77
DXMNSJ.BL     83 02-Jun-77       DTMNSJ.SYS    86 02-Jun-77
DTMNFB.SYS    97 02-^O

 137 Files, 3857 Blocks
 905 Free blocks
```

First the file name appears, followed by a dot and a file type that is frequently used to identify the file's format. For example, SYS represents a system file; other RT-11 file types used to represent different kinds of files are listed in Table 4-4. After the file type is a number that indicates the size of the file. The size is given in blocks, a term used to designate a standard amount of information. A file that is 1 to 10 blocks long is fairly small, while a file over 100 blocks in length is quite large. The date on which the file was created is shown at the right. This space is empty if a date was not specified (via the DATE command) on the day the file was created. If you let

Table 4-4 File Types

File Type	Meaning
.BAC	BASIC compiled file
.BAK	Editor backup file
.BAS	BASIC source file
.BAT	BATCH source file
.COM	Indirect command file
.CTL	BATCH control file
.DAT	BASIC or FORTRAN data file
.DBL	DIBOL source file
.DIR	Directory listing file
.FOR	FORTRAN source file
.LST	Listing file
.MAC	MACRO source file
.MAP	Linker map file
.OBJ	MACRO, FORTRAN, or DIBOL object output file or library file
.REL	Executable foreground program file
.SAV	Executable background program file
.SML	System MACRO library
.SYS	System files and handlers

the directory list to completion, at the end you are told how many files are on the volume, their total length, and the number of free blocks available for your use.

DIRECTORY/BRIEF

CTRL/C CTRL/C

You can also obtain an abbreviated directory, which omits file lengths and dates and lists only file names and file types in 5-column format. To do this, you use the DIRECTORY command with its /BRIEF option. Type the following, and after several lines have listed, interrupt the directory by typing two CTRL/C command characters. This double control command echoes two ^Cs, requesting the running program to abort immediately, independent of what the program is doing (one CTRL/C aborts an executing program waiting for input from the console terminal). Control returns to monitor command mode.

Long and Short Command Formats

```
.DIRECTORY/BRIEF VOL:*.* (RET)
 13-Jun-77
RKMNSJ.SYS    RKMNFB.SYS    RKMNXM.SYS    RKMNSJ.BL     DMMNSJ.SYS
DMMNFB.SYS    DMMNXM.SYS    DXMNSJ.SYS    DXMNFB.SYS    DXMNXM.SYS
DXMNSJ.BL     DTMNSJ.SYS    DTMNFB.SYS    DTMNSJ.BL     DSMNSJ.SYS
DSMNFB.SYS    DSMNXM.SYS    DPMNSJ.SYS    DPMNFB.SYS    DPMNXM.SYS

(CTRL/C) (CTRL/C)
```

DIRECTORY /PRINTER

Volume directories can also be printed on a line printer if one is available on your system. Check question 5 in the Hardware Configuration section of Chapter 2 to determine if your system has a line printer. Since listings print faster on a line printer than on the console terminal, it is to your advantage to use the line printer for large amounts of output. The /PRINTER option is used with the DIRECTORY command to cause a directory to be printed on the line printer instead of the terminal. Make sure your line printer is turned on, and then type the DIRECTORY command as shown (users who do not have a line printer can ignore this command):

Long and Short Command Format

```
.DIRECTORY/PRINTER (RET)
```

The entire listing may be quite long. When the line printer is done printing, retrieve the listing.

Initializing the Storage Volume

Initializing a storage volume completely clears its directory. A new (unused) volume should always be initialized before it is first used. In addition, any storage volume that contains files that are no longer needed can be initialized to recover the storage space. Note, however, that the effect of an initialize operation is to remove all file names from the directory. So before you initialize any volume, be sure that there are no files on it that you might later want.

Since you will use your storage volume to store several new files (created as a result of the various exercises in this manual), clear its directory using the monitor INITIALIZE command. This ensures that there is room on the volume for new files.

Long Command Format

```
.INITIALIZE (RET)
Device? VOL: (RET)
VOL:/Init are you sure?Y (RET)
```

(VOL: is the assigned logical device name for your storage volume.)

Short Command Format

```
.INITIALIZE VOL: (RET)
VOL:/Init are you sure?Y (RET)
```

The system prompt physical-device-name/Init are you sure? is always printed to provide an opportunity for you to verify the command. Typing a Y initiates the operation, while N aborts (ignores) the operation and returns control to monitor command mode. Check your command line, make sure you are initializing your storage volume, and then type a Y. Again list the directory of the storage volume. It should be empty.

Long and Short Command Formats

```
.DIRECTORY VOL: (RET)
 13-Jun-77

 0 Files, 0 Blocks
 4762 Free blocks
```

The number of blocks available for use on the volume is printed at the end of the directory and varies depending on the type of device you use as your storage volume.

The commands you have performed in this chapter have prepared the system for major operations that will follow. In Chapter 5 you begin by using the RT-11 editor to create text files that you will store on your initialized storage volume.

SUMMARY: INITIAL MONITOR COMMANDS

ASSIGN physical-device-name logical-device-name
: Assign a logical device name to a physical device name.

DATE
: Print the current date, if previously set.

DATE dd-mmm-yy
: Set the current date (day-month-year).

DIRECTORY dn:
: List the volume directory on the terminal (dn: is the code for the device name; the default storage volume (DK:) is assumed if dn: is not specified).

DEASSIGN
: Remove logical device assignments.

DIRECTORY/BRIEF dn:
: List a brief volume directory on the terminal, showing only file names.

DIRECTORY/PRINTER dn:
: List the volume directory on the line printer.

DIRECTORY/PRINTER/BRIEF dn:
: List a brief volume directory on the line printer.

INITIALIZE dn:
: Clear the directory of the indicated volume (dn: is the code for the device name and must be specified).

GT OFF
: Disable the VT11 display hardware.

GT ON
: Enable the VT11 display hardware so that the graphics screen replaces the terminal printer/screen as the terminal output device.

SHOW DEVICES
: Print the status of all current logical device name assignments.

TIME
: Print the current time, if previously set.

TIME hh:mm:ss
    Set the current time-of-day (hour:minute:second).

SUMMARY: SPECIAL CONTROL COMMANDS

CTRL/C CTRL/C
    Interrupt the current operation or program and return control to monitor command mode.

CTRL/E
    Direct terminal output to both the graphics screen and the terminal printer simultaneously. Type a second CTRL/E to return output control to only the graphics screen. (Valid only when VT11 display hardware is enabled.)

CTRL/O
    Inhibit the remainder of output from printing on the terminal.

CTRL/U
    Cancel every character in the current line.

DELETE
    Cancel the last character typed on the current line.

REFERENCES

LP11/LS11 Line Printer Manual (EK-LP11-TM-005). Maynard, Mass.: Digital Equipment Corporation, 1975.

A hardware manual for the owners and operators of LP11/LS11 line printers and for those who will be programming computers to interact with this device.

RT-11 Pocket Guide (DEC-11-ORRCB-A-D). Maynard, Mass.: Digital Equipment Corporation, 1977.

A summary of all RT-11 monitor commands and command options, and system utility program operating commands.

RT-11 System User's Guide (DEC-11-ORGDA-A-D). Maynard, Mass.: Digital Equipment Corporation, 1977.

A guide to the use of the RT-11 operating system. See Chapters 3 and 4.

CHAPTER 5
CREATING AND EDITING TEXT FILES

The ability to create and edit text files is one of the most useful features of the RT-11 operating system. Not only can you create computer programs, data files, memos, and reports on-line (i.e., under the control of the system), but you can alter them by adding or removing text without retyping the entire file.

You create and edit text files more often than you perform any other system operation. Therefore it is essential that you become familiar with the editing process as quickly as possible. Editing should become second nature to you as you learn to use the RT-11 computer system.

THE RT-11 EDITOR

The RT-11 editor is a system utility program called EDIT.SAV, which is stored as part of the RT-11 operating system on your system volume. Text files that you create with the editor are stored in the computer in ASCII format. ASCII, which stands for the American Standard Code for Information Interchange, is an industry-standard code that consists of a numeric representation for each of the alphabetic characters (A to Z), the numeric characters (0 to 9), the punctuation characters, and some special communication control characters. When you type text on the terminal keyboard, the system automatically converts the text to the appropriate ASCII codes; when you request listings on the terminal or line printer, the system converts the ASCII code back to the appropriate text characters.

The RT-11 editor uses a specially reserved area of computer memory to hold the text you are creating or editing. This area of memory is called the text buffer. When you create text, the characters that you type on the terminal keyboard are transmitted directly into the text buffer. When you edit already existing text, the characters are copied from the input file into the text buffer. Once in the text buffer, the characters are available for modification. When you have edited the text in the buffer to your satisfaction, the characters are moved out of the text buffer to the output file (Figure 5-1).

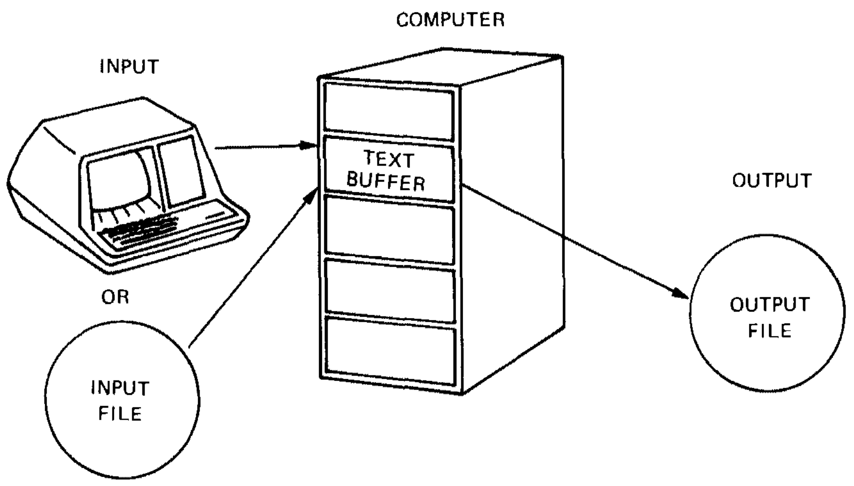


Figure 5-1 Editing with RT-11

Since the text buffer is a finite area of computer memory, you may at times try to input more text than the buffer can accommodate. If this condition becomes apparent to the editor, it prints a warning message on the terminal telling you that before you can input any more text, you must make room in the buffer, either by transferring text to the output file or by erasing text already in the buffer.

You can avoid this inconvenience during editing if you make use of a concept called paging. When you create a large text file, instead of typing the file as one long stream of text, divide it into individual pages of approximately 50-60 lines in length; this corresponds roughly to the size of a line printer of terminal listing page. You can then copy the text into and out of the buffer one page at a time. A single page of text is never too large for the text buffer and also fits nicely on the line printer or terminal perforated paper when you obtain a listing.

CREATING A TEXT FILE

EDIT/CREATE

You activate the editing capability by using the monitor EDIT command. When creating a file, you must use the /CREATE option followed by the file name and file type you want assigned to the new file. The default storage volume (DK:) serves as the default device, so unless you specify a device using one of the codes in Table 4-2, the editor creates the new file on the device DK: (which is the system volume, unless changed via ASSIGN).

First, if you are using display hardware, disable it with the monitor GT OFF command; the editor has a special display capability that is not described until later in the chapter:

Long and Short Command Format

```
    .GT OFF (RET)
```

Next, use the editor to create a short text file of only five lines. Call the file DECIND.USA and use the default storage volume (currently the same as the system volume) for the file.

Long and Short Command Format

```
    .EDIT/CREATE DECIND.USA (RET)
    *
```

Once the output file is open (that is, when the appropriate file has been established for output operations), the editor prints a prompting asterisk at the left margin. The asterisk indicates that control is in editing command mode and is your cue to enter an editing command.

INSERT

The editing command used to create text is the I (Insert) command. Type:

```
    *I
```

ESCAPE ESCAPE

All subsequent characters that you type on the terminal keyboard will now be entered into the text buffer just as you type them. Enter the following text exactly as shown, including all spaces and errors. Before you type the RETURN key, check the line to make sure that it matches what is shown here. Remember, if you make a typing mistake that is not intentional, you can use the DELETE key on the terminal keyboard to erase individual characters and the CTRL/U command to erase all characters on the current line. When you finish typing the five lines, type the ESCAPE (ALTMODE) key twice. The ESCAPE key echoes on the terminal as a $ and is used to execute an editing command and to return control to editing command mode.

```
*IWE HOLD THESE TRUTS TO BEE SELF-EVIDENT,(RET)
THAT ALL MEN ARE CREATED EQUAL, THAT THEY (RET)
HAVE UNRELIABLE TENDENCIES OF WHICH THEY (RET)
AR ENDOWED BY THEIR CREATOR, THAT AMONG (RET)
THESE ARE LIFE, LIBERTY AND HAPLENESS.(RET)
(ESC)(ESC)
*
```

EXIT

Forget for the moment that this text contains several misspellings and other errors, and assume instead that you are satisfied with it and ready to transfer it from the text buffer to the output file. The EX (Exit) editing command performs this function. This command terminates editing, transfers all text currently in the text buffer to the output file, closes the currently open output file (making it unavailable for further output operations), and returns control to monitor command mode, indicated by a dot at the left margin. Use the EX command to close the file DECIND.USA:

```
*EX(ESC)(ESC)
```

You now have a file on your system volume called DECIND.USA, consisting of the five lines of text you just created.

EDITING A TEXT FILE

EDIT

The file DECIND.USA needs editing. To edit an existing file, you again use the EDIT command to activate the editor. Next indicate in the command line the two-character device code for the volume on which the file resides (the default storage volume, DK:, is assumed). Following this, you indicate the file name and file type of the file. The editor then opens the file, making it available for input operations.

Thus, to open the file DECIND.USA for editing, type:

```
.EDIT DECIND.USA(RET)
*
```

READ

The EDIT command opens the input (and output) files. Use the R (Read) editing command to read the first page of text from the input file into the text buffer. No output occurs to the output file, but the file is available for output at a later time. The input file itself is not altered in any way.

```
R(ESC)(ESC)
*
```

BEGINNING

Whenever text is read into the text buffer, a pointer is automatically positioned at the beginning of the text. This pointer is an invisible indicator that serves as a target for editing commands. The pointer pinpoints the exact location in the file where the next character will be inserted. For example, when you finished inserting text earlier (just prior to using the EX command), the pointer was positioned at the end of the file. Now that the EDIT command has been used to read text into the text buffer, the pointer is positioned at the beginning of the text in the text buffer. If the pointer is not at the beginning and you want to move it there, you can use the B (Beginning) command; this command moves the pointer to the beginning of the text in the text buffer, no matter where the pointer is currently positioned:

```
B(ESC)(ESC)
*
```

LIST

With the pointer positioned at the beginning of the text buffer, you can use the L (List) editing command to list the text currently in the text buffer on your terminal printer. The List command lists text, starting at the pointer and continuing to whatever place you indicate by the command argument.

A command argument is simply a prefix to an editing command that sets limits on the command's actions. Command arguments are used frequently and are summarized in Table 5-1. Study this table for a moment before continuing.

Table 5-1 Command Arguments

Argument	Meaning
n	n represents any integer in the range –16383 to +16383; n may be preceded by a + or –. If no sign precedes n, it is assumed to be positive. Whenever an argument is acceptable in a command, its absence implies an argument of 1 (or –1 if only the – is present).
0	0 refers to the beginning of the current line.
/	/ refers to the end of text currently in the text buffer.

Thus, with the pointer positioned at the beginning of the text, use the / argument and the L command to list on the terminal all text in the buffer. The position of the pointer does not change. List the text and compare your output with the five lines shown on the following page; they should match exactly:

```
*/L(ESC)(ESC)
WE HOLD THESE TRUTS TO BEE SELF-EVIDENT,
THAT ALL MEN ARE CREATED EQUAL, THAT THEY
HAVE UNRELIABLE TENDENCIES OF WHICH THEY
AR ENDOWED BY THEIR CREATOR, THAT AMONG
THESE ARE LIFE, LIBERTY AND HAPLENESS.
*
```

If your output and the five lines above do not match exactly, then you probably typed some unintentional errors into DECIND.USA.

Unfortunately, the remaining EDIT commands in this exercise depend upon an exact reproduction of DECIND.USA to function properly. Therefore, since you are not yet familiar with the EDIT commands necessary to correct your file, an existing copy of DECIND.USA with intentional errors must be substituted.

Prepare the text buffer by erasing it with CTRL/C (ESC) (ESC). This unusual command combination is required by the EDIT program to exit without creating an output file. The structure of the command prevents you from accidentally eliminating a file with a single CTRL/C.

```
*(CTRL/C) (ESC) (ESC)
```

The monitor command mode period appears, signalling your departure from the editing command mode. Your system volume still contains the file DECIND.USA that you created earlier. However, it also contains the copy provided with the system, DEMOED.TXT, that you will use for the remainder of the exercise.

Before going any further, you must rename DEMOED.TXT to DECIND.USA to avoid confusion. A RENAME operation, explained fully in the FILE COPYING OPERATIONS section of Chapter 7, is the method of choice. Type RENAME DEMOED.TXT DECIND.USA (RET) .

```
.RENAME DEMOED.TXT DECIND.USA (RET)
```

The contents of DEMOED.TXT are now labelled DECIND.USA. Type EDIT DECIND.USA (RET) to open the file for input and the R command to read it into the text buffer.

```
.EDIT DECIND.USA (RET)
*R(ESC)(ESC)
```

Since the pointer automatically returns to the text's beginning with an R command, you can type /L to list the entire file.

```
*/L(ESC)(ESC)
WE HOLD THESE TRUTS TO BEE SELF-EVIDENT,
THAT ALL MEN ARE CREATED EQUAL, THAT THEY
HAVE UNRELIABLE TENDENCIES OF WHICH THEY
AR ENDOWED BY THEIR CREATOR, THAT AMONG
THESE ARE LIFE, LIBERTY AND HAPLENESS.
*
```

JUMP

The text contains errors and misspellings necessary for this chapter's proper functioning. To correct the errors, reposition the pointer so that it is near the text you wish to change. The J (Jump) command, for instance, in conjunction with a command argument, moves the pointer either backward or forward by the specified number of characters, including spaces. Type the J command now, using an argument of 18, to reposition the pointer ahead 18 places[1]:

```
*18J(ESC)(ESC)
*
```

Although you cannot see it, the pointer has moved from the beginning of the text buffer to the right of the 18th character. You can verify this using the List command again. The List command with no argument prints from the pointer to the end of the current line and thus exposes the location of the pointer:

```
*L(ESC)(ESC)
S TO BEE SELF-EVIDENT,
*
```

The characters above should match the current line on your terminal, showing the pointer positioned at the first error in the text where an H is missing in the word TRUTS. Since the pointer is positioned between the second T and the S, use the Insert command to insert an H in the proper place:

```
*IH(ESC)(ESC)
*
```

[1]Anytime you use the Jump command to move the pointer forward (or backward) by enough characters so that it moves to a new line, you must account for two extra characters in the command argument. This is because the editor treats the carriage return at the end of each line as two characters – a return and a line feed.

VERIFY

Now use the V (Verify) command to verify the line. The V command, which does not require arguments, prints the entire line containing the pointer (the current line) on the terminal. It allows you to verify that a correction was properly made. The pointer is not moved as a result of the V command; its position remains just to the right of the last inserted character (shown here by the arrow):

```
*V(ESC)(ESC)
WE HOLD THESE TRUTHS TO BEE SELF-EVIDENT,
*                         ↑
```

So far you have entered and executed editing commands one at a time. You can enter multiple commands by separating each individual command with a single ESCAPE. Typing two ESCAPEs then executes all the commands in the entire command string in consecutive order. For example, combine the J and L commands as shown in the following command string:

```
*7J(ESC)L(ESC)(ESC)
E SELF-EVIDENT,
*
```

7J moves the pointer seven positions to the right and L then lists from the pointer to the end of the line so that you can see the pointer's new position.

CTRL/X

A special CTRL command is available to erase multiple editing commands. The CTRL/X command (hold the CTRL key down and type the X key) causes the editor to ignore an entire command string that might extend over several lines if the I command is involved. The editor echoes with ^X, issues a carriage return, and prints an asterisk indicating that you are still in editing command mode and can enter a new command. For example, type:

```
*70J(ESC)ISTART A (RET)
NEW LINE (CTRL/X)
*
```

In addition to the CTRL/X command, you may still use the DELETE key to erase individual characters in the command line one at a time, and the CTRL/U command to erase all characters entered on the current command line.

DELETE

Since you used the CTRL/X to ignore this last command string, the pointer is still positioned at the next error in the file – just before the extra E in the word BEE. You can erase this extra character by using the D (Delete) command[1]. The D command removes one character (or space) to the right of the pointer for every +1 in its argument and one character to the left for every –1. Use the D command to erase the extra E and then verify the line (+1 is assumed if no a[illegible]nt is used):

```
*D(ESC)V(ESC)(ESC)
WE HOLD THESE TRUTHS TO BE SELF-EVIDENT,
*                          ↑
```

As you can see from the position of the pointer above (shown by the arrow), the D command does not actually move the pointer, but simply erases characters around the pointer. Since the extra E was erased, the pointer is now positioned between the E and the space.

ADVANCE

Just as the Jump command moves the pointer by characters, you can use the A (Advance) command to move the pointer by entire lines. Again you give the command an argument which indicates the number of lines, either forward or backward. The pointer is positioned at the beginning of the new line. Use the A command to move the pointer forward two lines, and then list the current line:

```
*2A(ESC)L(ESC)(ESC)
HAVE UNRELIABLE TENDENCIES OF WHICH THEY
*
```

KILL

This entire line does not belong in the text. To erase it, you could count the number of characters in the line and use this number as an argument to the D command; however, there is an easier way. The K (Kill) command erases the entire line following the pointer and positions the pointer at the beginning of the next line in the text. Type:

```
*K(ESC)L(ESC)(ESC)
AR ENDOWED BY THEIR CREATOR, THAT AMONG
*
```

[1]The Delete command should not be confused with the DELETE key on the terminal keyboard. While both perform the delete function, the D command is used to erase characters already within a text file; the DELETE key is used to erase freshly-typed characters in a command string or during text creation.

The pointer is now at the beginning of the next line in the text. As you can see, this line also contains an error, the word AR is incorrectly spelled. Use the J command to jump over two characters, insert the E, and then verify the line:

```
*2J(ESC)IE(ESC)V(ESC)(ESC)
ARE ENDOWED BY THEIR CREATOR, THAT AMONG
*  ↑
```

GET

The arrow shows where the pointer is now positioned. This line still contains an error – it is missing some text; the words WITH CERTAIN INALIENABLE RIGHTS should follow the word CREATOR. You can count the number of characters from the pointer to the second R in CREATOR and then jump the pointer by this number, or you can use the G (Get) command. The G command searches, from the pointer, for the first occurrence of a specified character string and leaves the pointer at the end of that string. Use the G command to search for the string OR (in CREATOR); then insert the missing words and list the lines that have changed. Notice how you use the carriage return to break the line into two parts (the (SP) symbol is used to show where you should insert spaces):

```
*GOR(ESC)I(SP)WITH(SP)CERTAIN(RET)
INALIENABLE(SP)RIGHTS(ESC)-A(ESC)2L(ESC)(ESC)
ARE ENDOWED BY THEIR CREATOR WITH CERTAIN
INALIENABLE RIGHTS, THAT AMONG
*
```

To list both lines, it was necessary to move the pointer back to the beginning of the first line you changed; this was done by the –A command. The 2L command then listed both lines. Notice where the pointer is; it was moved by the –A command and was not repositioned by the L command.

You must be careful when you use the Get command. This command searches for the first occurrence of the character string you specify. This character string may be any number of characters but must be unique if you want the pointer to move to the correct spot. For example, if the characters OR had occurred anywhere after the pointer and before the word CREATOR, the pointer would have stopped there instead and you would have inserted text in the wrong place.

The final errors in this text occur in the last line. The words THE PURSUIT OF are missing, and the word HAPLENESS is a misspelling. Use the Get command to move the pointer to the word

AND and insert the missing text. Move the pointer again with the Get command to the "PLE" of HAPLENESS, erase the LE and insert PI. Then verify the line.

```
*GAND(ESC)I(SP)THE(SP)PURSUIT(SP)OF(ESC)(ESC)
*GPLE(ESC)-2D(ESC)IPI(ESC)V(ESC)(ESC)
THESE ARE LIFE, LIBERTY AND THE PURSUIT OF HAPPINESS.
*
```

CTRL/L

Large text files of 50 lines or greater should be delimited into pages. To do this, you insert a form feed into the text at the place where you want the page to end. A form feed is typed as a CTRL/L (hold the CTRL key down and type the L key). Typing a CTRL/L inserts a form feed into the text, which the editor then recognizes as a page break.

Since this text file is only five lines long, there is really no need to delimit it as a page. However, for the sake of practice, insert a form feed at the end of this file. Then move the pointer to the beginning of the text buffer and list the entire text. Compare your text with that shown below. If errors remain in your file, fix them using the commands described so far.

```
*G.(ESC)I(RET)
(CTRL/L)                         (CTRL/L echoes as eight line feeds.)


(ESC)B(ESC)/L(ESC)(ESC)
WE HOLD THESE TRUTHS TO BE SELF-EVIDENT,
THAT ALL MEN ARE CREATED EQUAL, THAT THEY
ARE ENDOWED BY THEIR CREATOR WITH CERTAIN
INALIENABLE RIGHTS, THAT AMONG
THESE ARE LIFE, LIBERTY AND THE PURSUIT OF HAPPINESS.




*
```

This text is correct in spelling and content, but the last two lines should be justified to make reading them easier. The pointer is currently at the beginning of the text. Use the G command to search for the character string AMONG; then insert and delete text to justify the lines. Finally, list the text again:

```
*GAMONG(ESC)I(SP)THESE(SP)ARE(ESC)A(ESC)10D(ESC)B(ESC)/L(ESC)(ESC)
WE HOLD THESE TRUTHS TO BE SELF-EVIDENT,
THAT ALL MEN ARE CREATED EQUAL, THAT THEY
ARE ENDOWED BY THEIR CREATOR WITH CERTAIN
INALIENABLE RIGHTS, THAT AMONG THESE ARE
LIFE, LIBERTY AND THE PURSUIT OF HAPPINESS.




*
```

NEXT

Once you are satisfied with your text, you are ready to transfer it to the output file. You could use the EX command to transfer the text, as you did earlier in the section Creating a Text File. However, suppose your input file has additional pages of text that require editing. If you use the EX command, all remaining text in the input file will be read through the text buffer into the output file and the files closed without giving you a chance to do more editing. To avoid this, you can use the N (Next) command. This command transfers the text currently in the text buffer to the output file, clears the text buffer, and reads in the next page from the input file. The pointer is positioned at the beginning of the text buffer.

```
*N(ESC)(ESC)
?EDIT-F-End of input file        (No text remains in the
*                                 input file.)
```

If you use the N command when no text remains in the input file (as just happened), the editor prints a message on the terminal telling you so. At this point, you can type the EX command to close the file.

```
    *EX(ESC)(ESC)
```

When you close a file after editing, the editor creates a file on the default storage volume (or system volume). It gives this new file the file name and file type that you indicated for input. It then renames the input file so that the file retains its file name but is assigned a file type of .BAK. .BAK identifies it as a backup file, here an original input file retained in case of editing mistakes or accidental deletion of the new file. Thus you now have two versions of the DECIND file on your system volume: DECIND.USA, which is the edited version, and DECIND.BAK, which is the unedited (original) input file. Verify this using the monitor DIRECTORY command:

Long and Short Command Format

```
.DIRECTORY/BRIEF DECIND.*(RET)
 13-Jun-77
DECIND.BAK     1 09-May-77  DECIND.USA  1 13-Jun-77
 2 Files, 2 Blocks
 904 Free blocks
```

The * following DECIND. is a type of shorthand notation called wildcard construction. Here it means to list all files named DECIND regardless of their file type. Wildcard construction is explained in greater detail in the Multiple File Operations section of Chapter 7.

Whenever you edit the same file a number of times, new versions overwrite old versions. Thus only two versions of the edited file (filnam.BAK and filnam.typ) ever reside on a volume at one time.

USING UPPER- AND LOWER-CASE CHARACTERS

Later model terminals (e.g., LA36 DECwriters and VT52 DECSCOPE terminals) have the capability to print in upper- and lower-case. Certain line printers also have this capability. You can use the upper-/lower-case capability of these devices if you type the EL (Edit Lower) editing command before entering the text you want to insert in lower-case. The EL command instructs the system to accept all characters typed as they appear on the keyboard. The monitor facility which converts all alphabetic characters to upper case is disabled. In addition, the characters are echoed on the terminal printer or screen as upper- and lower-case characters.

Edit Lower

Open the file DECIND.USA again and type the EL command:

Long and Short Command Format

```
.EDIT DECIND.USA(RET)
*EL(ESC)(ESC)
*
```

Once you have typed the EL command, you can use the SHIFT key on the terminal to designate upper-case, just as you do on a typewriter. Editing commands may be entered as either upper- or lower-case characters. For example, type the following commands, which change the characters in the first line of the file DECIND.USA to upper- and lower-case:

```
*r(ESC)b(ESC)v(ESC)(ESC)
WE HOLD THESE TRUTHS TO BE SELF-EVIDENT,
*k(ESC)iWe hold these truths to be self-evident,(RET)
(ESC)-a(ESC)v(ESC)(ESC)
We hold these truths to be self-evident,
*
```

The upper- and lower-case capability is useful for reports, memos and other textual material that you list on upper-/lower-case devices. However, all characters are printed as upper-case if you list the file on a line printer or terminal that does not have the upper-/lower-case capability.

Edit Upper

If at any time you want to revert to strictly upper-case editing, type the EU (Edit Upper) command:

```
*eu(ESC)(ESC)
*
```

Upper-case editing is a default mode. Whenever you open a file for editing or create a new file, you must enter the EL command if you want to use the upper-/lower-case capability.

Close the file DECIND.USA by typing:

```
*EX(ESC)(ESC)
*
```

SUMMARY: EDITING COMMANDS

EDIT filespec
: Activate the editor and open the file for editing.

EDIT/CREATE filespec
: Activate the editor and create a new file.

Control Commands

CTRL/L
: Insert a form feed. The form feed character is used to delimit pages of text in a file (introduced as part of text by the Insert command).

CTRL/X
: Ignore all commands in the current editing command string.

Command Arguments

n(+ or –)
: n is an integer value between –16383 and +16383 which sets the range of a command's actions based on the pointer's current position.

0
: Beginning of the current line (the line containing the pointer).

/
: End of the text in the text buffer.

Input/Output Commands (pointer is not repositioned)
(x indicates that an argument may be used)

EX
: Exit; terminate editing, transfer the contents of the text buffer and the remainder of input file to the output file; close input and output files; return to monitor command mode.

xL
: List; list, from the pointer, x lines of text.

xN
: Next; write the contents of the text buffer to the output file, clear the text buffer, and read into it the next page from the input file; perform this write/read sequence x times.

V
: Verify; list the current line (the line containing the pointer) on the terminal.

Pointer Location Commands (pointer is repositioned)
(x indicates that an argument may be used)

xA
: Advance; move the pointer to the beginning of the xth line from the current pointer position.

B
: Beginning; move the pointer to the beginning of the text buffer.

xJ
: Jump; move the pointer forward or backward by x characters.

Text Modification Commands (pointer is repositioned)
(x indicates that an argument may be used)

xD
: Delete; erase x characters to the right (or left) of the pointer.

I text ESC
: Insert; insert text into the text buffer at the present pointer position.

xK
: Kill; erase x lines of text, beginning at the pointer.

Search Command (pointer is repositioned)
(x indicates that an argument may be used)

xG text
: Get; search the text buffer, beginning at the pointer, for the x occurrence of the indicated text string and leave the pointer at the end of the text string.

Upper-/Lower-Case Commands (pointer is not affected)

EL
: Edit Lower; accept characters typed at the keyboard as upper-/lower-case.

EU
: Edit Upper; revert back to upper-case editing (after EL).

USING A GRAPHICS DISPLAY TERMINAL DURING EDITING

If your system configuration includes VT11 display hardware, there are several advantages to your using it during editing[1]. First, the graphics screen becomes a window into the text buffer, exposing twenty lines of text at a time (the current line, the ten lines preceding it and the nine lines following it). Figure 5-2 illustrates this format. As you edit, the lines in view shift to conform to the current line. In addition, the pointer is visible and appears as a blinking cursor. Its position is automatically adjusted as you execute editing commands. Finally, four lines at the bottom of the screen display the last three command lines plus the current command line. Horizontal dashes separate the text of the file from your commands.

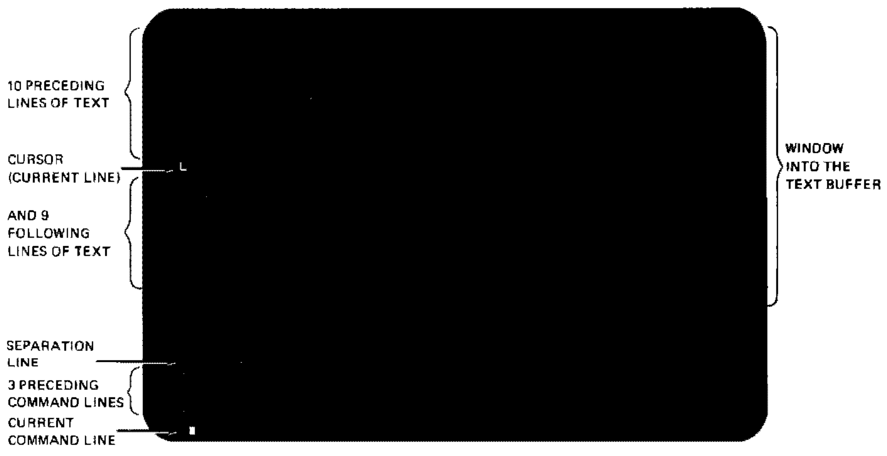

10 PRECEDING LINES OF TEXT
CURSOR (CURRENT LINE)
AND 9 FOLLOWING LINES OF TEXT
SEPARATION LINE
3 PRECEDING COMMAND LINES
CURRENT COMMAND LINE
WINDOW INTO THE TEXT BUFFER

Figure 5-2 Text Window Format

Normal Use of the Graphics Display

All editing commands and functions described so far can be used when the graphics screen is enabled. The only difference is that terminal I/O is rearranged on the screen as shown in Figure 5-2. Note that the L and V editing commands become superfluous since the

[1]If your system does not have VT11 display hardware, skip to the next section, Creating the Demonstration Programs.

pointer is always displayed on the screen. Also, since twenty lines of text are always displayed, any List command within that range is unnecessary.

Currently, your graphics screen is not enabled. To enable it, use the monitor GT ON command as you did in Chapter 4:

Long and Short Command Format

```
.GT ON(RET)
```

Now when you use the EDIT command to activate the editor, the graphics screen will be rearranged as shown in Figure 5-2. You can use the CTRL/E command, described in Chapter 4, to request simultaneous I/O on the terminal printer and graphics screen.

Immediate Mode

In addition to the regular editing capability, a quick and easy method of graphics editing, called immediate mode, is available. Immediate mode uses a simplified set of editing commands that are limited to pointer relocation and character deletion and insertion. Most of these commands are similar to the special CTRL commands because to type them you use the CTRL key in combination with another character key. However, the use of these particular control commands is meaningful only in the editor immediate mode. Table 5-2 lists the commands.

Table 5-2 Immediate Mode Commands

Command	Meaning
CTRL/N	Advance the cursor to beginning of next line (equivalent to A)
CTRL/G	Move the cursor to the beginning of the previous line (equivalent to −A)
CTRL/D	Move the cursor forward by one character (equivalent to J)
CTRL/V	Move the cursor back by one character (equivalent to −J)
DELETE	Delete the character immediately preceding the cursor (equivalent to −D)
ESCAPE	Return control to the editing command mode
double ESCAPE	summon immediate mode

Use the editor to open a new file called IMMODE.TXT:

Long and Short Command Format

```
.EDIT/CREATE IMMODE.TXT (RET)
*
```

ESCAPE ESCAPE

Now activate immediate mode. You do this by typing the ESCAPE key twice in response to the editing command mode asterisk. Since there are no other commands in the command line, the editor recognizes the double ESCAPE as an immediate mode command.

```
*(ESC)(ESC)
!
```

The editor responds by printing an exclamation mark in the command portion of the screen; the exclamation mark signifies that you are using immediate mode.

Character Insertion

Character insertion is the default operation and occurs whenever you type a character other than one of the immediate mode commands listed in Table 5-2.

The next several paragraphs demonstrate the use of the immediate mode commands on a selected portion of text. Remember that all characters that you type that are not immediate mode commands are treated as input. Commands do not echo on the graphics screen so all you ever see is the current text file. Type the following:

```
TO BE, OR NOT TO BE—THAT IS THE QUESTION: (RET)
WHETHER 'TIS NOBLER IN THE MIND AND HEART TO SUFFER (RET)
THE SLINGS OF OUTRAGEOUS FORTUNE (RET)
OR TO TAKE ARMS AGAINST A SEA OF TROUBLES, (RET)
AND BY OPPOSING END THEM? (RET)
```

CTRL/G

As you can see on the graphics screen, the cursor (pointer) is positioned at the beginning of a new line. CTRL/G, equivalent to −A in standard editing, moves the cursor to the beginning of the previous line; the cursor is repositioned immediately. Type:

```
(CTRL/G)
(CTRL/G)
(CTRL/G)
```

The cursor has moved backward three lines, one line for each CTRL/G command and is positioned before the line:

```
THE SLINGS OF OUTRAGEOUS FORTUNE,
```

CTRL/V

CTRL/V, equivalent to –J, moves the cursor back by one character. Move the cursor back over the carriage return and line feed at the end of the previous line by typing the CTRL/V command eleven times (remember, the carriage return and line feed count as two characters):

```
(CTRL/V)          (eleven (11) times)

WHETHER 'TIS NOBLER IN THE MIND AND HEART TO SUFFER
```

DELETE

This positions the cursor before the word TO. DELETE, equivalent to –D, deletes the character immediately preceding the cursor. Type the DELETE key ten times:

```
(DEL)             (ten (10) times)

WHETHER 'TIS NOBLER IN THE MIND TO SUFFER
```

CTRL/N

CTRL/N, equivalent to A, advances the cursor to the beginning of the next line:

```
(CTRL/N)

THE SLINGS OF OUTRAGEOUS FORTUNE,
```

CTRL/D

CTRL/D, equivalent to J, moves the cursor forward by one character; type CTRL/D ten times:

```
(CTRL/D)          (ten (10) times)

THE SLINGS OF OUTRAGEOUS FORTUNE,
```

Next type this text (it will be inserted immediately to the left of the cursor):

```
(SP) AND (SP) ARROWS
```

The text on the screen should now look as follows:

```
TO BE OR NOT TO BE–THAT IS THE QUESTION;
WHETHER 'TIS NOBLER IN THE MIND TO SUFFER
THE SLINGS AND ARROWS OF OUTRAGEOUS FORTUNE,
OR TAKE ARMS AGAINST A SEA OF TROUBLES,
AND BY OPPOSING END THEM?
```

Check your results and correct any other mistakes you may notice.

ESCAPE

To return to the standard editing command mode, type a single ESCAPE.

```
(ESC)
*
```

This ESCAPE command does not echo on the screen. You should notice that the exclamation point immediately disappears and the text window format returns; an asterisk appears immediately below the exclamation point on the screen.

You use immediate mode only to create and edit text. Operations that move text in and out of the text buffer must be done with standard editing commands.

CTRL/C ESCAPE ESCAPE

You do not need to save the text you have just created, so use the CTRL/C command and two ESCAPEs to return control directly to monitor command mode. As mentioned before, EDIT requires this unusual command combination to prevent an accidental CTRL/C from killing your text.

```
(CTRL/C) (ESC) (ESC)
```

CREATING THE DEMONSTRATION PROGRAMS

Following are two demonstration programs. One is written in the FORTRAN IV programming language and one is written in the MACRO-11 assembly language. Both programs are used in later chapters of this manual and both contain intentional misspellings and errors.

Use the editor to create these programs. Type them exactly as they are shown, including errors. Use tabs and spaces to format each line as shown (remember that tab stops are positioned every eight spaces across the terminal page). Use any of the editing commands described in this chapter. Activate the display editor and immediate mode if you wish.

When you are done, check each file carefully. The two files should match those shown here exactly, including tabs and spaces. Correct any errors that you find that are not intentional. Obtain a listing of each file using B (ESC)/L (ESC) (ESC) before closing the file.

Create the FORTRAN file first. Call it GRAPH.FOR and use the system volume for storage. Then create the MACRO program. Call it SUM.MAC and again use the system volume for storage.

NOTE

Knowledge of the FORTRAN IV and MACRO-11 languages is not necessary to create these demonstration programs.

```
C GRAPH.FOR      VERSION 1
C THIS PROGRAM PRODUCES A PLOT ON THE TERMINAL
C OF AN EXTERNAL FUNCTION, FUN(X,Y)
C THE LIMITS OF THE PLOT ARE DETERMINED BY THE DATA STATEMENTS
C 'STAB' IS FILLED WITH A TABLE OF WEIGHT FLAGS
C 'STRING' IS USED TO BUILD A LINE OF GRAPH FOR PRINTING

        SCAL(ZMIN,ZMAX,MAXZ,K)=ZMIN+FLOAT(K-1)*(ZMAX-ZMIN)/FLOAT(MAXZ-1)
        LOGICAL*1 STRING(13,3),STAB(100)
        DATA XMIN,XMAX,MAXX/-5,5,45/
        DATA YMIN,YMAX,MAXY/-5,5,72/
        DATA FMIN,FMAX/0.0,1.0/
        CALL SCOPY('- 1 2 3 4 5 6 7 8 9 +',STAB)
        MAXFLEN(STAB)
        DO 20 IX=1,MAXX
            X=SCAL(XMIN,XMAX,MAXX,IX)
            CALL REPEAT('*',STRING,MAXY)
            IF(IX.EQ.1 .OR. IX.EQ.MAXX) GOTO 20
                DO 10 IY=2,MAXY-1
                    Y=SCAL(YMIN,YMAX,MAXY,IY)
                    IFUN=2+INT(FLOAT(MAXF-3)*(FUN(X,Y)-FMIN)/(FMAX-FMIN))
10                  STRING(IY)=STAB(MINO(MAXF,MAXO(1,IFUN)))
30          CALL PUTSTRING(7,STRING,' ')
        CALL EXIT
        END

        FUNCTION FUN(X,Y)
        R=SQRT(X**2+Y**2)
        FUN=X*Y*R*EXP(-R))**2
        RETURN
        END
        .TITLE SUM.MAC  VERSION 1

        .MCALL .TTYOUT, .EXIT, .PRINT


        N = 70.           ;NO. OF DIGITS OF 'E' TO CALCULATE

;       'E' = THE SUM OF THE RECIPROCALS OF THE FACTORIALS
;       1/0! + 1/1! + 1/2! + 1/3! + 1/4! + 1/5! + ...

EXP:    .PRINT  #MESSAG            ;PRINT INTRODUCTORY TEXT
        MOV     #N,R5              ;NO. OF CHARS OF 'E' TO PRINT
FIRST:  MOV     #N+1,R0            ;NO. OF DIGITS OF ACCURACY
        MOV     #A,R1              ;ADDRESS OF DIGIT VECTOR
SECOND: ASL     @R1                ;DO MULTIPLY BY 10 (DECIMAL)
        MOV     @R1,-(SP)          ;SAVE *2
        ASL     @R1                ;*4
        ASL     @R1                ;*8
        ADD     (SP)+,(R1)+        ;NOW *10, POINT TO NEXT DIGIT
        DEC     R0                 ;AT END OF DIGITS?
        BNE     2ND                ;BRANCH IF NOT
        MOV     #N,R0              ;GO THRU ALL PLACES, DIVIDING
THIRD:  MOV     -(R1),R3           ;BY THE PLACES INDEX
        MOV     #-1,R2             ;INIT QUOTIENT REGISTER
FOURTH: INC     R2                 ;BUMP QUOTIENT
        SUB     R0,R3              ;SUBTRACT LOOP ISN'T BAD
        BCC     FOURTH             ;NUMERATOR IS ALWAYS < 10*N
        ADD     R0,R3              ;FIX REMAINDER
        MOV     R3,@R1             ;SAVE REMAINDER AS BASIS
                                   ;FOR NEXT DIGIT
        ADD     R2-2(R1)           ;GREATEST INTEGER CARRIES
                                   ;TO GIVE DIGIT
        DEC     R0                 ;AT END OF DIGIT VECTOR?
        BNE     THIRD              ;BRANCH IF NOT
        MOV     -(R1),R0           ;GET DIGIT TO OUTPUT
FIFTH:  SUB     #10.,R0            ;FIX THE 2.7 TO .7 SO
                                   ;THAT IT IS ONLY 1 DIGIT
        BCC     FIFTH              ;(REALLY DIVIDE BY 10)
        ADD     #10+'0,R0          ;MAKE DIGIT ASC II
        .TTYON                     ;OUTPUT THE DIGIT
        CLR     @R1                ;CLEAR NEXT DIGIT LOCATION
        DEC     R5                 ;MORE DIGITS TO PRINT?
        BNE     FIRST              ;BRANCH IF YES
        .EXIT                      ;WE ARE DONE

EXP:    .REPT   N+1
        .WORD   1                  ;INIT VECTOR TO ALL ONES
        .ENDR

MESSAG: .ASCII  /THE VALUE OF E IS:/ <15><12> /2./ <200>
        .EVEN

        .ENDEXP
```

When you have created and checked these two programs, obtained listings, and stored them as files on your system volume, go on to Chapter 6, Comparing Text Files. Chapter 6 demonstrates a proofreading aid that helps you evaluate your editing ability.

REFERENCES

RT-11 System User's Guide (DEC-11-ORGDA-A-D), Maynard, Mass.: Digital Equipment Corporation, 1977.

A guide to the use of the RT-11 operating system. See Chapter 5.

CHAPTER 6
COMPARING TEXT FILES

The RT-11 operating system provides a proofreading aid, called a source comparison, to help you quickly establish the differences between two ASCII text files. During a source comparison, the system compares the two files, character for character, and prints on the terminal (or line printer) any lines that contain differences.

Usually, you perform a source comparison against two files that you expect to be the same or at least similar in content. For example, if an individual has copied one of your files to make changes to it, you can quickly scan the changes by performing a source comparison between the new version and your original. Another use of a source comparison is to check edits that you've made to a file. By comparing the backup file against the edited version, you can proofread the changes since only the portions of text that are different are printed.

In this chapter, you will use source comparisons to find editing errors that may exist in the demonstration programs (SUM.MAC and GRAPH.FOR) that you created in Chapter 5. These demonstration programs contain intentional misspellings and misplaced text which you must correct before the programs can be used in later demonstrations. On your system volume is a counterpart for each file. These counterparts are provided as part of the RT-11 operating system so that you can use them to perform a source comparison against your own versions. The provided programs have essentially been carried one step further in the editing process than your own; they contain no editing errors. Therefore, when you compare them against your versions, the list of differences that is printed will reflect the typing errors that still exist in your versions – some of these errors are intentional; others you may have inadvertently introduced during editing. All must be corrected before you can use the programs.

PERFORMING A COMPARISON

The monitor command used to compare two text files is the DIFFERENCES command. When you type this command on the terminal, it activates the RT-11 utility program called SRCCOM.SAV, which is part of the RT-11 operating system stored on the system volume. The system prompts you for the input file names. Respond to the input prompts with the names of the files you want to compare; the default storage volume is the system volume. The output will be sent to the terminal which is the default device for output.

DIFFERENCES

The programs that you created in Chapter 5 are called SUM.MAC and GRAPH.FOR. Their respective counterparts on the system volume are called DEMOX1.MAC and DEMOF1.FOR. Use the DIFFERENCES command to compare the MACRO (.MAC) files first. The /MATCH option indicates the number of lines that determine a "match", explained in a moment.[1]

Long Command Format

```
.DIFFERENCES/MATCH:1 (RET)
File 1? DEMOX1.MAC (RET)
File 2? SUM.MAC (RET)
```

Short Command Format

```
.DIFFERENCES/MATCH:1 DEMOX1.MAC SUM.MAC (RET)
```

The list of differences printed on your console terminal should be similar to that shown below. It will show all the differences listed here, plus any others that you may have introduced yourself during editing.

Notice the format of the list. Individual sections are marked to help you become acquainted with the format. A description follows the list and you should refer to it as you study the list.

```
 A 1)1        .TITLE DEMOX1.MAC        (VERSION PROVIDED)
 A 2)1        .TITLE SUM.MAC  VERSION 1

 C 1)1        BNE      SECOND          ;BRANCH IF NOT
 D 1)         MOV      #N,RO           ;GO THRU ALL PLACES, DIVIDING
B  ****
 C 2)1        BNE      2ND             ;BRANCH IF NOT
 D 2)         MOV      #N,RO           ;GO THRU ALL PLACES, DIVIDING
   **********
 C 1)1        ADD      #10+'0,RO       ;MAKE DIGIT ASCII
 D 1)         .TTYON                   ;OUTPUT THE DIGIT
B  ****
 C 2)1        ADD      #10+'0,RO       ;MAKE DIGIT ASC II
 D 2)         .TTYON                   ;OUTPUT THE DIGIT
   **********
 C 1)1        .END     EXP
 B ****
 C 2)1        .ENDEXP
 D **********

   Files are different
```

[1] Users of display hardware may wish to enable both the graphics screen and the terminal printer by first typing the CTRL/E command.

The first line of each file is always printed for identification purposes (see lines A in the example list). Usually differences that occur in these two lines are intentional and reflect information that is unique to each file, such as name and file type, version or edit number, and perhaps date of creation.

The numbers that appear at the left margin of the list further identify the files. For example, 1)1 indicates the first page of the first file (the file entered first in the command, in this case, DEMOX1.MAC); 2)1 indicates the first page of the second file (SUM.MAC).

The lines of both files are compared character for character. Blank lines are ignored, but all other characters, including tabs and spaces, are compared. When two lines are found to be different, the system prepares a difference section which it will subsequently print (see B).

DIFFERENCES/ MATCH:n

The system prepares the difference section as follows. When it finds two lines that are different, it notes the page number and records the lines (see C). Next it searches for a match. A match is a certain number of lines in each file which are exactly the same. Since you specified a match of 1 in the /MATCH:n option (/MATCH:1), the system in this case searches for a single line in each file which is exactly the same. When the system finds a match, it records the last line of the match for identification purposes (see D). Then it prints the difference section and repeats the process, preparing a subsequent difference section if more differences exist. Individual difference sections are separated from one another by a long row of asterisks, while the short rows of asterisks separate the lines of the first file from those of the second.

A message is printed following the comparison. Files are different is printed if differences exist; No differences encountered is printed if the files are exactly the same.

Check the list printed on your terminal to find the errors the system detected. Mark each error on the listing of SUM.MAC that you obtained in Chapter 5.

Now perform a source comparison between the FORTRAN files, DEMOF1.FOR and GRAPH.FOR.

Long Command Format

```
    .DIFFERENCES/MATCH:1 (RET)
    File 1? DEMOF1.FOR (RET)
    File 2? GRAPH.FOR (RET)
```

Short Command Format

```
.DIFFERENCES/MATCH:1 DEMOF1.FOR GRAPH.FOR (RET)
```

```
1)1      C DEMOF1.FOR     (VERSION PROVIDED)
2)1      C GRAPH.FOR      VERSION 1

1)1      C 'STAB' IS FILLED WITH A TABLE OF HEIGHT FLAGS
1)       C "STRING" IS USED TO BUILD A LINE OF GRAPH FOR PRINTING
****
2)1      C 'STAB' IS FILLED WITH A TABLE OF WEIGHT FLAGS
2)       C "STRING" IS USED TO BUILD A LINE OF GRAPH FOR PRINTING
**********
1)1              MAXF=LEN(STAB)
1)               DO 20 IX=1,MAXX
****
2)1              MAXFLEN(STAB)
2)               DO 20 IX=1,MAXX
**********
1)1      30          CALL PUTSTR(7,STRING,' ')
1)               CALL EXIT
****
2)1      30          CALL PUTSTRING(7,STRING,' ')
2)               CALL EXIT
**********

Files are different
```

Likewise, mark the errors on the listing of GRAPH.FOR that you obtained in Chapter 5.

Now return to the section in Chapter 5 entitled "Editing a Text File." Review the editing commands described there and the summary at the end of the section. Use the appropriate commands to correct the files SUM.MAC and GRAPH.FOR. When you finish editing, again perform source comparisons against DEMOX1.MAC and DEMOF1.FOR. If you have edited the files correctly, this message should print on your console terminal in each case:

```
    No differences encountered
```

This message indicates that no differences were found during the comparison. Thus, your programs are ready for use in later demonstrations and you know how to successfully create and edit programs.

If differences still exist in your files and you cannot seem to resolve them by reediting, you may continue to the next chapter if you wish. However, you need practice editing and it is to your advantage to rework the examples in both Chapter 5 and this chapter.

SUMMARY: COMPARISON COMMAND

DIFFERENCES
List the differences between two ASCII text files.

DIFFERENCES/MATCH:n
Indicate the number of lines (n) to determine a match; the default number is 3.

REFERENCES

RT-11 System User's Guide (DEC-11-ORGDA-A-D). Maynard, Mass.: Digital Equipment Corporation, 1977.

A guide to the use of the RT-11 operating system. See Chapters 4 and 15.

CHAPTER 7
PERFORMING FILE MAINTENANCE OPERATIONS

The system volume, as it is initially supplied, contains only the files of the RT-11 operating system – the monitor files, the system device handlers, the system utility programs, and perhaps the language processors. Since the system volume serves as the default storage volume for all system operations (unless DK: was assigned to another volume), you will discover that it acquires many additional files during normal use. For example, files that you create with the editor are written on the system volume; edited files automatically create backup versions on the system volume; many utility programs create output and listing files on the system volume as part of their normal processing operations. By the time you finish an average session of computer operations, several new file names are added to the directory of your system volume. Eventually your system volume may become full and its directory cluttered with the names of files for which you have no use. To avoid this you should perform regular housekeeping, or file maintenance, operations as you use the system. You should update and transfer copies of your important files to other storage volumes for safekeeping and later use, and you should delete from your system and storage volume directories the names of files for which you no longer have a need.

The RT-11 operating system provides a number of monitor commands for this purpose. These commands activate the RT-11 utility programs called PIP.SAV, DUP.SAV, and DIR.SAV (which are part of the RT-11 operating system stored on your system volume) allowing you to perform file transfer and file erase operations. The commands used in this chapter show one way to maintain your system and storage volume. When you become more familiar with system operations and learn some of the commands not described here, you may prefer other methods.

FILE DIRECTORY OPERATIONS

Before you perform operations that might move or erase files on a volume, first list a directory of the volume involved. The directory tells you the full names of files, their sizes, and whether backup copies exist. A directory of your system volume shows the additional files that have been added to it through normal use.

First obtain a directory of your system volume (as you learned in Chapter 4), using the appropriate command to list it on either the terminal or the line printer. The directory is relatively long; let it list to completion.

Long and Short Command Formats

(Line printer)

```
.DIRECTORY/PRINTER (RET)
```

(Terminal)

```
.DIRECTORY (RET)
```

At the end of the system volume directory you should see several additional entries. These files are the result of the system operations you have performed so far:

```
DECIND.USA        2 13-JUN-77
DECIND.BAK        2 13-JUN-77
GRAPH .FOR        2 13-JUN-77
GRAPH .BAK        2 13-JUN-77
SUM   .MAC        4 13-JUN-77
SUM   .BAK        4 13-JUN-77
```

Next list a brief directory of your storage volume. This directory should be empty (void of any file names or file types) since you initialized the directory in Chapter 4.

Long and Short Command Formats

(Line printer)

```
.DIRECTORY/BRIEF/PRINTER VOL: (RET)
```

(Terminal)

```
.DIRECTORY/BRIEF VOL: (RET)
```

These directories give you the information you need to erase and copy files. For example, you know the additional files that are now on your system volume and you know that since the directory of the storage volume is empty, there is ample room on it for new files to be copied.

MULTIPLE FILE OPERATIONS

You often have occasion to perform the same utility operation on several files. For example, you may copy from one volume to another all files with the file type .MAC, or you may erase from a volume all files with the name TEST. Rather than perform the required operation on the files one at a time, it is easier to use a shorthand method provided by the RT-11 operating system called the wildcard construction. This construction allows you to substitute an asterisk (*) or percent sign (%) for a portion of the file name which is variable among all the files you want used in the operation. For example, specifying DECIND.* in a command causes the operation to act on all files with the file name DECIND, regardless of their file type; *.BAK causes the system to act on files with the file type BAK, regardless of their file name. Specifying TEST%.FOR causes the operation to act on all files having a type of FOR, starting with the four characters TEST, and having any fifth character (e.g., TESTA.FOR, TEST1.FOR, etc.).

A special use of the wildcard construction involves substitution of an asterisk for both file name and file type. *.* implies that all files, regardless of the file name or file type, are to be used in the operation.

Exercises in this chapter and throughout the remainder of the manual demonstrate various uses of the wildcard construction. However, it is valid only for the file maintenance commands listed in this chapter; the wildcard construction is not valid for any other commands.

FILE COPYING OPERATIONS

Storage volumes provide an area where you can store important files. Since most files are originally created on the default system volume, you must copy them from the system volume to the storage volume. The following exercises show you how to make backup copies on your storage volume of the two provided demonstration programs (DEMOF1.FOR and DEMOX1.MAC), and how to copy to the storage volume the two programs you created (GRAPH.FOR and SUM.MAC).

COPY

The monitor command that copies files between volumes is the COPY command. This command instructs the system to duplicate the file that you indicate as input; it then gives the new file the name and file type that you specify as output. The original version of the file is unaffected; that is, the original version is not physically moved to the new volume, but a copy of it is made there.

To copy GRAPH.FOR to your storage volume under the new name GRAPH.TWO, type:

Long Command Format

```
.COPY (RET)
From? GRAPH.FOR (RET)          (System volume is assumed
                               for input.)
To   ? VOL:GRAPH.TWO (RET)
```

Short Command Format

```
.COPY GRAPH.FOR VOL:GRAPH.TWO (RET)
```

The system makes an exact copy of the file GRAPH.FOR on the storage volume and gives the copy the name GRAPH.TWO. When the operation is complete, the monitor prints a period at the left margin and waits for you to enter the next command. This time, copy SUM.MAC to the storage volume.

Long Command Format

```
.COPY (RET)
From? SUM.MAC (RET)
To   ? VOL:SUM.MAC (RET)
```

Short Command Format

```
.COPY SUM.MAC VOL:SUM.MAC (RET)
```

The system copies the file SUM.MAC to your storage volume and gives the copy the name SUM.MAC.

Now, copy the two provided demonstration programs, DEMOF1.FOR and DEMOX1.MAC, to the storage volume.

Long Command Format

```
.COPY (RET)
From? DEMOF1.FOR (RET)
To   ? VOL:DEMOF1.FOR (RET)
.COPY (RET)
From? DEMOX1.MAC (RET)
To   ? VOL:DEMOX1.MAC (RET)
```

Short Command Format

```
.COPY DEMOF1.FOR VOL:DEMOF1.FOR (RET)

.COPY DEMOX1.MAC VOL:DEMOX1.MAC (RET)
```

A directory of your storage volume should verify that it now contains these four files:[1]

Long and Short Command Formats

```
.DIRECTORY VOL: (RET)
 13-Jun-77
GRAPH .TWO      2 13-Jun-77      DEMOF1.FOR      2 13-Jun-77
SUM   .MAC      3 13-Jun-77      DEMOX1.MAC      4 13-Jun-77
 4 Files, 11 Blocks
 4751 Free blocks
```

FILE RENAMING OPERATIONS

RENAME

The directory you just listed shows that you copied the GRAPH demonstration file to your storage volume under a new file type, .TWO. Assume you did not intend to copy it using a new file type and now wish that it were assigned its original file type, .FOR. Use the monitor RENAME command to rename the file already on the storage volume.[2]

Long Command Format

```
.RENAME (RET)
From? VOL:GRAPH.TWO (RET)
To   ? VOL:GRAPH.FOR (RET)
```

Short Command Format

```
.RENAME VOL:GRAPH.TWO VOL:GRAPH.FOR (RET)
```

The RENAME command simply changes the file name and/or file type of a file in the volume directory without altering or moving

[1]If you are using magtape or cassette as your storage volume, read the section in Appendix B entitled "Directory vs Nondirectory-Structured Volumes".

[2]Magtape and cassette users cannot use the RENAME command and should read Appendix B, "Alternate RENAME Operation for Magtape and Cassette Users".

the file itself. When you perform a rename operation, the volume indicated in the input and output portions of the command must be the same; otherwise a system message is printed.

Rename the file copies DEMOX1.MAC and DEMOF1.FOR presently on your storage volume to EXAMP.MAC and EXAMP.FOR respectively. Also rename a file currently on your system volume only, DEMOSP.MAC, to SPOOL.MAC for a later exercise.

```
.RENAME VOL:DEMOX1.MAC VOL:EXAMP.MAC (RET)

.RENAME VOL:DEMOF1.FOR VOL:EXAMP.FOR (RET)

.RENAME DEMOSP.MAC SPOOL.MAC (RET)
```

Again list a directory of your storage volume to verify that the renaming operation occurred.

Long and Short Command Formats

```
.DIRECTORY VOL: (RET)
 13-Jun-77
GRAPH .FOR      2 13-Jun-77       EXAMP .FOR      2 13-Jun-77
SUM   .MAC      3 13-Jun-77       EXAMP .MAC      4 13-Jun-77
 4 Files, 11 Blocks
 4751 Free blocks
```

FILE DELETION OPERATIONS

Once copies of your important files are stored on a storage volume, you can delete (erase) from the system (or any other) volume those files that you no longer need. The file deletion operation deletes the entry from the volume directory. Thus the space that the file occupies on the volume becomes available for reuse. Files that you want to delete generally include .BAK files created during editing, temporary files created by utility programs, or any other unnecessary files.

Now that you have copies of your important files, you can delete several file names from your system volume. For example, you can delete all files with a .BAK file type created as a result of editing. You can delete the file DECIND.USA, since this was created only for editing practice. Finally, you can delete the files GRAPH.FOR and SUM.MAC since copies of these are now on VOL:.

Do not delete EXAMP.FOR or EXAMP.MAC even though copies of these are also on VOL:. You should consider these two files part of the RT-11 operating system, and therefore should not be erased from the system volume. These copies can serve as additional backups for the files on the storage volume.

DELETE

The monitor DELETE command is used to delete file names from a volume. The DELETE command defaults to requesting confirmation from the user by printing each file name on the terminal before it deletes it. This gives you the opportunity to confirm each file before deletion. If you type a Y response, the system deletes the file name, while an N response instructs the system to ignore that file name and go on to the next. You can specify as many as six input files for deletion. Notice how you use the wildcard construction in one of the input files to delete all files with a .BAK file type.

Long Command Format

```
.DELETE  (RET)
Files? DECIND.USA,*.BAK,GRAPH.FOR,SUM.MAC
 Files deleted:
DK:DEMOF1.BAK  ? Y (RET)
DK:INTEXT.BAK  ? Y (RET)
DK:DECIND.BAK  ? Y (RET)
DK:DECIND.USA  ? Y (RET)
DK:SUM.MAC     ? Y (RET)
DK:GRAPH.BAK   ? Y (RET)
DK:DEMOX1.BAK  ? Y (RET)
DK:GRAPH.FOR   ? Y (RET)
```

Short Command Format

```
.DELETE DECIND.USA,*.BAK,GRAPH.FOR,SUM.MAC (RET)
DK:DECIND.USA? Y  (RET)
DK:DECIND.BAK? Y  (RET)
DK:GRAPH  .BAK? Y  (RET)
DK:SUM    .BAK? Y  (RET)
DK:GRAPH  .FOR? Y  (RET)
DK:SUM    .MAC? Y  (RET)
```

FILE LISTING OPERATIONS

You sometimes need to obtain a listing of a file before you can decide whether or not to delete it. In Chapter 5, you used the RT-11 editor to obtain listings of the files you created. You can also obtain listings of files using monitor commands. One command lists a file on the console terminal; another lists a file on the line printer.[1] The system volume is the assumed storage volume for the input file.

[1]If a line printer is available on your system, you should always use it for listings. Line printer listings are neater and print faster than terminal listings.

Type one of the following sets of commands to obtain listings of EXAMP.MAC and EXAMP.FOR.

PRINT

TYPE

Long Command Format

(Line Printer) (Terminal)

```
.PRINT (RET)                          .TYPE (RET)
Files? VOL:EXAMP.MAC (RET)            Files? VOL:EXAMP.MAC (RET)

.PRINT (RET)                          .TYPE (RET)
Files? VOL:EXAMP.FOR (RET)            Files? VOL:EXAMP.FOR (RET)
```

Short Command Format

(Line Printer) (Terminal)

```
.PRINT VOL:EXAMP.MAC (RET)            .TYPE VOL:EXAMP.MAC (RET)

.PRINT VOL:EXAMP.FOR (RET)            .TYPE VOL:EXAMP.FOR (RET)
```

These file maintenance operations are the kinds of operations that you should perform periodically as you use the system. File maintenance keeps your system and storage volumes up-to-date and provides maximum free space on volumes for new files.

SUMMARY: FILE MAINTENANCE COMMANDS

COPY
: Copy the specified file from one volume to another.

DELETE
: Delete the specified file(s) from the volume's directory. Confirmation required before deleting the file.

DIRECTORY
: List the volume directory on the terminal.

DIRECTORY/PRINTER
: List the volume directory on the line printer.

PRINT
: List the contents of the specified file on the line printer.

RENAME
: Give a new name to the specified file.

TYPE
: List the contents of the specified file on the terminal.

REFERENCES

RT-11 System User's Guide (DEC-11-ORGDA-A-D), Maynard, Mass.: Digital Equipment Corporation, 1977.

A guide to the use of the RT-11 operating system. See Chapter 4.

CHAPTER 8
CHOOSING A PROGRAMMING LANGUAGE

Programming languages and language processors are aids provided by the operating system to help you develop programs of your own. Whenever you plan to write a program, you must first decide on the programming language that you will use, since most computer systems support several. After you have chosen the language, you must design and code your program using appropriate language statements and being careful to follow language formatting rules and restrictions. Finally, you must use the corresponding language processor, which is stored on the system volume or on a volume of its own, to convert your program statements into a format suitable for execution.

HIGH-LEVEL VS MACHINE-LEVEL LANGUAGES

Hundreds of programming languages have been developed for computer systems. Some languages can be used only for specific applications or in conjunction with a particular computer system. Other languages are general purpose; they are suitable for a variety of problem-solving situations and, in addition, are easy to learn and use. The languages demonstrated in this manual include two well-known and widely-used high-level programming languages (BASIC and FORTRAN IV) and one RT-11 system-specific machine-level programming language (MACRO-11).

High-level languages, like BASIC and FORTRAN, are usually easy to learn and use. You write programs using language statements that need not deal with the specifics of the computer system. The language processor (and perhaps other utility programs as well) handle all conversions that are necessary for program execution. Since a single high-level language statement may perform several computer operations, and since you need not be concerned or familiar with the structure of the computer and peripheral devices, you can concentrate solely on solving the problem at hand. The language processor takes care of translating the statements into the appropriate computer information.

Thus, high-level languages are considered machine-independent languages because language statements are such that any program written in the language can usually be executed on an entirely different computer system (that supports the language) with no or relatively few modifications.

On the other hand, machine-level languages, like the assembly language MACRO-11, require that you do have knowledge of the computer and peripheral devices and how they work together. You write programs in formats that are closer to those required for execution. Since a single machine-level language statement usually performs only one computer operation, you must account in your program for each computer operation that will be required.

For this reason, machine-level languages are machine-dependent languages. The program is coded in a format that is not usually interchangeable among systems. Machine-level language programs can be efficient because the knowledgeable programmer will choose the fastest and most precise instructions for getting the job done.

Table 8-1 lists a comparison of high-level vs. machine-level languages.

Table 8-1 Language Comparisons

High-Level	Machine-Level
Easy to learn and use; no experience required	More difficult to learn and use; familiarity with the computer system required
Machine-independent	Machine-dependent
Many hidden conversions necessary for program execution; more computer memory is used	Only direct translation is necessary for program execution; less computer memory is used
Slower execution time	Faster execution time
Less efficient; the system makes decisions concerning computer operations	More efficient; the programmer makes decisions concerning computer operations
Easier to debug (find and fix errors)	Harder to debug (find and fix errors)
Easier to understand programs; functions added with less difficulty	Harder to understand programs; functions added with greater difficulty

In general, beginning programmers, students, commercial applications programmers, and the casual computer user tend to prefer high-level languages because they are less difficult to learn and use and produce fast results. System programmers, on the other hand, may prefer machine-level languages. The programs they write (those that make up an operating system, for example) must often be as fast, efficient, and concise as possible.

RT-11 PROGRAMMING LANGUAGES

The designers of a computer system generally select programming languages that they feel will satisfy and suit the current (or perhaps potential) system user environment. The RT-11 computer system is designed for use in many environments: education, business, laboratory, etc. Some of its applications include data acquisition and analysis, record keeping, control systems, and learning through computer simulation. RT-11 programmers and users include both the very knowledgeable and the student/beginner.

To satisfy the varied requirements of these environments, RT-11 supports several programming languages:

High-Level	Machine-Level
BASIC-11	MACRO-11
FORTRAN IV	
DIBOL	
APL	
FOCAL-11	

Whenever you choose one or more of these programming languages for your own use, consider the following criteria:

- What is your programming experience? What languages do you already know?
- How much time do you have to learn a new language?
- For what applications will you use the language? How important are program speed and efficiency?
- Will you use your program on any other computer systems?

If you are already familiar with a language supported by the system, you may prefer to continue using that language rather than spend time learning a new one. However, if you want to learn a language, consider your application. High-level languages handle most programming jobs. Unless you plan to write extremely detailed or time-critical programs you should select a high-level language.

If you are a beginning programmer, you may prefer to start with a language like BASIC or FOCAL. Both are conversational, interactive languages. Language statements use simple, English-like words and common mathematical expressions. You can request immediate answers to problems by using the immediate modes of the languages, or you can create detailed programs by combining single language statements into larger segments. FOCAL-11 is DIGITAL's programming language for solving numerical problems; BASIC-11 is a superset of the industry-standard BASIC developed at Dartmouth College. Chapter 10 of this manual describes BASIC-11 in more detail.

If your application mainly requires the use of complicated mathematical operations or mixed data types, you may prefer to select the programming language APL. This language uses a concise and powerful shorthand notation to perform arithmetic and logical operations on vectors, matrices, and arrays.

RT-11 FORTRAN IV is a superset of the industry-standard FORTRAN IV. This language has long been recognized for its use in the scientific field; in addition, it is one of the most commonly supported languages across systems. You may decide to choose FORTRAN IV because it is a more powerful language than either FOCAL or BASIC or because you plan to use your programs on more than one system. Chapter 9 of this manual describes FORTRAN IV in more detail.

Finally, if you are an experienced user, you may select the machine-level programming language MACRO-11. This is a powerful language that allows user programs to access and utilize every possible feature available on the RT-11 computer system. The language requires a considerable amount of computer experience and knowledge to be used effectively, however. The MACRO-11 language is best for you if you are a system programmer or an experienced high-level language programmer. It is described in more detail in Chapter 11 of this manual.

CHOOSING A LANGUAGE FOR THE DEMONSTRATION

Three RT-11 programming languages are demonstrated in the next several chapters of this manual; FORTRAN IV, BASIC-11, and MACRO-11. Consider your ability as a programmer. If you are a beginner, BASIC is probably the best language for you to start with:

FORTRAN is also a good choice. However, you need not be proficient in any of these programming languages to perform the exercises provided in this manual.

Your particular RT-11 computer system may not provide all three languages. First check question 6 in the Hardware Configuration section of Chapter 2 to find out which languages are available on your system.

Then select a language to continue the demonstration. If you choose FORTRAN IV, continue to Chapter 9. If you choose BASIC-11, go on to Chapter 10. If you choose MACRO-11, go to Chapter 11.

REFERENCES

Katzan, Harry Jr., *Information Technology, The Human Use of Computers*. New York: Mason & Lipscomb Publishers, Inc., Petrocelli Books, 1974.

A textbook covering basic computing concepts, programming languages, and topics in computers and society. See Part II, Chapters 7, 8, and 9.

PDP-11 Computer Family – Software and Services. Maynard, Mass.: Digital Equipment Corporation, 1977.

An overview of the available PDP-11 family products and services.

PDP-11 Software Handbook. Maynard, Mass.: Digital Equipment Corporation, 1975.

A general overview and introduction to available PDP-11 software, operating systems, and language processors. See Section III, Chapters 1, 2, and 5.

CHAPTER 9

RUNNING A FORTRAN IV PROGRAM

The FORTRAN IV programming language[1] is a machine-independent programming language that was originally designed as a quick and easy aid for solving mathematical equations and formulas. However, FORTRAN IV is a powerful language and not difficult to learn or use, and is also well-suited to many other kinds of applications.

THE FORTRAN IV PROGRAMMING LANGUAGE

FORTRAN (FORmula TRANslation) is an algebraically-oriented language. You write a FORTRAN program as a sequence of language statements that combine common English words with quasi-algebraic formulas. You then arrange groups of the language statements into logical units called program units. One or more program units comprise the entire executable FORTRAN source program.

When you are satisfied with the logic of your FORTRAN source program, you use the RT-11 editor to create it as a file (like you did in Chapter 5). You use tabs and spaces to properly format each line, and you may choose to insert comment statements throughout the source code to explain what various parts of the program are doing. When you have finished creating the program as a complete, edited file, you next enter it as input to the FORTRAN IV language processor, which is stored on your system volume or on a separate volume of its own. The FORTRAN IV language processor processes (compiles) the language statements, converting them into internal machine-language code called object code. This code is next processed by the system linker, which combines your program units and necessary system-supplied routines to make your program suitable for execution. The development of an executable FORTRAN program is represented in Figure 9-1.

[1]The PDP-11 FORTRAN IV programming language conforms to the specifications for American National Standard FORTRAN X3.9-1966.

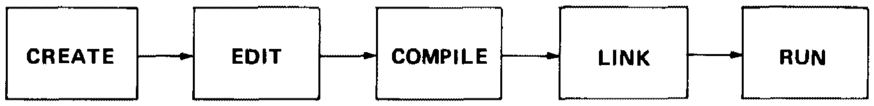


Figure 9-1 Evolution of a FORTRAN Program

THE FORTRAN IV LANGUAGE PROCESSOR

The FORTRAN IV language processor is a compiler that accepts information in one format (i.e., your source program) and translates it into another format (i.e., a machine language program). Since you originally use the editor to create a FORTRAN source program in ASCII format, you must next translate it into a machine format that the computer can use. The FORTRAN compiler performs the translation, producing as output a new version of the program in object format, called an object module. You may optionally instruct the FORTRAN compiler to produce a listing of the source program at the same time. Figure 9-2 is a diagram of the compiler's function.

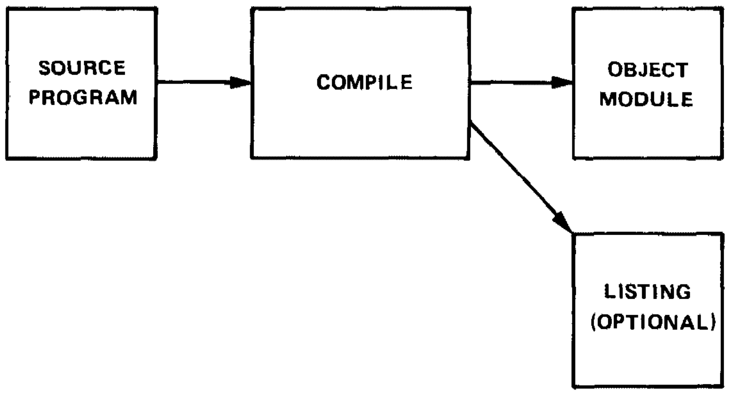


Figure 9-2 Function of a FORTRAN Compiler

USING LIBRARY MODULES

Typical FORTRAN IV programs often require similar operations. For example, most programs use routines and instructions that calculate square roots, exponentials, and other arithmetic functions; handle input and output operations; detect certain kinds of error conditions; test values; calculate subscripts; perform conversions; and other similar kinds of processes. Thus, these commonly-used operations have been gathered into a special file called SYSLIB.OBJ (default System Library), which is provided with the RT-11 operating system and is stored on your system volume.

During processing of your source program, the FORTRAN IV compiler examines each language statement in your program. If you use operations that are provided in SYSLIB, the compiler notes this and makes the appropriate references to SYSLIB. It translates all the information gathered during processing (your converted language statements and the references to SYSLIB) into numerical data called object code, a machine language code that the system linker can use. The result of the compilation, therefore, is an object format file, called an object module, which is automatically joined with SYSLIB (containing many object modules) and with any other required object modules, at link time. Linking all the necessary object modules together produces a complete, workable FORTRAN program.

COMPILING THE FORTRAN IV PROGRAM

In Chapter 5 you used the RT-11 editor to create a FORTRAN source program, which you then stored on your storage volume. Since a source program is in ASCII format, the next step is to use the FORTRAN IV compiler to convert it to object code.

Some RT-11 systems store the FORTRAN IV compiler on a volume apart from the system volume[1]. You can quickly determine whether the FORTRAN IV compiler is on your system volume by using the DIRECTORY command.

```
.DIRECTORY SY:FORTRA.SAV (RET)
```

Note the system response. If the directory entry for FORTRA.SAV is listed on your terminal, then the required FORTRAN files are on your system volume. However, if FORTRA.SAV did not appear in the directory listing, then the required files are not part of your system volume. Before you can use the compiler, you must make a volume substitution. Read the section in Appendix B entitled "Using the FORTRAN/BASIC Language Volume".

The next step is using the monitor COPY command to copy the FORTRAN source program from the storage volume (where you stored it in Chapter 7) back to the system volume which serves as the default volume for input/output operations.

[1]This is true for any RT-11 system volume that does not have enough free blocks to accommodate the FORTRAN system files. RX01 diskette is an example.

Remember that on your storage volume are two FORTRAN source programs, the one you created (GRAPH.FOR) and the one provided as part of the system (EXAMP.FOR). Which of these you should use depends on the results of the source comparison you performed in Chapter 6. If the comparison resulted in NO DIFFERENCES ENCOUNTERED, copy your own program (GRAPH.FOR) as follows:

Long Command Format

```
.COPY (RET)
From? VOL:GRAPH.FOR (RET)
To   ? GRAPH.FOR (RET)
```

Short Command Format

```
.COPY VOL:GRAPH.FOR GRAPH.FOR (RET)
```

However, if the FILES ARE DIFFERENT message was generated by the comparison, use the provided program (EXAMP.FOR) instead, copying it under the new name GRAPH.FOR:

Long Command Format

```
.COPY (RET)
From? VOL:EXAMP.FOR (RET)
To   ? GRAPH.FOR (RET)
```

Short Command Format

```
.COPY VOL:EXAMP.FOR GRAPH.FOR (RET)
```

The FORTRAN source file now resides on your system volume under the name GRAPH.FOR and is the file that you will process with the FORTRAN IV compiler. The command used to compile a FORTRAN source program is the monitor FORTRAN command.

FORTRAN

Use the FORTRAN command with its /LIST option to compile your program and produce a listing. The system prompt asks you to supply the input file name. You can omit typing the .FOR file type since the FORTRAN command assumes this file type unless you indicate otherwise. The system will assign the name GRAPH.OBJ to the object module and GRAPH.LST to the listing file and store both newly-created files on the default storage volume.

Long Command Format

```
.FORTRAN (RET)
Files? GRAPH/LIST (RET)
```

Short Command Format

```
.FORTRAN GRAPH/LIST (RET)
```

Compilation begins. If the compiler discovers an error during processing, it prints a message. In this particular case, you should see the following on your terminal printer or screen:

```
.MAIN.
?FORTRAN-I-[.MAIN.] Errors: 5, Warnings: 0
FUN
?FORTRAN-I-[FUN    ] Errors: 1, Warnings: 0
```

This indicates that during processing, the FORTRAN IV compiler found a total of six errors in the source program. It helps at this point to look at the listing produced by the compiler, because more information is shown there. Print the listing on either the line printer or terminal, using the appropriate command below:

Long Command Format

(Line printer) (Terminal)

```
.PRINT(RET)                   .TYPE(RET)
Files? GRAPH.LST (RET)        Files? GRAPH.LST(RET)
```

Short Command Format

(Line printer) (Terminal)

```
.PRINT GRAPH.LST (RET)        .TYPE GRAPH.LST (RET)
```

Your listing should look like the following example.

NOTE

It is not necessary that you understand the FORTRAN IV language or the way this program works to successfully complete the exercises in this chapter.

```
FORTRAN IV                      Tue 05-Jul-77 12:15:13                    PAGE 001

      C EXAMP.FOR        (VERSION PROVIDED)
      C THIS PROGRAM PRODUCES A PLOT ON THE TERMINAL
      C OF AN EXTERNAL FUNCTION, FUN(X,Y)
      C THE LIMITS OF THE PLOT ARE DETERMINED BY THE DATA STATEMENTS
      C "STAB" IS FILLED WITH A TABLE OF HEIGHT FLAGS
      C "STRING" IS USED TO BUILD A LINE OF GRAPH FOR PRINTING
0001          SCAL(ZMIN,ZMAX,MAXZ,K)=ZMIN+FLOAT(K-1)*(ZMAX-ZMIN)/FLOAT(MAXZ-1)
0002          LOGICAL*1 STRING(13,3),STAB(100)
0003          DATA XMIN,XMAX,MAXX/-5,5,45/
0004          DATA YMIN,YMAX,MAXY/-5,5,72/
0005          DATA FMIN,FMAX/0.0,1.0/
0006          CALL SCOPY('- 1 2 3 4 5 6 7 8 9 +',STAB)
0007          MAXF=LEN(STAB)
0008          DO 20 IX=1,MAXX
0009             X=SCAL(XMIN,XMAX,MAXX,IX)
0010             CALL REPEAT('*',STRING,MAXY)
0011             IF(IX.EQ.1 .OR. IX.EQ.MAXX) GOTO 20
0013              DO 10 IY=2,MAXY-1
0014                   Y=SCAL(YMIN,YMAX,MAXY,IY)
0015                   IFUN=2+INT(FLOAT(MAXF-3)*(FUN(X,Y)-FMIN)/(FMAX-FMIN))
0016   10              STRING(IY)=STAB(MINO(MAXF,MAXO(1,IFUN)))
0017   30        CALL PUTSTR(7,STRING,' ')
0018          CALL EXIT
0019          END


FORTRAN IV       Diagnostics for Program Unit .MAIN.

In line 0003,  Error:    Modes of variable "XMIN" and data item differ
In line 0004,  Error:    Modes of variable "YMIN" and data item differ
In line 0008,  Error:    Reference to undefined statement label
In line 0012,  Error:    Reference to undefined statement label
In line 0016,  Error:    Wrong number of subscripts for array "STRING"


FORTRAN IV       Storage Map for Program Unit .MAIN.

Local Variables, .PSECT $DATA, Size = 000334 (   110. words)

 Name      Type      Offset    Name      Type      Offset    Name      Type      Offset
XMIN     R*4        000214   YMIN     R*4        000220   FMIN     R*4        000224
FMAX     R*4        000230   ZMIN     R*4        000244   ZMAX     R*4        000250
MAXZ     I*2        000254   K        I*2        000256   MAXF     I*2        000260
X        R*4        000262   XMAX     R*4        000266   MAXX     I*2        000272
IX       I*2        000274   MAXY     I*2        000276   IY       I*2        000300
Y        R*4        000302   YMAX     R*4        000306   IFUN     I*2        000312
MINO     I*2        000314   MAXO     I*2        000316

Local and COMMON Arrays:

 Name     Type    Section Offset       Size        Dimensions
STRING L*1 Vec    $DATA   000000   000047 (    20.) (13,3)
STAB    L*1       $DATA   000047   000144 (    50.) (100)

Statement Functions and Processor-Defined Functions Referenced:

 Name   Type   Name   Type   Name   Type   Name   Type   Name   Type   Name   Type
SCAL     R*4 FLOAT    R*4

External SUBROUTINE or FUNCTION Subprograms Referenced:

 Name   Type   Name   Type   Name   Type   Name   Type   Name   Type   Name   Type
SCOPY    R*4 LEN      I*2 REPEAT   R*4 INT      I*2 FUN      R*4 PUTSTR   R*4
EXIT     R*4

FORTRAN IV                      Tue 05-Jul-77 12:15:17                    PAGE 001

0001          FUNCTION FUN(X,Y)
0002          R=SQRT(X**2+Y**2)
0003          FUN=X*Y*R*EXP(-R))**2
***** P
0004          RETURN
0005          END

FORTRAN IV       Diagnostics for Program Unit FUN

In line 0003,  Error:    [See source listing]

FORTRAN IV       Storage Map for Program Unit FUN

Local Variables, .PSECT $DATA, Size = 000020 (     8. words)

 Name      Type      Offset    Name      Type      Offset    Name      Type      Offset
FUN      R*4 Eqv    000004   X        R*4      @ 000000   Y        R*4      @ 000002
R        R*4        000010
```

```
External SUBROUTINE or FUNCTION Subprograms Referenced:

 Name   Type   Name   Type   Name   Type   Name   Type   Name   Type   Name   Type
SQRT    R*4
```

The first part of the listing shows the main program unit and consists of the language statements up to, but not including, the function. This is followed by a diagnostics list, then by a storage map. Next the language statements comprising the function program unit are listed, again followed by a diagnostics list and a storage map.

Before considering the individual sections of the program listing, first examine the program logic to determine what this program should do. The first few lines of this program are user comment lines that briefly describe the program. Essentially, this program produces on the terminal a graph of a "three-dimensional" function, FUN(X, Y). The graph is plotted using 45 lines down and 72 characters across the terminal page. The limits of the X and Y axes are +5 and -5. The third dimension, height, is a real number within the range 0 to 1 and is represented in the listing as a number within a scale of 1 to 9. These dimensions are illustrated in Figure 9-3.

The SCAL function determines the value of the next coordinate on the graph. The statements within the DO loops calculate the coordinates using the SCAL function and determine the height value. This is done for an entire line of coordinates across the terminal

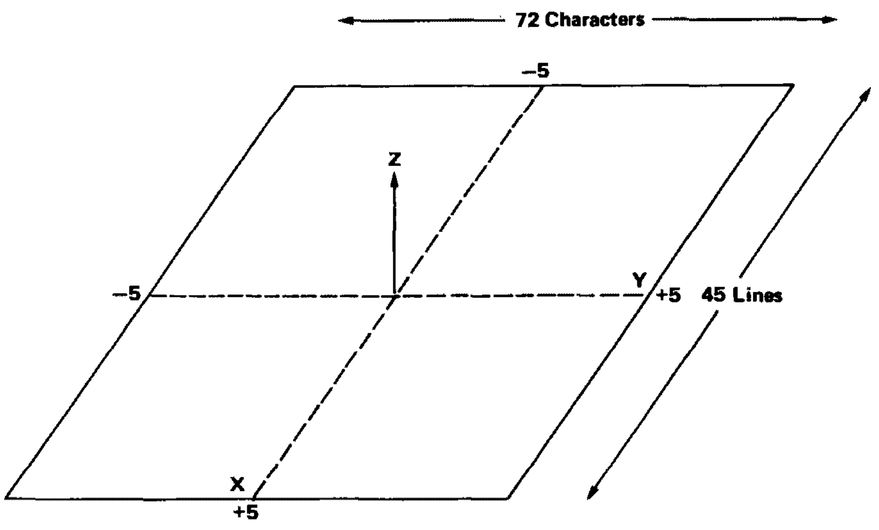


Figure 9-3 Dimensions of FUN(X, Y)

page. The entire line is then printed on the terminal using the CALL PUTSTR statement; the number 7 in this statement is the FORTRAN method of naming the terminal as the output device. This procedure is repeated until all 45 lines of the graph have been printed.

The function to be plotted is shown in the last few lines of the program. It is compiled as a separate program unit and you can edit these lines to plot any function of your choice (several alternate functions are suggested later in the chapter).

This program as it stands contains errors. The compiler detected certain error conditions during processing that prevent the program from working properly. The compiler printed appropriate messages in the diagnostics sections of the program listing.[1] Look first at the messages following the main program unit. Errors were discovered in lines 3, 4, 8, 12, and 16.

The messages for lines 3 and 4 indicate that the floating-point variables "XMIN" and "YMIN" are assigned integer values. The DATA statements must be changed. (Note that the same error exists for "XMAX" and "YMAX"; however, the compiler lists only the first error that it discovers in a line. Both "MAXX" and "MAXY" are integer variable names, so no error exists for them.) You must correct the DATA statements (lines 3 and 4), then, as follows:

```
DATA XMIN,XMAX,MAXX/-5.0,5.0,45/
DATA YMIN,YMAX,MAXY/-5.0,5.0,72/
```

The next two messages in the diagnostics section show that reference has been made from both lines 8 and 12 to an undefined label. (Line 12 is actually the second portion of line 11, the GO TO statement.) Statement label 20 is referenced in each case, but only labels 10 and 30 are shown in the program. This indicates that either a statement is missing, or that a typing error exists. Examination of the program logic shows a typing error in line 17. Label 30 should actually be 20. Correct line 17 as follows:

```
20          CALL PUTSTR(7,STRING,' ')
```

[1]Refer to the RT-11 System Message Manual for greater detail of any system messages printed.

The last message in this diagnostics section states that an incorrect number of subscripts was given for the array "STRING". Again, examination of program logic shows that the error is actually in line 2. "STRING" is really a vector (a one-dimension array), not a matrix (a 2-dimension array). Thus the comma is a typing error and line 2 should be changed as follows:

```
    LOGICAL*1 STRING(133),STAB(100)
```

Skip next to the diagnostics section for the FUN program unit. The message printed there refers you back to the source listing, to line 3. A letter "P" appears next to this line. The RT-11 System Message Manual describes a "P" error as an indication of unbalanced parentheses. Notice that the parentheses are not properly matched in this line. Thus, line 3 should be corrected as follows:

```
    FUN=(X*Y*R*EXP(-R))**2
```

This explains the errors flagged by the compiler in the diagnostics sections. Other sections of the program listing (storage map, for example) simply provide additional information that is helpful to programmers who wish to check the data types of various symbols and later make sure that object modules have been appropriately linked.

Before you can continue the exercises in this chapter, you must edit those statements in the source program that contain errors. If necessary, review the editing commands in Chapter 5. Then use the RT-11 editor to edit the file GRAPH.FOR on your system volume so that the five lines pointed out are error-free. Do not rename the file. When you are ready, recompile the program using the FORTRAN command and obtain a new object module and a new listing. This time the program should compile without error (i.e., no diagnostics should list). If diagnostics occur, you have not edited the program correctly. Compare listings and try to correct your errors or go back to the beginning of this chapter and repeat the demonstration.

LINKING OBJECT MODULES TOGETHER

The object module produced by the FORTRAN command is in itself incomplete. As mentioned earlier, it needs parts of the system library, SYSLIB, and perhaps other object modules and libraries as well, to form a complete functioning program.[1] All required object modules must be joined, or linked together, before the program can work.

[1]For more information on linking files and using library files, see Chapter 12 and 13 respectively.

Even if your program did not require any other object modules, you must still link it. In addition to joining object modules together, the link operation adjusts the object code to account for many program units being placed one after the other. The result of the link operation is a memory image load module, which is actually a picture of what computer memory looks like just prior to program execution. Figure 9-4 is a diagram of the link operation.

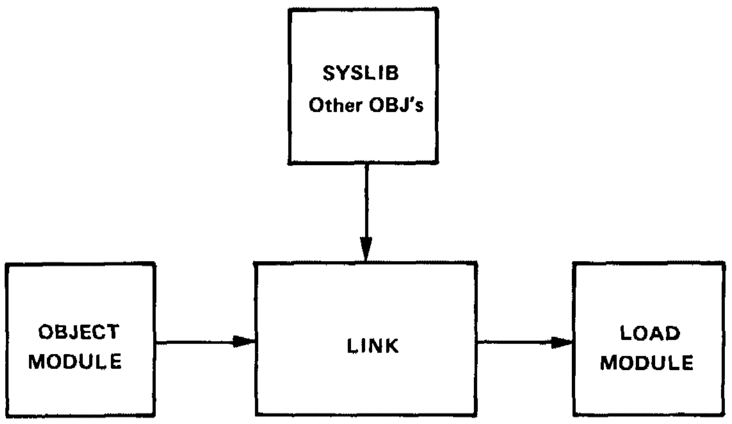


Figure 9-4 The Link Operation

LINK

To link the object modules, use the monitor LINK command. The system prompts you to enter the names of the input modules and any libraries other than the system library to be joined together. You can omit typing the .OBJ file types in the command line, since the LINK command assumes this file type for input. The system automatically assigns the file name of the first input file and a file type of .SAV to the output file. The linker will always scan the SYSLIB library if it is present on the system volume.

Long Command Format

```
.LINK (RET)
Files? GRAPH(RET)
```

Short Command Format

```
.LINK GRAPH(RET)
```

Any messages printed on the terminal identify error conditions discovered by the system during the link operation (for example, if you fail to specify all the object modules that are needed as input). However, assuming you edited your source program correctly and that it compiled without error, it should also now link without error.

A load module is one that you can run on the system. Unless your program contains logic errors that prevent it from running properly (errors which the system cannot always detect), running the .SAV version of your file should produce the results you intended. However, if logic errors exist within your program, running the program will produce either erroneous results or none at all. If this is the case, you must study the source program, rework it, reedit it, and perform the compile and link operations again.

RUNNING THE FORTRAN IV PROGRAM

If your FORTRAN program is error-free, running the .SAV version should produce the expected results. In this demonstration, running the GRAPH.SAV file should produce a graph on the terminal printer or screen.

Before you run GRAPH.SAV, you have the option of changing the output device from the terminal printer or screen to the line printer by using the monitor ASSIGN command to assign device names (see Chapter 4, Assigning Logical Names to Devices). If you prefer to print the graph on the line printer, simply assign the logical device name 7 (which is the FORTRAN code for the terminal) to the line printer code (LP:). You have designated a new output device without altering the source program. To change the device assignment to the line printer, type:

Long Command Format

```
.ASSIGN (RET)
Physical device name? LP: (RET)
Logical  device name? 7 (RET)
```

Short Command Format

```
.ASSIGN LP: 7 (RET)
```

This assignment remains in effect until you deassign the names or reboot the monitor.

RUN

Now, to execute the FORTRAN demonstration program, use the monitor RUN command. You can omit typing the .SAV file type since it is assumed within the RUN command. Type:

Long and Short Command Format

```
.RUN GRAPH (RET)
```

After a brief pause, the graph begins to print on the terminal (or line printer) and should look like the graph shown in Figure 9-5.

```
**********************************************************************************
*       111111111111111111                              111111111111111111       *
*     111111111111111111111                            111111111111111111111     *
*   11111111            11111                        11111            11111111   *
*  1111111                1111                      1111                1111111  *
* 111111     22222222222   111                      111   22222222222     111111 *
*11111     22222      2222  111                    111  2222      22222     11111*
*1111    2222      3     22  11                    11  22     3      2222    1111*
*1111   222    333333333  22  11                  11  22  333333333    222   1111*
*111    22   333       333 22 11                  11 22 333       333   22    111*
*111   222  333   4444   33 2  1                  1  2 33   4444   333  222   111*
*111   222  33   4444444  3  2 11                11 2  3  4444444   33  222   111*
*111   222  33  4444  444 33 2 11                11 2 33 444  4444  33  222   111*
*111   222  33  4444  444  3 2 11                11 2 3  444  4444  33  222   111*
*1111   222  33  44444444 33 2 11                11 2 33 44444444  33  222   1111*
*11111   222  33    444   3  2 11                11 2  3   444    33  222   11111*
*  1111    22   3333   333  2  1                  1  2  333   3333   22    1111  *
*   11111    222          22  11                  11  22          222    11111   *
*      11111     222222222  111                    111  222222222     11111      *
*          11111111      1111                        1111      11111111          *
*                   1111                                  1111                   *
*                                                                                *
*                                                                                *
*                                                                                *
*                   1111                                  1111                   *
*          11111111      1111                        1111      11111111          *
*      11111     222222222  111                    111  222222222     11111      *
*   11111    222          22  11                  11  22          222    11111   *
*  1111    22   3333   333  2  1                  1  2  333   3333   22    1111  *
*11111   222  33    444   3  2 11                11 2  3   444    33  222   11111*
*1111   222  33  44444444 33 2 11                11 2 33 44444444  33  222   1111*
*111   222  33  4444  444  3 2 11                11 2 3  444  4444  33  222   111*
*111   222  33  4444  444 33 2 11                11 2 33 444  4444  33  222   111*
*111   222  33   4444444  3  2 11                11 2  3  4444444   33  222   111*
*111   222  333   4444   33 2  1                  1  2 33   4444   333  222   111*
*111    22   333       333 22 11                  11 22 333       333   22    111*
*1111   222    333333333  22  11                  11  22  333333333    222   1111*
*1111    2222      3     22  11                    11  22     3      2222    1111*
*11111     22222      2222  111                    111  2222      22222     11111*
* 111111     22222222222   111                      111   22222222222     111111 *
*  1111111                1111                      1111                1111111  *
*   11111111            11111                        11111            11111111   *
*     111111111111111111111                            111111111111111111111     *
*       111111111111111111                              111111111111111111       *
**********************************************************************************
```

Figure 9-5 The Result of GRAPH.SAV

COMBINING OPERATIONS

EXECUTE

To produce these results, you first compiled the FORTRAN source program (GRAPH.FOR), then linked it with the default library (SYSLIB.OBJ), then ran the resulting .SAV file (GRAPH.SAV). You can combine these three operations using one monitor command, the EXECUTE command. This command instructs the system to select the appropriate language processor (which you indicate as an option), then process, link, and run the program. For example, to combine the compile-link-run operations that you performed in this chapter, you would use the following command (do not actually type this command until you have read the next section, Alternate Functions):

Long and Short Command Format

```
    .EXECUTE GRAPH/FORTRAN/LIST (RET)
```

ALTERNATE FUNCTIONS

The following are some alternate functions that you can substitute in your FORTRAN source program to produce different graphs. Simply reedit the program (GRAPH.FOR) so that lines 1-5 in the function portion at the end contain one of the following alternate functions. Then use the EXECUTE command to rerun the program. The source program compiles, links, and runs and the new graph automatically prints on the terminal (or lineprinter).

FUNCTION 1

```
    FUNCTION FUN(X,Y)
    FUN=EXP(-SQRT(X**2+Y**2))
    RETURN
    END
```

FUNCTION 2

```
    FUNCTION FUN(X,Y)
    R=SQRT(X**2+Y**2)
    FUN=X*Y*(R-3.)/(1.+EXP(3.*(R-3.5)))
    RETURN
    END
```

FUNCTION 3

```
    FUNCTION FUN(X,Y)
    FUN=EXP(+SQRT(X**2+Y**2))/1177.4
    RETURN
    END
```

SUMMARY: COMMANDS TO RUN FORTRAN PROGRAMS

EXECUTE
    Combine the compile-link-run operations into one command.

EXECUTE file/FORTRAN
    Combine the compile-link-run operations into one command, and specify the input file to be a FORTRAN file.

EXECUTE file
: Combine the compile-link-run operations into one command. Specify the libraries to be used during linking.

EXECUTE/LIST
: Combine the compile-link-run operations into one command. Obtain a listing file of the source program and print on line printer.

FORTRAN
: Compile the FORTRAN source program and produce an object module.

FORTRAN/LIST
: Compile the FORTRAN source program and produce both an object module and a listing file.

LINK
: Link individual object modules together to form a complete program and produce a load module.

RUN
: Run the indicated load module.

FILE MAINTENANCE

Before continuing further you should perform the necessary file maintenance operations. Obtain a directory of all files on your system volume that have the name GRAPH regardless of file type; these files were created as a result of the exercises in this chapter:

Long and Short Command Format

```
.DIRECTORY GRAPH.* (RET)
 05-Jul-77
GRAPH .BAK      2 24-Jun-77    GRAPH .SAV     20 05-Jul-77
GRAPH .FOR      2 05-Jul-77    GRAPH .LST      8 05-Jul-77
GRAPH .OBJ     16 05-Jul-77    GRAPH .MAP      3 23-Jun-77
 6 Files, 51 Blocks
 1231 Free blocks
```

The fact that you have corrected errors in the source file GRAPH.FOR makes the version of that file on your storage volume obsolete. Thus, transfer the updated copy from your system volume to VOL:, replacing the copy of GRAPH.FOR on the storage volume with the new version.

Long Command Format

```
.COPY (RET)
From? GRAPH.FOR (RET)
To   ? VOL:GRAPH.FOR (RET)
```

Short Command Format

```
.COPY GRAPH.FOR VOL:GRAPH.FOR (RET)
```

Next similarly transfer GRAPH.LST and GRAPH.SAV to your storage volume. This allows you to examine a listing or rerun the FORTRAN program without recompiling and relinking the source.

Long Command Format

```
.COPY (RET)
From? GRAPH.LST,GRAPH.SAV (RET)
To   ? VOL: (RET)
```

Short Command Format

```
.COPY GRAPH.LST,GRAPH.SAV VOL: (RET)
```

Once you have transferred all files of value to your storage volume, delete the useless files from the system volume (i.e., all the GRAPH files):

Long Command Format

```
.DELETE (RET)
Files? GRAPH.* (RET)
 Files deleted:
DK:GRAPH.BAK   ? Y (RET)
DK:GRAPH.SAV   ? Y (RET)
DK:GRAPH.FOR   ? Y (RET)
DK:GRAPH.LST   ? Y (RET)
DK:GRAPH.OBJ   ? Y (RET)
DK:GRAPH.MAP   ? Y (RET)
```

Short Command Format

```
.DELETE GRAPH.* (RET)
 Files deleted:
DK:GRAPH.BAK   ? Y (RET)
DK:GRAPH.SAV   ? Y (RET)
DK:GRAPH.FOR   ? Y (RET)
DK:GRAPH.LST   ? Y (RET)
DK:GRAPH.OBJ   ? Y (RET)
DK:GRAPH.MAP   ? Y (RET)
```

Finally, obtain an up-to-date directory listing of your storage volume so that you can see its current status:

Long and Short Command Format

```
 DIRECTORY VOL: (RET)
 08-Jul-77
GRAPH .FOR       2 05-Jul-77      EXAMP .FOR       2 28-Jan-77
GRAPH .LST       8 05-Jul-77      GRAPH .SAV      20 05-Jul-77
SUM   .MAC       3 13-Jun-77      EXAMP .MAC       4 25-Feb-77
 6 Files, 39 Blocks
 4723 Free blocks
```

This completes the FORTRAN demonstration. Continue to Chapter 12 to read about the linking process. If you followed the special instructions in Appendix B to load the language volume, leave this volume in device unit 0 until you have finished Chapter 12.

REFERENCES

McCracken, Daniel D., *A Simplified Guide to FORTRAN Programming.* New York: Wiley, 1974.

An introduction to programming in the FORTRAN language.

PDP-11 FORTRAN Language Reference Manual (DEC-11-LFLRA-C-D, DN1). Maynard, Mass.: Digital Equipment Corporation, 1977.

A reference manual and guide to programming in the PDP-11 FORTRAN IV language.

RT-11 FORTRAN IV Installation Guide (DEC-11-LRSIA-A-D) Maynard, Mass.: Digital Equipment Corporation, 1977.

An RT-11-specific manual that contains instructions for installing the RT-11 FORTRAN language processor, and that describes differences between versions and known problems.

RT-11 RSTS-E FORTRAN IV User's Guide (DEC-11-LRRUB-A-D). Maynard, Mass.: Digital Equipment Corporation, 1977.

An RT-11-specific manual that contains information necessary to compile, link, run, and debug a FORTRAN IV program.

CHAPTER 10
RUNNING A BASIC-11 PROGRAM

The BASIC-11 program language[1] is a machine-independent programming language that is one of the easiest languages for the beginning programmer to learn. It has both elementary language features that you use to write simple programs, and more advanced operations that allow you to produce complex and efficient programs. In addition, a special "immediate mode" lets you use BASIC like a calculator to obtain instant answers to mathematical problems.

THE BASIC-11 PROGRAMMING LANGUAGE

BASIC (Beginner's All-purpose Symbolic Instruction Code) is conversational in nature. It uses simple English keywords and common mathematical expressions to form easily-understood language statements.

You write a BASIC program as a series of one or more program lines. You begin each program line with a number that both identifies the line and indicates the order in which the line will be processed. Individual program lines contain one or more BASIC language statements that define the operations to be performed.

When you are satisfied with the logic of your BASIC source program, you create it as a file. However, unlike other programming languages that you may use, you create the file under the control of the BASIC language processor, which is part of the RT-11 operating system and is stored on your system volume or on a separate volume of its own. Thus, you use commands that are part of the BASIC language processor to create and edit the program, list it, run it, and save it for later use.

THE BASIC LANGUAGE PROCESSOR

The BASIC language processor is an interactive interpreter. It allows you to create and execute a program in its entirety or a few lines at

[1] BASIC-11 is a superset of the standard BASIC language developed at Dartmouth College.

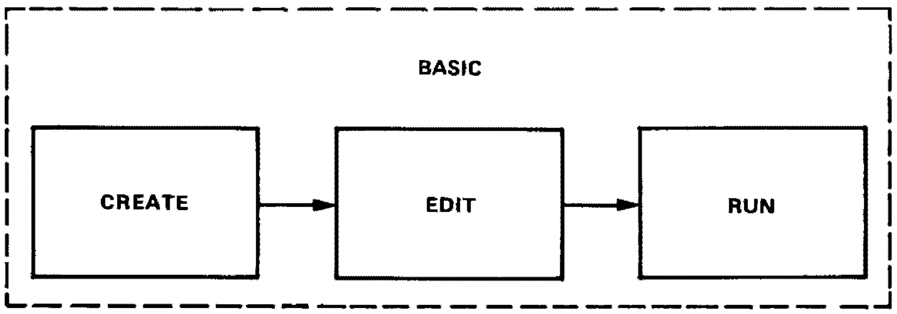


Figure 10-1 Functions of the BASIC Language Processor

a time. The interpreter examines each program language statement, interprets it, and executes it before going on to the next. If it discovers an error that prevents further processing, it prints a message on the terminal informing you of the error condition and stops. You correct the error so that execution can continue past that point, and then rerun the program.

USING THE BASIC INTERPRETER

The functions of program creation, editing, processing, and execution are all handled by the BASIC language processor. Some RT-11 systems store the BASIC interpreter (language processor) on a volume apart from the system volume.[1] You can quickly determine whether the BASIC interpreter is on your system volume by typing the monitor DIRECTORY command and specifying the BASIC.SAV program.

```
.DIRECTORY BASIC.SAV (RET)
```

Note the system response. If the directory entry for BASIC.SAV is listed on your terminal, then the required BASIC files are on your system volume and you are ready to use the interpreter. However, if BASIC.SAV did not appear in your listing, then the required files are not part of your system volume. Before you can use the interpreter, you must make a volume substitution. Read the section in Appendix B entitled "Using the FORTRAN/BASIC Language Volume".

[1]This is true for any RT-11 system volume that does not have enough free blocks to accommodate the BASIC system files. RX01 diskette is an example.

BASIC

Now use the monitor BASIC command to activate the BASIC interpreter:

Long and Short Command Format

```
.BASIC (RET)
BASIC-11/RT-11 V02-02
OPTIONAL FUNCTIONS (ALL, NONE, OR INDIVIDUAL)?
```

A prompting message is printed by BASIC. You must respond with an A, N, or I and a carriage return to indicate whether you want to preserve all, none, or some of the arithmetic functions initially provided by BASIC. BASIC's functions include operations that calculate random numbers, determine absolute values, convert octal and binary numbers to decimal, and so on. You can conserve memory space by saving only those functions that your program needs. However, for now, instruct BASIC to save all the functions by typing:

```
A (RET)
READY
```

BYE

BASIC prints the READY message to indicate that it is ready to accept a BASIC command. Any text that you type that is not preceded by a BASIC command is accepted as program (or immediate mode) input. If at any time you wish to return to the monitor command mode, simply type the BYE command following the READY message. READY appears after any completed BASIC execution, one interrupted by a double CTRL/C, or any BASIC wait condition terminated by a single CTRL/C.

NOTE

It is not necessary that you understand the BASIC language or the way the examples work to successfully perform the exercises in this chapter.

Immediate Mode

Immediate mode allows you to use the BASIC interpreter as you would a calculator to obtain immediate answers to arithmetic problems. You enter the appropriate BASIC statement keyword and any necessary mathematical formula. When you type a carriage return (the RET key), BASIC immediately calculates and prints the results. Use the terminal DELETE key and the CTRL/U command to correct any typing errors. For example, type:

```
PRINT (128+75)*3 (RET)
609
```

PRINT

BASIC adds the two numbers in parentheses, multiplies them by 3, and prints the answer. The PRINT statement causes the answer to be printed on the terminal. As another example:

```
PRINT INT(34.67) (RET)
 34

READY
```

The greatest integer less than or equal to 34.67 is printed.

You can combine several statements on a single line, or on several lines, including variable names, arithmetic equations, and data. Individual statements are separated from one another by a backslash (\) character. BASIC considers all the information, calculates the answer and prints it on the terminal. For example:

```
A=5\B=14\C=.3729 (RET)

READY
PRINT "THE HEIGHT IS";A*SIN(C)+B;"METERS" (RET)
THE HEIGHT IS 15.8216 METERS

READY
```

The first statement equates variable names with values; the second statement introduces a formula for calculating a result and prints it.

You can use immediate mode to solve fairly lengthy and complicated mathematical problems by combining statements and printing identifying messages. However, immediate mode information is temporary. You cannot save it, and you can change it only by retyping every statement line. If your needs are more complex, or if you want to save your statements, you should create a BASIC program.

Creating and Editing a BASIC Program

To create a BASIC program, you simply assign line numbers to language statements and then type the numbered statements on the terminal keyboard.

Now your program lines are saved in memory and you can transfer program control to specific lines within the program, repeat parts of the program any number of times, store the entire program for later use, and perform other similar operations that are not possible in immediate mode.

SUB

Once you have created the program, you use BASIC editing commands to list lines, change lines, add and erase lines, and correct typing errors. In addition to the DELETE key and the CTRL/U command, BASIC provides a SUB command (SUBSTITUTE) for the purpose of correcting typing errors. This command allows you to substitute new characters for existing ones in a line. For example, type:

```
10 PRINT "THIS IS A BADIC PROGRAM" (RET)
SUB 10 @BAD@BAS@ (RET)
10 PRINT "THIS IS A BASIC PROGRAM"

READY
```

The SUB command substitutes the letters BAS for BAD in line 10. Use a delimiting character (shown here as @) to separate the old text from the new. The delimiter can be any character as long as it is unique in the line. The corrected line is automatically printed by BASIC following use of the command. As another example, type:

```
15 B=10\C=5 (RET)
20 LET A-B+C\PRINT C  (RET)
```

There are two typing errors in line 20; the - should be an = and the C at the end of the line should be A. These errors can be corrected with the SUB command, as follows:

```
SUB 20 @-@=@ (RET)
20 LET A=B+C \ PRINT C

READY
SUB 20 @C@A@2  (RET)
20 LET A=B+C \ PRINT A

READY
```

The second SUB command changes the second occurrence (specified by the 2 after the last @) of C to A.

You can erase an entire line by typing the line number followed by a carriage return,

```
10 (RET)
```

DEL

or by using BASIC's DEL command[1]. Use the DEL command (DELETE) to erase a single line or several:

```
DEL 15-20 (RET)
```

This erases all numbered statement lines with numbers between and including lines 15 and 20.

LIST

To list lines of a program, BASIC provides the LIST command. First, create a few program lines:

```
5 FOR I=1 TO 10 (RET)
20 INPUT J (RET)
25 LET T=T+J (RET)
50 NEXT I (RET)
55 PRINT "THE TOTAL IS";T (RET)
88 END (RET)
```

List individual lines by specifying the line number. For example, type:

```
LIST 5 (RET)

NONAME     08-JUL-77   00:18:49

5 FOR I=1 TO 10

READY
```

LISTNH

Notice that BASIC prints a header line. Since you have not as yet assigned a name to your program, BASIC assigns it the name NONAME

[1]Do not confuse the BASIC DEL command with the DELETE key on the terminal keyboard.

and prints this name, along with the date (which is only correct if previously entered via the DATE monitor command) and the time when you use the LIST command. You can omit the header line by using the LISTNH command instead of the LIST command:

```
LISTNH 50-88 (RET)

50 NEXT I
55 PRINT "THE TOTAL IS";T
88 END

READY
```

By typing the LIST or LISTNH commands without indicating any line numbers, you can print on the terminal a listing of your entire program. Terminate the command with only a carriage return:

```
LISTNH (RET)
5 FOR I=1 TO 10
20 INPUT J
25 LET T=T+J
50 NEXT I
55 PRINT "THE TOTAL IS";T
88 END

READY
```

SCR

Finally to erase the entire program, which you must do before typing a new program, use the SCR command (SCRATCH). Type:

```
SCR (RET)

READY
```

All program lines are erased from memory.

SUMMARY: BASIC EDITING COMMANDS

line #
  Erase the indicated program lines.

DEL line #
  Erase the indicated program lines.

LIST
  List the entire program and print a header that includes the program name, date, and time.

LIST line #

List the indicated lines and print a header that includes the program name, date, and time.

LISTNH

List the entire program but do not print a header.

LISTNH line #

List the indicated lines but do not print a header.

SCR

Erase all program lines from memory and change the name to NONAME.

SUB line # @FIRST @SECOND @n

Replace the nth occurrence of the FIRST character(s) with the SECOND character(s) in the indicated line (default is n=1).

Create the following demonstration program[1] using the appropriate BASIC editing commands, exactly as it appears here. If you forget to insert a line, type it at the end or when you notice the omission; BASIC sorts and arranges lines by number prior to execution regardless of the order in which they are typed. When you are done, list the entire program and make a final check for typing errors.

```
100 REM THE PROGRAM 23 MATCHES
101 REM
110 PRINT "WE BEGIN WITH 23 MATCHES. YOU MOVE FIRST. YOU MAY TAKE"
115 PRINT "1, 2, OR 3 MATCHES. TYPE YOUR CHOICE FOLLOWED BY A CAR-"
120 PRINT "RIAGE RETURN. THEN THE COMPUTER CHOOSES 1, 2, OR 3"
125 PRINT "MATCHES. YOU CHOOSE AGAIN, AND SO ON. WHOEVER MUST"
130 PRINT "TAKE THE LAST MATCH, LOSES."
140 PRINT  \ LET M=23
200 REM THE HUMAN MOVES
201 REM
210 PRINT  \ PRINT "THERE ARE NOW";M;"MATCHES."
215 PRINT  \ PRINT "HOW MANY DO YOU TAKE";
230 INPUT H
240 IF H>M THEN 510
250 IF H<>INT(H THEN 510
260 IF H<=0 THEN 510
270 IF H>=4 THEN 510
280 LET M=M-H
290 IF M=0 THEN 410
300 REM THE COMPUTER MOVES
301 REM
305 IF M=1 THEN 440
310 LET R=M-4*INT(M/4)
320 IF R<>1 THEN 350
330 LET C=INT(3*RND)+1 \ GO TO 360
350 LET C=(R+3)-4*INT((R+3)/4)
360 LET M=M-C
370 IF M=0 THEN 440
380 PRINT  \ PRINT "THE COMPUTER TOOK";C;"....";
390 GO TO 310
400 REM SOMEBODY WON
401 REM
```

[1]23 Matches, 101 BASIC Computer Games, Maynard, Mass.: Digital Equipment Corporation, 1975.

```
410 PRINT  \ PRINT 'THE COMPUTER WON.' \ GO TO 999
440 PRINT  \ PRINT 'YOU WON.' \ GO TO 999
500 REM BAD INPUT
501 REM
510 PRINT 'ENTER ONLY 1, 2, OR 3.' \ GO TO 215
999 END
```

As you can see from the first few lines of the listing, this program is a mathematical game in which you match your logic against the program logic. The PRINT statements in the program print messages, game instructions, results, and so forth, on the terminal. The REM statements identify comment lines -- remarks that provide general information about the program, but that are ignored by BASIC during processing. The INPUT statement in line 230 allows you to supply data from the terminal. Depending on the value you enter, program control transfers to various other parts of the program. For example, if you type an illegal value, program control skips ahead to a PRINT statement in line 510 informing you of your mistake and then returns to line 215 to ask for a value again. The mathematical algorithms of this program are in lines 310 through 350 and determine the number of matches the computer will select based on your choice.

RUNNING A BASIC PROGRAM

RUN

Once you have typed the program and checked the listing to be sure that it corresponds to the example, you are ready to run it. The BASIC RUN command initiates program execution. This command prints a header that includes the program name, data, and time. If you want to omit the header line, type the RUNNH command instead.

```
RUNNH (RET)
```

If you typed the program correctly, you will see this text print on your terminal:

```
WE BEGIN WITH 23 MATCHES. YOU MOVE FIRST. YOU MAY TAKE
1, 2, OR 3 MATCHES. TYPE YOUR CHOICE FOLLOWED BY A CAR-
RIAGE RETURN. THEN THE COMPUTER CHOOSES 1, 2, OR 3
MATCHES. YOU CHOOSE AGAIN, AND SO ON. WHOEVER MUST
TAKE THE LAST MATCH, LOSES.

THERE ARE NOW 23 MATCHES.

HOW MANY DO YOU TAKE?
```

NOTE

If this response does not appear, you have not entered the program correctly. Compare your listing very carefully against the one provided earlier. Spacing does not matter, but all other characters must match. To correct your errors type CTRL/C, which, under control of BASIC only, returns you to BASIC command mode, indicated by the READY message. Correct the program and then rerun it.

When the program pauses and asks you a question, you must supply data, in this case a 1, 2, or 3. Type your choice (represented here by n) followed by a carriage return:

```
n (RET)
?SYNTAX ERROR AT LINE 250

READY
```

BASIC discovered an error[1] in line 250 that prevents further processing. Check line 250 in your listing or list it on the terminal:

```
LISTNH 250 (RET)

250 IF H<>INT(H THEN 510

READY
```

Note that a right parenthesis is missing after the second H in this line. Correct the line using the SUBSTITUTE command:

```
SUB 250 @(H@(H)@ (RET)
250 IF H<>INT(H) THEN 510

READY
```

You are ready to run the program again. Type:

```
RUNNH (RET)
```

BASIC begins processing at the start of the program.

```
WE BEGIN WITH 23 MATCHES. YOU MOVE FIRST. YOU MAY TAKE
1, 2, OR 3 MATCHES. TYPE YOUR CHOICE FOLLOWED BY A CAR-
RIAGE RETURN. THEN THE COMPUTER CHOOSES 1, 2, OR 3
MATCHES. YOU CHOOSE AGAIN, AND SO ON. WHOEVER MUST
TAKE THE LAST MATCH, LOSES.

THERE ARE NOW 23 MATCHES.

HOW MANY DO YOU TAKE?
```

[1]Refer to the RT-11 System Message Manual for greater detail of any messages printed during normal system use.

CTRL/C CTRL/C

Type your choice again. But notice this time that a different kind of error is detected. The BASIC interpreter has entered an infinite loop, a series of commands that it repeats endlessly. After several lines have printed, type a double CTRL/C; this interrupts execution and returns control to BASIC command mode.

```
r1 (RET)

THE COMPUTER TOOK 1 ....
THE COMPUTER TOOK 1 ....
THE COMPUTER TOOK 3 ....
THE COMPUTER TOOK 2 ....
THE COMPUTER TOOK 2 ....
THE COMPUTER TOOK 3 ....
THE COMPUTER TOOK 1 ....
THE COMPUTER TOOK 1 ....
THE COMPUTER TOOK 3 ....
THE COMPUTER TOOK 1 ....
THE COMPUTER TOOK 3 ....
(CTRL/C) (CTRL/C)
STOP AT LINE 380

READY
```

An infinite loop is a programming logic error. However, since the error does not prevent processing, BASIC does not print an error message. Instead BASIC is caught in a loop of instructions and executes them endlessly. This particular loop is obvious because it prints a line of text; other kinds of loops may not be so evident. At this point you must examine the program logic to determine why these instructions are being repeated.

Look at your listing of this program. The problem in this case is in line 390. This line instructs program control to return to line 310; therefore lines 310 through 390 are repeated endlessly without ever obtaining your next value choice. Program control should really return to line 210. Correct line 390 as follows:

```
SUB 390 @3@2 (RET)
390 GO TO 210

READY
```

Now you are ready to run the program again. This time the entire program should execute without error. Enter your value choices when requested. (A hint to playing the game: your first value choice determines whether you can win; if your first choice is wrong, the program has the advantage throughout.) A sample run follows.

```
RUNNH (RET)

WE BEGIN WITH 23 MATCHES. YOU MOVE FIRST. YOU MAY TAKE
1, 2, OR 3 MATCHES. TYPE YOUR CHOICE FOLLOWED BY A CAR-
RIAGE RETURN. THEN THE COMPUTER CHOOSES 1, 2, OR 3
MATCHES. YOU CHOOSE AGAIN, AND SO ON. WHOEVER MUST
TAKE THE LAST MATCH, LOSES.


THERE ARE NOW 23 MATCHES.

HOW MANY DO YOU TAKE? 1 (RET)

THE COMPUTER TOOK 1 ....
THERE ARE NOW 21 MATCHES.

HOW MANY DO YOU TAKE? 1 (RET)

THE COMPUTER TOOK 3 ....
THERE ARE NOW 17 MATCHES.

HOW MANY DO YOU TAKE? 2 (RET)

THE COMPUTER TOOK 2 ....
THERE ARE NOW 13 MATCHES.

HOW MANY DO YOU TAKE? 1 (RET)

THE COMPUTER TOOK 3 ....
THERE ARE NOW 9 MATCHES.

HOW MANY DO YOU TAKE? 1 (RET)

THE COMPUTER TOOK 3 ....
THERE ARE NOW 5 MATCHES.

HOW MANY DO YOU TAKE? 3 (RET)

THE COMPUTER TOOK 1 ....
THERE ARE NOW 1 MATCHES.

HOW MANY DO YOU TAKE? 0 (RET)
ENTER ONLY 1, 2, OR 3.

HOW MANY DO YOU TAKE? 1 (RET)

THE COMPUTER WON.

READY
```

SUMMARY: BASIC EXECUTION COMMANDS

RUN

Execute the BASIC program currently in memory; print a header line including the program name, date and version number.

RUNNH

Execute the BASIC program currently in memory; omit the header line.

CTRL/C

Under control of BASIC only, interrupt execution of the BASIC program and return control to BASIC command mode.

BYE
: Return control to monitor command mode (only when using BASIC).

FILE MAINTENANCE

SAVE

You can transfer the BASIC program currently in memory to a storage volume by using the SAVE command of BASIC. The SAVE command copies the program to the storage volume giving it the file name and file type that you indicate in the command line. A file type of .BAS is assigned automatically unless you indicate otherwise.

Use the SAVE command to store this BASIC program as MATCH.BAS on the storage volume (VOL:) as follows:

```
SAVE VOL:MATCH (RET)

READY
```

NEW

After you save a BASIC program on a storage volume, you can create a new program in memory by typing the BASIC NEW command. This command erases the current memory contents and asks you for a new program name:

```
NEW (RET)
NEW FILE NAME--
```

Type any file name you wish and BASIC assigns it to the file you create. Or you can respond by typing only a carriage return; BASIC then assigns the file name NONAME.

Another way to create a new program in memory is to type the BASIC SCR command. This command simply erases the current memory contents. It assigns the name NONAME:

```
SCR (RET)

READY
```

OLD

To use an existing BASIC program, one that you have previously stored on a storage volume, type the BASIC OLD command:

```
OLD (RET)
OLD FILE NAME--
```

Reply by typing the device name, file name and file type of the file that you want to use. If you omit an explicit device name, BASIC assumes DK: (the default volume), and if you omit an explicit file type, BASIC assumes .BAS. BASIC erases memory and then copies the program from the volume into memory. For example, type:

```
MATCH (RET)
READY
```

This copies DK: MATCH.BAS back into memory.

Assume that you have edited or changed the MATCH.BAS file and now want to transfer it back to VOL:. Since the file already exists as MATCH.BAS on that volume, you must use the BASIC REPLACE command:

```
REPLACE VOL:MATCH (RET)

READY
```

REPLACE

The REPLACE command replaces an existing file with a new version.

The SAVE and REPLACE commands copy a BASIC program from computer memory to a storage volume. As these commands copy the program, they convert it from the internal format used by BASIC to ASCII format. Thus, you can, if you prefer, use the RT-11 editor to create and edit BASIC programs, since the editor also uses ASCII format. However, many users would rather use BASIC to create and edit a BASIC program, since they can then run the program, reedit it, rerun it, and save the new version, all in BASIC command mode, rather than perform the several corresponding monitor commands.

The last file maintenance operation that you should perform is to obtain an up-to-date directory of your storage volume so that you can see its current status; however, you must return to monitor command mode to do this. Type the BYE command; this BASIC command (rather than CTRL/C) returns control to monitor command mode. Next use the DIRECTORY monitor command to check the status of your storage volume.

```
BYE (RET)

.DIRECTORY/BRIEF VOL: (RET)
 08-Jul-77
GRAPH .FOR    EXAMP .FOR    GRAPH .LST    GRAPH .SAV    SUM   .MAC
EXAMP .MAC    MATCH .BAS
 7 Files, 40 Blocks
 4722 Free blocks
```

SUMMARY: BASIC FILE MAINTENANCE COMMANDS

NEW
: Create a new BASIC program, assigning the file name indicated.

OLD
: Copy into memory (for use under BASIC) an existing BASIC program.

REPLACE
: Copy the BASIC program currently in memory to the indicated storage volume, replacing the version that already exists on that volume.

SAVE
: Copy the BASIC program currently in memory to the indicated storage volume.

This completes the BASIC demonstration. Before you continue to Chapter 14 to learn about program debugging, make sure that the main system volume is loaded in device unit 0. If you followed the special instructions in Appendix B to load the language volume, you should now stop the system, unload that volume, load the main system volume, and rebootstrap the system.

REFERENCES

BASIC-11 Language Reference Manual (DEC-11-LIBBB-A-D). Maynard, Mass.: Digital Equipment Corporation, 1976.

A reference manual and guide to programming in the BASIC-11 language.

BASIC-11/RT-11 Installation Guide (DEC-11-LIBTA-A-D). Maynard, Mass.: Digital Equipment Corporation, 1977.

An RT-11-specific manual that contains instructions for installing the RT-11 BASIC language processor and that lists known problems and differences between versions.

BASIC-11/RT-11 User's Guide (DEC-11-LIBUA-A-D). Maynard, Mass. Digital Equipment Corporation, 1977.

An RT-11-specific manual that contains information necessary to create, edit, run and debug a BASIC program.

CHAPTER 11

RUNNING A MACRO-11 ASSEMBLY LANGUAGE PROGRAM

The MACRO-11 programming language is a machine-dependent programming language developed for the PDP-11 programmer, or for the FORTRAN IV programmer who intends to combine assembly language routines and FORTRAN routines. ·The MACRO-1.1 language enables the knowledgeable programmer to access all the features of the RT-11 computer system using a precise and efficient programming code.

THE MACRO-11 ASSEMBLY LANGUAGE

The MACRO-11 assembly language uses the PDP-11 instruction set, a list of mnemonic instructions that correspond to various PDP-11 computer operations. These instructions allow you to add, compare, increment, complement, and perform many other manipulations on numerical data. The instructions are summarized in a pocket-sized folding card, called the PDP-11 Programming Card (Figure 11-1), and are described in detail in the PDP-11 Processor Handbook. By choosing the appropriate instructions, and by providing any additional data needed, you can create a complete program.

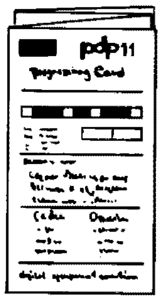


Figure 11-1 PDP-11 Programming Card

You write the MACRO-11 program as a sequence of lines, each a single assembly language statement in the following format:

```
LABEL:      OPERATOR OPERAND(S)      COMMENTS
```

The operator and/or operand are either instructions selected from the PDP-11 instruction set, data needed by the instructions, or assembler directives (instructions to the assembler to guide the assembly process). The optional statement label identifies the statement line so that you can refer to the instructions or data on that line from other parts of the program. Optional comments describe generally what operations are being done. Sequences of language statements constitute a routine (to perform a specific function); groups of routines and data compose the entire executable program.

When you are satisfied with the logic of your MACRO-11 source program, you use the RT-11 editor to create it as a file (like you did in Chapter 5). You use tabs and spaces to make the program more readable. When you have finished creating the program as a complete, edited file, you next enter it as input to the MACRO-11 language processor, which is part of the RT-11 operating system and is stored on your system volume. The MACRO-11 language processor processes (assembles) the language statements, converting them into an internal machine language code called object code. This code is next processed by the system linker, which combines your program units, making the program suitable for execution. Figure 11-2 represents the development of an executable MACRO-11 program.

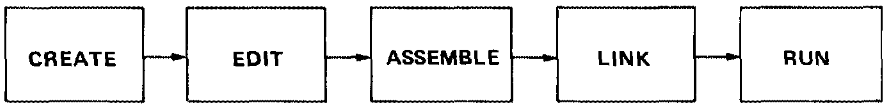


Figure 11-2 Evolution of a MACRO-11 Program

THE MACRO-11 LANGUAGE PROCESSOR

The MACRO-11 language processor is an assembler that accepts information in one format (i.e., your source program) and translates it into another format (i.e., a machine language program). The assembler interprets and processes the assembly language statements, one at a time, and generates one or more computer instructions or data items. Since you originally use the editor to create a MACRO-11 program in ASCII format, you must next translate it into a machine format that the computer can use. The MACRO-11 assembler performs this conversion, producing as output a new version of the program in object format, called an object module. You may request the MACRO assembler to produce a listing of the source program at the same time. The role of the assembler is represented below in Figure 11-3.

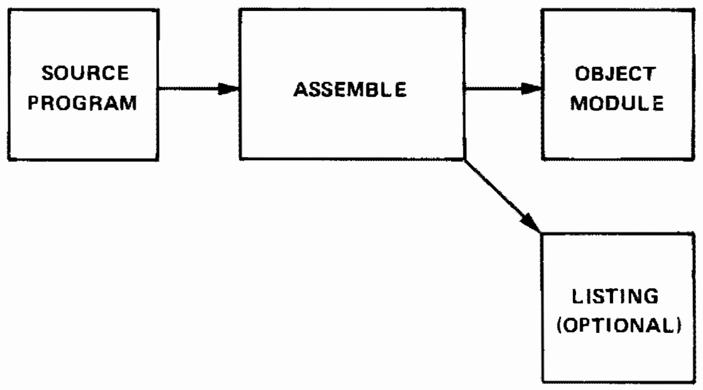


Figure 11-3 Function of a MACRO-11 Assembler

During assembly processing, the MACRO-11 assembler

- Accounts for all instructions used within the source program and determines their relative positions in computer memory; it does this by means of a storage location (program) counter

- Keeps track of all user-defined symbols and their respective values in a symbol table

- Converts assembly language mnemonics, user-defined symbols, and data values into their respective machine language (object code) equivalents

The Program Counter

The function of the program counter is to keep track of addresses in computer memory where instructions and data will be stored.

PDP-11 computer memories are composed of physical storage locations which can hold numerical data. These locations are numbered consecutively from 0 up to the highest memory location, which varies according to the amount of memory acquired with the computer system (see Figure 11-4). PDP-11 computers used in an RT-11 system have at least 16,384 bytes (8,192 words); most RT-11 systems have more than that number.

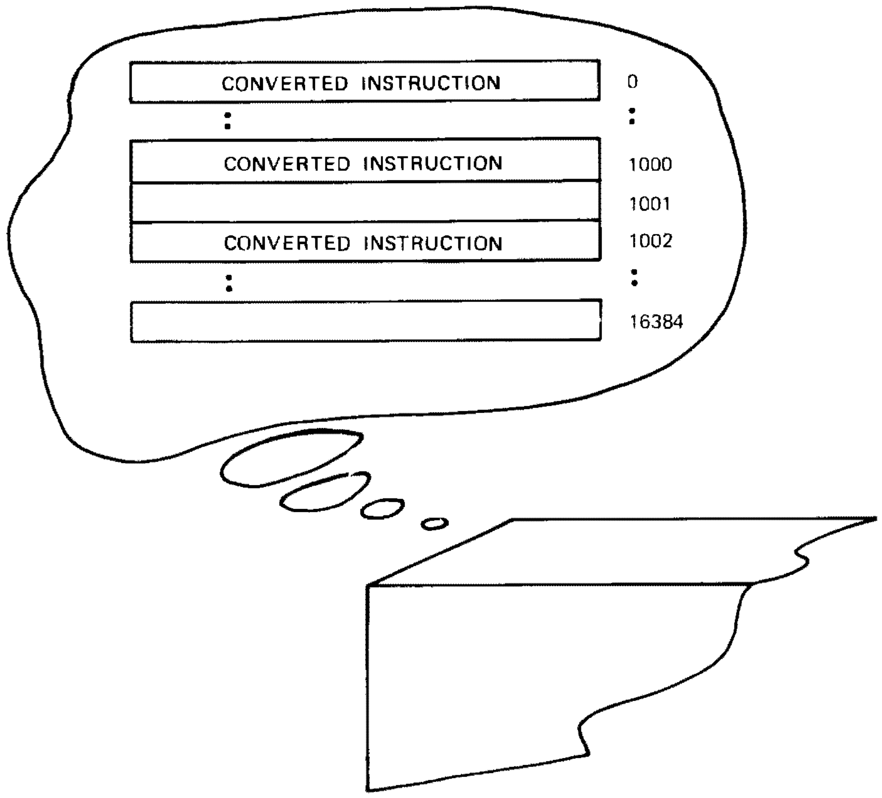


Figure 11-4 PDP-11 Computer Memory

During processing, the assembler converts each program language statement into numerical data (the object code) and assigns the data a relative storage location. The system linker will convert the relative storage locations assigned by the assembler to absolute storage locations in the computer memory[1]. The location's associated number is called its address. As the assembler translates and assigns each statement, it updates the value of the program counter accordingly.

The Symbol Table

Since you may not know which locations, or how many locations, the program needs, you use symbolic names (variables, constants, and labels) to represent individual locations and their contents. As the assembler processes the source program, it constructs a symbol table, which is a compiled list of all the symbolic names and labels that you have used within the program. The MACRO-11 assembler

[1]The system linker is discussed in Chapter 12.

defines each symbolic name by assigning an address or data value, as appropriate, and adds the symbol definition to the symbol table. After assembly, you can refer to the symbol table, which is printed at the end of the assembly listing, to find all symbol definitions.

Machine Language Code

The third function of the assembler is to convert your MACRO-11 source language statements into machine language code (the object module).

NOTE

The following information will aid your understanding of the assembly listing used later in this chapter.

Machine language code is numerical data in the form of binary numbers (numbers composed of only the digits 0 and 1). Binary numbers are appropriate because the digits 0 and 1 can be easily manipulated by the two-state electronic logic of the computer.

For example, a typical assembled instruction in PDP-11 computer memory looks like this:

```
location      location
address       contents
   .             .
   .             .
 01000        11000000
 01001        11100101
   .             .
   .             .
```

Since a single instruction requires two (or more) consecutive memory locations, the instruction is actually put together in memory in the following manner:

```
01001   1 1 1 0 0 1 0 1   1 1 0 0 0 0 0 0   01000
```

Each individual digit of the instruction is called a bit (binary digit). A single memory location, called a byte, contains 8 bits; two memory locations, called a PDP-11 word, contain 16 bits.

The byte in the even-numbered memory address is called the low-order byte and is stored first; the byte in the odd-numbered memory address is called the high-order byte and is stored next. Both bytes together form one PDP-11 16-bit word (Figure 11-5).

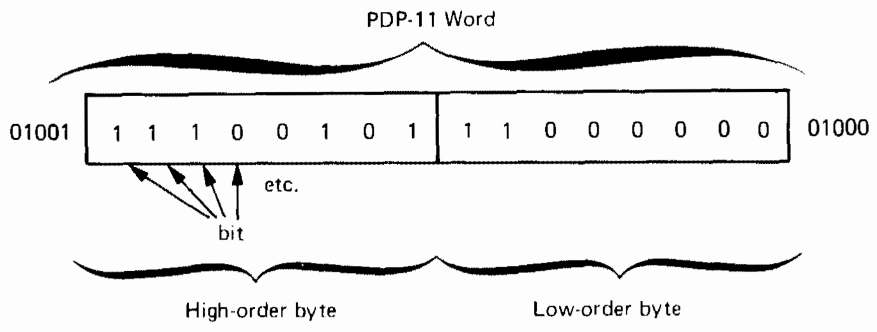


Figure 11-5 PDP-11 Word

The computer works in terms of 8-bit bytes and 16-bit words of binary data. However, binary numbers are generally too long and cumbersome to be used effectively by a programmer. But binary numbers can be easily represented as octal numbers (numbers composed of digits within the range 0 to 7). Since octal numbers are closer to the familiar decimal number system and are much more readable than binary numbers, the programmer more often uses octal numbers than binary numbers.

Table 11-1 shows the decimal numbers 0 through 10 and their respective octal and binary equivalents. Tables and formulas are available to calculate higher conversions (see the RT-11 Advanced Programmer's Guide for one such table).

Table 11-1 Decimal/Octal/Binary Conversion

Decimal	Octal	Binary
0	0	000
1	1	001
2	2	010
3	3	011
4	4	100
5	5	101
6	6	110
7	7	111
8	10	1 000
9	11	1 001
10	12	1 010

Thus, you can think of the binary instruction shown earlier in terms of its octal equivalent as follows (conversion is done from low-order to high-order byte in groups of three bits):

```
01001   1 1 1 0 0 1 0 1   1 1 0 0 0 0 0 0   01000

        1   6     2     7     0     0   =   162700(8)
```

A MACRO-11 assembly listing shows the addresses of memory locations and their contents as octal numbers. The octal numbers represent the respective binary machine language code that makes up the object module.

ASSEMBLING THE MACRO-11 PROGRAM

In Chapter 5 you used the RT-11 editor to create a MACRO-11 source program; you then stored it on your storage volume. Since a source program is in ASCII format, the next step is to use the MACRO-11 assembler to convert it to object code.

Copy the MACRO source program from the storage volume back to the system volume (which is the default volume for input/output operations).

On your storage volume are two MACRO source programs, the one you created (SUM.MAC) and the one provided for you (EXAMP.MAC). Which of these you should copy depends on the results of the source comparison you performed in Chapter 6. If the comparison resulted in NO DIFFERENCES ENCOUNTERED, copy your own program (SUM.MAC) as follows:

Long Command Format

```
.COPY (RET)
From? VOL:SUM.MAC (RET)
To   ? SUM.MAC (RET)
```

Short Command Format

```
.COPY VOL:SUM.MAC SUM.MAC (RET)
```

However, if the FILES ARE DIFFERENT message was generated, substitute EXAMP.MAC:

Long Command Format

```
.COPY (RET)
From? VOL:EXAMP.MAC (RET)
To   ? SUM.MAC (RET)
```

Short Command Format

```
.COPY VOL:EXAMP.MAC SUM.MAC (RET)
```

Whichever source file you copied now resides on your system volume under the name SUM.MAC and is the file that you will process with the MACRO-11 assembler. The command used to assemble a MACRO source program is the monitor MACRO command.

MACRO

Use the MACRO command with its /LIST and /CROSSREFERENCE options to assemble your source program and produce a cross-referenced assembly listing. The system prompt asks you to supply the input file name. You can omit typing the .MAC file type since the MACRO command assumes this file type unless you indicate otherwise. The system will automatically assign the name SUM.OBJ to the object module and SUM.LST to the listing file and store both newly-created files on the system volume.

Long Command Format

```
.MACRO (RET)
Files? SUM/LIST/CROSSREFERENCE (RET)
```

Short Command Format

```
.MACRO SUM/LIST/CROSSREFERENCE (RET)
```

Assembly begins. When it is finished, a message similar to the following prints on the terminal printer or screen:

```
ERRORS DETECTED:  6
```

This message indicates the number of lines in which the assembler detected errors during processing. In this case, the assembler found six lines in your source program with errors. It helps at this point to look at the listing produced by the assembler for information.

Long Command Format

```
.MACRO/LIST (RET)
Files? SUM (RET)
ERRORS DETECTED:  6
```

Short Command Format

```
.MACRO/LIST SUM (RET)
ERRORS DETECTED:  6
```

Your listing should look like the following example. An explanation of this listing follows. You should refer to the listing as you read the accompanying explanation.

NOTE

It is not necessary that you understand the MACRO-11 language or the way this program works to successfully complete the exercises in this chapter.

```
SUM.MAC VERSION 1      MACRO V03.00 8-JUL-77 01:16:17 PAGE 1

      1                                        .TITLE SUM.MAC  VERSION 1
      2
      3                                        .MCALL .TTYOUT, .EXIT, .PRINT
      4
      5
      6
      7          000106                        N = 70.           ;NO. OF DIGITS OF 'E' TO CALCULATE
      8
      9                                ;       'E' = THE SUM OF THE RECIPROCALS OF THE FACTORIALS
     10                                ;       1/0! + 1/1! + 1/2! + 1/3! + 1/4! + 1/5! + ...
     11
M    12 000000                         EXP:    .PRINT  #MESSAG         ;PRINT INTRODUCTORY TEXT
     13 000006  012705  000106                 MOV     #N,R5           ;NO. OF CHARS OF 'E' TO PRINT
     14 000012  012700  000107         FIRST:  MOV     #N+1,R0         ;NO. OF DIGITS OF ACCURACY
U    15 000016  012701  000000                 MOV     #A,R1           ;ADDRESS OF DIGIT VECTOR
     16 000022  006311                 SECOND: ASL     @R1             ;DO MULTIPLY BY 10 (DECIMAL)
     17 000024  011146                         MOV     @R1,-(SP)       ;SAVE *2
     18 000026  006311                         ASL     @R1             ;*4
     19 000030  006311                         ASL     @R1             ;*8
     20 000032  062621                         ADD     (SP)+,(R1)+     ;NOW *10, POINT TO NEXT DIGIT
     21 000034  005300                         DEC     R0              ;AT END OF DIGITS?
     22 000036  001371                         BNE     SECOND          ;BRANCH IF NOT
     23 000040  012700  000106                 MOV     #N,R0           ;GO THRU ALL PLACES, DIVIDING
     24 000044  014103                 THIRD:  MOV     -(R1),R3        ;BY THE PLACES INDEX
     25 000046  012702  177777                 MOV     #-1,R2          ;INIT QUOTIENT REGISTER
     26 000052  005202                 FOURTH: INC     R2              ;BUMP QUOTIENT
     27 000054  160003                         SUB     R0,R3           ;SUBTRACT LOOP ISN'T BAD
     28 000056  103375                         BCC     FOURTH          ;NUMERATOR IS ALWAYS < 10*N
     29 000060  060003                         ADD     R0,R3           ;FIX REMAINDER
     30 000062  010311                         MOV     R3,@R1          ;SAVE REMAINDER AS BASIS
     31                                                                ;FOR NEXT DIGIT
AR   32 000064  066167  000000  000000         ADD     R2-2(R1)        ;GREATEST INTEGER CARRIES
     33                                                                ;TO GIVE DIGIT
     34 000072  005300                         DEC     R0              ;AT END OF DIGIT VECTOR?
     35 000074  001363                         BNE     THIRD           ;BRANCH IF NOT
     36 000076  014100                         MOV     -(R1),R0        ;GET DIGIT TO OUTPUT
     37 000100  162700  000012         FIFTH:  SUB     #10.,R0         ;FIX THE 2.7 TO .7 SO
     38                                                                ;THAT IT IS ONLY 1 DIGIT
     39 000104  103375                         BCC     FIFTH           ;(REALLY DIVIDE BY 10)
     40 000106  062700  000070                 ADD     #10+'0,R0       ;MAKE DIGIT ASCII
U    41 000112  000000                         .TTYON                  ;OUTPUT THE DIGIT
     42 000114  005011                         CLR     @R1             ;CLEAR NEXT DIGIT LOCATION
     43 000116  005305                         DEC     R5              ;MORE DIGITS TO PRINT?
     44 000120  001334                         BNE     FIRST           ;BRANCH IF YES
     45 000122                                 .EXIT                   ;WE ARE DONE
     46
M    47 000124  000107                 EXP:    .REPT   N+1
     48                                        .WORD   1               ;INIT VECTOR TO ALL ONES
     49                                        .ENDR
     50
     51 000342     124     110     105 MESSAG: .ASCII  /THE VALUE OF E IS:/ <15><12> /2./ <200>
        000345     040     126     101
        000350     114     125     105
        000353     040     117     106
        000356     040     105     040
        000361     111     123     072
        000364     015     012     062

SUM.MAC VERSION 1      MACRO V03.00 8-JUL-77 01:16:17 PAGE 1-1

        000367     056     200
     52                                        .EVEN
     53
D    54          000000'                       .END    EXP
```

```
SUM.MAC VERSION 1        MACRO V03.00 8-JUL-77 01:16:17 PAGE 1-2
SYMBOL TABLE

A      = ******         FIFTH    000100R         FOURTH  000052R   N      = 000106      THIRD   000044R
EXP      000000R        FIRST    000012R         MESSAG  000342R   SECOND   000022R     .TTYON= ******

. ABS.  000000      000
        000372      001
ERRORS DETECTED:  6

VIRTUAL MEMORY USED:  537 WORDS  ( 3 PAGES)
DYNAMIC MEMORY AVAILABLE FOR  65 PAGES
DK:SUM,LP:SUM=DK:SUM/C

SUM.MAC VERSION 1        MACRO V03.00 8-JUL-77 01:16:17 PAGE S-1
CROSS REFERENCE TABLE (CREF V01-05 )


.TTYON    1-41
A         1-15
EXP       1-12#     1-47#     1-54
FIFTH     1-37#     1-39
FIRST     1-14#     1-44
FOURTH    1-26#     1-28
MESSAG    1-12      1-51#
N         1-7#      1-13      1-14      1-23      1-47
SECOND    1-16#     1-22
THIRD     1-24#     1-35

SUM.MAC VERSION 1        MACRO V03.00 8-JUL-77 01:16:17 PAGE M-1
CROSS REFERENCE TABLE (CREF V01-05 )


.EXIT     1-3#      1-45
.PRINT    1-3#      1-12
.TTYOU    1-3#


SUM.MAC VERSION 1        MACRO V03.00 8-JUL-77 01:16:17 PAGE E-1
CROSS REFERENCE TABLE (CREF V01-05 )


A         1-32
D         1-54
M         1-12      1-47
R         1-32
U         1-15*     1-41
```

This listing was printed on an 132-column line printer. The first part of the listing has four logical sections, as follows:

line number	octal memory address	octal instruction value(s)	statement line

The assembler assigns consecutive decimal line numbers to each line of the source program, including blank lines and comment lines. These numbers are used for reference purposes. The next column to the right shows the relative[1] even-numbered octal memory (byte) addresses of storage locations assigned by the program counter to each instruction in the program. This program has been assigned relative memory addresses 0 through 370. The third column (and possibly fourth and fifth) shows the octal equivalent of the assembled instruction or data value. An apostrophe following an octal value indicates a relative value that must be modified before it can be used (the actual value is determined during linking). Finally, the source program as you created it appears in the right-hand portion of the listing.

[1]The assembler assigns relative memory addresses to instructions. Actual addresses are not determined until the link operation is performed. Linking and address relocation are discussed in Chapter 12.

For example, look at line 19 of the listing:

```
     19 000030  006311                          ASL     @R1                     ;*8
```

The instruction ASL @R1 is stored in relative memory locations 30 and 31 as binary data (the comment, ;*8, is ignored):

```
31   0 0 0 0 1 1 0 0   1 1 0 0 1 0 0 1    30

     0  0     6     3     1     1
```

Some instructions require more than two memory locations, for example, those at lines 13 and 14. The number of memory locations required depends upon the operation.

Following the assembled code in the listing is the symbol table, an alphabetic listing of user-defined symbols and labels in the program and their respective definitions. Symbols are defined as values. For example, the symbolic variable name N is defined (in line 7) as 000106(octal) or 70(decimal), an absolute value. Labels are defined as addresses. The symbolic label FIRST is defined (in line 14) as 00012, a relocatable address (the R following 00012 in the symbol table indicates that the address will be relocated or modified during linking.) A row of asterisks next to any symbolic name in the table indicates that for some reason (possibly a programming error) the assembler could not define the symbol.

At the very end of the symbol table (where the . ABS. occurs) is the program's size information (or synopsis) in terms of the total number of octal storage locations it requires (in this case, 372). Following is the number of errors detected, and the amount of free and used memory pages (statistics provided by the assembler).

Following the symbol table is the cross reference (CREF) listing. The CREF listing is optional (as is the assembly listing) but provides you with useful reference and debugging information, especially if the program is large. The CREF listing can contain several kinds of tables of reference information, each beginning on a new page. The default tables are the three shown here.

Every reference in a CREF table shows the page number of the listing (in the preceding example, all references are on page 1), followed by the appropriate line number. A number sign following a line number indicates that this line is where a label or symbol definition occurs.

The first CREF table shown here lists alphabetically all user-defined symbol and label references.

The second CREF table lists alphabetically all macro symbol references. (Macro symbols are a special feature of the MACRO-11 assembly language; they are described shortly.)

The third CREF table lists alphabetically the codes of the errors detected during assembly. These errors must be corrected before you can run the program.

Now that you are familiar with the format of an assembly listing, go back to the beginning of the example listing to determine what this program should do.

The first two comment lines (preceded by semicolons) indicate that the program calculates the value of 'E', which is the sum of the inverse of the factorials between 1 and infinity. The algorithm used in this program is somewhat complicated (this was necessary to keep the program reasonably short). 'E' is calculated one digit at a time by using a difference function between its actual value and the current approximation for each new digit. The program forms:

$$1+(1+(1+\ldots+(1+((1+(1/N))/(N-1))/N-2))/\ldots/2)/1)$$

and is 2.11111. . . in the inverse factorial base system, which is the first sum shown in the program listing.

The statements at lines 1 through 7 define initial states to the assemblers such as the value of N, and designate the macros that will be used throughout the program.

Macros, from which the MACRO-11 language processor derives its name, are a very important and useful feature of the MACRO-11 assembly language. You can define as a macro any recurring sequence of coding instructions. By giving the macro a name, you can thereafter call it by name from any other part of the program using a single language statement.

In addition to user-defined macros, the RT-11 system provides system macros that your programs can access. System macros are defined in a special system library file called SYSMAC.SML (SML stands for System Macro Library). SYSMAC.SML is part of the RT-11 operating system and is stored on the system volume. If you request a system macro from your source program, the MACRO-11 assembler automatically searches SYSMAC.SML for the required information.

The system macros defined in SYSMAC.SML are calls to certain services performed by the RT-11 monitor such as terminal handling, input and output operations, program termination, file capabilities, and so on. The portion of the monitor that performs or is capable of getting the necessary program code to perform these services is always in memory and therefore is called the resident monitor. Thus, whenever your source program is in memory to be executed, the resident monitor is also there with its available services.

You communicate the need for a monitor service by issuing a programmed request in your source program. A programmed request consists simply of a macro call to a specific macro defined in SYSMAC.SML. The macro expands into the appropriate machine language code, which, during program execution, makes a request to the resident monitor to supply the desired service.

You specify all programmed requests that you intend to use in your source program in an .MCALL statement, like that shown at line 3 in the listing. For example, the programmed request .TTYOUT requests the monitor to print an ASCII character on the console terminal. During assembly, the .TTYOUT macro in SYSMAC.SML is expanded into machine language code. During program execution this code requests the resident monitor to take the indicated ASCII character and send it to the console terminal.

Line 12 in the program uses another programmed request, .PRINT, to print a message on the terminal.

Lines 13 through 15 are initialization instructions: they set initial values in three of the special registers. Lines 16 through 22 are a routine that does a multiplication by 10. Lines 23 and 24 are setup instructions for the division routine of lines 25 through 28. Lines 29 through 35 save the quotient and remainder. Lines 36 through 40 print the digits of E. Lines 43 and 44 count the number of digits.

The statements at lines 47 through 49 reserve a buffer area (a series of locations in memory) to be used by the program and therefore not to be assigned to other instructions. The statement at line 51 provides the data for printing the ASCII text message THE VALUE OF E IS: 2.

This program, however, contains errors. The assembler discovered six lines with errors that prevent the program from assembling properly. The assembler flags (points out) errors by printing a code letter in the assembly listing or on the terminal if no listing is requested.[1]

The first error occurs at line 12 and is an M error. This means a label was defined more than once. You can refer to a label any number of times, but you may define it only once. By looking at the CREF user symbol table, you can see that the label is defined at line 12 and again at line 47; one of these definitions is wrong. Examination of the program logic reveals that the definition at line 12 is correct. Before deciding how to change line 47, though, check the other errors to see if one of them indicates what should be done. In fact, the next error encountered (line 15) shows what is wrong. A U error identifies an undefined symbol. The label A is referenced in line 15, but is never defined within the program. It should logically be defined at line 47. Therefore, line 47 should be changed to read:

```
A:          .REPT    N+1
```

Thus, this one change eliminates three errors flagged by the assembler; those at lines 12, 15, and 47.

The next error occurs at line 32. Actually, the assembler flagged two errors here. An A error indicates an addressing problem and an R error indicates a register error (illegal use of a register, a special PDP-11 storage feature). If you look at the language statement in line 32, you can see that the ADD operator is followed by one operand. However, ADD is an instruction that requires two operands (two values to be added together) separated by a comma. This statement simply contains a typing error which can be corrected by inserting a comma between the R2 and the -2(R1). Thus, changing the line as follows both corrects the addressing problem and eliminates the illegal register expression:

```
ADD         R2,-2(R1)
```

[1]Refer to the RT-11 System Message Manual for greater detail on any system messages printed during normal system use.

At line 41 is another undefined symbol, the macro symbol .TTYON. Since the program designated the macro symbol .TTYOUT in line 3, this error indicates a misspelling. Correct line 41 to read:

```
    .TTYOUT
```

Finally, a D error occurs in line 54. This indicates that reference was made to a symbol that is defined more than once. This error has alrady been eliminated as a result of the correction made to line 47.

Thus, by changing the three lines indicated, you can correct all the errors flagged during assembly. So the next step is to edit the appropriate lines in the source program. If necessary, review the editing commands in Chapter 5, and then edit the file SUM.MAC on your system volume so that the three lines indicated are error-free. Do not rename the file. When you are ready, reassemble the program using the MACRO command and obtain a new object module and a new listing. This time the program should assemble without error. If errors occur, you have not edited the program correctly. Compare listings and try to correct your errors or go back to the beginning of this chapter and repeat the demonstration.

LINKING OBJECT MODULES TOGETHER

The object module produced by the MACRO command may in itself be incomplete. It may need to be joined with other object modules or library files to form a complete functioning program[1], since all required object modules must be joined before the program can work.

Thus you must next link the SUM object module with any other object modules it requires. However, the only file used by this program was the macro library file SYSMAC.SML, and it was used during assembly. So in this case, you do not need to join the SUM object module with any other modules. However, you must still link the file. The link operation, in addition to joining object modules together, also assigns absolute memory addresses to the relative addresses calculated by the MACRO-11 assembler. Since the memory addresses of one object module must be relocated to accommodate addresses used in another object module, the link operation serves to resolve all address changes. The result of the link is a memory image

[1]For more information on linking files and using library files, see Chapters 12 and 13 respectively.

load module, with all module links resolved and all absolute memory addresses and storage information assigned (Figure 11-6). The memory image module, then, is actually a picture of what computer memory looks like just prior to program execution.

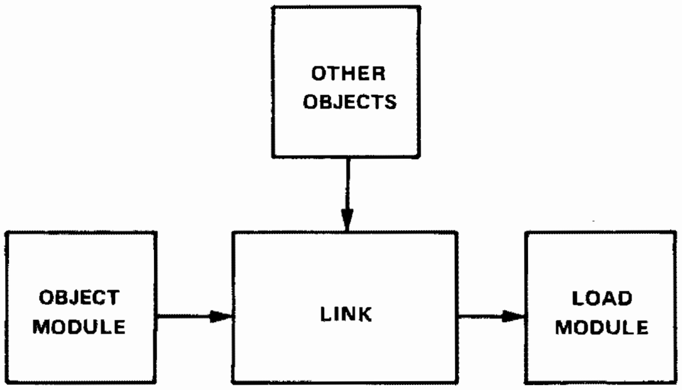


Figure 11-6 The Link Operation

LINK

To link the object modules, use the LINK command. The system prompts you to enter the names of the input object modules to be linked together. You can omit typing the .OBJ file type in the command line since the LINK command assumes this file type for input. After you have entered the input information, the system begins linking the object module. You do not have to specify an output file, since the system automatically assigns the file name of the first input file and a file type of .SAV to the output file.

Long Command Format

```
.LINK (RET)
Files? SUM (RET)
```

Short Command Format

```
.LINK SUM (RET)
```

Any messages printed inform you of error conditions discovered during the link operation (for example, if you fail to specify all the necessary input object modules that are needed). However, assuming you edited your source program correctly and that it assembled without error, it should also now link without error.

A load module is one that you can run on the system. Unless your program contains logic errors that prevent it from running properly (errors which the system cannot always detect), running the .SAV version of your program should produce the results you intended. However, if logic errors exist within your program, running the program will produce either erroneous results or none at all. If this is the case, you must study the source program, rework it, reedit it, then perform the assembly and link operations again.

RUNNING THE MACRO-11 PROGRAM

If your MACRO program is error-free, running the .SAV version should produce the expected results. In this demonstration, running the SUM.SAV file should produce a value on the terminal that is the constant E (2 followed by 70 digits).

To execute the MACRO demonstration program, use the monitor RUN command. You can omit typing the .SAV file type since the RUN command assumes this file type. Type the following and note the results printed on the terminal:

Long and Short Command Format

```
.RUN SUM (RET)
THE VALUE OF E IS:
2.5/606/606237.2301314.06525/130440275535025.714777373527447454055502.544
.
```

You can see that something is wrong. Slashes and periods appear in the result and indicate that an error still exists somewhere in the program.

Programming errors, called "bugs", can be very difficult to find and fix. A debugging aid is described in Chapter 14. You will use it to correct the program's final error and to rerun the program. For now, however, the error will be pointed out and explained.

Look at line 40 in the assembly listing. Notice that the line's instruction converts a digit into the appropriate ASCII code before printing it on the terminal. To do this, the constant 10 is added back into the value of the digit already stored in memory, and then the value is converted (via '0, which is the ASCII code for 0) to an ASCII code which can be printed. However, unless you explicitly designate a value as decimal, the assembler assumes all values used in the program are octal. Therefore, it interprets the constant as 10(octal), or 8(decimal), and adds the wrong value every time. The conversion consequently causes the codes of the ASCII characters / and . to be used as results in some cases. The codes of other digits, while representing numeric values, are also off by two. To correct this error, you must insert a period after the 10 in line 40. The period instructs the assembler to accept the constant value 10 as a decimal value.

COMBINING OPERATIONS

EXECUTE

To produce program results, you first assembled the MACRO source program (SUM.MAC), then linked it, then ran the resulting .SAV file (SUM.SAV). You can combine these three operations using one monitor command, the EXECUTE command. This command instructs the system to select the appropriate language processor (which you indicate using an option), then process, link, and run the program. For example, to combine the assemble-link-run operations you performed in this chapter, you use the following command:

Long Command Format

```
.EXECUTE (RET)
Files? SUM/LIST/CROSSREFERENCE (RET)
```

Short Command Format

```
.EXECUTE SUM/LIST/CROSSREFERENCE (RET)
ERRORS DETECTED:  0
THE VALUE OF E IS:
2.5/606/606237.2301314.06525/13044027553502 5.71477737352744745405502.544
```

Notice how you use the /LIST and /CROSSREFERENCE options following the file name to request both an assembly and a cross-referenced listing.

SUMMARY: COMMANDS TO RUN MACRO-11 PROGRAMS

EXECUTE
    Combine the assemble-link-run operations into one command.

EXECUTE file/MACRO
    Combine the process-link-run operations into one command and specify the input file to be a MACRO file.

EXECUTE/CROSSREFERENCE
    Produce a cross-referenced listing file.

EXECUTE/LIST
    Produce a listing file of the source program.

LINK
    Link individual object modules together to form a complete program and produce a load module.

MACRO
: Assemble the MACRO-11 source program and produce an object module.

MACRO/CROSSREFERENCE
: Assemble the MACRO-11 source program and produce both an object module and a cross-referenced listing file.

MACRO/LIST
: Assemble the MACRO-11 source program and produce both a listing on the line printer and an object module.

RUN
: Run the indicated load module.

FILE MAINTENANCE

Before continuing further, you should perform the necessary file maintenance operations. Obtain a directory of all files on your system volume that have the name SUM, regardless of file type; these files were created as a result of the exercises in this chapter:

Long and Short Command Format

```
.DIRECTORY SUM.* (RET)
 08-Jul-77
SUM    .SAV      2 08-Jul-77        SUM    .BAK      3 08-Jul-77
SUM    .MAC      3 08-Jul-77        SUM    .OBJ      1 08-Jul-77
SUM    .LST      9 08-Jul-77
 5 Files, 18 Blocks
 2832 Free blocks
```

The fact that you have corrected errors in the source file of SUM.MAC makes the version of that file on your storage volume obsolete. Thus, transfer the updated copy from your system volume back to VOL: replacing the copy of SUM.MAC on the storage volume with the new version:

Long Command Format

```
.COPY (RET)
From? SUM.MAC (RET)
To   ? VOL:SUM.MAC (RET)
```

Short Command Format

```
.COPY SUM.MAC VOL:SUM.MAC (RET)
```

Next, similarly transfer SUM.SAV to your storage volume. This allows you to rerun the MACRO program without reassembling and relinking the source.

Long Command Format

```
.COPY (RET)
From? SUM.SAV (RET)
To   ? VOL:SUM.SAV (RET)
```

Short Command Format

```
.COPY SUM.SAV VOL:SUM.SAV (RET)
```

Once you have transferred to your storage volume the files you want saved, delete from the system volume those you no longer need (i.e., all the SUM files):

Long Command Format

```
.DELETE/NOQUERY (RET)
Files? SUM.* (RET)
```

Short Command Format

```
.DELETE/NOQUERY SUM.* (RET)
```

Finally, obtain an up-to-date directory listing of your storage volume so that you can see its current status:

Long and Short Command Formats

```
.DIRECTORY VOL: (RET)
 08-Jul-77
GRAPH .FOR      2 05-Jul-77      EXAMP .FOR      2 28-Jan-77
SUM   .SAV      2 08-Jul-77      EXAMP .MAC      4 25-Feb-77
SUM   .MAC      3 08-Jul-77
 5 Files, 13 Blocks
 4749 Free blocks
```

This completes the MACRO demonstration. Continue now to Chapter 12 to learn more about the link operation.

REFERENCES

Eckhouse, Richard H. Jr., *Minicomputer Systems: Organization and Programming* (PDP-11). Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1975.

A detailed guide to programming concepts, operations, and applications involving minicomputers, with emphasis on the PDP-11.

PDP-11 MACRO-11 Language Reference Manual (AA-5075A-TC). Maynard, Mass.: Digital Equipment Corporation, 1977.

A reference manual for the PDP-11 programmer using the MACRO-11 assembly language.

PDP-11 Peripherals Handbook. Maynard, Mass.: Digital Equipment Corporation, 1976.

A technical description of the PDP-11 peripheral devices, including necessary programming information.

PDP-11 Processor Handbook. Maynard, Mass.: Digital Equipment Corporation, 1975.

A technical description of the various PDP-11 processors, including complete information concerning the PDP-11 instruction set.

PDP-11 Programming Card. Maynard, Mass.: Digital Equipment Corporation, 1975.

A pocket-sized folding card summary of PDP-11 machine instructions used by the various PDP-11 assembly language processors.

PDP-11 Software Handbook. Maynard, Mass.: Digital Equipment Corporation, 1975.

A general overview and introduction to available PDP-11 software, operation systems, and language processors. See especially Section I, Chapter 3, Section II, Chapter 2; and Section III, Chapter 1.

RT-11 Advanced Programmer's Guide (DEC-11-ORAPA-A-D). Maynard, Mass.: Digital Equipment Corporation, 1977.

An RT-11 system-specific programming manual for the MACRO-11 programmer.

RT-11 System User's Guide (DEC-11-ORGDA-A-D). Maynard, Mass.: Digital Equipment Corporation, 1977.

A guide to the use of the RT-11 operating system. See Chapters 4 and 10.

CHAPTER 12
LINKING OBJECT PROGRAMS

Programs that you write in the MACRO-11 and FORTRAN IV programming languages require additional processing after their conversion to object format. Before you can run them on the system, you must link them. The link operation:

- Joins together the object modules that use a symbol with the object module that defines it.
- Relocates individual object modules as necessary and assigns absolute (permanent) memory addresses; if necessary, it also defines an overlay structure.
- Produces a load module and an optional load map (Figure 12-1).

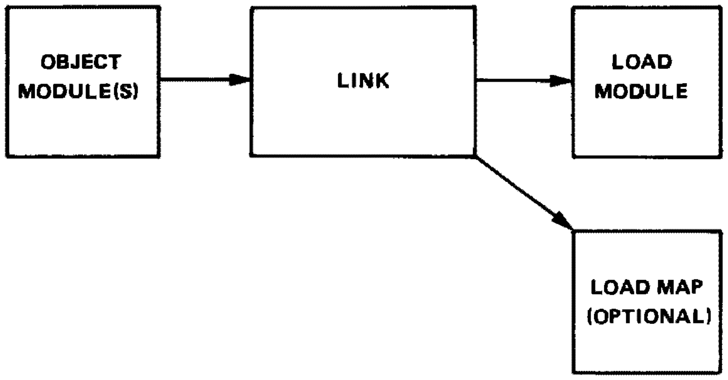


Figure 12-1 Link Functions

The advantage of program linking is that is allows you to use a modular approach to your programming. You can create an entire program as a series of smaller, independent subprograms. One of these you write as the main, or controlling, program, and the rest as subordinate subprograms and subroutines. You use the appropriate language processor to translate each part of the program into an object module. Then you use the linker to join all the object modules together into a complete, functioning unit.

Modular programming facilitates program creation and debugging. For example, several programmers can simultaneously work on a single program, each creating a portion of it. The individual portions, or subprograms, can be processed and linked with test programs and debugged for logic errors separately. Then all the object modules can be joined together to form a complete program that can be tested as a whole. If errors occur at this stage, only those object modules in error need be debugged and changed.

In addition, modular programming allows you to make use of library files. These are files that contain already-written and debugged subprograms and subroutines. Since you join library files with your program at link-time, their routines can be used by your program as needed.

RESOLVING SYMBOLIC AND LIBRARY REFERENCES

The linker reads through all the object modules that you indicate as input to the LINK command. It gathers and evaluates from them information provided by the language processor that is necessary for program linking. For each input module, this information includes the object code, information needed for relocation, the relative address of the first instruction, the global symbols used, and the absolue length of each program and program section.

One of the linker's first functions is to resolve all user-defined symbolic references and library references in the joined routines. There are actually two types of user-defined symbols – internal symbols and global symbols.

Internal symbols are limited to the object module in which they appear; thus, they cannot be referenced from any other module or defined in any other module. A program containing only internal symbolic references (such as are found in the demonstration program in Chapter 11) is complete itself and does not need to be joined with any other object programs at link-time. Thus, internal symbols are not resolved at link-time because they have already been resolved by the language processor.

Global symbols, on the other hand, are the key to modular programming. Global symbols provide the communication between object modules. Such symbols may be symbolic labels to instructions, symbolic labels to data, or symbols that are equated to a value or constant. Global symbols are defined in one object module and referenced from other separately-assembled or compiled object modules. Such symbols must be designated as global in the source code. The following segment of MACRO-11 assembly language code illustrates the use of global symbols.

```
.MAIN.  MACRO V03.00 5-JUL-77 13:39:00 PAGE 1

AU        000000  000000  000010' 000000           .GLOBAL A,C,VALUE    ;DECLARE A, C, AND VALUE
          000006  000030'
                                                                        ;AS GLOBAL SYMBOLS

          000010  013500                   A:      MOV     @(R5)+,R0    ;GLOBAL SYMBOL A IS DEFINED
                                                                        ;HERE AND CAN BE REFERENCED
                                                                        ;FROM OTHER MODULES, PROBABLY
                                                                        ;BY A SUBROUTINE CALL
          000012  016701  000014                   MOV     LOCAL,R1     ;LOCAL IS AN INTERNAL SYMBOL
                                                                        ;DEFINED AND REFERENCED ONLY
                                                                        ;WITHIN THIS MODULE

U         000016  004767  000000                   JSR     PC,C         ;CALL TO GLOBAL ROUTINE C,
                                                                        ;DEFINED IN ANOTHER MODULE

          000022  013501                           MOV     @(R5)+,R1
          000024  005003                           CLR     R3

          000026  000207                           RTS     PC

          000030  000011                   VALUE:  .WORD   11           ;GLOBAL SYMBOL DATA IS USED TO
                                                                        ;REFERENCE THIS DATA LOCATION

          000032  177777                   LOCAL:  .WORD   177777       ;INTERNAL SYMBOL USED FOR DATA
                  000001                           .END
```

While internal symbolic references, such as LOCAL in the example, can be resolved by the assembler or compiler within the single program unit, global references, such as C, cannot. They require other object modules. During translation, the language processor notes in the object module those symbols that are global. During linking, the linker keeps track of the global references and definitions found in all the object modules; and as linking proceeds, makes the appropriate correlations and modifies instructions or data as necessary. After linking, the linker prints on the terminal a list of all symbolic references that were not resolved (undefined globals), either due to a programming error or because all necessary object modules were not included in the link.

References to library files also involve the use of global symbols. You access the routines in a library by naming a routine as a global symbol in the source code of your program. You then link your program with the appropriate library file and the linker resolves the library references just as it does any global symbol. Library usage is discussed in greater detail in Chapter 13.

PROGRAM RELOCATION AND ADDRESS ASSIGNMENT

A second important function of the linker is to "fix" the relative memory addresses so that they are absolute.[1] The object module represents translated source instructions that have been assigned memory addresses relative to a base address of 0.

Look back at the assembly listing in Chapter 11. Note the second column; these addresses are relative to a base address of 0. Thus the first instruction is assembled at relative address 0, the second at

[1]FORTRAN and BASIC users who have not performed the demonstration in Chapter 11, may wish to read the section in that chapter entitled "The MACRO-11 Language Processor." That section explains the concept of converting and storing instructions in computer memory.

relative address 2, and so on. A program cannot actually be stored and run in memory using locations relative to address 0, however, because system information is already stored in some of these locations. For example, the RT-11 operating system uses byte addresses 40 through 57 to store information about the program currently executing. In addition, the RT-11 operating system uses locations in the upper range of memory for storing the resident monitor. Thus, the linker must assign memory addresses to your program that are not already in use or that will not be used during program execution. It must, therefore, assign absolute memory addresses to the relative addresses assigned by the language processor.

The linker normally starts assigning memory addresses at address 1000, since this begins a large section of free memory space. So, to obtain the actual addresses used for program loading, you must add the relocation constant 1000 to each relative address shown in the assembly listing.

A conflict arises when several individually-processed object modules are linked together. The linker cannot assign memory addresses starting at 1000 to every module, since address assignments of one would then override those of another. However, part of the information that the language processor calculates and passes to the linker is the size of each program section in each module. So the linker simply adds this size into the relocation constant for each module and assigns higher addresses, appropriately modifying all instructions and data as necessary to account for the relocation of each individual module. Figure 12-2 illustrates the relocation that must occur to accommodate object modules linked together.[1]

Absolute and Relocatable Program Sections

Just as global symbols allow you to create an entire program using several individual object modules, program sections allow you to create an object module as a series of individual sections. The advantages gained through the sectioning of programs relate primarily to control of memory allocation, program modularity, and more effective partitioning of memory. The linker processes the program section information in the object modules as directions on how to create the executable program image.

The FORTRAN IV and MACRO-11 language processors insert program sectioning information into the object module. The FORTRAN IV language processor does this automatically when program sectioning is implied by the source language statements in a user program. For example, FUNCTION, SUBROUTINE, and

[1]A load map for this relocation example is shown later in the chapter.

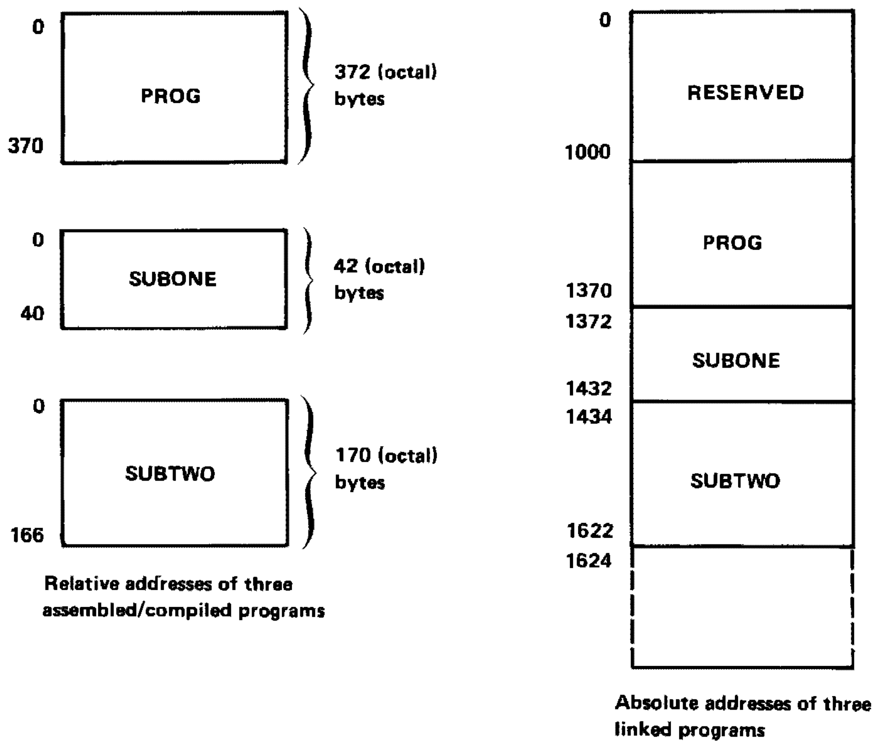


Figure 12-2 Object Module Relocation

COMMON statements result in the production of program section directives. In MACRO-11 assembly language, you are responsible for explicitly directing the assembler to output program section information for the linker. You do this via the .PSECT (or .CSECT and .ASECT) MACRO-11 assembly language statement.

Some of the basic functions associated with program sections are:

1. Instructions or data can be placed in absolute locations at memory. The named absolute program section (. ABS.) allows you to instruct the linker on exactly where to place program code or data. Declaring a section as part of the absolute program section instructs the assembler or compiler to use the internal value of the program counter as the physical memory address to be assigned after linking. This section is processed relative to absolute memory address 0 and is not relocated at link time.

2. Named relocatable program sections are used to group data or instructions into logical portions of memory. The FORTRAN COMMON statement invokes this construct to allow access to named data areas from many separate routines. Declaring a section as part of a named relocatable program section causes the section to be processed at relocatable address 0. Such sections are relocated by the linker.

3. A program section exists known as the blank program section. If you do not care to have exact control over where a portion (section) of a program will be placed in memory, use the blank program section. The linker treats this section as relocatable and the linker decides where to place it in the loadable memory image. The blank program section is the default for a MACRO-11 source program and remains in effect until an explicit program section is identified (the program example in Chapter 11 used the blank program section).

4. A program section can be identified as an instruction section. The linker, using this information, can provide automatic loading of declared overlay code when needed by the executing program (this will be discussed in more detail.

The language processor, then, actually maintains several program counters – one for the absolute program section, one for the unnamed relocatable program section, and as many as needed (maximum is 254) for named relocatable program sections. The assembled example that follows helps to explain this concept.

```
.MAIN.  MACRO V03.00 5-JUL-77 13:43:57 PAGE 1

 1
 2                                                                 ;UNNAMED RELOCATABLE PROGRAM
 3                                                                 ;SECTION IS DECLARED (BY DEFAULT)
 4                                                                 ;('.PSECT' IS ASSUMED)
 5
 6
 7
 8 000000   005000                     START:  CLR     R0
 9 000002   012701   000034'                   MOV     #BEG,R1
10 000006   062100                     LOOP:   ADD     (R1)+,R0
11 000010   022701   000044'                   CMP     #BEG+10,R1
12 000014   100374                             BPL     LOOP
13 000016   012767   002000   000020           MOV     #2000,ADDR
14 000024   005003                             CLR     R3
15
16 000000                                      .PSECT  CLEAR       ;NAMED RELOCATABLE PROGRAM
17 000000   012703   000100                    MOV     #100,R3     ;SECTION IS DECLARED (VIA '.PSECT NAME')
18 000004   012701   000044'                   MOV     #ADDR,R1
19
20 000010   005021                     AGAIN:  CLR     (R1)+
21 000012   005303                             DEC     R3
22 000014   001375                             BNE     AGAIN
23
24 000000                                      .ASECT              ;ABSOLUTE PROGRAM SECTION
25          000042                             .=42                ;DECLARED (VIA '.ASECT')
26 000042   001000                             .WORD   1000        ;THE VALUE 1000 WILL BE
27                                                                 ;STORED IN ABSOLUTE MEMORY LOCATION 42
28                                                                 ;WHEN THE PROGRAM IS EXECUTED
29
30 000026                                      .PSECT              ;BACK TO UNNAMED RELOCATABLE
31 000026   005267   000012                    INC     ADDR        ;PROGRAM SECTION
32 000032   000000                             HALT
33 000034   000001   000002   000003   BEG:    .WORD   1,2,3,4
   000042   000004
34
35 000044   000000                     ADDR:   .WORD   0
36                                     ;NOTE THAT YOU CAN WRITE LANGUAGE STATEMENTS THAT WILL BE LOADED
37                                     ;CONTIGUOUSLY IN MEMORY, BUT DO NOT NECESSARILY OCCUR CONTIGUOUSLY
38                                     ;IN THE SOURCE PROGRAM (I.E., THE CODE AT LINES 1 - 15AND 29 - 40)
39
40
41          000001                             .END
```

Since the system does not know at assembly (or compile) time in what actual memory locations each relocatable section goes, all references between sections (see line 18) are relative to the base of the section. This information is then passed to the linker so that it can make the appropriate adjustments at link time.

The Overlay Feature

The RT-11 linker is capable also of handling the special relocation and address assignments that are required whenever you indicate that an overlay structure is needed. An overlay structure is necessary when you write a program that is too large to fit in the available memory of your system. You write the program in discrete parts (some programming restrictions must be observed) so that your program can subsequently be executed in parts. One segment of the program is called the root segment and must remain in memory at all times. The root segment contains the necessary information for use by the other segments of the program, called overlay segments. Overlay segments are stored on storage volumes and brought into memory as needed. The purpose of the overlay structure is for parts of the program to share the available memory in such a way that when one part is complete, it is overlaid (and therefore erased) by another.

You indicate how to plan to overlay your program by using the /PROMPT option in the LINK command line. The linker then creates a load module that contains the necessary information for loading the appropriate segments as needed during execution. The RT-11 System User's Guide explains the overlay feature in more detail in Chapter 11. You need not specify an overlay structure for the examples demonstrated in this chapter.

PRODUCING A LOAD MODULE AND A LOAD MAP

The load module is the result of the linking processes described thus far; joining object modules, resolving symbolic and library references, relocating object modules, assigning absolute addresses, and creating an overlay structure if required. The load map is essentially a synopsis of the load module – that is, what memory looks like when the program is loaded and ready to be executed.

/MAP

In Chapters 9 and 11, you produced load modules but you did not request load maps. You obtain a load map by using the /MAP option with the LINK (or EXECUTE) command. At this time, relink the FORTRAN or MACRO object module that you stored on VOL: and use the /MAP option to produce a load map on the line printer.[1] The load map is created as a file on the default storage volume and has the name of the first input module and a file type of .MAP.

[1]FORTRAN users who followed the special instructions in Appendix B for loading the language volume should check that this volume is loaded in device unit 0.

Long Command Format

(Macro Object Module)

```
    .LINK (RET)
    Files? VOL:SUM/MAP (RET)
```

(FORTRAN object module)

```
    .LINK (RET)
    Files? VOL:GRAPH/MAP (RET)
```

Short Command Format

(MACRO object module)

```
    .LINK VOL:SUM/MAP (RET)
```

(FORTRAN object module)

```
    .LINK VOL:GRAPH/MAP (RET)
```

Now list the .MAP file on either the line printer or terminal, choosing the appropriate command:

Long Command Format

```
(Line printer)                 (Terminal)

.LINK/MAP (RET)                .LINK/MAP:TT: (RET)
Files? VOL:SUM (RET)           Files? VOL:SUM (RET)
```

Short Command Format

```
(Line printer)                 (Terminal)

.LINK/MAP VOL:SUM (RET)   .LINK/MAP:TT: VOL:SUM (RET)
```

For your convenience, both maps are provided here. In addition, a load map of the relocation example used in Figure 12-2 is also provided.

```
RT-11 LINK              Load Map        Tue 05-Jul-77 13:12:31
SUM   .SAV     Title:   SUM.MA  Ident:

Section  Addr    Size    Global  Value   Global  Value   Global  Value

. ABS.  000000  001000   (RW,I,GBL,ABS,OVR)
        001000  000372   (RW,I,LCL,REL,CON)

Transfer address = 001000, High limit = 001372 =   381. words
RT-11 LINK              Load Map        Tue 05-Jul-77 13:18:23
GRAPH .SAV     Title:   .MAIN.  Ident:  FORY02

Section  Addr    Size    Global  Value   Global  Value   Global  Value

. ABS.  000000  001000   (RW,I,GBL,ABS,OVR)
                         $USRSW  000000  $RF2A1  000000  $HRDWR  000000
                         .VIR    000000  .V014A  000001  $NLCHN  000006
                         $SYSV$  000007  $WASIZ  000131  $LRECL  000210
                         $TRACE  004737
OTS$I   001000  017074   (RW,I,LCL,REL,CON)
                         $$OTSI  001000  $CVTIF  001000  $CVTIC  001014
                         $CVTID  001014  CCI$    001026  CDI$    001026
                         $IC     001026  $ID     001026  CFI$    001042
                         $IR     001042  EXP     001126  MUF$PS  001466
                         MUF$MS  001472  MUF$IS  001502  $MULF   001510
                         MUF$SS  001522  $MLR    001522  SQRT    002032
                         DIF$PS  002226  DIF$MS  002232  DIF$IS  002242
                         $DIVF   002250  DIF$SS  002262  $DVR    002262
                         ADF$IS  002550  ADF$PS  002556  SUF$PS  002562
                         SUF$MS  002566  ADF$MS  002600  SUF$IS  002610
                         $ADDF   002616  $SUBF   002632  SUF$SS  002644
                         $SBR    002644  ADF$SS  002650  $ADR    002650
                         ADD$    002664  $OTI    003336  $$OTI   003340
                         $$SET   005046  IDINT   005342  INT     005342
                         MAX0    005370  MIN0    005414  ISN$    005440
                         $ISNTR  005444  LSN$    005460  $LSNTR  005464
                         ADI$SS  005620  ADI$SA  005624  ADI$SM  005630
                         ADI$IS  005634  ADI$IA  005640  ADI$IM  005644
                         ADI$MS  005650  ADI$MA  005654  ADI$MM  005660
                         SUI$SS  005664  SUI$SA  005670  SUI$SM  005674
                         SUI$IS  005700  SUI$IA  005704  SUI$IM  005710
                         SUI$MS  005714  SUI$MA  005720  SUI$MM  005724
                         ICI$S   005730  ICI$M   005734  ICI$P   005740
                         ICI$A   005742  DCI$S   005746  DCI$M   005752
                         DCI$P   005756  DCI$A   005760  MOF$SS  005764
                         MOF$SM  005776  MOF$SP  006006  LLE$    006012
                         LEQ$    006014  LGT$    006022  LGE$    006024
                         LNE$    006034  LLT$    006036  IOR$    006042
                         AND$    006046  EQV$    006054  XOR$    006056
                         TSL$S   006072  TSL$M   006076  TSL$I   006102
                         TSL$P   006110  RET$L   006116  RET$F   006122
                         RET$I   006130  RET$    006132  MOI$SS  006166
                         MOL$SS  006166  MOI$SM  006172  MOI$SA  006176
                         MOI$IS  006202  MOL$IS  006202  REL$    006202
                         MOI$IM  006206  MOI$IA  006212  MOI$MS  006216
                         MOI$MM  006222  MOI$MA  006226  MOI$0S  006232
                         MOI$0M  006236  MOI$0A  006242  MOI$1S  006246
                         MOI$1M  006254  MOI$1A  006262  EXIT    006270
                         NGD$S   006274  NGF$S   006274  NGD$M   006306
                         NGF$M   006306  NGD$P   006322  NGF$P   006322
                         NGD$A   006326  NGF$A   006326  CAI$    006332
                         CAL$    006340  MOI$IP  006370  MOI$SP  006372
                         MOI$PP  006400  MOI$MP  006404  MOI$PS  006414
                         MOI$PM  006422  MOI$PA  006430  MOI$0P  006436
                         MOI$1P  006444  CMI$SS  006454  CMI$SI  006460
                         CMI$SM  006464  CMI$IS  006470  CMI$II  006474
                         CMI$IM  006500  CMI$MS  006504  CMI$MI  006510
                         CMI$MM  006514  NMI$1M  006520  NMI$1I  006532
                         BLE$    006542  BEQ$    006544  BGT$    006552
                         BGE$    006554  BRA$    006556  BNE$    006562
                         BLT$    006564  MOF$RS  006574  MOF$RM  006602
                         MOF$RA  006612  MOF$RP  006616  MOF$MS  006622
                         MOF$PS  006634  MOF$MM  006640  MOF$MA  006652
                         MOF$MP  006660  MOF$PM  006666  MOF$PA  006672
                         MOF$PP  006676  MOL$SM  006702  MOL$SA  006706
                         MOL$MS  006712  MOL$MM  006722  MOL$MA  006726
                         MOL$SP  006732  MOL$PP  006740  MOL$MP  006744
                         MOL$PM  006754  MOL$PS  006762  MOL$PA  006766
                         MOL$IM  006774  MOL$IA  007002  MOL$IP  007010
                         STK$L   007020  STK$I   007024  STK$F   007030
                         MOI$RS  007040  MOL$RS  007040  MOI$RM  007044
                         MOI$RP  007050  MOI$RA  007052  $OTIS   007056
                         $$OTIS  007060  SAL$IM  007200  SAL$SM  007202
                         SVL$IM  007206  SVL$SM  007210  SAL$MM  007216
                         SVL$MM  007222  $CVTFB  007226  $CVTFI  007226
                         $CVTCB  007242  $CVTCI  007242  $CVTDB  007242
                         $CVTDI  007242  CIC$    007254  CID$    007254
                         CLC$    007254  CLD$    007254  $DI     007254
                         CIF$    007264  CLF$    007264  $RI     007264
                         CIL$    007376  CLI$    007402  TVL$    007404
```

```
Section  Addr    Size     Global  Value    Global  Value    Global  Value

                          $TVL    007404   TVF$    007412   $TVF    007412
                          TVD$    007420   $TVD    007420   TVQ$    007426
                          $TVQ    007426   TVP$    007434   $TVP    007434
                          TVI$    007442   $TVI    007442   END$    007576
                          ERR$    007610   $END    007622   $ERR    007640
                          IFW$    007662   $IFW    007666   IFW$$   007730
                          $CHKER  010000   $IOEXI  010024   $EOL    010052
                          EOL$    010054   $STPS   010170   STP$    010176
                          $STP    010176   FOO$    010202   $EXIT   010222
                          SAL$IP  010346   SAL$SP  010350   SVL$IP  010354
                          SVL$SP  010356   SAL$MP  010364   SVL$MP  010370
                          $ERRTB  010374   $ERRS   010501   $VINIT  014124
                          SAVRG$  014126   THRD$   014304   $PUTBL  014306
                          $GETBL  014516   $EOFIL  014702   $EOF2   014716
                          $PUTRE  014736   $WAIT   015166   $FCHNL  015230
                          $INITI  015326   $CLOSE  015440   $FIO    016612
                          $$FIO   016616   $DUMPL  017746
OTS$P    020074  000050    (RW,D,GBL,REL,OVR)
SYS$I    020144  000212    (RW,I,LCL,REL,CON)
                          LEN     020144   REPEAT  020162   SCOPY   020300
USER$I   020356  000000    (RW,I,LCL,REL,CON)
$CODE    020356  001316    (RW,I,LCL,REL,CON)
                          $$OTSC  020356   FUN     021234   PUTSTR  021402
OTS$O    021674  001016    (RW,I,LCL,REL,CON)
                          $$OTSO  021674   $OPEN   021674
SYS$O    022712  000000    (RW,I,LCL,REL,CON)
$DATAP   022712  000106    (RW,D,LCL,REL,CON)
OTS$D    023020  000006    (RW,D,LCL,REL,CON)
OTS$S    023026  000002    (RW,D,LCL,REL,CON)
                          $AOTS   023026
SYS$S    023030  000004    (RW,D,LCL,REL,CON)
                          $SYSLB  023030   $LOCK   023032   $CRASH  023033
$DATA    023034  000536    (RW,D,LCL,REL,CON)
USER$D   023572  000000    (RW,D,LCL,REL,CON)
.$$$$.   023572  000000    (RW,D,GBL,REL,OVR)

Transfer address = 020356, High limit = 023572 =  5053. words

RT-11 LINK                Load Map         Tue 05-Jul-77 13:53:10
LNEXP1.SAV       Title:   .MAIN.  Ident:

Section  Addr    Size     Global  Value    Global  Value    Global  Value

 . ABS.  000000  001000    (RW,I,GBL,ABS,OVR)
         001000  000034    (RW,I,LCL,REL,CON)

Transfer address = 000001, High limit = 001034 =   270. words
```

The second line has the name and file type of the load module created. Next, the absolute section and each named and unnamed section are listed under the SECTION column. To the right are abbreviated codes designating whether the section contains Instructions or Data, is Read/Write or Read Only, is a Local or Global section, is Relocatable or Absolute, is Concatenated or Overlaid. Below this falls a listing of all the global symbols (GLOBAL) and their values (VALUE). Finally, at the end of the map is the transfer address where the program actually starts when executed, followed by the high limit – the total number of bytes used by all the individual program sections.

Look first at the MACRO load map. The default absolute section starts at absolute location 0; its size is 1000 bytes. Thus, it extends from absolute memory location 0 through absolute memory location 777. The unnamed program section (there were no named program sections in this program) starts at absolute 1000; its size is 372 bytes. Thus it extends from absolute location 1000 to absolute location 1372. The high limit of this program (total bytes) is therefore 1372. Since this program is not linked to any other object modules, there are no global symbols and the rest of the map is blank.

Look now at the FORTRAN load map, remembering that it reflects the appropriate expansions into machine language code provided by the FORTRAN compiler. Again the absolute section extends from absolute 0 through absolute 777. Globals listed in the absolute section show the global variable names that the program uses as constants throughout the program.

The unnamed relocatable program section begins at absolute location 1000. Some of the named relocatable sections declared are OTSP, SYSI, and $CODE. Global symbols and their respective addresses appear to the right of all sections. The total number of bytes used is 22534.

The third load map again shows the absolute section, from absolute memory location 0 through 777. Next, the entry points of the modules (PROG, SUBONE and SUBTWO) are shown; 1000, 1372, and 1434. The transfer address is 1000 and the total number of bytes used is 1624, followed by that value in decimal words.

Load maps are most helpful when used in debugging to locate and correct assembly language programming errors. They are not generally obtained or used for FORTRAN programs, except to determine program size. In Chapter 13 you will see how a load map is used to debug the one remaining error in the MACRO demonstration program.

SUMMARY: COMMANDS FOR LINKING PROGRAMS

LINK
: Link individual object modules together to form a complete program and to produce a load module.

LINK/MAP
: Link individual object modules and produce a load map showing all address assignments made during linking.

NOTE

FORTRAN users who followed the special instructions in Appendix B to load the language volume should now stop the system, unload that volume, load the main system volume, and rebootstrap the system before going on to Chapter 13.

REFERENCE

RT-11 System User's Guide (DEC-11-ORGDA-A-D). Maynard, Mass.: Digital Equipment Corporation, 1977.

A guide to the use of the RT-11 operating system. See Chapter 11.

CHAPTER 13
CONSTRUCTING LIBRARY FILES

A library is a specially constructed file consisting of one or more programming routines. Generally, these routines provide services that you are apt to need repeatedly or services that are related and so have been gathered together for ease in use and storage. You use the routines in a library by joining the library file with your source program. Usually this occurs at link-time: but in the case of assembly language programs, it may also occur at assembly-time.

The RT-11 operating system provides several library files; SYSLIB and VTLIB for example. These libraries supply the monitor services, input and output routines, conversion routines, and other programming services that user programs may need. You can create other library files yourself. Thus you can construct libraries that contain routines specific to your programming needs or to the combined needs of those using your RT-11 system.

KINDS OF LIBRARY FILES

There are two kinds of library files – macro libraries and object libraries.

Macro Libraries

Macro libraries, such as SYSMAC.SML, are used by MACRO-11 source programs at assembly-time and consist entirely of macros. A macro is described in Chapter 11 as a recurring sequence of coding instructions which, when defined in a .MACRO statement, can then be called and used anywhere in your program. A macro library is merely several macro definitions gathered together into a single file. To use the macros in a macro library, you simply name those macros you plan to use in an .MCALL statement. When the assembler encounters the .MCALL statement during processing, it searches the appropriate macro library (SYSMAC.SML is default) for the definitions. It takes the definitions from the library and inserts them in a special table called the macro symbol table where they become available for use during assembly.

Object Libraries

Object libraries, such as SYSLIB, are used by assembled MACRO-11 source programs and/or by compiled FORTRAN IV source programs at link-time. These libraries consist of object modules that contain global routines; such routines have been defined with global entry points and then named as global symbols in the source program. During the link operation, the linker searches the object libraries to determine if they provide definitions for any undefined globals. If the linker finds definitions, it takes those object modules containing the definition from the library and includes them in the link.

A special table, called the global symbol table, lists each global in a given object library. You can print this list on the terminal or the line printer and thus keep track of an object library's current contents.

CREATING AND MAINTAINING A LIBRARY FILE

You create a library file by combining several macro routines, or several object modules, into a single larger file using the monitor LIBRARY command.

To build a macro library, first use the editor to create an ASCII text file that contains all the macro definitions. Then process this file using the LIBRARY command in combination with its /MACRO option. To update a macro library (that is, to add or delete macro definitions), simply edit the ASCII text file and then reprocess the file with the LIBRARY command.

To build an object library, again use the editor to create an ASCII text file. The file contains the routines and functions written as complete program segments in either the MACRO-11 assembly language or the FORTRAN IV programming language. Then process the file, producing an object module. Next use the LIBRARY command in combination with its /CREATE option. Once the library file is created, you update it (add and delete routines) by means of various other options to the LIBRARY command.

In the following exercises, you create an object library that contains three input object modules. The routines in two of these modules can be used by both MACRO and FORTRAN programs; the routine in the third module can be used only by FORTRAN programs.

To build the library file, first use the editor to create the three ASCII text files. Then convert the ASCII text files to object format. Finally, process the object files with the LIBRARY command. Once you create the library file, use LIBRARY command operations and options to add and delete modules and globals and to obtain a listing of the library file contents.

Creating Object Library Input Files

The first step in building an object library is to prepare the source code of the routines and functions that you choose to include in the library. Use the editor to create the following three text files, calling them FIRST.MAC. SECOND.MAC, and THIRD.FOR respectively. FORTRAN users should create all three files; MACRO users (who do not use FORTRAN) should create only the first two files.

FIRST.MAC

```
        .TITLE  COMB
        .MCALL  .PRINT
;       I=LEN(A)
        .GLOBL  LEN
LEN:    TST     (R5)+   ;SKIP # OF ARGS
        MOV     @R5,R0  ;GET STRING POINTER
1$:     TSTB    (R0)+   ;FIND END OF STRING
        BNE     1$      ;LOOP UNTIL NULL BYTE
        DEC     R0      ;BACK UP
        SUB     @R5,R0  ;CALC # OF CHARS IN STRING
        RTS     PC

;       CALL    PRINT(ISTRNG)
        .GLOBL  PRINT
PRINT:  MOV     2(R5),R0        ;ADDR OF ISTRNG
        .PRINT                  ;.PRINT
        RTS     PC              ;AND RETURN
        .END
```

SECOND.MAC

```
        .TITLE  ITTOUR
;       I=ITTOUR(ICHAR)
;       I=0     CHARACTER HAS BEEN OUTPUT
;        =1     RING BUFFER IS FULL
        .MCALL  .TTOUTR
        .GLOBL  ITTOUR
ITTOUR:MOVB     @2(R5),R0       ;GET CHARACTER
        .TTOUTR                 ;.TTOUTR
        BIC     R0,R0           ;CLEAR ERROR FLAG
        ADC     R0
        RTS     PC              ;RETURN
        .END
```

THIRD.FOR

```
C       CALL PUTSTR(LUN,AREA,CC)
        SUBROUTINE PUTSTR(LUN,AREA,CC)
        LOGICAL*1 AREA(250),CC
        IF(CC) GOTO 1
        WRITE (LUN,99)(AREA(I),I=1,LEN(AREA))
        RETURN
1       WRITE(LUN,99)CC,(AREA(I),I=1,LEN(AREA))
99      FORMAT(250A1)
        END
```

The routines in these files are representative of the kinds of services generally provided in a library file. They are, in fact, taken from the RT-11 system subroutine library, SYSLIB.

FIRST.MAC contains two global routines, LEN and PRINT. The LEN routine returns the number of characters in a string. PRINT outputs an ASCII string terminated with a zero byte to the terminal (it is the FORTRAN equivalent of the system macro .PRINT, used in the demonstration program in Chapter 11). SECOND.MAC contains one global routine, ITTOUR, which transfers a character to the console terminal. THIRD.FOR also contains one global routine – PUTSTR. This routine can be used only by FORTRAN programs and writes a variable-length character string on a specified FORTRAN logical unit (see GRAPH example).

Once you create these text files, the next step is to convert them from ASCII format to object format. You assemble or compile the text files as appropriate, first assembling FIRST.MAC and obtaining an object module (a listing is not necessary). FORTRAN users who are not familiar with the assembly process simply type the MACRO commands as shown.

Long Command Format

```
.MACRO (RET)
Files? FIRST (RET)
ERRORS DETECTED:  0
```

Short Command Format

```
.MACRO FIRST (RET)
ERRORS DETECTED:  0
```

The command creates an object module called FIRST.OBJ on the system volume. The assembler prints a message on the terminal indicating the number of errors encountered during assembly: this message should show 0 errors.

Likewise assemble SECOND.MAC. Again, no errors should occur.

Long Command Format

```
.MACRO (RET)
Files? SECOND (RET)
ERRORS DETECTED:  0
```

Short Command Format

```
    .MACRO SECOND (RET)
    ERRORS DETECTED:   0
```

If any errors occur during the assembly operations, you have incorrectly typed the source files. Find the correct the typing errors, and reassemble.

If you are a FORTRAN user, continue by compiling THIRD.FOR.

NOTE

If in Chapter 9 you needed to load the special FORTRAN/BASIC language volume, you must again load that volume before you can compile THIRD.FOR. Read Appendix B, "Substituting Volumes During Operations", before continuing.

Long Command Format

```
    .FORTRAN (RET)
    Files? THIRD (RET)
    PUTSTR
```

Short Command Format

```
    .FORTRAN THIRD (RET)
    PUTSTR
```

Notice that the compiler prints the name of the global (PUTSTR) generated. If any errors occur during the compile operation, you have incorrectly typed the source file. Find and correct the typing errors and recompile.

Once you have produced the object modules, you are ready to build the object library file.

Building the Object Library

Use the LIBRARY command in combination with its /CREATE option to construct a library file. You must indicate in the command the name of the library file and the names of the input object modules. Call the library file LIBRA and specify as input the two object modules, FIRST and SECOND. The LIBRARY command assumes that the input modules have the .OBJ file type (unless you indicate otherwise) and automatically assigns .OBJ to the library file.

LIBRARY/ CREATE

Long Command Format

```
.LIBRARY/CREATE (RET)
Library? LIBRA (RET)
Files   ? FIRST,SECOND(RET)
```

Short Command Format

```
.LIBRARY/CREATE LIBRA FIRST,SECOND (RET)
```

Once the CREATE operation is complete, obtain a listing of the library file's contents using the LIBRARY command with its LIST operation. The line printer is the assumed output device for the list file, although you may indicate a different output device by adding the 2-letter device code to the LIST option shown below.

LIBRARY/LIST

Long Command Format

(Line printer) (Terminal)

```
.LIBRARY/LIST (RET)          .LIBRARY/LIST:TT: (RET)
Library? LIBRA (RET)         Library? LIBRA (RET)
```

Short Command Format

(Line printer) (Terminal)

```
.LIBRARY/LIST LIBRA (RET) .LIBRARY/LIST:TT:LIBRA (RET)
```

The listing produced shows the library file's current contents. This library has three entry points: LEN and PRINT in the first module and ITTOUR in the second module.

```
RT-11 LIBRARIAN V03.05 FRI 08-JUL-77 11:03:29
LIBRA                  FRI 08-JUL-77 10:59:43

MODULE      GLOBALS     GLOBALS     GLOBALS

            LEN         PRINT
            ITTOUR
```

Updating the Object Library

Once you have created an object library, you use various LIBRARY command operations to update and maintain it – to add and delete modules and globals.

LIBRARY/ INSERT

If you created the THIRD.OBJ object module, you can add it to the library file using the INSERT option. If you did not create this module, read through this section anyway; the command steps apply to any object module you wish to insert.

Long Command Format

```
    .LIBRARY/INSERT (RET)
    Library? LIBRA (RET)
    Files   ? THIRD (RET)
```

Short Command Format

```
    .LIBRARY/INSERT LIBRA THIRD (RET)
```

This operation inserts the object module contained in the file THIRD.OBJ, including all its globals, into the library file LIBRA. Obtain a listing of the library contents, using the LIST option, to verify that the new globals have been added. The listing should look like this:

```
RT-11 LIBRARIAN V03.05 FRI 08-JUL-77 11:05:18
LIBRA                  FRI 08-JUL-77 11:04:21

MODULE       GLOBALS      GLOBALS      GLOBALS

             LEN          PRINT
             ITTOUR
             PUTSTR
```

This listing shows the complete library file containing the globals from all three modules.

LIBRARY/ REMOVE

You can remove individual globals by using the REMOVE option. For example, to remove the global ITTOUR, type:

Long Command Format

```
    .LIBRARY/REMOVE (RET)
    Library? LIBRA (RET)
    Global? ITTOUR (RET)
    Global?(RET)
```

Short Command Format

```
.LIBRARY/REMOVE LIBRA(RET)
Global? ITTOUR(RET)
Global?(RET)
```

The library file's contents now look like this:

```
RT-11 LIBRARIAN V03.05 FRI 08-JUL-77 11:10:22
LIBRA                  FRI 08-JUL-77 11:10:05

MODULE      GLOBALS     GLOBALS     GLOBALS

            LEN         PRINT
            PUTSTR
```

These are some of the library maintenance operations that you can perform using the LIBRARY command. Other library operations are available and are explained in the RT-11 System User's Guide, Chapter 12.

SUMMARY: COMMANDS FOR MAINTAINING LIBRARY FILES

LIBRARY/MACRO
  Create a macro library.

LIBRARY/CREATE
  Create an object library.

LIBRARY/REMOVE
  Remove globals from the object library.

LIBRARY/INSERT
  Insert object modules into the object library.

LIBRARY/LIST [:filespec]
  List the current contents of an object library on the line printer: [:filespec] is an optional output file and/or device.

FILE MAINTENANCE

Since all the object modules used in this chapter already exist as modules within the provided system library SYSLIB, there is no need to save them or the LIBRA library file. You can delete these object modules and their source files from your system volume using the DELETE command as follows (exclude THIRD.* from the command line if you did not create this file):

Long Command Format

```
.DELETE/NOQUERY (RET)
Files? FIRST.*,SECOND.*,THIRD.*,LIBRA.OBJ (RET)
```

Short Command Format

```
.DELETE/NOQUERY FIRST.*,SECOND.*,THIRD.*,LIBRA.OBJ (RET)
```

FORTRAN users who performed the special instructions given in Appendix B should also delete the THIRD files from the storage volume.

Long Command Format

```
.DELETE/NOQUERY (RET)
Files? VOL:THIRD.* (RET)
```

Short Command Format

```
.DELETE/NOQUERY VOL:THIRD.* (RET)
```

REFERENCE

RT-11 System User's Guide (DEC-11 ORGDA-A-D), Maynard, Mass.: Digital Equipment Corporation, 1977.

A guide to the use of the RT-11 operating system. See Chapter 12.

CHAPTER 14
DEBUGGING A USER PROGRAM

Debugging is the process of finding and fixing the programming errors that almost every user program initially contains. From your experience in Chapters 9, 10, and 11, you already know about some of the kinds of programming errors that can prevent a program from working properly when you run it on the system.

Frequently, debugging a program requires more time and persistence than actually writing the program code. Therefore, you should anticipate the debugging process throughout the entire program development cycle. That is, you should follow some common programming practices that help you to avoid making programming errors. When errors become apparent during the various phases of development, correct them immediately. Test the validity of any algorithms used within your program. Finally, even though the program appears to be working properly, check it thoroughly with test data.

AVOIDING PROGRAMMING ERRORS

There are several steps you can take to decrease the likelihood of introducing errors into your program and to make debugging easier should it become necessary.

First, always use a high-level language if one will suit your programming needs. High-level language programs tend to use fewer statements. English-like words and phrases in the language statements make the program logic easier to follow.

Design the program. Flowcharting the program, then correlating it with the program coding, is found to be helpful by many. This technique simplifies tracking the program logic and module interrelationships.

Use modular programming. Create the program as a series of smaller, self-contained subprograms. Debug the program in parts.

Maximize the use of subroutines, subprograms, and, in the case of assembly language programs, macros for frequently-needed functions. These help to structure the program and make it easier to alter or to add features that may be required in the future.

Make use of any software provided for you by the system such as library routines and functions. System software has already been debugged and can save you the trouble of recreating the services.

Make the general flow of a program proceed down the page. Avoid non-structured branching and convoluted logic as these tend to produce programs that are difficult to debug. Finally, use comments liberally throughout the program to show what individual statements or groups of statements do. Use spaces and tabs in the program code to make it easier to read.

Following these preventative steps eliminates many common programming errors and helps to create a programming style. However, even the most careful programmer may overlook a small detail: a typing error during program creation, an undefined label in the code, or some other programming bug. When something is overlooked, debugging becomes necessary

WHEN PROGRAMMING ERRORS OCCUR

There are three general types of programming errors – syntax, clerical, and logical.

Syntax errors are errors in the physical coding, such as omitting necessary portions of the statement (delimiters for example), reversing the order of information within the statement, or misspelling keywords or instructions.

Clerical errors are non-syntax errors in the physical coding, such as mistyped letters or digits in data. Clerical errors may result in valid statements that do not reflect correct programming logic.

Logical errors are errors made in program development.

The translating program (compiler/assembler/interpreter) generally catches syntax errors and flags them as such in the program listing or on the terminal. On the other hand, you must locate clerical and logical errors by reexamining the program code and logic, using one or more debugging techniques.

Some debugging techniques involve insertion of special debugging code within the program for checking its execution. For example, one way to locate logical errors is to write out intermediate results of a program. You can insert WRITE or PRINT statements at strategic points in the program logic to show the intermediate state of values being calculated. When debugging is complete, you can remove these statements or change them to comments.

You may also find it useful to write a special debugging subroutine that writes out values, particularly if the same variables must be examined in several places or many times.

Another method for finding logic errors is to break the program into smaller parts and test each part separately with artificial data (called unit testing). After you test all parts individually, you can test routine and module linkage to see that all related code fits together properly (called system testing).

Check the program with test data. A standard method for checking out modules is to write a test program that calls the program with possible options. The test should cause the program to execute all steps in all algorithms. Check programs first with representative data, then with improper data (data that is not in the correct range or size). Scramble input data to ensure that its sequence has no effect on the results. You should also do volume testing to see that the program works successfully with a representative amount of data.

Each programming language has special debugging aids for examining immediate states. For example, BASIC has a STOP statement that you can insert at strategic points in the program. When execution arrives at a STOP statement, it pauses and you can then use BASIC's immediate mode to examine variables, values, and so on. Use an immediate mode GO TO statement pointing to the appropriate line number to continue execution.

FORTRAN IV has a special DEBUG statement indicator, a D in the first column of a statement line. Operations in statements marked with a D can perform useful debugging functions, such as printing intermediate results. You can treat such statements as source text (and thus execute them) or as comments (and thus ignore them) depending on your use of a special compiler command option. In addition, FORTRAN IV has a traceback feature that locates the actual program unit and line number of a runtime error. If the program unit is a subroutine or function subprogram, the error handler traces back to the calling program unit and displays the name of that program unit and the line number where the call occurred. This process continues until the calling sequence has been traced back to a specific line number in the main program unit. Finally, FORTRAN IV has an optional interactive debugger called FDT (FORTRAN DEBUGGING TECHNIQUE) that can be linked with a user program.

For MACRO-11 users, RT-11 provides a special on-line debugging tool called ODT (On-line Debugging Technique). This is provided as part of the RT-11 operating system and is an object program on your system volume. It is used exclusively for debugging assembled MACRO-11 programs.

The use of ODT is described next for MACRO-11 users and for those FORTRAN IV users who will be combining MACRO and FORTRAN program code. Other users can continue to Chapter 15, or go back and perform one of the other language demonstrations. Refer to the reading path outlined in the Preface.

USING THE ON-LINE DEBUGGING TECHNIQUE

ODT is an interactive debugging tool that allows you to monitor program execution from the console terminal. ODT is provided as the object module ODT.OBJ on your system volume. To use it, you link ODT.OBJ with the assembled MACRO program that needs debugging. You then start execution of the resulting load module, not at the transfer address of your program, but at the entry point of the ODT module (shown on the linker load map as the global symbol O.ODT). Once ODT is started, you can use its special debugging commands to control the execution of your assembled machine language program from the console terminal, to examine memory locations, to change their contents, and to stop and continue program execution.

The MACRO demonstration program in Chapter 11 still contains one error which you can locate and correct using ODT. Several ODT debugging commands are demonstrated in the process.

Throughout the examples in this chapter you need to refer to the program assembly listing that you produced in Chapter 11 (SUM) and storage volume. Print it now on either the terminal or line printer:

Long Command Format

```
(Line printer)                 (Terminal)
.PRINT (RET)                   .TYPE (RET)
Files? VOL:SUM.LST (RET)       Files? VOL:SUM.LST (RET)
```

Short Command Format

(Line printer) (Terminal)

```
.PRINT VOL:SUM.LST (RET)    .TYPE VOL:SUM.LST (RET)
```

```
SUM.MAC VERSION 1       MACRO V03.00 8-JUL-77 00:16:03 PAGE 1


      1                                           .TITLE SUM.MAC  VERSION 1
      2
      3                                           .MCALL .TTYOUT, .EXIT, .PRINT
      4
      5
      6
      7         000106                            N = 70.         ;NO. OF DIGITS OF 'E' TO CALCULATE
      8
      9                                   ;       'E' = THE SUM OF THE RECIPROCALS OF THE FACTORIALS
     10                                   ;       1/0! + 1/1! + 1/2! + 1/3! + 1/4! + 1/5! + ...
     11
     12 000000                            EXP:    .PRINT  #MESSAG         ;PRINT INTRODUCTORY TEXT
     13 000006  012705  000106                    MOV     #N,R5           ;NO. OF CHARS OF 'E' TO PRINT
     14 000012  012700  000107            FIRST:  MOV     #N+1,R0         ;NO. OF DIGITS OF ACCURACY
     15 000016 \012701  000124'                   MOV     #A,R1           ;ADDRESS OF DIGIT VECTOR
     16 000022  006311                    SECOND: ASL     @R1             ;DO MULTIPLY BY 10 (DECIMAL)
     17 000024  011146                            MOV     @R1,-(SP)       ;SAVE #2
     18 000026  006311                            ASL     @R1             ;#4
     19 000030  006311                            ASL     @R1             ;#8
     20 000032  062621                            ADD     (SP)+,(R1)+     ;NOW #10, POINT TO NEXT DIGIT
     21 000034  005300                            DEC     R0              ;AT END OF DIGITS?
     22 000036  001371                            BNE     SECOND          ;BRANCH IF NOT
     23 000040  012700  000106                    MOV     #N,R0           ;GO THRU ALL PLACES, DIVIDING
     24 000044  014103                    THIRD:  MOV     -(R1),R3        ;BY THE PLACES INDEX
     25 000046  012702  177777                    MOV     #-1,R2          ;INIT QUOTIENT REGISTER
     26 000052  005202                    FOURTH: INC     R2              ;BUMP QUOTIENT
     27 000054  160003                            SUB     R0,R3           ;SUBTRACT LOOP ISN'T BAD
     28 000056  103375                            BCC     FOURTH          ;NUMERATOR IS ALWAYS < 10*N
     29 000060  060003                            ADD     R0,R3           ;FIX REMAINDER
     30 000062  010311                            MOV     R3,@R1          ;SAVE REMAINDER AS BASIS
     31                                                                   ;FOR NEXT DIGIT
     32 000064  060261  177776                    ADD     R2,-2(R1)       ;GREATEST INTEGER CARRIES
     33                                                                   ;TO GIVE DIGIT
     34 000070  005300                            DEC     R0              ;AT END OF DIGIT VECTOR?
     35 000072  001364                            BNE     THIRD           ;BRANCH IF NOT
     36 000074  014100                            MOV     -(R1),R0        ;GET DIGIT TO OUTPUT
     37 000076  162700  000012            FIFTH:  SUB     #10.,R0         ;FIX THE 2.7 TO .7 SO
     38                                                                   ;THAT IT IS ONLY 1 DIGIT
     39 000102  103375                            BCC     FIFTH           ;(REALLY DIVIDE BY 10)
     40 000104  062700  000070                    ADD     #10+'0,R0       ;MAKE DIGIT ASCII
     41 000110                                    .TTYOUT                 ;OUTPUT THE DIGIT
     42 000114  005011                            CLR     @R1             ;CLEAR NEXT DIGIT LOCATION
     43 000116  005305                            DEC     R5              ;MORE DIGITS TO PRINT?
     44 000120  001334                            BNE     FIRST           ;BRANCH IF YES
     45 000122                                    .EXIT                   ;WE ARE DONE
     46
     47 000124  000107                    A:      .REPT   N+1
     48                                           .WORD   1               ;INIT VECTOR TO ALL ONES
     49                                           .ENDR
     50
     51 000342     124     110     105    MESSAG: .ASCII  /THE VALUE OF E IS:/ <15><12> /2./ <200>
        000345     040     126     101
        000350     114     125     105
        000353     040     117     106
        000356     040     105     040
        000361     111     123     072
        000364     015     012     062

 SUM.MAC VERSION 1       MACRO V03.00 8-JUL-77 00:16:03 PAGE 1-1


         000367     056     200
     52                                            .EVEN
     53
     54          000000'                           .END    EXP

 SUM.MAC VERSION 1       MACRO V03.00 8-JUL-77 00:16:03 PAGE 1-2
 SYMBOL TABLE

 A        000124R       FIFTH   000076R       FOURTH  000052R       N      = 000106       THIRD   000044R
 EXP      000000R       FIRST   000012R       MESSAG  000342R       SECOND   000022R

 . ABS.  000000     000
         000372     001
 ERRORS DETECTED:  0

 VIRTUAL MEMORY USED:  537 WORDS  ( 3 PAGES)
 DYNAMIC MEMORY AVAILABLE FOR  60 PAGES
 DK:SUM,DK:SUM/C=DK:SUM


 SUM.MAC VERSION 1       MACRO V03.00 8-JUL-77 00:16:03 PAGE S-1
 CROSS REFERENCE TABLE (CREF V01-05 )


 A         1-15     1-47#
 EXP       1-12#    1-54
 FIFTH     1-37#    1-39
 FIRST     1-14#    1-44
 FOURTH    1-26#    1-28
 MESSAG    1-12     1-51#
 N         1-7#     1-13      1-14      1-23      1-47
 SECOND    1-16#    1-22
 THIRD     1-24#    1-35


 SUM.MAC VERSION 1       MACRO V03.00 8-JUL-77 00:16:03 PAGE M-1
 CROSS REFEPENCE TABLE (CREF V01-05 )


 .EXIT     1-3#     1-45
 .PRINT    1-3#     1-12
 .TTYOU    1-3#     1-41
```

Now link the MACRO program object module (SUM.OBJ) stored on the storage volume (VOL:) with ODT.OBJ by using the /DEBUG

LINK/DEBUG

option, and print a load map directly on the terminal or lineprinter choosing the appropriate command line below:

Long Command Format

(Lineprinter) (Terminal)

```
.LINK/MAP/DEBUG (RET)            .LINK/MAP:TT:/DEBUG (RET)
Files? VOL:SUM (RET)             Files? VOL:SUM (RET)
```

Short Command Format

(Lineprinter) (Terminal)

```
.LINK/MAP/DEBUG SUM(RET)  .LINK/MAP:TT:/DEBUG SUM(RET)
```

```
RT-11 LINK                 Load Map        Tue 05-Jul-77 13:07:12
SUM   .SAV      Title:     ODT      Ident:

Section  Addr    Size      Global   Value   Global   Value-  Global   Value

. ABS.   000000  001000     (RW,I,GBL,ABS,OVR)
         001000  006472     (RW,I,LCL,REL,CON)
                           O.ODT    001232

Transfer address = 001232, High limit = 007472 =  1949. words
```

Look at the load map and note that ODT, which is always linked first in memory by the /DEBUG option, starts at address 1000. The two modules together, ODT and SUM, reside in memory up to location 7472, the high limit. Look at the symbol table listing for the MACRO program. This shows that the program is 372 (octal) words long. To find where the MACRO program begins in memory, subtract 372 from the high address, 7472. The MACRO program starts at location 7100.

To load and start execution of the load module, use the monitor RUN command. The RUN command brings the entire load module, called SUM.SAV, into the absolute (physical) memory locations shown in the load map and begins execution automatically at the starting, or transfer, address of the first module in memory, which is ODT. Type:

Long and Short Command Formats

```
.RUN SUM(RET)
 ODT  V01.06
*
```

ODT prints an identifying message on the terminal and an asterisk indicating that you are in ODT command mode and can enter an ODT command. You are now using ODT to control the execution of your program.[1] ODT commands let you execute the entire program or only portions of it, examine individual locations of locations, examine the contents of the PDP-11 general registers, and change the contents of any locations you wish. If you make a mistake while you are typing any commands, type the DEL key; ODT reponds with a ? and an asterisk, allowing you to enter another command.

Look at locations 6 through 16 in the assembly listing. With ODT, you can examine these locations in memory as follows (all ODT commands use octal numbers, as does the assembly listing):

```
*7106/012705 (LF)
007110 /000106 (LF)
007112 /012700 (LF)
007114 /000107 (LF)
007116 /012701 (RET)
```

By typing a location address and a slash, you open that location for examination and possible modification. A line feed closes that location and opens the next sequential location for examination. A carriage return simply closes the currently open location.

Note that since the MACRO program was linked to begin at address 7100, you must add the constant 7100 to each address shown in the assembly listing to obtain the actual address used during loading. ODT can do this for you using special internal locations called relocation registers. Each register can be set to a relocation constant. Thus, if you have linked several modules together, you can set various relocation registers to the appropriate relocation constants of the individual modules. You then indicate in your command which register to use, and ODT automatically adds the constant in that register to the address specified in your command. For example, set relocation register 0 to 7100:

```
*7100;0R
```

[1] Be sure to read Chapter 16 of the RT-11 System User's Guide before you use ODT with any of your own programs. You must observe certain precautions when you write your program and when you load it with ODT. For example, you should make sure that ODT is not loaded into memory locations used by your program. There are steps you can take to prevent this from occuring.

Now, to examine locations 100 through 110, type:

```
*0,100/000012 (LF)
0,000102 /103375 (LF)
0,000104 /062700 (LF)
0,000106 /000070 (LF)
0,000110 /104341 (RET)
```

Indicate the number of the relocation register (followed by a comma) in your commands, since generally you will have more than one register set at a time.

Execute the MACRO program now using the ODT ;G command, indicating in the command where you wish execution to start. In this case, the program's start (transfer) address is 7100, so type:

```
*0,0;G
THE VALUE OF E IS:
2.5/606/606237.2301314.06525/130440275535025.71477737352744745405502.544
.
```

As you discovered in Chapter 11, these program results are incorrect, Note that a period has printed, indicating that you are back in monitor command mode. This particular MACRO program returns to the monitor after execution. Therefore, to continue using ODT, you must RUN the load module again:

Long and Short Command Format

```
.RUN SUM (RET)

 ODT  V01.06
*
```

Changes that you make to a program while using ODT, and ODT register assignments that you make, are temporary. Thus when you restart ODT, you must reenter any commands, such as relocation register commands, that you want to remain in effect. Reset relocation register 0:

```
*7100;0R
```

To help you find programming errors, ODT provides a breakpoint feature. Setting one or more breakpoints in a program causes program control to pause at those locations during execution. When control pauses, ODT prints a short message on the terminal, informing you that a breakpoint has occurred and showing the location at which execution has stopped. This pause returns control to ODT and gives you the opportunity to examine and possibly modify variables or data. Breakpoints are numbered from 0 to 7, thus you can have a total of eight breakpoints set at various instructions in the program at one time.

For example, set breakpoint 0 at location 22 (line 16 in the assembly listing) and breakpoint 1 at location 40 (line 23):

```
    *0,22;0B
    *0,40;1B
```

Now when you run the program, control pauses first at location 22. Since the breakpoint was set at the instruction at location 22, that instruction has not yet been executed, but all preceding instructions have:

```
    *0,0;G
    TB0;0,000022
```

Note the message that ODT prints when execution reaches the breakpoint. Normally when execution encounters a breakpoint, only the breakpoint number and location are printed on the terminal. In this case, the letter T precedes the breakpoint message. This happens because of the way the ODT program uses the console terminal. The assembly instruction at line 12 of the assembly listing (.PRINT) requests the monitor to print a program message at the same time that ODT needs to print the breakpoint message. ODT, however, has higher priority. By the time the .PRINT request starts to print the program message, execution reaches the breakpoint and gives control to ODT. The .PRINT request has time to print only one character of its message before ODT takes over and prints the breakpoint message. When the program regains control, its message will continue printing from the second character.

Program control has paused at location 22 in the MACRO program. Look in the assembly listing at the instructions that occur there. The instruction at location 16 (line 15) stores the address of the digit vector (at label A) in register 1 (R1). Examine the contents of register 1 to discover what this address is; then open the address and examine its contents and the contents of the next several addresses following it using two new ODT commands, $ and @:

```
    *$1/007224 @
    0,000124 /000001 (LF)
    0,000126 /000001 (LF)
    0,000130 /000001 (LF)
    0,000132 /000001 (RET)
```

The $ command opens for examination the contents of one of the general PDP-11 registers 0 through 7. The @ command uses the contents of the currently open location as an address and opens that location for examination. Notice that the digit vector A, which begins at location 124, has been initialized to the value 1, the precise value indicated by the comments at line 48 of the program listing.

If you were to continue program execution now, the branch instruction at line 22 of the assembly listing would cause program control to loop back to the instruction at line 16 where breakpoint 0 is set, again causing execution to pause. Since you wanted to continue to the next breakpoint (set at location 40), you must first cancel the breakpoint at location 22. To do this, type:

```
*;0B
```

This removes the breakpoint at location 22. The number (in this case 0) indicates which breakpoint is to be removed. Now continue program execution using the ;P command (proceed from breakpoint): execution progresses through the loop and continues until it reaches the breakpoint set at location 40:

```
*;P
HB1;0,000040
```

(Note that the monitor has time to print the second character, and perhaps additional characters, of the program message before ODT gains control.) Now examine the contents of several of the program vector locations beginning at location 124;

```
*0,124/000012 (LF)
0,000126 /000012 (LF)
0,000130 /000012 (LF)
0,000132 /000012 (RET)
```

The instructions prior to the breakpoint at location 40 constitute a multiplication routine. This routine multiplies the vector contents by 10 (12 octal), as you have just verified.

You can see how the breakpoint feature is a very useful debugging aid. It allows you to execute selected portions of a program and verify that data and variables are being used correctly during execution. You can use the breakpoint feature to locate the error that is in this program.

First, clear all previously set breakpoints (in this case, there is only the one at location 40) by typing the ;B command with no argument.

```
*;B
```

Now set a breakpoint at location 110 (line 41 of the assembly listing). You want to verify the data that is being passed to the monitor in register 0 in the ADD instruction in line 40. Type:

```
*0,110;0B
*;P
EB0;0,000110
```

Now examine the contents of register 0:

```
*$0/000065 \065 =5 (RET)
```

At this point in execution, register 0 contains 000065. The backslash (\) command prints the low-order byte of the opened location on the console terminal and also converts this to an ASCII character (if it is a valid ASCII code) and prints the character. In this case, the number 5 prints. If you look back at the program results printed earlier in this chapter, you can see that 5 is the first digit of the tabulated result (following the message "THE VALUE OF E IS 2"). If you are experienced in mathematics, you know this result is incorrect because the approximate value of E is 2.718. And you now also know that the program error is not in the interface to the monitor service used to print the result (.TTYOUT), but occurs somewhere before location 110. So the next step in debugging this program is to set a breakpoint at some earlier point in the program logic and to rerun the program. You must restart ODT to do this. Return to monitor mode by typing CTRL/C. The remainder of the program message prints on the terminal, then the monitor period appears, indicating that you are in monitor mode:

```
* (CTRL/C)
VALUE OF E IS:
2.
```

Restart ODT and reset relocation register 0:

```
.RUN SUM (RET)

 ODT  V01.06
*7100;0R
```

Set a breakpoint at location 76 (line 37 in the assembly listing) and start program execution at its beginning:

```
*0,76;0B
*0,0;G
TB0;0,000076
```

Again, examine register 0 to verify its contents:

```
*$0/000033 (RET)
```

By following the program logic in the assembly listing, you know that the value in register 0 should at this point be 33(octal) (2.7, previously multiplied by 10, = 27(decimal) = 33(octal)). So the value in register 0 is correct. From this, you can deduce that the error must occur somewhere between locations 76 and 110. The proper step now is to check the assembly listing, where you find the error at line 40. The decimal point that should follow the 10, identifying it as a decimal 10, is missing. Therefore the program treats the 10 as and octal 10, or 8(decimal), making each digit in the result off by an additive factor of 2. The data in location 106, then, should be 72, not 70. Since this data has not yet been used, you can change it now with ODT and continue program execution; if it had been used, you would need to restart ODT and then change the data. To change the contents of a location, simply open the location, type in the new contents, and close the location using a carriage return.

```
*0,104/000070  72 (RET)
```

Now eliminate all breakpoints and continue program execution; the correct results should print:

```
*;P
HE VALUE OF E IS:
2.7182818284590452353602874713526624977572470936999595749669676277240766
```

SUMMARY: COMMANDS FOR DEBUGGING PROGRAMS

To Start ODT

LINK/DEBUG
: Link the assembled program (the program to be debugged) with the ODT object module.

To Use ODT[1]

(LF)
: Close the currently open location and open the next sequential location for examination and possible modification.

[1] Only a very few of the available debugging commands have been demonstrated in this chapter. Consult Chapter 16 of the RT-11 System User's Guide for all ODT commands.

RET
: Close the currently open location.

addr/
: Open the location indicated (addr) for examination and possible modification.

addr;G
: Begin program excution at the indicated address (addr).

;P
: Continue program execution from wherever it was stopped at a breakpoint.

addr;nB
: Set one of the eight available breakpoints (n) at the indicated address (addr).

;nB
: Cancel the indicated breakpoint (n).

;B
: Cancel all breakpoints.

addr;nR
: Set on of the eight available relocations registers (n) to the relocation constant value indicated by addr.

$n
: Open one of the eight general registers (n) for examination and possible modification.

@
: Use the contents of the currently open location as an address; close the currently open location; go to the new address and open it for examination and possible modification.

\
: Print on the console terminal the low-order byte of the currently open location; if possible, convert the value to an ASCII code and print the corresponding character on the terminal.

FILE MAINTENANCE

Changes you make with ODT are temporary. Therefore you should now use the editor to correct the source program SUM.MAC. You should edit line 40 so that it reads:

```
ADD     #10.+'0,R0          ;MAKE DIGIT ASCII
```

The file SUM.MAC is currently stored on the storage volume VOL:. Edit this file and update the comment; then reassemble, relink, and rerun it to verify that it is correct. When you have done this, store the updated version of the source file on the storage volume under the same name (SUM.MAC).

After you have corrected and rerun the program, continue on to Chapter 15, or go back and perform one of the other language demonstrations. Refer to the reading path outlined in the Preface.

REFERENCE

RT-11 Sytem User's Guide (DEC-11-ORGDA-A-D). Maynard, Mass.: Digital Equipment Corporation, 1977.

A guide to the use of the RT-11 operating system. See Chapter 16.

CHAPTER 15
USING THE FOREGROUND/BACKGROUND MONITOR

A special feature of the RT-11 operating system is that it provides a choice of operating environments. Thus far, you have used its single-job environment to run the system utility programs and the demonstration programs one at a time. A second environment, called the foreground/background environment, is also available. This environment causes two independent programs to reside in memory at the same time and to execute concurrently.

Because there are different operating environments, there are actually different monitors. You are familiar with the single-job (SJ) monitor. You have used the single-job monitor so far to control the system and to perform the various exercises in this manual.

To use the foreground/background environment, you activate a second monitor, called the foreground/background (FB) monitor. The FB monitor is simply an extension of the SJ monitor; it is completely compatible with the SJ monitor, but provides extended monitor command operations for controlling a 2-job environment.[1]

THE FOREGROUND/ BACKGROUND ENVIRONMENT

The foreground/background environment is designed so that two programs can (but need not) share memory and run concurrently. One of these programs you designate as the foreground program. The system gives priority to the foreground program (or job, as it is usually called) and allows it to run until some condition, perhaps waiting for an I/O completion, causes it to relinquish control to the other program (the background job). The system then allows the background job to run until the foreground job again requires control, and so on. The two programs thereby share system resources. Whenever the foreground program is idle, the background program runs. Yet whenever the foreground program requires service, its requests are immediately satisfied. To the user at the terminal, the two programs appear to run simultaneously.

[1]The RT-11 operating system also provides a third operating environment called the extended memory environment. This environment is governed by the extended memory (XM) monitor and allows advanced users to utilize up to 128K (words) of memory. See Chapter 3 of the RT-11 Advanced Programmer's Guide for more information.

Foreground priority programs are generally time-critical. For example, you may want to designate as the foreground job a program that collects and analyzes data. Background programs are usually nontime-critical. Thus, you can continue to do program development using monitor commands to run the editor, the FORTRAN compiler, the linker, and so forth, as the background job.

Foreground/background operation requires that you have at least 16K words of computer memory (each 4K equals 4096 words) plus a system clock. Not all RT-11 computer systems support foreground/background operation since the hardware it requires is optional. To determine if your system can support FB operation, check the Hardware Configuration section in Chapter 2. If you have at least 16K of memory and the system accepts a TIME command, you can use the foreground/background monitor to perform the exercises in this chapter. Otherwise, you do not have the hardware that is necessary to support an FB environment and you should skip ahead to Chapter 16.

CHANGING MONITORS

Whenever you bootstrap the RT-11 system, it prints a message on the console terminal telling you which monitor has been loaded into memory. The message for the single-job monitor is:

```
RT-11SJ    V03-xx

.
```

BOOT

The single-job monitor is currently in memory. To use the FB environment, you must reboot the system so that the FB monitor is loaded into memory, overwriting the SJ monitor. You use the monitor BOOT command to make this switch.

If you have not entered the date and time, do so before booting the FB monitor. These features remain active throughout the booting procedure if the BOOT command is used.

Refer to question 3b in Chapter 2 to obtain the device code for your system volume and substitute this 2-character code for sy in the command line shown below:

Long Command Format

```
.BOOT (RET)
Device or file? syMNFB (RET)

RT-11FB     V03-xx
```

Short Command Format

```
.BOOT syMNFB (RET)

RT-11FB     V03-xx
```

Once the system executes the BOOT command, the monitor formerly in memory is no longer active. It is replaced by the alternate monitor. The message printed on the console terminal tells you which monitor has been loaded.[1]

USING THE FB MONITOR

Using the FB monitor is essentially no different than using the SJ monitor. All commands that are legal in the SJ environment are legal in the FB environment; their syntax and use are exactly the same. In addition, programs that you write for the single-job environment can always run as the background job in the FB environment.

Since the FB monitor is actually an extension of the SJ monitor, it provides some additional commands and programming features that the SJ monitor does not have. These allow you to control the 2-job environment. They let you interact with the two jobs and let the two jobs interact with one another.

Communication in a Two-Job Environment

When two jobs run simultaneously, you must have some means of indicating to which job you are directing commands. Likewise, the two jobs must have the means to identify themselves when they have messages to print. The following are some conventions that apply to system communication in a 2-job environment.

1. The foreground job has priority. If both the foreground and the background job are ready to print output at the same time, the foreground job does so first. The foreground job prints a complete line, then the background job prints a complete line and so on.

[1]To reboot the single-job monitor, simply reply to the BOOT command's DEVICE OR FILE ? prompt by typing syMNSJ.SYS (RET)

2. Either job can interrupt your input at the terminal if it has a message to print.

3. Messages printed by the background job are preceded by the characters B>.

4. Messages printed by the foreground job are preceded by the characters F>.

5. Typed commands are initially directed to the background job. You can redirect control alternately to the foreground and background jobs using the CTRL/F and CTRL/B commands.

   To direct typed input to the foreground job, type CTRL/F. This command instructs the monitor that all subsequent terminal input (commands and text) is directed to the foreground job. Typing this command causes the system to print an F> on the terminal unless output is already coming from the foreground job. Command input remains directed to the foreground job until the foreground job terminates, or until it is redirected to the background job via CTRL/B.

   To direct typed input to the background job, type CTRL/B. This command instructs the monitor that all subsequent terminal input (commands and text) is directed to the background job. Typing this command causes the system to print a B> on the terminal unless output is already coming from the background job. Command input remains directed to the background job until redirected to the foreground job via CTRL/F.

These conventions apply only if two jobs are running simultaneously. If only one job is running, communication is the same as in the single-job environment.

Creating the Foreground Job

In this demonstration you use the FB monitor to run two programs. You run the editor as the background job to create a short text file while you run a printing output program as the foreground job.

The printing program resides on your system volume as a file called SPOOL.MAC. Its function is to transfer all files on an assigned volume called SPL: that have a file type of .LST to another device, deleting the files from the assigned volume once they are transferred. Generally, the line printer serves as the output device so that the .LST files can be printed. However, if you do not have a lineprinter available, you can use your storage volume.

While the foreground program processes the .LST files, you use the editor to create a short text file, giving this text file a .LST file type so that it too can be processed by the SPOOL program once it is created.

The SPOOL program is an assembly language source file and must be assembled and linked before you can use it. If you performed the demonstration in Chapter 11, you are already familiar with assembly/link operations and the following command explanations can serve as review. If you did not read Chapter 11, simply type the command lines as shown. Following assembly, the system prints a message on the terminal indicating the number of errors encountered during assembly. This message will show 0 errors.

Long Command Format

```
.MACRO (RET)
Files? SPOOL/LIST (RET)
ERRORS DETECTED:  0
```

Short Command Format

```
.MACRO SPOOL/LIST (RET)
ERRORS DETECTED:  0
```

LINK/ FOREGROUND

The output resulting from this MACRO command includes an object file called SPOOL.OBJ and a listing file called SPOOL.LST. The command creates both files on your system volume. You must link the .OBJ file to produce a runnable foreground program. You use the LINK command, just as you have in earlier chapters, but you also use the /FOREGROUND option[1]. This option produces a load module with a .REL file type which signifies to the system that the file is a foreground program and is to be run as the priority job.

Long Command Format

```
.LINK/FOREGROUND (RET)
Files? SPOOL (RET)
```

Short Command Format

```
.LINK/FOREGROUND SPOOL (RET)
```

[1]This command option also applies to compiled FORTRAN programs that are to be linked as a foreground job.

Since the purpose of this foreground program is to process files that have a .LST file type, the next step is to provide some .LST files for it to use. The file SPOOL.LST, just created by the MACRO command, can serve as one. The background program you create using the editor can serve as another. If you want to list additional files as part of this exercise, create them now so that they have .LST file types. Remember that the SPOOL program deletes the .LST files from the system volume once they are processed.

Executing the Foreground and Background Jobs

Now you are ready to operate the 2-job environment. First decide which device to use for the output of the foreground program. If you have a line printer, use it for the output device or use your storage volume; in the latter case, the SPOOL program simply transfers the .LST files to the storage volume and deletes them from the system volume.

The program assumes that the output device is the line printer. Therefore, if you prefer to use your storage device, assign the line printer code (LP:) using the ASSIGN command. Type the following command, substituting the 2-character code from Table 4-2 for the storage volume in place of xx (line printer users may ignore this command):

Long Command Format

```
.ASSIGN (RET)
Physical device name? xx: (RET)
Logical  device name? LP: (RET)
```

Short Command Format

```
.ASSIGN xx: LP: (RET)
```

LOAD

When you use the FB monitor, you must always load into memory the peripheral device handlers needed by the foreground program. You use the monitor LOAD command to make a device handler permanently resident in memory. Since the SPOOL program uses the line printer, you must load the LP device handler. If you have assigned the code LP: to another device, the system automatically loads the assigned handler. Type:

Load Command Format

```
.LOAD (RET)
Device? LP:(RET)
```

Short Command Format

```
.LOAD LP:(RET)
```

The command to load and start execution of the foreground job is FRUN. It is similar to the RUN command except the system automatically loads and starts the execution of the foreground .REL program. Use this command to start the execution of SPOOL.REL.

FRUN

Long and Short Command Format

```
.FRUN SPOOL (RET)

.
F>
ASSIGN or LOAD SPL
B>
```

Here is an example of foreground communication. The foreground SPOOL program detected an error condition that prevented its further execution. Before it printed an explanatory message, however, the system first identified the message as foreground output by printing the characters F>. The background monitor next printed the characters B> and a period, indicating that control returned to monitor command mode. Command input remains directed to the background job.

The message printed by the foreground job (ASSIGN or LOAD SPL) informs you that before you can use the program you must make another device assignment – you must assign the logical name SPL: to whatever device contains the .LST files. In this case, that device is the system volume. Substitute the 2-character code for your system volume (refer to step 3b of Chapter 2) in place of sy in the command below:

Long Command Format

```
.ASSIGN
Physical device name? sy: (RET)
Logical  device name? SPL: (RET)
```

Short Command Format

```
.ASSIGN sy: SPL: (RET)
```

Once you make this assignment, you are ready to run the foreground job again.

Long and Short Command Format

```
.FRUN SPOOL (RET)

.
F>
Started OK, LP & SPL devices are assigned & loaded

B>
```

If you are using the line printer as the output device, notice that a listing begins to print on it almost immediately. This is the foreground job executing. You will not be aware of the foreground job processing the .LST files if you are using your storage volume as the output device.

If the foreground program runs out of .LST files to process, it simply waits for you to provide more, checking at 30 second intervals until then. Thus you can ignore the foreground job for now and concentrate on using the editor as the background job. Run the editor to create the text file shown below. Call this file TEXT.LST. When you have finished entering the text, close the file with the EX command.

Long and Short Command Format

```
. EDIT/CREATE TEXT.LST (RET)
*I (RET)
AS I AM INSERTING THIS TEXT, THE FOREGROUND JOB IS SENDING (RET)
THE .LST FILES TO THE LINE PRINTER, OR WHATEVER OUTPUT DEVICE (RET)
I ASSIGNED, ONE AFTER THE OTHER.  I AM RUNNING THE EDITOR (RET)
ASTHE BACKGROUND JOB.  I COULD JUST AS EASILY BE RUNNING (RET)
BASIC, FORTRAN, ANOTHER MACRO PROGRAM, OR ANY OTHER SYSTEM (RET)
UTILITY OR USER-WRITTEN PROGRAM. (RET)
(ESC) (ESC)
*EX (ESC) (ESC)
```

Since this file has a .LST file type, the foreground job will process it. In fact, as long as there are .LST files on your system volume, the SPOOL foreground processes them. When it runs out of files to process, it simply waits for more. Meanwhile, you can continue to work in the background.

When you think the foreground program is done processing all the .LST files that you have provided (for example, if the line printer stops printing), obtain a directory listing of your system volume. There should be no .LST files left.

Long and Short Command Format

```
.DIRECTORY *.LST(RET)
 20-Jul-77

 0 Files, 0 Blocks
 607 Free blocks
```

If there are still files to be processed, wait a bit, then obtain another directory listing. While you wait, you can create another file, rerun one of the previous demonstrations, or perform any other system operation that you wish. You can use the background of an FB environment in the same way as the SJ environment.

When the SPOOL program has processed all the available .LST files, you should terminate the foreground job. To do this, you must first use the CTRL/F command to direct terminal input to the foreground. Type:

```
. (CTRL/F)
F>
```

The system prints the characters F> to remind you that you are now directing command input to the foreground job. Use the double CTRL/C command to interrupt and terminate the execution of the foreground job and return control to the background job.

```
(CTRL/C)
(CTRL/C)

B>
.
```

Since you are now using only the background of the foreground/background environment, the system is operating like a single-job system.

You should unload the foreground job and the LP handler to reclaim the memory space for background use. Use the monitor UNLOAD command as follows:

UNLOAD

Long and Short Command Format

```
    .UNLOAD FG,LP:(RET)
```

FG represents the foreground job and you should use this code whenever you want to unload it. You represent devices by their 2-character device codes.

Retrieve the listings produced as the result of this demonstration from the line printer. If you used your storage volume as the output device for the SPOOL program, obtain a directory listing to see that the .LST files were transferred as expected:

Long and Short Command Format

```
.DIRECTORY VOL:*.LST(RET)
 20-Jul-77
SPOOL .LST  14 20-Jul-77 TEXT.LST  1 20-Jul-77
 2 Files, 15 Blocks
 4747 Free blocks
```

(The listings and the directory may be shown here in a different order since the SPOOL program processed them as they became available.)

The foreground program has access to all the system features available to a background program – opening and closing files, reading and writing data, and so on. However, before you begin to write and use programs in the foreground, be sure to read Chapter 1 of the RT-11 Advanced Programmer's Guide for coding restrictions.

SUMMARY: COMMANDS USED IN AN FB ENVIRONMENT

BOOT
: Bootstrap the indicated monitor (RT-11SJ, RT-11FB, RT-11XM) on the system volume.

CTRL/B
: Direct all keyboard input to the background job (until CTRL/F).

CTRL/F
: Direct all keyboard input to the foreground job (until CTRL/B).

FRUN
    Load and start execution of the foreground job.

LOAD dh
    Make the indicated device handler (dh) resident in memory.

UNLOAD dh
    Make the indicated device handler (dh) non-resident in memory, reclaiming its memory space.

UNLOAD FG
    Reclaim the memory space used by the foreground job.

FILE MAINTENANCE

If you reassigned the device name LP: to your storage volume, first use the DEASSIGN command to restore its original assignment:

Long and Short Command Format

```
.DEASSIGN LP:(RET)
```

During this exercise you created several .LST files on your system volume: these were all deleted as a result of foreground job execution. You assembled the source file SPOOL.MAC and produced an .OBJ file, linking it to produce SPOOL.REL. Thus, you should save on your storage volume the files SPOOL.REL and SPOOL.MAC and delete from your system volume the file SPOOL.OBJ. Do not delete SPOOL.MAC since this file was distributed as part of the RT-11 operating system. You may also retain SPOOL.REL for later general use as a line printer spooling program.

Long Command Format

```
.COPY (RET)
From? SPOOL.MAC,SPOOL.REL(RET)
To  ? VOL:*.*(RET)
 Files copied:
DK:SPOOL.MAC   to VOL:SPOOL.MAC
DK:SPOOL.REL   to VOL:SPOOL.REL

.DELETE/NOQUERY (RET)
Files? SPOOL.OBJ(RET)
```

Short Command Format

```
.COPY SPOOL .MAC,SPOOL.REL VOL:*.* (RET)
 Files copied:
DK:SPOOL.MAC    to VOL:SPOOL.MAC
DK:SPOOL.REL    to VOL:SPOOL.REL

.DELETE/NOQUERY SPOOL.OBJ (RET)
```

Finally, obtain a brief directory listing of your storage volume so that you can see its current status:

Long and Short Command Format

```
.DIRECTORY/BRIEF VOL: (RET)

SPOOL .LST    TEXT  .LST    EXAMP .FOR    EXAMP .MAC    MATCH .BAS
SUM   .OBJ    GRAPH .FOR    GRAPH .LST    SUM   .MAC    GRAPH .OBJ
SUM   .LST    SPOOL .MAC    SPOOL .REL
 13 Files, 79 Blocks
 4683 Free blocks
```

TEXT.LST and SPOOL.LST appear if you used the storage volume as the output device for the SPOOL program.

REFERENCES

RT-11 Advanced Programmer's Guide (DEC-11-ORAPA-A-D), Maynard, Mass.: Digital Equipment Corporation, 1977.

A technical manual providing RT-11 programming concepts. See Chapter 1.

RT-11 System User's Guide (DEC-11-ORGDA-A-D), Maynard, Mass.: Digital Equipment Corporation, 1977.

A guide to the use of the RT-11 operating system. See Chapters 2, 3 and 4.

CHAPTER 16
USING INDIRECT FILES

The RT-11 system provides an operational aid called an indirect file that allows the system to run unattended. An indirect file is a file composed entirely of monitor operating commands. When you start the execution of the indirect file, the monitor processes these commands in consecutive order. So once you have created an indirect file and started its execution, you can direct your attention to other tasks or even physically leave the system, since the monitor executes the commands automatically and consecutively.[1]

The kinds of operations that RT-11 can best perform in an indirect file are those that involve much computer processing but that do not require your supervision or intervention. For example, multiple assemblies, compilations, and data transfer operations are ideal operations for indirect file processing. Also, any series of commands that you are likely to type often can easily run as an indirect file.

CREATING AN INDIRECT FILE

Use the editor to create an indirect file as a text file. You can call the file by any file name you wish, but you should give it a file type of .COM, since this file type is the default used by the monitor to locate the file.

You structure the lines of text that make up an indirect file just like keyboard input. Thus, if you were to list the indirect file it would look like terminal keyboard text without any monitor prompts.

[1]The indirect file concept is similar to BATCH processing. Although indirect files lack many of the BATCH capabilities, they are easier to use than BATCH (The RT-11 computer system also supports a BATCH processor discussed in RT-11 System User's Guide).

Entering Monitor Commands

You enter monitor commands into the indirect file as you would on the terminal. As an example, both of the following accomplish the same operation when executed as part of an indirect file:

```
COPY (RET)
INFIL.MAC (RET)
OUTFIL.MAC (RET)

COPY INFIL.MAC OUTFIL.MAC (RET)
```

Since monitor prompts are not included in the indirect file, using the long command format requires that you anticipate each prompt and its proper response. It is suggested that you use the short command format and insert the command as a single line of text. Terminate each command line with a carriage return.

Using the Editor to Create an Indirect File

The indirect file that you will now create incorporates several of the commands previously demonstrated in this manual. Thus it serves both as an example of the format of indirect file input and as a brief review of the monitor commands used to copy, process, and delete files. In addition, one new command, DEASSIGN, is demonstrated.

Use the EDIT/CREATE monitor command to create a file called INDCT.COM, inserting the commands according to the directions in the right-hand column. When you have finished creating the file, list it and check for typing errors. Correct any errors you find and then close the file using the EX editing command.

Long and Short Command Format

```
.EDIT/CREATE INDCT.COM (RET)
```

```
*IDATE 12-MAY-77 (RET)
TIME 8:00:00 (RET)
```

Enter a hypothetical date and time (if your system has a clock).

```
DATE (RET)
```

Print the date.

```
DEASSIGN (RET)
```

Deassign all previous device assignments and set new ones as follows:

```
ASSIGN TT: LP: (RET)
```

Assign the logical name LP: to the terminal.

```
ASSIGN XX VOL: (RET)
```

Assign the device code of the storage volume (xx) to the logical name VOL:.

`DIRECTORY/BRIEF VOL: (RET)`	List an abbreviated directory of VOL:.
`COPY VOL:GRAPH.FOR GRAPH.FOR (RET)`	FORTRAN users insert this command to copy the FORTRAN demo program to the system volume.
`COPY VOL:SUM.MAC SUM.MAC (RET)`	MACRO users insert this command to copy the MACRO demo program to the system volume.
`COPY VOL:MATCH.BAS MATCH.BAS (RET)`	BASIC users insert this command to copy the BASIC demo program to the system volume.

`FORTRAN/LIST GRAPH (RET)`
`LINK/MAP GRAPH (RET)`
	FORTRAN users who do not need to load the language volume include these commands to compile and link the demo program.

`MACRO/LIST/CROSSREFERENCE SUM (RET)`
`LINK/MAP SUM (RET)`
	All users assemble and link the demo program.

`RENAME MATCH.BAS MATCH.MAP (RET)`	BASIC users simply rename the demo program.

`MACRO/LIST/CROSSREFERENCE SPOOL (RET)`
`LINK/FOREGROUND/MAP SPOOL (RET)`
	All users assemble and link the SPOOL file.

`DIRECTORY *.OBJ (RET)`	List a directory of .OBJ files.
`DELETE/NOQUERY GRAPH.* (RET)`	FORTRAN users delete the GRAPH files.
`DELETE/NOQUERY SUM.* (RET)`	MACRO users delete the SUM files.
`DELETE/NOQUERY MATCH.MAP (RET)`	BASIC users delete the MATCH file.
`DEASSIGN (RET)`	Deassign all device assignments.
`TIME (RET)`	If your system has a clock, print the time to show how long total processing took.

```
(ESC) (ESC)
*B/L (ESC)(ESC)
DATE 12-MAY-77
TIME 8:00:00

DATE

DEASSIGN

ASSIGN TT: LP:

ASSIGN RK1: VOL:

DIRECTORY/BRIEF VOL:

COPY VOL:GRAPH.FOR GRAPH.FOR

COPY VOL:SUM.MAC SUM.MAC

COPY VOL:MATCH.BAS MATCH.BAS

FORTRAN/LIST GRAPH
LINK/MAP GRAPH

MACRO/LIST/CROSSREFERENCE SUM
LINK/MAP SUM

RENAME MATCH.BAS MATCH.MAP

MACRO/LIST/CROSSREFERENCE SPOOL
LINK/FOREGROUND/MAP SPOOL

DIRECTORY *.OBJ

DELETE/NOQUERY GRAPH.*

DELETE/NOQUERY SUM.*

DELETE/NOQUERY MATCH.MAP

DEASSIGN

TIME
```

Now terminate the insert command and list the indirect file to check for errors. (Example input is shown here.)

```
*EX (ESC)(ESC)
```

Close the file INDCT.COM.

EXECUTING AN INDIRECT FILE

Once you create an indirect file with the editor and check it for errors, you are ready to start its execution. You can run an indirect file under control of the single-job monitor or as the background job under control of the foreground/background monitor. If you run an indirect file in the background of a foreground/background system while a foreground job is running however, you must take care to avoid conflicts between nondirectory-structured devices of the two jobs. For example, the jobs should not request the same magnetic tape or cassette.

The command to start the execution of an indirect file is the At sign (@) character followed by the appropriate file name (the file type .COM is assumed unless you indicate otherwise). Execution starts immediately and the system processes commands in the indirect file in consecutive order. Each command is echoed on the terminal as it is processed. If an error within the indirect file affects the processing of a command, the system prints a system message on the terminal and stops execution of the entire file. Therefore, it is particularly important that you check your indirect file for errors before you start it and then leave the area. You can stop execution of an indirect file at any time by typing two CTRL/Cs.

Run the indirect file that you have just created by typing:

```
.@INDCT
```

It takes a minute or two for the commands in this file to be processed and for the listings to print. If your system has a clock, the time printed at the end of execution tells you exactly how long command processing has taken. Following is an example run.

```
.@INDCT

.DATE 12-MAY-77

.TIME 8:00:00

.DATE
12-May-77

.DEASSIGN

.ASSIGN TT: LP:

.ASSIGN RK1: VOL:

.DIRECTORY/BRIEF VOL:
 12-May-77
SUM   .MAC     GRAPH .FOR     SPOOL .MAC     MATCH .BAS
 4 Files, 13 Blocks
 4749 Free blocks

.COPY VOL:GRAPH.FOR GRAPH.FOR

.COPY VOL:SUM.MAC SUM.MAC

.COPY VOL:MATCH.BAS MATCH.BAS

.FORTRAN/LIST GRAPH
FORTRAN IV                 Thu 12-May-77 08:00:14                PAGE 001

      C GRAPH.FOR        VERSION 1
      C THIS PROGRAM PRODUCES A PLOT ON THE TERMINAL
      C OF AN EXTERNAL FUNCTION, FUN(X,Y)
      C THE LIMITS OF THE PLOT ARE DETERMINED BY THE DATA STATEMENTS
      C "STAB" IS FILLED WITH A TABLE OF HEIGHT FLAGS
      C "STRING" IS USED TO BUILD A LINE OF GRAPH FOR PRINTING
0001        SCAL(ZMIN,ZMAX,MAXZ,K)=ZMIN+FLOAT(K-1)*(ZMAX-ZMIN)/FLOAT(MAXZ-1)
0002        LOGICAL*1 STRING(133),STAB(100)
0003        DATA XMIN,XMAX,MAXX/-5.0,5.0,45/
0004        DATA YMIN,YMAX,MAXY/-5.0,5.0,72/
0005        DATA FMIN,FMAX/0.0,1.0/
0006        CALL SCOPY('- 1 2 3 4 5 6 7 8 9 +',STAB)
0007        MAXF=LEN(STAB)
0008        DO 20 IX=1,MAXX
0009           X=SCAL(XMIN,XMAX,MAXX,IX)
0010           CALL REPEAT('*',STRING,MAXY)
0011           IF(IX.EQ.1 .OR. IX.EQ.MAXX) GOTO 20
0013            DO 10 IY=2,MAXY-1
0014                Y=SCAL(YMIN,YMAX,MAXY,IY)
0015                IFUN=2+INT(FLOAT(MAXF-3)*(FUN(X,Y)-FMIN)/(FMAX-FMIN))
0016   10           STRING(IY)=STAB(MINO(MAXF,MAXO(1,IFUN)))
0017   20      CALL PUTSTR(7,STRING,' ')
0018        CALL EXIT
0019        END
.MAIN.
?FORTRAN-I-[.MAIN.] Errors: 0, Warnings: 2
FORTRAN IV      Storage Map for Program Unit .MAIN.

Local Variables, .PSECT $DATA, Size = 000470 (  156. words)

 Name     Type     Offset   Name     Type     Offset   Name     Type     Offset
 XMIN    R*4      000352   XMAX     R*4      000356   MAXX     I*2      000362
 YMIN    R*4      000364   YMAX     R*4      000370   MAXY     I*2      000374
 FMIN    R*4      000376   FMAX     R*4      000402   ZMIN     R*4      000416
 ZMAX    R*4      000422   MAXZ     I*2      000426   K        I*2      000430
 MAXF    I*2      000432   IX       I*2      000434   X        R*4      000436
 IY      I*2      000442   Y        R*4      000444   IFUN     I*2      000450

 Local and COMMON Arrays:
```

```
 Name     Type    Section Offset         Size       Dimensions
STRING L*1        $DATA   000000  000205 (    67.) (133)
STAB   L*1        $DATA   000205  000144 (    50.) (100)

Statement Functions and Processor-Defined Functions Referenced:

 Name   Type  Name   Type   Name   Type   Name   Type   Name   Type   Name   Type
SCAL     R*4 FLOAT    R*4

External SUBROUTINE or FUNCTION Subprograms Referenced:

 Name   Type  Name   Type   Name   Type   Name   Type   Name   Type   Name   Type
SCOPY    R*4 LEN      I*2 REPEAT   R*4 INT      I*2 FUN      R*4 MINO     I*2
MAXO     I*2 PUTSTR  R*4 EXIT     R*4
FORTRAN IV       Y02.09      Thu 12-May-77 08:01:49                  PAGE 001

0001          FUNCTION FUN(X,Y)
0002          R=SQRT(X**2+Y**2)
0003          FUN=(X*Y*R*EXP(-R))**2
0004          RETURN
0005          END
FUN
FORTRAN IV       Storage Map for Program Unit FUN

Local Variables, .PSECT $DATA, Size = 000024 (   10. words)

 Name     Type    Offset   Name     Type    Offset   Name     Type    Offset
FUN      R*4 Eqv  000004  X        R*4    @ 000000  Y        R*4    @ 000002
R        R*4      000010

External SUBROUTINE or FUNCTION Subprograms Referenced:

 Name   Type  Name   Type   Name   Type   Name   Type   Name   Type   Name   Type
SQRT     R*4 EXP      R*4

.LINK/MAP GRAPH
RT-11 LINK                  Load Map        Thu 12-May-77 08:02:25
GRAPH .SAV       Title:     .MAIN.  Ident:  FORY02

Section  Addr    Size     Global  Value    Global  Value    Global  Value

. ABS.  000000  001000     (RW,I,GBL,ABS,OVR)
                          $USRSW  000000   $RF2A1  000000   $HRDWR  000000
                          .VIR    000000   .V014A  000001   $NLCHN  000006
                          $SYSV$  000007   $WASIZ  000131   $LRECL  000210
                          $TRACE  004737
OTS$I   001000  017074     (RW,I,LCL,REL,CON)
                          $$OTSI  001000   $CVTIF  001000   $CVTIC  001014
                          $CVTID  001014   CCI$    001026   CDI$    001026
                          $IC     001026   $ID     001026   CFI$    001042
                          $IR     001042   EXP     001126   MUF$PS  001466
                          MUF$MS  001472   MUF$IS  001502   $MULF   001510
                          MUF$SS  001522   $MLR    001522   SQRT    002032
                          DIF$PS  002226   DIF$MS  002232   DIF$IS  002242
                          $DIVF   002250   DIF$SS  002262   $DVR    002262
                          ADF$IS  002550   ADF$PS  002556   SUF$PS  002562
                          SUF$MS  002566   ADF$MS  002600   SUF$IS  002610
                          $ADDF   002616   $SUBF   002632   SUF$SS  002644
                          $SBR    002644   ADF$SS  002650   $ADR    002650
                          ADD$    002664   $OTI    003336   $$OTI   003340
                          $$SET   005046   IDINT   005342   INT     005342
                          MAXO    005370   MINO    005414   ISN$    005440
                          $ISNTR  005444   LSN$    005460   $LSNTR  005464
                          ADI$SS  005620   ADI$SA  005624   ADI$SM  005630
                          ADI$IS  005634   ADI$IA  005640   ADI$IM  005644
                          ADI$MS  005650   ADI$MA  005654   ADI$MM  005660
                          SUI$SS  005664   SUI$SA  005670   SUI$SM  005674
                          SUI$IS  005700   SUI$IA  005704   SUI$IM  005710
                          SUI$MS  005714   SUI$MA  005720   SUI$MM  005724
                          ICI$S   005730   ICI$M   005734   ICI$F   005740
                          ICI$A   005742   DCI$S   005746   DCI$M   005752
                          DCI$F   005756   DCI$A   005760   MOF$SS  005764
                          MOF$SM  005776   MOF$SF  006006   LLE$    006012
                          LEQ$    006014   LGT$    006022   LGE$    006024
                          LNE$    006034   LLT$    006036   IOR$    006042
                          AND$    006046   EQV$    006054   XOR$    006056
                          TSL$S   006072   TSL$M   006076   TSL$I   006102
                          TSL$F   006110   RET$L   006116   RET$F   006122
                          RET$I   006130   RET$    006132   MOI$SS  006166
                          MOL$SS  006166   MOI$SM  006172   MOI$SA  006176
                          MOI$IS  006202   MOL$IS  006202   REL$    006202
                          MOI$IM  006206   MOI$IA  006212   MOI$MS  006216
                          MOI$MM  006222   MOI$MA  006226   MOI$0S  006232
                          MOI$0M  006236   MOI$0A  006242   MOI$1S  006246
                          MOI$1M  006254   MOI$1A  006262   EXIT    006270
                          NGD$S   006274   NGF$S   006274   NGD$M   006306
                          NGF$M   006306   NGD$F   006322   NGF$F   006322
                          NGD$A   006326   NGF$A   006326   CAI$    006332
                          CAL$    006340   MOI$IP  006370   MOI$SP  006372
                          MOI$PP  006400   MOI$MP  006404   MOI$PS  006414
                          MOI$PM  006422   MOI$PA  006430   MOI$0P  006436
                          MOI$1P  006444   CMI$SS  006454   CMI$SI  006460
                          CMI$SM  006464   CMI$IS  006470   CMI$II  006474
                          CMI$IM  006500   CMI$MS  006504   CMI$MI  006510
                          CMI$MM  006514   NMI$1M  006520   NMI$1I  006532
                          BLE$    006542   BEQ$    006544   BGT$    006552
                          BGE$    006554   BRA$    006556   BNE$    006562
                          BLT$    006564   MOF$RS  006574   MOF$RM  006602
```

```
                            MOF$RA   006612   MOF$RP   006616   MOF$MS   006622
                            MOF$PS   006634   MOF$MM   006640   MOF$MA   006652
                            MOF$MP   006660   MOF$PM   006666   MOF$PA   006672
                            MOF$PP   006676   MOL$SM   006702   MOL$SA   006706
                            MOL$MS   006712   MOL$MM   006722   MOL$MA   006726
                            MOL$SP   006732   MOL$PP   006740   MOL$MP   006744
                            MOL$PM   006754   MOL$PS   006762   MOL$PA   006766
                            MOL$IM   006774   MOL$IA   007002   MOL$IP   007010
                            STK$L    007020   STK$I    007024   STK$F    007030
                            MOI$RS   007040   MOL$RS   007040   MOI$RM   007044
                            MOI$RP   007050   MOI$RA   007052   $OTIS    007056
                            $$OTIS   007060   SAL$IM   007200   SAL$SM   007202
                            SVL$IM   007206   SVL$SM   007210   SAL$MM   007216
                            SVL$MM   007222   $CVTFB   007226   $CVTFI   007226
                            $CVTCB   007242   $CVTCI   007242   $CVTDB   007242
                            $CVTDI   007242   CIC$     007254   CID$     007254
                            CLC$     007254   CLD$     007254   $DI      007254
                            CIF$     007264   CLF$     007264   $RI      007264
                            CIL$     007376   CLI$     007402   TVL$     007404
                            $TVL     007404   TVF$     007412   $TVF     007412
                            TVD$     007420   $TVD     007420   TVQ$     007426
                            $TVQ     007426   TVP$     007434   $TVP     007434
                            TVI$     007442   $TVI     007442   END$     007576
                            ERR$     007610   $END     007622   $ERR     007640
                            IFW$     007662   $IFW     007666   IFW$$    007730
                            $CHKER   010000   $IOEXI   010024   $EOL     010052
                            EOL$     010054   $STPS    010170   STP$     010176
                            $STP     010176   FOO$     010202   $EXIT    010222
                            SAL$IP   010346   SAL$SP   010350   SVL$IP   010354
                            SVL$SP   010356   SAL$MP   010364   SVL$MP   010370
                            $ERRTB   010374   $ERRS    010501   $VINIT   014124
                            SAVRG$   014126   THRD$    014304   $PUTBL   014306
                            $GETBL   014516   $EOFIL   014702   $EOF2    014716
                            $PUTRE   014736   $WAIT    015166   $FCHNL   015230
                            $INITI   015326   $CLOSE   015440   $FIO     016612
                            $$FIO    016616   $DUMPL   017746
OTS$P     020074   000050    (RW,D,GBL,REL,OVR)
SYS$I     020144   000212    (RW,I,LCL,REL,CON)
                            LEN      020144   REPEAT   020162   SCOPY    020300
USER$I    020356   000000    (RW,I,LCL,REL,CON)
$CODE     020356   001316    (RW,I,LCL,REL,CON)
                            $$OTSC   020356   FUN      021234   PUTSTR   021402
OTS$O     021674   001016    (RW,I,LCL,REL,CON)
                            $$OTSD   021674   $OPEN    021674
SYS$O     022712   000000    (RW,I,LCL,REL,CON)
$DATAP    022712   000106    (RW,D,LCL,REL,CON)
OTS$D     023020   000006    (RW,D,LCL,REL,CON)
OTS$S     023026   000002    (RW,D,LCL,REL,CON)
                            $AOTS    023026
SYS$S     023030   000004    (RW,D,LCL,REL,CON)
                            $SYSLB   023030   $LOCK    023032   $CRASH   023033
$DATA     023034   000536    (RW,D,LCL,REL,CON)
USER$D    023572   000000    (RW,D,LCL,REL,CON)
.$$$$.    023572   000000    (RW,D,GBL,REL,OVR)

Transfer address = 020356, High limit = 023572 =  5053. words

.MACRO/LIST/CROSSREFERENCE SUM

SUM.MAC VERSION 1        MACRO V03.00 12-MAY-77 08:07:29 PAGE 1

                                        .TITLE SUM.MAC  VERSION 1

                                        .MCALL .TTYOUT, .EXIT, .PRINT



          000106                        N = 70.          ;NO. OF DIGITS OF 'E' TO CALCULATE

                                ;       'E' = THE SUM OF THE RECIPROCALS OF THE FACTORIALS
                                ;       1/0! + 1/1! + 1/2! + 1/3! + 1/4! + 1/5! + ...

   000000                       EXP:    .PRINT  #MESSAG         ;PRINT INTRODUCTORY TEXT
   000006  012705  000106               MOV     #N,R5           ;NO. OF CHARS OF 'E' TO PRINT
   000012  012700  000107       FIRST:  MOV     #N+1,R0         ;NO. OF DIGITS OF ACCURACY
   000016  012701  000124'              MOV     #A,R1           ;ADDRESS OF DIGIT VECTOR
   000022  006311               SECOND: ASL     @R1             ;DO MULTIPLY BY 10 (DECIMAL)
   000024  011146                       MOV     @R1,-(SP)       ;SAVE *2
   000026  006311                       ASL     @R1             ;*4
   000030  006311                       ASL     @R1             ;*8
   000032  062621                       ADD     (SP)+,(R1)+     ;NOW *10, POINT TO NEXT DIGIT
   000034  005300                       DEC     R0              ;AT END OF DIGITS?
   000036  001371                       BNE     SECOND          ;BRANCH IF NOT
   000040  012700  000106               MOV     #N,R0           ;GO THRU ALL PLACES, DIVIDING
   000044  014103               THIRD:  MOV     -(R1),R3        ;BY THE PLACES INDEX
   000046  012702  177777               MOV     #-1,R2          ;INIT QUOTIENT REGISTER
   000052  005202               FOURTH: INC     R2              ;BUMP QUOTIENT
   000054  160003                       SUB     R0,R3           ;SUBTRACT LOOP ISN'T BAD
   000056  103375                       BCC     FOURTH          ;NUMERATOR IS ALWAYS < 10*N
   000060  060003                       ADD     R0,R3           ;FIX REMAINDER
   000062  010311                       MOV     R3,@R1          ;SAVE REMAINDER AS BASIS
                                                                ;FOR NEXT DIGIT
   000064  060261  177776               ADD     R2,-2(R1)       ;GREATEST INTEGER CARRIES
                                                                ;TO GIVE DIGIT
   000070  005300                       DEC     R0              ;AT END OF DIGIT VECTOR?
   000072  001364                       BNE     THIRD           ;BRANCH IF NOT
   000074  014100                       MOV     -(R1),R0        ;GET DIGIT TO OUTPUT
   000076  162700  000012       FIFTH:  SUB     #10.,R0         ;FIX THE 2.7 TO .7 SO
                                                                ;THAT IT IS ONLY 1 DIGIT
   000102  103375                       BCC     FIFTH           ;(REALLY DIVIDE BY 10)
   000104  062700  000070               ADD     #10+'0,R0       ;MAKE DIGIT ASCII
   000110                               .TTYOUT                 ;OUTPUT THE DIGIT
   000114  005011                       CLR     @R1             ;CLEAR NEXT DIGIT LOCATION
   000116  005305                       DEC     R5              ;MORE DIGITS TO PRINT?
   000120  001334                       BNE     FIRST           ;BRANCH IF YES
   000122                               .EXIT                   ;WE ARE DONE

   000124  000107               A:      .REPT   N+1
```

```
     48                                           .WORD   1                ;INIT VECTOR TO ALL ONES
     49                                           .ENDR
     50
     51 000342      124      110      105  MESSAG: .ASCII  /THE VALUE OF E IS!/ <15><12> /2./ <200>
        000345      040      126      101
        000350      114      125      105
        000353      040      117      106
        000356      040      105      040
        000361      111      123      072
        000364      015      012      062
SUM.MAC VERSION 1       MACRO V03.00 12-MAY-77 08:07:29 PAGE 1-1


        000367      056      200
     52                                           .EVEN
     53
     54            000000'                        .END    EXP
SUM.MAC VERSION 1       MACRO V03.00 12-MAY-77 08:07:29 PAGE 1-2
SYMBOL TABLE

A        000124R         FIFTH    000076R         FOURTH  000052R         N       = 000106          THIRD    000044R
EXP      000000R         FIRST    000012R         MESSAG  000342R         SECOND  000022R

. ABS.   000000      000
         000372      001
ERRORS DETECTED:   0

VIRTUAL MEMORY USED:  537 WORDS  ( 3 PAGES)
DYNAMIC MEMORY AVAILABLE FOR  63 PAGES
DK:SUM,LP:SUM=DK:SUM/C

ERRORS DETECTED:   0

SUM.MAC VERSION 1       MACRO V03.00 12-MAY-77 08:07:29 PAGE S-1
CROSS REFERENCE TABLE (CREF V01-05 )


A          1-15      1-47#
EXP        1-12#     1-54
FIFTH      1-37#     1-39
FIRST      1-14#     1-44
FOURTH     1-26#     1-28
MESSAG     1-12      1-51#
N          1-7#      1-13      1-14      1-23      1-47
SECOND     1-16#     1-22
THIRD      1-24#     1-35
SUM.MAC VERSION 1       MACRO V03.00 12-MAY-77 08:07:29 PAGE M-1
CROSS REFERENCE TABLE (CREF V01-05 )


.EXIT      1-3#      1-45
.PRINT     1-3#      1-12
.TTYOU     1-3#      1-41

.LINK/MAP SUM
RT-11 LINK              Load Map         Thu 12-May-77 08:11:03
SUM   .SAV      Title:  SUM.MA  Ident:

Section  Addr   Size    Global  Value   Global  Value   Global  Valu.

. ABS.   000000  001000   (RW,I,GBL,ABS,OVR)
         001000  000372   (RW,I,LCL,REL,CON)

Transfer address = 001000, High limit = 001372 =   381. words

.RENAME MATCH.BAS MATCH.MAP

.MACRO/LIST/CROSSREFERENCE SPOOL

DEMOSP  LINE PRINTER SPOOLER    MACRO V03.00 12-MAY-77 08:11:24 PAGE 1


      1                                           .TITLE  DEMOSP  LINE PRINTER SPOOLER
      2                                           .IDENT  /X01.01/
      3
      4                                   ; THIS PROGRAM WILL LIST ANY FILE THAT IS ON DEVICE 'SPL' WITH EXTENSION 'LST'
      5                                   ; AND ONCE LISTED IT WILL BE DELETED. BEFORE 'FRUN PRINT', THE USER MUST
      6                                   ; ASSIGN THE DEVICE WHICH HAS THE FILES TO 'SPL'.
      7                                   ;       EX: .ASSIGN RK1:SPL
      8                                   ; ALSO, ANY NON-RESIDENT HANDLERS MUST BE LOADED.
      9                                   ;       EX: .LOAD LP,DT
     10                                   ;            .FRUN PRINT
     11                                   ; IF THE SPL DEVICE IS OFF LINE OR A DIRECTORY READ
     12                                   ; ERROR THE PROGRAM WILL GO INTO A WAIT STATE AND TRY AGAIN LATER.
     13
     14                                   .MCALL  .READW, .WRITW, .LOOKUP, .DELETE, .PURGE, .TWAIT
     15                                   .MCALL  .DSTATU,.PRINT, .RCTRLO, .EXIT, .CLOSE, ..V2..
     16 000000                            ..V2..
     17
     18         000052                            ERRBYT  = 52             ;EMT ERROR BYTE ADDRESS
     19         001000                            WDCT    = 512.           ;WORD COUNT FOR READ'S (2 BLOCKS)
     20
     21                                   ; PLACE THE EMT REQUESTS BLOCKS BEFORE THE CODE SO THAT THE USR CAN
     22                                   ; SWAP OVER THE AREA STARTING AT 'START'.
     23
     24                                           .NLIST  BEX
     25 000000     000     024            TIMBLK: .BYTE   0,24             ;TWAIT CODE
     26 000002  000004'                           .WORD   TIME
     27 000004  000000  003410            TIME:   .WORD   0,60.*30.        ;WAIT 30 SECONDS
     28 000010  074514  000000  000000    DBLK:   .RAD50  /SPL      LST/
     29 000020  046636  000000            LP:     .RAD50  /LP0   /
     30 000024                            AREA:   .BLKW   5                ;EMT REQUEST AREA
     31
     32 000036                            START:  .RCTRLO                  ;MAKE THE TTY SPEAK IF ERROR
     33 000040  012702  000742'                   MOV     #BUFF,R2         ;R2 -> FREE SPACE BUFFER AREA
     34 000044  012737  000036' 000046            MOV     #START,@#46      ;USR LOAD ADDR.(MUST HAVE ONE FOR FG)
     35 000052                                    .DSTATU R2,#LP           ;SEE IF LP LOADED
     36 000062  103403                            BCS     1$
     37 000064  005762  000004                    TST     4(R2)            ;IF ENTRY POINT=0,
     38 000070  001003                            BNE     2$               ;NOT IN CORE
     39 000072  012700  000566'           1$:     MOV     #MSG0,R0         ;R0 -> LP NOT LOADED MESSAGE
     40 000076  000426                            BR      QUIT
     41
     42 000100                            2$:     .LOOKUP #AREA,#1,#LP     ;OPEN CHANNEL 1 FOR OUTPUT
     43 000124  103762                            BCS     1$
     44 000126  012701  000010'                   MOV     #DBLK,R1         ;R1 -> DEV:FILNAM.EXT
     45 000132                                    .DSTATUS R2,R1           ;SEE IF SPL LOADED
     46 000140  103403                            BCS     3$               ;FAILED?
     47 000142  005762  000004                    TST     4(R2)            ;IF ENTRY POINT = 0 THEN NOT LOADED
     48 000146  001004                            BNE     GO
     49 000150  012700  000574'           3$:     MOV     #MSG1,R0         ;PREPARE FOR FAILOR
     50 000154                            QUIT:   .PRINT
     51 000156                                    .EXIT                    ;QUIT AFTER ERROR
     52
     53 000160                            GO:     .PRINT  #OK              ;PRINT ALL IS OK MESSAGE
     54 000166                            FIND:   .PURGE  #0               ;FREE CHANNEL 0 FOR USE
     55 000174  005061  000002                    CLR     2(R1)            ;WHEN FILENAME IS 0 THEN
     56 000200                                    .LOOKUP #AREA,#0,R1      ; READ'S WILL BE ABSOLUTE BLOCK MODE
     57 000222  012702  000742'           1$:     MOV     #BUFF,R2         ;R2 -> BUFFER AREA FOR READS
DEMOSP  LINE PRINTER SPOOLER    MACRO V03.00 12-MAY-77 08:11:24 PAGE 1-1
```

```
     58 000226  012703  000001                 MOV     #1,R3           ;DIRECTORY STARTS AT BLK 6
     59 000232  006303                 2$:     ASL     R3              ;BY 2
     60 000234  062703  000004                 ADD     #4,R3
     61 000240                                 .READW  #AREA,#0,R2,#WDCT,R3      ;CHAN,BUFF,WDCT,BLK
     62 000274  103411                         BCS     5$              ;DIRECTORY READ ERROR,
     63                                                                ;DEVICE OFF LINE, SO WAIT.
     64 000276  010205                 3$:     MOV     R2,R5           ;COPY START OF BUFR PTR
     65 000300  062705  000012                 ADD     #12,R5          ;R5 -> FILE STATUS WD
     66 000304  032715  004000         4$:     BIT     #4000,@R5       ;END OF SEGMENT IF=4XXX
     67 000310  001407                         BEQ     6$
     68 000312  016203  000002                 MOV     2(R2),R3        ;NEXT DIR SEG PTR
     69 000316  001345                         BNE     2$              ;0 IF NO NEXT SEG
     70 000320  012700  000000'        5$:     MOV     #TIMBLK,R0      ;NO FILE TO LIST SO WAIT
     71 000324                                 .TWAIT
     72 000326  000717                         BR      FIND
     73
     74 000330  032725  002000         6$:     BIT     #2000,(R5)+     ;PERMANENT ENTRY ?
     75 000334  001404                         BEQ     7$              ;NO IF EQ
     76 000336  026527  000004  047014         CMP     4(R5),#^RLST    ;YES,LOOK AT EXTENSION
     77 000344  001403                         BEQ     COPIER          ;IF EXTENSION = LST THEN GO COPY FILE
     78 000346  062705  000014         7$:     ADD     #14,R5          ;SKIP OTHER FILE ENTRY WORDS
     79 000352  000754                         BR      4$
     80
     81 000354  012561  000002         COPIER: MOV     (R5)+,2(R1)     ;PUT FILNAM IN DBLK
     82 000360  012561  000004                 MOV     (R5)+,4(R1)
     83 000364                                 .LOOKUP #AREA,#2,R1
     84 000406  103667                         BCS     FIND            ;LOOKUP FAILED
     85 000410  005005                         CLR     R5              ;RELATIVE BLK #
     86
     87                                        ; HAVE THE FILE, NOW LIST IT
     88
     89 000412                         1$:     .READW  #AREA,#2,R2,#WDCT,R5
     90 000446  103424                         BCS     3$              ;READ ERROR OR EOF
     91 000450  010004                         MOV     R0,R4           ;NUMBER OF WORDS ACTUALLY READ
     92 000452                                 .WRITW  #AREA,#1,R2,R4,R5
     93 000504  103003                         BCC     2$
     94 000506  012700  000726'                MOV     #ERROUT,R0      ;R0 -> MSG FOR OUTPUT ERROR
     95 000512  000620                         BR      QUIT
     96
     97 000514  005725                 2$:     TST     (R5)+           ;BUMP BLK #
     98 000516  000735                         BR      1$
     99
    100 000520  105737  000052         3$:     TSTB    @#ERRBYT        ;EMT ERROR BYTE
    101 000524  001403                         BEQ     4$              ;EOF IF=0
    102 000526                                 .PRINT  #ERRIN
    103 000534                         4$:     .CLOSE  #2              ;CLOSE INPUT CHANNEL & DELETE LST FILE
    104 000542                                 .DELETE #AREA,#2,R1     ;R1 -> DBLK
    105 000564  000600                         BR      FIND            ;CONTINUE
    106
    107                                        .ENABL  LC
    108 000566     116     157     040 MSG0:   .ASCIZ  /No LP/
    109 000574     101     123     123 MSG1:   .ASCIZ  /ASSIGN or LOAD SPL/
    110 000617     123     164     141 OK:     .ASCIZ  /Started OK, LP & SPL devices are assigned & loaded/
    111 000702     114     123     124 ERRIN:  .ASCIZ  /LST file read error/
    112 000726     127     162     151 ERROUT: .ASCIZ  /Write error/
    113                                        .EVEN
    114
DEMOSP  LINE PRINTER SPOOLER    MACRO V03.00 12-MAY-77 08:11:24 PAGE 1-2


    115 000742                         BUFF:   .BLKB   4096.+START-.   ;ROOM FOR USR & BUFFER AREA
    116
    117         000036                         .END    START
DEMOSP  LINE PRINTER SPOOLER    MACRO V03.00 12-MAY-77 08:11:24 PAGE 1-3
SYMBOL TABLE

AREA    000024R         ERRBYT= 000052          GO      000160R         OK      000617R         TIME     000004R
BUFF    000742R         ERRIN   000702R         LP      000020R         QUIT    000154R         WDCT   = 001000
COPIER  000354R         ERROUT  000726R         MSG0    000566R         START   000036R         ...V1  = 000002
DBLK    000010R         FIND    000166R         MSG1    000574R         TIMBLK  000000R         ...V2  = 000027

. ABS.  000000          000
        010036          001
ERRORS DETECTED:  0

VIRTUAL MEMORY USED:  2329 WORDS  ( 10 PAGES)
DYNAMIC MEMORY AVAILABLE FOR  63 PAGES
DK:SPOOL,LP:SPOOL=DK:SPOOL/C

ERRORS DETECTED:  0

DEMOSP  LINE PRINTER SPOOLER    MACRO V03.00 12-MAY-77 08:11:24 PAGE S-1
CROSS REFERENCE TABLE (CREF V01-05 )


...V1   1-16#   1-35    1-42    1-42    1-42    1-45    1-54    1-56    1-56    1-56    1-61    1-61    1-71    1-83
        1-83    1-83    1-89    1-89    1-92    1-92    1-103   1-103   1-104   1-104   1-104
...V2   1-42    1-42    1-42#   1-42#   1-54    1-54#   1-56    1-56    1-56#   1-56#   1-61    1-61    1-61#   1-61#
        1-83    1-83    1-83#   1-83#   1-89    1-89    1-89#   1-89#   1-92    1-92    1-92#   1-92#   1-103   1-103#
        1-104   1-104   1-104#  1-104#
AREA    1-30#   1-42    1-56    1-61    1-83    1-89    1-92    1-104
BUFF    1-33    1-57    1-115#
COPIER  1-77    1-81#
DBLK    1-28#   1-44
ERRBYT  1-18#   1-100
ERRIN   1-102   1-111#
ERROUT  1-94    1-112#
FIND    1-54#   1-72    1-84    1-105
GO      1-48    1-53#
LP      1-29#   1-35    1-42
MSG0    1-39    1-108#
MSG1    1-49    1-109#
OK      1-53    1-110#
QUIT    1-40    1-50#   1-95
START   1-32#   1-34    1-115   1-117
TIMBLK  1-25#   1-70
TIME    1-26    1-27#
WDCT    1-19#   1-61    1-89
DEMOSP  LINE PRINTER SPOOLER    MACRO V03.00 12-MAY-77 08:11:24 PAGE M-1
CROSS REFERENCE TABLE (CREF V01-05 )


...CM0  1-16#   1-35    1-45
...CM1  1-16#   1-42    1-56    1-61    1-83    1-89    1-92
...CM2  1-16#   1-42    1-42    1-56    1-56    1-61    1-61    1-61    1-61    1-71    1-83    1-83    1-89    1-89
        1-89    1-89    1-92    1-92    1-92    1-92    1-104   1-104
...CM3  1-16#   1-54    1-103
...CM4  1-16#   1-61    1-89    1-92
...CM5  1-16#   1-35    1-42    1-45    1-56    1-61    1-71    1-83    1-89    1-92    1-104
...CM6  1-16#   1-71
..V2..  1-15#   1-16
.CLOSE  1-15#   1-103
.DELET  1-14#   1-104
.DSTAT  1-15#   1-35    1-45
.EXIT   1-15#   1-51
.LOOKU  1-14#   1-42    1-56    1-83
.PRINT  1-15#   1-50    1-53    1-102
.PURGE  1-14#   1-54
.RCTRL  1-15#   1-32
.READW  1-14#   1-61    1-89
.TWAIT  1-14#   1-71
.WRITW  1-14#   1-92
```

```
.LINK/FOREGROUND/MAP SPOOL
RT-11 LINK                        Load Map          Thu 12-May-77 08:11:09
SPOOL .REL          Title:        DEMOSP  Ident:    X01.01

Section  Addr       Size          Global  Value     Global  Value    Global  Value

. ABS.  000000    001000     (RW,I,GBL,ABS,OVR)
        001000    010036     (RW,I,LCL,REL,CON)

Transfer address = 001036, High limit = 011036 =  2319. words

.DIRECTORY *.OBJ
 12-May-77

GRAPH .OBJ     16 12-May-77         SUM    .OBJ      1 12-May-77
SPOOL .OBJ      2 12-May-77         SYSLIB.OBJ     198 23-Jun-77
MUBRTE.OBJ      1 04-May-77         MUBTAB.OBJ       1 04-May-77
MUBZN1.OBJ      1 04-May-77         MUBET1.OBJ       1 04-May-77
MUBXT1.OBJ      1 04-May-77         MUBZ1 .OBJ       1 04-May-77
MUBS1D.OBJ      1 04-May-77         FORLIB.OBJ     157 26-Apr-77
 20 Files, 611 Blocks
 564 Free blocks

.DELETE/NOQUERY GRAPH.*

.DELETE/NOQUERY SUM.*

.DELETE/NOQUERY MATCH.MAP

.DEASSIGN

.TIME
08:19:52
```

SUMMARY: COMMAND TO START AN INDIRECT FILE

@filnam.COM
 Start the execution of the specified indirect file (filnam.COM).

CTRL/C CTRL/C
 Halt execution of the indirect command file (use with caution).

FILE MAINTENANCE

This indirect file contains commands that perform the appropriate copy and delete file maintenance operations. If the commands were not already part of the file, you would need to perform the appropriate file maintenance commands, in monitor command mode, after execution.

REFERENCE

RT-11 System User's Guide (DEC-11-ORGDA-A-D), Maynard, Mass.: Digital Equipment Corporation, 1977.

A guide to the use of the RT-11 operating system. See Chapter 4.

CHAPTER 17
ADVICE TO NEW USERS

This manual introduces you to several common RT-11 functions but is neither exhaustive nor comprehensive in its treatment of system features, commands, or their options. For many, these fundamental system operations are sufficient; other users, however, may need or want to learn a programming language, extended system features, or the internal workings of the RT-11 system. These people should consult the references at the end of each chapter, the *RT-11 Documentation Directory*, or the *RT-11 System User's Guide*. The *RT-11 Documentation Directory* lists all RT-11-related material available from DIGITAL: the User's Guide explains in detail each command contained in this manual and additional monitor commands, including all possible command options.

The *Introduction to RT-11* has shown you the right way to use some important system features and their associated monitor commands. This information, combined with the following basic guidelines for using the system, can help you to avoid pitfalls common to new users:

- Do not become dependent on a single copy of a file. Always make a backup copy of any useful file.

- When using the editor, do not insert text in large segments. Divide long editing sessions into short ones so that user (or hardware) errors do not cost long hours of editing. Close the file with the EX command and begin editing again from where you left off.

- Avoid careless use of wildcard operations that manipulate multiple files. Use the /QUERY option to verify the operation to be performed.

- When using indirect files or BATCH streams, avoid operations that manipulate any of the system (.SYS) files or the indirect file in use. Check the indirect file carefully for errors before you use it. Once the command stream is initiated, you may be unable to detect and prevent possibly serious errors.

- If you run two jobs under control of the foreground/background monitor, be sure there is no conflict of nondirectory-structured devices (LP:, MT:, CT:, PC:, TT:) used by the two jobs.

APPENDIX A
MANUAL BOOTSTRAPPING OPERATIONS

This appendix describes the manual bootstrapping procedures used for PDP-11 computers that do not have the automatic bootstrapping capability described in Chapter 2. Three categories are covered:

Typing the Bootstrapping on the Terminal Keyboard

Using a Pushbutton Console to Bootstrap

Using a Switch Register Console to Bootstrap

TYPING THE BOOTSTRAP ON THE TERMINAL KEYBOARD

The bootstrap for your RT-11 computer system consists of a series of 6-digit numbers that you must type on the terminal keyboard. First, obtain the bootstrap from the RT-11 System Generation Manual and copy the numbers into the space below:

Now, type each number in the column on your terminal keyboard using the following method (if you make a mistake, type the DELETE key on the terminal keyboard once for each typing error and then retype the digit(s)):

1. Type 001000
2. Type slash (/)
3. Type the first number in the bootstrap column
4. Type the LINE FEED key on the keyboard
5. Type the next number in the bootstrap column

6. Repeat steps 4 and 5 until you have typed all the numbers in the column

7. Type the RETURN key on the keyboard

8. Type 1000G

9. Continue to Step 11 in Chapter 2

USING A PUSHBUTTON CONSOLE TO BOOTSTRAP

If your computer has a pushbutton console on its front panel similar to that shown in Figure A-1, you can use the buttons to manually give the computer the information it needs to bootstrap the system.

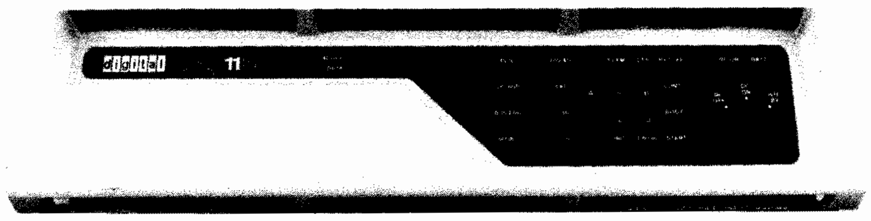


Figure A-1 Pushbutton Console

The bootstrap for your RT-11 computer system consists of a series of 6-digit numbers which you must load into the computer using the push-button console. First, obtain the bootstrap of your system device from the RT-11 System Generation Manual and copy the numbers into the space provided below. If your system has a hardware bootstrap[1], the bootstrap will consist of only two numbers which you should copy into the left-hand space; otherwise, the bootstrap will consist of two columns of numbers labeled Location and Contents which you should copy into the right-hand space:

	Hardware Bootstrap	Other Bootstraps
Load Address =		
Start Address =		

[1]A hardware bootstrap is bootstrapping information that is already in computer memory but that you must activate by entering a load address and a start address, each a 6-digit number.

To activate the hardware bootstrap, set the numbers into the pushbuttons using the following method (if you make a mistake, push the button labeled CLR, then reenter the number):

1. Push the appropriate buttons for the load address (read the number from left to right)
2. Push LAD
3. Push the appropriate buttons for the start address (read the number from left to right)
4. Push the button labeled CNTRL and while holding it down, push the button labeled START
5. Continue to step 11 in Chapter 2

To activate other bootstraps, set the numbers into the pushbuttons using the following method (if you make a mistake, push the button labeled CLR, then reenter the number):

1. Push 1000 (read the number from left to right)
2. Push LAD
3. Push the appropriate buttons for the first number in the Contents column (read the number from left to right)
4. Push DEP; push CLR
5. Push the appropriate buttons for the next number in the Contents column (read the number from left to right)
6. Repeat steps 4 and 5 until all numbers in the column have been used
7. Push 1000
8. Push LAD
9. Push the button labeled CNTRL and while holding it down push the button labeled START
10. Continue to step 11 in Chapter 2

USING A SWITCH REGISTER CONSOLE TO BOOTSTRAP

If your computer has a switch register console on the front panel similar to those shown in Figure A-2, you can use the switches to manually give the computer the bootstrapping information it needs to start the system.

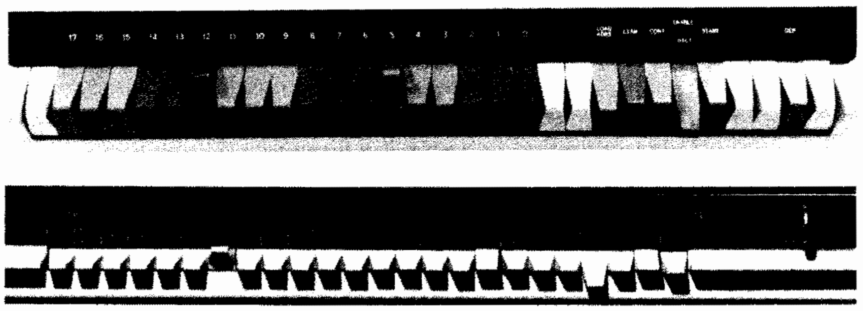


Figure A-2 Switch Register Consoles

Several switches on the console are spring-loaded. This means that the switch moves in only one direction and returns to its initial position after you use it. You must set the remaining switches either up or down as instructed.

The bootstrap for your RT-11 computer system consists of a series of 6-digit numbers which you must load into the computer using the switch register console. First, obtain the bootstrap of your system device from the RT-11 System Generation Manual and copy the numbers into the space provided below. If your system has a hardware bootstrap[1], the bootstrap consists of only two numbers, which you should copy into the left-hand space; otherwise, the bootstrap consists of two columns of numbers labeled Location and Contents which you should copy into the right-hand space:

	Hardware Bootstrap	Other Bootstraps
Load Address =		
Start Address =		

Next convert the numbers in the column to binary numbers using the conversion process shown in Table A-1.

[1] A hardware bootstrap is bootstrapping information that is already in computer memory but that you must activate by entering a load address and a start address, each a 6-digit number.

Table A-1 Binary Conversion

Octal		Binary
0	=	000
1	=	001
2	=	010
3	=	011
4	=	100
5	=	101
6	=	110
7	=	111

For example, the number 173100 is converted to 001 111 011 001 000 000. You set this 18-digit binary number into the switch register by placing each individual switch in an up position for a 1 or a down position for a 0. The number 173100 is set into the switch register as follows:

↓↓↑ ↑↑↑ ↓↑↑ ↓↓↑ ↓↓↓ ↓↓↓

The number 012700 is converted to 000 001 010 111 000 000 and is set into the switch register as follows:

↓↓↓ ↓↓↑ ↓↑↓ ↑↑↑ ↓↓↓ ↓↓↓

NOTE

The switch register is the group of switches appearing on the left of the console. Your switch register may have only 16 switches rather than 18; in this case you can ignore the lefthand two digits of the binary number when you set the switches.

To activate the hardware bootstrap:

1. Set the switch register to the appropriate positions for the load address

2. Press the spring-loaded LOAD ADDR switch

3. Set the switch register to the appropriate positions for the start address

4. Press the spring-loaded START switch

5. Continue to step 11 in Chapter 2

To activate other bootstraps, set the numbers into the switch register using the following method:

1. Set the switch register to the appropriate positions for the number 001000

2. Press the spring-loaded LOAD ADDR switch

3. Set the switch register to the appropriate positions for the first number in the Contents column

4. Press the spring-loaded DEP switch

5. Set the switch register to the appropriate positions for the next number in the Contents column

6. Repeat steps 4 and 5 until all the numbers in the column have been used

7. Set the switch register to the appropriate positions for the number 001000

8. Press the spring-loaded LOAD ADDR switch

9. Press the spring-loaded START switch

10. Continue to step 11 in Chapter 2

APPENDIX B
SELECTED SYSTEM TOPICS

The remarks in this appendix cover a variety of topics that should prove helpful to you as you perform the demonstrations in the manual. Included, for example, are instructions for starting and stopping the system, alternate methods for performing some system operations, and directions for using the language volume. The sections are listed here in the order in which they are referenced from within the text of the manual.

STOPPING AND STARTING THE SYSTEM

You can plan to take a break at the end of any individual chapter in this manual. If you intend to be away from the computer system for any length of time, you should halt the system and remove your belongings so that others may use the system hardware.

Stopping the System

Perform the following steps in order:

1. Stop the computer. Press HALT switch if your computer operator's console has switches; hold the CNTRL button down and push the HLT/SS button if your computer operator's console has pushbuttons.

2. Unload the system volume. Turn the device unit to an off-line condition and remove the system volume.

3. Unload the storage volume. Turn the device unit to an off-line condition and remove the storage volume.

4. Remove and save all terminal and line printer output listings.

Starting the System

Perform the following steps in order:

1. Follow the bootstrap procedure in Chapter 2.

2. Enter the current date and time-of-day (Chapter 4).

3. Make any necessary logical device assignments. For the examples in this manual, you must assign the logical name VOL: to your storage volume (Chapter 4).

THE SYSTEM STOPS UNEXPECTEDLY

If for any reason the computer system stops unexpectedly, request help from an experienced user. Once the problem is diagnosed, start the system by following the procedure above.

SUGGESTIONS FOR BOOTSTRAPPING THE SYSTEM

You must be able to bootstrap your RT-11 system before you can perform the demonstrations in this manual. Three common bootstrapping problems and suggestions for their correction are described below.

1. You cannot locate the bootstrapping information provided by the DIGITAL representative who installed your system.

   First, if an experienced RT-11 user is available to help you, ask this person to fill in the missing information in the *RT-11 System Generation Manual*. Then retry the bootstrap procedures in Chapter 2 of this manual.

   If no one is available to help you, consult the appropriate hardware manuals for the devices that are part of your system; these manuals provide a description of the device and operating procedures. Read the system build and start operations that are outlined in the *RT-11 System Generation Manual*. Then try the bootstrap procedures in Chapter 2 again.

2. You have followed the bootstrapping instructions correctly but your system printed a message other than what you expected.

   a. If the message is one of the following:

```
?BOOT-F-Insufficient memory

?BOOT-F-I/O error

?BOOT-F-No memory management hardware

?BOOT-F-No monitor file on volume
```

   it is a bootstrap error message and indicates that a problem in the system is preventing bootstrapping. These four messages are fully explained in the *RT-11*

System Message Manual, but you should not attempt to correct the problem yourself if an experienced user is available to help.

b. If the message is one of the following:

```
RT-11FB     V03-xx

RT-11XM     V03-xx
```

a valid RT-11 V3 monitor program has been bootstrapped, but it is not the one you should be using. Reboot the correct monitor program by typing the following commands on the terminal (sy is the appropriate 2-character code for your system volume – see question 3b in the Hardware Configuration section of Chapter 2); (RET) indicates that you should type the RETURN key on your terminal keyboard:

```
.BOOT(RET)
Device or file? syMNSJ.SYS(RET)
```

c. Any other message indicates that an old version of RT-11 (V1, V2, V2B, V2C) has been bootstrapped. Only Version 3 and later releases of RT-11 can be used to perform the demonstrations in this manual.

3. You followed the bootstrapping instructions correctly but nothing happened, i.e., there was no terminal response at all.

   Retry the bootstrap procedure from the beginning. Before you begin, be sure that the system volume is properly mounted in device unit 0. Check that the computer is on but is not running (the light labeled RUN should not be lit); if it is running, stop it as described above. Check that the terminal is on-line and that its baud rate switch (if present) is set to 300. If you are using a display, be sure the screen is bright enough. If your terminal uses a paper printer, be sure that the paper is properly loaded.

BACKING UP THE SYSTEM VOLUME

A copy of the system volume should have been made during system installation. This copy is called the master copy and should be stored away for safekeeping. If you cannot locate a master copy for your system volume, make one before you continue. Backup instructions are in the *RT-11 System Generation Manual* and should be performed by an experienced user.

DIRECTORY VS NONDIRECTORY-STRUCTURED VOLUMES

Storage volumes are called file-structured volumes because they are capable of physically storing files. They can be further categorized as directory-structured and nondirectory-structured volumes based on their method of directory information storage, collection, and printing.

The directory information kept on a volume includes file names and file types, dates of creation, and (in most cases) file lengths. When you type the DIRECTORY command, this directory information prints on the terminal. Volumes such as disk, diskette, and DECtape keep this information in a single place at the beginning of the volume. Each time you add or erase a file, the directory information at the beginning of the volume is updated accordingly. Thus, these volumes have a true volume directory and are said to be directory-structured. Magtape and cassette volumes, on the other hand, do not keep directory information in any single place on the tape but rather with each individual file. Their directory information is obtained by sequentially reading through all the files on the tape and collecting the directory for printing as each file is encountered. Thus, these volumes are said to be nondirectory-structured.

You can list the volume directories in either a complete or an abbreviated format. Complete volume directories include the file name, file type, file length (usually), and date of creation if you entered a date via the DATE command before creation. For most volumes, the directory format is as follows:

```
08-JUL-77
FILE .TYP   26  23-JUN-77
```

Cassette directories are slightly different. Their directories do not indicate file lengths, but instead show a sequence number for each file:

```
08-JUL-77
FILE .TYP    0  23-JUN-77
```

The sequence number simply indicates whether the file is continued from another cassette. 0 means the file is not continued from another cassette while any other number indicates that the file is continued. The number of blocks printed at the end of a cassette directory does not represent the total size of the files on the volume, but instead represents the total of the sequence numbers.

Abbreviated volume directories are handled the same for all directory-structured and nondirectory-structured volumes; they include only the file name and file type, and are printed in five columns on the terminal. For more information about directory-structured and nondirectory-structured volumes, see the *RT-11 System User's Guide*, Chapter 3.

ALTERNATE RENAME OPERATION FOR MAGTAPE AND CASSETTE USERS

Because of the sequential (nondirectory-structured) nature of magtapes and cassettes, you cannot use the RENAME monitor command. To perform the RENAME operation, you must first copy the file using the new file name and then erase the old file name.

Thus, to change the name of GRAPH.TWO on your magtape or cassette storage volume to GRAPH.FOR, first make a copy of GRAPH.TWO, giving the new file the name GRAPH.FOR:

Long Command Format

```
    .COPY (RET)
    From? VOL:GRAPH.TWO (RET)
    To   ? GRAPH.FOR(RET)
```

Short Command Format

```
    .COPY VOL:GRAPH.TWO GRAPH.FOR (RET)
```

Now there are two copies of the GRAPH file. Erase the one not wanted using the monitor DELETE command (this command is described in Chapter 7 in the section entitled "File Delete Operations."):

Long Command Format

```
    .DELETE/NOQUERY (RET)
    Files? VOL:GRAPH.TWO (RET)
```

Short Command Format

```
    .DELETE/NOQUERY VOL:GRAPH.TWO (RET)
```

A single copy of GRAPH.FOR now resides on your default storage (system) volume. Copy the file onto your MT: or CT: storage volume:

Long Command Format

```
.COPY (RET)
From? GRAPH.FOR (RET)
To   ? VOL:GRAPH.FOR (RET)
```

Short Command Format

```
.COPY GRAPH.FOR VOL:GRAPH.FOR (RET)
```

Delete the original file:

Long Command Format

```
.DELETE/NOQUERY (RET)
Files? GRAPH.FOR (RET)
```

Short Command Format

```
.DELETE/NOQUERY GRAPH.FOR (RET)
```

The combined effect of these four commands is to "rename" GRAPH.TWO to GRAPH.FOR.

USING THE FORTRAN/BASIC LANGUAGE VOLUME

During system installation, a special system volume was created specifically for your use with this manual. This volume contains the FORTRAN and/or BASIC language processors and the necessary monitor files required to use these language processors. Before you can perform the FORTRAN or BASIC demonstrations, you must substitute this FORTRAN/BASIC language volume for the system volume that is currently mounted in device unit 0. The language volume then becomes, and is used like, the system volume during the course of the FORTRAN and BASIC demonstrations.

Make sure no system operations are in progress (the monitor prompting period should appear at the left margin of the terminal printer) and stop the system (see "Stopping and Starting the System", this appendix). Now remove the system volume currently loaded in device unit 0 and insert the language volume, write-protected. Bootstrap the system (see "Stopping and Starting the System", this appendix). The following monitor message should appear:

```
RT-11SJ      V03-xx
```

Write-enable the volume. Then enter the current date and time-of-day and assign the logical name VOL: to your storage volume, just as you did in Chapter 4. When you have done this, you are ready to run the language demonstration. Return to the main text of the manual.

SUBSTITUTING VOLUMES DURING OPERATIONS

Diskette users and FORTRAN users who have the FORTRAN language processor on a volume apart from their system volume must occasionally perform the kinds of file copying and volume swapping operations described below. These operations are necessary when the files you need to use are not stored on the volume(s) currently mounted. The situation requires that you make the appropriate volume substitutions before you continue.

Thus, before you can compile the FORTRAN file THIRD.FOR, you must substitute the language volume containing the FORTRAN compiler for the system volume currently loaded in device unit 0. However, first you must copy the file THIRD.FOR to your storage volume so that it will be available to use.

Long Command Format

```
.COPY (RET)
From? THIRD.FOR (RET)
To   ? VOL:THIRD.FOR (RET)
```

Short Command Format

```
.COPY THIRD.FOR VOL:THIRD.FOR (RET)
```

Stop the system, remove the system volume currently loaded in unit 0, and insert the language volume write-protected. See "Stopping and Starting the System" (this appendix) if necessary. The following message should appear when you bootstrap the language volume.

```
RT-11SJ      V03-xx
```

Write-enable the volume. Then enter the current date and time-of-day and assign the logical name VOL: to your storage volume, just as you did in Chapter 4.

Next compile the FORTRAN program THIRD.FOR, which is now on VOL:

Long Command Format

```
.FORTRAN (RET)
Files? VOL:THIRD.FOR (RET)
PUTSTR
```

Short Command Format

```
.FORTRAN VOL:THIRD (RET)
PUTSTR
```

This command causes the object module to be created on the default storage volume (DK:) which is presently the system volume (i.e., the language volume). If errors occur during the compile operation, they indicate that you have incorrectly typed the source file. In this case, you must edit the file THIRD.FOR, recompile, and then copy the file to VOL:. Once you have an object module that compiles without error and is stored on VOL:, reload the main system volume in unit 0. Again, follow the directions in "Stopping and Starting the System". Once you have bootstrapped the volume, write-enable the system volume, enter the current date and time-of-day, and assign the logical name VOL: to your storage volume.

Now copy the object module on VOL: back to the system volume.

Long Command Format

```
.COPY (RET)
From? VOL:THIRD.OBJ (RET)
To   ? THIRD.OBJ (RET)
```

Short Command Format

```
.COPY VOL:THIRD.OBJ THIRD.OBJ (RET)
```

Return to Chapter 13 to the section entitled "Building the Object Library."

GLOSSARY

Absolute address
: The binary number that is assigned as the address of a physical memory storage location.

Absolute section
: The portion of a program in which the programmer has specified physical memory locations of data items.

Access time
: The interval between the instant at which data is required from or for a storage device and the instant at which the data actually begins moving to or from the device.

ADC (Analog to Digital converter)
: A circuit which converts analog signals to binary data.

Address
: A label, name or number that designates a location in memory where information is stored.

Algorithm
: A prescribed set of well-defined rules or processes for the solution of a problem in a finite number of steps.

Alphanumeric
: Referring to the subset of ASCII characters that includes the 26 alphabetic characters and the 10 numeric characters.

ANSI
: American National Standards Institute.

APL (A Programming Language)
: A condensed, high-level language capable of describing complex information processing in convenient notation. It uses arrays as basic data elements and manipulates them with a set of powerful operators. Statements are usually interpreted during execution and require no compilation whatsoever.

Application program (or package)
: A program that performs a function specific to a particular end-user's (or class of end-user's) needs. An application program can be any program that is not part of the basic operating system.

Argument
: A variable or constant value supplied with a command that controls its action, specifically its location, direction, or range.

Array
: An ordered arrangement of subscripted variables.

ASCII
: The American Standard Code for Information Interchange; a standard code using a coded character set consisting of 8-bit coded characters for upper and lower case letters, numbers, punctuation and special communication control characters.

Assembler
: A program that translates symbolic source code into machine instructions by replacing symbolic operation codes with binary operation codes and symbolic addresses with absolute or relocatable addresses.

Assembly language
: A symbolic programming language that normally can be translated directly into machine language instructions and is, therefore, specific to a given computing system.

Assembly listing
: A listing, produced by an assembler, that shows the symbolic code written by a programmer next to a representation of the actual machine instructions generated.

Asynchronous
: Pertaining to an event triggered by the occurrence of an unrelated event rather than "synchronous" or related operations scheduled by time intervals.

Background program
: A program operating automatically, at a low priority, when a higher priority (foreground) program is not using system resources.

Backup file
: A copy of a file created for protection in case the primary file is unintentionally lost or destroyed.

Base address
: An address used as the basis for computing the value of some other relative address; the address of the first location of a program or data area.

BASIC (Beginner's All-purpose Symbolic Instruction Code)
: An interactive, "algebraic" type of computer language that combines English words and decimal numbers. It is a widely available, standardized, simple beginner's language capable of handling industry and business applications.

Batch processing
: A processing method in which programs are run consecutively without operator intervention.

Baud
: A unit of signaling speed (one bit per second).

Binary
: The number system with a base of two used by internal logic of all digital computers.

Binary code
: A code that uses two distinct characters, usually the numbers 0 and 1.

Bit
: A binary digit. The smallest unit of information in a binary system of notation. It corresponds to a 1 or 0 and one digit position in a physical memory word.

Block
: A group of physically adjacent words or bytes of a specified size that is peculiar to a device. The smallest system-addressable segment on a mass-storage device in reference to I/O.

Bootstrap
: A technique or routine whose first instructions are sufficient to load the remainder of itself and start a complex system of programs.

BOT (Beginning of Tape)
: A reflective marker applied to the backside of magtape which identifies the beginning of the magtape's recordable surface.

Bottom address
: The lowest memory address into which a program is loaded.

Breakpoint
: A location at which program operation is suspended to allow operator investigation.

Buffer
: A storage area used to temporarily hold information being transferred between two devices or between a device and memory. A buffer is often a special register or a designated area of memory.

Bug
: A flaw in the design or implementation of a program which may cause erroneous results.

Bus
A circuit used as a power supply or data exchange line between two or more devices.

Byte
The smallest memory-addressable unit of information. In a PDP-11 computer system, a byte is equivalent to eight bits.

Call
A transfer from one part of a program to another with the ability to return to the original program at the point of the call.

Calling sequence
A specified arrangement of instructions and data necessary to pass parameters and control to a given subroutine.

Central processing unit (CPU)
A unit of a computer that includes the circuits controlling the interpretation and execution of instructions.

Character
A single letter, numeral, or symbol used to represent information.

Character pointer
The place where the next character typed will be entered. (The character pointer is visible as a blinking cursor on VT-11 display hardware.) During editing, the character pointer indicates the place in an ASCII text file where the next character typed will be entered into the file.

Clear
To erase the contents of a storage location by replacing the contents, normally with 0s or spaces.

Clock
A device that generates regular periodic signals for synchronization.

Code
A system of symbols and rules used for representing information – usually refers to instructions executed by computer.

Coding
To write instructions for a computer using symbols meaningful to the computer itself or to an assembler, compiler or other language processor.

Command
A word, mnemonic, or character, which, by virtue of its syntax in a line of input, causes a computer system to perform a predefined operation.

Command language
The vocabulary used by a program or set of programs that directs the computer system to perform predefined operations.

Command language interpreter
The program that translates a predefined set of commands into instructions that a computer system can interpret.

Command string
A line of input to a computer system that generally includes a command, one or more file specifications, and optional qualifiers.

Compile
To produce binary code from symbolic instructions written in a high-level source language.

Compiler
A program that translates a high-level source language into a language suitable for a particular machine.

Computer
A machine that can be programmed to execute a repertoire of instructions. Programs must be stored in the machine before they can be executed.

Computer program
A plan or routine for solving a problem on a computer.

Computer system
A data processing system that consists of hardware devices, software programs, and documentation that describes the operation of the system.

Concatenation
The joining of two strings of characters to produce a longer string.

Conditional assembly
The assembly of certain parts of a symbolic program only when certain conditions are met during the assembly process.

Configuration
A particular selection of hardware devices or software routines or programs that function together.

Console terminal
: A keyboard terminal that acts as the primary interface between the computer operator and the computer system. It is used to initiate and direct overall system operation through software running on the computer.

Constant
: A value that remains the same throughout a distinct operation. (Compare with Variable.)

Context switching
: The saving of key registers and other memory areas prior to switching between jobs with different modes of execution, as in background/foreground programming.

Conversational
: See Interactive.

CPU
: See central processing unit.

Crash
: A hardware crash is the complete failure of a particular device, sometimes affecting the operation of an entire computer system. A software crash is the complete failure of an operating system usually characterized by some failure in the system's protection mechanisms or flaw in the executing software.

Create
: To open, write data to, and close a file for the first time.

Cross reference listing
: A printed listing that identifies all references in a program to each specific symbol in a program. It includes a list of all symbols used in a source program and the statements where they are defined or used.

Current location counter
: A counter kept by an assembler to determine the address assigned to an instruction or constant being assembled.

Data
: A term used to denote any or all facts, numbers, letters, and symbols. Basic elements of information that can be processed by a computer.

Data base
: An organized collection of interrelated data items that allows one or more applications to process the items without regard to physical storage locations.

Data collection
The act of bringing data from one or more points to a central point for eventual processing.

Debug
To detect, locate, and correct coding or logic errors in a computer program.

Default
The value of an argument, operand, or field assumed by a program if not specifically supplied by the user.

Define
To assign a value to a variable or constant.

Delimiter
A character that separates, terminates, or organizes elements of a character string, statement, or program.

Device
A hardware unit such as an I/O peripheral, magnetic tape drive, card reader, etc. Often used erroneously to mean "volume".

Device control unit
A hardware unit that electronically supervises one or more of the same type of devices. It acts as the link between the computer and the I/O devices.

Device handler
A routine that drives or services an I/O device and controls the physical hardware activities on the device.

Device independence
The ability to program I/O operations independently of the device for which the I/O is intended.

Device name
A unique name that identifies each device unit on a system. It usually consists of a 2-character device mnemonic followed by an optional device unit number and a colon. For example, the common device name for RK05 disk drive unit 1 is "RK1:".

Device unit
One of a set of similar peripheral devices (e.g., disk unit 0, DECtape unit 1, etc.). May be used synonymously with volume.

Diagnostics
Pertaining to a set of procedures for the detection and isolation of a malfunction or mistake.

Digit
A character used to represent one of the non-negative integers smaller than the radix (e.g., in decimal notation, one of the characters 0 to 9; in octal notation, one of the characters 0 to 7; in binary notation, one of the characters 0 and 1).

Direct access
See Random access.

Directive
Assembler directives are mnemonics in an assembly language source program that are recognized by the assembler as commands to control a specific assembly process.

Directory
A table that contains the names of and pointers to files on a mass-storage volume.

Directory-structured
Refers to a storage volume with a true volume directory at its beginning that contains information (file name, file type, length, and date-of-creation) about all the files on the volume. Such volumes include all disks, diskettes, and DECtapes.

Disk device
An auxiliary storage device on which information can be read or written.

Display
A peripheral device used to portray data graphically (normally refers to some type of cathode-ray tube system).

Downtime
The time interval during which a device or system is inoperative.

Echo
The printing by an I/O device, such as terminal or CRT, of characters typed by the programmer.

Edit
To arrange and/or modify the format of data (e.g., to insert or delete characters).

Editor
A program that interacts with the user to enter text into the computer and edit it. Editors are language independent and will edit anything in character representation.

Effective address
The address actually used in the execution of a computer instruction.

Emulator
A hardware device that permits a program written for a specific computer system to be run on a different type of computer system.

Entry point
A location in a subroutine to which program control is transferred when the subroutine is called.

EOT (End Of Tape)
A reflective marker applied to the backside of magtape which precedes the end of the reel.

Error
Any discrepancy between a computed, observed, or measured quantity and the true, specified, or theoretically correct value or condition.

Execute
To carry out an instruction or run a program on the computer.

Expression
A combination of operands and operators that can be evaluated to a distinct result by a computing system.

Extension
Historically-used synonym for file type.

External storage
A storage medium other than main memory, e.g., a disk or tape.

Field
A specified area of a record used for a particular category of data.

FIFO (first in/first out)
A data manipulation method in which the first item stored is the first item processed.

File
A logical collection of data treated as a unit, which occupies one or more blocks on a mass-storage volume such as disk or magtape, and has an associated file name (and file type).

File maintenance
The activity of keeping a mass-storage volume and its directory up to date by adding, changing, or deleting files.

File name
The alphanumeric character string assigned by a user to identify a file. It can be read by both an operating system and a user. A

file name has a fixed maximum length that is system dependent. (The maximum in an RT-11 operating system is six characters, the first of which must be alphabetic. Spaces are not allowed.)

File type
: The alphanumeric character string assigned to a file either by an operating system or a user. It can be read by both the operating system and the user. System-recognizable file types are used to identify files having the same format or type. If present in a file specification, a file type follows the file name in a file specification, separated from the file name by a period. A file type has a fixed maximum length that is system dependent. (The maximum in an RT-11 operating system is three characters, excluding the preceding period and not including any spaces.)

File specification
: A name that uniquely identifies a file maintained in any operating system. A file specification generally consists of at least three components: a device name identifying the volume on which the file is stored, a file name, and a file type.

File-structured device
: A device on which data is organized into files. The device usually contains a directory of the files stored on the volume. (For example, a disk is a file-structured device, but a line printer is not.)

Flag
: A variable or register used to record the status of a program or device; the noting of errors by a translating program.

Floating point
: A number system in which the position of the radix point is indicated by the exponent part and another part represents the significant digits or fractional part (e.g., 5.39×10^8 – Decimal; 137.3×8^4 – Octal; 101.10×2^{13} – Binary).

Flowchart
: A graphical representation for the definition, analysis, or solution of a problem, in which symbols are used to represent operations, data, flow, and equipment.

FOCAL (FOrmula CALculator)
: An on-line interactive, service program designed to help scientists, engineers, and students solve numerical problems. The language consists of short imperative English statements which are easy to learn. FOCAL is used for simulating mathematical models, for curve plotting, for handling sets of simultaneous equations, and for many other kinds of problems.

Foreground
The area in memory designated for use by a high-priority program. The program that gains the use of machine facilities immediately upon request.

FORTRAN (FORmula TRANslation)
A problem-oriented language designed to permit scientists and engineers to express mathematical operations in a form with which they are familiar. It is also used in a variety of applications including process control, information retrieval, and commercial data processing.

Full duplex
In communication, pertaining to a simultaneous, 2-way independent "asynchronous" transmission.

Function
An algorithm accessible by name and contained in the system software which performs commonly-used operations. For example, the square root calculation function.

Garbage
Meaningless signals or bit patterns in memory.

General register
One of eight 16-bit internal registers in the PDP-11 computer. These are used for temporary storage of data.

Global
A value defined in one program module and used in others. Globals are often referred to as entry points in the module in which they are defined and as externals in the other modules that use them.

Hack
A seemingly inspired, but obscure, solution that is superior by some measure to a straightforward one.

Half duplex
Pertaining to a communication system in which 2-way communication is possible, but only one way at a time.

Handler
See device handler.

Hardware
The physical equipment components of a computer system.

Hardware bootstrap
A bootstrap that is inherent in the hardware and need only be activated by specifying the appropriate load and start address.

High-level language
A programming language whose statements are typically translated into more than one machine language instruction. Examples are BASIC, FORTRAN and FOCAL.

High-order byte
The most significant byte in a word. The high-order occupies bit positions 8 through 15 of a PDP-11 word and is always an odd address.

Image mode
Refers to a mode of data transfer in which each byte of data is transferred without any interpretation or data changes.

Indirect address
An address that specifies a storage location containing either a direct (effective) address or another indirect (pointer) address.

Indirect file
A file containing commands that are processed sequentially, but that could have been entered interactively at a terminal.

Industry-standard
A condition, format, or definition that is accepted as the norm by the majority of the (computer) industry.

Initialize
To set counters, switches, or addresses to starting values at prescribed points in the execution of a program, particularly in preparation for re-execution of a sequence of code. To format a volume in a particular file-structured format in preparation for use by an operating system..

Input
The data to be processed; the process of transferring data from external storage to internal storage.

Input/Output device
A device attached to a computer that makes it possible to bring information into the computer or get information out.

Instruction
A coded command that tells the computer what to do and where to find the values it is to work with. A symbolic instruction looks more like ordinary language and is easier for people to deal with. Symbolic instructions must, however, be changed into machine instructions (usually by another program) before they can be executed by the computer.

Interactive processing
A technique of user/system communication in which the operating system immediately acknowledges and acts upon requests entered by the user at a terminal. Compare with batch processing.

Interface
A shared boundary. An interface might be a hardware component to link two devices or it might be a portion of storage or registers accessed by two or more computer programs.

Internal Storage
The storage facilities forming an integral physical part of the computer and directly controlled by the computer, e.g., the registers of the machine and main memory.

Interpreter
A computer program that translates then executes a source language statement before translating (and executing) the next statement.

Interrupt
A signal that, when activated, causes a transfer of control to a specific location in memory, thereby breaking the normal flow of control of the routine being executed.

Interrupt driven
Pertaining to software that uses the interrupt facility of a computer to handle I/O and respond to user requests: RT-11 is such a system.

Interrupt Vector
Two words containing the address of an interrupt service routine and the processor state at which that routine is to execute.

Iteration
Repetition of a group of instructions.

Job
A group of data and control statements which does a unit of work, e.g., a program and all of its related subroutines, data, and control statements; also, a batch control file.

Kluge
A crude, makeshift solution to a problem.

Label
One or more characters used to identify a source language statement or line.

Language
A set of representations, conventions, and rules used to convey information.

Latency
The time from initiation of a transfer operation to the beginning of actual transfer; i.e., verification plus search time. The delay while waiting for a rotating memory to reach a given location.

Library
A file containing one or more macro definitions or one or more relocatable object modules that are routines that can be incorporated into other programs.

LIFO (last in/first out)
A data manipulation method in which the last item stored is the first item processed; a push down stack.

Light pen
A device resembling a pencil or stylus which can detect a fluorescent CRT screen. Used to input information to a CRT display system.

Linkage
In programming, code that connects two separately-coded routines and passes values and/or control between them.

Linked file
A file whose blocks are joined together by references rather than consecutive locations.

Linker
A program that combines many relocatable object modules into an executable module. It satisfies global references and combines program sections.

Listing
The printed copy generated by a line printer or terminal.

Load
To store a program or data in memory. To place a volume on a device unit and put the unit on-line.

Load map
A table produced by a linker that provides information about a load module's characteristics (e.g., the transfer address, the global symbol values, and the low and high limits of the relocatable code).

Load module
A program in a format ready for loading and executing.

Location
An address in storage or memory where a unit of data or an instruction can be stored.

Locked
Pertaining to routines in memory that are not presently (and may never be) candidates for swapping or other shifting around.

Logical device name
An alphanumeric name assigned by the user to represent a physical device. The name can then be used synonymously with the physical device name in all references to the device. Logical device names are used in device-independent systems to enable a program to refer to a logical device name which can be assigned to a physical device at run-time.

Loop
A sequence of instructions that is executed repeatedly until a terminal condition prevails.

Low-order byte
The least significant byte in a word. The low-order byte occupies bit positions 0 through 7 in a PDP-11 word and is always an even address.

Machine instruction
An instruction that a machine can recognize and execute.

Machine language
The actual language used by the computer when performing operations.

Macro
An instruction in a source language that is equivalent to a specified sequence of assembler instructions, or a command in a command language that is equivalent to a specified sequence of commands.

Main program
The module of a program that contains the instructions at which program execution begins. Normally, the main program exercises primary control over the operations performed and calls subroutines or subprograms to perform specific functions.

Manual input
The entry of data by hand into a device at the time of processing.

Mask
A combination of bits that is used to manipulate selected portions of any word, character, byte, or register while retaining other parts for use.

Mass storage
Pertaining to a device that can store large amounts of data readily accessible to the computer.

Matrix
A rectangular array of elements. Any matrix can be considered an array.

Memory
: Any form of data storage, including main memory and mass storage, in which data can be read and written. In the strict sense, memory refers to main memory.

Memory image
: A replication of the contents of a portion of memory, usually in a file.

Mnemonic
: An alphabetic representation of a function or machine instruction.

Monitor
: The master control program that observes, supervises, controls or verifies the operation of a computer system. The collection of routines that controls the operation of user and system programs, schedules operations, allocates resources, performs I/O, etc.

Monitor command
: An instruction or command issued directly to a monitor from a user.

Monitor command mode
: The state of the operating system (indicated by a period at the left margin) which allows monitor commands to be entered from the terminal.

Mount a volume
: To logically associate a physical mass storage medium with a physical device unit. To place a volume on a physical device unit (for example, place a magtape on a magtape drive and put the drive on-line).

Multiprocessing
: Simultaneous execution of two or more computer programs by a computer which contains more than one central processor.

Multiprogramming
: A processing method in which more than one task is in an executable state at any one time, even with one CPU.

Nondirectory-structured
: Refers to a storage volume that is sequential in structure and therefore has no volume directory at its beginning. File information (file name, file type, length, and date-of-creation) is provided with each file on the volume. Such volumes include magtape and cassette.

Nonfile-structured device
: A device, such as paper tape, line printer, or terminal, in which data cannot be organized as multiple files.

Object Code
: Relocatable machine language code.

Object module
: The primary output of an assembler or compiler, which can be linked with other object modules and loaded into memory as a runnable program. The object module is composed of the relocatable machine language code, relocation information, and the corresponding global symbol table defining the use of symbols within the module.

Object Time System
: The collection of modules that is called by compiled code in order to perform various utility or supervisory operations (e.g., FORTRAN object time system).

Octal
: Pertaining to the number system with a radix of eight; for example, octal 100 is decimal 64.

ODT
: On-line Debugging Technique: an interactive program for finding and correcting errors in programs. The user communicates in octal notation.

Off-line
: Pertaining to equipment or devices not currently under direct control of the computer.

Offset
: The difference between a base location and the location of an element related to the base location. The number of locations relative to the base of an array, string, or block.

One's complement
: A number formed by interchanging the bit polarities in a binary number: e.g., 1s become 0s; 0s become 1s.

On-line
: Pertaining to equipment or devices directly connected to and under control of the computer.

Op-code (operation code)
: The part of a machine language instruction that identifies the operation the instruction will ask the CPU to perform.

Operand
: That which is operated upon. An operand is usually identified by an address part of an instruction.

Operating system
: The collection of programs, including a monitor or executive and system programs, that organizes a central processor and peripheral devices into a working unit for the development and execution of application programs.

Operation
: The act specified by a single computer instruction. A program step undertaken or executed by a computer, e.g., addition, multiplication, comparison. The operation is usually specified by the operator part of an instruction.

Operation code
: See op-code.

Operator's console
: The set of switches and display lights used by an operator or a programmer to determine the status of and to start the operation of the computer system.

Option
: An element of a command or command string that enables the user to select from among several alternatives associated with the command. In the RT-11 computer system, an option consists of a slash character (/) followed by the option name and, optionally, a colon and an option value.

Output
: The result of a process; the transferring of data from internal storage to external storage.

Overflow
: A condition that occurs when a mathematical operation yields a result whose magnitude is larger than the program is capable of handling.

Overlay segment
: A section of code treated as a unit that can overlay code already in memory and be overlaid by other overlay segments when called from the root segment or another resident overlay segment.

Overlay structure
: A program overlay system consisting of a root segment and optionally one or more overlay segments.

Page
: That portion of a text file delimited by form feed characters and generally 50-60 lines long. Corresponds approximately to a physical page of a program listing.

Parameter
: A variable that is given a constant value for a specific purpose or process.

Parity
: A binary digit appended to an array of binary digits to make the sum of all bits always odd or always even.

Patch
: To modify a routine in a rough or expedient way, usually by modifying the binary code rather than re-assembling it.

PC
: See Program counter.

PDP
: Programmed data processor.

Peripheral device
: Any device distinct from the computer that can provide input and/or accept output from the computer.

Physical device
: An I/O or peripheral storage device connected to or associated with a computer.

Priority
: A number associated with a task that determines the preference its requests for service receive from the monitor, relative to other tasks requesting service.

Process
: A set of related procedures and data undergoing execution and manipulation by a computer.

Processor
: In hardware, a data processor. In software, a computer program that includes the compiling, assembling, translating, and related functions for a specific programming language (e.g., FORTRAN processor).

Processor status word
: A register in the PDP-11 that indicates the current priority of the processor, the condition of the previous operation, and other basic control items.

Program
: A set of machine instructions or symbolic statements combined to perform some task.

Program counter (PC)
: A register used by the central processor unit to record the locations in memory (addresses) of the instructions to be executed. The PC (register 7 of the 8 general registers) always contains the address of the next instruction to be executed, or the second or third word of the current instruction.

Program development
: The process of writing, entering, translating, and debugging source programs.

Program section
: A named, contiguous unit of code (instructions or data) that is considered an entity and that can be relocated separately without destroying the logic of the program.

Programmed request
: A set of instructions (available only to programs) that is used to invoke a monitor service.

Protocol
: A formal set of conventions governing the format and relative timing of information exchange between two communicating processes.

PSW
: See Processor status word.

Queue
: Any dynamic list of items; for example, items waiting to be scheduled or processed according to system or user assigned priorities.

Radix
: The base of a number system; the number of digit symbols required by a number system.

RAM (random access memory)
: See Random access.

Random access
: Access to data in which the next location from which data is to be obtained is not dependent on the location of the previously obtained data. Contrast Sequential access.

Read-only memory (ROM)
: Memory whose contents are not alterable by computer instructions.

Real-time processing
: Computation performed while a related or controlled physical activity is occurring so that the results of the computation can be used in guiding the process.

Record
: A collection of related items of data treated as a unit; for example, a line of source code or a person's name, rank, and serial number.

Recursive
: A repetitive process in which the result of each process is dependent upon the result of the previous one.

Re-entrant
Pertaining to a program composed of a shareable segment of pure code and a non-shareable segment which is the data area.

Register
See General register.

Relative address
The number that specifies the difference between the actual address and a base address.

Relocate
In programming, to move a routine from one portion of storage to another and to adjust the necessary address references so that the routine, in its new location, can be executed.

Resident
Pertaining to data or instructions that are normally permanently located in main memory.

Resource
Any means available to users, such as computational power, programs, data files, storage capacity, or a combination of these.

Restart
To resume execution of a program.

ROM
See Read-only memory.

Root segment
The segment of an overlay structure that, when loaded, remains resident in memory during the execution of a program.

Routine
A set of instructions arranged in proper sequence to cause a computer to perform a desired operation.

Run
A single, continuous execution of a program.

Sector
A physical portion of a mass storage device.

Segment
See Overlay segment.

Sequential access
Access to data in which the next location from which data is to be obtained sequentially follows the location of the previously obtained data. Contrast Random access.

Software
: The collection of programs and routines associated with a computer (e.g., compilers, library routines).

Software bootstrap
: A bootstrap that is activated by manually loading the instructions of the bootstrap and specifying the appropriate load and start address.

Source code
: Text, usually in the form of an ASCII format file, that represents a program. Such a file can be processed by an appropriate system program.

Source language
: The system of symbols and syntax easily understood by people that is used to describe a procedure that a computer can execute.

Spooling
: The technique by which I/O with slow devices is placed on mass storage devices to await processing.

Storage
: Pertaining to a device into which data can be entered, in which it can be held, and from which it can be retrieved at a later time.

String
: A connected sequence of entities such as a line of characters.

Subprogram
: A program or a sequence of instructions that can be called to perform the same task (though perhaps on different data) at different points in a program, or even in different programs.

Subroutine
: See Subprogram.

Subscript
: A numeric valued expression or expression element that is appended to a variable name to uniquely identify specific elements of an array. Subscripts are enclosed in parentheses. There is a subscript for each dimension of an array. Multiple subscripts must be separated by commas. For example, a two-dimensional subscript might be (2,5).

Supervisory programs
: Computer programs that have the primary function of scheduling, allocating, and controlling system resources rather than processing data to produce results.

Swapping
: The process of moving data from memory to a mass storage device, temporarily using the evacuated memory area for another purpose, and then restoring the original data to memory.

Synchronous
: Pertaining to related events where all changes occur simultaneously or in definite timed intervals.

Syntax
: The structure of expressions in a language and the rules governing the structure of a language.

System program
: A program that performs system-level functions. Any program that is part of or supplied with the basic operating system (e.g., a system utility program).

System volume
: The volume on which the operating system is stored.

Table
: A collection of data into a well-defined list.

Terminal
: An I/O device, such as an LA36 terminal, that includes a keyboard and a display mechanism. In PDP-11 systems, a terminal is used as the primary communication device between a computer system and a person.

Timesharing
: A method of allocating resources to multiple users so that the computer, in effect, processes a number of programs concurrently.

Toggle
: To use switches on the computer operator's console to enter data into the computer memory.

Translate
: To convert from one language to another.

Trap
: A conditional jump to a known memory location performed automatically by hardware as a side effect of executing a processor instruction. The address location from which the jump occurs is recorded. It is distinguished from an interrupt which is caused by an external event.

Truncation
: The reduction of precision by ignoring one or more of the least significant digits; e.g., 3.141597 truncated to four decimal digits is 3.141.

Turnkey
: Pertaining to a computer system sold in a ready-to-use state.

Two's complement
: A number used to represent the negative of a given value in many computers. This number is formed from the given binary value by changing all 1s to 0s and all 0s to 1s and then adding 1.

Underflow
: A condition that occurs when a mathematical operation yields a result whose magnitude is smaller than the smallest amount the program can handle.

User program
: An application program.

Utility program
: Any general-purpose program included in an operating system to perform common functions.

Variable
: The symbolic representation of a logical storage location that can contain a value that changes during a processing operation.

Vector
: A consecutive list of associated data.

Volume
: A mass storage medium that can be treated as file-structured data storage.

Wildcard operation
: A shorthand method of referring to all files with a specific characteristic in their name.

Word
: Sixteen binary digits treated as a unit in PDP-11 computer memory.

Write-enable
: The condition of a volume that allows transfers that would write information on it.

Write-protect
: The condition of a volume that is protected against transfers that would write information on it.

INDEX

READER'S COMMENTS

NOTE: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. Problems with software should be reported on a Software Performance Report (SPR) form. If you require a written reply and are eligible to receive one under SPR service, submit your comments on an SPR form.

Did you find errors in this manual? If so, specify by page.

Did you find this manual understandable, usable, and well-organized? Please make suggestions for improvement.

Is there sufficient documentation on associated system programs required for use of the software described in this manual? If not, what material is missing and where should it be placed?

Please indicate the type of user/reader that you most nearly represent.

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Non-programmer interested in computer concepts and capabilities

Name ______________________ Date ______________________

Organization ______________________

Street ______________________

City ______________________ State ____________ Zip Code ____________
or
Country

Please cut along this line.

- Fold Here -

- - - - - - - - - - - - - - - Do Not Tear - Fold Here and Staple - - - - - - - - - - - - - - -

FIRST CLASS
PERMIT NO. 33
MAYNARD, MASS.

BUSINESS REPLY MAIL
NO POSTAGE STAMP NECESSARY IF MAILED IN THE UNITED STATES

Postage will be paid by:

digital

Software Documentation
146 Main Street ML 5-5/E39
Maynard, Massachusetts 01754

pdp11

RT-11 System User's Guide

Order No. DEC-11-ORGDA-A-D

digital

CONTENTS

CONTENTS (Cont.)

CONTENTS (Cont.)

CONTENTS (Cont.)

CONTENTS (Cont.)

CONTENTS (Cont.)

CONTENTS (Cont.)

CONTENTS (Cont.)

CONTENTS (Cont.)

FIGURES

FIGURES (Cont.)

TABLES

CONTENTS (Cont.)

TABLES (Cont.)

PREFACE

This manual describes how to use the RT-11 system; it provides enough information for you to perform ordinary tasks such as program development, program execution, and file maintenance. This manual is appropriate for you if you are already familiar with computer software fundamentals and have some experience using RT-11. If you have no RT-11 experience, you should read the *Introduction to RT-11* before consulting this manual. If you have experience with an earlier release of RT-11 (this is version 3), you should read the *RT-11 System Release Notes* to learn how RT-11 V03 differs from earlier versions. If you are interested in more sophisticated programming techniques or in system programming, you should read this manual first and then proceed to the *RT-11 Advanced Programmer's Guide*.

The next section, Chapter Summary, briefly describes the chapters in this manual and suggests a reading path to help you use the manual efficiently.

CHAPTER SUMMARY
The first two chapters make up Part I of this manual, RT-11 Overview. Read Part I to gain an understanding of the RT-11 system as a whole.

Chapter 1 describes the program development process in general as well as the system software and hardware components.

Chapter 2 describes the three monitors that are available with an RT-11 system.

Chapters 3 and 4 compose Part II of the manual, System Communication. Read Part II to become familiar with RT-11 system conventions and to learn how to interact with the RT-11 monitor directly from the console terminal.

Chapter 3 describes system conventions, such as data formats, file naming conventions, and terminal keyboard special functions.

Chapter 4 introduces the keyboard monitor commands. These important commands are your means of communicating with the monitor and performing computer tasks.

Part III, Text Editing, consists of Chapter 5, EDIT. Read Chapter 5 to learn how to manipulate text on the RT-11 system.

Part IV, Utility Programs, consists of 10 chapters that describe the many programs provided with the RT-11 system. If you are an advanced user, you may want to read Chapters 6 through 15 to learn about the RT-11 system programs in detail. However, if you are a new user or primarily a high-level language programmer, you do not have to understand how these system programs work to make use of them through the monitor command language (described in Chapter 4).

Chapter 6 describes the Command String Interpreter and explains the command syntax you use to communicate with the RT-11 system programs.

Chapters 7 through 9 describe the RT-11 system utility programs, PIP, DUP, and DIR.

Chapter 10 describes MACRO, the RT-11 assembly language.

Chapters 11 through 15 describe the RT-11 system utility programs, LINK, LIBR, DUMP, FILEX, and SRCCOM.

Part V, Altering Assembled Programs, explains the use of some sophisticated programming tools.

Chapters 16 through 18 describe the RT-11 programs, ODT, PATCH, and PAT. These three programs can help you debug programs and make changes to programs that are already assembled.

Appendix A contains a description of RT-11 BATCH. Appendix B contains a table of the keyboard monitor commands, their abbreviations, and their system program equivalents.

DOCUMENTATION CONVENTIONS

This section describes the symbolic conventions used throughout this manual. Familiarize yourself with these conventions before you continue reading the manual.

Conventions used in this manual include the following items:

1. Examples consist of actual computer output wherever possible. In the examples, responses entered by a user are shown in red to distinguish them from computer output, which is shown in black.
2. Unless the manual indicates otherwise, terminate all commands or command strings with a carriage return. Where necessary, this manual uses the symbol (RET) to represent a carriage return, (LF) to represent a line feed, (SP) for a space, and (TAB) to represent a tab.
3. Terminal and console terminal are general terms used throughout all RT-11 documentation to represent any terminal device, including DECwriters, displays, and Teletypes[1].
4. To produce several characters in system commands you must type a combination of keys concurrently. For example, hold down the CTRL key and type O at the same time to produce the CTRL/O character. Key combinations such as this one are documented as CTRL/O, CTRL/C, SHIFT/N, etc.
5. In descriptions of command syntax, capital letters represent the command name, which you must type. Lower case letters represent a variable, for which you must supply a value.

   Square brackets [ ] enclose optional choices; you can include the item in brackets, or you can omit it, as you choose.

   Braces { } enclose a group of options from which you can choose only one.

   The ellipsis symbol (. . .) represents repetition. You can repeat the item that precedes the ellipsis.

   The hyphen (-) is a continuation character. Use it at the end of a line if you continue a command string to another line.

   The following is a typical example of command syntax:

   ```
   DELETE[/option . . .] filespec[/option . . .]
   ```

   This example shows that you must type the word DELETE, and that you can follow it with one or more options of your choice (none are required). You must then leave a space and supply a file specification. The file specification can also be followed by one or more options (none are required). Here is a typical command string:

   ```
   .DELETE/NOQUERY DT1:MYFILE.FOR
   ```

[1]Teletype is a registered trademark of the Teletype Corporation.

PART I
RT-11 OVERVIEW

RT-11 is a single-user programming and operating system for the PDP-11 series of computers. This system can use a wide range of peripherals and can access up to 124K (126,976) words of either solid state or core memory. (4K words of the maximum 128K (131,072) words of memory are reserved for device interfacing.)

Three system monitors are provided by RT-11: the single-job monitor (SJ), the foreground/background monitor (FB), and the extended memory monitor (XM).

The single-job monitor allows one program at a time to reside in memory. The program executes until it completes or until you interrupt it with a keyboard command.

The foreground/background monitor allows two independent programs to reside in memory at one time. The foreground program, however, takes priority over the background program. RT-11 allows the background program to execute whenever the foreground program is in a wait state. Typically, the foreground program performs a time-dependent task, such as sampling material every few seconds and then analyzing the resultant data. A background program, on the other hand, usually performs a time-independent task, such as file maintenance or program development. This sharing of resources between two tasks greatly increases the efficiency of your RT-11 system.

The extended memory monitor provides all the features of the foreground/background monitor and, in addition, allows you to access up to 124K (126,976) words of memory. The other two monitors are restricted to 28K words of main memory. (4K words of the 32K words of memory available are reserved for device interfacing.)

These three monitors are upward compatible. That is, the foreground/background monitor provides all the features of the single-job monitor, and the extended memory monitor offers all the features of the foreground/background monitor.

You control the RT-11 system from the console terminal. The monitor commands that you use to direct the system are described in Chapter 4 of this manual.

In addition to the three monitors, RT-11 provides a full complement of system programs that can perform some more specific tasks than the keyboard monitor commands can. If you are an average user, though, the keyboard monitor commands should be sufficient for your needs. There is a summary of the system programs in Section 1.2; they are described in more detail in individual chapters of this manual.

RT-11 also supports a variety of language processors including MACRO-11, an assembly language, and several high-level languages such as FORTRAN IV and BASIC.

The following two chapters describe system software and hardware components, program development, and the three RT-11 monitors.

CHAPTER 1
SYSTEM COMPONENTS

This chapter describes briefly the software and hardware components available for you to use with the RT-11 system. The software components include the text editor and the many system programs that perform specific tasks. The hardware components include system clocks, printing and display terminals, external storage devices (such as magnetic tape drives), and other peripheral devices (such as card readers and line printers).

1.1 PROGRAM DEVELOPMENT

Computer systems (such as RT-11) are ideal for program development. You can make use of the programming tools available on your system to develop programs to suit your needs. The number and type of tools available on any given system depend on many factors (including the size of the system, its application, and its cost). Most DIGITAL systems, however, provide several basic program development aids. These aids generally include an editor, an assembler, a linker, a debugger, and a librarian. A high level language, such as FORTRAN or BASIC, is also usually available.

You can use an editor to create and modify textual material. Text may be the lines of code that make up a source program written in some programming language, or it may be other ASCII data. Text may be reports, memos, or, in fact, any subject matter you wish. In this respect, using an editor is analogous to using a typewriter; you sit at a keyboard and type text. However, the advantages of an editor far exceed those of a typewriter. Once text has been created, you can modify, relocate, replace, merge, or delete it, all by means of simple editing commands. When you are satisfied with your text, you can save it on a storage device where it is available for later reference.

If you use the editor to write a source program, development does not stop with the creation of this program. Since the computer cannot understand any language but machine language (which is a set of binary command codes), you need an intermediary program to convert source code into the instructions the computer can execute. This is the function of an assembler or language translator.

The assembler accepts alphanumeric representations of PDP-11 coding instructions (i.e., mnemonics), interprets the code, and produces as output the appropriate object code. You can direct the assembler to generate a listing of both the source code and binary output, as well as more specific listings that are helpful during the program debugging process. In addition, the assembler is capable of detecting certain common coding errors and issuing appropriate warnings.

The assembler produces output called object output because it is composed of object (or binary) code. On PDP-11 systems, the object output is called a module; it contains your source program in the binary language that is acceptable to a PDP-11 computer.

Source programs may be complete and functional by themselves; however, some programs are written in such a way that they must be used with other programs (or modules) to form a complete and logical flow of instructions. For this reason, the object code produced by the assembler must be relocatable. That is, assignment of memory locations must be deferred until the code is combined with all other necessary object modules. The linker performs this function.

The linker combines and relocates separately-assembled object programs. The output produced by the linker is a load module, the final linked program that is ready for execution. You can, at your choice, request a load map that displays all addresses assigned by the linker.

You can very rarely create a program that does not contain at least one unintentional error, either in the logic of the program or in its coding. You may discover errors while you are editing your program, or the assembler may find errors during the assembly process and inform you by means of error codes. The linker may also catch certain errors and issue appropriate messages. Often, however, it is not until execution that you discover that your program is not working properly. Programming errors may be extremely difficult to find, and for this reason, a debugging tool is usually available to aid you in determining the cause of your error.

A debugging program allows you to interactively control the execution of your program. With it, you can examine the contents of individual locations, search for specific bit patterns, set designated stopping points during execution, change the contents of locations, continue execution, and test the results, all without editing and reassembling the program.

When programs are successfully written and executed, they are useful to other programmers. Often, routines that are common to many programs (such as input and output routines) or sections of code that are used over and over again, are more useful if they are placed in a library where they can be retrieved by any interested user. A librarian provides such a service by allowing creation of a library file. Once created, the library can be expanded, updated, or listed.

High-level languages simplify your work by providing an alternate means, other than assembly language mnemonics, of writing a source program. Generally, high-level languages are easy to learn. A single command causes the computer to perform many machine-language instructions. You do not need to know about the mechanics of the computer to use a high-level language. In addition, some high-level languages (like BASIC) offer a special immediate mode that allows you to solve equations and formulas as though you were using a calculator. You can concentrate on solving the problem rather than on using the system.

These are a few of the programming tools offered by most computer systems. The next section summarizes specific programming aids available to you as an RT-11 user.

1.2 SYSTEM SOFTWARE COMPONENTS

The following is a brief summary of the specific system programs and programming available to you as an RT-11 user:

1. The keyboard monitor commands (described in Chapter 4) are your means of controlling the system. You can use these English-language commands to perform file maintenance, library maintenance, handler modification, program development, and program execution. If you are an average user, the keyboard monitor commands should be sufficient for your needs.
2. The text editor (EDIT, described in Chapter 5) creates or modifies source files for use as input to language-processing programs such as the assembler or FORTRAN. EDIT contains text manipulation commands that permit quick and easy editing of a text file. EDIT also allows you to use a VT11 or VS60 display processor if one is part of the hardware configuration.
3. The peripheral interchange program (PIP, described in Chapter 7) is the RT-11 file maintenance program. It transfers files among all devices that are part of the RT-11 system and renames or deletes files.
4. The device utility program (DUP, described in Chapter 8) performs general device utilities such as initializing devices, duplicating their contents, and reorganizing files on the devices. It operates only on RT-11 file-structured devices.
5. The directory program (DIR, described in Chapter 9) produces directory listings.
6. The MACRO assembler (described in Chapter 10) is a 2-pass assembler that assembles one or more ASCII source files of statements and assembler language instructions into a single binary object file.
7. The linker (LINK, described in Chapter 11) converts a collection of object modules from compiled or assembled programs and subroutines into a memory image file that RT-11 can load and execute. LINK provides some optional features that:

   a. Search library files for subroutines that you specify
   b. Produce a load map that lists the assigned absolute addresses
   c. Provide overlay capabilities to very large programs
   d. Produce files suitable for execution in the foreground.

8. The librarian (LIBR, described in Chapter 12) lets you create and maintain libraries of functions and routines. These routines are stored on a random access device in library files, where the linker can reference them. You can also create MACRO libraries to be used by the MACRO assembler.
9. DUMP (described in Chapter 13) prints for examination all or any part of a file in octal words, octal bytes, ASCII and/or Radix-50 characters.
10. The file exchange utility (FILEX, described in Chapter 14) transfers files between DECsystem-10, PDP-11 RSTS, and DOS BATCH on DECtape and disks, and between RT-11 and IBM systems on diskettes.
11. The source compare utility (SRCCOM, described in Chapter 15) performs a character-by-character comparison of two ASCII text files. You can request that the differences be listed in an output file or directly on the line printer or terminal to ensure that edits have been performed correctly.
12. On-line debugging technique (ODT, described in Chapter 16) aids you in debugging assembled and linked object programs. It can:

    a. Print and optionally change the contents of specified locations
    b. Execute all or part of the object program
    c. Single-step through the program
    d. Search the object program for bit patterns.

13. The patching utility program (PATCH, described in Chapter 17) performs minor modifications to memory image files (output files produced by the linker).
14. The object module patching program (PAT, described in Chapter 18) performs minor modifications to files in object format (output files produced by the FORTRAN compiler or the MACRO assembler). It can merge several object files into one.
15. The RT-11 FORTRAN system subroutine library (described in the *RT-11 Advanced Programmer's Guide*) is a collection of FORTRAN callable routines that make the programmed requests and various utility functions available to you as a FORTRAN programmer. This library also provides a string manipulation package and 2-word integer package for RT-11 FORTRAN.
16. BATCH (Appendix A) is a complete job-control language that allows RT-11 to operate unattended.

1.3 SYSTEM HARDWARE COMPONENTS

The smallest RT-11 system, one that uses the SJ monitor exclusively, requires a PDP-11 series computer with at least 8K words of memory, a random-access device, and a console terminal. The addition of the FB monitor requires another 8K words of memory and either a line frequency or a programmable clock. The addition of the XM monitor requires a KT11 memory management unit and still another 8K words of memory.

The RT-11 operating system adapts itself to take advantage of any amount of memory on a system and does not need to be reconfigured for a particular memory size. The SJ monitor operates in systems ranging from 8K words to 28K words in memory size. The FB monitor operates in systems ranging from 16K words to 28K words in memory size. The XM monitor operates in systems ranging from 24K words to 124K (126,976) words in memory size.

Table 1-1 lists the devices that RT-11 supports.

Table 1-1 RT-11 Hardware Components

Type	Controller	Device
Disk		
Cartridge	RK11	RK05/RK05F
	RK611	RK06
Fixed-head	RF11	RS11
	RH11	RJS03, RJS04
Removable Pack	RP11	RP02, RP03
Diskette	RX11	RX01
DECtape	TC11	TU56
Magtape	TM11/TMA11	TU10, TS03
	RH11	TJU16, TU45
Cassette	TA11	TU60
High-Speed Paper Tape Reader/Punch	PC11	PC11 (both)
	PR11	PR11 (reader only)
Line Printer	LS11	LS11, LA180
	LV11	LV11 (printer only)
	LP11	all LP11 controlled printers
Card Reader	CR11	CR11
	CM11	CM11
Terminal	DL11	LT33, LT35, LA30P, LA36, LA120, VT50, VT52, VT55, VT05 VT61
Display Processor	VT11	VR14-L, VR17-L
	VS60	
Clock		KW11-L, KW11-P
Terminal and Clock	DL11-W	terminal/clock combination

CHAPTER 2
OPERATING ENVIRONMENTS

The RT-11 system offers three complete operating environments: single-job (SJ) operation, foreground/background (FB) operation, and extended memory (XM) operation. You control each environment with the appropriate monitor: SJ, FB, and XM.

You must define your needs before deciding which environment to use and consequently which monitor to run. The following sections provide information to help you ascertain which monitor is suitable for your application.

2.1 RT-11 SINGLE-JOB MONITOR

The RT-11 single-job monitor provides a single-user, single-program system that can operate in as little as 8K words of memory. The SJ monitor is useful for extensive program development; since the monitor itself requires only 2K words of memory, there are at least 6K words left for your program and its buffers and tables. The SJ environment is also suitable for running programs that require a high data transfer rate, since the SJ monitor services interrupts quickly.

You can use all the system programs (listed in Section 1.2) under the SJ monitor. Monitor commands and programmed requests are also available to you as an SJ user.

In summary, the SJ monitor is smaller and faster than the FB and XM monitors; it is most useful when you are concerned with program size versus available memory and when you need a dedicated system.

2.2 RT-11 FOREGROUND/BACKGROUND MONITOR

Quite often, the central processor of a computer system spends much of its time waiting for some external event to occur. Usually, this event is a real-time interrupt or the completion of an I/O transfer. This situation is particularly true of real-time jobs. The foreground/background environment lets you take advantage of the unused processor capacity to accomplish lower-priority tasks.

In a foreground/background system, the foreground job is the time-critical, real-time job, and the FB monitor gives it priority over the background job. Whenever the foreground job reaches a state in which no useful processing can be done until some external event occurs, the monitor executes the background job, if possible. The background job then runs until the foreground job is again ready to execute. The processor then interrupts the background job and resumes the foreground job.

In effect, the RT-11 foreground/background monitor allows a time-dependent job to run in the foreground while a time-independent job, such as program development, runs in the background. All RT-11 system programs can run as the background job in a FB system. Thus, you can run FORTRAN, BASIC, MACRO, etc. in the background while the foreground is collecting, storing, and analyzing data. In addition, the FB monitor gives you the ability to set timer routines, suspend and resume FB jobs, and send data and messages between the two jobs. The FB monitor is most often used for laboratory work, data acquisition, and real-time applications.

You can link most of the programs you write for an RT-11 system to run as foreground jobs. There are a few coding restrictions, which are explained in the *RT-11 Advanced Programmer's Guide*. A foreground program has access to all of the features available to the background job (opening and closing files, reading and writing data, etc.).

2.3 RT-11 EXTENDED MEMORY MONITOR

The extended memory monitor (XM) is an extension of the foreground/background (FB) environment. Generally, references in this manual to FB operation also apply to XM operation. The single-job monitor does not support extended memory. The XM monitor permits either foreground or background jobs to extend their effective logical program space beyond the 32K word restriction imposed by the 16-bit address word of the PDP-11 processors. The XM monitor manages extended memory space as a system resource and dynamically allocates it as you request. A program can map selected portions of its addressing space into extended memory by means of a set of programmed requests. A detailed description of extended memory and how to use it appears in the *RT-11 Advanced Programmer's Guide*.

2.4 FACILITIES AVAILABLE ONLY IN RT-11 FB

Some features available to you as a FB user include:

1. **Mark time.** The .MRKT programmed request allows your program to set clock timers for specified amounts of time. When the timer runs out, the system enters the routine that you specify. You can enter as many mark time requests as you need, providing that you reserve system queue space. The mark time feature is available to SJ monitor users as a SYSGEN option.
2. **Timed wait.** The .TWAIT programmed request allows your program to "sleep" until a period of time that you specify elapses. A foreground program, for example, may need to act on sample data and write it to mass storage once every few minutes. While the foreground program is idle, the background program can run.
3. **Send data, receive data.** The .SDAT and .RCVD programmed requests permit the foreground and background programs to communicate with each other. The send and receive data functions let one program send messages or data of variable size blocks to the other program. For example, you can transfer data directly from a foreground collection program to a background analysis program.
4. **Channel copy.** The .CHCOPY programmed request allows two programs to share the same data file.
5. **Device.** The .DEVICE programmed request allows you to turn off specific devices upon program termination.
6. **Protect.** The .PROTECT programmed request lets you protect the vectors that one program uses from interference by another program.
7. **Channel status.** The .CSTAT programmed request returns status data about an open channel.

You can learn more about these programmed requests and how to use them in Chapter 2 of the *RT-11 Advanced Programmer's Guide*.

2.5 FACILITIES AVAILABLE ONLY IN RT-11 XM

An optional extension of the FB environment is the extended memory monitor (XM), which permits you to extend the logical address space for either foreground or background jobs. Some features available to you only when you use the XM monitor are:

1. **Create a region.** The .CRRG programmed request allows you to allocate a region in extended memory for the current program.
2. **Eliminate a region.** The .ELRG programmed request eliminates an extended memory region and returns it to the free list so it can be used by other programs.
3. **Create an address window.** The .CRAW programmed request unmaps and eliminates conflicting address windows, creates new windows to address extended memory, and maps new windows to the regions you specify. It directs the monitor to give the program a window into the region it has created. This request allows the program to access the physical memory as if it were local to the program.
4. **Eliminate an address window.** The .ELAW programmed request unmaps and eliminates address windows.
5. **Map.** The .MAP programmed request lets you map and remap windows.
6. **Status.** The .GMCX programmed request returns status data about window mapping.
7. **Unmap.** The .UNMAP programmed request lets you unmap a window.

You can learn more about these programmed requests and how to use them in Chapter 3 of the *RT-11 Advanced Programmer's Guide*.

PART II
SYSTEM COMMUNICATION

The monitor is the center of RT-11 system communications; it provides access to system and user programs, performs input and output functions, and enables control of background and foreground jobs.

You communicate with the monitor through programmed requests and keyboard commands. You can use the keyboard commands (described in Chapter 4) to load and run programs, start or restart programs at specific addresses, modify the contents of memory, and assign and deassign alternate device names, to name only a few of the functions.

Programmed requests (described in detail in Chapter 2 of the *RT-11 Advanced Programmer's Guide*) are source program instructions that request the monitor to perform monitor services. These instructions allow assembly language programs to use the available monitor features. A running program communicates with the monitor through programmed requests. FORTRAN programs have access to programmed requests through the system subroutine library. Programmed requests can, for example, manipulate files, perform input and output, and suspend and resume program operations.

The two chapters in this part describe system conventions and contain information that helps you get started with RT-11. Chapter 4 introduces the keyboard monitor commands, which are your means of controlling the RT-11 system.

CHAPTER 3
SYSTEM CONVENTIONS

This chapter contains information to help you start using the RT-11 system. It describes:

- Startup procedure
- Data formats
- Physical device names
- File names and file types
- Device structures
- Special function keys
- Foreground input and output
- Monitor type-ahead feature

Before you operate the RT-11 system, you should be familiar with the special character commands, file naming procedures and other conventions that are standard to the system. These conventions are described in this chapter.

3.1 SYSTEM STARTUP

For information on building the system and loading the monitor, refer to the *Introduction to RT-11*, to the *RT-11 System Generation Manual*, or to any instructions provided by your DIGITAL representative.

When the system has been built and you load the monitor into memory, the monitor prints one of the following identification messages on the terminal:

```
RT-11SJ Vnnx-nnx
RT-11FB Vnnx-nnx
RT-11XM Vnnx-nnx
```

The message that prints indicates which monitor (SJ, FB, or XM) is loaded; you establish which is to be loaded during the system build operation.

Vnnx represents the version and release number of the monitor – for example, V03, for Version 3 (release A). nnx represents the library submission number and the patch level – for example, 01B, for library number 1 (patch level B).

As soon as a monitor takes control of the system, it attempts to execute keyboard monitor commands from an indirect file called STARTS.COM for the SJ monitor, STARTF.COM for the FB monitor, and STARTX.COM for the XM monitor. You can place commands in this startup file to perform routine tasks for you, such as assigning logical device names to physical devices or setting the current date. (Indirect files are discussed in Section 4.3.) If the monitor does not find the appropriate file, it issues a warning message. The system then prints its prompt (.) indicating that it is ready to accept commands. You should now write-enable the system device.

To bring up an alternate monitor while under control of the one currently running, use the BOOT command described in Section 4.4 of this manual.

3.2 DATA FORMATS

The RT-11 system stores data in two formats: ASCII and binary. The binary data can be organized in many formats, including object, memory image, relocatable image, and load image.

Files in ASCII format conform to the American National Standard Code for Information Interchange, in which each character is represented by a 7-bit code. Files in ASCII format include program source files created by the editor and BASIC, listing and map files created by various system programs, and data files consisting of alphanumeric characters.

Files in binary object format consist of data and PDP-11 machine language code. Object files are the files the assembler or FORTRAN compiler outputs; they are used as input to the linker.

The linker can output files in one of three formats: 1) memory image format (.SAV), 2) relocatable image format (.REL), or 3) load image format (.LDA).

A memory image file (.SAV) is a picture of what memory looks like after you load a program. The file itself requires the same number of disk blocks as the corresponding number of 256-word memory blocks. A memory image file does not require relocation, and can run in an SJ environment or as a background program under the FB or XM monitor.

A relocatable image file (.REL) differs from a memory image file. Although the relocatable file is linked as though its bottom address were 1000, relocation information is included with its memory image. When you call the program with the FRUN command, the file is relocated as it is loaded into memory. A relocatable image file can run in a foreground environment.

You can produce a load image (.LDA) file for compatibility with the PDP-11 paper tape system. The absolute binary loader loads this file. You can load and execute load image files in stand-alone environments without relocating them.

There are a number of other types of binary data that different parts of the RT-11 system use in addition to the more common types listed here.

3.3 PHYSICAL DEVICE NAMES

When you request services from the monitor, it is sometimes necessary to specify a physical peripheral device on which the service is to be performed. You can reference devices by means of a standard 2-character device name. Table 3-1 lists each name and its related device. If you do not specify a unit number for devices with more than one unit, the system assumes unit 0.

In addition to using the fixed names shown in Table 3-1, you can assign logical names to devices. A logical name takes precedence over a physical name and thus provides device independence. With this feature, you do not have to rewrite a program that is coded to use a specific device if the device becomes unavailable. You associate logical names with physical devices by using the ASSIGN command, which is described in Section 4.4.

3.4 FILE NAMES AND FILE TYPES

You can reference files symbolically by a name of one to six alphanumeric characters (followed, optionally, by a period and a file type of up to three alphanumeric characters). No spaces or tabs are allowed in the file name or file type. The file type generally indicates the format or contents of a file. It is a good practice to conform to the standard file types for RT-11. If you do not specify a file type for an input or output file, most system programs assign an appropriate default file type. Table 3-2 lists the standard file types used in RT-11.

3.5 DEVICE STRUCTURES

RT-11 devices are categorized according to two characteristics: 1) the device's physical structure and 2) the device's method of processing information. All RT-11 devices are either randomly accessed or sequentially accessed.

Random-access devices allow the system to process blocks of data in random order – that is, independent of the data's physical location on the device or its location relative to any other information. All disks and DECtape fall into this category. Random-access devices are sometimes called block-replaceable devices, because you can manipulate (rewrite) individual data blocks without affecting other data blocks on the device.

Table 3-1 Permanent Device Names

Permanent Name	I/O Device
CR:	CR11/CM11 Card Reader
CTn:	TA11 Cassette (n is 0 or 1)
DK:	The default logical storage device for all files. DK: is initially the same as SY:
DKn:	The specified unit of the same device type as DK: if DK: is unassigned
DMn:	RK06 Disk (n is an integer in the range 0-7)
DPn:	RP02, RP03 Disk (n is an integer in the range 0-7)
DSn:	RJS03/4 Fixed-Head Disks (n is an integer in the range 0-7)
DTn:	DECtape (n is an integer in the range 0-7)
DXn:	RX01 Floppy Disk (n is an integer in the range 0-3)
EL:	Error Logging Handler
LP:	Line Printer
MMn:	TJU16/TU45 (industry compatible) Magtape (n is an integer in the range 0-7)
MTn:	TM11/TMA11/TS03 (industry compatible) Magtape (n is an integer in the range 0-7)
NL:	Null device
PC:	PC11 combined High-Speed Paper Tape Reader and Punch
RF:	RF11 Fixed-Head Disk Drive
RKn:	RK05 Disk Cartridge Drive (n is an integer in the range 0-7)
SY:	The default logical system device; the device and unit from which the system is bootstrapped
SYn:	The specified unit of the same device type as SY: if SY: is unassigned
TT:	Console Terminal Keyboard and Printer

Table 3-2 Standard File Types

File Type	Meaning
.BAD	Files with bad (unreadable) blocks; you can assign this file type whenever bad areas occur on a device. The .BAD file type makes the file permanent in that area, preventing other files from using it and consequently becoming unreadable
.BAK	Editor backup file
.BAS	BASIC source file (BASIC input)
.BAT	BATCH command file
.COM	Indirect file
.CTL	BATCH control file generated by the BATCH compiler
.CTT	BATCH internal temporary file
.DAT	BASIC or FORTRAN data file
.DBL	DIBOL source file
.DIF	SRCCOM output file
.DIR	Directory listing file
.DMP	DUMP output file
.FOR	FORTRAN IV source file (FORTRAN input)
.LDA	Absolute binary file (optional linker output)
.LOG	BATCH log file
.LST	Listing file (MACRO, FORTRAN, LIBR, or DIBOL output)
.MAC	MACRO source file (MACRO or SRCCOM input)
.MAP	Map file (linker output)
.OBJ	Relocatable binary file (MACRO or FORTRAN output, linker input, LIBR input and output)
.REL	Foreground job relocatable image (linker output, default for monitor FRUN command)
.SAV	Memory image; default for R, RUN, SAVE and GET keyboard monitor commands; also default for linker output
.SML	System MACRO library
.SOU	Temporary source file generated by BATCH
.STB	Symbol table file in object format containing all the symbols produced during a link
.SYS	System files and handlers

Sequential-access devices require sequential processing of data; the order in which the system processes the data must be the same as the physical order of the data. RT-11 devices that are sequential devices are magtape, cassette, paper tape reader and punch, card reader, line printer, terminal, and the null device.

File-structured devices are those devices that allow the system to store data under assigned file names. RT-11 devices that are file-structured include all disk, DECtape, magtape, and cassette devices. Non-file-structured devices, however, contain a single logical collection of data. These devices, including the line printer, card reader, terminal, and paper tape reader and punch, are generally used for reading and listing information.

File-structured devices that have a standard RT-11 directory at the beginning are RT-11 directory-structured devices. A device directory consists of a series of directory segments that contain the names and lengths of the files on that device. The system updates the directory each time a program moves, adds, or deletes a file on the device. The *RT-11 Software Support Manual* contains a more detailed explanation of a device directory. RT-11 directory-structured devices include all disks and DECtape. Non-RT-11 directory-structured devices are file-structured devices that do not have the standard RT-11 directory structure. For example, some devices, such as magtape and cassette, store directory-type information at the beginning of each file, but the system must read the device sequentially to obtain all information about all files.

The *RT-11 Software Support Manual* explains methods of interfacing a device with a user-defined directory structure to the RT-11 system.

Table 3-3 shows the relationships among devices, access methods, and structures.

Table 3-3 Device Structures

Device	Random-Access	Sequential-Access	File-Structured	Non-file-Structured	RT-11 directory-Structured	Non-RT-11 directory-Structured
Disk	x		x		x	
DECtape	x		x		x	
Magtape		x	x			x
Cassette		x	x			x
Paper tape		x		x		
Card reader		x		x		
Line printer		x		x		
Terminal		x		x		

3.6 SPECIAL FUNCTION KEYS

Special function keys and keyboard commands let you communicate with the RT-11 monitor to allocate system resources, manipulate memory images, start programs, and use foreground/background services.

The special functions of certain terminal keys you need for communication with the keyboard monitor are explained in Table 3-4. In the FB system, the keyboard monitor runs as a background job when no other background job is running.

Enter CTRL commands by holding the CTRL key down while typing the appropriate letter.

3.7 FOREGROUND/BACKGROUND TERMINAL I/O

Console input and output under FB are independent functions; therefore, you can type input to one job while another job prints output. You may be in the process of typing input to one job when the system is ready to print output from the other job on the terminal. In this case, the job that is ready to print interrupts you and prints the message on the terminal; the system does not redirect input control to this job, however, unless you type a CTRL/B or CTRL/F. If

Table 3-4 Special Function Keys

Key	Function
CTRL/A	CTRL/A is valid only after you type the monitor GT ON command and use the display. CTRL/A, a command that does not echo on the terminal, pages output if you use it after a CTRL/S. The system permits console output to resume until the screen is completely filled again; text currently displayed scrolls upward off the screen. CTRL/A has no special meaning if GT ON is not in effect.
CTRL/B	CTRL/B causes the system to direct all keyboard input to the background job. The FB monitor echoes B> on the terminal. The system takes at least one line of output from the background job. The foreground job, however, has priority, so the system returns control to the foreground job when it has output. CTRL/B directs all typed input to the background job until a CTRL/F redirects input to the foreground job. CTRL/B has no special meaning when used under a single-job monitor or when a SET TT NOFB command is in effect.
CTRL/C	CTRL/C terminates program execution and returns control to the keyboard monitor. CTRL/C echoes^C on the terminal. You must type two CTRL/Cs to terminate execution unless the program to be terminated is waiting for terminal input or is using the TT handler for input. In these cases, one CTRL/C is sufficient to terminate execution. Under the FB monitor, the job that is currently receiving input is the job that is stopped (determined by the most recently typed command, CTRL/F or CTRL/B). To ensure that the command is directed to the proper job, type CTRL/B or CTRL/F before typing CTRL/C.
CTRL/E	The CTRL/E command causes all terminal output to appear on both the display screen and the console terminal simultaneously. CTRL/E is valid after you type the monitor GT ON command and use the display. The command does not echo on the terminal. A second CTRL/E disables console terminal output. CTRL/E has no special meaning if GT ON is not in effect.
CTRL/F	CTRL/F causes the system to direct all keyboard input to the foreground job and take all output from the foreground job. The FB monitor echoes F> on the terminal unless output is already coming from the foreground job. If no foreground job exists, the monitor prints an error message and directs control to the background job. Otherwise, control remains with the foreground job until redirected to the background job (with CTRL/B) or until the foreground job terminates. CTRL/F has no special meaning when used under a single-job monitor, or when a SET TT NOFB command is in effect.
CTRL/O	CTRL/O causes RT-11 to suppress teleprinter output while continuing program execution. CTRL/O echoes^O on the terminal. RT-11 reenables teleprinter output when one of the following occurs: 1. You type second a CTRL/O. 2. You return control to the monitor by typing CTRL/C or by issuing the .EXIT request. 3. The running program issues a .RCTRLO programmed request (see Chapter 2 of the *RT-11 Advanced Programmer's Guide*). RT-11 system programs reset CTRL/O to the echoing state each time you enter a new command string.

(Continued on next page)

Table 3-4 (Cont.) Special Function Keys

Key	Function
CTRL/Q	CTRL/Q resumes printing characters on the terminal from the point printing previously stopped because of a CTRL/S. CTRL/Q does not echo and has no special meaning under the FB monitor if a SET TT NOPAGE command is in effect.
CTRL/S	CTRL/S temporarily suspends output to the terminal until you type a CTRL/Q. CTRL/S does not echo. Under the FB monitor, CTRL/S is not intercepted by the monitor if TT NOPAGE is in effect.
CTRL/U	CTRL/U deletes the current input line and echoes as ^U followed by a carriage return at the terminal. (The current line is defined as all characters back to, but not including, the most recent line feed, CTRL/C, or CTRL/Z.)
CTRL/Z	CTRL/Z terminates input when used with the terminal device handler (TT). It echoes ^Z on the terminal. The CTRL/Z itself does not appear in the input buffer. If TT is not being used, CTRL/Z has no special meaning.
DELETE or RUBOUT	DELETE deletes the last character from the current line and echoes a backslash plus the character deleted. Each succeeding DELETE deletes and echoes another character. The system prints an enclosing backslash when you type a key other than DELETE. This erasure is performed from right to left up to the beginning of the current line. If you are using a video display terminal, DELETE deletes characters with a backspace, space, backspace sequence. Your corrections appear on the screen; RUBOUT does not enclose them with backslash characters.

you type input to one job while the other has output control, the system suppresses the echo of the input until the job accepting input gains output control; at this point, all accumulated input echoes.

If the foreground job and background job are ready to print output at the same time, the foreground job has priority. The system prints output from the foreground job until it encounters a line feed. At that point, output from the background job prints until a line feed is encountered, and so forth.

When the foreground job terminates, control reverts automatically to the background job.

3.8 TYPE-AHEAD FEATURE

The monitor has a type-ahead feature that lets you enter terminal input while a program is executing. For example:

```
.DIRECTORY/PRINTER
DATE
```

While the first command line is executing, you can type the second line. The system stores this terminal input in a buffer and uses it when the system completes the first operation.

If you type a single CTRL/C while the system is in this mode, the system puts CTRL/C into the buffer. The program currently executing exits when you make a terminal input request. Typing a double CTRL/C returns control to the monitor immediately.

If type-ahead input exceeds the input butter capacity (usually 80 characters), the terminal bell rings and the system accepts no characters until a program uses part of the type-ahead buffer, or until you delete characters. No input is lost. Type-ahead is particularly useful when you specify multiple command lines to system programs. If you terminate a job by typing two CTRL/Cs, the system discards any unprocessed type-ahead.

If you use type-ahead with EDIT or BASIC, the system does not echo characters on the terminal but stores them in the buffer until the system processes a new command. The program echoes the characters only when it actually uses them.

CHAPTER 4
INTERACTIVE COMMANDS

Keyboard commands allow you to communicate with the RT-11 system. You enter keyboard commands at the terminal and the operating system immediately acknowledges and acts upon these requests.

4.1 COMMAND SYNTAX

This section describes the syntax conventions this manual uses to discuss the monitor command language. The Preface to this manual contains a more detailed list of the symbolic conventions used throughout the manual. You should familiarize yourself with the symbols and their meanings before you continue reading this chapter.

The system accepts commands in two ways: as a complete string containing all the information necessary to execute a command, or as a partial string. In the latter case, the system prompts you to supply the rest of the information. Terminate each command with a carriage return.

The general syntax for a command is:

```
COMMAND[/option...] input-filespec[/option...] output-filespec[/option...]
```

or

```
COMMAND[/option...]
PROMPT1? input-filespec[/option...]
PROMPT2? output-filespec[/option...]
```

where

COMMAND	is the command name.
/option	represents a command qualifier that specifies the exact action to be taken. Any option you supply here applies to the entire command string.
input-filespec	represents the file on which the action is to be taken.
/option	represents a file qualifier that specifies more detailed information about that particular file.
output-filespec	represents the file that is to receive the results of the operation.
/option	represents a file qualifier that specifies more detailed information about that particular file.

This manual provides a graphic illustration to clarify the syntax for each of the keyboard monitor commands. See Figure 4-1 for an illustration of a typical command. The illustrations provide a ready-reference list of the options that the commands accept, as well as information that makes the commands easier to use. The following list describes the conventions that are used in the illustrations.

1. Capital letters represent command names or options, which you must type as shown. (Abbreviations are discussed later in Section 4.1.)
2. Lower case letters represent arguments or variables for which you must supply values. For options that accept numeric arguments, the system interprets the values as decimal, unless otherwise stated. Some values, usually memory addresses, are interpreted as octal; these cases are noted in the accompanying text.
3. Square brackets [] enclose optional choices; you can include the item that is enclosed in the brackets or you can omit it, as you choose. If a vertical list of items is enclosed in square brackets, you can combine the options that appear in the list. However, if an option is set off from the others by blank lines (see /BOOT and /DEVICE in Figure 4-1), you cannot combine that option with any other option in the list.
4. Braces { } enclose options that are mutually exclusive. You can choose only one option from a group of options that appear in braces.
5. It is conventional to place command options (those qualifiers that apply to the entire command line) immediately after the command. However, it is also acceptable to specify a command option after a file specification. File options (those that qualify a particular file specification) must appear in the command line directly after the file to which they apply. The illustration for each command shows which options are file qualifiers, and whether they must follow input or output file specifications.
6. A line such as [NO] QUERY represents two mutually exclusive options: QUERY and NOQUERY.
7. Underlining indicates default options.

```
COPY[   /BOOT          ] (SP) input-filespecs [{/DOS[/OWNER:[nnn,nnn]]}] (SP) output-filespec [{/ALLOCATE:size       }]
    [                  ]                      [{/INTERCHANGE         }]                     [{/DOS                 }]
    [   /DEVICE        ]                      [{/POSITION:n          }]                     [{/INTERCHANGE[:size]  }]
    [                  ]                      [{/TOPS                }]                     [{/POSITION:n          }]
    [  {/ASCII  }      ]
    [  {/BINARY }      ]
    [  {/IMAGE  }      ]
    [   /CONCATENATE   ]
    [   /EXCLUDE       ]
    [   /IGNORE        ]
    [   /[NO] LOG      ]
    [   /NEWFILES      ]
    [   /PACKED        ]
    [   /PREDELETE     ]
    [   /[NO] QUERY    ]
    [   /[NO] REPLACE  ]
    [   /SETDATE       ]
    [   /SLOWLY        ]
    [   /SYSTEM        ]
```

Figure 4-1 Sample Command Syntax Illustration

A filespec represents a specific file and the device on which it is stored. Its syntax is:

dev:filnam.typ

where

dev:	represents either a logical device name or a physical device name, which is a two- or three-character name from Table 3-1.
filnam	represents the one- to six-character alphanumeric name of the file.
.typ	represents the one- to three-character alphanumeric file type, some of which are listed in Table 3-2.

There are several ways to indicate the device on which a file is stored. You can explicitly type the device name in the file specification:

```
    DX1:TEST.LST
```

You can omit the device name:

```
    TEST.LST
```

In this case, the system assumes that the file is stored on device DK:.

If you want to specify several files on the same device, you can use a technique called factoring:

```
    DT0:(TEST.LST,TESTA.LST,TESTB.LST)
```

The command shown above has the same meaning and is easier to use than the next command.

```
    DT0:TEST.LST,DT0:TESTA.LST,DT0:TESTB.LST
```

When you use factoring, as the example above shows, the device outside the parentheses applies to each file specification inside the parentheses. Without factoring, the system interprets each file specification to be DK:filespec unless you explicitly specify another device name.

NOTE

There is a restriction on the use of factoring in a command line. The command string that results from the expansion of the line you enter must not exceed 80 characters in length. If you use six-character file names and you also use factoring, specify only five files in a command line.

If you omit the file type in a file specification, the system assumes one of a number of defaults, depending on which command you issue. The MACRO command, for example, assumes a file type of .MAC for the input file specification, and the PRINT command assumes .LST. Some commands (such as COPY) do not assume a particular file type. If you need to specify a file with no file type in a command that assumes a default file type, type a period after the file name. For example, to run the file called TEST, type:

```
    RUN TEST.
```

If you omit the period after the file name, the system assumes a .SAV file type and tries to execute a file called TEST.SAV.

You can enter up to six input files and up to three output files for some commands. If the command string does not fit on one line of your terminal, use the hyphen (-), followed by a carriage return, as a continuation character and break the string into smaller sections. Use a carriage return to terminate the command string.

Some of the command and file qualifiers are mutually exclusive options. You should avoid using a combination of options that gives contradictory instructions to the system. For example:

```
    .DELETE/QUERY/NOQUERY TEST.LST
```

This command is not meaningful. Some mutually exclusive options are less obvious; these are noted, where necessary, in the list of options following each command and are enclosed by braces in the graphic representation of the command syntax.

The keyboard monitor commands are all English-language words. This feature makes the commands easier for you to understand and use. However, it can become tedious to type words like CROSSREFERENCE and ALLOCATE frequently. You can use as abbreviations the minimum number of characters that are needed to make the command or option unique. Table B-1 in Appendix B lists the minimum abbreviations for the commands and options.

An easy way to abbreviate a command or qualifier, and one that is always correct, is to use the first four characters or the first six characters if the qualifier starts with NO. For example:

CONCATENATE can be shortened to CONC
NOCONCATENATE can be shortened to NOCONC

The system prints an error message if you use an abbreviation that is not unique. For example, typing the following command produces an error, because C could mean COPY or COMPILE.

```
C TEST.LST
```

The prompting form of the command may be easier for you to learn if you are a new user. If you type a command followed by a carriage return, the system prompts you for an input file specification:

```
COPY/CONCATENATE
From?
```

You should enter the input file specification and a carriage return:

```
DX1:(TEST.LST,TESTA.LST)
```

The system prompts you for an output file specification:

```
To ?
```

You should enter the output file specification and a carriage return:

```
DX2:TEST.LST
```

The command now executes.

The system continues to prompt for an input and output file specification until you provide them. If you respond to a prompt by entering only a carriage return, the prompt prints again. You can combine the normal form of a command with the prompting form, as this example shows.

```
.COPY   ABC.LST
To   ?  DEF.LST
```

The system always prompts you for information if any required part of the command is missing. You can also enter just an option in response to a prompt. The two following examples are equivalent.

```
.COPY
From ?   *.MAC/NOLOG
To   ?   *.BAK

.COPY
From ?   /NOLOG
From ?   *.MAC
To   ?   *.BAK
```

4.2 WILDCARDS

Some commands accept wildcards (% and *) in place of the file name, file type, or characters in the file name or file type. The system ignores the contents of the wild field and selects all the files that match the remaining fields.

An asterisk (*) can replace a file name:

```
    *.MAC
```

The system selects all files on device DK: that have a .MAC file type, regardless of their name.

An asterisk (*) can replace a file type:

```
    TEST.*
```

The system selects all files on device DK: that are named TEST, regardless of their file type.

An asterisk (*) can replace both a file name and a file type:

```
    *.*
```

The system selects all files on device DK:.

An embedded asterisk (*) can replace any number of characters in the input file name or file type:

```
    A*B.MAC
```

The system selects all files on device DK: with a file type of .MAC whose file names start with A and end with B. For example, AB, AXB, AYYB, etc. would be selected.

The percent symbol (%) is always considered an embedded wildcard. It can replace a single character in the input file name or file type.

```
    A%B.MAC
```

The system selects all files on device DK: with a file type of .MAC whose file names are three characters long, start with A, and end with B. For example, AXB, AYB, AZB, etc. would be selected.

Table 4-1 lists commands that support wildcards.

Table 4-1 Commands Supporting Wildcards

Command	Accepts Wildcards in Input File Specification	Accepts Wildcards in Output File Specification
COPY	X	X
DELETE	X	
DIRECTORY	X	
HELP	X	
PRINT	X	
RENAME	X	X
TYPE	X	

For the commands that support wildcards the system has a special way of interpreting the file specifications you type. You can omit certain parts of the input and output specifications, and the system assumes an asterisk (*) for the omitted item. Table 4-2 shows the defaults that the system assumes for the input and output specifications of the valid commands.

Table 4-2 Wildcard Defaults

Command	Input Default	Output Default
COPY, RENAME	*.*	*.*
DIRECTORY	DK:*.*	
PRINT, TYPE	*.LST	
DELETE	filnam.*	

For example, if you need to copy all the files called MYPROG from DK: to DX1:, use this command:

```
.COPY/NOQUERY MYPROG  DX1:
```

The system interprets this command to mean:

```
.COPY/NOQUERY  DK:MYPROG.*  DX1:*.*
```

The system copies all the files called MYPROG, regardless of their file type, to device DX1: and gives them the same names.

If you need a directory listing of all the files on device DK:, type the following command:

```
.DIRECTORY
```

The system interprets this command to mean:

```
.DIRECTORY  DK:*.*
```

To list on the printer all the files on device DK: that have a .LST file type, use this command:

```
.PRINT  DK:
```

The system interprets this command to mean:

```
.PRINT  DK:*.LST
```

To delete all the files on device DK: called MYPROG, regardless of their file type, use this command:

```
.DELETE/NOQUERY  MYPROG
```

The system interprets this to mean:

```
.DELETE/NOQUERY  DK:MYPROG.*
```

You can use the SET WILDCARDS EXPLICIT command (described in Section 4.4) to change the way the system interprets these commands.

4.3 INDIRECT FILES

You can group together as a file a collection of keyboard commands that you want to execute sequentially. This collection is called an indirect command file, or indirect file. Indirect files are best suited for tasks that require a significant amount of computer time and that do not require your supervision or intervention. Any series of commands that you are likely to type often can also run easily as an indirect file. The indirect file concept is similar to BATCH processing. Although indirect files lack some of the capabilities of BATCH, they are easier to use, use the same commands as normal operations, and generally require less memory overhead than the BATCH processor. (RT-11 BATCH is described in Appendix A of this manual.) This section describes how to create indirect files and how to execute them.

4.3.1 Creating Indirect Files

Create an indirect file by using the EDIT/CREATE command described in Section 4.4. It is conventional to use a .COM file type for an indirect file, but you can choose any file name that you wish. Structure the lines of text to look like keyboard input, placing one command on each line of the file and terminating each line with a carriage return. Do not include the prompt character (.) in the line. Any keyboard monitor command you can type at the terminal you can also include in an indirect file. The following file, for example, prints the date and time, and creates backup copies of all FORTRAN source files:

```
DATE
TIME
COPY *.FOR *.BAK
```

Control returns to the monitor at the console terminal after this indirect file executes.

In addition to using the keyboard monitor commands, you can also run one of the RT-11 system utility programs in an indirect file. In this case, structure your input to conform to the Command String Interpreter syntax described in Chapter 6. The following file starts the directory system utility program and lists the directory of two devices on the line printer.

```
R DIR
LP:=CT0:/C:3
LP:=DT1:/C:3
^C
```

Note that the last command line is ^C. This is not the standard CTRL/C sequence you enter by holding down the CTRL key and typing a C. Rather, it is a readable CTRL/C that consists of two separate characters: a circumflex (uparrow) followed by a C. This sequence represents CTRL/C in indirect files because the two-character sequence is easier to read if you list the contents of the indirect file with the PRINT or TYPE command. This two-character sequence terminates the directory program so that control returns to the monitor when the indirect file finishes executing. Otherwise, the directory program would be left waiting for input from the console terminal when the indirect file finishes executing.

Remember to terminate the last command line with a carriage return, as you would any other line.

Some commands normally require a response from you as they execute. The INITIALIZE command, for example, prints the ARE YOU SURE? message and waits for you to type Y and a carriage return before it executes. The DELETE command requests confirmation from you before it deletes a file. There are three ways to control interaction with the executing command. One way is to use the /NOQUERY option on each command that allows it. This option suppresses the confirmation messages entirely when you use the command in an indirect file. A second procedure is suitable for a command like INITIALIZE, which has only one confirmation query. INITIALIZE can accept your response from within the indirect file. Place the Y response on a separate line in the indirect file, as the following example shows.

```
INITIALIZE/DOS DT1:
Y
```

A third method of interacting applies to a command like DELETE. This command can have a variable number of confirmation queries, especially if you use a wildcard in the file specification. This type of command accepts your responses directly from the terminal and allows you to make a decision before deleting each file. However, in this case the indirect file cannot operate unattended.

There is yet another way to deal with commands that require a response from you. Both the INITIALIZE and LINK commands have options that prompt you for data. This section describes two methods of responding to these prompts, when more than just a Y response is required.

The INITIALIZE command with the /VOLUMEID option permits you to specify a volume ID and owner name for a device. You can place your responses in the indirect file, as this example shows:

```
INITIALIZE/NOQUERY/VOLUMEID DT:
TAPE6
PAYROLL
```

You can change the indirect file so that the prompts appear on the console terminal and you can type your responses there:

```
INITIALIZE/NOQUERY/VOLUMEID DT:
^C
```

The ^C informs the system that the responses are to be entered at the terminal. Execution of the indirect file pauses until you enter the responses.

Similarly, the LINK command lets you specify some data either in the indirect file or from the console terminal. The following example contains the response to the TRANSFER prompt.

```
LINK/TRANSFER MYPROG,ODT
O.ODT
```

You can specify the same information interactively, as this example shows:

```
LINK/TRANSFER MYPROG,ODT
^C
```

The ^C informs the system that the response to the prompt is to be entered at the terminal. Execution of the indirect file pauses until you enter your response.

You can specify overlays to the LINK command by either of these two methods. The following indirect file links an overlaid program consisting of a root module and four overlay modules that reside in two overlay segments.

```
LINK/PROMPT ROOT
OVR1/O:1
OVR2/O:1
OVR3/O:2
OVR4/O:2//
```

Note in the above example that two slashes (//) terminate the module list. You can also enter all or part of the overlay information interactively, as this example shows:

```
LINK/PROMPT ROOT
OVR1/O:1
^C
```

The ^C informs the system that more overlay information is to be entered from the terminal. Execution of the indirect file pauses when the system requires the information. Respond to the asterisk prompt by entering the overlay information. Terminate the last overlay line with two slashes (//). Execution of the indirect file then proceeds. Chapter 11 describes the LINK program and explains how to use overlays.

If you need to link more than six modules, you can specify the extra modules on the next line in the indirect file, as this example shows:

```
LINK/PROMPT FIL1,FIL2,FIL3,FIL4,FIL5,FIL6
FIL7,FIL8//
```

Or, you can enter the extra modules from the terminal:

```
LINK/PROMPT FIL1,FIL2,FIL3,FIL4 FIL5,FIL6
^C
```

Execution of the indirect file pauses until you enter the remaining module names. Remember to follow the last name by two slashes (//).

You can include comments in an indirect file to help you document your work. These comments do not print on the console terminal when the indirect file executes. Begin a comment with an exclamation point (!). The system ignores any characters it finds between the exclamation point and the end of the current line. The following example shows an indirect file that contains comments.

```
!INDIRECT FILE TO ASSEMBLE THE MONITOR
R MACRO
RK1:RT11SJ=SJ,SYCND,KMON,USR,RMONSJ,KMOVLY
RK1:RKBTSJ=SJ,SYCND,BSTRAP    !ASSEMBLE THE BOOT
RK1:RK=SJ,SYSDEV,SYCND,RK     !AND RK DRIVER
RK1:SYSTBL=SJ,SYCND,SYSTBL    !AND SYSTBL
^C
!ALL DONE
```

4.3.2 Executing Indirect Files

You can execute indirect files under the SJ monitor, or in the background area under the FB or XM monitor.

To execute an indirect file, specify a command string according to the following syntax:

```
@filespec
```

where

@	is the monitor command that indicates an indirect file.
filespec	represents the name and file type of the indirect file, as well as the device on which it is stored. The default file type is .COM.

If you omit the device specification, DK: is assumed. If you specify any other block-replaceable device, the monitor automatically loads the handler for that device. It is conventional to type the indirect file command directly in response to the monitor's prompt, as this example shows:

```
.@INDCT
```

However, you can place the indirect command anywhere in a keyboard monitor command string, as long as it is the last element in the string, not including comments. For example:

```
DELETE/NOQUERY @INDCT!COMMENTS
```

This is a valid command string. The first line of the file should contain the list of files to be deleted. In the example above, assume the first line of the indirect file is:

```
*.BAK
```

This is the command that will actually execute:

```
DELETE/NOQUERY *.BAK
```

Check your indirect file carefully for errors before you execute it. When the monitor or any program that has control of the system encounters an illegal command line, or if an execution error of any kind occurs, that particular line does not execute properly. Execution of the indirect file does proceed, however, until any program that may be running relinquishes control to the monitor. Be careful of this if you run a system utility program in an indirect file, as this example shows:

```
R  PIP
DX1:*.*=DX0:*.*
DX0:*.MAC/D
^C
PRINT DX0:*.LST
```

If device DX1: becomes full before all the files from DX0: are copied to it, the second line of the indirect file does not execute completely. Execution then passes to the next line and the system deletes all MACRO files from DX0:. The ^C returns control to the monitor, which aborts the rest of the indirect file. This example shows that it is possible to destroy files accidentally because of the way indirect files execute. To be safe, use only keyboard monitor commands in an indirect file. This way the monitor gets control after each operation and can abort the indirect file as soon as it detects an error. A better way to perform the same operations as the indirect file shown above is as follows:

```
    COPY DX0:*.* DX1:*.*
    DELETE DX0:*.MAC
    PRINT DX0:*.LST
```

You can use the SET ERROR command, described in Section 4.4, to define the severity of error that causes an indirect file to stop executing.

Normally, as each line of an indirect file executes, it echoes on the console terminal so that you can observe the progress of the job. However, you can use the SET TT QUIET command, described in Section 4.4, to suppress this printout. In this case, only the prompting messages, if any, print. You can stop execution of an indirect file at any time by typing two CTRL/C characters. Control returns to the monitor and you can enter a new command. You can also abort the indirect file by typing a single CTRL/C in response to a query or prompt. If you use an indirect file to execute a MACRO program, read Section 2.4.15 of the *RT-11 Advanced Programmer's Guide* to learn about certain restrictions on using the .EXIT call with indirect files.

You can call another indirect file from within an indirect file. This procedure is called nesting. Restrict nesting to three levels of indirect files. The following example shows two-level nesting. Assume a programmer types this command at the console terminal in response to the monitor's prompt:

```
    @FIRST
```

The file FIRST.COM contains these lines:

```
    DATE
    TIME
    COPY *.MAC *.BAK
    @SECOND
    PRINT C
    DIRECTORY/PRINTER DK:
    DELETE/NOQUERY *.MAC
```

When this file executes it calls another indirect file, SECOND.COM, which contains this line:

```
    MACRO/CROSSREFERENCE A+B+C/LIST
```

When file SECOND.COM finishes executing, control returns to file FIRST.COM at the line following the indirect file specification. FIRST.COM then prints the contents of the file C.LST on the line printer, followed by a directory listing of device DK:. Then control returns to the monitor at the console terminal.

4.3.3 Startup Indirect Files

Section 3.1 introduced the startup indirect command files: STARTS.COM (for SJ), STARTF.COM (for FB), and STARTX.COM (for XM). Each monitor automatically invokes its own indirect command file when you bootstrap the system. You can modify these files to perform standard system configurations for you. Since many of the system parameters are reset by a bootstrap operation (see the SET command, Section 4.4), you should use the startup indirect files to set the system parameters you normally use. For example, if you use the FB monitor and have a visual display console terminal that supports hardware tabs, add the SET TT: SCOPE and SET TT: TAB commands to the file STARTF.COM. You could also include a SET TT: QUIET command at the beginning of STARTF.COM and a SET TT: NOQUIET command at the end to suppress extra type-out at bootstrap time. If you have a list of commands that you need to execute regardless of the monitor you bootstrap, include these commands in a separate indirect file, such as COMMON.COM, and invoke this file from all three startup indirect files. The following example shows a typical STARTF.COM file.

```
SET   TT:   QUIET                  !TURN OFF TTY PRINTING
SET   TT:   SCOPE
SET   TT:   TAB
@COMMON                            !PERFORM COMMON OPERATIONS
SET   TT:   NOQUIET                !TURN ON TTY PRINTING
```

If you use BATCH frequently, use a startup indirect file to assign devices and load handlers. You can also use the startup indirect files to run your own programs, set the date, or do other housekeeping chores.

4.4 KEYBOARD MONITOR COMMANDS

The keyboard monitor commands are your means of communicating with the system and controlling the monitor. This section lists the keyboard monitor commands in alphabetical order. Each command description includes the command syntax, a table of valid options, and some sample command lines, as well as a general discussion of how to use the command.

You can type almost all the commands to any of the three monitors. The exceptions are FRUN, SUSPEND, and RESUME. These are not legal for the SJ monitor because they apply to foreground programs.

Any reference to the background program applies as well to the program running under the SJ monitor. Any reference to FB operation also applies to the XM operation.

If you make a mistake in a command line, or if the system cannot perform the action you request, an error message prints on your terminal. The error message indicates which error occurred; see the *RT-11 System Message Manual* for a more complete description of the error and for the recommended action you should take. The error message also indicates which system utility program detected the error. This is for your information only and requires no action.

The APL command invokes the APL interpreter.

```
APL
```

APL has its own command language. Therefore, the APL command accepts no options and no file specifications.

The ASSIGN command associates the logical name you specify with a physical device.

```
ASSIGN (SP) physical-device-name (SP) logical-device-name
```

In the command syntax illustrated above, physical-device-name represents the RT-11 standard permanent name that refers to a particular device. Table 3-1 contains a list of these names. The term logical-device-name represents an alphanumeric name, from one to three characters long, that you assign to a particular device. Note that you should not use spaces or tabs in the logical device name. If you omit the physical device name, the system prompts you with Physical device name?. If you omit the logical device name, the system prompts you with Logical device name?.

The ASSIGN command can simplify programming. When you write a program, for example, you can request input from a device called INP: and direct output to a device called OUT:. When you are ready to execute the program, you can assign those logical names to the actual physical devices you need to use for that job. The ASSIGN command is especially helpful when a program refers to a device that is not available on a certain system; the ASSIGN command allows you to redirect input and output to an available device.

If the logical device name you supply is already associated with a physical device, the system disassociates the logical name from that physical device and assigns it to the current device. You can assign only one logical name with each ASSIGN command, but you can use several ASSIGN commands to assign different logical names to the same device. You can also use the ASSIGN command to assign FORTRAN logical units to physical devices.

If you are running under the foreground/background monitor (FB), FB is not allowed as a logical device name. However, it is valid under the single-job monitor. Note that the following names are always illegal logical device names: BA, FG, and EL.

The following command, for example, causes data that you write to device OUT: to print on the line printer.

```
.ASSIGN LP: OUT:
```

If your program attempts to access a device by using a logical name (such as OUT:) and you do not issue an appropriate ASSIGN command, an error occurs in the program.

The following command redirects printer output to the terminal.

```
.ASSIGN TT: LP:
```

The command shown above illustrates how you can run a program that specifically references LP: without using a line printer.

The next command redefines the default file device.

```
.ASSIGN RK1: DK:
```

If you supply a file specification and omit the device name, it now defaults to RK1:. Note that this does not affect the default system device, SY:.

The last example is typical for a system that uses a dual drive diskette device. Several users can share the same system software on DX0: and maintain their own data files on diskettes that they run in drive 1. When you use the following command, references to files without an explicit device name automatically access DX1:.

```
.ASSIGN DX1: DK:
```

Use the SHOW DEVICES command to display logical device name assignments on the terminal.

The B (Base) command sets a relocation base. To obtain the address of the location to be referenced, the system adds this relocation base to the address you specify in a subsequent Examine or Deposit command.

```
B[ (SP) address]
```

In the command syntax shown above, address represents an octal address that the system uses as a base address for subsequent Examine and Deposit commands. If the address you supply is an odd number, the system decreases it by one to make the address even. Note that if you do not specify an address, this command sets the base to zero.

Use the Base command when using the Examine and Deposit commands to reference linked modules. (Note that the Base command has no effect on program execution.) The system adds the current base address to the value you supply in an Examine or Deposit command. You can set the current base address to the address where a particular module is loaded. Then you can use the relocatable addresses printed in the assembler, compiler, or map listing of that module to reference locations within the module.

The following command sets the base to 0.

```
.B
```

The next two commands both set the base to 1000.

```
.B 1000
.B 1001
```

The BASIC command invokes the BASIC language interpreter.

```
BASIC
```

BASIC has its own command language. Therefore, the BASIC command accepts no options and no file specifications.

The BOOT command directs a new monitor to take control of the system. It can also read into memory a new copy of the monitor that is currently controlling the system.

```
BOOT (SP) filespec
```

In the command syntax illustrated above, filespec represents the device or monitor file to be bootstrapped. If you omit the filespec, the system prompts you with Device or file?. The BOOT command can perform either of two operations: 1) a hardware bootstrap of a specific device, or 2) a direct bootstrap of a particular monitor file that does not affect the bootstrap blocks on the device.

To perform a hardware bootstrap, specify only a device name in the command line. The following devices are legal for this operation: DT0:, RK0:-RK7:, RF:, SY:, DK:, DP0:-DP7:, DX0:-DX1:, DM0:-DM7:, and DS0:-DS7:. The hardware bootstrap operation gives control of the system to the particular monitor whose bootstrap is written on the device. (You can change this monitor by using the COPY/BOOT command.) This example bootstraps the single-job monitor, RKMNSJ, whose bootstrap information is written on device DK:.

```
.BOOT DK:

RT-11SJ  V03-01
```

To bootstrap a particular monitor file, specify that file name and the device on which it is stored, if necessary, in the command line. SY: is the default device and .SYS is the default file type. Note that the first two characters of the physical device name and the monitor file name must be the same, as in the following example.

```
.BOOT DX0:DXMNSJ

RT-11SJ  V03-01
```

You can use the BOOT command to alternate between the single-job and foreground/background monitors. When you use the BOOT command to change monitors you do not have to reenter the date and time. The system clock, however, can lose a few seconds during a reboot. The next example bootstraps the foreground/background monitor on device SY:, which is currently RK0:.

```
.BOOT RKMNFB

RT-11FB  V03-01
```

The system recognizes only the RT-11 standard monitor names. You cannot, therefore, bootstrap a monitor file that has been given a non-standard name.

The CLOSE command makes permanent all output files that are currently open in the background job.

```
CLOSE
```

The CLOSE command accepts no options or arguments.

You can use the CLOSE command to make tentative open files permanent; otherwise, they do not appear in a normal directory listing and the space associated with the files is available for reuse. The CLOSE command is particularly useful after you type a CTRL/C to abort a background job. You can also use it after an unexpected program termination. The CLOSE command preserves any new files that were being used by the terminated program. Note that the CLOSE command has no effect on a foreground job and that you cannot use CLOSE on files opened on magnetic tape or cassette.

The CLOSE command does not work if your program defines new input or output channels (with the .CDFN programmed request). Because CTRL/C or .EXIT resets channel definitions, the CLOSE command has no effect on channels it does not recognize.

The following example shows how the CLOSE command makes temporary files permanent.

```
.R PROG
     .
     .
     .
^C^C
.CLOSE
```

The COMPILE command invokes one or more language processors to assemble or compile the files you specify.

```
COMPILE[ /LIST[:filespec] [/ALLOCATE:size]            ] (SP) filespecs [/LIBRARY]
       [ /[NO] OBJECT[:filespec] [/ALLOCATE:size]     ]                [/PASS:1 ]
       [                                              ]
       [ { /DIBOL                                   } ]
       [ {  [/ALPHABETIZE        ]                  } ]
       [ {  [/CROSSREFERENCE     ]                  } ]
       [ {  [/[NO] LINENUMBERS   ]                  } ]
       [ {  [/ONDEBUG            ]                  } ]
       [ {  [/[NO] WARNINGS      ]                  } ]
       [ { /FORTRAN                                 } ]
       [ {  [/CODE:type          ]                  } ]
       [ {  [/DIAGNOSE           ]                  } ]
       [ {  [/EXTEND             ]                  } ]
       [ {  [/HEADER             ]                  } ]
       [ {  [/I4                 ]                  } ]
       [ {  [/[NO] LINENUMBERS   ]                  } ]
       [ {  [/ONDEBUG            ]                  } ]
       [ {  [/[NO] OPTIMIZE[:type]]                 } ]
       [ {  [/RECORD:length      ]                  } ]
       [ {  [/SHOW[:value]       ]                  } ]
       [ {  [/STATISTICS         ]                  } ]
       [ {  [/[NO] SWAP          ]                  } ]
       [ {  [/UNITS:n            ]                  } ]
       [ {  [/[NO] VECTORS       ]                  } ]
       [ {  [/WARNINGS           ]                  } ]
       [ { /MACRO                                   } ]
       [ {  [/CROSSREFERENCE[:type[...:type]]]      } ]
       [ {  [/DISABLE:value[...:value]         ]    } ]
       [ {  [/ENABLE:value[...:value]          ]    } ]
       [ {  [/[NO] SHOW:value                  ]    } ]
```

In the command line shown above, filespecs represents one or more files to be included in the compile or assembly. The default file types for the output files are .LST for listing files and .OBJ for object files. The defaults for input files depend on the particular language processor involved. These defaults include .MAC for MACRO files, .FOR for FORTRAN files, and .DBL for DIBOL files.

To compile (or assemble) multiple source files into a single object file, separate the files by plus (+) signs in the command line. Unless you specify otherwise, the system creates an object file with the same name as the first input file and gives it an .OBJ file type. To compile multiple files in independent compilations, separate the files by commas (,) in the command line. This generates a corresponding object file for each set of input files. You can combine up to six files for a compilation producing a single object file.

Language options are position dependent. That is, they have different meanings depending on where you place them in the command line. Options that qualify a command name apply across the entire command string. Options that follow a file specification apply only to the file (or group of files separated by plus signs) that they follow in the command string.

You can specify the entire COMPILE command as one line, or you can rely on the system to prompt you for information. The COMPILE command prompt is Files?.

There are several ways to establish which language processor the COMPILE command invokes. One way is to specify a language-name option, such as /MACRO, which invokes the MACRO assembler. Another way is to omit the language-name option and explicitly specify the file type for the source files. The COMPILE command then invokes the language processor that corresponds to that file type. Specifying the file SOURCE.MAC, for example, invokes the MACRO assembler. A third way to establish the language processor is to let the system choose a file type of .MAC, .DBL, or .FOR for the source file you name. To do this, the handler for the device you specify must be loaded. If you specify DX1:A and the DX handler is loaded, the system searches for source files A.MAC and A.DBL, in that order. If it finds one of these files, the system invokes the corresponding language processor. If it cannot find one of these files, or if the device handler associated with the input file is not resident, the system assumes a file type of .FOR and invokes the FORTRAN compiler.

If the language processor selected as a result of one of the procedures described above is not on the system device (SY:), the system issues an error message.

The following sections explain the options you can use with the COMPILE command.

/ALLOCATE:size – Use this option with /LIST or /OBJECT to reserve space on the device for the output file. The argument, size, represents the number of blocks of space to allocate. The meaningful range for this value is from 1 to 32767. A value of -1 is a special case that creates the largest file possible on the device.

/ALPHABETIZE – Use this option with DIBOL to alphabetize the entries in the symbol table listing. This is useful for program maintenance and debugging.

/CODE:type – Use this option with FORTRAN to produce object code that is designed for a particular hardware configuration. The argument, type, represents a three-letter abbreviation for the type of code to produce. The legal values are the following: EAE, EIS, FIS, and THR. See Section 1.1.1 of the *RT-11/RSTS/E FORTRAN IV User's Guide* for a complete description of the types of code and their functions.

/CROSSREFERENCE[:type[. . . :type]] – Use this option with MACRO or DIBOL to generate a symbol cross-reference section in the listing. This information is useful for program maintenance and debugging. Note that the system does not generate a listing by default. You must also specify /LIST in the command line to get a cross-reference listing.

With MACRO, this option takes an optional argument. The argument, type, represents a one-character code that indicates which sections of the cross-reference listing the assembler should include. Table 4-10 summarizes the valid arguments and their meaning.

/DIAGNOSE – Use the option with FORTRAN to help analyze an internal compiler error. /DIAGNOSE expands the crash dump information to include internal compiler tables and buffers. Submit the diagnostic printout to DIGITAL with an SPR form. The information in the listing can help the DIGITAL programmers locate the compiler error and correct it.

/DIBOL – This option invokes the DIBOL language processor to compile the associated files.

/DISABLE:value[. . . :value] – Use this option with MACRO to specify a .DSABL directive. Table 4-11 summarizes the arguments and their meaning. See Section 6.2 of the *PDP-11 MACRO Language Reference Manual* for a description of the directive and a list of all legal values.

/ENABLE:value[. . . :value] – Use this option with MACRO to specify an .ENABL directive. Table 4-11 summarizes the arguments and their meaning. See Section 6.2 of the *PDP-11 MACRO Language Reference Manual* for a description of the directive and a list of all legal values.

/EXTEND – Use this option with FORTRAN to change the right margin for source input lines from column 72 to column 80.

/FORTRAN – This option invokes the FORTRAN language processor to compile the associated files.

/HEADER – Use this option with FORTRAN to include in the printout a list of options that are currently in effect.

/I4 – Use this option with FORTRAN to allocate two words for the default integer data type (FORTRAN only uses one-word integers) so that it takes the same physical space as real variables.

/LIBRARY – Use this option with MACRO to identify the file the option qualifies as a macro library file; use it only after a macro library file specification in the command line. The MACRO assembler looks first to any macro libraries you specify before going to the default system macro library, SYSMAC.SML, to satisfy references (made with the .MCALL directive) from MACRO programs. In the example below, the two files A.FOR and B.FOR are compiled together, producing B.OBJ and B.LST. The MACRO assembler assembles C.MAC, satisfying .MCALL references from MYLIB.MAC and SYSMAC.SML. It produces C.OBJ and C.LST.

```
.COMPILE A+B/LIST/OBJECT,MYLIB/LIBRARY+C.MAC/LIST/OBJECT
```

/LINENUMBERS – Use this option with DIBOL or FORTRAN to include internal sequence numbers in the executable program. These are especially useful in debugging programs. This is the default operation.

/NOLINENUMBERS – Use this option with DIBOL or FORTRAN to suppress the generation of internal sequence numbers in the executable program. This produces a smaller program and optimizes execution speed. Use this option to compile only those programs that are already debugged; otherwise the DIBOL or FORTRAN error messages are difficult to interpret.

/LIST[:filespec] – You must specify this option to produce a compilation or assembly listing. The /LIST option has different meanings depending on where you put it in the command line.

If you specify /LIST without a file specification in the list of options that immediately follows the command name, the system generates a listing that prints on the line printer. If you follow /LIST with a device name, the system creates a listing file on that device. If the device is a file-structured device, the system stores the listing file on that device, assigning it the same name as the input file with a .LST file type. The following command produces a listing on the terminal.

```
.COMPILE/LIST:TT: A.FOR
```

The next command creates a listing file called A.LST on RK3:.

```
.COMPILE/LIST:RK3: A.MAC
```

If the /LIST option contains a name and file type to override the default of .LST, the system generates a listing file with that name. The following command, for example, compiles A.FOR and B.FOR together, producing files A.OBJ and FILE1.OUT on device DK:.

```
.COMPILE/FORTRAN/LIST:FILE1.OUT A+B
```

You cannot use a command line like the next one. In this example, the second listing file would replace the first one and, therefore, cause an error.

```
.COMPILE/LIST:FILE2 A.MAC,B.MAC
```

Another way to specify /LIST is to type it after the file specification to which it applies. To produce a listing file with the same name as a particular input file, you can use a command similar to this one:

```
.COMPILE/DIBOL A+B/LIST:RK3:
```

The command shown above compiles A.DBL and B.DBL together, producing files DK:A.OBJ and RK3:B.LST. If you specify a file name on a /LIST option following a file specification in the command line, it has the same meaning as when it follows the command. The following two commands have the same results.

```
.COMPILE/MACRO A/LIST:B

.COMPILE/MACRO/LIST:B A
```

Both the commands shown above generate as output files A.OBJ and B.LST.

Remember that file options apply only to the file (or group of files that are separated by plus signs) that they follow in the command string. For example:

```
.COMPILE A.MAC/LIST,B.FOR
```

This command compiles A.MAC, producing A.OBJ and A.LST. It also compiles B.FOR, producing B.OBJ. However, it does not produce any listing file for the compilation of B.FOR.

/MACRO – This option invokes the MACRO assembler to assemble the associated files.

/OBJECT[:filespec] – Use this option to specify a file name or device for the object file. Because the COMPILE command creates object files by default, the following two commands have the same meaning.

```
.COMPILE/FORTRAN A

.COMPILE/FORTRAN/OBJECT A
```

Both commands compile A.FOR and produce A.OBJ as output. The /OBJECT option functions like the /LIST option; it can be either a command or a file qualifier.

As a command option, /OBJECT applies across the entire command string. The following command, for example, assembles A.MAC and B.MAC separately, creating object files A.OBJ and B.OBJ on RK1:.

```
.COMPILE/OBJECT:RK1: A.MAC,B.MAC
```

Use /OBJECT as a file option to create an object file with a specific name or destination. The following command compiles A.DBL and B.DBL together, creating files B.LST and B.OBJ.

```
.COMPILE/DIBOL A+B/LIST/OBJECT
```

/NOOBJECT – Use this option to suppress creation of an object file. As a command option, /NOOBJECT suppresses all object files; as a file option, it suppresses only the object file produced by the related input files. In this command, for example, the system compiles A.FOR and B.FOR together, producing files A.OBJ and B.LST. It also compiles C.DBL and produces C.LST, but does not produce C.OBJ.

```
.COMPILE A.FOR+B.FOR/LIST,C.DBL/NOOBJECT/LIST
```

/ONDEBUG – Use this option with DIBOL to include a symbol table in the object file. You can then use a debugging program to find and correct errors in the object file.

Use /ONDEBUG with FORTRAN to include debug lines (those that have a D in column one) in the compilation. You do not, therefore, have to edit the file to include these lines in the compilation or to logically remove them. This option is useful in debugging a program. You can include messages, flags, and conditional branches to help you trace program execution and find an error.

/OPTIMIZE[:type] – Use this option with FORTRAN to enable certain options that optimize object code for various conditions. The argument, type, represents the three-letter code for the type of optimization to enable. Table 4-4 summarizes the codes and their meanings.

/NOOPTIMIZE[:type] – Use this option with FORTRAN to disable certain options that optimize object code for various conditions. The argument, type, represents the three-letter code for the type of optimization to disable. Table 4-4 summarizes the codes and their meanings.

/PASS:1 – Use this option with MACRO on a prefix macro file to process that file during pass-1 of the assembly only. This option is useful when you assemble a source program together with a prefix file that contains only macro definitions, since these definitions do not need to be redefined in pass-2 of the assembly. The following command assembles a prefix file and a source file together, producing files PROG1.OBJ and PROG1.LST.

```
    .COMPILE/MACRO PREFIX/PASS:1+PROG1/LIST/OBJECT
```

/RECORD:length – Use this option with FORTRAN to override the default record length of 132 characters for ASCII sequential formatted input and output. The meaningful range for the argument, length, is from 4 to 4095.

/SHOW:value – Use this option with FORTRAN to control FORTRAN listing format. The argument, value, represents a code that indicates which listings the compiler is to produce. Table 4-5 summarizes the codes and their meanings.

Use this option with MACRO to specify any MACRO .LIST directive. Table 4-12 summarizes the valid arguments and their meanings. Section 6.1.1 of the *PDP-11 MACRO Language Reference Manual* explains how to use these directives.

/NOSHOW:value – Use this option with MACRO to specify any MACRO .NLIST directive. Table 4-12 summarizes the valid arguments and their meanings. Section 6.1.1 of the *PDP-11 MACRO Language Reference Manual* explains how to use these directives.

/STATISTICS – Use this option with FORTRAN to include in the listing compilation statistics, such as amount of memory used, amount of time elapsed, and length of the symbol table.

/SWAP – Use this option with FORTRAN to permit the USR (user service routine) to swap over the FORTRAN program in memory. This is the default operation.

/NOSWAP – Use this option with FORTRAN to keep the USR resident during execution of a FORTRAN program. This may be necessary if the FORTRAN program uses some of the RT-11 System Subroutine Library calls (see Chapter 4 of the *RT-11 Advanced Programmer's Guide*). If the program frequently updates or creates a large number of different files, making the USR resident can improve program execution. However, the penalty for making the USR resident is 2K words of memory.

/UNITS:n – Use this option with FORTRAN to override the default number of logical units (6) to be open at one time. The maximum value you can specify for n is 16.

/VECTORS – This option directs FORTRAN to use tables to access multidimensional arrays. This is the default mode of operation.

/NOVECTORS – This option directs FORTRAN to use multiplication operations to access multidimensional arrays.

/WARNINGS – Use this option to include warning messages in DIBOL or FORTRAN compiler diagnostic error messages. These messages call certain conditions to your attention, but do not interfere with the compilation. This is the default operation for DIBOL.

/NOWARNINGS – Use this option with DIBOL to suppress warning messages during compilation. These messages are for your information only; they do not affect the compilation. This is the default operation for FORTRAN.

The COPY command performs a variety of file transfer and maintenance operations.

```
COPY[ /BOOT          ] (SP) input-filespecs[ {/DOS[/OWNER:[nnn,nnn]]} ] (SP) output-filespec [ {/ALLOCATE:size        } ]
    [                ]                    [ {/INTERCHANGE         } ]                   [ {/DOS                  } ]
    [ /DEVICE        ]                    [ {/POSITION:n          } ]                   [ {/INTERCHANGE[:size]   } ]
    [                ]                    [ {/TOPS                } ]                   [ {/POSITION:n           } ]
    [ {/ASCII }      ]
    [ {/BINARY}      ]
    [ {/IMAGE }      ]
    [ /CONCATENATE   ]
    [ /EXCLUDE       ]
    [ /IGNORE        ]
    [ /[NO] LOG      ]
    [ /NEWFILES      ]
    [ /PACKED        ]
    [ /PREDELETE     ]
    [ /[NO] QUERY    ]
    [ /[NO] REPLACE  ]
    [ /SETDATE       ]
    [ /SLOWLY        ]
    [ /SYSTEM        ]
```

The COPY command transfers:

- One file to another file
- A number of files to a single file by concatenation
- One device to another device
- A bootstrap to a device.

In the command syntax shown above, input-filespecs represents the data to copy. The input-filespec can be a device name, if you use the /DEVICE option. Otherwise, you can specify as many as six files for input. Output-filespec represents the device or file to receive the data. You can specify only one output device or file.

Normally, commas separate the input files if you specify more than one. However, you can separate them by plus (+) signs if you want to combine them. In this case, you can also omit the /CONCATENATE option, as the following example shows.

```
.COPY A.FOR+B.FOR C.FOR
```

This command combines DK:A.FOR with DK:B.FOR and stores the results in DK:C.FOR.

You can use wildcards in the input or output file specification of the command. However, the output file specification cannot contain embedded wildcards. Note that for all operations except CONCATENATE, if you use a wildcard in the input file specification, the corresponding output file name or file type must be a *. This example uses wildcards correctly:

```
.COPY A%B.MAC *.BAK
```

In the CONCATENATE operation, the output specification must represent a single file. Therefore, no wildcards are allowed.

You can enter the COPY command as one line, or you can rely on the system to prompt you for information. The COPY command prompts are: From? for the input file specification and To? for the output file specification.

The system has a special way of handling system (.SYS) files and files that cover bad blocks (.BAD files). So that you do not copy system files by accident when you use a wildcard in the file specification, the system requires you to use the /SYSTEM option when you need to copy system files. To copy a .BAD file, you must specify it by explicitly giving its file name and file type. Since .BAD files cover bad blocks on a device, you usually do not need to copy, delete, or otherwise manipulate these files.

The following sections describe the COPY command options and include command examples.

/ALLOCATE:size – Use this option after the output file specification to reserve space on the device for the output file. The argument, size, represents the number of blocks of space to allocate. The meaningful range for this value is from 1 to 32767. A value of -1 is a special case that creates the largest file possible on the device.

/ASCII – This option copies files in ASCII mode, ignoring nulls and rubout characters. It converts data to the ASCII 7-bit format, and treats CTRL/Z (32 octal) as the logical end-of-file on input. Files that consist of ASCII-format data include source files you create with the editor, map files, and list files. The following example copies a FORTRAN source program from DX0: to DX1:, giving it a new name, and reserves 50 blocks of space for it.

```
.COPY/ASCII DX0:MATRIX.FOR DX1:TEST.FOR/ALLOCATE:50
```

/BINARY – Use this option to copy formatted binary files. These include .OBJ files produced by the assembler or the FORTRAN compiler, and .LDA files produced by the linker. The system verifies checksums and prints a warning if a checksum error occurs. If this happens, the copy operation does not complete. Note that you cannot copy library files with the /BINARY option because of a checksum error. Copy them in image mode. The following command copies a binary file from DK: to a diskette.

```
.COPY/BINARY ANALYZ.OBJ DX1:*.*
```

/BOOT – This option copies bootstrap information from a monitor file to blocks 0 and 2 through 5 of a random access device. This permits you to use that device as a system device. Note that you cannot combine /BOOT with any other option. Before you use the /BOOT option, make sure that the appropriate monitor file is already stored on the disk. To create a bootable system diskette, for example, you could use the foreground/background file called DXMNFB.SYS. If you copy the monitor file onto the diskette from another device, be careful not to rename it. The COPY/BOOT operation recognizes only standard RT-11 monitor file names. You can use a procedure similar to the following to create a system device:

1. Initialize the disk. Use the monitor INITIALIZE command to do this.
2. Copy files onto the disk. Use the COPY/SYSTEM command for this step.
3. Use COPY/BOOT to write the monitor bootstrap onto the disk.

The following example shows how to create a system diskette.

```
.INITIALIZE DX1:

DX1:/Init are you sure?Y

.COPY/SYSTEM DX0:*.* DX1:*.*
 Files copied:
DX0:DXMNSJ.SYS to DX1:DXMNSJ.SYS
DX0:DT.SYS     to DX1:DT.SYS
DX0:DX.SYS     to DX1:DX.SYS
DX0:TT.SYS     to DX1:TT.SYS
DX0:LP.SYS     to DX1:LP.SYS
DX0:DIR.SAV    to DX1:DIR.SAV
DX0:DUP.SAV    to DX1:DUP.SAV
```

```
DX0:ABC.MAC     to DX1:ABC.MAC
DX0:AAF.MAC     to DX1:AAF.MAC
DX0:CT.SYS      to DX1:CT.SYS
DX0:PIP.SAV     to DX1:PIP.SAV
DX0:MT.SYS      to DX1:MT.SYS
DX0:MM.SYS      to DX1:MM.SYS
DX0:COMB.       to DX1:COMB.
DX0:DXMNFB.SYS to DX1:DXMNFB.SYS

.COPY/BOOT DX1:DXMNFB.SYS DX1:
```

/CONCATENATE – Use this option to combine several input files into a single output file. Remember that wildcards are illegal in the output file specification. This option is particularly useful to combine several object modules into a single file for use by the linker or librarian. The following command combines all the .FOR files on DX1: into a file called MERGE.FOR on DX0:.

```
.COPY/CONCATENATE DX1:*.FOR DX0:MERGE.FOR
 Files copied:
DX1:A.FOR       to DX0:MERGE.FOR
DX1:B.FOR       to DX0:MERGE.FOR
DX1:C.FOR       to DX0:MERGE.FOR
```

/DEVICE – This option copies block for block the image of one device to another. You cannot combine any other option with /DEVICE. This option copies one disk to another without changing the file structure or the location of the files on the device. This is convenient in that the bootstrap blocks also remain unchanged. You can also copy disks that are not in RT-11 format, as long as they have no bad blocks. If the system encounters a bad block during the COPY/DEVICE operation, it prints an error message. However, it then retries the operation and performs the copy one block at a time. If only one error message prints, you can assume that the transfer completed correctly.

If one device is smaller than the other, the system copies only as many blocks as the smaller device contains. It is possible to copy blocks between disk and magtape, even though magtape is not a random access device. The data is stored on tape formatted in 1K word blocks. There is room for only one disk image on a magtape. The following command copies an image of DX0: to DX1:.

```
.COPY/DEVICE DX0: DX1:

DX1:/Copy are you sure?Y
```

Respond to the query message by typing Y and a carriage return. Any other response cancels the command and the COPY operation does not proceed.

/DOS – Use this option to transfer files between RSTS/E or DOS-11 format and RT-11 format. The option must appear in the command line after the file to which it applies. Valid input devices are DECtape and RK05; the only valid output device is DECtape. The only other options allowed with /DOS are /ASCII, /BINARY, /IMAGE, and /OWNER:[nnn,nnn]. The following command transfers a BASIC source file from a DOS-11 disk to an RT-11 disk.

```
.COPY RK:PROG.BAS/DOS/OWNER:[200,200] SY:*.*
```

The next command copies a memory image file from an RT-11 disk to a RSTS/E format DECtape.

```
.COPY DUMP.SAV DT:*.*/DOS
```

/EXCLUDE – This option copies all the files on a device except the ones you specify. The following command copies all files from DX0: to DX1: except .OBJ and .SAV files.

```
.COPY/EXCLUDE DX0:(*.OBJ,*.SAV) DX1:*.*
```

/IGNORE – Use this option to ignore input errors during a copy operation. /IGNORE forces a single-block data transfer, which you can invoke at any other time with the /SLOWLY option. Use /IGNORE if an input error occurred when you tried to perform a normal copy operation. This procedure can sometimes recover a file that is otherwise unreadable. If there is still an error, an error message prints on the terminal, but the copy operation continues. This option is illegal with /DOS, /TOPS, and /INTERCHANGE.

/IMAGE – If you enter a command line without an option, or if you use the /IMAGE option, the copy operation proceeds in image mode. Use this method to transfer memory image files and any files other than ASCII or formatted binary. Note that you cannot reliably transfer memory image files to or from paper tape, or to the line printer or console terminal. You can image-copy ASCII and binary data with the following restrictions:

1. For ASCII data, there is no check for nulls.
2. For binary data, there is no checksum consideration.

This command copies a text file to a DECtape for storage:

```
.COPY LETTER.TXT DT0:*.*
```

/INTERCHANGE[:size] – This option transfers data in interchange (proposed ANSI standard) format between RT-11 block-replaceable devices and interchange diskettes that are compatible with IBM 3741 format. The option must appear in the command line after the file to which it applies. If the output file is to be in interchange format, you can specify the length of each record. The argument, size, represents the record length in characters. The following command transfers the RT-11 file WAIT.MAC from device DK: to device DX1: in interchange format, giving it the name WAIT.MA. The record length is set to 128 (decimal) bytes.

```
.COPY WAIT.MAC DX1:*.*/INTERCHANGE:128.
```

/LOG – This option lists on the terminal the names of the files that were copied by the current command. Normally, the system prints a log only if there is a wildcard in the file specification. If you specify /QUERY, the system prints the name of each file and asks you for confirmation before the operation proceeds. In this case, the query messages replace the log, unless you specifically type /LOG/QUERY in the command line. The following example shows a copy command line and the resulting log.

```
.COPY DX1:*.SAV DX0:*.SAV
 Files copied:
DX1:DIR.SAV     to DX0:DIR.SAV
DX1:DUP.SAV     to DX0:DUP.SAV
DX1:PIP.SAV     to DX0:PIP.SAV
```

/NOLOG – This option prevents a list of the files copied from printing on the terminal.

/NEWFILES – Use this option in the command line if you want to copy only those files that have the current date. The following example shows a convenient way to back up all new files after a session at the computer.

```
.COPY/NEWFILES *.* DX1:*.*
 Files copied:
DK:A.FOR        to DX1:A.FOR
DK:B.FOR        to DX1:B.FOR
DK:C.FOR        to DX1:C.FOR
```

/OWNER:[nnn,nnn] – Use this option with /DOS to represent a DOS-11 user identification code (UIC) for a DOS-11 input device. Note that the square brackets are part of the UIC; you must type them. The initial default for the UIC is [1,1]. If you supply a UIC, it becomes the default for all future transfers.

/PACKED – This option copies files in PDP-10, DOS, or interchange mode. You can use /PACKED on an input file specification with the /TOPS, /DOS, or /INTERCHANGE option to transfer files to RT-11 format.

/POSITION:n – Use this option when you copy files to or from magtape or cassette. The /POSITION:n option lets you direct the tape operation; you can move the tape and perform an operation at the point you specify. For all operations, omitting the argument, n, has the same effect as setting n equal to 0 (n is interpreted as a decimal number).

For magtape read (copy from tape) operations, the /POSITION:n option initiates these procedures:

1. If n is 0:
   The tape rewinds and the system searches for the file you specify. If you specify more than one file, the tape rewinds before each search. If the file specification contains a wildcard, the tape rewinds only once and then the system copies all the appropriate files.
2. If n is a positive integer:
   The system looks for the file at file sequence number n. If the file it finds there is the one you specify, the system copies it. Otherwise, the system prints an error message. If you use a wildcard in the file specification, the system goes to file sequence number n and then begins to look for the appropriate files.
3. If n is -1:
   The system starts its search at the current position. Note that if the current position is not the beginning of the tape, it is possible that the file you specify will not be found, even though it does exist on the tape.

For magtape write (copy to tape) operations, the /POSITION:n option has this effect:

1. If n is 0:
   The tape rewinds before the system copies each file. A warning message prints on the terminal if the system finds another file on the tape with the same name and file type.
2. If n is a positive integer:
   The system goes to file sequence number n or to the logical end of tape, whichever comes first. Then it enters the file you specify. If you specify more than one file, or if you use a wildcard in the file specification, the tape does not rewind before the system writes each file, and the system does not check for duplicate file names.
3. If n is -1:
   The system goes to the logical end of tape and enters the file you specify. It does not check for duplicate file names.
4. If n is -2:
   The tape rewinds between each copy operation. The system enters the file you specify at logical end-of-tape or at the first occurrence of a duplicate file name.

The system also has special procedures for handling cassettes. For cassette read (copy from tape) operations, the /POSITION:n option initiates these procedures:

1. If n is 0:
   The cassette rewinds and the system searches for the file you specify. If you specify more than one file, or if you use a wildcard in the file specification, the cassette rewinds before each search.
2. If n is a positive integer:
   The system starts from the cassette's present position and searches for the file you specify. If the system does not find the file you specify before it reaches the nth file from its starting position, it reads the nth file. Note that if the starting position is not the beginning of the tape, it is possible that the system will not find the file you specify, even though it does exist on the tape.

3. If n is a negative integer:
   The cassette rewinds, then the system follows the procedure outlined in step 2 above.

For cassette write (copy to tape) operations, the /POSITION:n option has this effect:

1. If n is 0:
   The cassette rewinds and the system writes the file you specify at the logical end-of-tape. The system automatically deletes any file it finds along the way that has the same name and file type as the file you specify.
2. If n is a positive integer:
   The system starts from the cassette's present position and searches n files ahead, deleting along the way any file it finds that has the same name and file type as the file you specify. If the system does not reach the logical end-of-tape before it reaches the nth file from its starting position, it enters the file you specify over the nth file and deletes any files beyond it on the tape. If the system reaches the logical end-of-tape before it reaches the nth file, it writes the file you specify at the end-of-tape position.
3. If n is a negative integer:
   The cassette rewinds, then the system follows the same procedure outlined in step 2 above.

Section 7.2.1 contains more detailed information about operations involving magtape and cassette.

/PREDELETE – This option deletes a file on the output device if you copy a file with the same name to that device. The system deletes the file on the output device before the copy occurs. Normally, the system deletes a file of the same name after the copy operation successfully completes. This option is useful for operations involving devices that have limited space, such as diskette. Be careful when you use the /PREDELETE option; if for any reason the input file is unreadable, the output file will already have been deleted and you can be left with no useable version of the file.

/QUERY – If you use this option, the system requests confirmation from you before it performs the operation. /QUERY is particularly useful on operations that involve wildcards, when you may not be completely sure which files the system selected for an operation. The /QUERY option is valid on the COPY command only if both input and output are in RT-11 format. Note that if you specify /QUERY in a copy command line that also contains a wildcard in the file specification, the confirmation messages that print on the terminal replace the log messages that would normally appear. You must respond to a query message by typing Y (or anything that begins with a Y) and a carriage return to initiate execution of a particular operation. The system interprets any other response as NO and it does not perform the specific operation. The following example copies three of the four FOR files stored on DK: to DX1:.

```
.COPY/QUERY DK:*.FOR DX1:*.*
 Files copied:
DK:A.FOR        to DX1:A.FOR       ? Y
DK:B.FOR        to DX1:B.FOR       ? Y
DK:C.FOR        to DX1:C.FOR       ? NO
DK:DEMOF1.FOR   to DX1:DEMOF1.FOR? Y
```

/NOQUERY – This option suppresses the confirmation message that the system prints for some operations, such as COPY/DEVICE.

/REPLACE – This is the default mode of operation for the COPY command. If a file exists on the output device with the same name as the file you specify for output, the system deletes that duplicate file after the copy operation successfully completes.

/NOREPLACE – This option prevents execution of the copy operation if a file with the same name as the output file you specify already exists on the output device. /NOREPLACE is valid only if both the input and output are in RT-11 format.

/SETDATE – This option causes the system to put the current date on all files it transfers, unless the current system date is zero. Normally, the system preserves the existing file creation date when it copies a file block for block. This option is invalid for operations involving magtape and cassette because the system always uses the current date for tape files.

/SLOWLY – This option transfers files one block at a time. On some devices, a single-block transfer increases the chances of an error-free transfer. Use this option if a previous copy operation failed because of a read or write error.

/SYSTEM – Use this option if you need to copy system (.SYS) files. If you omit this option, the .SYS files are excluded from all operations and a message is printed on the terminal to remind you.

/TOPS – This option transfers files on DECsystem-10 DECtape to RT-11 format. The option must follow the input file specification. Note that DECtape is the only valid input device. You cannot perform this copy operation while a foreground job is running. Use /PACKED with /TOPS to convert from TOPS-10 7-bit ASCII format to standard PDP-11 byte ASCII format. The following command copies in ASCII format all the files named MODULE from the DECsystem-10 DECtape DT0: to RT-11 device RK0:.

```
.COPY/ASCII DT0:MODULE.*/TOPS RK0:*.*
```

The D (Deposit) command deposits values in memory beginning at the location you specify.

```
D (SP) address=value[,... value]
```

In the command syntax illustrated above, address represents an octal address that, when added to the relocation base value from the Base command (if you used one), provides the actual address where the system must deposit the values. The argument, value, represents the new contents of the address. If you do not specify a value, the system assumes a value of 0. If you specify more than one value and separate the values by commas, the system deposits the values in sequential locations beginning at the location you specify.

The Deposit command accepts both word and byte addresses, but it always executes the command as though you specified a word address. (If you specify an odd address, the system decreases it by one to make it even.) The Deposit command stores all values as word quantities.

Use commas to separate multiple values in the command line. Two or more adiacent commas cause the system to deposit 0s at the location you specify and at the following locations, if indicated.

Note that you cannot specify an address that references a location outside the area of the background job. You can use the D command with GET and START to temporarily alter a program's execution. Use the SAVE command before START to make the alteration permanent.

The following command deposits 0s into locations 300, 302, 304, and 306.

```
.D 300=,,,
```

The next command sets the base address to 0.

```
.B
```

The following command deposits 3705 into location 1000.

```
.D 1000=3705
```

The next command sets the relocation base to 1000.

```
.B 1000
```

The last command puts 2503 into location 1500 and 22 into location 1502.

```
.D 500=2503,22
```

Use the DATE command to set or to inspect the current system date.

```
DATE[ (SP) dd-mmm-yy]
```

In the command syntax shown above, dd represents the day (a decimal number from 1 to 31), mmm represents the first three characters of the name of the month, and yy represents the year (a decimal number from 73 to 99).

To enter a date into the system, specify the date in the format described above. You should do this as soon as you bootstrap the system. The system uses this date for newly created files, for files that you transfer to magtape or cassette, and for listing files. The following example enters the current date.

```
.DATE 18-MAY-77
```

The system automatically changes the date each day at midnight. However, it does not change the date correctly at the end of each month. Do this by issuing the DATE command.

To display the current system date, type the DATE command without an argument, as this example shows.

```
.DATE
18-May-77
```

The DEASSIGN command disassociates a logical device name from a physical device name.

```
DEASSIGN[ (SP) logical-device-name]
```

In the command syntax illustrated above, logical-device-name represents an alphanumeric name, from one to three characters long, that is assigned to a particular device. Note that spaces and tabs are not permitted in the logical device name.

To remove the assignment of a particular logical device name to a physical device, specify that logical device name in the command line. The following example disassociates the logical name INP: from the physical device to which it is assigned.

```
.DEASSIGN INP:
```

If you specify a logical name that is not currently assigned, the system prints an error message, as this example shows.

```
.DEASSIGN INP:
?KMON-F-Logical name not found
```

To disassociate all logical names from physical devices, type the DEASSIGN command without an argument. The following example disassociates all logical device names (except DK: and SY:) from physical devices.

```
.DEASSIGN
```

If DK: is assigned to a device (such as DX1:, for example), the following command disassociates DK: from DX1: and restores the default association of DK: to SY:, the system device.

```
.DEASSIGN DK:
```

The DELETE command deletes the files you specify.

```
DELETE ┌/DOS            ┐ (SP) filespecs
       │                │
       │/INTERCHANGE    │
       │                │
       │/EXCLUDE        │
       │/LOG            │
       │/NEWFILES       │
       │/POSITION:n     │
       │/[NO]QUERY      │
       └/SYSTEM         ┘
```

In the command syntax shown above, filespecs represents the files to be deleted. You can specify up to six files; separate them by commas. You can enter the DELETE command as one line, or you can rely on the system to prompt you for information. If you omit the file specification, the DELETE command prompts you with Files?. If you delete a file accidentally, it is possible to recover the file if you act immediately. A procedure for doing this is described in Chapter 8.

The system has a special way of handling system (.SYS) files and files that cover bad blocks (.BAD files). So that you do not delete system files by accident when you use a wildcard in the file specification, the system requires you to use the /SYSTEM option when you need to delete system files. To delete a .BAD file, you must specify it by explicitly giving its file name and file type. Since .BAD files cover bad blocks on a device, you usually do not need to copy, delete, or otherwise manipulate these files.

Another feature of the DELETE command is that the system always requests confirmation from you before it actually deletes a file. You must respond to the query message by typing Y followed by a carriage return in order to execute the command.

The following sections describe the options you can use with the DELETE command.

/DOS – Use this option to delete a file that is in DOS-11 or RSTS/E format. Remember that the valid devices for this type of file are disks and DECtape. You cannot combine any other option with /DOS.

/EXCLUDE – This option deletes all the files on a device except the ones you specify. The following command, for example, deletes all files from DX1: except .SAV files. Remember to use /SYSTEM if you need to include .SYS files in the operation.

```
.DELETE/EXCLUDE DX0:*.SAV
?PIP-W-No .SYS action
 Files deleted:
DX0:ABC.OLD   ? Y
DX0:AAF.OLD   ? Y
DX0:COMB.     ? Y
DX0:MERGE.OLD ? Y
```

/INTERCHANGE – Use this option to delete from a diskette a file that is in interchange (proposed ANSI standard) format. You cannot combine any other option with /INTERCHANGE.

/LOG – This option lists on the terminal a log of the files that are deleted by the current command. Note that if you specify /LOG, the system does not ask you for confirmation before execution proceeds. Use both /LOG and /QUERY to invoke logging and querying.

/NEWFILES – Use this option to delete only the files that have the current system date. This is a convenient way to remove all the new files that you just created in a session at the computer. The following example deletes the backup files created today.

```
.DELETE/NEWFILES DX1:*.BAK
 Files deleted:
DX1:MERGE.BAK ? Y
```

/POSITION:n – You can use this option when you delete files from cassette. It permits you to direct the tape operation; you can move the tape and perform an operation at the point you specify. Omitting the argument, n, has the same effect as setting n equal to 0 (n is interpreted as a decimal number). The /POSITION:n option has the following effect:

1. If n is 0:
   The cassette rewinds and the system searches for the file you specify. If you specify more than one file, or if you use a wildcard in the file specification, the cassette rewinds before each search.
2. If n is a positive integer:
   The system starts from the cassette's present position and searches for the file you specify. If the system does not find the file you specify before it reaches the nth file from its starting position, it deletes the nth file. Note that if the starting position is not the beginning of the tape, it is possible that the system will not find the file you specify, even though it does exist on the tape.
3. If n is a negative integer:
   The cassette rewinds, then the system follows the procedure outlined in step 2 above.

/QUERY – Use this option to request a confirmation message from the system before it deletes each file. This option is particularly useful on operations that involve wildcards, when you may not be completely sure which files the system selected for the operation. This is the default mode of operation. Note that specifying /LOG eliminates the automatic query; you must specify /QUERY with /LOG to retain the query function. You must respond to a query message by typing Y (or anything that begins with a Y) and a carriage return to initiate execution of a particular operation. The system interprets any other response as NO and it does not perform the operation. The following example shows querying. Only one file is deleted.

```
.DELETE DX1:*.*
 Files deleted:
DX1:ABC.MAC    ? N
DX1:AAF.MAC    ? Y
DX1:MERGE.FOR  ? N
```

/NOQUERY – This option suppresses the confirmation message that the system prints before it deletes each file.

/SYSTEM – Use this option if you need to delete system (.SYS) files. If you omit this option, the system files are excluded from the delete operation, and a message is printed on the terminal to remind you.

The DIBOL command invokes the DIBOL compiler to compile one or more source programs.

```
DIBOL [/LIST[:filespec] [/ALLOCATE:size]          ] (SP) filespecs
      [/[NO] OBJECT[:filespec] [/ALLOCATE:size]   ]
      [                                           ]
      [/ALPHABETIZE                               ]
      [/CROSSREFERENCE                            ]
      [/[NO] LINENUMBERS                          ]
      [/ONDEBUG                                   ]
      [/[NO] WARNINGS                             ]
```

In the command syntax illustrated above, filespecs represents one or more files to be included in the compilation. If you omit a file type for an input file, the system assumes .DBL. Output default file types are .LST for listing files and .OBJ for object files. To compile multiple source files into a single object file, separate the files by plus (+) signs in the command line. Unless you specify otherwise, the system creates an object file with the same name as the first input file and gives it an .OBJ file type. To compile multiple files in independent compilations, separate the files by commas (,) in the command line. This generates a corresponding object file for each set of input files.

Language options are position dependent. That is, they have different meanings depending on where you place them in the command line. Options that qualify a command name apply across the entire command string. Options that follow a file specification apply only to the file (or group of files separated by plus signs) that they follow in the command string.

You can enter the DIBOL command as one line, or you can rely on the system to prompt you for information. The DIBOL command prompt is: Files? for the input specification.

The *DIBOL-11 Language Reference Manual* contains more detailed information about using DIBOL. The following sections describe the options you can use with the DIBOL command.

/ALLOCATE:size – Use this option with /LIST or /OBJECT to reserve space on the device for the output file. The argument, size, represents the number of blocks of space to allocate. The meaningful range for this value is from 1 to 32767. A value of -1 is a special case that creates the largest file possible on the device.

/ALPHABETIZE – Use this option to alphabetize entries in the symbol and label tables. This is useful for program maintenance and debugging.

/CROSSREFERENCE – This option generates a symbol cross-reference section in the listing. This options adds as many as four separate sections to the listing. These sections are: 1) symbol cross-reference table, 2) label cross-reference table, 3) external subroutine cross-reference table, 4) COMMON cross-reference table. Note that the system does not generate a listing by default. You must also specify /LIST in the command line to get a cross-reference listing.

/LINENUMBERS – This option generates line numbers for the program during compilation. These line numbers are referenced by the symbol table segment, label table segment, and the cross-reference listing; they are especially useful in debugging DIBOL programs. This is the default operation.

/NOLINENUMBERS – This option suppresses the generation of line numbers during compilation. This produces a smaller program and optimizes execution speed. Use this option to compile only those programs that are already debugged; otherwise the DIBOL error messages are difficult to interpret.

/LIST[:filespec] – You must specify this option to produce a DIBOL compilation listing. The /LIST option has different meanings depending on where you place it in the command line.

If you specify /LIST without a file specification in the list of options that immediately follows the command name, the DIBOL compiler generates a listing that prints on the line printer. If you follow /LIST with a device name, the system creates a listing file on that device. If the device is a file-structured device, the system stores the listing file on that device, assigning it the same name as the input file with a .LST file type. The following command produces a listing on the terminal.

```
.DIBOL/LIST:TT: A
```

The next command creates a listing file called A.LST on RK3:.

```
.DIBOL/LIST:RK3: A
```

If the /LIST option contains a name and file type to override the default of .LST, the system generates a listing file with that name. The following command, for example, compiles A.DBL and B.DBL together, producing files A.OBJ and FILE1.OUT on device DK:.

```
.DIBOL/LIST:FILE1.OUT A+B
```

You cannot use a command line like the next one. In this example, the second listing file would replace the first one and, therefore, cause an error.

```
.DIBOL/LIST:FILE2 A,B
```

Another way to specify /LIST is to type it after the file specification to which it applies. To produce a listing file with the same name as a particular input file, you can use a command similar to this one:

```
.DIBOL A+B/LIST:RK3:
```

The command shown above compiles A.DBL and B.DBL together, producing files DK:A.OBJ and RK3:B.LST. If you specify a file name on a /LIST option following a file specification in the command line, it has the same meaning as when it follows the command. The following two commands have the same results:

```
.DIBOL A/LIST:B

.DIBOL/LIST:B A
```

Both the above commands generate as output files A.OBJ and B.LST.

Remember that file options apply only to the file (or group of files that are separated by plus signs) that they follow in the command string. For example:

```
.DIBOL A/LIST,B
```

This command compiles A.DBL, producing A.OBJ and A.LST. It also compiles B.DBL, producing B.OBJ. However, it does not produce any listing file for the compilation of B.DBL.

/OBJECT[:filespec] – Use this option to specify a file name or device for the object file. Because DIBOL creates object files by default, the following two commands have the same meaning.

```
.DIBOL A

.DIBOL/OBJECT A
```

Both commands compile A.DBL and produce A.OBJ as output. The /OBJECT option functions like the /LIST option; it can be either a command or a file qualifier.

As a command option, /OBJECT applies across the entire command string. The following command, for example, compiles A.DBL and B.DBL separately, creating object files A.OBJ and B.OBJ on RK1:.

```
.DIBOL/OBJECT:RK1: A,B
```

Use /OBJECT as a file option to create an object file with a specific name or destination. The following command compiles A.DBL and B.DBL together, creating files B.LST and B.OBJ.

```
.DIBOL A+B/LIST/OBJECT
```

/NOOBJECT – Use this option to suppress creation of an object file. As a command option, /NOOBJECT suppresses all object files; as a file option, it suppresses only the object file produced by the related input files. In this command, for example, the system compiles A.DBL and B.DBL together, producing files A.OBJ and B.LST. It also compiles C.DBL and produces C.LST, but does not produce C.OBJ.

```
.DIBOL A+B/LIST,C/NOOBJECT/LIST
```

/ONDEBUG – This option includes a symbol table in the object file. You can then use a debugging program to find and correct errors in the object file.

/WARNINGS – Use this option to include warning messages in DIBOL compiler diagnostic error messages. These messages call certain conditions to your attention, but they do not interfere with the compilation. This is the default operation.

/NOWARNINGS – Use this option to suppress warning messages during compilation. These messages are for your information only; they do not affect the compilation.

The DIFFERENCES command compares two files and lists the differences between them in a file or on a device.

```
DIFFERENCES [ {/OUTPUT:filespec[/ALLOCATE:size]} ]  (SP) filespec 1,filespec 2
            [ {/PRINTER                      } ]
            [ {/TERMINAL                     } ]
            [                                  ]
            [  /BLANKLINES                     ]
            [  /[NO] COMMENTS                  ]
            [  /FORMFEED                       ]
            [  /MATCH:n                        ]
            [  /[NO] SPACES                    ]
```

In the command syntax shown above, filespec1 represents the first file to be compared and filespec2 represents the second file to be compared. The default output device is the console terminal. The default file type for input files is .MAC; for output files it is .DIF. You can specify the entire command on one line, or you can rely on the system to prompt you for information. The DIFFERENCES command prompts are File 1? and File 2?.

The DIFFERENCES command is particularly useful when you want to compare two similar versions of a source program. A file comparison listing highlights the changes made to a program during an editing session. The following sections describe the various options you can use with the DIFFERENCES command. Following the descriptions of the options is a sample listing and an explanation of how to interpret it.

/ALLOCATE:size – Use this option with /OUTPUT to reserve space on the device for the output listing file. The value, size, represents the number of blocks of space to allocate. The meaningful range for this value is from 1 to 32767. A value of -1 is a special case that creates the largest file possible on the device.

/BLANKLINES – Use this option to include blank lines in the file comparison. Normally, the system disregards blank lines.

/COMMENTS – When you use this option, the system includes in the file comparison all assembly language comments (text on a line preceded by a semicolon) it finds in the two files. This is the default operation.

/NOCOMMENTS – Use this option to exclude comments (text on a line preceded by a semicolon) and spacing (spaces and tabs) from the comparison. This is useful if you are comparing two MACRO source programs with similar contents but different formats.

/FORMFEED – Use this option to include form feeds in the output listing. Normally, the system compares form feeds but does not include them in the output listing.

/MATCH:n – Use this option to specify the number of lines from each file that must agree to constitute a match. The value, n, is an integer in the range 1 to 200. The default value for n is 3.

/OUTPUT:filespec – Use this option to specify a device and file name for the output listing file. Normally, the listing appears on the console terminal. If you omit the file type for the listing file, the system uses .DIF.

/PRINTER – Use this option to print the listing of differences on the printer. Normally, the listing appears on the console terminal.

/SPACES – This option includes spacing (spaces and tabs) in the file comparison. This is the default operation. This is particularly useful when you are comparing two text files and must pay careful attention to spacing.

/NOSPACES – Use this option to exclude spacing (spaces and tabs) from the file comparison. This is useful when you are comparing two source programs whose contents are similar but whose formats are different.

/TERMINAL – Use this option to make the list of differences appear on the console terminal. This is the default operation.

To understand how to interpret the output listing, first look at the following two text files.

```
.TYPE FILE1.TXT
       FILE1
HERE'S A BOTTLE AND AN HONEST FRIEND!
  WHAT WAD YE WISH FOR MAIR, MAN?
WHA KENS, BEFORE HIS LIFE MAY END,
  WHAT HIS SHAME MAY BE O' CARE, MAN?
THEN CATCH THE MOMENTS AS THEY FLY,
  AND USE THEM AS YE OUGHT, MAN: --
BELIEVE ME, HAPPINESS IS SLY,
  AND COMES NOT AY WHEN SOUGHT, MAN.

                           --SCOTTISH SONG


.TYPE FILE2.TXT
       FILE1
HERE'S A BOTTLE AND AN HONEST FRIEND!
  WHAT WAD YE WISH FOR MAIR, MAN?
WHA KENS, BEFORE HIS LIFE MAY END,
  WHAT HIS SHARE MAY BE O' CARE, MAN?
THEN CATCH THE MOMENTS AS THEY FLY,
  AND USE THEM AS YE OUGHT, MAN: --
BELIEVE ME, HAPPINESS IS SHY,
  AND COMES NOT AY WHEN SOUGHT, MAN.

                           --SCOTTISH SONG
```

Notice that FILE1.TXT contains two typing errors. In the fourth line of the song, "shame" should be "share." In the seventh line, "sly" should be "shy."

The following command compares the two files, creating a listing file called DIFF.TXT.

```
.DIFFERENCES/MATCH:1/OUTPUT:DIFF.TXT FILE1.TXT,FILE2.TXT

%FILES ARE DIFFERENT
```

The following listing shows file DIFF.TXT.

```
.TYPE DIFF.TXT
1)1            FILE1
2)1            FILE1
```

```
1)1         WHAT HIS SHAME MAY BE O' CARE, MAN?
1)        THEN CATCH THE MOMENTS AS THEY FLY,
****
2)1         WHAT HIS SHARE MAY BE O' CARE, MAN?
2)        THEN CATCH THE MOMENTS AS THEY FLY,
**********
1)1       BELIEVE ME, HAPPINESS IS SLY,
1)          AND COMES NOT AY WHEN SOUGHT, MAN.
****
2)1       BELIEVE ME, HAPPINESS IS SHY,
2)          AND COMES NOT AY WHEN SOUGHT, MAN.
**********
```

If the files are different, the system always prints the first line of each file as identification.

```
1)1              FILE1
2)1              FILE1
```

The numbers at the left margin have the form n)m, where n represents the source file (either 1 or 2) and m represents the page of that file on which the specific line is located.

The system next prints a blank line and then lists the differences between the two files. The /MATCH:n option was used in this example to set to 1 the number of lines that must agree to constitute a match.

The first three lines of the song are the same in both files, so they do not appear in the listing. The fourth line contains the first discrepancy. The system prints the fourth line from the first file, followed by the next matching line as a reference.

```
1)1           WHAT HIS SHAME MAY BE O' CARE, MAN?
1)          THEN CATCH THE MOMENTS AS THEY FLY,
****
```

The four asterisks terminate the differences section from the first file.

The system then prints the fourth line from the second file, again followed by the next matching line as a reference:

```
2)1           WHAT HIS SHARE MAY BE O' CARE, MAN?
2)          THEN CATCH THE MOMENTS AS THEY FLY,
**********
```

The ten asterisks terminate the listing for a particular difference section.

The system scans the remaining lines in the files in the same manner. When it reaches the end of each file, it prints the %FILES ARE DIFFERENT message on the terminal.

If you compare two files that are identical, the system does not create an output file or listing, as this example shows.

```
.DIFFERENCES FILE1.TXT,FILE1.TXT
NO DIFFERENCES ENCOUNTERED
```

The DIRECTORY command lists information you request about a device, a file, or a group of files.

```
DIRECTORY [ { /OUTPUT:filespec[/ALLOCATE:size] } ]  [(SP) filespecs[/BEGIN]]
          [ { /PRINTER                         } ]
          [ { /TERMINAL                        } ]
          [                                      ]
          [   /BADBLOCKS[/FILES]                 ]
          [                                      ]
          [   /DOS[/OWNER:[nnn,nnn]]             ]
          [                                      ]
          [   /INTERCHANGE                       ]
          [                                      ]
          [   /TOPS                              ]
          [                                      ]
          [   /VOLUMEID                          ]
          [                                      ]
          [ { /BEFORE[date] }                    ]
          [ { /DATE[date]   }                    ]
          [ { /NEWFILES     }                    ]
          [ { /SINCE[date]  }                    ]
          [ { /ALPHABETIZE[/REVERSE]        }    ]
          [ { /ORDER[:category] [/REVERSE]  }    ]
          [ { /SORT[:category] [/REVERSE]   }    ]
          [   /BLOCKS                            ]
          [   /BRIEF                             ]
          [   /COLUMNS:n                         ]
          [   /DELETED                           ]
          [   /EXCLUDE                           ]
          [   /FAST                              ]
          [   /FREE                              ]
          [   /FULL                              ]
          [   /OCTAL                             ]
          [   /POSITION                          ]
          [   /SUMMARY                           ]
```

In the command syntax shown above, filespecs represents the device, file, or group of files whose directory information you request. The DIRECTORY command can list directory information about a specific device, such as the number of files stored on the device, their names, and their creation dates. It can list details about certain files, too, including their names, their file types, and their size in blocks. You can specify up to six files explicitly, but you can obtain directory information about many files by using wildcards in the file specification. The DIRECTORY command can also print a device directory summary, and it can organize its listings in several ways, such as alphabetically or chronologically.

Normally, the DIRECTORY command prints listings in two columns on the terminal. Read these listings as you would read a book: read across the columns, moving from left to right, one row at a time. Directory listings that are sorted (with /ALPHABETIZE, /ORDER, or /SORT) are an exception to this. Read these listings by reading the left column from top to bottom, then reading the right column from top to bottom.

The DIRECTORY command does not prompt you for any information. If you omit the file specification, the system lists directory information about device DK:, as this example shows.

```
.DIRECTORY
 19-May-77
DXMNSJ.SYS    88 08-Apr-77     AAF   .MAC      2 19-Apr-77
FIX463.SAV     2 29-Jul-76     ABC   .MAC      4 19-Apr-77
JMUL  .OBJ     1 03-May-77     DEMOFG.MAC      5 18-Jan-77
PTCH  .BAK     1 05-May-77     CT    .SYS      5 08-Apr-77
DX    .SYS     3 08-Apr-77     MERGE .FOR      6 24-Apr-77
MYPROG.MAC     7 24-Feb-77     VTMAC .MAC      7 31-Aug-76
ALIB  .OBJ     3 03-May-77     MX    .SYS      9 08-Apr-77
DXMNFB.SYS    97 08-Apr-77     DIR   .SAV     16 08-Apr-77
DUP   .SAV    17 13-Apr-77     PIP   .SAV     16 14-Apr-77
 18 Files, 289 Blocks
 191 Free blocks
```

If you specify only a device in the file specification, the system lists directory information about all the files on that device. If you specify a file name, the system lists information about just that file, as this example shows.

```
.DIRECTORY DX0:MYPROG.MAC
 19-May-77
MYPROG.MAC     7 24-Feb-77
 1 Files, 7 Blocks
 191 Free blocks
```

The following sections describe the options you can use with the DIRECTORY command and provide sample directory listings. Some of the options accept a date or part of a date as an argument. The syntax for specifying the date is:

[:dd] [:mmm] [:yy]

where

dd	represents the day (a decimal integer in the range 1-31).
mmm	represents the first three characters of the name of the month.
yy	represents the year (a decimal integer in the range 73-99).

The default value for the date is the current system date. If you specify just the day, the system interprets it as the given day of the current month and year. If you specify just the month, the system interprets it as the first day of the given month in the current year. If you specify only the year, the system interprets it as the start of that year. If the current system date is not set, it is considered 0 (the same as for an undated file in a directory listing).

/ALLOCATE:size – Use this option with /OUTPUT to reserve space on the device for the output listing file. The value, size, represents the number of blocks of space to allocate. The meaningful range for this value is from 1 to 32767. A value of -1 is a special case that creates the largest file possible on the device.

/ALPHABETIZE – This option lists the directory of the device you specify in alphabetical order by file name and file type. It has the same effect as the /ORDER:NAME option.

/BADBLOCKS – Sometimes devices (disks and DECtapes) are manufactured with bad blocks, or they develop bad blocks as a result of use and age. Use the /BADBLOCKS option to scan a device and locate bad blocks on it. The system prints the absolute block number of these blocks on the devices that return hardware errors when the system tries to read them. This procedure does not destroy data that is already stored on the device. Remember that block numbers are octal and the first block on a device is block 0. If a device has no bad blocks, only the heading prints, as this example shows.

```
.DIRECTORY/BADBLOCKS DX1:
BAD BLOCKS  TYPE  FILENAME    REL BLK
```

/BEFORE[date] – This option prints a directory of files created before the date you specify. The following command lists on the terminal all files stored on device DX0: that were created before April 1977.

```
.DIRECTORY/BEFORE:APR DX0:
 24-May-77
FIX463.SAV      2 29-Jul-76     DEMOFG.MAC     5 18-Jan-77
MYPROG.MAC      7 24-Feb-77     VTMAC .MAC     7 31-Aug-76
 4 Files, 21 Blocks
 191 Free blocks
```

/BEGIN – This option lists the directory of the device you specify, beginning with the file you name and including all the files that follow it in the directory. The occurrence of file names in the listing is the same as the order of the files on the device.

The following example lists the file VTMAC.MAC on device DX0: and all the files that follow it in the directory.

```
.DIRECTORY DX0:VTMAC.MAC/BEGIN
 24-May-77
VTMAC .MAC      7 31-Aug-76     ALIB  .OBJ     3 03-May-77
MX    .SYS      9 08-Apr-77     DXMNFB.SYS    97 08-Apr-77
DIR   .SAV     16 08-Apr-77     DUP   .SAV    17 13-Apr-77
PIP   .SAV     16 14-Apr-77
 7 Files, 165 Blocks
 191 Free blocks
```

/BLOCKS – This option prints a directory of the device you specify and includes the starting block number in decimal of all the files listed. The following example lists the directory of DX0:, including the starting block numbers of files.

```
.DIRECTORY/BLOCKS DX0:
 19-May-77
DXMNSJ.SYS  88 08-Apr-77   14    AAF   .MAC     2 19-Apr-77   102
FIX463.SAV   2 29-Jul-76  104    ABC   .MAC     4 19-Apr-77   106
JMUL  .OBJ   1 03-May-77  110    DEMOFG.MAC     5 18-Jan-77   138
PTCH  .BAK   1 05-May-77  143    CT    .SYS     5 08-Apr-77   150
DX    .SYS   3 08-Apr-77  155    MERGE .FOR     6 24-Apr-77   158
MYPROG.MAC   7 24-Feb-77  164    VTMAC .MAC     7 31-Aug-76   171
ALIB  .OBJ   3 03-May-77  178    MX    .SYS     9 08-Apr-77   189
DXMNFB.SYS  97 08-Apr-77  207    DIR   .SAV    16 08-Apr-77   327
DUP   .SAV  17 13-Apr-77  343    PIP   .SAV    16 14-Apr-77   360
 18 Files, 289 Blocks
 191 Free blocks
```

/BRIEF – This option lists only file names and file types, omitting file lengths and associated dates. It produces a 5-column listing, as the following example shows.

```
.DIRECTORY/BRIEF DX0:
 19-May-77
DXMNSJ.SYS    AAF   .MAC    FIX463.SAV    ABC   .MAC  JMUL  .OBJ
DEMOFG.MAC    PTCH  .BAK    CT    .SYS    DX    .SYS  MERGE .FOR
MYPROG.MAC    VTMAC .MAC    ALIB  .OBJ    MX    .SYS  DXMNFB.SYS
DIR   .SAV    DUP   .SAV    PIP   .SAV
 18 Files, 289 Blocks
 191 Free blocks
```

/COLUMNS:n – Use this option to list a directory in a specific number of columns. The value, n, represents an integer in the range 1-9. Normally, the system uses two columns for regular listings and five columns for brief listings. The following example lists the directory information for device MT0: in one column.

```
.DIRECTORY/COLUMNS:1/POSITION MT0:
 15-Apr-77
VTMAC .MAC      7 15-Apr-77      1
SYCND .MAC      5 15-Apr-77      2
DIRECT.MAC    112 15-Apr-77      3
PIPSYM.MAC      4 15-Apr-77      4
PIP005.MAC    176 15-Apr-77      5
VTMAC .MAC      7 15-Apr-77      6
SYCND .MAC      5 15-Apr-77      7
DIRECT.MAC    112 15-Apr-77      8
PIPSYM.MAC      4 15-Apr-77      9
PIP005.MAC    176 15-Apr-77     10
 10 Files, 608 Blocks
```

In the example shown above, the numbers in the rightmost column represent the magtape file sequence numbers, which appear because of the /POSITION option.

/DATE[date] – Use this option to include in the directory listing only those files with the date you specify. The following command lists all the files on device DX0: that were created on 8 April 1977.

```
.DIRECTORY/DATE:8:APR:77 DX0:
 19-May-77
DXMNSJ.SYS     88 08-Apr-77    CT    .SYS      5 08-Apr-77
DX    .SYS      3 08-Apr-77    MX    .SYS      9 08-Apr-77
DXMNFB.SYS     97 08-Apr-77    DIR   .SAV     16 08-Apr-77
 6 Files, 218 Blocks
 191 Free blocks
```

/DELETED – This option lists a directory of the device you specify, listing the file names, types, sizes, creation dates and starting block numbers in decimal of files that have been deleted but whose file name information has not been destroyed. The file names that print represent either tentative files or files that have been deleted. This can be useful in recovering files that have been accidentally deleted. Once you identify the file name and location, you can use DUP to rename the area. See Section 8.2.1 for this procedure. The following command lists files on device DT1: that have been deleted.

```
.DIRECTORY/DELETED DT1:
 19-May-77
TEST  .LST  530 27-Apr-77     48
 0 Files, 0 Blocks
 0 Free blocks
```

Note in the example shown above that, since a deleted file does not really exist, the total number of files, blocks, and free blocks is 0.

/DOS – Use this option to list the directory of a device that is in RSTS/E or DOS/BATCH format. The only other options valid with /DOS are /BRIEF, /FAST, and /OWNER. The valid devices are DECtape and RK05.

/EXCLUDE – This option lists a directory of all the files on a device except those files you specify. The following example lists all files on DX0: except the .SAV and .SYS files.

```
.DIRECTORY/EXCLUDE DX0:(*.SAV,*.SYS)
 24-May-77
AAF   .MAC      2 19-Apr-77     ABC   .MAC      4 19-Apr-77
JMUL  .OBJ      1 03-May-77     DEMOFG.MAC      5 18-Jan-77
PTCH  .BAK      1 05-May-77     MERGE .FOR      6 24-Apr-77
MYPROG.MAC      7 24-Feb-77     VTMAC .MAC      7 31-Aug-76
ALIB  .OBJ      3 03-May-77
 9 Files, 36 Blocks
 191 Free blocks
```

/FAST – This option lists only file names and file types, omitting file lengths and associated dates. This option is the same as /BRIEF.

/FILES – Use this option with /BADBLOCKS to print the file names of bad blocks. This is particularly useful if the device is not a standard RT-11 directory-structured device. If the system does not find any bad blocks, it prints only the heading, as this example shows.

```
.DIRECTORY/BADBLOCKS/FILES DT1:
BAD BLOCKS  TYPE  FILENAME    REL BLK
```

/FREE – Use this option to print a directory of unused areas and their size. This example lists the unused areas on device DK:.

```
.DIRECTORY/FREE
 19-May-77
< UNUSED >    1          < UNUSED >      1
< UNUSED >    1          < UNUSED >      2
< UNUSED >    1          < UNUSED >      2
< UNUSED >   24          < UNUSED >     38
< UNUSED >   40          < UNUSED >      3
< UNUSED >    1          < UNUSED >      2
< UNUSED >    5          < UNUSED >      2
< UNUSED >   98
 0 Files, 0 Blocks
 221 Free blocks
```

/FULL – This option lists the entire directory, including unused areas and their sizes in blocks (decimal). The following example lists the entire directory for device DT0:.

```
.DIRECTORY/FULL DT0:
 19-May-77
EDIT1 .DEM      1 03-May-77     EDIT2 .DEM      1 03-May-77
FIX463.SAV      2 29-Jul-76     FILE1 .TXT      1 19-May-77
EDIT3 .DEM      1 03-May-77     FORTRA.SAV    201 01-May-77
PUTSTR.OBJ      7 14-Apr-77     PROMPT.KEP      2 05-May-77
```

```
PROMPT.SAV      2 05-May-77    ROOT   .SAV      3 05-May-77
ROOT   .KEP     3 05-May-77    PROMPT.BAK       2 05-May-77
PROMPT.MAC      2 05-May-77    PROMPT.OBJ       1 05-May-77
OVRLAY.BAK      1 05-May-77    PTCH   .BAK      1 05-May-77
PTCH   .MAC     1 05-May-77    OVRLAY.MAC       1 05-May-77
PTCH   .OBJ     1 05-May-77    OVRLAY.OBJ       1 05-May-77
FILE2 .TXT      1 19-May-77    < UNUSED >     328
 21 Files, 236 Blocks
 328 Free blocks
```

/INTERCHANGE – Use this option to list the directory of a diskette that is in interchange (proposed ANSI standard) format. The only other options valid with /INTERCHANGE are /BRIEF and /FAST.

/NEWFILES – This option includes in the directory listing only those files that were created today. This is a convenient way to list the files you created in a session at the computer. The following command lists the new files on 19 May 1977

```
.DIRECTORY/NEWFILES DT0:
 19-May-77
FILE1 .TXT      1 19-May-77    FILE2 .TXT       1 19-May-77
 2 Files, 2 Blocks
 328 Free blocks
```

/OCTAL – This option lists the sizes (and starting block numbers if you also use /BLOCKS) in octal. If the device you specify is a magtape or cassette, the system prints the sequence numbers in octal. The following example shows an octal listing of device DX0:.

```
.DIRECTORY/OCTAL DX0:
 19-May-77
DXMNSJ.SYS    130 08-Apr-77    AAF    .MAC      2 19-Apr-77
FIX463.SAV      2 29-Jul-76    ABC    .MAC      4 19-Apr-77
JMUL   .OBJ     1 03-May-77    DEMOFG.MAC       5 18-Jan-77
PTCH   .BAK     1 05-May-77    CT     .SYS      5 08-Apr-77
DX     .SYS     3 08-Apr-77    MERGE .FOR       6 24-Apr-77
MYPROG.MAC      7 24-Feb-77    VTMAC .MAC       7 31-Aug-76
ALIB   .OBJ     3 03-May-77    MX     .SYS     11 08-Apr-77
DXMNFB.SYS    141 08-Apr-77    DIR    .SAV     20 08-Apr-77
DUP    .SAV    21 13-Apr-77    PIP    .SAV     20 14-Apr-77
 18 Files,    441 Blocks
    277 Free blocks
```

/ORDER[:category] – This option sorts the directory of a device according to the category you specify. Table 4-3 summarizes the categories and their functions.

Table 4-3 Sort Categories

Category	Explanation
DATE	Sorts the directory chronologically by creation date. Files that have the same date are sorted alphabetically by file name and file type
NAME	Sorts the directory alphabetically by file name. Files that have the same file name are sorted alphabetically by file type (this has the same effect as the /ALPHABETIZE option).
POSITION	Lists the files in order by their position on the device. This is the same as using /ORDER with no category.
SIZE	Sorts the directory based on file size in blocks. Files that are the same size are sorted alphabetically by file name and file type.
TYPE	Sorts the directory alphabetically by file type. Files that have the same file type are sorted alphabetically by file name.

The following examples list the directory of device DX0:, in order by each of the categories.

```
.DIRECTORY/ORDER:DATE DX0:
 19-May-77
FIX463.SAV     2 29-Jul-76    MX    .SYS    9 08-Apr-77
VTMAC .MAC     7 31-Aug-76    DUP   .SAV   17 13-Apr-77
DEMOFG.MAC     5 18-Jan-77    PIP   .SAV   16 14-Apr-77
MYPROG.MAC     7 24-Feb-77    AAF   .MAC    2 19-Apr-77
CT    .SYS     5 08-Apr-77    ABC   .MAC    4 19-Apr-77
DIR   .SAV    16 08-Apr-77    MERGE .FOR    6 24-Apr-77
DX    .SYS     3 08-Apr-77    ALIB  .OBJ    3 03-May-77
DXMNFB.SYS    97 08-Apr-77    JMUL  .OBJ    1 03-May-77
DXMNSJ.SYS    88 08-Apr-77    PTCH  .BAK    1 05-May-77
 18 Files, 289 Blocks
 191 Free blocks


.DIRECTORY/ORDER:NAME DX0:
 19-May-77
AAF   .MAC     2 19-Apr-77    DXMNSJ.SYS   88 08-Apr-77
ABC   .MAC     4 19-Apr-77    FIX463.SAV    2 29-Jul-76
ALIB  .OBJ     3 03-May-77    JMUL  .OBJ    1 03-May-77
CT    .SYS     5 08-Apr-77    MERGE .FOR    6 24-Apr-77
DEMOFG.MAC     5 18-Jan-77    MX    .SYS    9 08-Apr-77
DIR   .SAV    16 08-Apr-77    MYPROG.MAC    7 24-Feb-77
DUP   .SAV    17 13-Apr-77    PIP   .SAV   16 14-Apr-77
DX    .SYS     3 08-Apr-77    PTCH  .BAK    1 05-May-77
DXMNFB.SYS    97 08-Apr-77    VTMAC .MAC    7 31-Aug-76
 18 Files, 289 Blocks
 191 Free blocks


.DIRECTORY/ORDER:POSITION DX0:
 19-May-77
DXMNSJ.SYS    88 08-Apr-77    MERGE .FOR    6 24-Apr-77
AAF   .MAC     2 19-Apr-77    MYPROG.MAC    7 24-Feb-77
FIX463.SAV     2 29-Jul-76    VTMAC .MAC    7 31-Aug-76
ABC   .MAC     4 19-Apr-77    ALIB  .OBJ    3 03-May-77
JMUL  .OBJ     1 03-May-77    MX    .SYS    9 08-Apr-77
DEMOFG.MAC     5 18-Jan-77    DXMNFB.SYS   97 08-Apr-77
PTCH  .BAK     1 05-May-77    DIR   .SAV   16 08-Apr-77
CT    .SYS     5 08-Apr-77    DUP   .SAV   17 13-Apr-77
DX    .SYS     3 08-Apr-77    PIP   .SAV   16 14-Apr-77
 18 Files, 289 Blocks
 191 Free blocks


.DIRECTORY/ORDER:SIZE DX0:
 19-May-77
JMUL  .OBJ     1 03-May-77    MERGE .FOR    6 24-Apr-77
PTCH  .BAK     1 05-May-77    MYPROG.MAC    7 24-Feb-77
AAF   .MAC     2 19-Apr-77    VTMAC .MAC    7 31-Aug-76
FIX463.SAV     2 29-Jul-76    MX    .SYS    9 08-Apr-77
ALIB  .OBJ     3 03-May-77    DIR   .SAV   16 08-Apr-77
```

```
DX     .SYS     3 08-Apr-77    PIP    .SAV    16 14-Apr-77
ABC    .MAC     4 19-Apr-77    DUP    .SAV    17 13-Apr-77
CT     .SYS     5 08-Apr-77    DXMNSJ.SYS     88 08-Apr-77
DEMOFG.MAC      5 18-Jan-77    DXMNFB.SYS     97 08-Apr-77
 18 Files, 289 Blocks
 191 Free blocks


.DIRECTORY/ORDER:TYPE DX0:
 19-May-77
PTCH   .BAK     1 05-May-77    DIR    .SAV    16 08-Apr-77
MERGE  .FOR     6 24-Apr-77    DUP    .SAV    17 13-Apr-77
AAF    .MAC     2 19-Apr-77    FIX463.SAV      2 29-Jul-76
ABC    .MAC     4 19-Apr-77    PIP    .SAV    16 14-Apr-77
DEMOFG.MAC      5 18-Jan-77    CT     .SYS     5 08-Apr-77
MYPROG.MAC      7 24-Feb-77    DX     .SYS     3 08-Apr-77
VTMAC  .MAC     7 31-Aug-76    DXMNFB.SYS     97 08-Apr-77
ALIB   .OBJ     3 03-May-77    DXMNSJ.SYS     88 08-Apr-77
JMUL   .OBJ     1 03-May-77    MX     .SYS     9 08-Apr-77
 18 Files, 289 Blocks
 191 Free blocks
```

/OUTPUT:filespec – Use this option to specify a device and file name for the output listing file. Normally, the directory listing appears on the console terminal. If you omit the file type for the listing file, the system uses .DIR.

/OWNER:[nnn,nnn] – Use this option with /DOS to specify a user identification code (UIC). Note that the square brackets are part of the UIC; you must type them.

/POSITION – Use this option to list the file sequence numbers of files stored on a magtape. See /COLUMNS:n for a sample listing.

/PRINTER – Use this option to print the directory listing on the line printer. The default output device is the terminal.

/REVERSE – This option lists a directory in the reverse order of the sort you specify with /ALPHABETIZE, /ORDER, or /SORT. The following example sorts the directory of DX0: and lists it in reverse order by size.

```
.DIRECTORY/ORDER:SIZE/REVERSE DX0:
 24-May-77
DXMNFB.SYS     97 08-Apr-77    CT     .SYS     5 08-Apr-77
DXMNSJ.SYS     88 08-Apr-77    DEMOFG.MAC      5 18-Jan-77
DUP    .SAV    17 13-Apr-77    ABC    .MAC     4 19-Apr-77
DIR    .SAV    16 08-Apr-77    ALIB   .OBJ     3 03-May-77
PIP    .SAV    16 14-Apr-77    DX     .SYS     3 08-Apr-77
MX     .SYS     9 08-Apr-77    AAF    .MAC     2 19-Apr-77
MYPROG.MAC      7 24-Feb-77    FIX463.SAV      2 29-Jul-76
VTMAC  .MAC     7 31-Aug-76    JMUL   .OBJ     1 03-May-77
MERGE  .FOR     6 24-Apr-77    PTCH   .BAK     1 05-May-77
 18 Files, 289 Blocks
 191 Free blocks
```

/SINCE[date] – This option lists a directory of all files stored on the device you specify that were created on or after the date you specify. The following command lists only those files on DX0: that were created on or after 3 May 1977.

```
.DIRECTORY/SINCE:3:MAY:77 DX0:
 19-May-77
JMUL  .OBJ      1 03-May-77     PTCH  .BAK      1 05-May-77
ALIB  .OBJ      3 03-May-77
 3 Files, 5 Blocks
 191 Free blocks
```

/SORT[**:category**] – This option sorts the directory of a device according to the category you specify. This is the same as /ORDER[:category].

/SUMMARY – This option lists a summary of the segment structure of the device directory. The following example lists the segment structure of the directory for device DK:.

```
.DIRECTORY/SUMMARY
 19-May-77

   72 Files in segment 1

   46 Files in segment 2

   36 Files in segment 4

   33 Files in segment 3

   33 Files in segment 5

   16 Available segments, 5 in use

 220 Files, 4543 Blocks
 219 Free blocks
```

/TERMINAL – This option lists directory information on the console terminal. This is the default operation.

/TOPS – Use this option to list the directory of a DECtape that is in PDP-10 format. The only other options valid with /TOPS are /BRIEF and /FAST.

/VOLUMEID – Use this option to display the volume identification of a particular device. The following example displays the volume ID of device DK:.

```
.DIRECTORY/VOLUMEID

 VOL ID=BL10
 OWNER NAME=JOYCE
```

The DUMP command can print on the terminal or line printer, or write to a file all or any part of a file in octal words, octal bytes, ASCII characters, and/or Radix-50 characters. It is particularly useful for examining directories and files that contain binary data.

```
DUMP [ {/OUTPUT:filespec[/ALLOCATE:size]}  ]  (SP) filespec
     [ {/PRINTER                        }  ]
     [ {/TERMINAL                       }  ]
     [                                     ]
     [  /[NO]ASCII                         ]
     [  /BYTES                             ]
     [  /IGNORE                            ]
     [  /ONLY:block                        ]
     [  /RAD50                             ]
     [  [/START:block] [/END:block]        ]
     [  /WORDS                             ]
```

In the command syntax shown above, filespec represents the device or file you need to examine. If you do not specify an output file, the listing prints on the line printer. If you do not specify a file type for an output file, the system uses .DMP. You can specify the entire command on one line, or you can rely on the system to prompt you for information. The DUMP command prompt is Device or file?.

Notice that some of the options (/ONLY, /START, and /END) accept a block number as an argument. Remember that all block numbers are in octal, and that the first block of a device or file is block 0. To specify a decimal block number, follow the number by a decimal point. If you are dumping a file, the block numbers you specify are relative to the beginning of that file. If you are dumping a device, the block numbers are the absolute (physical) block numbers on that device.

The system handles operations that involve magtape and cassette differently from operations involving random access devices. If you dump an RT-11 file-structured tape and specify only a device name in the file specification, the system reads only as far as the logical end-of-tape. Logical end-of-tape is indicated by an end-of-file label followed by two tape marks. For non-file-structured tape, logical end-of-tape is indicated by two consecutive tape marks. If you dump a cassette and specify only the device name in the file specification, the results are unpredictable. For magtape dumps, tape mark messages appear in the ouptut listing as the system encounters them on the tape.

The following sections describe the options you can use with the DUMP command. Following the options are some sample listings and an explanation of how to interpret them.

/ALLOCATE:size – Use this option with /OUTPUT to reserve space on the device for the output listing file. The value, size, represents the number of blocks of space to allocate. The meaningful range for this value is from 1 to 32767. A value of -1 is a special case that creates the largest file possible on the device.

/ASCII – This option prints the ASCII equivalent of each octal word or byte that is dumped. A dot (.) represents characters that are not printable. This is the default operation.

/NOASCII – Use this option to suppress the ASCII output, which appears in the right hand column of the listing. This allows the listing to fit in 72 columns.

/BYTES – Use this option to display information in octal bytes.

/END:block – Use this option to specify an ending block number for the dump. The system dumps the device or file you specify beginning with block 0 (unless you use /START) and continuing until it dumps the block you specify with /END.

/IGNORE – Use this option to ignore errors that occur during a dump operation. Use /IGNORE if an input error occurred when you tried to perform a normal dump operation.

/ONLY:block – Use this option to dump only the block number you specify.

/OUTPUT:filespec – Use this option to specify a device and file name for the output listing file. Normally, the listing appears on the line printer. If you omit the file type for the listing file, the system uses .DMP.

/PRINTER – This option causes the output listing to appear on the line printer. This is the default operation.

/RAD50 – This option prints the Radix-50 equivalent of each octal word that is dumped.

/START:block – Use this option to specify a starting block number for the dump. The system dumps the device or file beginning at the block number you specify with /START and continuing to the end of the device or file (unless you use /END).

/TERMINAL – This option causes the output listing to appear on the console terminal. Normally, the listing appears on the line printer.

/WORDS – This option displays information in octal words. This is the default operation.

The following command dumps block 1 of the file SYSMAC.MAC. The output listing, which shows octal bytes and their ASCII equivalent, is stored in file MACLIB.DMP. The PRINT command prints the contents of the file on the line printer.

```
    .DUMP/OUTPUT:MACLIB/BYTES/ONLY:1 SYSMAC.MAC

    .PRINT MACLIB.DMP
```

```
DK:SYSMAC.MAC
BLOCK NUMBER 00001
000/ 040 124 117 040 124 110 105 123 105 040 114 111 103 105 116 123
         T   O       T   H   E   S   E       L   I   C   E   N   S
020/ 105 040 124 105 122 115 123 056 040 124 111 124 114 105 040 124
     E       T   E   R   M   S   .       T   I   T   L   E       T
040/ 117 040 101 116 104 040 117 127 116 105 122 123 110 111 120 040
     O       A   N   D       O   W   N   E   R   S   H   I   P
060/ 117 106 040 124 110 105 040 015 012 073 040 123 117 106 124 127
     O   F       T   H   E       .   .   ;       S   O   F   T   W
100/ 101 122 105 040 123 110 101 114 114 040 101 124 040 101 114 114
     A   R   E       S   H   A   L   L       A   T       A   L   L
120/ 040 124 111 115 105 123 040 122 105 115 101 111 116 040 111 116
         T   I   M   E   S       R   E   M   A   I   N       I   N
140/ 040 104 111 107 111 124 101 114 056 015 012 073 015 012 073 040
         D   I   G   I   T   A   L   .   .   .   ;   .   .   ;
160/ 124 110 105 040 111 116 106 117 122 115 101 124 111 117 116 040
     T   H   E       I   N   F   O   R   M   A   T   I   O   N
200/ 111 116 040 124 110 111 123 040 123 117 106 124 127 101 122 105
     I   N       T   H   I   S       S   O   F   T   W   A   R   E
220/ 040 111 123 040 123 125 102 112 105 103 124 040 124 117 015 012
         I   S       S   U   B   J   E   C   T       T   O   .   .
240/ 073 040 103 110 101 116 107 105 040 127 111 124 110 117 125 124
     ;       C   H   A   N   G   E       W   I   T   H   O   U   T
260/ 040 116 117 124 111 103 105 040 101 116 104 040 123 110 117 125
         N   O   T   I   C   E       A   N   D       S   H   O   U
```

```
300/ 114  104  040  116  117  124  040  102  105  040  103  117  116  123  124  122
     L    D         N    O    T         B    E         C    O    N    S    T    R
320/ 125  105  104  015  012  073  040  101  123  040  101  040  103  117  115  115
     U    E    D    .    .    ;         A    S         A         C    O    M    M
340/ 111  124  115  105  116  124  040  102  131  040  104  111  107  111  124  101
     I    T    M    E    N    T         B    Y         D    I    G    I    T    A
360/ 114  040  105  121  125  111  120  115  105  116  124  040  103  117  122  120
     L         E    Q    U    I    P    M    E    N    T         C    O    R    P
400/ 117  122  101  124  111  117  116  056  015  012  073  015  012  073  040  104
     O    R    A    T    I    O    N    .    .    .    ;    .    .    ;         D
420/ 111  107  111  124  101  114  040  101  123  123  125  115  105  123  040  116
     I    G    I    T    A    L         A    S    S    U    M    E    S         N
440/ 117  040  122  105  123  120  117  116  123  111  102  111  114  111  124  131
     O         R    E    S    P    O    N    S    I    B    I    L    I    T    Y
460/ 040  106  117  122  040  124  110  105  040  125  123  105  015  012  073  040
          F    O    R         T    H    E         U    S    E    .    .    ;
500/ 117  122  040  122  105  114  111  101  102  111  114  111  124  131  040  117
     O    R         R    E    L    I    A    B    I    L    I    T    Y         O
520/ 106  040  111  124  123  040  123  117  106  124  127  101  122  105  040  117
     F         I    T    S         S    O    F    T    W    A    R    E         O
540/ 116  040  105  121  125  111  120  115  105  116  124  015  012  073  040  127
     N         E    Q    U    I    P    M    E    N    T    .    .    ;         W
560/ 110  111  103  110  040  111  123  040  116  117  124  040  123  125  120  120
     H    I    C    H         I    S         N    O    T         S    U    P    P
600/ 114  111  105  104  040  102  131  040  104  111  107  111  124  101  114  056
     L    I    E    D         B    Y         D    I    G    I    T    A    L    .
620/ 015  012  073  015  012  073  040  105  106  054  112  104  054  114  120  054
     .    .    ;    .    .    ;         E    F    ,    J    D    ,    L    P    ,
640/ 102  103  054  104  126  054  103  122  054  110  112  015  012  014  056  115
     B    C    ,    D    V    ,    C    R    ,    H    J    .    .    .    .    M
660/ 101  103  122  117  040  056  056  126  061  056  056  015  012  056  115  103
     A    C    R    O         .    .    V    1    .    .    .    .    .    M    C
700/ 101  114  114  011  056  056  056  103  115  060  054  056  056  056  103  115
     A    L    L    .    .    .    .    C    M    0    ,    .    .    .    C    M
720/ 061  054  056  056  056  103  115  062  054  056  056  056  103  115  063  054
     1    ,    .    .    .    C    M    2    ,    .    .    .    C    M    3    ,
740/ 056  056  056  103  115  064  054  056  056  056  103  115  065  054  056  056
     .    .    .    C    M    4    ,    .    .    .    C    M    5    ,    .    .
760/ 056  103  115  066  015  012  056  056  056  126  061  075  061  056  015  012
     .    C    M    6    .    .    .    .    .    V    1    =    1    .    .    .
```

In the printout above, the heading shows which file was dumped and which block of the file follows. The numbers in the leftmost column indicate the byte offset from the beginning of the block. Remember that these are all octal values, and that there are two bytes per word. The octal bytes that were dumped appear in the next eight columns. The ASCII equivalent of each octal byte appears underneath the byte. The system substitutes a dot (.) for non-printing codes, such as those for control characters.

The last example shows block 6 (the directory) of device RK0:. The output is in octal words with Radix-50 equivalents below each word.

```
    .DUMP/NOASCII/RAD50/ONLY:6 RK0:
```

```
RK01/N/X/016
BLOCK NUMBER  00006
000/ 000020 000002 000005 000000 000046 002000 071105 055202
       P      B      E             A     YX    RKM    NSJ
020/ 075273 000130 000015 012105 002000 071105 054162 075273
     SYS     BH      M    CI/     YX    RKM    NFB    SYS
040/ 000141 000015 012105 002000 071105 055515 075273 000150
      BQ      M    CI/     YX    RKM    NXM    SYS     BX
060/ 000015 012105 002000 015425 055202 075273 000132 000015
       M    CI/     YX    DMM    NSJ    SYS     BJ      M
100/ 012105 002000 015425 054162 075273 000143 000015 012105
     CI/     YX    DMM    NFB    SYS     BS      M    CI/
120/ 002000 015425 055515 075273 000152 000015 012105 002000
      YX    DMM    NXM    SYS     BZ      M    CI/     YX
140/ 016315 055202 075273 000130 000015 012105 002000 016315
     DXM    NSJ    SYS     BH      M    CI/     YX    DXM
160/ 054162 075273 000141 000015 012105 002000 016315 055515
     NFB    SYS     BQ      M    CI/     YX    DXM    NXM
200/ 075273 000141 000015 012105 002000 016055 055202 075273
     SYS     BQ      M    CI/     YX    DTM    NSJ    SYS
220/ 000130 000015 012105 002000 016055 054162 075273 000141
      BH      M    CI/     YX    DTM    NFB    SYS     BQ
240/ 000015 012105 002000 016055 055515 075273 000151 000015
       M    CI/     YX    DTM    NXM    SYS     BY      M
260/ 012105 002000 016005 055202 075273 000130 000015 012105
     CI/     YX    DSM    NSJ    SYS     BH      M    CI/
300/ 002000 016005 054162 075273 000141 000015 012105 002000
      YX    DSM    NFB    SYS     BQ      M    CI/     YX
320/ 016005 055515 075273 000150 000015 012105 002000 015615
     DSM    NXM    SYS     BX      M    CI/     YX    DPM
340/ 055202 075273 000130 000015 012105 002000 015615 054162
     NSJ    SYS     BH      M    CI/     YX    DPM    NFB
360/ 075273 000141 000015 012105 002000 015615 055515 075273
     SYS     BQ      M    CI/     YX    DPM    NXM    SYS
400/ 000151 000015 012105 002000 070575 055202 075273 000130
      BY      M    CI/     YX    RFM    NSJ    SYS     BH
420/ 000015 012105 002000 070575 054162 075273 000141 000015
       M    CI/     YX    RFM    NFB    SYS     BQ      M
440/ 012105 002000 070575 055515 075273 000150 000015 012105
     CI/     YX    RFM    NXM    SYS     BX      M    CI/
460/ 002000 071105 056573 075273 000123 000015 012105 002000
      YX    RKM    NSK    SYS     BC      M    CI/     YX
500/ 016315 056573 075273 000123 000015 012105 002000 016040
     DXM    NSK    SYS     BC      M    CI/     YX    DT
520/ 000000 075273 000002 000015 012105 002000 015600 000000
            SYS      B      M    CI/     YX    DP
540/ 075273 000002 000015 012105 002000 016300 000000 075273
     SYS      B      M    CI/     YX    DX            SYS
560/ 000003 000015 012105 002000 070560 000000 075273 000002
       C      M    CI/     YX    RF            SYS      B
600/ 000015 012105 002000 071070 000000 075273 000002 000015
       M    CI/     YX    RK            SYS      B      M
620/ 012105 002000 015410 000000 075273 000004 000015 012105
     CI/     YX    DM            SYS      D      M    CI/
640/ 002000 015770 000000 075273 000002 000015 012105 002000
      YX    DS            SYS      B      M    CI/     YX
```

```
660/ 100040 000000 075273 000002 000015 012105 002000 046600
     TT            SYS      B      M    CI/     YX    LP
700/ 000000 075273 000002 000015 012105 002000 012620 000000
            SYS      B      M    CI/     YX    CR
720/ 075273 000003 000015 012105 002000 052140 000000 075273
     SYS      C      M    CI/     YX    MT            SYS
740/ 000010 000015 012105 002000 051510 000000 075273 000011
       H      M    CI/     YX    MM     SYS             I
760/ 000015 012105 002000 054540 000000 075273 000002 000015
       M    CI/     YX    NL            SYS      B      M
```

The E (Examine) command prints in octal the contents of an address on the console terminal.

```
E (SP) address[-address]
```

In the command syntax illustrated above, address represents an octal address that, when added to the relocation base value from the Base command (if you used one), provides the actual address that the system examines. This command permits you to open specific locations in memory and inspect their contents. It is most frequently used after a GET command to examine locations in a program.

The Examine command accepts both word and byte addresses, but it always executes the command as though you specified a word address. (If you specify an odd address, the system decreases it by one to make it even.)

If you specify more than one address (in the form address1-address2), the system prints the contents of address1 through address2, inclusive. The second address (address2) must always be greater than the first address. If you do not specify an address, the system prints the contents of relative location 0.

Note that you cannot examine addresses outside the background area.

The following example prints the contents of location 1000, assuming the relocation base is 0.

```
.E 1000

127401
```

The next command sets the relocation base to 1000.

```
.B 1000
```

The following command prints the contents of locations 2000 through 2005.

```
.E 1001-1005

127401 007624 127400
```

The EDIT command invokes the text editor.

```
EDIT [ { /CREATE                              } ]  (SP) filespec[/ALLOCATE:size]
     [ { /INSPECT                             } ]
     [ { /OUTPUT:filespec[/ALLOCATE:size]     } ]
```

The text editor is a program that creates or modifies ASCII text files or source files for use as input to programs such as the MACRO assembler or the FORTRAN compiler. The editor reads ASCII files from any input device, makes specified changes and writes the file on any output device. It also allows efficient use of VT11 or VS60 display hardware, if this is part of the system configuration.

The editor considers a file to be divided into logical units called pages. A page of text is generally 50-60 lines long (delimited by form feed characters) and corresponds approximately to a physical page of a program listing. The editor reads one page of text at a time from the input file into its internal buffers where the page becomes available for editing. You can then use editing commands to:

- Locate text to be changed
- Execute and verify the changes
- List an edited page on the console terminal
- Output a page of text to the output file.

In the command syntax illustrated above, filespec represents the file you need to edit. You can enter the EDIT command on one line, or you can rely on the system to prompt you for information. If you do not supply a file specification for the file to edit, the system prompts you with File?. If you do not specify any option with the EDIT command, the text editor performs an edit backup operation on the file you name in the file specification. To do this, it changes the name of the original file, giving it a file type of .BAK when you finish making your editing changes. The actual file renaming occurs when you successfully exit by using an EX, EF, or EB command. You can also perform an edit backup operation while you are working with the text editor by using the Edit Backup (EB) command, which is described in Chapter 5.

When you issue an EDIT command, the system invokes the text editor. It is possible to receive an error or warning message as a result of this command. If, for example, the file you need to edit does not exist on device DK:, the editor issues an error message and remains in control.

```
.EDIT/INSPECT EXAMP3.TXT
?EDIT-F-File not found
*^C$$
```

When a situation like this occurs, you can either issue another command directly to the text editor or enter CTRL/C followed by two ESCAPEs to return control to the monitor.

The following sections describe the options you can use with the EDIT command. A more complete description of the text editor is contained in Chapter 5.

/ALLOCATE:size – Use this option with /OUTPUT or after the file specification to reserve space on the device for the output file. The value, size, represents the number of blocks of space to allocate. The meaningful range for this value is from 1 to 32767. A value of -1 is a special case that creates the largest file possible on the device.

/CREATE – Use this option to build a new file. You can also create a new file while you are working with the text editor by using the Edit Write (EW) command, which is described in Chapter 5. The following example creates a file called NEWFIL.TXT on device DK:, inserts one line of text, and then closes the file.

```
    .EDIT/CREATE NEWFIL.TXT
    *ITHIS IS A NEW FILE.
    $$
    *EX$$
```

/INSPECT – Use this option to open a file for reading. This option does not create any new output files. You can also open a file for inspection while you are working with the text editor by using the Edit Read (ER) command, which is explained in Chapter 5.

The following command opens an existing file for inspection, lists its contents, and then exits.

```
    .EDIT/INSPECT NEWFIL.TXT
    *R$$
    */L$$
    THIS IS A NEW FILE.
    *^C$$
```

/OUTPUT:filespec – This option directs the text you edit to the file you specify, leaving the input file unchanged. You can also write text to an output file while you are working with the text editor by using the Edit Write (EW) command, which is explained in Chapter 5. The following command reads file ORIG.TXT and writes the edited text to file CHANGE.TXT.

```
    .EDIT/OUTPUT:CHANGE.TXT  ORIG.TXT
    *
```

The EXECUTE command invokes one or more language processors to assemble or compile the files you specify. It also links object modules and initiates execution of the resultant program.

```
EXECUTE[ /EXECUTE[:filespec] [/ALLOCATE:size]          ] (SP) filespecs [/LIBRARY]
       [ /LIST[:filespec] [/ALLOCATE:size]             ]                [/PASS:1 ]
       [ /MAP[:filespec] [/ALLOCATE:size] [/WIDE]      ]
       [ /OBJECT[:filespec] [/ALLOCATE:size]           ]
       [                                               ]
       [ /BOTTOM:n                                     ]
       [ /DEBUG[:filespec]                             ]
       [ /LINKLIBRARY[:filespec]                       ]
       [ /[NO] RUN                                     ]
       [                                               ]
       [ { /DIBOL                                    } ]
       [ {  [/ALPHABETIZE          ]                 } ]
       [ {  [/CROSSREFERENCE       ]                 } ]
       [ {  [/[NO] LINENUMBERS     ]                 } ]
       [ {  [/ONDEBUG              ]                 } ]
       [ {  [/[NO] WARNINGS        ]                 } ]
       [ { /FORTRAN                                  } ]
       [ {  [/CODE:type            ]                 } ]
       [ {  [/DIAGNOSE             ]                 } ]
       [ {  [/EXTEND               ]                 } ]
       [ {  [/HEADER               ]                 } ]
       [ {  [/I4                   ]                 } ]
       [ {  [/[NO] LINENUMBERS     ]                 } ]
       [ {  [/ONDEBUG              ]                 } ]
       [ {  [/[NO] OPTIMIZE[:type] ]                 } ]
       [ {  [/RECORD:length        ]                 } ]
       [ {  [/SHOW[:value]         ]                 } ]
       [ {  [/STATISTICS           ]                 } ]
       [ {  [/[NO] SWAP            ]                 } ]
       [ {  [/UNITS:n              ]                 } ]
       [ {  [/[NO] VECTORS         ]                 } ]
       [ {  [/WARNINGS             ]                 } ]
       [ { /MACRO                                    } ]
       [ {  [/CROSSREFERENCE[:type[...:type]]]       } ]
       [ {  [/DISABLE:value[...:value]       ]       } ]
       [ {  [/ENABLE:value[...:value]        ]       } ]
       [ {  [/[NO] SHOW:value                ]       } ]
```

In the command line shown above, filespecs represents one or more files to be included in the compilation assembly. The default file types for the output files are .LST for listing files, .MAP for load map files, .OBJ for object files, and .SAV for memory image files. The defaults for input files depend on the particular language processor involved. These defaults include .MAC for MACRO files, .FOR for FORTRAN files, and .DBL for DIBOL files.

To compile (or assemble) multiple source files into a single object file, separate the files by plus (+) signs in the command line. Unless you specify otherwise, the system creates an object file with the same name as the first input file and gives it an .OBJ file type. To compile multiple files in independent compilations, separate the files by commas (,) in the command line. This generates a corresponding object file for each set of input files. The system then links together all the object files and creates a single executable file. You can combine up to six files for a compilation producing a single object file. You can specify the entire EXECUTE command as one line, or you can rely on the system to prompt you for information. The EXECUTE command prompt is Files?.

There are several ways to establish which language processor the EXECUTE command invokes. One way is to specify a language-name option, such as /MACRO, which invokes the MACRO assembler. Another way is to omit the language-name option and explicitly specify the file type for the source files. The EXECUTE command then invokes the language processor that corresponds to that file type. Specifying the file SOURCE.MAC, for example, invokes the MACRO assembler. A third way to establish the language processor is to let the system choose a file type of .MAC, .DBL, or .FOR for the source file you name.

To do this, the handler for the device you specify must be loaded. If you specify DX1:A, and the DX handler is loaded, the system searches for source files A.MAC and A.DBL, in that order. If it finds one of these files, the system invokes the corresponding language processor. If it cannot find one of these files, or if the device handler associated with the input file is not resident, the system assumes a file type of .FOR and invokes the FORTRAN compiler.

If the language processor selected as a result of one of the procedures described above is not on the system device (SY:), the system issues an error message.

Language options are position dependent. That is, they have different meanings depending on where you place them in the command line. Options that qualify a command name apply across the entire command string. Options that follow a file specification apply only to the file (or group of files separated by plus signs) that they follow in the command string.

The following sections describe the options you can use with the EXECUTE command.

/ALLOCATE:size – Use this option with /EXECUTE, /LIST, /MAP, or /OBJECT to reserve space on the device for the output file. The argument, size, represents the number of blocks of space to allocate. The meaningful range for this value is from 1 to 32767. A value of -1 is a special case that creates the largest file possible on the device.

/ALPHABETIZE – Use this option with DIBOL to alphabetize the entries in the symbol table listing. This is useful for program maintenance and debugging.

/BOTTOM:n – Use this option to specify the lowest address to be used by the relocatable code in the load module. The argument, n, represents a 6-digit unsigned even octal number. If you do not use this option, the system positions the load module so that the lowest address is location 1000 (octal). This option is illegal for foreground links.

/CODE:type – Use this option with FORTRAN to produce object code that is designed for a particular hardware configuration. The argument, type, represents a three-letter abbreviation for the type of code to produce. The legal values are the following: EAE, EIS, FIS, and THR. See Section 1.1.1, Compiler Generated Code, of the *RT-11/RSTS/E FORTRAN IV User's Guide* for a complete description of the types of code and their function.

/CROSSREFERENCE[:type[. . . :type]] – Use this option with MACRO or DIBOL to generate a symbol cross-reference section in the listing. This information is useful for program maintenance and debugging. Note that the system does not generate a listing by default. You must also specify /LIST in the command line to get a cross-reference listing.

With MACRO, this option takes an optional argument. The argument, type, represents a one-character code that indicates which sections of the cross-reference listing the assembler should include. Table 4-10 summarizes the valid arguments and their meaning.

/DEBUG[:filespec] – Use this option to link ODT (online debugging technique, described in Chapter 16) with your program to help you debug it. If you supply the name of another debugging program, the system links the debugger you specify with your program. The debugger is always linked low in memory relative to your program.

/DIAGNOSE – Use the option with FORTRAN to help analyze an internal compiler error. /DIAGNOSE expands the crash dump information to include internal compiler tables and buffers. Submit the diagnostic printout to DIGITAL with an SPR form. The information in the listing can help the DIGITAL programmers locate the compiler error and correct it.

/DIBOL – This option invokes the DIBOL language processor to compile the associated files.

/DISABLE:value[. . . :value] – Use this option with MACRO to specify a .DSABL directive. Table 4-11 summarizes the arguments and their meaning. See Section 6.2 of the *PDP-11 MACRO Language Reference Manual* for a description of the directive and a list of all legal values.

/ENABLE:value[. . . :value] – Use this option with MACRO to specify an .ENABL directive. Table 4-11 summarizes the arguments and their meaning. See Section 6.2 of the *PDP-11 MACRO Language Reference Manual* for a description of the directive and a list of all legal values.

/EXECUTE[:filespec] – Use this option to specify a file name or device for the executable file. Because the EXECUTE command creates executable files by default, the following two commands have the same meaning:

```
.EXECUTE  MYPROG

.EXECUTE/EXECUTE  MYPROG
```

Both commands link MYPROG.OBJ and produce MYPROG.SAV as a result. The /EXECUTE option has different meanings when it follows the command and when it follows the file specification. The following command creates an executable file called PROG1.SAV on device RK1:.

```
.EXECUTE/EXECUTE:RK1:  PROG1,PROG2
```

The next command creates an executable file called MYPROG.SAV on device DK:.

```
.EXECUTE  RTN1,RTN2,MYPROG/EXECUTE
```

/EXTEND – Use this option with FORTRAN to change the right margin for source input lines from column 72 to column 80.

/FORTRAN – This option invokes the FORTRAN language processor to compile the associated files.

/HEADER – Use this option with FORTRAN to include in the printout a list of options that are currently in effect.

/I4 – Use this option with FORTRAN to allocate two words for the default integer data type (FORTRAN only uses one-word integers) so that it takes the same physical space as real variables.

/LIBRARY – Use this option with MACRO to identify the file the option qualifies as a macro library file. Use it only after a library file specification in the command line. The MACRO assembler looks first to the library associated with the most recent /LIBRARY option to satisfy references (made with the .MCALL directive) from MACRO programs. It then looks to any libraries you specified earlier in the command line, and it looks last to SYSMAC.SML.

In the example below, the two files A.FOR and B.FOR are compiled together, producing B.OBJ and B.LST. The MACRO assembler assembles C.MAC, satisfying .MCALL references from MYLIB.MAC and SYSMAC.SML. It produces C.OBJ and C.LST. The system then links B.OBJ and C.OBJ together, resolving undefined references from SYSLIB.OBJ and produces the executable file B.SAV. Finally, the system loads and executes B.SAV.

```
.EXECUTE  A+B/LIST/OBJECT,MYLIB/LIBRARY+C.MAC/LIST/OBJECT
```

/LINENUMBERS – Use this option with DIBOL or FORTRAN to include internal sequence numbers in the executable program. These are especially useful in debugging programs. This is the default operation.

/NOLINENUMBERS – Use this option with DIBOL or FORTRAN to suppress the generation of internal sequence numbers in the executable program. This produces a smaller program and optimizes execution speed. Use this option to compile only those programs that are already debugged; otherwise the DIBOL or FORTRAN error messages are difficult to interpret.

/LINKLIBRARY:filespec – Use this option to include the library file name you specify as an object module library during the linking operation. Repeat the option if you need to specify more than one library file.

/LIST[:filespec] – You must specify this option to produce a compilation or assembly listing. The /LIST option has different meanings depending on where you put it in the command line.

If you specify /LIST without a file specification in the list of options that immediately follows the command name, the system generates a listing that prints on the line printer. If you follow /LIST with a device name, the system creates a listing file on that device. If the device is a file-structured device, the system stores the listing file on that device, assigning it the same name as the input file with a .LST file type. The following command produces a listing on the terminal.

```
.EXECUTE/LIST:TT A.FOR
```

The next command creates a listing file called A.LST on RK3:.

```
.EXECUTE/LIST:RK3: A.MAC
```

If the /LIST option contains a name and file type to override the default of .LST, the system generates a listing file with that name. The following command, for example, compiles A.FOR and B.FOR together, producing files A.OBJ and FILE1.OUT on device DK:. It then links A.OBJ (using SYSLIB.OBJ as needed) and produces A.SAV.

```
.EXECUTE/NORUN/FORTRAN/LIST:FILE1.OUT A+B
```

You cannot use a command line like the next one. In this example, the second listing file would replace the first one and, therefore, cause an error.

```
.EXECUTE/LIST:FILE2 A.MAC,B.MAC
```

Another way to specify /LIST is to type it after the file specification to which it applies. To produce a listing file with the same name as a particular input file, you can use a command similar to this one:

```
.EXECUTE/DIBOL A+B/LIST:RK3:
```

The command shown above compiles A.DBL and B.DBL together, producing files DK:A.OBJ and RK3:B.LST. It then links A.OBJ (using SYSLIB.OBJ as needed) and produces DK:A.SAV. If you specify a file name on a /LIST option following a file specification in the command line, it has the same meaning as when it follows the command. The following two commands have the same results.

```
.EXECUTE/MACRO A/LIST:B

.EXECUTE/MACRO/LIST:B A
```

Remember that file options apply only to the file (or group of files that are separated by plus signs) that they follow in the command string. For example:

```
.EXECUTE/NORUN A.MAC/LIST,B.FOR
```

This command compiles A.MAC, producing A.OBJ and A.LST. It also compiles B.FOR, producing B.OBJ. However, it does not produce any listing file for the compilation of B.FOR. Finally, the system links A.OBJ and B.OBJ together, producing A.SAV.

/MACRO – This option invokes the MACRO assembler to assemble the associated files.

/MAP[:filespec] – You must specify this option to produce a load map after a link operation. The /MAP option has different meanings depending on where you put it in the command line. It follows the same general rules outlined above for /LIST.

/OBJECT[:filespec] – Use this option to specify a file name or device for the object file. Because the EXECUTE command creates object files by default, the following two commands have the same meaning:

```
.EXECUTE/FORTRAN A

.EXECUTE/FORTRAN/OBJECT A
```

Both commands compile A.FOR and produce A.OBJ as output. The /OBJECT option functions like the /LIST option; it can be either a command or a file qualifier.

As a command option, /OBJECT applies across the entire command string. The following command, for example, assembles A.MAC and B.MAC separately, creating object files A.OBJ and B.OBJ on RK1:.

```
.EXECUTE/OBJECT:RK1: A.MAC,B.MAC
```

Use /OBJECT as a file option to create an object file with a specific name or destination. The following command compiles A.DBL and B.DBL together, creating files B.LST, B.OBJ, and B.SAV.

```
.EXECUTE/DIBOL A+B/LIST/OBJECT/EXECUTE
```

/ONDEBUG – Use this option with DIBOL to include a symbol table in the object file. You can then use a debugging program to find and correct errors in the object file.

Use /ONDEBUG with FORTRAN to include debug lines (those that have a D in column one) in the compilation. You do not, therefore, have to edit the file to include these lines in the compilation or to logically remove them. This option is useful in debugging a program. You can include messages, flags, and conditional branches to help you trace program execution and find an error.

/OPTIMIZE:type – Use this option with FORTRAN to enable certain options that optimize object code for various conditions. The value, type, represents the three-letter code for the type of optimization to enable. Table 4-4 summarizes the codes and their meanings.

/NOOPTIMIZE:type – Use this option with FORTRAN to disable certain options that optimize object code for various conditions. The value, type, represents the three-letter code for the type of optimization to disable. Table 4-4 summarizes the codes and their meanings.

/PASS:1 – Use this option with MACRO on a prefix macro file to process that file only during pass-1 of the assembly. This option is useful when you assemble a source program together with a prefix file that contains only macro definitions, since these do not need to be redefined in pass-2 of the assembly. The following command assembles a prefix file and a source file together, producing files PROG1.OBJ, PROG1.LST, and PROG1.SAV.

```
.EXECUTE/NORUN/MACRO PREFIX/PASS:1+PROG1/LIST/OBJECT/EXECUTE
```

/RECORD:length – Use this option with FORTRAN to override the default record length of 132 characters for ASCII sequential formatted input and output. The meaningful range for length is from 4 to 4095.

/RUN – Use this option to initiate execution of your program if there are no errors in the compilation or the link. This is the default operation.

/NORUN – Use this option to suppress execution of your program. The system performs only the compilation and the link.

/SHOW[:value] – Use this option with FORTRAN to control FORTRAN listing format. The argument, value, represents a code that indicates which listings the compiler is to produce. Table 4-5 summarizes the codes and their meaning.

Use this option with MACRO to specify any MACRO .LIST directive. Table 4-12 summarizes the valid arguments and their meaning. Section 6.1.1, .LIST and .NLIST Directives, of the *PDP-11 MACRO Language Reference Manual* explains how to use these directives.

/NOSHOW:value – Use this option with MACRO to specify any MACRO .NLIST directive. Table 4-12 summarizes the valid arguments and their meaning. Section 6.1.1, .LIST and .NLIST Directives, of the *PDP-11 MACRO Language Reference Manual* explains how to use these directives.

/STATISTICS – Use this option with FORTRAN to include in the listing compilation statistics, such as amount of memory used, amount of time elapsed, and length of the symbol table.

/SWAP – Use this option with FORTRAN to permit the USR (user service routine) to swap over the FORTRAN program in memory. This is the default operation.

/NOSWAP – Use this option with FORTRAN to keep the USR resident during execution of a FORTRAN program. This may be necessary if the FORTRAN program uses some of the RT-11 system subroutine library calls (see Chapter 4 of the *RT-11 Advanced Programmer's Guide*). If the program frequently updates or creates a large number of different files, making the USR resident can improve program execution. However, the penalty for making the USR resident is 2K words of memory.

/UNITS:n – Use this option with FORTRAN to override the default number of logical units (6) to be open at one time. The maximum value you can specify for n is 16.

/VECTORS – This option directs FORTRAN to use tables to access multidimensional arrays. This is the default mode of operation.

/NOVECTORS – This option directs FORTRAN to use multiplication operations to access multidimensional arrays.

/WARNINGS – Use this option to include warning messages in DIBOL or FORTRAN compiler diagnostic error messages. These messages call certain conditions to your attention, but do not interfere with the compilation. This is the default operation for DIBOL.

/NOWARNINGS – Use this option with DIBOL to suppress warning messages during compilation. These messages are for your information only; they do not affect the compilation. This is the default operation for FORTRAN.

/WIDE – Use this option with /MAP to produce a wide load map listing. Normally, the listing is wide enough for three GLOBAL VALUE columns, which is suitable for paper with 72 or 80 columns. The /WIDE option produces a listing that is six GLOBAL VALUE columns wide, which is ideal for a 132-column page.

The FOCAL command invokes the FOCAL language interpreter.

```
FOCAL
```

FOCAL has its own command language. Therefore, the FOCAL command accepts no options and no file specifications.

The FORTRAN command invokes the FORTRAN IV compiler to compile one or more source programs.

```
FORTRAN  /LIST[:filespec] [/ALLOCATE:size]            (SP) filespecs
         /[NO] OBJECT[:filespec] [/ALLOCATE:size]

         /CODE:type
         /DIAGNOSE
         /EXTEND
         /HEADER
         /I4
         /[NO] LINENUMBERS
         /ONDEBUG
         /[NO] OPTIMIZE[:type]
         /RECORD:length
         /SHOW[:value]
         /STATISTICS
         /[NO] SWAP
         /UNITS:n
         /[NO] VECTORS
         /WARNINGS
```

In the command syntax illustrated above, filespecs represents one or more files to be included in the compilation. If you omit a file type for an input file, the system assumes .FOR. Output default file types are .LST for listing files and .OBJ for object files. To compile multiple source files into a single object file, separate the files by plus (+) signs in the command line. Unless you specify otherwise, the system creates an object file with the same name as the first input file and gives it an .OBJ file type. To compile multiple files in independent compilations, separate the files by commas (,) in the command line. This generates a corresponding object file for each set of input files.

Language options are position dependent. That is, they have different meanings depending on where you place them in the command line. Options that qualify a command name apply across the entire command string. Options that follow a file specification apply only to the file (or group of files separated by plus signs) that they follow in the command string. You can enter the FORTRAN command as one line, or you can rely on the system to prompt you for information. The FORTRAN command prompt is Files? for the input specification.

The *RT-11/RSTS/E FORTRAN IV User's Guide* contains more detailed information about using FORTRAN. The following sections describe the options you can use with the FORTRAN command.

/ALLOCATE:size – Use this option with /LIST or /OBJECT to reserve space on the device for the output file. The argument, size, represents the number of blocks of space to allocate. The meaningful range for this value is from 1 to 32767. A value of -1 is a special case that creates the largest file possible on the device.

/CODE:type – Use this option to produce object code that is designed for a particular hardware configuration. The argument, type, represents a three-letter abbreviation for the type of code to produce. The legal values are the following: EAE, EIS, FIS, and THR. See Section 1.1.1 of the *RT-11/RSTS/E FORTRAN IV User's Guide* for a complete description of the types of code and their functions.

/DIAGNOSE – Use this option to help analyze an internal compiler error. /DIAGNOSE expands the crash dump information to include internal compiler tables and buffers. Submit the diagnostic printout to DIGITAL with an SPR form. The information in the listing can help the DIGITAL programmers locate the compiler error and correct it.

/EXTEND – Use this option to change the right margin for source input lines from column 72 to column 80.

/HEADER – This option includes in the printout a list of options that are currently in effect.

/I4 – Use this option to allocate two words for the default integer data type (FORTRAN uses one-word integers) so that it takes the same physical space as real variables.

/LINENUMBERS – Use this option to include internal sequence numbers in the executable program. These are especially useful in debugging a FORTRAN program. They identify the FORTRAN statements that cause run-time diagnostic error messages. This is the default operation.

/NOLINENUMBERS – This option suppresses the generation of internal sequence numbers in the executable program. This produces a smaller program and optimizes execution speed. Use this option to compile only those programs that are already debugged; otherwise the line numbers in FORTRAN error messages are replaced by question marks and the messages are difficult to interpret.

/LIST[:filespec] – You must specify this option to produce a FORTRAN compilation listing. The /LIST option has different meanings depending on where you place it in the command line.

If you specify /LIST without a file specification in the list of options that immediately follows the command name, the FORTRAN compiler generates a listing that prints on the line printer. If you follow /LIST with a device name, the system creates a listing file on that device. If the device is a file-structured device, the system stores the listing file on that device, assigning it the same name as the input file with a .LST file type. The following command produces a listing on the terminal.

```
.FORTRAN/LIST:TT: A
```

The next command creates a listing file called A.LST on RK3:.

```
.FORTRAN/LIST:RK3: A
```

If the /LIST option contains a name and file type to override the default of .LST, the system generates a listing file with that name. The following command, for example, compiles A.FOR and B.FOR together, producing files A.OBJ and FILE1.OUT on device DK:.

```
.FORTRAN/LIST:FILE1.OUT A+B
```

You cannot use a command line like the next one. In this example, the second listing file would replace the first one and, therefore, cause an error.

```
.FORTRAN/LIST:FILE2 A,B
```

Another way to specify /LIST is to type it after the file specification to which it applies. To produce a listing file with the same name as a particular input file, you can use a command similar to this one:

```
.FORTRAN A+B/LIST:RK3:
```

The above command compiles A.FOR and B.FOR together, producing files DK:A.OBJ and RK3:B.LST. If you specify a file name on a /LIST option following a file specification in the command line, it has the same meaning as when it follows the command. The following two commands have the same results.

```
.FORTRAN A/LIST:B

.FORTRAN/LIST:B A
```

Both the above commands generate as output files A.OBJ and B.LST.

Remember that file options apply only to the file (or group of files that are separated by plus signs) that they follow in the command string. For example:

```
    .FORTRAN A/LIST,B
```

This command compiles A.FOR, producing A.OBJ and A.LST. It also compiles B.FOR, producing B.OBJ. However, it does not produce any listing file for the compilation of B.FOR.

/OBJECT[:filespec] – Use this option to specify a file name or device for the object file. Because FORTRAN creates object files by default, the following two commands have the same meaning.

```
    .FORTRAN A

    .FORTRAN/OBJECT A
```

Both commands compile A.FOR and produce A.OBJ as output. The /OBJECT option functions like the /LIST option; it can be either a command or a file qualifier.

As a command option, /OBJECT applies across the entire command string. The following command, for example, compiles A.FOR and B.FOR separately, creating object files A.OBJ and B.OBJ on RK1:.

```
    .FORTRAN/OBJECT:RK1: A,B
```

Use /OBJECT as a file option to create an object file with a specific name or destination. The following command compiles A.FOR and B.FOR together, creating files B.LST and B.OBJ.

```
    .FORTRAN A+B/LIST/OBJECT
```

/NOOBJECT – Use this option to suppress creation of an object file. As a command option, /NOOBJECT suppresses all object files; as a file option, it suppresses only the object file produced by the related input files. In this command, for example, the system compiles A.FOR and B.FOR together, producing files A.OBJ and B.LST. It also compiles C.FOR and produces C.LST, but does not produce C.OBJ.

```
    .FORTRAN A+B/LIST,C/NOOBJECT/LIST
```

/ONDEBUG – Use this option to include debug lines (those that have a D in column one) in the compilation. You do not, therefore, have to edit the file to include these lines in the compilation or to logically remove them. This option is useful in debugging a program. You can include messages, flags, and conditional branches to help you trace program execution and find an error.

/OPTIMIZE:type – Use this option to enable certain options that optimize object code for various conditions. The argument, type, represents the three-letter code for the type of optimization to enable. Table 4-4 summarizes the codes and their meanings.

Table 4-4 Optimization Codes

Code	Meaning
BND	Global register bindings for inline code generation
CSE	Common subexpression elimination
SPD	Optimization for speed of execution as opposed to minimal program size
STR	Strength reduction optimization

/NOOPTIMIZE:type – Use this option to disable certain options that optimize object code for various conditions. The argument, type, represents the three-letter code for the type of optimization to disable. Table 4-4 summarizes the codes and their meanings.

/RECORD:length – Use this option to override the default record length of 132 characters for ASCII sequential formatted input and output. The meaningful range for length is from 4 to 4095.

/SHOW[:value] – Use this option to control FORTRAN listing output. The argument, value, represents a code that indicates which listings the compiler is to produce. Table 4-5 summarizes the codes and their meaning. You can combine options by specifying the sum of their numeric codes. For example:

```
/SHOW:7
                  or
/SHOW:ALL
```

The two options shown above have the same meaning. If you specify no code, the default value is 3, a combination of SRC and MAP.

Table 4-5 FORTRAN Listing Codes

Code	Meaning
0	Lists diagnostics only
1 or SRC	Lists source program and diagnostics
2 or MAP	Lists storage map and diagnostics
3	Lists diagnostics, source program, and storage map
4 or COD	Lists generated code and diagnostics
7 or ALL	Lists diagnostics, source program, storage map, and generated code

/STATISTICS – Use this option to include compilation statistics in the listing, such as amount of memory used, amount of time elapsed, and length of the symbol table.

/SWAP – Use this option to permit the USR (user service routine) to swap over the FORTRAN program in memory. This is the default operation.

/NOSWAP – This option keeps the USR resident during execution of a FORTRAN program. This may be necessary if the FORTRAN program uses some of the RT-11 System Subroutine Library calls (see Chapter 4 of the *RT-11 Advanced Programmer's Guide*). If the program frequently updates or creates a large number of different files, making the USR resident can improve program execution. However, the penalty for making the USR resident is 2K words of memory.

/UNITS:n – Use this option to override the default number of logical units (6) to be open at one time. The maximum value you can specify for n is 16.

/VECTORS – This option directs FORTRAN to use tables to access multidimensional arrays. This is the default mode of operation.

/NOVECTORS – This option directs FORTRAN to use multiplication operations to access multidimensional arrays.

/WARNINGS – Use this option to include warning messages in FORTRAN compiler diagnostic error messages. These messages call certain conditions to your attention, but do not interfere with the compilation. A warning message prints, for example, if you change an index within a DO loop, or if you specify a variable name longer than six characters.

The FRUN command initiates foreground jobs.

```
FRUN (SP) filespec ┌/N:n┐
                   │/P  │
                   └/T:n┘
```

In the command syntax illustrated above, filespec represents the program to execute. Because this command runs a foreground job, it is valid for the FB and XM monitors only.

If another foreground job is active when you issue the FRUN command, an error message prints on the terminal. You can run only one foreground job at a time. If a terminated foreground job is occupying memory, the system reclaims that region for your program. Then, if the system finds your program and if your program fits in the available memory, execution begins.

The following sections describe the options you can use with FRUN. Note that the option must follow the file specification in the command line.

/N:n – Use this option to reserve space in memory over the actual program size. The argument, n, represents the number of words of memory to allocate. You must use this option to execute a FORTRAN foreground job.

/P – Use this option to help you debug a program. When you type the carriage return at the end of the command string, the system prints the load address of your program and waits. You can examine or modify the program (by using ODT, described in Chapter 16) before starting execution. You must use the RESUME command to restart the foreground job. The following command loads the program DEMOSP.REL, prints the load address, and waits for a RESUME command to begin execution.

```
.FRUN DEMOSP/P
Loaded at 127276
.RESUME
```

/T:n – Use this option to assign a terminal to interact with the foreground job. The argument, n, represents a terminal logical unit number. The default value is 0, which represents the original console terminal. By assigning a different terminal to interact with the foreground job, you eliminate the need for the foreground and background jobs to share the console terminal. Note that the original console terminal still interacts with the background job and with the keyboard monitor, unless you use the SET TT: CONSOL command to change this.

The GET command loads a memory image file into memory.

```
GET (SP) filespec
```

In the command syntax shown above, filespec represents the memory image file to be loaded. The default file type is .SAV. Note that magtape and cassette are not block-replaceable devices and, therefore, are not permitted with the GET command. Use the GET command for a background job only. You cannot use GET on a virtual program that executes under the XM monitor. The GET command is useful when you need to modify or debug a program. You can use GET with the Base, Deposit, Examine, and START commands to test changes. Use the SAVE command to make these changes permanent. You can combine programs by issuing multiple GET commands, as the following example shows. This example loads a program, DEMOSP.SAV, loads ODT.SAV (on-line debugging technique, described in Chapter 16), and starts the program using the address of ODT's entry point, O.ODT.

```
.GET DEMOSP

.GET ODT

.START

ODT V01.04
*
```

If more than one program requires the same locations in memory, the program you load later overlays the previous program. Note that you cannot use GET to load overlay segments of a program; it can load only the root. If the file you need to GET resides on a device other than the system device, the system automatically loads that device handler into memory when you issue the GET command. This prevents problems from occurring if you use the START command and your program is overlaid.

The GT command enables or disables the VT11 or VS60 display hardware.

```
GT (SP) ⎧ OFF          ⎫
        ⎨              ⎬
        ⎪ ON ⎡/L:n⎤    ⎪
        ⎩    ⎣/T:n⎦    ⎭
```

When you issue the GT OFF command, you disable the display hardware. The printing console terminal then becomes the device that transmits your commands to the system.

When you issue the GT ON command, the display screen replaces the printing console terminal. The display screen offers some advantages over the printing terminal: 1) it is quieter than a printing terminal, 2) it is faster than a printing terminal, 3) it does not require a supply of paper, and 4) it is the device for which the text editor's immediate mode is intended. The display screen can speed up the editing process (see Chapter 5 for information on how to use the text editor). You can use CTRL/A, CTRL/S, CTRL/E, and CTRL/Q to control scrolling. These commands are explained in Section 3.6. Note that RT-11 does not permit you to use display hardware (with GT ON) in an 8K configuration. You cannot issue GT ON when a foreground job is active; this causes the system to print an error message. Issue the GT ON command before you begin execution of the foreground job. ODT (on-line debugging technique, described in Chapter 16) is the only system program that cannot use the display screen. Its output always appears on the console terminal.

Table 4-6 Display Screen Values

Screen Size	Lines	Top Position
12 inch	1-31	1-744
17 inch (or larger)	1-40	1-1000

The following options let you control the number of lines that appear on the screen and position the first line vertically.

/L:n – Use this option to change the number of lines of text that display on the screen. Table 4-6 shows the valid range for the argument, **n**, in decimal. If you do not use this option, the system determines the screen size and automatically assigns the largest valid value.

/T:n – Use this option to change the top position of the scroll display. Table 4-6 shows the valid range for the argument, n, in decimal. If you do not use this option, the system determines the screen size and automatically assigns the largest valid value.

The following command enables the display screen.

```
.GT ON
```

The next command disables the display screen.

```
.GT OFF
```

The HELP command lists useful information.

```
HELP [ { /PRINTER  } ] [ (SP) topic[ (SP) subtopic[:item] ] ]
       { /TERMINAL }
```

In the command syntax shown above, topic represents a specific subject about which you need information. In the help file supplied with RT-11, the topics are the keyboard monitor commands. The subtopic represents a specific category within a topic. In the RT-11 help file, the subtopics are syntax, semantics, options, and examples. The item represents one member of the subtopic group. You can specify more than one item in the command line if you separate the items by colons (:).

The HELP command permits you to access the file HELP.TXT. The help file distributed with RT-11 contains information about the keyboard monitor commands and how to use them. However, the concept of the help file is a general one. That is, you can create your own help file to supply quick reference material on any subject. Structure your HELP.TXT file in the same format as the standard RT-11 HELP.TXT. Note that the HELP command reads the file that is specifically named HELP.TXT. There are only two options you can use with the HELP command. They are /PRINTER and /TERMINAL.

/PRINTER – Use this option to list helpful information on the line printer.

/TERMINAL – This option lists helpful information on the console terminal. This is the default operation.

The following examples all make use of the standard RT-11 help file.

The following command lists all the topics for which assistance is available.

```
.HELP *

HELP       Lists helpful information
APL        Invokes the APL language interpreter
ASSIGN     Associates a logical device name with a physical device
BASIC      Invokes the BASIC language interpreter
.
.
.
```

The next command lists all the information about the DATE command.

```
.HELP DATE

DATE       Sets or displays the current system date

  SYNTAX
           DATE[ dd-mmm-yy]

  SEMANTICS
           All numeric values are decimal;  mmm  represents  the  first
           three characters of the name of the month.
```

```
   OPTIONS
          None

   EXAMPLES
          DATE 12-MAY-77
```

The next command lists all the options that are valid with the DIRECTORY command.

```
 .HELP DIRECTORY OPTIONS

   OPTIONS
     ALLOCATE:size
         Use with /OUTPUT to reserve space for the output listing file
   ALPHABETIZE
         Sorts the directory in alphabetical order  by  file  name  and
         type
   .
   .
   .
```

The last command lists information about the /BRIEF option for the DIRECTORY command.

```
 .HELP DIRECTORY OPTIONS:BRIEF

     BRIEF
         Lists only file names and file types of files;  same as /FAST
```

Use the INITIALIZE command to clear and initialize a device directory.

```
INITIALIZE ┌ /DOS[/[NO]QUERY]                ┐ (SP) device
           │                                 │
           │ /FILE:filespec                  │
           │                                 │
           │ /INTERCHANGE[/[NO]QUERY]        │
           │                                 │
           │ /[NO]QUERY                      │
           │ /VOLUMEID[:ONLY]                │
           │ /SEGMENTS:n                     │
           │ { /REPLACE[:RETAIN] }           │
           └ { /BADBLOCKS        }           ┘
```

In the command syntax illustrated above, device represents the device you need to initialize. The initialize operation must always be the first operation you perform on a new device after you receive it from the manufacturer. This procedure destroys any data that may already exist on a device. After you use the INITIALIZE command, there are no files in the directory. If you use the INITIALIZE command with no options, the system simply initializes the device directory. You can enter the INITIALIZE command as one line, or you can rely on the system to prompt you for the name of the device with Device?. The following sections describe the options you can use with INITIALIZE and give some examples of their use.

/BADBLOCKS – Use this option to scan a device (disk or DECtape) for bad blocks and write .BAD files over them. For each bad block the system encounters on the device, it creates a file called FILE.BAD to cover it. After the device is initialized and the scan completed, the directory consists only of FILE.BAD entries that cover the bad blocks. This procedure ensures that the system will not attempt to access these bad blocks during routine operations. If the system finds a bad block in either the boot block or the device directory, it prints an error message and the device is not usable. The following command initializes device RK1: and scans for bad blocks.

```
.INITIALIZE/BADBLOCKS RK1:

RK1:/Init are you sure?Y
```

/DOS – Use this option to initialize a DECtape for DOS-11 format.

/FILE:filespec – Use this option to initialize a magtape and create a bootable tape. For filespec, substitute dev:MBOOT.BOT. This file is distributed with RT-11 for this purpose only. Consult the *RT-11 System Generation Manual* for more information. The following example creates a bootable magtape:

```
.INITIALIZE/FILE:MBOOT.BOT  MT0:
```

/INTERCHANGE – Use this option to initialize a diskette for interchange (proposed ANSI standard) format. The following example initializes DX1: in interchange format.

```
.INITIALIZE/INTERCHANGE DX1:
DX1:/Z ARE YOU SURE?  Y
```

/QUERY – This option prompts you for confirmation before it initializes a device. Respond by typing a Y followed by a carriage return to initiate execution of the command. The system interprets a response beginning with any other character to mean NO. /QUERY is the default operation.

/NOQUERY – Use this option to suppress the confirmation message that the system prints before it proceeds with the initialization.

/REPLACE[:RETAIN] – Use the /REPLACE option to scan the disk for bad blocks when you initialize an RK06. If the system finds any bad blocks, it creates a replacement table so that routine operations access good blocks instead of bad ones. Thus, the disk appears to consist of only good blocks. Note, though, that accessing this replacement table slows response time for routine input and output transactions. If you use :RETAIN with /REPLACE, the system initializes the RK06 but does not create a replacement table for bad blocks. Instead, it uses the replacement table that is already on the device as a result of a previous initialization. This procedure allows the initialization to proceed faster.

/SEGMENTS:n – Use this option if you need to initialize a disk and change the number of directory segments. The number of segments in the directory determines the number of files that can be sorted on a device. The system allows a maximum of 72 files per directory segment, and 31 directory segments per device. The argument, n, represents the number of directory segments you need to create. The valid range for n is from 1 to 31 (decimal). Table 4-7 shows the default values of n for standard RT-11 devices.

Table 4-7 Default Directory Sizes

Device	Size (decimal) of Directory in Segments
RK05	16
DT	4
RF	4
DS	4
DP	31
DX	4
RK06	31

/VOLUMEID[:ONLY] – Use this option to write a volume identification on a device when you initialize it. This identification consists of a volume ID (up to 12 characters long for a block-replaceable device, up to 6 characters long for magtape) and an owner name (up to 12 characters long for a block-replaceable device, up to 10 characters long for magtape). The following example initializes device RK1: and writes a volume identification on it.

```
.INITIALIZE/VOLUMEID RK1:

RK1:/Init are you sure?Y

 VOL ID?BACKUP2

 OWNER NAME?ENGINEERING
```

Use /VOLUMEID:ONLY to write a new volume identification on a device without reinitializing the device.

The INSTALL command installs the device you specify into the system.

```
INSTALL (SP) device[ ,... device]
```

In the command syntax shown above, device represents the name of the device to be installed. The INSTALL command accepts no options. The INSTALL command allows you to install into the system tables a device that was not originally built into the system. (A device handler must exist in the system tables before you can use that device.) The device occupies the first available device slot. Using the INSTALL command does not change the monitor disk image; it only modifies the system tables of the monitor that is currently in memory.

You can enter the command on one line, or you can rely on the system to prompt you for information. The INSTALL command prompt is Device?.

When you specify a device name, the system searches the system device for the corresponding device handler file. For SJ and FB systems, if LP: is to be installed, the INSTALL command searches for the file SY:LP.SYS. For XM systems, INSTALL searches for SY:LPX.SYS. The INSTALL command does not allow a device handler built for a different configuration of the system to be installed in a given system. For example, you cannot install an error logging handler if your currently running monitor is not designed for error logging. Note that you cannot install the following device names: FG (with FB or XM monitor only), and BA.

To permanently install a device, include the INSTALL command in the standard system startup indirect command file. This file is invoked as an indirect file automatically when you boot the system. The INSTALL command also allows you to configure a special system for a single session without having to reconfigure to get back to the standard device configuration. Rebooting the system restores the original device configuration. Note that if there are no free device slots (use the SHOW DEVICES command to determine this), you must remove an existing device (with the REMOVE command) before you can install a new device.

The following command installs the card reader into the system tables from the file CR.SYS. Note that the colon (:) that follows the device handler name is optional.

```
.INSTALL CR:
```

The next example installs the line printer, the card reader, and DECtape.

```
.INSTALL LP:,CR:,DT:
```

The LIBRARY command lets you create, update, modify, list, and maintain library files.

```
LIBRARY[ /LIST[:filespec] [/ALLOCATE:size]           ] (SP) library[(SP) filespecs[{/REPLACE}]]
       [ /[NO] OBJECT[:filespec] [/ALLOCATE:size]    ]              [             [{/UPDATE }]]
       [                                             ]
       [ [{/CREATE }]                                ]
       [ [{/EXTRACT}]                                ]
       [ [{/INSERT }]                                ]
       [ [{/MACRO  }]                                ]
       [ /DELETE                                     ]
       [ /PROMPT                                     ]
       [ /REMOVE                                     ]
```

In the command syntax illustrated above, library represents the library file name and filespecs represents the input module file names. Separate the library file specification from the module file specifications by a space. Separate the module file specifications by commas. The system uses .LST as the default file type for library directory listing files. It also uses .OBJ as the default file type for object libraries and object input files, and it uses .MAC for macro libraries and macro input files. The default operation, if you do not specify an option, is /INSERT. If you do not specify a library file in the command line, the system prompts you with Library?. If you specify /CREATE, /INSERT, or /MACRO and omit the module file specification, the system prompts you with Files?. If you specify /EXTRACT, the system prompts you with File?. Note that no other options cause the File? or Files? prompts.

The LIBRARY command can perform all the functions listed above on object library files. It can also create macro library files for use with the MACRO-11 assembler. A library file is a direct access file (a file that has a directory) that contains one or more modules of the same module type. The system organizes the library files so that the linker and MACRO-11 assembler can access them rapidly. Each object library is a file that contains a library header, library directory, and one or more object modules. The object modules in a library file can be routines that are repeatedly used in a program, routines that are used by more than one program, or routines that are related and simply gathered together for convenience. The contents of the library file are determined by your needs. An example of a typical object library file is the default system library, SYSLIB.OBJ, used by the linker. An example of a macro library file is SYSMAC.SML.

You access object modules in a library file from another program by making calls or references to their global symbols; you link the object modules with the program that uses them by using the LINK command to produce a single executable module. Each input file for an object library consists of one or more object modules, and is stored on a device under a specific file name and file type. Once you insert an object module into a library file, you no longer reference the module by the file name of which it was a part. Reference it now by its individual module name. For example, the input file FORT.OBJ may exist on DT2: and can contain an object module called ABC. Once you insert the module into a library, reference only ABC and not FORT.OBJ.

The input files normally do not contain main programs but only subprograms, functions and subroutines. The library files must never contain a FORTRAN "BLOCK DATA" subprogram: there is no undefined global symbol to cause the linker to load it automatically.

The following sections describe the LIBRARY command options and explain how to use them. The last section under this command describes the LIBRARY prompting sequence and order of execution for commands that combine two or more LIBRARY options. Chapter 12 contains more detailed information on object and macro libraries. The following sections describe the options available with the LIBRARY command.

/ALLOCATE:size – Use this option with /LIST or /OBJECT to reserve space on the device for the output file. The argument, size, represents the number of blocks of space to allocate. The meaningful range for this value is from 1 to 32767. A value of -1 is a special case that creates the largest file possible on the device.

/CREATE – Use this option to create an object library. Specify a library name followed by the file specifications for the modules that are to be included in that library. The following command, for example, creates a library called NEWLIB.OBJ from the modules contained in files FIRST.OBJ and SECOND.OBJ.

```
    .LIBRARY/CREATE NEWLIB FIRST,SECOND
```

/DELETE – Use this option to delete an object module and all its associated global symbols from the library. Specify the library name in the command line. The system prompts you for the names of the modules to delete. The prompt is:

```
    Module name?
```

Respond with the name of a module. (Be sure to specify a module name and not a global name.) Follow each module name with a carriage return. Enter a carriage return on a line by itself to terminate the list of module names. The following example deletes modules SGN and TAN from the library called NEWLIB.OBJ.

```
    .LIBRARY/DELETE NEWLIB
    Module name? SGN
    Module name? TAN
    Module name?
```

/EXTRACT – Use this option to extract an object module from a library and store it in a file with the same name as the module and a file type of .OBJ. You cannot combine this option with any other option. The system prompts you for the name of the object module to be extracted. The prompt is:

```
    Global?
```

If you specify a global name, the system extracts the entire module of which that global is a part. Follow each global name with a carriage return. Enter a carriage return on a line by itself to terminate the list of global symbols. The following example, which also shows the system prompts, extracts the module ATAN from the library called NEWLIB.OBJ, storing it in file ATAN.OBJ on DX1:.

```
    .LIBRARY/EXTRACT
    Library? NEWLIB
    File    ? DX1:ATAN
    Global  ? ATAN
    Global  ?
```

/INSERT – Use this option to insert an object module into an existing library. Although you can insert two or more object modules having the same name, this practice is not recommended because of the difficulty involved in replacing or updating these modules. Note that /INSERT is the default operation. If you do not specify any option, insertion takes place. The following example inserts the modules contained in the files THIRD.OBJ and FOURTH.OBJ into the library called OLDLIB.OBJ.

```
    .LIBRARY/INSERT OLDLIB THIRD,FOURTH
```

/LIST[:filespec] – Use this option to obtain a directory listing of an object library. The following example obtains a directory listing of OLDLIB.OBJ on the terminal (the line printer is the default device).

```
    .LIBRARY/LIST:TT:  OLDLIB
```

The directory listing prints global symbol names. A plus sign (+) in the module column indicates a continued line. See Section 12.2.7 for a procedure to include module names in the directory listing.

You can also use /LIST with other options (except /MACRO) to obtain a directory listing of an object library after you create or modify it. The following command, for example, inserts the modules contained in the files THIRD.OBJ and FOURTH.OBJ into the library called OLDLIB.OBJ, and prints a directory listing of the library on the terminal.

```
.LIBRARY/INSERT/LIST:TT:   OLDLIB THIRD,FOURTH
```

You cannot obtain a directory listing of a macro library (see /MACRO).

/MACRO – Use this option to create a macro library. Note that this is the only valid function for a macro library. You can create a macro library, but you cannot list or modify it. To update a macro library, simply edit the ASCII text file and then reprocess the file with the LIBRARY/MACRO command. The following example creates a macro library called NEWLIB.MAC from the ASCII input file SYSMAC.MAC.

```
.LIBRARY/MACRO NEWLIB SYSMAC
```

/OBJECT[:filespec] – The system creates object library files by default as a result of executing a LIBRARY command. When you modify an existing library, the system actually makes the changes to the library you specify, thus creating a new, updated library that it stores under the same name as the original library. Use this option to give a new name to the updated library file and preserve the original library. The following example creates a library called NEWLIB.OBJ, which consists of the library OLDLIB.OBJ plus the modules that are contained in files THIRD.OBJ and FOURTH.OBJ.

```
.LIBRARY/INSERT/OBJECT:NEWLIB OLDLIB THIRD,FOURTH
```

/NOOBJECT – Use this option to suppress the creation of a new object library as a result of a LIBRARY command.

/PROMPT – Use this option to specify more than one line of input file specifications in a LIBRARY command. This option is valid with all other library functions except the /EXTRACT option. You must specify // as the last input in order to properly terminate the input list. The following example creates a macro library called MACLIB.MAC from seven input files.

```
.LIBRARY/MACRO/PROMPT MACLIB A,B,C,D
*E,F,G
*//
```

/REMOVE – This option permits you to delete a specific global symbol from a library file's directory. Since globals are only deleted from the directory (and not from the object module itself), all the globals that were previously deleted are restored whenever you update that library, unless you use /REMOVE again to delete them. This feature lets you recover a library if you have inadvertently deleted the wrong global. The system prompts you for the names of the global symbols to remove. The prompt is:

```
Global?
```

Respond with the name of a global symbol to be removed. Follow each global symbol with a carriage return. Enter a carriage return on a line by itself to terminate the list of global symbols. The following example deletes the globals GA, GB, GC, and GD from the library OLDLIB.OBJ.

```
.LIBRARY/REMOVE OLDLIB
Global? GA
Global? GB
Global? GC
Global? GD
Global?
```

/REPLACE – Use this option to replace modules in an existing object library with modules of the same name contained in the files you specify. If an old module does not exist with the same name as the input module you specify, or if you specify /REPLACE with a library file name, the system prints an error message and ignores the command. The following example replaces a module called SQRT in the library MATHLB.OBJ with a new module, also called SQRT, from the file called MFUNCT.OBJ.

```
    .LIBRARY MATHLB MFUNCT/REPLACE
```

Note that the /REPLACE option must follow each file specification that contains a module to be inserted into the library.

/UPDATE – This option combines the functions of /INSERT and /REPLACE. Specify it after each file specification to which it applies. If the modules in the input file already exist in the library, the system replaces those library modules. If the modules in the input file do not exist in the library, the system inserts them. The following example updates the library OLDLIB.OBJ.

```
    .LIBRARY OLDLIB FIRST/UPDATE,SECOND/UPDATE
```

You can combine the LIBRARY options with the exceptions of /EXTRACT and /MACRO, which you cannot combine with most of the other functions. Table 4-8 lists the sequence in which the system executes the LIBRARY options and prompts you for additional information.

Table 4-8 LIBRARY Execution and Prompting Sequence

Option	Prompt
/CREATE	
/DELETE	Module name?
/REMOVE	Global?
/UPDATE	
/REPLACE	
/INSERT	
/LIST	

The following example combines several options.

```
  LIBRARY/LIST:TT:/REMOVE/INSERT NEWLIB LIB2/REPLACE,LIB3
 Global? SQRT
 Global?
 RT-11 LIBRARIAN V03.05  FRI 15-JUL-77 00:08:37
 NEWLIB                  FRI 15-JUL-77 00:08:35

 MODULE          GLOBALS         GLOBALS         GLOBALS

                 COS             SIN
                 DATAN           DATAN2
                 ATAN            ATAN2
                 DCOS            DSIN
```

The command executes in the following sequence:

1. Removes global SQRT from NEWLIB
2. Replaces any duplicates of the modules in the file LIB2.OBJ
3. Inserts the modules in the file LIB3.OBJ
4. Lists the directory of NEWLIB.OBJ on the terminal.

The LINK command converts object modules produced by an RT-11 supported language processor into a format suitable for loading and execution.

```
LINK [ /[NO] EXECUTE[:filespec]                      ]  (SP) filespecs
     [ /MAP[:filespec] [/ALLOCATE:size] [/WIDE]      ]
     [                                               ]
     [ { /LDA                        }               ]
     [ {                             }               ]
     [ { /FOREGROUND[:stacksize]     }               ]
     [ { /FILL:n                     }               ]
     [ {                             }               ]
     [ { /BOTTOM:n                   }               ]
     [ { /FILL:n                     }               ]
     [ { /RUN                        }               ]
     [ { /STACK[:n]                  }               ]
     [                                               ]
     [ /BOUNDARY:value                               ]
     [ /DEBUG[:filespec]                             ]
     [ /EXTEND:n                                     ]
     [ /INCLUDE                                      ]
     [ /LIBRARY:filespec                             ]
     [ /LINKLIBRARY:filespec                         ]
     [ /PROMPT                                       ]
     [ /ROUND:n                                      ]
     [ /SLOWLY                                       ]
     [ /TRANSFER[:n]                                 ]
```

The RT-11 system lets you separately assemble a main program and each of its subroutines without assigning an absolute load address at assembly time. The linker can then process the object modules of the main program and subroutines to relocate each object module and assign absolute addresses. It links the modules by correlating global symbols that are defined in one module and referenced in another, and it creates the initial control block for the linked program. The linker can also create an overlay structure (if you specify the /PROMPT option) and include the necessary run-time overlay handlers and tables. The linker searches libraries you specify to locate unresolved global symbols, and it automatically searches the default system library, SYSLIB.OBJ, to locate any remaining unresolved globals. Finally, the linker produces a load map (if you specify /MAP) that shows the layout of the executable module. Read Chapter 11 for a more detailed explanation of the RT-11 linker.

In the command syntax illustrated above, filespecs represents the object modules to be linked. Each input module should be stored on a random-access device (disk or DECtape); the output device for the load map file can be any RT-11 device. The output for an .LDA file (if you specify /LDA) can also be any RT-11 device, even those that are not block replaceable, such as paper tape.

The default file types are as follows:

Load Module	:	.SAV, .REL(/FOREGROUND), .LDA(/LDA)
Map Output	:	.MAP
Object Module	:	.OBJ

If you specify two or more files to be linked together, separate the files by commas. The system creates an executable file with the same name as the first file in the input list (unless you use /EXECUTE to change it).

The following sections describe the LINK command options and explain how to use them. The last section under this command describes the LINK prompting sequence for commands that combine two or more LINK options.

/ALLOCATE:size – Use this option with /MAP to reserve space on the device for the output file. The argument, size, represents the number of blocks of space to allocate. The meaningful range for this value is from 1 to 32767. A value of -1 is a special case that creates the largest file possible on the device.

/BOTTOM:n – Use this option to specify the lowest address to be used by the relocatable code in the load module. The argument, n, represents a six-digit unsigned even octal number. If you do not use this option, the linker positions the load module so that the lowest address is location 1000 (octal). This option is illegal for foreground links.

/BOUNDARY:value – Use the /BOUNDARY option to start a specific program section on a particular address boundary. The system generates a whole number multiple of the value you specify for the starting address of the program section. The argument, value, must be a power of 2. The system extends the size of the previous program section to accommodate the new starting address for the specific section. When you have entered the complete LINK command, the system prompts you for the name of the section whose starting address you need to modify. The prompt is:

```
    Boundary section?
```

Respond with the appropriate program section name. Terminate your response with a carriage return.

/DEBUG[:filespec] – Use this option to link ODT (on-line debugging technique, described in Chapter 16) with your program to help you debug it. If you supply the name of another debugging program, the system links the debugger you specify with your program. The system links the debugger low in memory relative to your program.

/EXECUTE[:filespec] – Use this option to specify a file name or device for the executable file. Because the LINK command creates executable files by default, the following two commands have the same meaning.

```
    .LINK MYPROG

    .LINK/EXECUTE MYPROG
```

Both commands link MYPROG.OBJ and produce MYPROG.SAV as a result. The /EXECUTE option has different meanings when it follows the command and when it follows the file specification. The following command creates an executable file called PROG1.SAV on device RK1:.

```
    .LINK/EXECUTE:RK1:  PROG1,PROG2
```

The next command creates an executable file called MYPROG.SAV on device DK:.

```
    .LINK RTN1,RTN2,MYPROG/EXECUTE
```

/NOEXECUTE – Use this option to suppress creation of an executable file.

/EXTEND:n – This option allows you to extend a program section to a specific octal value, n. The resultant program section size is equal to or greater than the value you specify, depending on the space the object code actually requires. When you have entered the complete LINK command, the system prompts you for the name of the program section you need to extend. The prompt is:

```
    Extend section?
```

Respond with the appropriate program section name. Terminate your response with a carriage return.

/FILL:n – Use this option to initialize unused locations in the load module and place a specific value in those locations. The argument, n, represents the octal value to place in the unused locations. Note that the linker automatically initializes unused locations in the load module to 0; use this option to place another value in those locations. This option can be useful in eliminating random results that occur when a program references uninitialized memory by mistake. It can also help you determine which locations have been modified by the program and which are left unchanged.

/FOREGROUND[:stacksize] – This option produces an executable file in relocatable (.REL) format for use as a foreground job under the FB or XM monitor. You cannot use .REL files in the single-job system. This option assigns the default file type .REL to the executable file. The argument, stacksize, represents the number of bytes of stack space to allocate for the foreground job. The value you supply is interpreted as an octal number; specify an even number. Follow n with a decimal point (n.) to represent a decimal number. The default value is 128 (decimal) bytes of stack space.

/INCLUDE – This option lets you take global symbols from any library and include them in the linked memory image. When you have entered the complete LINK command, the system prompts you for a list of global symbols to include in the load module. The prompt is:

```
    Library search?
```

Respond by typing the global symbols to be included in the load module. Type a carriage return after each global symbol. Type a carriage return on a line by itself to terminate the list. This provides a method for forcing modules (that are not called by other modules) to be loaded from the library.

/LDA – This option produces an executable file in LDA format. The LDA-format file can be output to any device, including those that are not block-replaceable, such as the paper tape punch or cassette. This option assigns the default file type .LDA to the executable file. This option is useful for files that you need to load with the Absolute Binary Loader.

/LIBRARY – This option is the same as /LINKLIBRARY. It is included for compatibility with other systems.

/LINKLIBRARY:filespec – You can use this option to include the library file you specify as an object module library in the linking operation. This option is not necessary because the system automatically recognizes library files in the linking operation; it is provided for compatibility with the EXECUTE command.

/MAP[:filespec] – You must specify this option to produce a load map listing. The /MAP option has different meanings depending on where you put it in the command line.

If you specify /MAP without a file specification in the list of options that immediately follows the command name, the system generates a listing that prints on the line printer. If you follow /MAP with a device name, the system creates a map file on that device. If the device is a file-structured device, the system stores the listing file on that device, assigning it the same name as the first input file with a .LST file type. The following command produces a load map on the terminal.

```
    .LINK/MAP:TT:   MYPROG
```

The next command creates a map listing file called MYPROG.LST on RK3:.

```
    .LINK/MAP:RK3:   MYPROG
```

If the /MAP option contains a name and file type to override the default of .LST, the system generates a listing with that name. The following command, for example, links PROG1 and PROG2, producing a map listing file called MAP.OUT on device DK:.

```
    .LINK/MAP:MAP.OUT PROG1,PROG2
```

Another way to specify /MAP is to type it after the file specification to which it applies. To link a file and produce a map listing file with the same name, use a command similar to this one.

```
    .LINK PROG1,PROG2/EXECUTE/MAP
```

The command shown above links PROG1 and PROG2, producing files PROG2.SAV and PROG2.MAP. If you specify a file name on a /MAP option following a file specification in the command line, it has the same meaning as when it follows the command.

/PROMPT – Use this option to enter additional lines of input. The system continues to accept lines of linker input until you enter two slashes (//). Chapter 11 describes the commands you can enter directly to the linker. The /PROMPT option also gives you a convenient way to create an overlaid program from an indirect file. The file HERB.COM contains these lines:

```
    A/PROMPT
    SUB1/O:1
    SUB2/O:1
    SUB3,SUB4/O:1
    //
```

The following command produces an executable file, DK:HERB.SAV, and a link map on the printer.

```
    .LINK/MAP @HERB
```

/ROUND:n – This option rounds up the section you specify so that the size of the root segment is a whole number multiple of the value, n, you supply. The argument, n, must be a power of 2. When you have entered the complete LINK command, the system prompts you for the name of the section that you need to round. The prompt is:

```
    Round section?
```

Respond with the appropriate program section name. Terminate your response with a carriage return.

/RUN – Use this option to initiate execution of the resultant .SAV file. This option is valid for background jobs only.

/SLOWLY – This option instructs the system to allow the largest possible memory area for the link symbol table at the expense of making the link process slower. Use this option only if an attempt to link a program failed because of symbol table overflow.

/STACK[:n] – This option lets you modify the stack address. This address, location 42, is the address that contains the value for the stack pointer. When your program executes, the stack pointer (SP) is automatically set to the contents of location 42. The argument, n, is an even, unsigned six-digit octal number that defines the stack address. When you have entered the complete LINK command, the system prints the following prompt message if you did not already specify a numeric value for n.

```
    Stack symbol?
```

Respond with the global symbol whose value is the stack address. You cannot specify a number at this point. Terminate your response with a carriage return. If you specify a nonexistent symbol, the system prints an error message. It then sets the stack address to 1000 (for memory image files) or to the bottom address if you used /BOTTOM.

/TRANSFER[:n] – The transfer address is the address at which a program starts when you initiate execution with R, RUN, or FRUN. The /TRANSFER option lets you specify the start address of the load module. The argument, n, is an even, unsigned six-digit octal number that defines the transfer address. When you have entered the complete LINK command, the system prints the following prompt message if you did not already specify a numeric value for n:

```
Transfer symbol?
```

Respond with the global symbol whose value is the transfer address. You cannot specify a number at this point. Terminate your response with a carriage return. If you specify a nonexistent symbol, an error message prints and the linker sets the transfer address to 1 so the system cannot execute the program. If the transfer address you specify is odd, the program does not execute after loading and control returns to the monitor.

/WIDE – Use this option with /MAP to produce a wide load map listing. Normally, the listing is wide enough for three GLOBAL VALUE columns, which is suitable for paper with 72 or 80 columns. The /WIDE option produces a listing that is six GLOBAL VALUE columns wide, which is ideal for a 132-column page.

This section describes the prompting sequence that occurs when you combine the LINK options. Table 4-9 lists the sequence in which the system prompts you for additional information.

Table 4-9 LINK Prompting Sequence

Option	Prompt
/TRANSFER	Transfer symbol?
/STACK	Stack symbol?
/EXTEND:n	Extend section?
/BOUNDARY:value	Boundary section?
/ROUND:n	Round section?
/INCLUDE	Library search?

If you combine any of the options listed in Table 4-9, the system prompts you for information in the sequence shown in the table. Note that the Library search? prompt is always last. This is the only prompt that accepts more than one line as a response. For all the prompts, terminate your response with a carriage return. Terminate your list of responses to the Library search? prompt by placing a carriage return on a line by itself. Note that if the command lines are in an indirect file and the system encounters an end-of-file before all the prompting information has been supplied, it prints the prompt messages on the terminal.

The LOAD command makes a device handler resident in memory for use with BATCH or foreground/background jobs.

```
LOAD (SP) device[=jobtype] [,... device[=jobtype]]
```

In the command syntax shown above, device represents the device handler to be made resident; jobtype, which can have the values B or F, assigns the device handler to the background or foreground job, respectively. The jobtype specification is invalid with the SJ monitor.

The LOAD command helps control system execution by bringing a device handler into memory and optionally allocating the device to a job. The system allocates memory for the handler as needed. Before you use a device in a foreground program with the FB monitor, or any device at all with the XM monitor, you must first load the device handler. A device can be owned exclusively by either the foreground or background job. (Note that BATCH, if running, is considered to be a background job under the FB and XM monitors.) This exclusivity prevents the input and output of two different jobs from being intermixed on the same non-file-structured device. In the following example, magtape belongs to the background job while DECtape is available for use by either the background or foreground job; the line printer is owned by the foreground job. All three handlers are made resident in memory.

```
.LOAD DT:,MT:=B,LP:=F
```

Different units of the same random-access device controller can be owned by different jobs. Thus, for example, DT1: can belong to the background job while DT5: can belong to the foreground job. If no ownership is indicated, the device is available for public use. To change ownership of a device, use another LOAD command. It is not necessary to first unload the device. For example, if the line printer has been loaded into memory and assigned to the foreground job as in the example above, the following command reassigns it to the background job without unloading the handler first.

```
.LOAD LP:=B
```

Note, however, that if you interrupt an operation that involves magtape or cassette, you must unload (with the UNLOAD command) then reload the appropriate device handler (MM, MT, or CT).

You cannot assign ownership of the system unit (the unit you bootstrapped) of a system device, and any attempt to do so is ignored. You can, however, assign ownership of other units of the same type as the system device. LOAD is valid for use with user-assigned names. For example:

```
.ASSIGN RK1: XY
.LOAD XY:=F
```

If you are using the diskette monitor, loading the necessary device handlers into memory can improve system performance since no handlers need to be loaded dynamically from the diskette. Use the SHOW DEVICES command to display on the terminal the status of device handler and device ownership.

The MACRO command invokes the MACRO assembler to assemble one or more source files.

```
MACRO ┌/LIST[:filespec] [/ALLOCATE:size]          ┐ (SP) filespecs ┌/LIBRARY┐
      │/[NO] OBJECT[:filespec] [/ALLOCATE:size]   │                └/PASS:1  ┘
      │                                           │
      │/CROSSREFERENCE[:type[...:type] ]          │
      │/DISABLE:value[...:value]                  │
      │/ENABLE:value[...:value]                   │
      │/[NO] SHOW[:value]                         │
      └                                           ┘
```

In the command syntax shown above, filespecs represents one or more files to be included in the assembly. If you omit a file type for an input file, the system assumes .MAC. Output default file types are .LST for listing files and .OBJ for object files.

To assemble multiple source files into a single object file, separate the files by plus (+) signs in the command line. Unless you specify otherwise, the system creates an object file with the same name as the first input file and gives it an .OBJ file type. To assemble multiple files in independent assemblies, separate the files by commas (,) in the command line. This generates a corresponding object file for each set of input files.

Language options are position dependent. That is, they have different meanings depending on where you place them in the command line. Options that qualify a command name apply across the entire command string. Options that follow a file specification apply only to the file (or group of files separated by plus signs) that they follow in the command string.

You can enter the MACRO command as one line, or you can rely on the system to prompt you for information. The MACRO command prompt is Files? for the input specification. The system prints on the terminal the number of errors MACRO detects during an assembly, as this printout shows:

```
.MACRO/CROSSREFERENCE PROG1+PROG2/LIST/OBJECT
ERRORS DETECTED:  0
```

Chapter 10 and the *PDP-11 MACRO Language Reference Manual* contain more detailed information about using MACRO. The following sections describe the options you can use with the MACRO command.

/ALLOCATE:**size** – Use this option with /LIST or /OBJECT to reserve space on the device for the output file. The argument, size, represents the number of blocks of space to allocate. The meaningful range for this value is from 1 to 32767. A value of -1 is a special case that creates the largest file possible on the device.

/CROSSREFERENCE:type[. . . :**type**] – Use this option to generate a symbol cross-reference section in the listing. This information is useful for program maintenance and debugging. Note that the system does not generate a listing by default. You must also specify /LIST in the command line to get a cross-reference listing. The argument, type, represents a one-character code that indicates which sections of the cross-reference listing the assembler should include. Table 4-10 summarizes the valid arguments and their meanings.

/DISABLE:value[. . . :**value**] – Use this option to specify a MACRO .DSABL directive. See Section 6.2 of the *PDP-11 MACRO Language Reference Manual* for a description of the directive and a list of all legal values. Table 4-11 summarizes the arguments and their meaning.

Table 4-10 Cross-reference Sections

Argument	Section Type
S	User-defined symbols
R	Register symbols
M	Macro symbolic names
P	Permanent symbols (instructions, directives)
C	Control sections (.CSECT and .PSECT symbolic names)
E	Error codes
no argument	Equivalent to :S:M:E

Table 4-11 .DSABL and .ENABL Directive Summary

Argument	Default	Enables or Disables
ABS	disable	Absolute binary output
AMA	disable	Assembles all absolute addresses as relative addresses
CDR	disable	Treats source columns 73 and greater as comments
FPT	disable	Floating point truncation
GBL	disable	Treats undefined symbols as globals
LC	disable	Accepts lower case ASCII input
LSB	disable	Local symbol block
PNC	enable	Binary output
REG	enable	Mnemonic definitions of registers

/ENABLE:value[. . . :value] – Use this option to specify a MACRO .ENABL directive. See Section 6.2 of the *PDP-11 MACRO Language Reference Manual* for a description of the directive and a list of all legal values. Table 4-11 summarizes the arguments and their meaning.

/LIBRARY – This option identifies the file it qualifies as a library file; use it only after a macro library file specification in the command line. The MACRO assembler looks first to the library file or files you specify and then to the system library, SYSMAC.SML, to satisfy references (made with the .MCALL directive) from MACRO programs. In the example below, the command string includes two user libraries.

```
    .MACRO MYLIB1/LIBRARY+A+MYLIB2/LIBRARY+B
```

When MACRO assembles file A, it looks first to the library, MYLIB1.MAC, and then to SYSMAC.SML to satisfy .MCALL references. When it assembles file B, MACRO searches MYLIB2.MAC, MYLIB1.MAC, and then SYSMAC.SML, in that order, to satisfy references.

/LIST[:filespec] – You must specify this option to produce a MACRO assembly listing. The /LIST option has different meanings depending on where you place it in the command line.

If you specify /LIST without a file specification in the list of options that immediately follows the command name, the MACRO assembler generates a listing that prints on the line printer. If you follow /LIST, with a device name, the system creates a listing file on that device. If the device is a file-structured device, the system stores the listing file on that device, assigning it the same name as the input file with a .LST file type. The following command produces a listing on the terminal.

```
    .MACRO/LIST:TT: A
```

The next command creates a listing file called A.LST on RK3:.

```
    .MACRO/LIST:RK3: A
```

If the /LIST option contains a name and file type to override the default of .LST, the system generates a listing file with that name. The following command for example, assembles A.MAC and B.MAC together, producing files A.OBJ and FILE1.OUT on device DK:.

```
    .MACRO/LIST:FILE1.OUT A+B
```

You cannot use a command like the next one. In this example, the second listing file would replace the first one and, therefore, cause an error.

```
    .MACRO/LIST:FILE2 A,B
```

Another way to specify /LIST is to type it after the file specification to which it applies. To produce a listing file with the same name as a particular input file, you can use a command similar to this one:

```
    .MACRO A+B/LIST:RK3:
```

The above command assembles A.MAC and B.MAC, producing files DK:A.OBJ and RK3:B.LST. If you specify a file name on a /LIST option following a file specification in the command line, it has the same meaning as when it follows the command. The following two commands have the same results:

```
    .MACRO A/LIST:B

    .MACRO/LIST:B A
```

Both the above commands generate as output files A.OBJ and B.LST.

Remember that file options apply only to the file (or group of files that are separated by plus signs) they follow in the command string. For example:

```
    .MACRO A/LIST,B
```

This command assembles A.MAC, producing A.OBJ and A.LST. It also assembles B.MAC, producing B.OBJ. However, it does not produce any listing file for the assembly of B.MAC.

/OBJECT[:filespec] – Use this option to specify a file name or device for the object file. Because MACRO creates object files by default, the following two commands have the same meaning.

```
    .MACRO A

    .MACRO/OBJECT A
```

Both commands assemble A.MAC and produce A.OBJ as output. The /OBJECT option functions like the /LIST option; it can be either a command or a file qualifier.

As a command option, /OBJECT applies across the entire command string. The following command, for example, assembles A.MAC and B.MAC separately, creating object files A.OBJ and B.OBJ on RK1:.

```
    .MACRO/OBJECT:RK1: A,B
```

Use /OBJECT as a file option to create an object file with a specific name or destination. The following command assembles A.MAC and B.MAC together, creating files B.LST and B.OBJ.

```
    .MACRO A+B/LIST/OBJECT
```

/NOOBJECT – Use this option to suppress creation of an object file. As a command option, /NOOBJECT suppresses all object files; as a file option, it suppresses only the object file produced by the related input files. In this command, for example, the system assembles A.MAC and B.MAC together, producing files A.OBJ and B.LST. It also assembles C.MAC and produces C.LST, but does not produce C.OBJ.

```
    .MACRO A+B/LIST,C/NOOBJECT/LIST
```

/PASS:1 – Use this option on a prefix macro file to process that file during pass-1 of the assembly only. This option is useful when you assemble a source program together with a prefix file (one that contains only macro definitions), since these definitions do not need to be redefined in pass-2 of the assembly. The following command assembles a prefix file and a source file together, producing files PROG1.OBJ and PROG1.LST.

```
    .MACRO PREFIX.MAC/PASS:1+PROG1/LIST/OBJECT
```

/SHOW:value – Use this option to specify any MACRO .LIST directive. Section 6.1.1 of the *PDP-11 MACRO Language Reference Manual* explains how to use these directives. Table 4-12 summarizes the valid arguments and their meaning.

Table 4-12 .LIST and .NLIST Directive Summary

Argument	Default	Controls listing of
SEQ	list	Source line sequence numbers
LOC	list	Location counter
BIN	list	Generated binary code
BEX	list	Binary extensions
SRC	list	Source code
COM	list	Comments
MD	list	Macro definitions, repeat range expansions
MC	list	Macro calls, repeat range expansions
ME	nolist	Macro expansions
MEB	nolist	Macro expansion binary code
CND	list	Unsatisfied conditionals, .IF and .ENDC statements
LD	nolist	Listing directives with no arguments
TOC	list	Table of Contents
TTM	terminal mode	Listing output format
SYM	list	Symbol table

/NOSHOW:value – Use this option to specify any MACRO .NLIST directive. Section 6.1.1 of the *PDP-11 MACRO Language Reference Manual* explains how to use these directives. Table 4-12 summarizes the valid arguments and their meaning.

The PRINT command lists the contents of one or more files on the line printer.

```
PRINT ┌/COPIES:n  ┐ (SP) filespecs
      │/DELETE    │
      │/[NO] LOG  │
      │/NEWFILES  │
      └/QUERY     ┘
```

In the command syntax illustrated above, filespecs represents the file or files to be printed. You can explicitly specify up to six files as input to the PRINT command. The system prints the files in the order in which you specify them in the command line. You can also use wildcards in the file specification. In this case, the system lists the files in the order in which they occur in the directory of the device you specify. If you specify more than one file, separate the files by commas. If you omit the file type for a file specification, the system assumes .LST. You can specify the entire command on one line, or you can rely on the system to prompt you for information. The PRINT command prompt is Files?. Note that if the output device is an LP05, you must terminate the file with a line feed, form feed, or carriage return.

The following sections describe the PRINT command options and include command examples.

/COPIES:n – Use this option to print more than one copy of the file. The meaningful range of values for the decimal argument, n, is from 2 to 32 (1 is the default). The following command, for example, prints three copies of the file REPORT.LST on the line printer.

```
.PRINT/COPIES:3 REPORT
```

/DELETE – Use this option to delete a file after it prints on the line printer. This option must appear following the command in the command line. The PRINT/DELETE operation does not ask you for confirmation before it executes. You must use /QUERY for this function. The following example prints a BASIC program on the line printer, then deletes it from DX1:.

```
.PRINT/DELETE DX1:PROG1.BAS
```

/LOG – This option lists on the terminal the names of the files that are printed by the current command. Normally the system prints a log only if there is a wildcard in the file specification. If you specify /QUERY, the query messages replace the log, unless you specifically type /LOG/QUERY in the command line. The following example shows a PRINT command and the resulting log.

```
.PRINT/LOG/DELETE REPORT
 Files copies/deleted:
DK:REPORT.LST to LP:
```

/NOLOG – This option prevents a list of the files that were printed from typing out on the terminal. You can use this option to suppress the log when you use a wildcard in the file specification.

/NEWFILES – Use this option in the command line if you need to print only those files that have the current date. The following example shows a convenient way to print all new files after a session at the computer.

```
.PRINT/NEWFILES *.LST
 Files copied:
DK:OUTFIL.LST   to LP:
DK:REPORT.LST   to LP:
```

/QUERY – If you use this option, the system requests confirmation from you before it performs the operation. /QUERY is particularly useful on operations that involve wildcards, when you may not be completely sure which files the system selected for an operation. Note that if you specify /QUERY in a PRINT command line that also contains a wildcard in the file specification, the confirmation messages that print on the terminal replace the log messages that would normally appear. You must respond to a query message by typing Y (or anything that begins with Y) and a carriage return to initiate execution of a particular operation. The system interprets any other response to mean NO; it does not perform the specific operation. The following example uses /QUERY.

```
.PRINT/QUERY *.LST
 Files copied:
DK:OUTFIL.LST    to LP:? NO
DK:REPORT.LST    to LP:? Y
```

The R command loads a memory image file from the system device into memory and starts execution.

```
R (SP) filespec
```

In the command syntax shown above, filespec represents the program to be executed. The default file type is .SAV. The default device is SY:. The R command is similar to the RUN command except that the file you specify in an R command string must be on the system device (SY:). Use the R command only with background jobs. The following command loads and executes MYPROG.SAV from device SY:.

```
.R MYPROG
```

The REENTER command starts the program at its reentry address (the start address minus two).

```
REENTER
```

The REENTER command accepts no options or arguments. REENTER does not clear or reset any memory areas. Use it to avoid reloading the same program for repetitive execution. You can use REENTER to return to a system program or to any program that allows for a REENTER after the program terminates. You can also use REENTER after you have used two CTRL/Cs to interrupt those programs.

If you issue the REENTER command and it is not valid for a program, the message ?KMON-F-Illegal command prints. You must start that program with an R or RUN command.

In the following example the directory program (DIR) lists the directory of DK: on the line printer. Two CTRL/Cs interrupt the listing and return to the monitor. REENTER starts DIR at its reentry address and DIR prompts for a line of input.

```
    .R DIR
    *LP:=DK:*.*
    ^C
    ^C
    .
    .REENTER
    *
```

Note in the example above that using REENTER does not continue the directory listing where it was interrupted.

The REMOVE command removes a device from the system tables.

```
REMOVE (SP) device[ , . . . device]
```

In the command syntax shown above, device represents the device to remove from the system tables. The REMOVE command accepts no options. You can enter the REMOVE command on one line, or you can rely on the system to prompt you for information. The REMOVE command prompt is Device?.

Using the REMOVE command does not change the monitor disk image; it only modifies the system tables of the monitor currently in core. This allows you to configure a special system for a single session at the computer without having to reconfigure to return to your standard device configuration. Bootstrapping the system device restores the original device configuration. To permanently REMOVE a device, include the REMOVE command in the standard system startup indirect command file.

You cannot remove the following system devices: SY (the handler for the system device), BA (the BATCH handler), and TT (the terminal handler). You can use the INSTALL command to install a new device after using the REMOVE command to remove a device (thus creating a free device slot).

The following command removes the line printer handler and the card reader handler from the system. Note that the colons (:) are optional.

```
.REMOVE  LP:,CR:
```

Use the SHOW DEVICES command to display on the terminal a list of devices that are currently available on your system.

The RENAME command assigns a new name to an existing file.

```
RENAME [/[NO] LOG      ] (SP) input-filespecs (SP) output-filespec
       [/NEWFILES      ]
       [/QUERY         ]
       [/[NO] REPLACE  ]
       [/SETDATE       ]
       [/SYSTEM        ]
```

In the command syntax illustrated above, input-filespecs represents the files to be renamed, and output-filespec represents the new name. You can specify up to six input files, but only one output file. Note that the device specification must be the same for input and output; you cannot rename a file from one device to another. If a file exists with the same name and file type as the output file you specify, the system deletes the existing file unless you use the /NOREPLACE option to prevent this.

The system has a special way of handling system (.SYS) files and files that cover bad blocks (.BAD) files. So that you do not rename system files by accident when you use a wildcard in the file specification, the system requires you to use the /SYSTEM option when you need to rename system files. To rename a .BAD file, you must specify it by explicitly giving its file name and file type. Since .BAD files cover bad blocks on a device, you usually do not need to rename or otherwise manipulate these files.

The following sections describe the options you can use with the RENAME command.

/LOG – This option lists on the terminal the files that were renamed by the current command. Normally, the system prints a log only if there is a wildcard in the file specification. If you specify /QUERY, the query messages replace the log (unless you specifically type /LOG/QUERY in the command line).

This example demonstrates logging.

```
.RENAME DX0:A%%.MAC DX0:*.FOR
 Files renamed:
DX0:ABC.MAC     to DX0:ABC.FOR
DX0:AAF.MAC     to DX0:AAF.FOR
```

/NOLOG – This option prevents a list of the files that are renamed from appearing on the terminal.

/NEWFILES – Use this option in the command line if you want to rename only those files that have the current date. This is a convenient way to access all new files after a session at the computer.

/QUERY – If you use this option, the system requests confirmation from you before it performs the operation. /QUERY is particularly useful on operations that involve wildcards, when you may not be completely sure which files the system selected for the operation. Note that if you specify /QUERY in a command line that also contains a wildcard in the file specification, the confirmation messages that print on the terminal replace the log messages that would normally appear. You must respond to a query message by typing Y (or anything that begins with Y) and a carriage return to initiate execution of a particular operation. The system interprets any other response to mean NO; it does not perform the specific operation. This example demonstrates querying.

```
.RENAME/QUERY DX0:(PIP1.SAV PIP.SAV)
 Files renamed:
DX0:PIP1.SAV    to DX0:PIP.SAV    ? Y
```

/REPLACE – This is the default mode of operation for the RENAME command. If a file exists with the same name as the file you specify for output, the system deletes that duplicate file when it performs the rename operation.

/NOREPLACE – This option prevents execution of the rename operation if a file with the same name as the output file you specify already exists on the same device. The following example uses /NOREPLACE. In this case, the output file already exists and no action occurs.

```
.RENAME/NOREPLACE DX0:TEST.SAV DX0:DUP.SAV
?PIP-W-Output file found, no operation performed DX0:TEST.SAV
```

/SETDATE – This option causes the system to put the current date on all files it renames, unless the current system date is not set. Normally, the system preserves the existing file creation date when it renames a file. The following example renames files and changes their dates.

```
.RENAME/SETDATE DX0:(*.FOR *.OLD)
 Files renamed:
DX0:ABC.FOR     to DX0:ABC.OLD
DX0:AAF.FOR     to DX0:AAF.OLD
DX0:MERGE.FOR   to DX0:MERGE.OLD
```

/SYSTEM – Use this option if you need to rename system (.SYS) files. If you omit this option, the system files are excluded from the rename operation and a message is printed on the terminal to remind you of this. This example renames MM.SYS to MX.SYS.

```
.RENAME/SYSTEM DX0:MM.SYS DX0:MX.SYS
```

The RESET command resets several background system tables and does a general clean-up of the background area.

```
RESET
```

The RESET command accepts no options or arguments. The RESET command causes the system to purge all open input/output channels, initialize the user program memory area, and unload any device handlers that were not explicitly made resident with the LOAD command. It also disables CTRL/O, clears locations 40-53, and resets the KMON (keyboard monitor) stack pointer. Use RESET before you execute a program if a device or the monitor needs reinitialization, or when you need to discard the results of previously issued GET commands. The RESET command had no effect on the foreground job. The following example uses the RESET command before running a program.

```
.RESET
.R MYPROG
```

The RESUME command continues execution of the foreground job at the point the SUSPEND command was issued.

```
RESUME
```

No arguments or options are permitted with the RESUME command. When you issue the RESUME command, the foreground job enters any completion routines that were scheduled while the job was suspended. Note that RESUME is valid only with the FB and XM monitors. The following command resumes execution of the foreground job that is currently suspended.

```
.RESUME
```

You can also use the RESUME command to execute a foreground job that you start with FRUN using /P.

The RUN command loads a memory image file into memory and starts execution.

```
RUN (SP) filespec [ { (SP) input-list [ (SP) output-list] } ]
                  [ { (SP) argument                      } ]
```

In the command syntax illustrated above, filespec represents the program to execute. The system assumes a .SAV file type for the executable file, which can reside on any RT-11 block-replaceable device. The default device is DK:. The RUN command automatically loads the device handler for the device you specify if it is not already resident. This eliminates the need to explicitly load a device handler when you run an overlaid program from a device other than the system device. The RUN command executes only those programs that have been linked to run as background jobs. You cannot use RUN on a virtual job that executes under the XM monitor. The following command, for example, executes MYPROG.SAV, which is stored on device DX1:.

```
.RUN DX1:MYPROG
```

You can also specify in the RUN command an argument to pass to the program, or a list of input and output specifications. This allows you to specify a line of input for a user program or for a system utility program (which accepts file specifications in the special syntax described in Chapter 6). The system automatically converts the input-list and the output-list you specify into a format that the CSI (Command String Interpreter) accepts. For example, to execute the directory program (DIR) and obtain a complete listing of the directory of DX1: on the printer, you can use the following command.

```
.RUN DIR DX1:*.* LP:/E
.
```

This command has the same effect as the following lines.

```
.RUN DIR
*LP:/E=DX1:*.*
*^C
.
```

Note that when you use either an argument or an input-list and output-list with RUN, control returns to the monitor when the program completes.

The SAVE command writes memory areas in memory image format to the file and device that you specify.

```
SAVE (SP) filespec[ (SP) parameters]
```

In the command syntax shown above, filespec represents the file to be saved on a block-replaceable device. If you do not specify a file type, the system uses .SAV. The parameters represent memory locations to be saved.

Parameters are of the form:

address[-address(2)] [,address(3)[-address(n)]]

where

address	is an octal value representing a specific block of memory locations to be saved. If you specify more than one address, each address must be higher than the previous one. RT-11 transfers memory in 256-word blocks beginning on boundaries that are multiples of 256 (decimal). If the locations you specify make a block that is less than 256 words, the system saves additional words to make a 256-word block.

The system saves memory from location 0 to the highest memory address specified by the parameter list or to the program high limit (location 50 in the system communication area). Initially, the system gives the start address and the JSW (Job Status Word) the default value 0 and sets the stack to 1000. If you want to change these or any of the following addresses, you can use the Deposit command to alter them and the SAVE command to save the correct areas.

AREA	LOCATION
Start address	40
Stack	42
JSW	44
USR address	46
High address	50
Fill characters	56

If you change the values of the addresses, it is your responsibility to reset them to their default values. For more information concerning these addresses refer to the *RT-11 Advanced Programmer's Guide*. Note that the SAVE command does not write the overlay segments of programs; it saves only the root segment.

The following command saves location 10000 through 11777 and 14000 through 14777. It stores the contents of these locations in the file FILE1.SAV on device DK:.

```
    .SAVE FILE1 10000-11000,14000-14100
```

The next example sets the reenter bit in the JSW and saves locations 1000 through 5777 in file PRAM.SAV on device SY:.

```
    .D 44=2000
    .SAVE SY:PRAM 1000-5777
```

The SET command changes device handler characteristics and certain system configuration parameters.

```
SET (SP) {physical-device-name} (SP) condition
         {item               }
```

In the command syntax illustrated above, physical-device-name represents the device handler whose characteristics you need to modify.

See Table 3-1 for a list of the standard RT-11 permanent device names. The argument, item, represents a system parameter that you need to modify. The system items you can change include error handling (SET ERROR) and wildcard handling (SET WILDCARDS). Table 4-13 lists the devices and items you can modify as well as the valid conditions for these devices and items. If you set more than one condition for a device, separate the conditions by commas. With the exception of the SET TT, SET USR, and SET item commands, the SET command locates the file SY:device.SYS and permanently modifies it. The SET commands are valid for all three RT-11 monitors unless otherwise specified. They permanently modify the device handlers (except where noted); this means that the conditions remain set even across a reboot. For those SET commands that do not permanently modify the device handlers, the conditions return to the default setting after a reboot. To make these settings appear permanent, include the appropriate SET commands in your system's startup indirect command file (see Section 4.3.3). The command you enter must be completely valid for the modification to take place. If a handler is already loaded when you issue a SET command for it, you must unload the handler and install a fresh copy from the system device for the modification to have an effect on execution. Note that the colon (:) after each device name is optional.

PDP-11 WORD

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
UNUSED (ALWAYS 0)				ZONE 12	ZONE 11	ZONE 0	ZONE 1	ZONE 2	ZONE 3	ZONE 4	ZONE 5	ZONE 6	ZONE 7	ZONE 8	ZONE 9

Figure 4-2 Format of a 12-bit Binary Number

Table 4-13 SET Device Conditions

Device or Item	Condition	Action
CR:	CODE=n	Modifies the card reader handler to use either the DEC 026 or DEC 029 card codes. The argument, n, must be either 26 or 29. The default value is 29.
CR:	CRLF	Appends a carriage return/line feed combination to each card image. This is the normal mode.
CR:	NOCRLF	Transfers each card image without appending a carriage return/line feed combination. The default is CRLF.
CR:	HANG	Waits for you to make a correction if the reader is not ready at the start of a transfer. This is the normal mode.

(Continued on next page)

Table 4-13 (Cont.) SET Device Conditions

Device or Item	Condition	Action
CR:	NOHANG	Generates an immediate error if the device is not ready at the start of a transfer. The handler waits (regardless of how the condition is set) if the reader becomes not ready during a transfer (i.e., the input hopper is empty, but an end-of-file card has not been read). The default is HANG.
CR:	IMAGE	Causes each card column to be stored as a 12-bit binary number, one column per word. The CODE option has no effect in IMAGE mode. Figure 4-2 illustrates the format of the 12-bit binary number. This format allows the system to read binary card images. It is especially useful if you use a special encoding of punch combinations. Mark-sense cards can be read in this mode. The default is NOIMAGE.
CR:	NOIMAGE	Allows the normal translation (as specified by the CODE option) to take place. The system packs data one column per byte. It translates invalid punch combinations into the error character, ASCII backslash (\), which is octal code 134. This is the normal mode.
CR:	TRIM	Removes trailing blanks from each card that the system reads. You should not use TRIM and NOCRLF together because card boundaries become difficult to read. TRIM is the normal mode.
CR:	NOTRIM	Transfers a full 80 characters per card. The default is TRIM.
CT:	RAW	Performs a read-after-write check for every record written. It retries if an output error occurs. If three retries fail, the system indicates an output error. The default is NORAW.
CT:	NORAW	Writes every record directly without reading it back for verification. This setting significantly increases transfer rates at the risk of increased error rates. This is the normal mode.
EDIT	EDIT	Invokes the text editor EDIT with the keyboard monitor EDIT command. This is the normal mode. The system returns to this condition after a reboot.
EDIT	TECO	Invokes the text editor TECO with the keyboard monitor EDIT command. The default is EDIT. The system returns to that condition after a reboot.
ERROR	ERROR	Causes indirect command files and keyboard monitor commands that perform multiple operations (such as EXECUTE, which combines assembling, linking, and running) to abort if errors or severe errors occur. An example of an error is an undefined symbol in an assembly. An example of a severe error is a device that is write-locked when the system attempts to write to it. If either condition occurs, the indirect command file or keyboard monitor command aborts the next time the monitor get control of the system. This is the normal setting. The system returns to this condition after a reboot.

(Continued on next page)

Table 4-13 (Cont.) SET Device Conditions

Device or Item	Condition	Action
ERROR	NONE	Allows indirect command files and keyboard monitor commands to continue to execute even though they contain significant errors. Most monitor fatal errors still cause the indirect command file or keyboard monitor command to abort. See SET ERROR ERROR. SET ERROR ERROR is the default setting. The system returns to that condition after a reboot.
ERROR	SEVERE	Causes indirect command files and keyboard monitor commands to abort if severe errors occur. See SET ERROR ERROR. SET ERROR ERROR is the default setting. The system returns to that condition after a reboot.
ERROR	WARNING	Causes indirect command files and keyboard monitor commands to abort if warnings, errors, or severe errors occur. See SET ERROR ERROR. SET ERROR ERROR is the default setting. The system returns to that condition after a reboot.
LP:	CR	Sends carriage returns to the printer. To allow overstriking on the printer, use this condition for any FORTRAN program that uses formatted input and output. Use CR also for any LS11 or LP05 line printer to prevent loss of the last line in the buffer. This is the normal mode.
LP:	NOCR	Prevents the system from sending carriage returns to the printer. This setting produces a significant increase in printing speed on LP11 printers. The line printer controller causes a line feed to perform the functions of a carriage return. The default is CR.
LP:	CTRL	Passes all characters, including nonprinting control characters, to the printer. Use this condition to pass the bell character to the LA180 printing terminal. You can use this mode for LS11 line printers. (Other line printers print a space for a control character.) The default is NOCTRL.
LP:	NOCTRL	Ignores non-printing control characters. This is the normal mode.
LP:	FORM0	Issues a form feed before a request to print blocks 0. This is the normal mode.
LP:	NOFORM0	Turns off FORM0 mode. The default is FORM0.
LP:	HANG	Waits for you to make a correction if the line printer is not ready or becomes not ready during printing. If you expect output from the line printer and the system does not respond or appears to be idle, check to see if the line printer is powered on and ready to print. This is the normal mode.

(Continued on next page)

Table 4-13 (Cont.) SET Device Conditions

Device or Item	Condition	Action
LP:	NOHANG	Generates an immediate error if the line printer is not ready. The default is NOHANG.
LP:	LC	Allows the system to send lower case characters to the printer. Use this condition if your printer has a lower case character set. The default is NOLC.
LP:	NOLC	Translates lower case characters to upper case before printing. This is the normal mode.
LP:	TAB	Sends TAB characters to the LA180 line printer. The default is NOTAB.
LP:	NOTAB	Does not send TAB characters to the line printer. This is the normal mode.
LP:	WIDTH=n	Sets the line printer width to n, where n is an integer between 30 and 255, inclusive. The system ignores any characters that print past column n.
MM:	DEFALT=9	Returns to default settings for 9-track tape. The 9-track defaults are: DENSE=809 ODDPAR NODUMP
MM:	DENSE=[800 or 809 or 1600]	Sets density for the 9-track tape handler. Do not alter the density setting within a volume. A density setting of 1600 bits per inch (BPI) automatically sets parity to odd. The valid density settings for 9-track tape are: 800 BPI 1600 BPI
MM:	ODDPAR	Sets parity to odd for 9-track tape. DIGITAL recommends this setting.
MM:	NOODDPAR	Sets parity to even for 9-track tape. DIGITAL does not recommend this setting for normal operation, and provides it only for compatibility with other systems.
MT:	DEFALT=[7 or 9]	Returns to default settings for 7- or 9-track tape. The 7-track defaults are: DENSE=807 ODDPAR DUMP

(Continued on next page)

Table 4-13 (Cont.) SET Device Conditions

Device or Item	Condition	Action
MT: (Cont.)		The 9-track defaults are: DENSE=809 ODDPAR NODUMP
MT:	DENSE=[200 or 556 or 807 or 800 or 809]	Sets density for 7- or 9-track tape. 807 represents 800 BPI for 7-track tape; 800 or 809 represents 800 BPI for 9-track tape. Do not alter the density within a tape volume. You must set density to 807 for 7 track tape if you want dump mode. The valid density settings for 7 and 9 track tape are: 7-track: 200 BPI 556 BPI 800 BPI 800 BPI Dump 9-track: 800 BPI
MT:	DUMP	Writes bytes to 7-track tape. You must also set density to 807.
MT:	ODDPAR	Sets parity to odd for 7- or 9-track tape. DIGITAL recommends this setting.
MT:	NOODDPAR	Sets parity to even for 7- or 9-track tape. DIGITAL does not recommend this setting for normal operation, and provides it only for compatibility with other systems.
TT:	CONSOL=n	Directs the system to use as the console terminal, the terminal whose logical unit number you specify. The default value is 0, which represents the original console terminal. The terminal whose logical unit number you specify must not be currently attached by the foreground job. The system returns to this default after a reboot.
TT:	CRLF	Issues a carriage return/line feed combination on the console terminal whenever you attempt to type past the right margin. You can change the margin with the WIDTH command. This is the normal mode. This setting is not valid for the SJ monitor. The system returns to this condition after a reboot.
TT:	NOCRLF	Takes no special action at the right margin. This setting is not valid for the SJ monitor. The default is CRLF. The system returns to that condition after a reboot.

(Continued on next page)

Table 4-13 (Cont.) SET Device Conditions

Device or Item	Condition	Action
TT:	FB	Treats CTRL/B and CTRL/F as background and foreground program control characters and does not transmit them to your program. This is the normal mode. This setting is not valid for the SJ monitor. The system returns to this condition after a reboot.
TT:	NOFB	Causes CTRL/B and CTRL/F to have no special meaning. Issue SET TT NOFB to KMON, which runs as a background job, to disable all communication with the foreground job. To enable communication with the foreground job, issue the command SET TT FB. This setting is not valid for the SJ monitor. The default is FB. The system returns to that condition after a reboot.
TT:	FORM	Indicates that the console terminal is capable of executing hardware form feeds. This setting is not valid for the SJ monitor.
TT:	NOFORM	Simulates form feeds by generating eight line feeds. This setting is not valid for the SJ monitor. This is the normal mode. The system returns to this condition after a reboot.
TT:	HOLD	Enables the Hold Screen mode of operation for the VT50 terminal. The command has no effect on any other terminal, but it can cause a left square bracket ([) to print. This setting is valid for all monitors. This is the normal mode. The system returns to this condition after a reboot.
TT:	NOHOLD	Disables the Hold Screen mode of operation for the VT50 terminal. The command has no effect on any other terminal, but it can cause a backslash (\) to print. This setting is valid for all monitors. The default is HOLD. The system returns to that condition after a reboot.
TT:	PAGE	Treats CTRL/S and CTRL/Q characters as terminal output hold and unhold flags and does not transmit them to your program. This setting is not valid for the SJ monitor. This is the normal mode. The system returns to this condition after a reboot.
TT:	NOPAGE	Causes CTRL/S and CTRL/Q to have no special meaning. This setting is not valid for the SJ monitor. The default is PAGE. The system returns to that condition after a reboot.
TT:	QUIET	Prevents the system from echoing lines from indirect files. The default is NOQUIET. The system returns to that condition after a reboot.
TT:	NOQUIET	Echoes lines from indirect files. This is the default mode. The system returns to this condition after a reboot.
TT:	SCOPE	Echoes RUBOUT characters as backspace-space-backspace. Use this mode if your console terminal is a VT50, VT05, VT52, VT55, VT61, or if GT ON is in effect. This setting is not valid for the SJ monitor. The default is NOSCOPE. The system returns to that condition after a reboot.

(Continued on next page)

Table 4-13 (Cont.) SET Device Conditions

Device or Item	Condition	Action
TT:	NOSCOPE	Echoes each RUBOUT character as a backslash followed by the character deleted. This is the normal mode. This setting is not valid for the SJ monitor. The system returns to this condition after a reboot.
TT:	TAB	Indicates that the console terminal is capable of executing hardware tabs. This setting is not valid for the SJ monitor. The default is NOTAB. The system returns to that condition after a reboot.
TT:	NOTAB	Simulates tab stops every eight positions. VT05 and VT50 terminals generally have hardware tabs. This setting is not valid for the SJ monitor. This is the normal mode. The system returns to this condition after a reboot.
TT:	WIDTH=n	Sets the terminal width to n, where n is an integer between 30 and 255. The system initially sets the width to 72. This setting is not valid for the SJ monitor. (See SET TT CRLF.) The system returns to width 72 after a reboot.
WILDCARDS	EXPLICIT	Causes the system to recognize file specifications exactly as you type them. If you omit a file name or a file type in a file specification the system does not automatically replace the missing item with an asterisk (*). Wildcards are described in Section 4.2. The default is IMPLICIT. The system returns to that condition after a reboot.
WILDCARDS	IMPLICIT	Causes the system to interpret missing fields in file specifications of certain commands as asterisks (*). Wildcards are described in Section 4.2 of this manual. Table 4-2 shows how the system interprets commands that have missing fields. This is the normal mode. The system returns to this condition after a reboot.
USR	SWAP	Allows the background job to place the USR in a swapping state. This setting is not valid for the XM monitor. This is the normal mode. The system returns to this condition after a reboot.
USR	NOSWAP	Prevents the background job from placing the USR in a swapping state This setting is not valid for the XM monitor. The default is SWAP. The system returns to that condition after a reboot.

The following examples illustrate the SET command. This command allows the system to send lower case characters to the printer:

```
.SET LP: LC
```

The next command sets the system wildcard default to implicit.

```
.SET WILDCARDS IMPLICIT
```

As a result of this command the system inserts an asterisk in place of a missing file name or file type in a file specification for certain commands. See Table 4-2 for a list of these commands.

The SHOW command prints at the terminal all the devices known to the system and any logical names assigned to these devices.

```
SHOW[ (SP) DEVICES]
```

The devices the system lists are those known by the RT-11 monitor currently running in memory. This list reflects any additions or deletions you have made with the INSTALL and REMOVE commands. The final entry in the listing shows whether the USR is set to SWAP or NOSWAP. The listing also includes additional information about particular devices. The informational messages and their meanings are:

(B) or =B	Indicates that the device or unit is assigned to the background job. (For FB and XM monitors only.)
(F) or =F	Indicates that the device or unit is assigned to the foreground job. (For FB and XM monitors only.)
<FREE>	Shows that the device slot is unused. You can use the INSTALL command to install a device into the free slot. Create a free slot by using the REMOVE command to remove a device.
(LOADED)	Shows that the handler for the device has been loaded into memory with the LOAD command.
(RESIDENT)	Indicates that the handler for the device is included in the resident monitor.
=logical-device-name(1), logical-device-name(2) . . . ,logical-device-name(n)	Shows that the device or unit has been assigned the indicated logical device names with the ASSIGN command.

The following example was created under the FB monitor. It shows the status of all devices known to the system.

```
.SHOW
TT   (Resident)
RK   (Resident)
  RK0    = SY
<Free>
<Free>
DX   (Loaded)
  DX0  (B)
  DX1      = DK
DT
MT   (Loaded=F)
CT
LP         = OUT
<Free>
<Free>
BA
EL
NL
<Free>

USR Swap
```

The listing shows first that TT and RK are resident in memory. The other device handlers known to the system are: DX, DT, MT, CT, LP, BA, EL, and NL. There are five free slots in the table. RK0: has the logical name SY: and DX1: has the logical name DK:. The logical name OUT: is assigned to LP:. The DX handler is loaded and device DX0: belongs to the background job. The MT handler is loaded and belongs to the foreground job. The USR is set to SWAP.

The SQUEEZE command consolidates in a single area all unused blocks on the device you specify.

```
SQUEEZE [/OUTPUT:device ] (SP) device
        [/[NO] QUERY    ]
```

In the command syntax illustrated above, device represents the disk or DECtape to be compressed. To perform a squeeze operation, the system moves all the files to the beginning of the device you specify, producing a single unused area after the group of files. The squeeze operation does not change the bootstrap blocks of a device. The system prints a confirmation message before it performs the squeeze operation. You must type Y followed by a carriage return to execute the command.

The squeeze operation does not move files with .BAD file types. This feature prevents you from reusing bad blocks that occur on a disk. The system inserts files before and after .BAD files until the space between the last file it moved and the .BAD file is smaller than the next file to be moved.

If you perform a squeeze operation on the system device, the system automatically reboots when the compress operation completes. This reboot takes place in order to prevent system crashes that might occur when the monitor file is moved.

/OUTPUT:filespec – Use this option to transfer all the files from the input device to the output device in compressed format. This operation leaves the input device unchanged. The output device must be an initialized disk or DECtape. (Use the INITIALIZE command to do this.) Note that the system never queries you for confirmation before this operation proceeds. If the output device is not initialized, the system prints an error message and does not execute the command. The following example transfers all the files from RK0: to RK1: in compressed format, leaving RK0: unchanged.

```
.SQUEEZE/OUTPUT:RK1: RK0:
```

/QUERY – This option causes the system to print a confirmation message before it executes a squeeze operation. You must respond by typing a Y followed by a carriage return for execution to proceed. This is the default operation. /QUERY is meaningless with the /OUTPUT option.

/NOQUERY – Use this option to suppress the confirmation message that prints before a squeeze operation executes. The following command compresses all the files on device DT1: and does not query.

```
.SQUEEZE/NOQUERY DT1:
```

The START command initiates execution of the program currently in memory (loaded with the GET command) at the address you specify.

```
START[ (SP) address]
```

In the command syntax shown above, address is an even octal number representing any 16-bit address. If you omit the address or if you specify 0, the system uses the starting address that is in location 40. If the address you specify does not exist or is invalid for any reason, a trap to location 4 occurs and the monitor prints an error message. Note that this command is valid for background jobs only. The following command loads MYPROG.SAV into memory and begins execution.

```
.GET MYPROG
.START
```

The next example loads MYPROG.SAV and ODT.SAV into memory, and begins execution at ODT's starting address.

```
.GET MYPROG
.GET ODT
.START
ODT V01.04
*
```

The SUSPEND command stops execution of the foreground job.

```
SUSPEND
```

No arguments or options are accepted with this command. The SUSPEND command is not valid for the SJ monitor. The system permits foreground input and output that are already in progress to finish; however, it issues no new input or output requests and enters no completion routines (see the *RT-11 Advanced Programmer's Guide* for a detailed explanation of completion routines). You can continue execution of the job by typing the RESUME command. The following command suspends execution of the foreground job that is currently running.

```
.SUSPEND
```

Use the TIME command to set the time of day or to display the current time of day.

```
TIME[ (SP) hh:mm:ss]
```

In the command syntax shown above, hh represents hours (from 0 to 23); mm represents minutes (from 0 to 59) and ss represents seconds (from 0 to 59). The system keeps time on a 24-hour clock.

To enter the time of day, specify the time in the format described above. You should do this as soon as you bootstrap the system. The following example enters the time, 11:15:00 A.M.

```
.TIME 11:15
```

As this example shows, if you omit one of the arguments the system assumes 0. The system automatically resets the time each day at midnight.

To display the current time of day, type the TIME command without an argument, as this example shows.

```
.TIME
11:15:01
```

When the RT-11 system is installed, the clock rate is preset to 60 cycles. Consult the *RT-11 System Generation Manual* for information on setting the clock to a 50-cycle rate.

The TYPE command types (or prints) the contents of one or more files on the terminal.

```
TYPE ┌/COPIES:n ┐ (SP) filespecs
     │/DELETE   │
     │/[NO]LOG  │
     │/NEWFILES │
     └/QUERY    ┘
```

In the command syntax illustrated above, filespecs represents the file or files to be typed. You can explicitly specify up to six files as input to the TYPE command. The system types the files in the order in which you specify them in the command line. You can also use wildcards in the file specification. In this case, the system types the files in the order in which they occur in the directory of the device you specify. If you specify more than one file, separate the files by commas. If you omit the file type for a file specification, the system assumes .LST. You can specify the entire command on one line, or you can rely on the system to prompt you for information. The TYPE command prompt is Files?.

The following sections describe the TYPE command options and include command examples.

/COPIES:n – Use this option to type more than one copy of the file. The meaningful range of values for the decimal argument, n, is from 2 to 32 (1 is the default). The following command, for example, types three copies of the file REPORT.LST on the terminal.

```
.TYPE/COPIES:3 REPORT
```

/DELETE – Use this option to delete a file after it is typed on the terminal. This option must appear following the command in the command line. The TYPE/DELETE operation does not ask you for confirmation before it executes. You must use /QUERY for this function. The following example types a BASIC program on the terminal, then deletes it from DX1:.

```
.TYPE/DELETE DX1:PROG1.BAS
```

/LOG – This option prints on the terminal the names of the files that were typed by the current command. Normally, the system prints a log only if there is a wildcard in the file specification. If you specify /QUERY, the query message replaces the log, unless you specifically type /LOG/QUERY in the command line. The following example shows a TYPE command and the resulting log.

```
.TYPE/LOG OUTFIL.LST
 Files copied:
DK:OUTFIL.LST   to TT:
```

/NOLOG – This option prevents a list of the files that were typed from printing on the terminal. You can use this option to suppress the log if you use a wildcard in the file specification.

/NEWFILES – Use this option in the command line if you need to type only those files that have the current date. The following example shows a convenient way to type all new files after a session at the computer.

```
.TYPE/NEWFILES *.LST
 Files copied:
DK:REPORT.LST   to TT:
```

/QUERY – If you use this option, the system requests confirmation from you before it performs the operation. /QUERY is particularly useful on operations that involve wildcards, when you may not be completely sure which files the system selected for an operation. Note that if you specify /QUERY in a TYPE command line that also contains a wildcard in the file specification, the confirmation messages that print on the terminal replace the log messages that would normally appear. You must respond to a query message by typing Y (or anything that begins with Y) and a carriage return to initiate execution of a particular operation. The system interprets any other response as NO and it does not perform the specific operation.

```
.TYPE/QUERY/DELETE *.LST
 Files copied/deleted:
DK:OUTFIL.LST    to TT:? NO
DK:REPORT.LST    to TT:? Y
```

The UNLOAD command makes handlers that were previously loaded non-resident, thus freeing the memory space they occupied.

```
UNLOAD (SP) device[ ,... device]
```

In the command syntax shown above, device represents the device handler to unload.

UNLOAD clears ownership for all units of the device type you specify. A request to unload the system device handler clears ownership for any assigned units for that device, but the handler itself remains resident. After you issue the UNLOAD command, the system returns any memory it frees to a free memory list. The background job eventually reclaims free memory. Note that if you interrupt an operation that involves magtapes or cassette, you must unload and then load (with the LOAD command) the appropriate device handler (MM, MT, or CT).

The system does not accept an UNLOAD command while a foreground job is running if the foreground job owns any units of that device. This is because a handler that the foreground job needs might become nonresident. You can unload a device while a foreground job is running if none of its units belong to the foreground job.

A special function of this command is to remove a terminated foreground job and reclaim memory, since the system does not automatically return the space occupied by the foreground job to the free memory list. The next command unloads the foreground job and frees the memory it occupied. This command is valid only if the foreground job is not running.

```
.UNLOAD RK:
```

The following command clears ownership of all units of RK if RK: is the system device.

```
.UNLOAD LP:,DT:
```

The next command releases the line printer and DECtape handlers and frees the area they previously held.

```
.UNLOAD FG
```

PART III
TEXT EDITING

You use an editor to create and modify textual material. PART III describes the RT-11 text editor, EDIT, and explains how to use it.

CHAPTER 5
TEXT EDITOR

The text editor (EDIT) is a program that creates or modifies ASCII source files for use as input to other system programs such as the MACRO assembler or the FORTRAN compiler. EDIT, which accepts commands you type at the terminal, reads ASCII files from any input device, makes specific changes, and writes on any output device. EDIT allows efficient use of VT11 or VS60 display hardware, if they are part of the system configuration.

The editor considers a file to be divided into logical units called pages. A page of text is generally 50-60 lines long (delimited by form feed characters) and corresponds approximately to a physical page of a program listing. The editor reads one page of text at a time from the input file into its internal buffers where the page becomes available for editing. You can then use editing commands to:

- Locate text to be changed
- Execute and verify the changes
- List an edited page on the console terminal
- Output a page of text to the output file.

5.1 CALLING AND USING EDIT

You can call the text editor when you are at monitor level. The syntax of the command is:

```
EDIT [ { /CREATE                              } ]  (SP) filespec[/ALLOCATE:size]
     [ { /INSPECT                             } ]
     [ { /OUTPUT:filespec[/ALLOCATE:size]     } ]
```

See Section 4.4 for a description of the EDIT command and its options.

5.2 MODES OF OPERATION

Normally, the editor operates in either command mode or text mode. In command mode the editor interprets all input you type on the keyboard as commands to perform some operation. In text mode the editor interprets all typed input as text to replace, insert into, or append to the contents of the text buffer.

Immediately after being loaded into memory and started, the editor is in command mode. EDIT prints an asterisk at the left margin of the console terminal page to indicate that it is ready to accept a command. Terminate all commands by pressing the ESCAPE key twice in succession. Execution of commands proceeds from left to right. Should EDIT encounter an error before it begins execution of a command string, it prints an error message followed by an asterisk at the beginning of a new line, indicating that it is still in command mode and awaiting a legal command. EDIT does not execute the command in error or any succeeding command. You should retype the command correctly.

To enter text mode, type a command that must be followed by a text string. These commands insert, replace, exchange, or otherwise manipulate text. When you type one of these commands, EDIT recognizes all succeeding characters as part of the text string until it encounters an ESCAPE character. The ESCAPE terminates the text string and causes the editor to reenter command mode.

You can use a special editing mode, called immediate mode, whenever the VT-11 display hardware is running. Section 5.7.2 describes this mode.

5.3 SPECIAL KEY COMMANDS

Table 5-1 lists the EDIT key commands. Type a control command by holding down the CTRL key while typing the appropriate character.

Table 5-1 EDIT Key Commands

Key	Explanation
ESCAPE, ALTMODE, or SEL	Echoes $. A single ESCAPE terminates a text string. A double ESCAPE (two consecutive ESCAPEs) executes the command string. For example: `*GMOV A,B$-1D$$` The first ESCAPE ($) terminates the text object (MOV A,B) of the Get command. The double ESCAPE ($$) terminates the Delete command and executes the entire command string. In this example, the character B will be deleted as a result of execution.

CTRL/C	Echoes at the terminal as ^C. If EDIT encounters a CTRL/C as a command in command mode, it terminates execution and returns control to the monitor. You can restart the editor by typing R EDIT or REENTER in response to the monitor's prompt. If EDIT encounters a CTRL/C in a text object, EDIT includes the CTRL/C in the text object, just like any other character. If the editor is executing a lengthy command and you want to stop EDIT, type two CTRL/C commands in succession. This will abort the command, generate the ?EDIT-F-COMMAND ABORTED error message, and return the editor to command mode. For example:
	`*I^C^C^C$$`
	`*^C$$`
	In the first command, the three CTRL/C characters are part of the text object of the Insert command. EDIT treats them like any other character. In the second command string, the CTRL/C occurs at command level, and causes the editor to terminate.
	If no commands (other than CLOSE) are executed between the time you terminate the editor and the time you issue a REENTER command, the text buffer is preserved exactly as it was at program termination. However, only the text buffer is preserved. The input and output files are closed, and the save and macro buffers are reinitialized.
	If you inadvertently terminate an editing session before the output file can be closed, you can often use the monitor CLOSE command to make permanent the portion of the output file that has already been written (see Section 4.4). You can then reenter the editor, open a new output file, and continue the editing session.

(Continued on next page)

Table 5-1 (Cont.) EDIT Key Commands

Key	Explanation
CTRL/O	Echoes ^O and a carriage return. Inhibits printing on the terminal until completion of the current command string. Typing a second CTRL/O resumes output.
CTRL/U	Echoes ^U and a carriage return. Deletes all the characters on the current terminal input line. (Equivalent to pressing the RUBOUT key until all the characters back to the beginning of the line are deleted.)
RUBOUT or DELETE	Deletes a character from the current command line; echoes a backslash followed by the character deleted. Each succeeding RUBOUT you type deletes and echoes another character. An enclosing backslash prints when you type a key other than RUBOUT. This erasure is done from right to left. Since EDIT accepts multiple line commands, RUBOUT can delete past the carriage return/line feed combination and delete characters on the previous line. You can use RUBOUT in both command and text modes.
TAB	Spaces to the next tab stop. Tab stops are positioned every eight spaces on the terminal; pressing the TAB key causes the carriage to advance to the next tab position.

CTRL/X	Echoes ^X and a carriage return. CTRL/X causes the editor to ignore the entire command string you are currently entering. The editor prints a carriage return/line feed combination and an asterisk to indicate that you can enter another command. For example:
	`*IABCD`
	`EFGH^X`
	`*`
	A CTRL/U would cause only deletion of EFGH; CTRL/X erases the entire command.

5.4 COMMAND STRUCTURE

EDIT commands fall into eight general categories. Table 5-2 lists these categories and the commands they include.

Table 5-2 EDIT Command Categories

Category	Commands	Section
File open and close	Edit Backup	5.6.1.3
	Edit Read	5.6.1.1
	Edit Write	5.6.1.2
	End File	5.6.1.4
File input/output	EXit	5.6.2.4
	Next	5.6.2.3
	Read	5.6.2.1
	Write	5.6.2.2

(Continued on next page)

Table 5-2 (Cont.) EDIT Command Categories

Category	Commands	Section
Immediate mode	ESCAPE	5.7.2
	CTRL D	5.7.2
	CTRL G	5.7.2
	CTRL N	5.7.2
	CTRL V	5.7.2
	RUBOUT	5.7.2
Pointer location	Advance	5.6.3.3
	Beginning	5.6.3.1
	Jump	5.6.3.2
Search	Find	5.6.4.2
	Get	5.6.4.1
	Position	5.6.4.3
Text listing	List	5.6.5.1
	Verify	5.6.5.2
Text modification	Change	5.6.6.4
	Delete	5.6.6.2
	eXchange	5.6.6.5
	Insert	5.6.6.1
	Kill	5.6.6.3
Utility	Edit Console	5.7.1
	Edit Display	5.7.1
	Edit Lower	5.6.7.6
	Edit Upper	5.6.7.6
	Edit Version	5.6.7.5
	Executive Macro	5.6.7.4
	Macro	5.6.7.3
	Save	5.6.7.1
	Unsave	5.6.7.2

The general syntax for all the EDIT commands, with the exception of the immediate mode commands, is:

```
[n]C[text]$
```

or

```
[n]C$
```

where

n	represents one of the legal arguments from Table 5-3.
C	represents a 1- or 2-letter command.
text	represents a string of successive ASCII characters.

As a rule, commands are separated from one another by a single ESCAPE; however, if the command requires no text, the separating ESCAPE is not necessary. Commands are terminated by a single ESCAPE; typing a second ESCAPE begins execution. (You use ESCAPE differently when immediate mode is in effect; Section 5.7.2 details its use in this case.)

The syntax of display editor commands is somewhat different from the normal editing command format, and is described in Section 5.7.

5.4.1 Arguments

An argument is positioned before a command letter. It specifies either the particular portion of text to be affected by the command or the number of times to perform the command. With some commands, this specification is implicit and no argument is needed; other editing commands require an argument. Table 5-3 lists the possible arguments and their meanings.

Table 5-3 Command Arguments

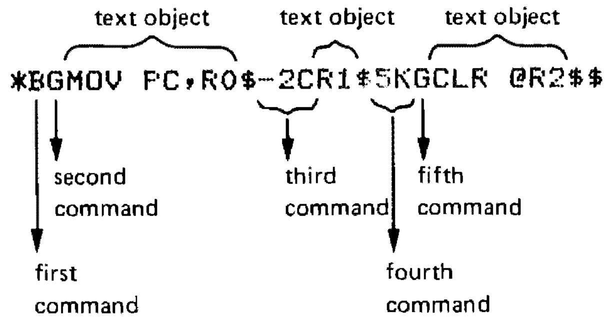
Argument	Meaning
n	Stands for any integer in the range -16383 to +16383 and may, except where noted, be preceded by a plus (+) or minus (-) sign. If no sign precedes n, it is assumed to be a positive number. The absence of n implies a 1 (or -1 if a minus sign precedes a command). n can represent the number of characters or lines forward or backward (+ or -) to move the pointer, or it can represent the number of times to execute the operation.
0	Indicates the text between the beginning of the current line and the reference pointer (see Section 5.4.3).
/	Refers to the text between the reference pointer and the end of the text in the buffer.
=	Use only with the J, D, and C commands to represent -n, where n is equal to the length of the last text argument used.

The roles of all arguments are explained more specifically in the following sections.

5.4.2 Command Strings

All EDIT command strings are terminated by two successive ESCAPE characters. Use spaces, carriage returns, and line feeds within a command string to increase command readability. EDIT ignores them unless they appear in a text string. Commands to insert text can contain text strings that are several lines long. Each line you enter is terminated by the carriage return key, which inserts both a carriage return and a line feed character into the text. The entire command is terminated by a double ESCAPE.

You can string several commands together and execute them in sequence. For example:



```
*BGMOV PC,R0$-2CR1$5KGCLR @R2$$
```

where

B	is the first command.
GMOV PC,R0	is the second command (MOV PC,R0 is the text object).
-2CR1	is the third command (R1 is the text object).
5K	is the fourth command.
GCLR @R2	is the fifth command (CLR @R2 is the text object).
$	separates the end of each text object from the following command.
$$	executes the commands.

Execution of a command string begins when you type the double ESCAPE and proceeds from left to right. Except when they are part of a text string, EDIT ignores spaces, carriage returns, line feeds, and single ESCAPEs. For example:

```
*BGMOV R0$=CCLR R1$AV$$
```

You can also type this command as:

```
*B$ GMOV R0$
=CCLR R1$
A$ V$$
```

Execution of the two commands will be the same.

5.4.3 The Current Location Pointer

Most EDIT commands function with respect to a movable reference pointer that is normally located between the most recent character operated upon and the next character in the buffer. It is important to think of this pointer as being between two characters and never directly on a character. At the start of editing operations, the pointer precedes the first character in the buffer, although it is not displayed on the console terminal. At any given time during the editing procedure, think of the pointer as representing the current position of the editor in the text. The pointer moves during editing operations according to the type of editing operation being performed. Refer to text in the buffer as so many characters or lines preceding or following the pointer.

5.4.4 Character- and Line-Oriented Command Properties

Edit commands are either character-oriented or line-oriented: character-oriented commands affect a specified number of characters preceding or following the pointer; line-oriented commands operate on entire lines of text.

The argument of character-oriented commands specifies the number of characters in the buffer on which to operate. If n is unsigned (positive), the command operates in a forward direction. If n is preceded by a minus sign (negative), the command moves the reference pointer backwards. (LF), (RET), and null characters, although not printed. are embedded in text lines, counted as characters in character-oriented commands, and treated as any other text characters. When you press the (RET) key, both a carriage return and a line feed character are inserted into the text. For example, assume the pointer is positioned as indicated in the following text (↑ represents the current position of the pointer):

```
MOV   #VECT,R2(RET)(LF)↑
CLR   @R2(RET)(LF)
```

The EDIT command −2J moves the pointer back two characters to precede the carriage return character.

```
MOV     #VECT,R2(RET)(LF)
                ↑
CLR     @R2(RET)(LF)
```

The command 10J advances the pointer forward by ten characters and places it between the (RET) and (LF) characters at the end of the second line. Note that the tab character preceding @R2 is also counted as a single character.

```
MOV     #VECT,R2(RET)(LF)
CLR     @R2(RET)(LF)
                ↑
```

Finally, to place the pointer after the C in the first line, use a −14J command. The J (Jump) command is explained in Section 5.6.3.2.

```
MOV     #VECT,R2(RET)(LF)
             ↑
CLR     @R2 (RET)(LF)
```

When you use line-oriented commands, the argument of the commands specifies the number of lines on which to operate. Because EDIT counts the line-terminating characters to determine the number of lines on which to operate, an argument, n, does not affect the same number of lines forward (positive) as it affects backward (negative). For example, the argument −1 applies to the line beginning with the first character following the second previous end-of-line and ending with the character preceding the pointer. The argument 1 in a line-oriented command, however, applies to the text beginning with the first character following the pointer and ending at the first end-of-line. Thus, if the pointer is at the center of the line, the argument −1 affects one and one-half lines backwards from the pointer and the argument 1 affects one-half line beyond the pointer.

For example, assume the buffer contains:

```
MOV     PC,R1(RET)(LF)
       ↑
ADD     #DRIV-.,R1(RET)(LF)
MOV     #VECT,R2(RET)(LF)
CLR     @R2(RET)(LF)
```

The command to advance the pointer one line (1A) causes the following change:

```
MOV     PC,R1(RET)(LF)
ADD     #DRIV-.,R1(RET)(LF)
↑
MOV     #VECT,R2(RET)(LF)
CLR     @R2(RET)(LF)
```

The command 2A moves the pointer over two (RET)(LF) combinations to precede the fourth line:

```
MOV     PC,R1(RET)(LF)
ADD     #DRIV-.,R1(RET)(LF)
MOV     #VECT,R2(RET)(LF)
CLR     @R2(RET)(LF)
↑
```

Assume the buffer contains:

```
MOV     PC,R1(RET)(LF)
ADD     #DRIV-.,R1(RET)(LF)
MOV     #VECT,R2(RET)(LF)
             ↑
CLR     @R2 (RET)(LF)
```

A command of - 1A moves the pointer back by one and one-half lines to precede the second line.

```
  MOV   PC,R1(RET)(LF)
 ↑ADD   #DRIV-.,R1(RET)(LF)
  MOV   #VECT,R2(RET)(LF)
  CLR   @R2(RET)(LF)
```

Now a command of - 1A moves the pointer back by only one line.

```
 ↑MOV   PC,R1(RET)(LF)
  ADD   #DRIV-.,R1(RET)(LF)
  MOV   #VECT,R2(RET)(LF)
  CLR   @R2(RET)(LF)
```

5.4.5 Command Repetition

You can execute portions of a command string more than once by enclosing the portion in angle brackets (<>) and preceding the left angle bracket with the number of iterations you desire. The syntax is:

```
n<command>
```

For example:

```
C1$C2$n<C3$C4$>C5$$
```

where

C	represents a command.
n	represents an iteration argument.

Commands C1 and C2 each execute once, then commands C3 and C4 execute n times. Finally, command C5 executes once and the command line is finished. The iteration argument (n) must be a positive number (in the range 1 through 16,383) and, if you do not specify it, it is assumed to be 1. If the number is negative or too large, an error message prints. You can nest iteration brackets up to 20 levels. EDIT checks command lines to make certain the brackets are correctly used and match prior to execution.

Essentially, enclosing a portion of a command string in iteration brackets and preceding it with an iteration argument (n) is equivalent to typing that portion of the string n times. For example:

```
*BGAAA$3<-DIB$-J>V$$
*BGAAA$-DIB$-J-DIB$-J-DIB$-JV$$
```

These two strings are equivalent.

Similarly, the following two strings are equivalent:

```
*B3<2<AD>V>$$
*BADADVADADVADADV$$
```

The following bracket structures are examples of legal usage:

```
<<><<<><>>>>
<<<>>><><>
```

The following bracket structures are examples of illegal combinations that will cause an error message since the brackets are not properly matched:

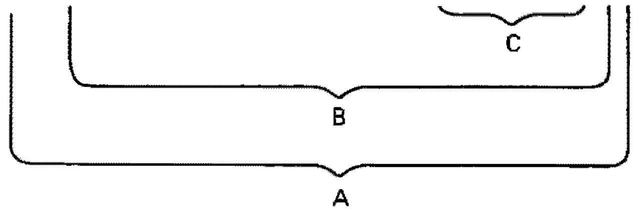
```
><><
<<<>>
```

During command repetition, execution proceeds from left to right until a right bracket is encountered. EDIT then returns to the last left bracket encountered, decreases the iteration counter, and executes the commands within the brackets. When the counter is decreased to 0, EDIT looks for the next iteration count to the left and repeats the same procedures. The overall effect is that EDIT works its way to the innermost brackets and then works its way back again. The most common use for iteration brackets is found in commands, such as Unsave (U), that do not accept repeat counts. For example:

```
*3<U>$$
```

Assume you want to read a file called SAMP (stored on device DK:), and you want to change the first four occurrences of the instruction MOV #200,R0 on each of the first five pages to MOV #244,R4. Enter the following command line:

```
*EBSAMP$5<N4<BGMOV  #200,R0$=J$3<G0$=C4>>>EX$$
```

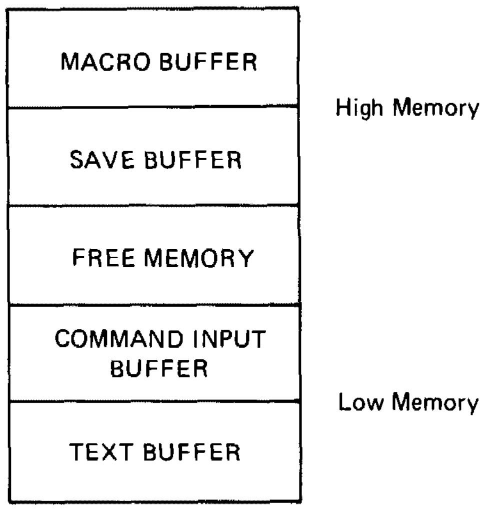


The command line contains three sets of iteration loops (A,B,C) and executes as follows:

Execution initially proceeds from left to right; EDIT opens the file SAMP for input and reads the first page into memory. EDIT moves the pointer to the beginning of the buffer and initiates a search for the character string MOV #200,R0. When it finds the string, EDIT positions the pointer at the end of the string, but the =J command moves the pointer back, so that it is positioned immediately preceding the string. At this point, execution has passed through each of the first two sets of iteration loops (A,B) once. The innermost loop (C) is next executed three times, changing the 0s to 4s. Control now moves back to pick up the second iteration of loop B, and again moves from left to right. When loop C has executed three times, control again moves back to loop B. When loop B has executed a total of four times, control moves back to the second iteration of loop A, and so forth, until all iterations have been satisfied.

5.5 MEMORY USAGE

The memory area used by the editor is divided into four logical buffers as follows:



The text buffer contains the current page of text you are editing, and the command input buffer holds the command you are currently typing at the terminal. If a command you are currently entering is within ten characters of exceeding the space available in the command buffer, the following message prints on the terminal.

```
?EDIT-W-Command buffer almost full
```

If you can complete the command within ten characters, you can finish entering the command; otherwise you should press the ESCAPE key twice to execute that portion of the command line already completed. The message prints each time you enter a character in one of the last ten spaces.

If you attempt to enter more than ten characters, EDIT prints the following message and aborts the command.

```
?EDIT-F-Command buffer full;no command(s) executed
```

This will never occur if you heed the preceding warning and terminate the command immediately.

The save buffer contains text stored with the Save (S) command, and the macro buffer contains the command string macro entered with the Macro (M) command. (Both commands are explained in Section 5.6.7.)

EDIT does not allocate space for the macro and save buffers until an M or S command executes. Once you enter an M or S command, a 0M or 0U (Unsave) command returns that space to the free area.

The size of each buffer automatically expands and contracts to accommodate the text you are entering; if there is not enough space available to accommodate required expansion of any of the buffers, EDIT prints the error message:

```
?EDIT-F-Insufficient memory
```

5.6 EDITING COMMANDS

This section describes the commands and procedures required to:

- Read text from the input files to the buffer
- Create a backup version of the file
- List the contents of the buffer on the terminal
- Move the reference pointer

- Locate specific characters or strings of characters within the text buffer
- Insert, relocate, or delete text in the buffer
- Close the output file
- Terminate the editing session.

The following sections are arranged, in order, by category of command function, as illustrated in Table 5-2.

5.6.1 File Open and Close Commands

You can use file open and close commands to:

- Open an existing file for input and prepare it for editing
- Open a file for output of newly created or edited text
- Open an existing file for editing and create a backup version of it
- Close an open output file.

5.6.1.1 Edit Read – The Edit Read (ER) command opens an existing file for input and prepares it for editing. *Only one file can be open for input at a time.*

The syntax of the command is:

```
ERdev:filnam.typ$
```

The string argument (dev:filnam.typ) is limited to 19 characters and specifies the file to be opened. If you do not specify a device, DK: is assumed. If a file is currently open for input, EDIT closes the file and opens the new one.

Edit Read does not input a page of text nor does it affect the contents of the other user buffers.

You can use Edit Read on a file that is already open to close that file for input and reposition EDIT at its beginning. The first Read command following any Edit Read command inputs the first page of the file.

```
*ERDT1:SAMP.MAC$$
```

This command string, for example, opens the file SAMP.MAC on device DT1: for input.

NOTE

If you enter EDIT with the monitor EDIT/INSPECT or EDIT/OUTPUT command, an Edit Read command is automatically performed on the file named in the EDIT command.

5.6.1.2 Edit Write – The Edit Write (EW) command opens a file for output of newly created or edited text. However, no text is output and the contents of the buffers are not affected. Only one file can be open for output at a time. EDIT closes any output files currently open and preserves any edits made to the file.

The syntax of the command is:

```
EWdev:filnam.typ[n]$
```

The string argument (dev:filnam.typ [n]) is limited to 19 characters and is the name you assign to the output file being opened. If you do not specify a device, DK: is assumed. [n] is an optional decimal number that represents the length of the file to be opened. Note that the square brackets [ ] are part of the argument, n. You must type them. If you do not specify [n], the default size will be used. That is, the system will choose the larger of 1) one-half the largest available space, and 2) the second largest available space. If this is not adequate for the output file size, you must close this file and open another when this one becomes full. You should use the [n] construction whenever there is doubt as to whether enough space is available on the device for one output file.

If a file with the same name already exists on the device, EDIT deletes the existing file when you type an Exit, End File, or another Edit Write command. EDIT prints the warning message:

```
    ?EDIT-W-Superseding existing file
```

The following command, for example, opens for output the file FILE.BAS on device DK: and allocates 11 blocks of space for it.

```
    *EWFILE.BAS[11]$$
```

NOTE

If you enter EDIT with the monitor EDIT/CREATE command, an Edit Write command is automatically performed on the file named in the EDIT command. If you enter EDIT with the monitor EDIT/OUTPUT command, an Edit Write is automatically performed on the file named with the /OUTPUT option.

5.6.1.3 Edit Backup – Use the Edit Backup (EB) command to open an existing file for editing and at the same time create a backup version of the file. EDIT closes any input and output file currently opened. No text is read or written with this command.

The syntax of the command is:

```
    EBdev:filnam.typ [n] $
```

The device designation, file name, and file type are limited to 19 characters. If you do not specify a device, DK: is assumed. [n] is optional and represents the length of the file to be opened; if you do not specify [n], the default size will be used. That is, the system will choose the larger of 1) one-half the largest available space, and 2) the second largest available space.

The file you indicate in the command line must already exist on the device you designate, since text will be read from this file as input. At the same time, EDIT opens an output file under the same file name and file type. When the output file is closed, EDIT renames the original file (used as input) with the current file name and a .BAK file type and deletes any previous file with this file name and a .BAK file type. EDIT closes the new output file and assigns it the name you specify in the EB command. This renaming of files takes place when an Exit, End File, or subsequent Edit Write or Edit Backup command executes. If you terminate the editing session with a CTRL/C command before the output file is closed, the new output file is not made permanent, and the renaming of the current version to .BAK does not take place.

```
    *EBSY:BAS1.MAC$$
```

This command opens BAS1.MAC on device SY:. When editing is complete, the old BAS1.MAC becomes BAS1.BAK, and the new file becomes BAS1.MAC. EDIT deletes any previous version of BAS1.BAK.

NOTE

In EB, ER, and EW commands, leading spaces between the command and the file name are not permitted because EDIT assumes the file name to be a text string. All dev:filnam.typ specifications for EB, ER, and EW commands conform to the RT-11 conventions for file naming and are identical to file names entered in command strings used with other system programs.

If you enter EDIT with an unqualified monitor EDIT command, an Edit Backup command is automatically performed on the file named in the EDIT command.

5.6.1.4 End File – The End File (EF) command closes the current output file and makes it permanent. You can use the EF command to create an output file from a section of a large input file or to close an output file that is full before you open another file. Modifiers are illegal with an EF command. Note that an implied EF command is included in EW and EB commands.

The syntax of the command is:

```
EF
```

Table 5-4 illustrates the relationship between the file open and close commands and the buffers and files themselves.

Table 5-4 EDIT Commands and File Status

Command	Input File	Text Buffer	Output File
ERXXX$	Opens XXX for input; closes existing input file, if any	Unchanged	Unchanged
EWXXX$	Unchanged	Unchanged	Opens XXX for output; closes existing output file, if any; performs .BAK renaming if EB is in effect
EBXXX$	Opens XXX for input; closes existing input file, if any	Unchanged	Opens a temporary file for output; closes existing output file, if any; performs .BAK renaming if EB is in effect
EF$	Unchanged	Unchanged	Closes output file; performs .BAK renaming if EB is in effect
EX$	Copies to output file	Copies to output file	Closes output file after copying complete; performs .BAK renaming if EB is in effect

5.6.2 File Input/Output Commands

You use file input/output commands to:

- Read text from an input file into the buffer
- Copy lines of text from the buffer into an output file
- Terminate the editing session.

5.6.2.1 Read – Before you can edit text, you must read the input file into the buffer. The Read (R) command reads a page of text from the input file (previously specified in an ER or EB command) and appends it to the current contents, if any, of the text buffer.

The command is:

```
R
```

No arguments are used with the R command. If text resides in the buffer prior to the R command, the pointer does not move; however, if no text resides in the buffer, the pointer is placed at the beginning of the buffer. EDIT transfers text to the buffer until one of the following conditions occurs:

1. A form feed character, signifying the end of the page, is encountered.
2. The text buffer is 500 characters from being full. (When this condition occurs, the Read command inputs up to the next carriage return/line feed combination, then returns to command mode. An asterisk prints as though the read were complete, but text will not have been fully input).
3. An end-of-file is encountered, (the ?EDIT-F-END OF INPUT FILE message prints when all text in the file has been read into memory and no more input is available).

The maximum number of characters that you can bring into memory with an R command depends on the system configuration and the memory requirements of other system components. EDIT prints an error message if the read exceeds the memory available or if no input is available.

The following example edits a file using the EB and R commands.

```
*EBSJK1.BAS$$
```

This command opens SJK1.BAS on DK: and permits modification.

```
*R/L$$
THIS IS PAGE ONE OF
FILE SJK1.BAS.
```

This command reads the first page of SJK1.BAS into the buffer. The pointer is placed at the beginning of the buffer. /L lists the contents of the buffer on the terminal beginning at the pointer and ending with the last character in the buffer.

5.6.2.2 Write – The Write (nW) command copies lines of text from the text buffer to the output file (as specified in the EW or EB command). The contents of the buffer are not altered and the pointer is left unchanged (unless an output error occurs).

NOTE

EDIT uses a system of intermediate buffers to store output before it actually writes the data to an output file. The Write command logically writes to the file, but actual output to a device does not occur until the intermediate buffer fills. When the editor closes a file (that is, after you issue an EF, EB, EX, or EW command), the editor writes from the buffer to the file and the file is complete. If the editor does not close a file (if you exit with CTRL/C and use the CLOSE command), it is possible that the output file will be missing the last 512 characters.

The syntax of the command is:

```
nW
```

The argument you supply with the W command determines the lines of text to copy. Table 5-5 lists the arguments for the W command and their effect.

Table 5-5 Write Command Arguments

Argument	Meaning
n	Writes n lines of text beginning at the pointer and ending with the nth end-of-line character to the output file.
-n	Writes n lines of text to the output file beginning with the first character on the -nth line and terminating at the pointer.
0	Writes to the output file the current line up to the pointer.
/	Writes to the output file the text between the pointer and the end of the buffer.

If the buffer is empty when the write executes, no characters are output.

The following examples illustrate the use of the W command.

```
*5W$$
```

This command writes the five lines of text following the pointer into the current output file.

```
*-2W$$
```

This command writes the two lines of text preceding the pointer into the current output file.

```
*B/W$$
```

This command writes the entire text buffer to the current output file.

NOTE

If an output file fills while a Write command is executing, EDIT prints the ?EDIT-F-OUTPUT FILE FULL message. In this case, EDIT positions the reference pointer after the last character it wrote successfully. You can then use the following recovery procedure:

1. Close the current output file. (EF command)
2. Open a new output file. (EW command)
3. Delete the characters just written by using -nD or -nK, where n is any arbitrary number that exceeds the number of lines or characters in the buffer.
4. Resume output.

5.6.2.3 Next – The Next (nN) command writes the contents of the text buffer to the output file, deletes the text from the buffer, and reads the next page of the input file into the buffer. The pointer is positioned at the beginning of the buffer. The syntax of the command is:

```
nN
```

If you specify the argument n with the Next command, the sequence is executed n times.

If EDIT encounters the end of the input file when trying to execute an N command, it prints ?EDIT-F-END OF INPUT FILE to indicate that no further text remains in the input file. Since the contents of the buffer has already been transferred to the output file, the buffer is empty.

Using the N command is a quick way to write edited text to the output file and set up the next page of text in the buffer. The N command functions as though it were a combination of the Write, Delete, Read, and Beginning commands. (Delete is a text modification command, described in Section 5.6.6.2; the Beginning command is a pointer relocation command, described in Section 5.6.3.1.) Using the N command with an argument is a convenient way to set up text in the buffer, if you already know its page location. The N command operates in a forward direction only; therefore, you cannot specify negative arguments with an N command.

In the following example, an N command copies an input file with more than one page of text to the output file.

```
*EBDK:TEST.MAC$$
```

This command opens the file TEST.MAC on device DK: and creates a new file entitled TEST.MAC for output.

```
*N/L$$
THIS IS PAGE ONE OF
FILE TEST.MAC.
```

This command reads the first page of the input file, TEST.MAC, into the buffer and lists the entire page on the terminal.

```
*N/L$$
?EDIT-F-End of input file
*
```

This command transfers the contents of the buffer to the output file, clears the buffer, and encounters the end of the file. Because it cannot complete the N sequence, EDIT prints ?EDIT-F-END OF INPUT FILE on the terminal. The buffer is empty and the entire input file has been written to the output file.

5.6.2.4 EXit – Type the Exit (EX) command to terminate an editing session. The Exit command does the following:

- Writes the text buffer to the output file
- Transfers the remainder of the input file to the output file
- Closes all open files
- Renames the backup file with a .BAK file type if an EB command is in effect
- Returns control to the monitor.

The command is:

```
EX
```

No arguments are accepted. Essentially, Exit copies the remainder of the input file into the output file and returns to the monitor. Exit is legal only when there is an output file open. If an output file is not open and you want to terminate the editing session, return to the monitor with CTRL/C.

NOTE

You must issue an EF or EX command in order to make an output file permanent. If you use CTRL/C to return to the monitor without issuing an EF command, the current output file will not be saved. (You can, however, make permanent that portion of the text file that has already been written out by using the monitor CLOSE command.)

An example of the contrasting uses of the EF and EX commands follows. Assume an input file, SAMPLE, contains several pages of text. The first and second pages of the file will be made into separate files called SAM1 and SAM2, respectively; the remaining pages of text will then make up the file SAMPLE. This can be done using these commands:

```
*EWSAM1$$
*ERSAMPLE$$
*RNEF$$
*EWSAM2$$
*NEF$$
*EWSAMPLE$EX$$
```

Note that the EF commands are not strictly necessary in this example since the EW command closes a currently open output file before opening another.

5.6.3 Pointer Relocation Commands

Pointer relocation commands allow you to change the current location of the reference pointer within the text buffer.

5.6.3.1 Beginning – The Beginning (B) command moves the current location of the pointer to the beginning of the text buffer.

The command is:

```
B
```

There are no arguments.

For example, assume the buffer contains:

```
MOVB  5(R1),@R2
ADD   R1,(R2)+
CLR   @R2
       ↑
MOVB  6(R1),@R2
```

The B command moves the pointer to the beginning of the text buffer.

```
*B$$
```

The text buffer now looks like this:

```
↑MOVB  5(R1),@R2
 ADD   R1,(R2)+
 CLR   @R2
 MOVB  6(R1),@R2
```

5.6.3.2 Jump – The Jump (nJ) command moves the pointer past the specified number of characters in the text buffer. The syntax of the command is:

nJ

Table 5-6 shows the arguments for the J command and their meanings.

Table 5-6 Jump Command Arguments

Argument	Meaning
(+ or -) n	Moves the pointer (forward or backward) n characters.
0	Moves the pointer to the beginning of the current line (equivalent to 0A).
/	Moves the pointer to the end of the text buffer (equivalent to /A).
=	Moves the pointer backward n characters, where n equals the length of the last text argument used.

Negative arguments move the pointer toward the beginning of the buffer; positive arguments move it toward the end. Jump treats carriage returns, line feeds, and form feed characters the same as any other character, counting one buffer position for each one.

The following commands illustrate the use of the J command.

```
*3J$$
```

This command moves the pointer ahead three characters.

```
*-4J$$
```

This command moves the pointer back four characters.

```
*B$GABC$=J$$
```

This command moves the pointer so that it immediately precedes the first occurrence of ABC in the buffer.

5.6.3.3 Advance – The Advance (nA) command is similar to the Jump command except that it moves the pointer a specific number of lines (rather than single characters) and leaves it positioned at the beginning of the line. The syntax of the command is:

nA

Table 5-7 lists the arguments for the A command and their meanings.

Table 5-7 Advance Command Arguments

Argument	Meaning
n	Moves the pointer forward n lines and positions it at the beginning of the nth line.
-n	Moves the pointer backward past n carriage return/line feed combinations and positions it at the beginning of the -nth line.
0	Moves the pointer to the beginning of the current line (equivalent to 0J).
/	Moves the pointer to the end of the text buffer (equivalent to /J).

Following are examples that use the A command.

```
*3A$$
```

This command moves the pointer ahead three lines.

Assume the buffer contains:

```
CLR     @R2
          ↑
```

The following command moves the pointer to the beginning of the current line:

```
*0A$$
```

Now the buffer looks like this:

```
CLR     @R2
↑
```

5.6.4 Search Commands

Use search commands to locate specific characters or strings of characters within the text buffer.

NOTE

Search commands always have positive arguments. They search ahead in the file. This means that you cannot search for a character string that has already been written to the output file. To do this, you must first close the currently open files (with EX) then edit the file that was just used for output (with EB).

5.6.4.1 Get – The Get (nG) command is the basic search command in EDIT. It searches the current text buffer for the nth occurrence of a specific text string starting at the current location of the pointer. If you do not supply the argument n, EDIT searches for the first occurrence of the text object. The search terminates when EDIT either finds the nth occurrence or encounters the end of the buffer. If the search is successful, EDIT positions the pointer to follow the last character of the text object. EDIT notifies you of an unsuccessful search by printing ?EDIT-F-SEARCH FAILED. In this instance, EDIT positions the pointer after the last character in the buffer.

The syntax of the command is:

```
nGtext$
```

The argument (n) must be positive. If you omit it, EDIT assumes it to be 1.

The text string can be any length and must immediately follow the G command. EDIT makes the search on the portion of the text between the pointer and the end of the buffer.

For example, assume the pointer is at the beginning of the buffer shown below.

```
 MOV    PC,R1
↑ADD    #DRIV-.,R1
 MOV    #VECT,R2
 CLR    @R2
 MOVB   5(R1),@R2
 ADD    R1,(R2)+
 CLR    @R2
 MOVB   6(R1),@R2
```

The following command searches for the first occurrence of the characters ADD following the pointer and places the pointer after the searched characters.

```
*GADD$$
```

Now the buffer looks like this:

```
MOV    PC,R1
ADD↑   #DRIV-.,R1
```

The next command searches for the third occurrence of the characters @R2 following the pointer and leaves the pointer immediately following the text object.

```
*3G@R2$$
```

The buffer is changed to:

```
ADD    R1,(R2)+
CLR    @R2↑
```

After successfully completing a search command, EDIT positions the pointer immediately following the text object. Using a search command in combination with =J places the pointer in front of the text object, as follows:

```
*GTEST$=J$$
```

This command combination places the pointer before TEST in the text buffer.

5.6.4.2 Find – The Find (nF) command starts at the current pointer location and searches the entire input file for the nth occurrence of the text string. If EDIT does not find the nth occurrence of the text string in the current buffer, it automatically performs a Next command and continues the search on the new text in the buffer. When the search is successful, EDIT leaves the pointer immediately following the nth occurrence of the text string. If the search fails (i.e., EDIT detects the end-of-file for the input file and does not find the nth occurrence of the text string), EDIT prints ?EDIT-F-SEARCH FAILED. In this instance, EDIT positions the pointer at the beginning of an empty text buffer. When you use the F command, EDIT deletes the contents of the buffer after writing it to the output file.

The syntax of the command is:

```
nFtext$
```

The argument (n) must be positive. EDIT assumes it to be 1 if you do not supply another value.

You can use an F command to copy all remaining text from the input file to the output file by specifying a non-existent text object. The Find command functions like a combination of the Get and Next commands.

The following example uses the F command.

```
*2FMOVB    6(R1),@R2$$
```

This command searches the entire input file for the second occurrence of the text string MOVB   6(R1),@R2. EDIT places the pointer following the text string. EDIT writes the contents of each unsuccessfully searched buffer to the output file.

5.6.4.3 Position – The Position (nP) command is identical to the find (F) command with one exception. The F command transfers the contents of the text buffer to the output file as each page is unsuccessfully searched, but the P command deletes the contents of the buffer after it is searched, without writing any text to the output file.

The syntax of the command is:

nPtext$

The argument (n) must be positive. If you omit it, EDIT assumes it to be 1.

The nP command searches each page of the input file for the nth occurrence of the text object starting at the pointer and ending with the last character in the buffer. If EDIT finds the nth occurrence, it positions the pointer following the text object, deletes all pages preceding the one containing the text object, and positions the page containing the text object in the buffer.

If the search is unsuccessful, EDIT clears the buffer and does not transfer any text to the output file. EDIT positions the pointer at the beginning of an empty text buffer.

The position command is a combination of the Get, Delete, and Read commands; it is most useful as a means of placing the pointer in the input file. For example, if your aim in the editing session is to create a new file from the second half of the input file, a position search saves time.

The following example uses the P command.

```
*P3$$
```

This command searches the input file for the first occurrence of the text object, 3. EDIT positions the pointer after the text object.

```
*0L$$
INPUT FILE PAGE 3
```

The command lists on the terminal the current line up to the pointer.

5.6.5 Text Listing Commands

5.6.5.1 List – The List (nL) command prints at the terminal lines of text as they appear in the buffer. The syntax of the command is:

nL

An argument preceding the L command indicates the portion of text to print. For example, the command, 2L, prints on the terminal the text beginning at the pointer and ending with the second end-of-line character. The pointer is not altered by the L command. Table 5-8 lists arguments and their effect upon the list command.

Table 5-8 List Command Arguments

Argument	Meaning
n	Prints at the terminal n lines beginning at the pointer and ending with the nth end-of-line character.
-n	Prints all characters beginning with the first character on the -nth line and terminating at the pointer.
0	Prints the current line up to the pointer. Use this command to locate the pointer within a line.
/	Prints the text between the pointer and the end of the buffer.

These examples illustrate the use of the L command.

```
*-2L$$
```

This command prints all characters starting at the second preceding line and ending at the pointer.

```
*4L$$
```

This line prints all characters beginning at the pointer and terminating at the 4th carriage return/line feed combination.

Assuming the pointer location is:

```
MOVB  5(R1),@R2
ADD↑  R1,(R2)+
```

The following command prints the previous one and one-half lines up to the pointer:

```
*-1L$$
```

The terminal output looks like this:

```
MOVB    5(R1),@R2
ADD
```

5.6.5.2 Verify – The Verify (V) command prints at the terminal the entire line in which the pointer is located. It provides a ready means of determining the location of the pointer after a search completes and before you give any editing commands. (The V command combines the two commands 0LL.) You can also type the V command after an editing command to allow proofreading of the results. No arguments are allowed with the V command. The location of the pointer does not change.

5.6.6 Text Modification Commands

You can use the following commands to insert, relocate, and delete text in the text buffer.

5.6.6.1 Insert – The Insert (I) command is the basic command for inserting text. EDIT inserts the text you supply at the location of the pointer, then places the pointer after the last character of the new text.

The syntax of the command is:

```
Itext$
```

No arguments are allowed with the insert command, and the text string is limited only by the size of the text buffer and the space available. All characters except ESCAPE are legal in the text string. ESCAPE terminates the text string.

NOTE

If you forget to type the I command, the text will be executed as commands.

EDIT automatically protects against overflowing the text buffer during an insert. If the I command is the first command in a multiple command line, EDIT ensures that there will be enough space for the insert to be executed at least once. If repetition of the command exceeds the available memory, an error message prints.

The following example illustrates the use of the I command.

```
*IMOV           #BUFF,R2
MOV             #LINE,R1
MOVB            -1(R2),R0$$
*
```

This command inserts the text at the current location of the pointer and leaves the pointer positioned after R0.

DIGITAL recommends that you insert large amounts of text into the file in small sections rather than all at once. This way, you are less vulnerable to loss of time and effort due to machine failure or human error. This is the recommended technique for inserting large amounts of text:

1. Open the file with the EB command
2. Insert or edit a few pages of text
3. Insert a unique text string (like ????) to mark your place
4. Use the Exit command to preserve the work you have done so far
5. Start again, using the F command to search for the unique string you used to mark your place
6. Delete your marker and continue editing.

By using this procedure, you reduce your loss (should there be a machine or human error) to the few pages of text on which you just worked.

5.6.6.2 Delete – The Delete (nD) command is a character-oriented command that deletes n characters in the text buffer beginning at the current location of the pointer. The syntax of the command is:

```
nD
```

If you do not specify n, EDIT deletes the character immediately following the pointer. Upon completion of the D command, EDIT positions the pointer immediately before the first character following the deleted text. Table 5-9 lists each argument for the D command and its effect.

Table 5-9 Delete Command Arguments

Argument	Meaning
n	Deletes n characters following the pointer. Places the pointer before the first character following the deleted text.
-n	Deletes n characters preceding the pointer. Places the pointer before the first character following the deleted text.
0	Deletes the current line up to the pointer. The position of the pointer does not change (equivalent to 0K).
/	Deletes the text between the pointer and the end of the buffer. Positions the pointer at the end of the buffer (equivalent to /K).
=	Deletes -n characters, where n equals the length of the last text argument used.

The following examples illustrate the use of the D command.

```
*-2D$$
```

This command deletes the two characters immediately preceding the pointer.

```
*B$FMOV R1$=D$$
```

This command string deletes the text string MOV R1. (=D in combination with a search command deletes the indicated text string.)

Assume the text buffer contains the following:

```
ADD   R1,(R2)+
CLR  ↑@R2
```

The following command deletes the current line up to the pointer:

```
*0D$$
```

The buffer now contains:

```
ADD   R1,(R2)+
↑@R2
```

5.6.6.3 Kill – The Kill (nK) command removes n lines of text (including the carriage return and line feed characters) from the page buffer, beginning at the pointer and ending with the nth end-of-line. The syntax of the command is:

```
nK
```

EDIT places the pointer at the beginning of the line following the deleted text. Table 5-10 describes each argument and its effect upon the Kill command.

Table 5-10 Kill Command Arguments

Argument	Meaning
n	Removes the character string (including the carriage return/line feed combination) beginning at the pointer and ending at the nth end-of-line.
-n	Removes the character string beginning at the nth end-of-line preceding the pointer and ending at the pointer. Thus, if the pointer is at the center of a line, the modifier -1 deletes one and one-half lines preceding it.
0	Removes the current line up to the pointer (equivalent to 0D).
/	Removes the characters beginning at the pointer and ending with the last line in the text buffer (equivalent to /D).

The following examples use the K command.

```
*2K$$
```

This command deletes lines starting at the current location of the pointer and ending at the second carriage return/ line feed combination.

Assume the text buffer contains the following:

```
ADD     R1,(R2)+
CLR↑    @R2
MOVB    6(R1),@R2
```

This command removes the characters beginning at the pointer and ending with the last line in the text buffer:

```
*/K$$
```

The buffer now contains:

```
ADD     R1,(R2)+
CLR↑
```

Kill and Delete commands perform the same function, except that Kill is line-oriented and Delete is character-oriented.

5.6.6.4 Change – The Change (nC) command changes a specific number of characters following the pointer. The syntax of the command is:

```
nCtext
```

A C command is equivalent to a Delete command followed by an Insert command. You must insert a text object following the nC command. Table 5-11 lists each argument and its effect upon the C command.

Table 5-11 Change Command Arguments

Argument	Meaning
n	Replaces n characters following the pointer with the specified text. Places the pointer after the inserted text.
-n	Replaces n characters preceding the pointer with the specified text. Places the pointer after the inserted text.
0	Replaces the current line up to the pointer with the specified text. Places the pointer after the inserted text (equivalent to 0X).
/	Replaces the text beginning at the pointer and ending with the last character in the buffer. Places the pointer after the inserted text (equivalent to /X).
=	Replaces -n characters with the indicated text string, where n represents the length of the last text argument used.

The size of the text is limited only by the size of the text buffer and the space available. All characters are legal except ESCAPE, which terminates the text string.

If the C command is to be executed more than once (i.e., it is enclosed in angle brackets) and if there is enough space available for the command to be entered, it will be executed at least once (provided it appears first in the command string). If repetition of the command exceeds the available memory, an error message prints.

The following examples use the C command.

```
*5C#VECT$$
```

This command replaces the five characters to the right of the pointer with #VECT.

Assume the text buffer contains the following:

```
CLR     @R2
MOV↑    5(R1),@R2
```

The next command replaces the current line up to the pointer with the specified text.

```
*0CADDB$$
```

The buffer now contains:

```
CLR     @R2
ADDB↑   5(R1),@R2
```

You can use =C with a Get command to replace a specific text string. Here is an example:

```
*GFIFTY:$=CFIVE:$
```

This command finds the occurrence of the text string FIFTY: and replaces it with the text string FIVE:.

5.6.6.5 eXchange – The eXchange (nX) command is similar to the change command except that it changes lines of text, instead of a specific number of characters. The syntax of the command is:

```
nXtext
```

The nX command is identical to an nK command followed by an Insert command. Table 5-12 lists each argument and its effect upon the eXchange command.

Table 5-12 eXchange Command Arguments

Argument	Meaning
n	Replaces n lines including the carriage return and line feed characters following the pointer. Places the pointer after the inserted text.
-n	Replaces n lines including the carriage return and line feed characters preceding the pointer. Positions the pointer after the inserted text.
0	Replaces the current line up to the pointer with the specified text. Positions the pointer after the inserted text (equivalent to 0C).
/	Replaces the text beginning at the pointer and ending with the last character in the buffer with the specified text (equivalent to /C). Positions the pointer after the inserted text.

All characters are legal in the text string except ESCAPE, which terminates the text.

If the X command is enclosed within angle brackets to allow more than one execution, and if there is enough memory space available for the X command to be entered, EDIT executes it at least once (provided it is first in the command string). If repetition of the command exceeds the available memory, an error message prints.

The following example uses the X command.

```
*2XADD   R1,(R2)+
 CLR     @R2
 $$
*
```

This command exchanges the two lines to the right of the pointer with the text string.

5.6.7 Utility Commands

During the editing session, you can store text in external buffers and subsequently restore this text when you need it later on in the editing session. The following sections describe the commands that perform this function.

5.6.7.1 Save – The Save (nS) command lets you store text in an external buffer called a save buffer (described previously in Section 5.5), and subsequently insert it in several places in the text.

The syntax of the command is:

```
nS
```

The Save command copies n lines, beginning at the pointer, into the save buffer. The S command operates only in the forward direction; therefore, you cannot use a negative argument. The Save command destroys any previous contents of the save buffer; however, EDIT does not change the location of the pointer or the contents of the text buffer.

If you specify more characters than the save buffer can hold, EDIT prints ?EDIT-F-INSUFFICIENT MEMORY. None of the specified text is saved.

For example, assume the text buffer contains the following assembly-language subroutine:

```
; SUBROUTINE MSGTYP
; WHEN CALLED, EXPECTS R0 TO POINT TO AN
; ASCII MESSAGE THAT ENDS IN A ZERO BYTE,
; TYPES THAT MESSAGE ON THE USER TERMINAL

MSGTYP:     TSTB     (R0)              ; DONE?
            BEQ      MDONE             ; YES-RETURN
MLOOP:      TSTB     @#177564          ; NO-IS TERMINAL READY?
            BPL      MLOOP             ; NO-WAIT
            MOVB     (R0)+,@#177566    ; YES PRINT CHARACTER
            BR       MSGTYP            ; LOOP
MDONE:      RTS      PC                ; RETURN
```

The following command stores the entire subroutine in the save buffer (assuming the pointer is at the beginning of the buffer):

```
*12S$$
```

You can insert the contents of the save buffer into a program whenever you choose by using the Unsave command.

5.6.7.2 Unsave – The Unsave (U) command inserts the entire contents of the save buffer into the text buffer at the pointer and leaves the pointer positioned following the inserted text. You can use the U command to move blocks of text or to insert the same block of text in several places. Table 5-13 lists the U commands and their meanings.

Table 5-13 U Command and Arguments

Command	Meaning
U	Inserts the contents of the save buffer into the text buffer.
0U	Clears the save buffer and reclaims the area for text.

The only argument the U command accepts is 0.

The contents of the save buffer are not destroyed by the Unsave command (only by the 0U command) and can be unsaved as many times as desired. If the Unsave command causes an overflow of the text buffer, the ?EDIT-F-INSUFFICIENT MEMORY error message prints, and the command does not execute.

For example:

```
*U$$
```

This command inserts the contents of the save buffer into the text buffer.

5.6.7.3 Macro – The Macro (M) command inserts a command string into the EDIT macro buffer. Table 5-14 lists the M commands and their meanings.

Table 5-14 M Command and Arguments

Command	Meaning
M/command string/	Stores the command string in the macro buffer.
0M or M//	Clears the macro buffer and reclaims the area for text.

The slash (/) represents the delimiter character. The delimiter is always the first character following the M command, and can be any character that does not appear in the macro command string itself.

Starting with the character following the delimiter, EDIT places the macro command string characters into its internal macro buffer until the delimiter is encountered again. At this point, EDIT returns to command mode. The macro command does not execute the macro string; it merely stores the command string so that the Execute Macro (EM) command can execute later. The Macro command does not affect the contents of the text or save buffers.

All characters except the delimiter are legal macro command string characters, including single ESCAPEs to terminate text commands. All commands, except the M and EM commands, are legal in a command string macro.

In addition to using the 0M command, you can type the M command immediately followed by two identical characters (assumed to be delimiters) and two ESCAPE characters to clear the macro buffer.

The following examples illustrate the use of the M command.

```
*M//$$
```

This command clears the macro buffer.

```
*M/GRO$-C1$/$$
```

This command stores a macro to change R0 to R1.

NOTE

Be careful to choose infrequently-used characters as macro delimiters; use of frequently-used characters can lead to inadvertent errors. For example:

```
*M GMOV RO$=CADD R1$ $$
?EDIT-F-No file open for input
```

In this case, it was intended that the macro be GMOV R0$=CADD R1$ but since the delimiter character (the character following the M) is a space, the space following MOV is used as the second delimiter, terminating the macro. EDIT then returns an error when it interprets the R as a Read command.

5.6.7.4 Execute Macro – The Execute Macro (nEM) command executes the command string previously stored in the macro buffer by the M command.

The syntax of the command is:

```
nEM
```

The argument (n) must be positive. The macro is executed n times and returns control to the next command in the original command string.

The following example uses the EM command.

```
*M/BGRO$-C1$/$$
*B1000EM$$
?EDIT-F-Search failed
*
```

This command sequence executes the macro stored in the previous example. EDIT prints an error message when it reaches the end of the buffer. (This macro changes all occurrences of R0 in the text buffer to R1.)

```
*IMOV PC,R1$2EMICLR @R2$$
*
```

This command inserts MOV PC,R1 into the text buffer, then executes the command in the macro buffer twice before inserting CLR @R2 into the text buffer.

5.6.7.5 Edit Version – The Edit Version (EV) command displays the version number of the editor in use on the console terminal.

The command is:

```
EV
```

This example displays the running version of EDIT:

```
*EV$$
V03.36
*
```

5.6.7.6 Upper- and Lower-Case Commands – If you have an upper- and lower-case terminal as part of your hardware configuration, you can take advantage of the upper- and lower-case capability of this terminal. Two editing commands, EL and EU, permit this.

When the editor is first started with the EDIT command, upper-case mode is assumed; all characters you type are automatically translated to upper case. To allow processing of both upper- and lower-case characters, enter the Edit Lower command. For example:

```
*EL$$
*i You can enter text and commands in UPPER and lower case.$$
*
```

The editor now accepts and echoes upper- and lower-case characters received from the keyboard and prints text on the terminal in upper and lower case.

To return to upper-case mode, use the Edit Upper command:

```
*EU$$
```

Control also reverts to upper-case mode upon exit from the editor (with EX or CTRL/C).

Note that when you issue an EL command, you can enter EDIT commands in either upper or lower case. Thus, the following two commands are equivalent:

```
*GTEXT$=Cnew text$V$$

*gTEXT$=cnew text$v$$
```

The editor automatically translates (internally) all commands to upper case independent of EL or EU.

NOTE

When you use EDIT in EL mode, make sure that text arguments you specify in search commands have the proper case. The command GTeXt$, for example, will not match TEXT, text, or any combination other than TeXt.

5.7 THE DISPLAY EDITOR

In addition to all functions and commands mentioned thus far, the editor can use VT-11 and VS-60 display hardware that may be part of the system configuration (GT40, GT44, DECLAB 11/40, DECLAB 11/34). The most obvious feature is the ability to use the display screen rather than the console terminal for printing all terminal input and output. Another feature is that the top of the display screen functions like a window into the text buffer. When all the features of the display editor are in use, a 12 in. screen displays text as shown in Figure 5-1.

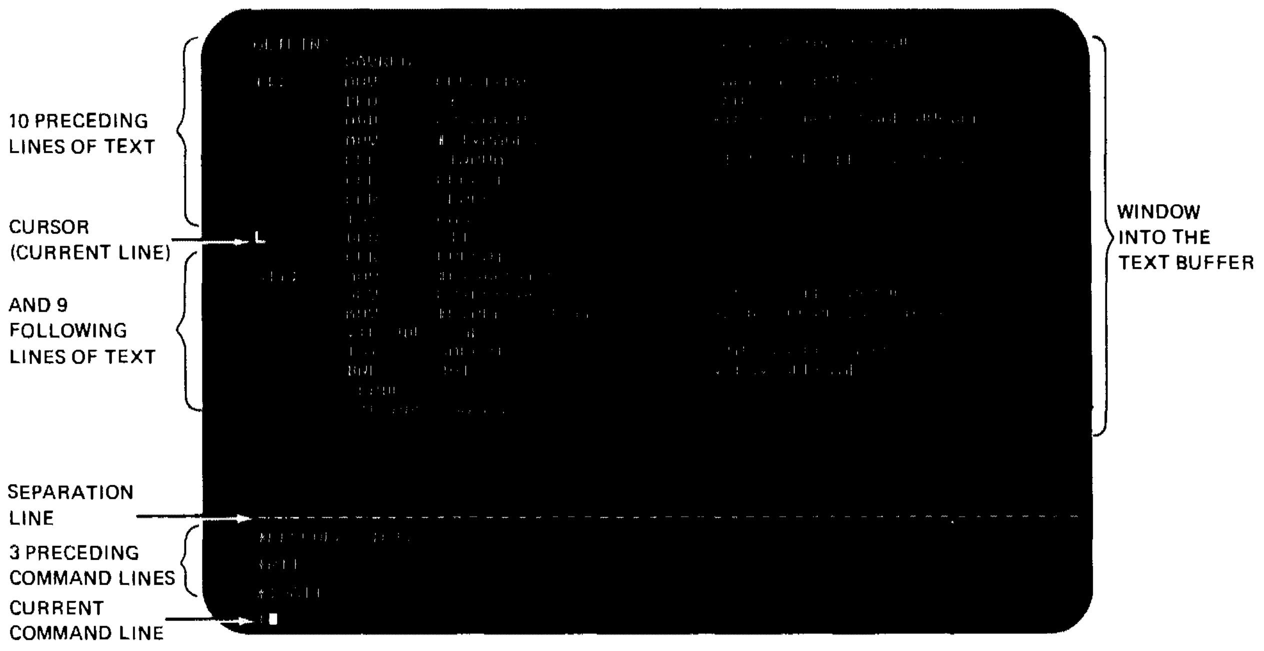

10 PRECEDING LINES OF TEXT
CURSOR (CURRENT LINE)
AND 9 FOLLOWING LINES OF TEXT
SEPARATION LINE
3 PRECEDING COMMAND LINES
CURRENT COMMAND LINE
WINDOW INTO THE TEXT BUFFER

Figure 5-1 Display Editor Format, 12 in. Screen

The major advantage is that you can now see immediately where the pointer is. The pointer appears between characters on the screen as a blinking L-shaped cursor and you can see it easily. Remember that pressing the (RET) key causes both a carriage return and a line feed character to be inserted into the text. Note that if the pointer is placed between a carriage return and line feed, it appears in an inverted position at the beginning of the next line.

In addition to displaying the current line (the line containing the cursor), the 15 lines of text preceding the current line and the 14 lines following it are also in view on a 17 in. screen. Each time you execute a command string (with a double ESCAPE), EDIT refreshes this portion of the screen so that it reflects the results of the commands you just performed.

The lower section of the 17 in. screen contains eight lines of editing commands. The command line you are currently entering is last, preceded by the most recent command lines. A horizontal line of dashes separates this section from the text portion of the screen. As you enter new command lines, previous command lines scroll upward off the command section so that only eight command lines are ever in view.

A 12 in. screen displays 20 lines of text and 4 command lines.

5.7.1 Using the Display Editor

The display features of the editor are automatically invoked whenever the system scroller is in use (a monitor GT ON command is in effect) and you start the editor. However, if the system does not contain display hardware, the display features are not enabled.

Providing that the system does contain display hardware and that you wish to employ the screen during the editing session, you can activate it in one of two ways, whether or not the display is in use. All editing commands and functions previously discussed in this chapter are valid for use.

1. If the scroller is in use (the GT ON monitor command is in effect), EDIT recognizes this and automatically uses the screen for display of text and commands. However, it rearranges the scroller so that a window into the text buffer appears in the top two-thirds of the screen, while the bottom third displays command lines. This arrangement is shown in Figure 5-1.

   You can use the Edit Console command to return the scroller to its normal mode so that text and commands use the full screen, and the window is eliminated.

   The command is:

   ```
   EC
   ```

   This example uses the EC command:

   ```
   *BAEC2L$$
   ```

   This command lists the second and third lines of the current buffer on the screen; there is no window into the text buffer at this point.

   EDIT ignores subsequent EC commands if the window into the text buffer is not being displayed.

   To recall the window, use the Edit Display command:

   ```
   ED
   ```

   The screen is again arranged as shown in Figure 5-1.

2. Assume the scroller is not is use (the GT ON command is not in effect). When you call EDIT with the .EDIT command, an asterisk appears on the console terminal. Use the ED command at this time to provide the window into the text buffer; however, commands continue to be echoed to the console terminal.

   When you use ED in this case, it must be the first command you issue; otherwise, it becomes an illegal command (since the memory used by the display buffer and code, amounting to over 600 words, is reclaimed as working space). You cannot use the display again until you load a fresh copy of EDIT.

   While the display of the text window is active, EDIT ignores ED commands.

   Typing the EC command clears the screen and returns all output to the console terminal.

NOTE

After an editing session that uses the ED command is over, clear the screen by typing the EC command or by returning to the monitor and using the monitor RESET command. Failure to do this may cause unpredictable results.

5.7.2 Setting the Editor to Immediate Mode

An additional mode is available in EDIT to provide easier and faster interaction during the editing session. This mode is called immediate mode and combines the most-used functions of the text and command modes – namely, repositioning the pointer and deleting and inserting characters.

You can only use immediate mode when the VT-11 display hardware is active and the editor is running. Enter it by typing two ESCAPEs (only) in response to the command mode asterisk:

```
    *$$
```

The editor responds by echoing an exclamation point on the screen.

```
!
```

The exclamation character remains on the screen as long as control is in immediate mode.

Once you enter immediate mode, you can use only the commands in Table 5-15. Any other commands or characters are treated as text to be inserted. None of these commands echoes, but the text appearing on the screen is constantly refreshed and updated during the editing process.

To return control to the display editor's normal command mode at any time while in immediate mode, type a single ESCAPE. The editor responds with an asterisk and you can proceed using all normal editing commands. (Immediate mode commands you type at this time will be accepted as command mode input characters.) To return control to the monitor while in immediate mode, type ESCAPE to return to command mode, then type CTRL/C followed by two ESCAPEs.

Table 5-15 Immediate Mode Commands

Command	Meaning
CTRL/N	Advances the pointer (cursor) to the beginning of the next line (equivalent to A).
CTRL/G	Moves the pointer (cursor) to the beginning of the previous line (equivalent to -A).

(Continued on next page)

Table 5-15 (Cont.) Immediate Mode Commands

Command	Meaning
CTRL/D	Moves the pointer (cursor) forward by one character (equivalent to J).
CTRL/V	Moves the pointer (cursor) back by one character (equivalent to -J).
RUBOUT or DELETE	Deletes the character immediately preceding the pointer (cursor) (equivalent to -D).
ESCAPE or ALTMODE	Single character returns control to command mode; double character directs control to immediate mode.
Any character other than those above	Inserts the character as text positioned immediately before the pointer (cursor) – equivalent to I.

5.8 EDIT EXAMPLE

The following example illustrates the use of some of the EDIT commands to change a program stored on the device DK:. Sections of the terminal output are coded by letter and corresponding explanations follow the example.

```
A {  EDIT/OUTPUT:TEST2.MAC TEST1.MAC
    *R$$
  { *//L$$
    ;TEST PROGRAM

    START:  MOV     #1000,SP        ;INITIALIZE STACK
            MOV     #MSG,RO         ;POINT RO TO MESSAGE
B {         JSR     PC,MSGTYP       ;PRINT IT
            HALT                    ;STOP
    MSG:    .ASCII/IT WORKS/
            .BYTE 15
            .BYTE 12
            .BYTE 0
C {
    *B$1J$5D$$
  { *GPROGRAM$$
D { *0L$$
  { ;PROGRAM*I TO TEST SUBROUTINE MSGTYP, TYPES
E { ;"THE TEST PROGRAM WORKS"
    ;ON THE TEMI\IM\RMINAL$$
F {  G.ASCII/$$
    *8CTHE TEST PROGRAM WORKS$$
  { *F.BYTE^X
G { *G.BYTE 0$V$$
            .BYTE 0
```

```
   *I
           .END
   $B/L$$
   ;PROGRAM TO TEST SUBROUTINE MSGTYP. TYPES
   ;"THE TEST PROGRAM WORKS"
   ;ON THE TERMINAL

   START:  MOV     #1000,SP          ;INITIALIZE STACK
           MOV     #MSG,R0           ;POINT R0 TO MESSAGE
H          JSR     PC,MSGTYP         ;PRINT IT
           HALT                      ;STOP
   MSG:    .ASCII/THE TEST PROGRAM WORKS/
           .BYTE 15
           .BYTE 12
           .BYTE 0
           .END

   *EX$$
I
   .
```

A Calls the EDIT program and prints *. The input file is TEST1.MAC; the output file is TEST2.MAC. Reads the first page of input into the buffer.

B Lists the buffer contents.

C Places the pointer at the beginning of the buffer. Advances the pointer one character (past the ;) and deletes the TEST.

D Positions the pointer after PROGRAM and verifies the position by listing up to the pointer.

E Inserts text. Uses RUBOUT to correct typing error.

F Searches for .ASCII/ and changes IT WORKS to THE TEST PROGRAM WORKS.

G Types CTRL/X to cancel the P command. Searches for .BYTE 0 and verifies the location of the pointer with the V command.

H Inserts text. Returns the pointer to the beginning of the buffer and lists the entire contents of the buffer.

I Closes the input and output files after copying the current text buffer as well as the rest of the input file into the output file. EDIT returns control to the monitor.

5.9 EDIT ERROR CONDITIONS

The editor prints an error message whenever a detectable error condition occurs. EDIT checks for three general types of error conditions: 1) syntax errors, 2) execution errors and, 3) macro execution errors. This section describes the error message form for each type of error condition.

Before it executes any commands, EDIT first scans the entire command string for errors in command syntax, such as illegal arguments or an illegal combination of commands. If the editor finds an error of this type, it prints a message of this form:

```
?EDIT-F-Message; no command(s) executed
```

You should retype the command.

If a command string is syntactically correct, EDIT begins execution. Execution errors, such as buffer overflow or input and output errors, can still occur. In this case, EDIT prints a message of the form:

```
?EDIT-F-Message
```

EDIT executes all commands preceding the one in error. It does not execute the command in error or any commands that follow it.

When an error occurs during execution of a macro, EDIT prints a message of the form:

```
?EDIT-F-Message in macro; no command(s) executed
or
?EDIT-F-Message in macro
```

Most errors are syntax errors. These are usually easy to correct before execution.

The *RT-11 System Message Manual* contains a complete list of the EDIT error messages, along with recommended corrective action for each error.

PART IV

UTILITY PROGRAMS

The following chapters describe in detail the system programs available to you as an RT-11 user. You can take advantage of nearly all of the capabilities of the RT-11 system by using the keyboard monitor commands, which are described in Chapter 4. However, it is the system utility programs (and not the monitor itself) that actually perform many of the system's functions. When you issue a monitor COPY command, for example, it is a system utility program (PIP, DUP, or FILEX, in this case) that performs the copy operation. Part IV of this manual, Utility Programs, explains how to carry out utility operations, those not performed directly by the monitor, by running a specific system utility program instead of using the keyboard monitor commands. It is not necessary to have an understanding of the material contained in Part IV in order to use the RT-11 system. However, the information in Part IV may be of interest to you if you have experience with a previous version of RT-11, or if you are a systems programmer and need to perform certain functions with the utility programs that are not available with the keyboard monitor commands. Note that the syntax the Command String Interpreter requires for input and output specifications is different from the syntax you use to issue a keyboard monitor command. Chapter 6, the Command String Interpreter, describes the general syntax of the specification string that the system utility programs accept, and explains certain conventions and restrictions. Read this chapter carefully before you use any of the system utility programs directly, and bear in mind that there are many differences between issuing a monitor command and running a utility program. Chapters 7 through 15 describe the system utility programs themselves.

CHAPTER 6
COMMAND STRING INTERPRETER

The Command String Interpreter (CSI) is the part of the RT-11 system that accepts a line of ASCII input, usually from you at the console terminal, and interprets it as a string of input specifications, output specifications, and options for use by a system utility program. To call a utility program, respond to the dot (.) printed by the keyboard monitor by typing R followed by a program name and a carriage return. This example shows how to call the directory program (DIR):

```
.R DIR
```

The Command String Interpreter prints an asterisk (*) at the left margin on the terminal, indicating that it is ready to accept a list of specifications and options. The following section describes the syntax of the specifications and options you can enter.

6.1 COMMAND STRING INTERPRETER SYNTAX

Once you have started a system program, you must enter the appropriate information before any operation can be performed. You type a specification string in response to the prompting asterisk. The specifications are in the following general syntax:

    output-filespecs/option=input-filespecs/option

(A few system programs – EDIT and PATCH, for example – require you to enter this information slightly differently. Complete instructions are provided in the appropriate chapters.)

In all cases, the syntax for output-filespec is:

    dev:filnam.typ[n], . . . dev:filnam.typ[n]

The syntax for input-filespec is:

    dev:filnam.typ , . . . dev:filnam.typ

The syntax for /option is:

    /o:oval or /o:dval.

where

dev: represents either a logical device name or a physical device name, which is a 2- or 3-character name from Table 3-1.

If you do not supply a device name, the system uses device DK:. DK:, or whatever device you specify for the first file in a list of input or output files, applies to all the files in that input or output list, until you supply a different device name. For example:

```
*DT1:FIRST.OBJ,LP:=TASK.1,RK1:TASK.2,TASK.3
```

	This command is interpreted as follows: `*DT1:FIRST.OBJ,LP:=DK:TASK.1,RK1:TASK.2,RK1:TASK.3` File FIRST.OBJ is stored on device DT1:. File TASK.1 is stored on default device DK:. Files TASK.2 and TASK.3 are stored on device RK1:. Notice that file TASK.1 is on device DK:. It is the first file in the input file list and the system uses the default device DK:. Device DT1: applies only to the file on the output side of the command.
filnam.typ	represents the name of a file (consisting of one to six alphanumeric characters followed optionally by a dot and a zero to three character file type). No spaces or tabs are allowed in the file name or file type. As many as three output and six input files are allowed.
[n]	is an optional declaration of the number of blocks (n) you need for an output file; n is a decimal number (<65,535) enclosed in square brackets immediately following the output filnam.typ to which it applies.
/o:oval or /o:dval.	represents one or more options whose functions vary according to the program you are using (refer to the option table in the appropriate chapter); oval is either an octal number or one to three alphanumeric characters (the first of which must be alphabetic) that the program converts to Radix-50 characters; dval. is a decimal number followed by a decimal point. This manual uses the /o:oval construction throughout, except for the keyboard monitor commands, where all values are interpreted as decimal (unless indicated otherwise) and the decimal point after a value is not necessary. However, the /o:dval. format is always valid. Generally, these options and their associated values, if any, should follow the device and file name to which they apply. If the same option is to be repeated several times with different values (e.g., /L:MEB/L:TTM/L:CND) you can abbreviate the line as /L:MEB:TTM:CND. You can mix octal, Radix-50, and decimal values.
=	If required, is a delimiter that separates the output and input fields. You can use the < sign in place of the = sign. You can omit the separator entirely if there are no output files.

6.2 PROMPTING CHARACTERS

Table 6-1 summarizes the characters RT-11 prints either to indicate that the system is awaiting your response or to specify which job (foreground or background) is producing output.

Table 6-1 Prompting Characters

Character	Explanation
.	The keyboard monitor is waiting for a command.
^	When the console terminal is being used as an input file, the uparrow (or circumflex) prompts you to enter information from the keyboard. Typing a CTRL/Z marks the end-of-file.
>	The > character identifies (only if a foreground job is active) which job, foreground or background, is producing the output that currently appears on the console terminal. Each time output from the background job is to appear, B> prints first, followed by the output. If the foreground job is to print output, F> prints first.
*	The current system utility program is waiting for a line of specifications and options.

CHAPTER 7
PERIPHERAL INTERCHANGE PROGRAM (PIP)

The peripheral interchange program (PIP) is a file transfer and file maintenance utility program for RT-11. You can use PIP to transfer files between any of the RT-11 devices (listed in Table 3-1) and to merge, rename, and delete files.

7.1 CALLING AND USING PIP

To call PIP from the system device, respond to the dot (.) printed by the keyboard monitor by typing:

```
R PIP (RET)
```

The Command String Interpreter prints an asterisk at the left margin of the terminal and waits for you to type a command string. If you type only a carriage return at this point, PIP prints its current version number and prompts you again for a command string. You can type CTRL/C to halt PIP and return control to the monitor when PIP is waiting for input from the console terminal. You must type two CTRL/Cs to abort PIP at any other time. To restart PIP, type R PIP or REENTER followed by a carriage return in response to the monitor's dot. Chapter 6, Command String Interpreter, describes the general syntax of the command line that PIP accepts. You can type as many as six input file names, but only one output file name is allowed. You can put command options at the end of the command string or type them after any file name in the string. Operations involving magtape are an exception to this rule because the /M option is device dependent, and has a different meaning when you specify it on the input or output side of a command line. Type any number of options in a command line, as long as only one operation (insertion, deletion, etc.) is represented. You can, however, combine copy and delete operations on one line. If you specify a command involving random access devices for which the output specification is the same as the input specification, PIP does not move any files. However, it can change the creation dates on the files if you use /T, or it can rename the files if you use /R.

Since PIP performs file transfers for all RT-11 data formats (ASCII, object, and image), it does not assume file types for either input or output files. You must explicitly specify all file types, where file types are applicable.

On random access devices, such as disks and DECtape, PIP operations retain a file's creation date. If the file's creation date is 0, PIP gives it the current system date. However, in transfers to and from magtape and cassette, PIP always gives files the current system date.

You can use all variations of the wildcard construction for the input file specifications in the PIP command line (Section 4.3 describes wildcard usage). Output file specifications cannot contain embedded wildcards. If you use any wild character in an input file specification, the corresponding output file name or file type must be an asterisk. The concatenate copy operation is an exception to this rule because it does not allow wildcards in the output specification. These two lines are examples of wildcard usage:

```
**.B=A%B.MAC

*B.*=A%B.MAC
```

The first command string is legal. The second generates an error message because the file name field of the input specification contains a wildcard and the output specification is not *.

The following command, for example, deletes all files with the file type .BAK (regardless of their file names) from device DK:.

```
**.BAK/D
```

The next command renames all files with a .BAK file type (regardless of file names) so that these files now have a .TST file type (maintaining the same file names).

```
**.TST=*.BAK/R
```

PIP performs operations on files in the order in which you specify them in the command string. However, if the specification contains a wildcard, PIP operates on the files in the order in which they appear in the device directory. PIP ignores system files with the file type .SYS unless you also use the /Y option. PIP prints the error message ?PIP-W-NO.SYS ACTION if you omit the /Y option on a command that would operate on .SYS files.

PIP ignores all files with the file type .BAD unless you explicitly specify both the file name and file type in the command string. PIP does not print a warning message when it does not include .BAD files in an operation. Because of the way PIP handles .BAD files, you cannot use a wildcard (*.BAD) to perform any operation on them.

This example transfers all files, including system files, (regardless of file name or file type) from device DK: to device RK1:. It does not transfer .BAD files.

```
*RK1:*.*/Y=*.*
```

7.2 PIP OPTIONS

Certain options permit you to perform various operations with PIP. Table 7-1 summarizes the operations that PIP performs. If you do not specify an option, PIP assumes that the operation is a file transfer in image mode. The following sections are organized by function. Operations involving magtape and cassette are discussed first because these operations are treated uniquely by PIP. The other functions (copy, delete, rename, log, and query) are described next. Explanations of the options are arranged alphabetically in the discussions of the appropriate functions.

Table 7-1 PIP Options

Option	Section	Explanation
/A	7.2.2.2	Copies files in ASCII mode, ignoring nulls and rubouts. It converts to 7-bit ASCII and treats CTRL/Z (32 octal) as the logical end-of-file on input (the default copy mode is image).
/B	7.2.2.3	Copies files in formatted binary mode (the default copy mode is image).
/C	7.2.2.4	Can be used with another option. It causes PIP to include only files with the current date in the specified operation.
/D	7.2.3	Deletes input files from a specific device. Note that PIP does not automatically query before it performs the operation. If you combine /D with a copy operation, PIP performs the delete operation after the copy completes. This option is illegal in an input specification with magtape.
/G	7.2.2.5	Ignores any input errors that occur during a file transfer and continues copying.
/K:n	7.2.2.6	Makes n copies of the output files to LP:, TT:, or PC:.
/M:n	7.2.1	You can use /M:n when I/O transfers involve either cassette or magtape. (See Section 7.2.1, Operations Involving Magtape or Cassette.)

(Continued on next page)

Table 7-1 (Cont.) PIP Options

Option	Section	Explanation
/N	7.2.2.7	Does not copy or rename a file if a file with the same name exists on the output device. This option protects you from accidentally deleting a file. This option is illegal for magtape and cassette in the output specification.
/O	7.2.2.8	Deletes a file on the output device if you copy a file with the same name to that device. The delete operation occurs before the copy operation. This option is illegal for magtape and cassette in the output specification.
/P	7.2.2.9	Copies or deletes all files except those you specify.
/Q	7.2.6	Use only with another operation. The /Q option causes PIP to print the name of each file to be included in the operation you specify. You must respond with a Y to include a particular file.
/R	7.2.4	Renames the file you specify. This operation is illegal for magtape and cassette.
/S	7.2.2.10	Copies files one block at a time.
/T	7.2.2.11	Puts the current date on all files you copy or rename, unless the current date is 0. This option is illegal for magtape and cassette; operations involving those devices always use the current date.
/U	7.2.2.12	Copies and concatenates all files you specify.
/W	7.2.5	Prints on the terminal a log of copy, rename, and delete operations.
/Y	7.2.2.13	Includes .SYS files in the operation you specify. You cannot modify or delete these files unless you use the /Y option.

7.2.1 Operations Involving Magtape and Cassette

PIP handles operations that involve magtape and cassette devices differently from operations that involve random access devices, such as disks and DECtape. That is because magtape and cassette are sequential access devices. This means that files are stored serially, one after another, on the device and that there is no directory at the beginning of each device that lists the files and gives their location. Because of the serial nature of tape, you can access only one file at a time on each device unit. Avoid commands that specify the same device unit number for both the input and output files – they are illegal. The /M:n option is designed to make operations that involve magtape and cassette more efficient. This option lets you specify different tape handling procedures for PIP to follow. The following sections outline the operations that involve magtape and cassette and describe the different procedures for using these devices that you can specify with the /M:n option. Remember that when you use the /M:n option, n is interpreted as an octal number. You must use n. (n followed by a decimal point) to represent a decimal number.

7.2.1.1 Using Cassette – The cassette is an inexpensive auxiliary storage medium. Cassettes are typically used to store data such as text files or source programs. Clear plastic leader indicates the beginning-of-tape (BOT) and physical end-of-tape (EOT). A special sentinel file marks the end of current data and indicates where new data can begin. The /M:n option lets you position the tape a particular way, or rewind it, before beginning an operation. You can also use it to specify a special procedure for tape handling during cassette operations with PIP. The following operations are valid for use with cassettes: /A, /B, /C, /D, /G, /M, /P, /Q, /R, /S, /U, /W, and /Y.

These options are illegal with cassettes: /K, /N, /O, /R, and /T. If you omit the /M:n option in a cassette operation, the cassette rewinds before each operation. Using /M:0 has the same effect. The character n represents a count of the number of files from the present position on the cassette. Note that the /M:n option has a different meaning for cassette and magtape. Section 7.2.1.2 describes how to use /M:n with magtape.

For cassette read (copy from tape) operations, the /M:n option initiates these procedures:

1. If n is 0:

   The cassette rewinds and PIP searches for the file you specify. If you specify more than one file, or if you use a wildcard in the file specification, the cassette rewinds before PIP searches for each file.

2. If n is a positive integer:

   PIP starts from the cassette's present position and searches for the file you specify. If PIP does not find the file by the time it reaches the nth file from its starting position, it uses the nth file for the read operation. Note that if PIP's starting position is not the beginning of the cassette, it is possible that PIP will not find the file you specify, even though it does exist on the tape.

3. If n is a negative integer:

   The cassette rewinds, then PIP follows the procedure outlined in step 2 above.

For cassette write (copy to tape) operations, the /M:n option initiates these procedures:

1. If n is 0:

   The cassette rewinds and PIP writes the file you specify starting at the logical end-of-tape (LEOT) position. PIP automatically deletes any file it finds along the way that has the same name and file type as the file you specify.

2. If n is a positive integer:

   PIP starts from the cassette's present position and searches n files ahead, deleting along the way any file it finds that has the same name and file type as the file you specify. If it does not reach LEOT before it reaches the nth file from its starting position, it enters the file you specify over the nth file and deletes any files beyond it on the tape. If PIP reaches LEOT before it reaches the nth file, it writes the file you specify at the end-of-tape.

3. If n is a negative integer:

   The cassette rewinds, then PIP follows the same procedure outlined in step 2 above.

If you are copying a file to cassette and reach the physical end-of-tape before the copy completes, PIP automatically continues the file on another cassette. The cassette device handler prints the CTn: PUSH REWIND OR MOUNT NEW VOLUME message. If you want to halt the copy operation at this point, push the cassette rewind button. The tape rewinds, PIP prints an error message, and then PIP prompts you for a new command. However, if you want to continue the file on another cassette, remove the first cassette and put another initialized cassette in its place. The new cassette rewinds immediately. PIP then continues copying the file. The continued part of the file has the same file name and file type as the first part of the file, but PIP adds 1 to its sequence number to show that it is a continued file. Make sure you have a supply of initialized cassettes handy for cassette copy operations; you cannot interrupt the copy operation to initialize a cassette when PIP is waiting for a new volume. The following example shows a copy operation that fills one cassette and continues to another.

```
*CT1:*.*=RK:RK*.SYS,%%.SYS/Y/W/M:1
 Files copied:
RK:RKMNSJ.SYS  to CT1:RKMNSJ.SYS
CT1: PUSH REWIND OR MOUNT NEW VOLUME
RK:RKMNFB.SYS  to CT1:RKMNFB.SYS
RK:DT.SYS      to CT1:DT.SYS
RK:DP.SYS      to CT1:DP.SYS
RK:DX.SYS      to CT1:DX.SYS
RK:RF.SYS      to CT1:RF.SYS
RK:RK.SYS      to CT1:RK.SYS
RK:DM.SYS      to CT1:DM.SYS
RK:DS.SYS      to CT1:DS.SYS
RK:TT.SYS      to CT1:TT.SYS
RK:LP.SYS      to CT1:LP.SYS
RK:CR.SYS      to CT1:CR.SYS
RK:MT.SYS      to CT1:MT.SYS
RK:MM.SYS      to CT1:MM.SYS
RK:NL.SYS      to CT1:NL.SYS
RK:PC.SYS      to CT1:PC.SYS
RK:EL.SYS      to CT1:EL.SYS
RK:CT.SYS      to CT1:CT.SYS
RK:BA.SYS      to CT1:BA.SYS
*
```

A directory listing of the second cassette shows that the first file, RKMNFB.SYS, is continued from a previous tape. (The number of blocks in a cassette directory listing is not meaningful; it really represents the total of sequence numbers in the directory.)

```
.DIRECTORY CT1:
 15-Apr-77
RKMNFB.SYS    1 15-Apr-77     DT    .SYS    0 15-Apr-77
DP    .SYS    0 15-Apr-77     DX    .SYS    0 15-Apr-77
RF    .SYS    0 15-Apr-77     RK    .SYS    0 15-Apr-77
DM    .SYS    0 15-Apr-77     DS    .SYS    0 15-Apr-77
TT    .SYS    0 15-Apr-77     LP    .SYS    0 15-Apr-77
CR    .SYS    0 15-Apr-77     MT    .SYS    0 15-Apr-77
MM    .SYS    0 15-Apr-77     NL    .SYS    0 15-Apr-77
PC    .SYS    0 15-Apr-77     EL    .SYS    0 15-Apr-77
CT    .SYS    0 15-Apr-77     BA    .SYS    0 15-Apr-77
 18 Files, 1 Blocks
```

If you are reading a file from cassette that is continued on another volume, the cassette handler also prints the CTn: PUSH REWIND OR MOUNT NEW VOLUME message when it reaches the end of the first tape. To abort the operation, push the cassette rewind button; PIP then issues an error message and prompts for a new command. To continue the read operation, remove the first cassette and mount the second one in its place. The second cassette rewinds immediately and PIP searches for a file with the correct name and sequence number. PIP repeats the new volume message if it does not find the correct file. The following example copies a file that is continued on a second cassette.

```
*RK1:*.*=CT1:RKMNFB.SYS/Y/W
 Files copied:
CT1: PUSH REWIND OR MOUNT NEW VOLUME
CT1:RKMNFB.SYS to RK1:RKMNFB.SYS
*
```

If you type a double CTRL/C during any output operation to cassette, PIP does not write a sentinel file at the end of the tape. Consequently, you cannot transfer any more data to the cassette unless you follow one of these two recovery procedures:

1. First, rewind the cassette. Then, transfer all good files from the interrupted cassette to another cassette and initialize the interrupted cassette as the following example shows. Use any arbitrarily large number for /M:n.

   ```
   *CT1:*.*=CT0:DMPX.MAC,EXAMP.FOR/M:1000
   *^C
   .R DUP
   *CT0:/Z/Y
   *
   ```

2. Determine the sequential number of the file that was interrupted and use the /M:n construction to enter a replacement file (either a new file or a dummy) over the interrupted file. PIP writes the replacement file and a sentinel file (LEOT) after it. The following example assumes the bad file is the fourth file on the cassette.

   ```
   *CT0:DUMMY.FIL=DT0:GLOBAL.MAC/M:-4
   *^C

   .DIRECTORY CT0:
    19-Apr-77
   DMPX   .MAC      0 19-Apr-77      MATCH .BAS      0 19-Apr-77
   EXAMP .FOR       0 19-ApR-77      DUMMY .FIL      0 19-Apr-77
    4 Files, 0 Blocks
   ```

   A directory listing of the cassette shows three files and the replacement file.

To copy multiple files to a cassette with a wildcard command, use the following:

```
*CTn:*.*=dev:*.*/M:1
```

Continue to mount new cassettes in response to the PUSH REWIND OR MOUNT NEW VOLUME message. Do not abort the process at any time (using two CTRL/Cs) since continuation files may not be completed and no sentinel file will be written on the cassette.

To read multiple files from a cassette, use a command like the following one. Use any arbitrarily large number for /M:n.

```
*dev:*.*=CTn:*.*/M:1000
```

Whenever PIP detects a continued volume, the PUSH REWIND OR MOUNT NEW VOLUME message appears, until the entire file has been copied (assuming that you mount each sequential cassette in response to each occurrence of the message). When PIP copies the final section of a continued file, it returns to command level. To copy the remaining files on that cassette, reissue the command:

```
*dev:*.*=CTn:*.*/M:1000
```

Repeat the process as often as necessary to copy all files. Do not abort the process at any time (using two CTRL/Cs) since continuation files may not be completed.

7.2.1.2 Using Magtape – Magnetic tape is a convenient auxiliary storage medium for large amounts of data. Magtapes are often used as backup for disks. Reflective strips indicate the beginning and end of the tape. A special label (an EOF1 or EOV1 tape label) followed by two tape marks indicates the end of current data and indicates where new data can begin. The following PIP options are valid for use with magtape: /A, /B, /C, /G, /M, /P, /Q, /S, /U, /W, and /Y. These options are illegal with magtape: /D, /K, /N, /O, /R, and /T. The /M:n option lets you direct the tape operation; you can move the tape and perform an operation at the point you specify. The /M:n option can be different for the output and input side of the command line. Since the option applies to the device and not to the files, you can specify one /M:n option for the output file and one for the input files. The /M:n option has a different meaning for cassette and magtape. Section 7.2.1.1 describes how to use /M:n with cassette.

Sometimes PIP begins an operation at the current position. To determine the current position, the magtape handler backspaces from its present position on the tape until it finds either an EOF indicator or the beginning of tape, whichever comes first. PIP then begins the operation with the file immediately following the EOF or BOT. The magtape handler also has a special procedure for locating a file with sequence number n:

1. If the file sequence number is greater than the current position, PIP searches the tape in the forward direction.
2. If the file sequence number is more than one file before the current position, or if the file sequence number is less than five files from the beginning-of-tape (BOT), the tape rewinds before PIP begins its search.
3. If the file sequence number is at the current position, or if it is one file past the current position, PIP searches the tape in the reverse direction.

Whenever you fetch or load a new copy of the magtape handler, the tape position information is lost. The "new" handler searches backwards until it locates either BOT or a label from which it can learn the position of the tape. It then operates normally, according to steps 1, 2, and 3 described above.

If you omit the /M:n option, the tape rewinds between each operation. Using /M:0 has the same effect as omitting /M:n. When n is positive, it represents the file sequence number. When n is negative, it represents an instruction to the magtape handler.

For magtape read (copy from tape) operations, the /M:n option initiates these procedures:

1. If n is 0:

   The tape rewinds and PIP searches for the file you specify. If you specify more than one file, the tape rewinds before each search. If the file specification contains a wildcard, the tape rewinds only once and then PIP copies all the appropriate files.

2. If n is a positive integer:

   PIP goes to file sequence number n. If the file it finds there is the one you specify, PIP copies it, Otherwise, PIP prints the ?PIP-F-FILE NOT FOUND message. If you use a wildcard in the file specification PIP goes to file sequence number n and then begins to search for matching files.

3. If n is -1:

   PIP starts the search at the current position. Note that if the current position is not the beginning of the tape, it is possible that PIP will not find the file you specify, even though it does exist on the tape.

For magtape write (copy to tape) operations, the /M:n option initiates these procedures:

1. If n is 0:

   The tape rewinds before PIP copies each file. PIP prints a warning message if it finds a file with the same name and file type as the input file and does not perform the copy operation.

2. If n is a positive integer:

   PIP goes to the file sequence number n and enters the file you specify. If PIP reaches LEOT before it finds file sequence number n, it prints the ?PIP-F-FILE SEQUENCE NUMBER NOT FOUND message. If you specify more than one file or if you use a wildcard in the file specification, the tape does not rewind before PIP writes each file, and PIP does not check for duplicate file names.

3. If n is -1:

   PIP goes to the LEOT and enters the file you specify. It does not check for duplicate file names.

4. If n is -2:

   The tape rewinds between each copy operation. PIP enters the file at LEOT or at the first occurrence of a duplicate file name.

If PIP reaches the physical end-of-tape before it completes a copy operation, it cannot continue the file on another tape volume. Instead, it deletes the partial file by backspacing and writing a logical end-of-tape over the file's header label. You must restart the operation and use another magtape.

If you type two CTRL/Cs during any output operation to magtape, PIP does not write a logical end-of-tape at the end of the data. Consequently, you cannot transfer any more data to the tape unless you follow one of the two following recovery procedures. In addition, if the magtape handler was loaded (with the monitor LOAD command), you must unload it before you can access the tape.

1. Transfer all good files from the interrupted tape to another tape and initialize the interrupted tape in the following manner:

   ```
   *dev1:*.*=dev0:*.*
   ^C
   .R DUP
   *dev0:/Z/Y
   ```

2. Determine the sequential number of the file that was interrupted and use the /M:n construction to enter a replacement file (either a new file or a dummy) over the interrupted file. PIP writes the replacement file and a good LEOT after it. The following example assumes the bad file is the fourth file on the tape:

   ```
   *dev0:file.new=file.dum/M:4
   ```

7.2.2 Copy Operations

The following sections describe the types of copy operation that PIP can perform. PIP copies files in image, ASCII, and binary format. Other options let you change the date on the files, access .SYS files, combine files, and perform other similar operations. PIP automatically allocates the correct amount of space for new files in copy operations (except for concatenation). For block-replaceable devices, PIP stores the new file in the first empty space large enough to accommodate it. If an error occurs during a copy operation, PIP prints a warning message, stops the copy operation, and prompts you for another command. You cannot copy .BAD files unless you specifically type each file name and file type.

7.2.2.1 Image Mode – If you enter a command line without an option, PIP copies files onto the destination device in image mode. Note that you cannot reliably transfer memory image files to or from paper tape, or to the line printer or console terminal. PIP can image-copy ASCII and binary data but it does not do any of the data checking described in Sections 7.2.2.2 or 7.2.2.3.

The following command makes a copy of the file named XYZ.SAV on device DK: and assigns it the name ABC.SAV. (Both files exist on device DK: following the operation.)

```
*ABC.SAV=XYZ.SAV
```

The next example copies from DK: all .MAC files whose names are three characters long and begin with A. PIP stores the resulting files on DX1:.

```
*DX1:*.*=A%%.MAC
```

7.2.2.2 ASCII Mode (/A) – Use the /A option to copy files in 7-bit ASCII mode. PIP ignores nulls and rubouts in an ASCII mode file transfer. PIP treats CTRL/Z (32 octal) as logical end-of-file if it encounters that character in the input file. The following command copies F2.FOR from device DK: onto device DT1: in ASCII mode and assigns it the name FI.FOR.

```
*DT1:F1.FOR=F2.FOR/A
```

7.2.2.3 Binary Mode (/B) – Use the /B option to transfer formatted binary files (such as .OBJ files produced by the assembler or the FORTRAN compiler and .LDA files produced by the linker). The following command, for example, transfers a formatted binary file from the paper-tape reader to device DK: and assigns it the name FILE.OBJ.

```
*DK:FILE.OBJ=PC:/B
```

When performing formatted binary transfers, PIP verifies checksums and prints a warning if a checksum error occurs. If there is a checksum error and you did not use /G to ignore the error, PIP does not perform the copy operation. You cannot copy library files with the /B option; PIP prints the ?PIP-F-LIBRARY FILE NOT COPIED message. Copy them in image mode.

7.2.2.4 The Newfiles Option (/C) – Use the /C option in the command line if you want to copy only those files with the current date. Specify /C only once in the command line. It applies to all the file specifications in the entire command. The following command copies (in ASCII mode) all files named ITEM1.MAC that also have the current date. It also copies the file ITEM2.MAC, if it has the current date, from DK: to DT2:. It combines all these files under the name NN3.MAC.

```
*DT2:NN3.MAC=ITEM1.MAC*/C,ITEM2.MAC/A/U
```

The next command copies all files with the current date (except .SYS and .BAD files) from DK: to DX1:. This is an example of an efficient way to back up all new files after a session at the computer.

```
*DX1:*.*=*.*/C
```

7.2.2.5 The Ignore Errors Option (/G) – Use the /G option to copy files, but ignore all input errors. This option forces a single-block transfer, which you can invoke at any other time with the /S option. Use the /G option if an input error occurred when you tried to perform a normal copy operation. The procedure can sometimes recover a file that is otherwise unreadable. If an error still occurs, PIP prints the ?PIP-W-INPUT ERROR message and continues the copy operation. The following command, for example, copies the file TOP.SAV in image mode from device DT1: to device DK: and assigns it the name ABC.SAV.

```
*ABC.SAV=DT1:TOP.SAV/G
```

The next command copies files F1.MAC and F2.MAC in ASCII mode from device DT1: to device DT2:. This command creates one file with the name COMB.MAC, and ignores any errors that occur during the operation.

```
*DT2:COMB.MAC=DT1:F1.MAC,F2.MAC/A/G/U
```

7.2.2.6 The Copies Option (/K:n) – The /K:n option directs PIP to generate n copies of the file you specify. The only legal output devices are the console terminal, the line printer, and paper-tape punch. Normally, each copy of the file begins at the top of a page; copies are separated by form feeds.

```
*LP:=FOO.LST/K:3
```

This command, for example, prints three copies of the listing file, FOO.LST, on the line printer.

7.2.2.7 Noreplace Option (/N) – The /N option prevents execution of a copy or rename operation if a file with the same name as the output file already exists on the output device. This option is illegal for magtape and cassette. The following example uses the /N option.

```
*DX0:CT.SYS=DK:CT.SYS/Y/N
?PIP-W-Output file found, no operation performed DK:CT.SYS
*
```

The file named CT.SYS already exists on DX0:, and the copy operation does not proceed.

7.2.2.8 The Predelete Option (/O) – The /O option deletes a file on the output device if you copy a file with the same name to that device. PIP deletes the file on the output device before the copy operation occurs. Normally, PIP deletes a file of the same name after the copy completes. This option is illegal for magtape and cassette. The following example uses the /O option.

```
*RK1:TEST1.MAC=DT2:TEST.MAC/O
```

If a file named TEST1.MAC already exists on RK1:, PIP deletes it before copying TEST.MAC from DT2: to TEST1.MAC on RK1:.

7.2.2.9 The Exclude Option (/P) – The /P option directs PIP to transfer all files except the ones you specify.

```
*DT0:*.*=DX1:*.MAC/P
```

This command, for example, directs PIP to transfer all files from DX1: to DT0: except the .MAC files.

7.2.2.10 The Single-block Transfer Option (/S) – The /S option directs PIP to copy files one block at a time. On some devices, this operation increases the chances of an error-free transfer. You can combine the /S option with other PIP copy options. For example:

```
*RK1:TEST.MAC=RK0:TEST.MAC/S
```

PIP performs this transfer one block at a time.

7.2.2.11 The Setdate Option (/T) – This option causes PIP to put the current date on all files it transfers, unless the current date is 0. Normally, PIP preserves the existing file creation date on copy and rename operations. This option is invalid for operations involving magtape and cassette because PIP always uses the current date for tape files. The following command puts the current date on all the files stored on device DK:.

```
**.*=*.*/Y/T
```

Note that the command shown above changes only the dates; PIP does not move or change the files in any other way.

7.2.2.12 The Concatenate Option (/U) – To combine more than one file into a single file, use the /U option. This option is particularly useful to combine several object modules into a single file for use by the linker or librarian. PIP does not accept wildcards on the output specification. The following examples use the /U option.

```
*DK:AA.OBJ=DT1:BB.OBJ,CC.OBJ,DD.OBJ/U
```

The command shown above transfers files BB.OBJ, CC.OBJ and DD.OBJ to device DK: as one file and assigns this file the name AA.OBJ.

```
*DT3:MERGE.MAC=DT2:FILE2.MAC,FILE3.MAC/A/U
```

This command merges ASCII files FILE2.MAC and FILE3.MAC on DT2: into one ASCII file named MERGE.MAC on device DT3:.

7.2.2.13 The System Files Option (/Y) – Use the /Y option if you need to perform an operation on system files (.SYS). For example:

```
**.*=DT3:*.*/G/Y
```

This command copies to device DK:, in image mode, all files (including .SYS files) from device DT3:. Because of the /G option, PIP ignores any input errors.

7.2.3 The Delete Operation (/D)

Use the /D option to delete one or more files from the device you specify. Note that PIP does not automatically query you before it performs the operation; you must use /Q. Remember to use the /Y option to delete .SYS files. You cannot delete .BAD files unless you name each one specifically, including file name and file type. You can specify only six files in a delete operation unless you use wildcards. You must always indicate a file specification in the command line. A delete command consisting only of a device name (dev:/D) is invalid. The delete option is also illegal for magtape. The following examples illustrate the delete operation.

```
*FILE1.SAV/D
```

The command shown above deletes FILE1.SAV from device DK:.

```
*DX1:*.*/D
?PIP-W-No .SYS action
*
```

The command shown above deletes all files from device DX1: except those with a .SYS or .BAD file type. If there is a file with a .SYS file type, PIP prints a warning message to remind you that this file has not been deleted.

```
**.MAC/D
```

This command deletes all files with a .MAC file type from device DK:.

7.2.4 The Rename Operation (/R)

Use the /R option to rename a file you specify as input, giving it the name you specify in the output specification. You must supply an equal number of input and output files that reside on the same device. PIP prints an error message if the command specifications are not valid. Use the /Y option with /R if you rename .SYS files. You cannot use /R with magtape or cassette.

The rename command is particularly useful when a file on disk or DECtape contains bad blocks. By renaming the file, giving it a .BAD file type, you can ensure that the file permanently resides in that area of the device. Thus, the system makes no other attempts to use the bad area. Once you give a file a .BAD file type, you cannot move it during a compress operation. You cannot rename .BAD files unless you specifically indicate both the file name and file type. The following examples illustrate the rename operation.

```
*DT1:F1.MAC=DT1:F0.MAC/R
```

The command shown above renames F0.MAC to F1.MAC on device DT1:.

```
*DX1:OUT.SYS=DX1:CT.SYS/Y/R
```

This command renames file CT.SYS to OUT.SYS.

7.2.5 The Logging Operation (/W)

When you use the /W option, PIP prints a list of all files copied, renamed, or deleted. The /W option is useful if you do not want to take the time to use the query mode (the /Q option, described in Section 7.2.6), but you do want a list of the files operated on by PIP.

PIP prints the log for an operation on the terminal beneath the command line. This example shows logging with the delete operation.

```
*DX1:*.*/D/W
?PIP-W-No .SYS action
 Files deleted:
DX1:TEST.MAC
DX1:FIX463.SAV
DX1:GRAPH.BAK
DX1:DMPX.MAC
DX1:MATCH.BAS
DX1:EXAMP.FOR
DX1:GRAPH.FOR
DX1:GLOBAL.MAC
DX1:PROSEC.MAC
DX1:KB.MAC
DX1:EXAMP.MAC
*
```

7.2.6 The Query Option (/Q)

Use the /Q option with another PIP operation to list all files and to confirm individually which of these files should be processed. Typing a Y (or any string that begins with Y) followed by a carrige return causes the named file to be processed; typing anything else excludes the file. The following example deletes files from DX1:.

```
*DX1:*.*/D/Q
 Files deleted:
DX1:FIX463.SAV?
DX1:GRAPH.BAK ? Y
DX1:DMPX.MAC  ?
DX1:MATCH.BAS ?
DX1:EXAMP.FOR ?
DX1:GRAPH.FOR ? Y
DX1:GLOBAL.MAC? Y
DX1:PROSEC.MAC? Y
DX1:KB.MAC     ?
DX1:EXAMP.MAC ?
*
```

CHAPTER 8
DEVICE UTILITY PROGRAM (DUP)

The device utility program (DUP) is a device maintenance utility program you can use with the RT-11 system. DUP creates files on file-structured RT-11 devices (disks, DECtape, magtape, and cassette). It can also extend files on certain file-structured devices (disks and DECtape), and it can compress, image copy, initialize, or boot RT-11 file-structured devices. DUP does not operate on non-file-structured devices (line printer, card reader, terminal, and paper tape).

8.1 CALLING AND USING DUP

To call DUP from the system device, respond to the dot (.) printed by the keyboard monitor by typing:

```
R DUP(RET)
```

The Command String Interpreter prints an asterisk (*) at the left margin of the terminal and waits for a command string. If you enter only a carriage return in response to the asterisk, DUP prints its current version number. You can type CTRL/C to halt DUP and return control to the monitor when DUP is waiting for input from the console terminal. You must type two CTRL/Cs to abort DUP at any other time. The /S, /T, and /C operations, however, lock out the CTRL/C command until the operation completes; these three operations cannot be interrupted with CTRL/C. To restart DUP, type R DUP or REENTER in response to the monitor's dot. Chapter 6, Command String Interpreter, describes the general syntax of the command line that DUP accepts. DUP accepts only one input file specification and one output file specification in the command line.

8.2 DUP OPTIONS

Certain options are available for use with DUP. These options are divided into two categories: 1) Action and 2) Mode. Action options cause specific operations to occur. You can use these options alone or with valid mode options. Usually, you can specify only one action option at a time. Mode options modify action options. Table 8-1 illustrates whch mode options you can use with a particular action option.

Table 8-1 DUP Options and Categories

Action	Mode
C	W,Y
I	W,Y
K	W,F,H
O	W,Y
S	W,X,Y
T	W,Y
U	W
V	W
Z	W,B,N,R,V,Y

Note that /V can be either an action or a mode option, depending on how you use it.

You can use DUP action options to create files, copy devices, scan for bad blocks, perform a bootstrap operation, and so on. You can use the DUP mode options to modify the action options, where necessary. The following sections describe the various DUP options and give examples of typical uses. Table 8-2 summarizes the options you can use with DUP.

Table 8-2 DUP Options

Option	Section	Explanation
/B	8.2.11.4	Use with /Z to write files with the file type .BAD over any bad blocks DUP finds on the disk to be initialized.
/C:m[:n]	8.2.1	Creates a file on the device you specify; m represents the starting block number (in octal) and n represents the size of the file in blocks.
/F	8.2.3	Use with the /K option to output the file name containing the bad block together with the relative block number of the bad block in the file.
/H	8.2.3	Use with the /K option to read the bad block, write to the bad block, and then read it again. This operation does not destroy information already stored on the device.
/I[:rstart :rstop :wstart]	8.2.2	Copies the image of a disk to another disk or magtape or from magtape to disk. The arguments :rstart, :rstop, and :wstart represent block numbers.
/K[:start [:stop]]	8.2.3	Scans a device for bad blocks and outputs the octal address of the logical bad blocks to the output device. The arguments :start and :stop represent block numbers.
/N:n	8.2.11.1	Use with /Z to set the number of directory segments you require if you do not want the default size; n is an integer in the range 1-37 (octal).
/O	8.2.4	Boots the device or file you specify.
/R[:RET]	8.2.11.3	Use with /Z to scan the RK06 device for bad blocks and to create a replacement table on the disk for any bad blocks DUP finds. If you use :RET, DUP retains the replacement table that is already on the disk and does not pre-scan the disk for bad blocks.
/S	8.2.5	Compresses a disk (or DECtape) onto itself or onto another disk (or DECtape); the output device, if any, must be initialized.
/T:n	8.2.6	Extends an existing file by the number of blocks you indicate by :n.
/U	8.2.7	Writes the bootstrap portion of the monitor file in blocks 0 and 2-5 of the target device.
/V[:VOL]	8.2.8, 8.2.11.2	Prints the user ID and owner name. Use it with /Z (as a mode option) to insert a user ID and owner name in block 1 of the initialized disk, or in the VOL1 header block on magtape (not applicable for cassette). Using /V:VOL as an action option causes only the ID and owner name to be changed, and does not initialize the device (not applicable for cassette).

(Continued on next page)

Table 8-2 (Cont.) DUP Options

Option	Section	Explanation
/W	8.2.9	Use with any action option (but only one) to initiate an operation and then pause. This is useful on small, single-disk systems because it lets you replace the system device with another disk before performing an operation.
/X	8.2.5	Use with /S to inhibit automatic booting of the system device when it is compressed.
/Y	8.2.10	Use with /C, /I, /O, /S, /T, or /Z to inhibit the dev:/xxxx ARE YOU SURE? message and the FOREGROUND JOB LOADED, CONTINUE? message and ensure immediate execution of the operation.
/Z[:n]	8.2.11	Initializes the directory of the device you specify. The size of the directory defaults to the standard RT-11 size; use :n to allocate extra directory words for each entry beyond the default.

8.2.1 The Create Option (/C:m[:n])

The /C option creates a file with a specific name, location, and size on the block-replaceable device that you specify. This command is useful to recover files that have been deleted. The /C option only creates a directory entry for the file. It does not store any data in the file. You must specify both the file name and file type of the file to be created. The syntax of the command is:

```
filespec=/C:m[:n]
```

where

filespec	represents the device, file name and file type of the file to be created.
:m	represents the numeric value, in octal, of the starting block of the file to be created.
:n	represents the size of the file in blocks. If you do not supply a value for n, DUP creates a 1-block file.

You can use the /C option to cover bad blocks on a disk by creating a file with a file type .BAD to cover the bad area.

Use /C also to recover accidentally-deleted files. In this case, use DIR to obtain a listing of the device. Use the /E and the /Q options in DIR; obtain a separate listing with each one. DIR lists files, tentative files, empty areas, and the sizes of all areas. You can then assign a file name to the area that contains the data you lost.

You can also use DUP to set aside a file on disk without performing any input or output operations on the file.

When you use the /C option, make sure that the area in which the file is to be created is empty. If there are more blocks in the empty than the file you are creating needs, DUP attempts to put the extra blocks in empties that are contiguous to the file you are creating. If there is not enough room in contiguous empties, the error message ?DUP-F-ILLEGAL CONTIGUOUS FILE prints and DUP does not create the file. The /C option checks for duplicate file names. If the file name you specify already exists on the device, DUP issues an error message and does not create a second file with the same name.

This is an example of a command that uses /C:

```
        *DK1:FILE.MAC=/C:140:3
```

This command creates a file named FILE.MAC consisting of blocks 140, 141, and 142 on device DK1:.

8.2.2 The Image Copy Option (/I)

The /I option copies block for block from one device to another. (This operation is not applicable for magtape or cassette.) If DUP encounters a bad block, it prints an error message. However, it retries the operation and performs the copy one block at a time. If only one error message prints, you can assume that the transfer completed correctly. The /I option is often used to copy one disk to another without changing the file structure or location of files on the device. In this case, it is an added convenience that you do not have to copy a boot block to the device. You can also copy disks that are not in RT-11 format, if they have no bad blocks.

Qualifiers to the /I option let you specify the blocks to be read from the input device; you can also specify a starting block number on the output device for the write operation. The syntax of the command is:

```
     output-device:=input-device:/I[:rstart:rstop:wstart]
```

where

:rstart	represents the starting block number on the input device for the read operation.
:rstop	represents the ending block number on the input device for the read operation.
:wstart	represents the starting block number on the output device for the write operation.

The command string must include an input and an output specification; there is no default device. If you need to specify a block number, you must supply all three block values. The /I operation does not copy to or from a device that has logical bad blocks. (Physical bad blocks can be logically replaced or covered, as Sections 8.2.11.3 and 8.2.11.4 describe.) If one device is smaller than the other, DUP copies only the number of blocks of the smaller device.

You can copy blocks between disk and magtape with /I. DUP stores the data on the tape, formatting it in 1K word blocks. It is possible to store only one disk image on a magtape, regardless of the size of the tape.

The following examples use the /I option. The file name A is not significant; it is a dummy file name required by the Command String Interpreter.

```
        *RK1:A=RK0:/I

        RK1:/Copy are you sure?
```

The command shown above copies all blocks from DK: to RK1:.

```
        *RK1:A=RK0:/I:0:500:501

        RK1:/Copy are you sure?Y
```

This command copies blocks 0-500 from RK0: to RK1:, starting at block 501.

8.2.3 The Bad Block Scan Option (/K)

Sometimes devices (disks and DECtapes) are manufactured with bad blocks, or they develop bad blocks as a result of use and age. You can use the /K option to scan a device and locate bad blocks on it. DUP prints the absolute block number of those blocks on the device that return hardware errors when DUP tries to read them. If you specify an output device (only TT: and LP: are valid), DUP prints the bad block report on that device. Remember that block numbers are octal and the first block on a device is block 0. If DUP finds no bad blocks, it prints only the header. A complete scan of a disk pack takes from one to several minutes depending on the size of the device. It does not destroy data that is stored on the device.

DUP reads only one block at a time when it scans a disk for bad blocks. Errors can occur on a multi-block copy even if DUP does not detect any with /K. Copy the data to a scratch disk with the /I option to discover any other bad blocks. You should scan a device for bad blocks before using /S to compress the device; if a read error occurs during a compress operation, the device may become unuseable.

You can scan selected portions of a device by specifying a beginning and ending block number. The syntax of this command is:

```
[output-device:=]input-device:/K[:start[:stop]]
```

where

:start	represents the block number of the first block to be scanned.
:stop	represents the block number of the last block to be scanned.

If you specify only a starting block number, DUP scans from the block you specify to the end of the device. You cannot specify an ending block number unless you also specify a starting block number.

If the device to be scanned has files on it, you can use /F with the /K option to print the name of the file containing the bad block together with the relative block number within the file that is bad.

You can use /H with /K to read the bad block, write to the bad block, and then read it again. If the block is still bad, DUP reports a HARD error. If the block recovers, DUP reports a SOFT error. This procedure does not destroy data already stored on the device.

The following command line uses the /K option to scan the entire disk, RK1:.

```
*RK1:/K/F
BAD BLOCKS FILENAME        REL BLK TYPE
  6615     EMPTY1.TST       6547   HARD
  6645     EMPTY2.TST       6577   HARD
  7255     EMPTY3.TST       7207   HARD
```

8.2.4 The Boot Option (/O)

The /O option can perform two operations: 1) a hardware bootstrap of a specific device and 2) a bootstrap of a particular monitor file that does not affect the bootstrap blocks on the device. The command syntax for a device bootstrap is as follows:

```
dev:/O
```

This operation has the same results as a hardware bootstrap. Legal devices for the boot operation are DT0:, RK0:-RK7:, RF:, SY:, DK:, DP0:-DP7:, DX0:-DX1:, DM0:-DM7:, and DS0:-DS7:.

Use the following syntax to boot the monitor you specify without changing the bootstrap on the device.

```
dev:monitor-name/O
```

This makes it easy for you to switch from one monitor to another. Whether bootstrapping a specific monitor or a specific device, DUP checks to see if the bootstrap blocks are correctly formatted. If the boot operation you request is invalid for any reason, DUP prints an error message and waits for another command.

When you reboot with the /O option, you do not have to reenter the date and time of day with the monitor DATE and TIME commands. However, the clock does lose a few seconds during the reboot.

The following command reboots the RT-11 system under the single-job monitor:

```
*RK0:RKMNSJ.SYS/O

RT-11SJ     V03.01
```

Notice in this command that the device you specify must be the same device type as the first two characters of the monitor file indicate. Because of this restriction on the monitor-name bootstrap operation, the following command is illegal:

```
*RK0:DXMNFB.SYS/O
```

However, the next command is a valid one:

```
*RK0:RKMNFB.SYS/O
```

8.2.5 The Squeeze Option (/S)

Use the /S option to compress a device (disk or DECtape) onto itself or onto another disk or DECtape. To do this, DUP moves all the files to the beginning of the device, producing a single, unused area after the group of files. The squeeze operation does not change the bootstrap blocks of a device. The output device you specify, if any, must be an initialized device. If you specify an output device, DUP does not query you for confirmation before it performs the operation. If you do not specify an output device, DUP prints the ARE YOU SURE? message and waits for your response before proceeding. You must type Y followed by a carriage return to execute the command. Since it is critical to perform an error-free squeeze operation, be sure to scan a device (with /K) before you use /S.

The /S option does not move files with .BAD file types. This feature prevents you from reusing bad blocks that occur on a disk. You can rename files containing bad blocks, giving them a .BAD file type, and DUP then leaves them in place when you execute a /S. DUP inserts files before and after .BAD files until the space between the last file it moved and the .BAD file is smaller than the next file to be moved. If an error occurs during a squeeze operation, DUP continues the operation, performing it one block at a time. If only one error message prints, you can assume that the operation completed correctly.

The syntax of the command is:

```
[output-device=]input-device/S
```

Do not use /S on the system device (SY:) when a foreground job is loaded. A ?DUP-F-CANNOT WRITE SY: WHILE FJOB LOADED error message results if you attempt this and DUP ignores the /S operation. You must unload the foreground job before using the /S option.

NOTE

If you perform a compress operation on the system device, the system automatically reboots when the compress operation is completed. This operation takes place in order to prevent system crashes that can occur when the monitor file is moved.

You can use /X with /S to suppress the automatic reboot and leave DUP running. However, you should use /X only if you are certain that the monitor file will not move. Even then, you should reboot the system when the squeeze operation completes if the device handlers have moved. If you specify the /X option but for some reason the USR cannot be made resident, DUP reboots the system anyway. If you use /X and the system is not rebooted, the ?DUP-W-REBOOT message prints. This is a warning message; it is for your information only.

The following examples use the /S command:

```
*SY:/S

 SY:/Squeeze are you sure?Y

RT-11SJ     V03.01
```

The command shown above compresses the files on the system device and reboots the system when the compress operation completes.

```
*DT1:A=DT2:/S
```

This command transfers all the files from device DT2: to device DT1:, leaving DT2: unchanged. The file name A is not significant; it is a dummy file name required by the Command String Interpreter.

8.2.6 The Extend Option (/T:n)

Use the /T option to extend the size of a file. The syntax of the command is:

filespec/T:n

where

filespec	represents the device, file name, and file type of the file to be extended.
n	represents the number of blocks to add to the file.

You can extend a file in this manner only if it is followed by an unused area at least n blocks long. Any blocks not required by the extend operation remain in the unused area.

The following example uses the /T option:

```
*DT1:ZYZ.TST/T:100
```

This command assigns 100 more blocks to the file named ZYZ.TST on device DT1:.

8.2.7 The Bootstrap Copy Option (/U)

In order to use a disk as a system device, you must copy a bootstrap onto the disk. To do this, first make sure that the appropriate monitor file is stored on the disk. For a diskette system, for example, you could use the foreground/background monitor file called DXMNFB.SYS. If you copy the monitor file onto the diskette from another device, be careful not to rename it. DUP recognizes only standard RT-11 monitor file names in the bootstrap copy operation. Use the /U option to copy the bootstrap portion of the monitor file into absolute blocks 0 and 2-5 of the device. You can then use the /O option to boot the device.

To copy a bootstrap for the single-job monitor on RK1:, for example, use the following procedure:

1. Obtain a formatted disk. (Most disks and DECtapes are formatted by the manufacturer. However, the RT-11 System Generation Manual does outline the procedure for re-formatting an RK05 disk.)

2. Initialize the disk with /Z.

3. Copy files onto the disk.

4. Copy the monitor onto the disk.

5. Copy the monitor bootstrap onto the disk with /U.

The following example shows how to initialize a diskette, copy files to it, and write a bootstrap onto the diskette:

```
*DX1:/Z/Y
```

The command shown above (step 2 of the procedure described above) initializes the diskette.

```
*DX1:A=DX0:/S
```

This command, which combines steps 3 and 4, squeezes all the files from DX0: onto DX1:.

```
*DX1:A=DX1:DXMNFB.SYS/U
```

The last command (step 5) writes the bootstrap for the diskette foreground/background monitor onto the bootstrap blocks (blocks 0 and 2-5) of DX1:. The file name A is not significant; it is a dummy file name required by the Command String Interpreter.

8.2.8 The Volume ID Option (/V[:VOL])

You can use the /V option as an action option to print the volume ID of a device or to change the volume ID without initializing the device. The syntax of the command is:

device:/V[:VOL]

where

device:	is the device whose volume ID you want to display or change.

If you specify only /V, the volume ID and owner name of the device you specify print out on the console terminal. If you specify /V:VOL, DUP assumes you need to change the volume ID and owner name. DUP prompts you for a volume ID:

```
VOL ID?
```

Respond with a volume ID that is up to 12 characters long for a block-replaceable device, or up to 6 characters long for magtape. Terminate your response with a carriage return. DUP then prompts for an owner name:

```
OWNER NAME?
```

Respond with an owner name that is up to 12 characters long for a block-replaceable device, or up to 10 characters long for magtape. Terminate your response with a carriage return. DUP ignores characters you type beyond the legal length. The /V:VOL command changes only the volume ID and owner name; it does not initialize the device. Section 8.2.11.2 describes how to use /V with the /Z option to initialize a device and write volume identification on it.

DUP stores the volume ID and owner name information in block 1 of a disk. The volume ID is stored in words 236-241 (decimal), the owner name is stored in words 242-247, and the format type, which is always RT11A, is stored in words 248-253. The remainder of block 1 (words 0-235 and 254-255) is reserved for the system to use. If you are initializing a magnetic tape, DUP stores the volume identification information in the VOL1 header block of the magtape. The volume ID is stored in bytes 5-10 and the owner name is stored in bytes 41-50. The first byte of the header block is byte 1; DUP stores VOL1 information up to byte 80.

The following example uses the /V:VOL option:

```
*RK1:/V:VOL/Y

 VOL ID? VOUCHERS

 OWNER NAME? PAYABLES
```

This command writes a new volume ID and owner name on device RK1:.

8.2.9 The Small, Single-disk System Option (/W)

The /W option is useful for small (8K), single-disk systems. It is a mode option that you can use with any of the action options. However, you can perform only one operation at a time. The /W option initiates execution of a command, but then pauses and prints the message CONTINUE?. At this time you can remove the system disk and mount the disk on which you actually want the operation to take place. When the new disk is loaded, type a Y followed by a carriage return to execute the operation. When the operation completes (except the /O operation, which boots the system), the "CONTINUE?" message again prints. Replace the system device and type a Y followed by a carriage return. The asterisk (*) prompt prints and DUP waits for you to enter another command. The following example uses the /W option:

```
*DX1:/K/F/W
CONTINUE?Y
BAD BLOCKS   TYPE   FILENAME     REL BLK
CONTINUE?Y
*
```

This command directs DUP to scan the disk for bad blocks. During the first pause, the system disk is removed and another disk is mounted. A Y is typed and the scan operation executes. During the second pause, the system disk is replaced and another Y is typed. DUP prompts for another command.

NOTE

It is not necessary to use the /W option to change disks if the USR can be made resident with the SET USR NOSWAP command. In this case you can change disks when the asterisk (*) prompt prints. Type and execute the command with the new disk in place. Replace the system disk when the next prompt prints.

There is one exception to the general usage of /W. You cannot use the /U option to write a bootstrap on another disk if you have a single-disk system with only 8K words of memory. Follow this procedure to write a bootstrap on another disk:

1. Make the USR resident:

```
.SET USR NOSWAP
```

2. Call the MDUP program (a program similar to DUP, but smaller):

```
.R MDUP
*
```

3. Change disks when MDUP prompts with an asterisk (*), as shown in step 2.

   The new disk must already have the monitor file stored on it. Then enter the /U command to copy the bootstrap, as this example shows:

```
*RK0:A=RK0:RKMNSJ.SYS/U
```

   When MDUP prints another asterisk, replace the system disk and type CTRL/C to return to the monitor.

8.2.10 The Noquery Option (/Y)

Use the /Y option to suppress the query messages that some commands print. The following options normally print the FOREGROUND JOB LOADED, CONTINUE? message if a foreground job is loaded when you issue one of these DUP commands: /C, /I, /O, /S, /T, and /Z. You must respond to the query message by typing Y followed by a carriage return for the operation to proceed. Some other options (/C, /I, /O, /S, /V, and /Z) print the ARE YOU SURE? message and wait for your response. If a foreground job is loaded and you specify one of these options, DUP combines the two query messages into one message and waits for your response. You can suppress all these messages and the pause associated with them by specifying /Y in the command string.

8.2.11 The Directory Initialization Option (/Z[:n])

You must initialize a device before you can store files on it. Use the /Z option to clear and initialize the directory of an RT-11 directory-structured device. The /Z operation must always be the first operation you perform on a new device after you receive it, formatted, from a manufacturer. After you use /Z, there are no files in the directory.

The syntax of the command is as follows:

```
device:/Z[:n]
```

In this command, the optional argument, n, is an octal number (greater than or equal to 1) indicating the change in size of each directory entry on a directory-structured device. The size of the directory determines the number of files that can be stored on a device. The system allows a maximum of 72 files per directory segment, and 31 directory segments per device. Each segment uses two blocks of available disk space. If you do not specify n, each entry is seven words long (for file name, creation date, and file length information). When extra words are allocated, the number of entries per directory segment decreases. The formula for determining the number of entries per directory segment is:

5-7/((# of extra words) +7)

For example, if you use /Z:1, you can make 63 entries per segment. RT-11 does not normally support non-standard directory formats. DIGITAL does not recommend altering the directory format. The number of directory segments in the directory defaults to the decimal value shown in Table 8-3 for the specified device.

Table 8-3 Default Directory Sizes

Device	Size (decimal) of Directory in Segments
RK05	16
DT	4
RF	4
DS	4
DP	31
DX	4
RK06	31

8.2.11.1 Changing Directory Segments (/N:n) – If you do not want the default size of the device, use /N with /Z to set the number of directory segments for entries in the directory. The syntax of the command is as follows:

```
/N:n
```

In this command, n represents the number of directory segments; n is an integer in the range 1-31.

The following example initializes the directory on device RK1: and allocates six directory segments.

```
*RK1:/Z/N:6

RK1:/Init are you sure?Y
```

8.2.11.2 Storing Volume ID (/V) – When you initialize a disk or magtape, DUP stores a default device ID of RT11A in block 1 of the device. You can use the /V option with /Z to insert a user ID and owner name in block 1 of the device. For example, the following command initializes device RK1: and prompts you for a volume ID and owner name. Section 8.2.8 illustrates these prompts and shows how to respond to them.

```
*RK1:/Z/V

RK1:/Init are you sure?Y

 VOL ID? VOUCHERS

 OWNER NAME? PAYABLES
```

8.2.11.3 Replacing Bad Blocks (/R[:RET]) – You can use the /R option with /Z if the device being initialized is an RK06. If DUP finds any bad blocks, it builds a replacement table of good blocks for them. The replacement table is stored in words 0-63 of block 1. (/R supports up to 32 bad blocks.) The RK06 then appears to have no bad blocks. Files that span the bad block use a replacement block instead of the bad block. The replacement blocks are located in the last cylinder of the disk. Speed of input and output operations decreases only when the replacement blocks for bad blocks are accessed. You can avoid this overhead by using the /B option (see Section 8.2.11.4) and not using bad block replacement. If DUP finds any bad blocks in a non-replaceable part of the disk, DUP reports that the disk is bad. When you initialize a device and want to retain the bad block replacement table that was created by a previous /R command, use /R:RET. The /R:RET option makes it easy to reinitialize an RK06 without rescanning it. After a disk is initialized with the /Z/R option combination, a scan of the disk with /K should reveal no bad blocks. If DUP finds a bad block during the /Z/R operation that is in blocks 0 through 5, it reports that the disk is not usable. If DUP finds a bad block that is not already marked on the disk as such, it prints the ?DUP-W-UNMARKED BAD BLOCK message. This disk is not usable and must be reformatted by the manufacturer. If DUP finds bad blocks in the device directory, it prints a warning message. Bad blocks in the directory can cause considerable overhead and slow system performance on ENTER, LOOKUP, and CLOSE operations.

8.2.11.4 Covering Bad Blocks (/B) – To scan the disk for bad blocks and write files over them, use the /B option with /Z. For every bad block DUP encounters on the device, it creates a file called FILE.BAD to cover it. After the disk is initialized and the scan completed, the directory consists only of file FILE.BAD entries that cover the bad blocks. If DUP finds a bad block in the boot block or the directory, it prints an error message and the disk is not usable.

/R and /B are mutually exclusive options. You can use one or the other, but not both.

CHAPTER 9
THE DIRECTORY PROGRAM (DIR)

The directory program (DIR) performs a wide range of directory listing operations. It can list directory information about a specific device, such as the number of files stored on the device, their names, and their creation dates. DIR can list details about certain files, too, including their names, their file types, and their size in blocks. DIR can also print a device directory summary, and it can organize its listings in several ways, such as alphabetically or chronologically.

9.1 CALLING AND USING DIR

To call DIR from the system device, respond to the dot (.) printed by the keyboard monitor by typing:

```
R DIR(RET)
```

The Command String Interpreter prints an asterisk at the left margin of the terminal and waits for you to enter a command string. If you enter only a carriage return in response to the asterisk, DIR prints its current version number. You can type CTRL/C to halt DIR and return control to the monitor when DIR is waiting for input from the console terminal. You must type two CTRL/Cs to abort DIR at any other time. To restart DIR, type R DIR or REENTER in response to the monitor's dot. Chapter 6, Command String Interpreter, describes the general syntax of the command line that DIR accepts. Unless otherwise indicated, numeric arguments are interpreted as octal. Remember to put a decimal point after a decimal number to distinguish it from an octal number. Some of the DIR options accept a date as an argument in the command line. The syntax for specifying the date is:

```
dd.:mmm:yy.
```

where

dd.	represents the day (a decimal integer in the range 1-31).
mmm	represents the month (the first three characters of the name of the month).
yy.	represents the year (a decimal integer in the range 73-99).

You can specify only one input device and one output device, but you can specify up to six file names on the input device. The default device for output is the terminal. The default file type for an output file is .DIR. The default device for input is DK:. If you omit the input specification completely, DIR uses DK:*.*. If you do not supply an option, DIR performs the /L operation. Note that wildcards are valid with DIR for the input specification only.

Directory listings normally print on the terminal in two columns. Read the entries across the columns, moving from left to right, one row at a time. Directory listings that are sorted, however, are an exception to this. (Sorted directories are produced by /A, /R, and /S.) Read these listings by reading the left column from top to bottom, then reading the right column from top to bottom.

9.2 DIR OPTIONS

You can perform many different directory operations by specifying options in the DIR command line. Table 9-1 summarizes the operations these options permit you to perform with DIR. The following sections describe the various DIR options and give examples that use the options. The sections are arranged alphabetically by option.

Table 9-1 DIR Options

Option	Section	Explanation
/A	9.2.1	Lists the directory of the device you specify in alphabetical order by file name and type (this is the same as /S:NAM).
/B	9.2.2	Lists the directory of the device you specify, including file names and types, creation dates, starting block number in decimal, and the number of blocks in each file. For magtape, the starting block number is the file sequence number.
/C:n	9.2.3	Lists the directory in n columns; n is an integer in the range 1-9. The default value is two columns for normal listings and five columns for abbreviated listings.
/D[:date]	9.2.4	Includes in the directory listing only those files with the date you specify. If you do not supply a date, DIR uses the system's current date.
/E	9.2.5	Lists the device directory including unused spaces and their sizes. An empty space on a cassette directory represents a deleted file.
/F	9.2.6	Prints in five columns a short directory (file names and types only) of the device you specify.
/G	9.2.7	Lists the file you specify and all files that follow it in the directory. This option does not list any files that precede the file you specify.
/J[:date]	9.2.8	Prints a directory of the files created on or after the date you specify. If you do not supply a date, DIR uses the system's current date.
/K[:date]	9.2.9	Prints a directory of files created before the date you specify. If you do not supply a date, DIR uses the system's current date.
/L	9.2.10	Lists the directory of the device you specify, including the number of files, their dates, and the number of blocks each file occupies. (This is the default operation.)
/M	9.2.11	Lists a directory of unused areas of the device you specify.
/N	9.2.12	Lists a summary of the device directory.
/O	9.2.13	Similar to /L but lists the sizes and block numbers of the files in octal.
/P	9.2.14	Prints a directory of the device you specify, excluding the files you list.
/Q	9.2.15	Lists a directory of the device you specify, listing the file names and types, sizes, creation dates and starting block numbers of files that have been deleted and whose file name information has not been destroyed.
/R	9.2.16	Lists the files in the reverse order of the sort specified with /A or /S.
/S[:xxx]	9.2.17	Lists the directory of the device you specify in the order you specify; xxx indicates the order in which DIR sorts the listing (xxx can be DAT, NAM, POS, SIZ, or TYP).

9.2.1 The Alphabetical Option (/A)

The /A option lists the directory of the device you specify in alphabetical order by file name and type. It has the same effect as the /S:NAM option. The following example lists the directory of device DX0: in alphabetical order on the terminal.

```
*DX0:/A
 13-Apr-77
DIR    .SAV    16 15-Mar-77     LP     .SYS      2 01-Mar-77
DT     .SYS     2 01-Mar-77     MACRO  .SAV     46 01-Mar-77
DUP    .SAV    17 04-Mar-77     PIP    .SAV     16 16-Mar-77
DXMNSJ.SYS     91 01-Mar-77     SYSMAC.MAC      27 18-Feb-77
EDIT   .SAV    21 01-Mar-77     TT     .SYS      2 01-Mar-77
LINK   .SAV    25 01-Mar-77
 11 Files, 265 Blocks
 215 Free Blocks
```

9.2.2 The Block Number Option (/B)

The /B option prints a directory of the device you specify and includes the starting block number in decimal of all the files listed. The following example lists the directory of device DX0:, including the starting block numbers of files.

```
*DX0:/B
 13-Apr-77
DXMNSJ.SYS     91 01-Mar-77      14     TT     .SYS      2 01-Mar-77     105
LP     .SYS     2 01-Mar-77     107     DT     .SYS      2 01-Mar-77     109
EDIT   .SAV    21 01-Mar-77     111     LINK   .SAV     25 01-Mar-77     132
DUP    .SAV    17 04-Mar-77     157     DIR    .SAV     16 15-Mar-77     174
PIP    .SAV    16 16-Mar-77     190     MACRO  .SAV     46 01-Mar-77     206
SYSMAC.MAC     27 18-Feb-77     252
 11 Files, 265 Blocks
 215 Free blocks
```

9.2.3 The Columns Option (/C:n)

The /C[:n] option lists the directory in the number of columns you specify. The argument, n, represents an integer in the range 1-9. If you do not use the /C:n option, DIR lists the directory in two columns for normal listings and five columns for abbreviated listings. The following command, for example, lists on the terminal the directory of device DX1: in one column.

```
*DX1:/C:1
 13-Apr-77
FORTRA.SAV    191 28-Feb-77
BASIC .SAV     51 25-Feb-77
SYSLIB.OBJ    200 31-Mar-77
 2 Files, 442 Blocks
 38 Free blocks
```

9.2.4 The Date Option (/D[:date])

The /D[:date] option includes in the directory listing only those files with the date you specify. The default date is the system's current date. For example, the following command lists on the terminal all the files that were created on 1 March 1977.

```
*DX0:/D:01.:MAR:77.
 13-Apr-77
DXMNSJ.SYS     91  01-Mar-77     TT     .SYS      2  01-Mar-77
LP     .SYS     2  01-Mar-77     DT     .SYS      2  01-Mar-77
EDIT   .SAV    21  01-Mar-77     LINK   .SAV     25  01-Mar-77
MACRO  .SAV    46  01-Mar-77
 7 Files, 149 Blocks
 215 Free blocks
```

9.2.5 The Entire Option (/E)

The /E option lists the entire directory including the unused areas and their sizes in blocks (decimal). The following example lists on the terminal the entire directory of device DX1:, including unused areas.

```
*DX1:/E
 03-May-77
DIR    .SAV    16  08-Apr-77     DUP    .SAV     17  13-Apr-77
ABC    .MAC     4  19-Apr-77     AAF    .MAC      2  19-Apr-77
PIP    .SAV    16  14-Apr-77     COMB   .SAV      4  19-Apr-77
MERGE  .FOR     6  24-Apr-77     < UNUSED >     415
 7 Files, 65 Blocks
 415 Free blocks
```

9.2.6 The Fast Option (/F)

The /F option lists only file names and file types, omitting file lengths and associated dates. For example, the following command lists on the terminal only file names and types from device DT0:.

```
*DT0:/F
 13-Apr-77
DMPX   .MAC     MATCH  .BAS     EXAMP  .FOR     GRAPH  .FOR     SPOOL  .MAC
GLOBAL.MAC      PROSEC.MAC      KB     .MAC     EXAMP  .MAC     FIX463.SAV
GRAPH  .BAK     DTMNSJ.SYS
 12 Files, 167 Blocks
 397 Free blocks
```

9.2.7 The Begin Option (/G)

The /G option lists the directory of the device you specify, beginning with the file you specify and including all the files that follow it in the directory. Usually, the disk you are using as a system device contains a number of files that the operating system needs. These files include .SYS monitor files, .SAV utility program files, and various .OBJ, .MAC, and .BAT files. They are generally grouped together and usually list at the beginning of a normal device directory. Files that you create and use, such as source files and text files, are also grouped together and follow the operating system files in the directory. If you specify the name of the last system file with the /G in the command line, DIR prints a directory of only those files that you created and stored on the device. The following command, for example, lists the last system file (CT.SYS) and all the user files that follow it.

```
*DX0:CT.SYS/G
 15-Apr-77
CT     .SYS     5  08-Apr-77     TEXT   .TST     18  28-Jan-77
PROG   .BAS     3  15-Apr-77     DMPX   .MAC      3  15-Apr-77
MATCH  .BAS     3  15-Apr-77     EXAMP  .FOR      2  15-Apr-77
GRAPH  .FOR     2  15-Apr-77     GLOBAL.MAC       2  15-Apr-77
PROSEC.MAC      2  15-Apr-77     KB     .MAC     33  15-Apr-77
EXAMP  .MAC     4  15-Apr-77     FIX463.SAV       2  29-Jul-76
GRAPH  .BAK    18  28-Jan-77
 13 Files, 97 Blocks
 199 Free blocks
```

9.2.8 The Since Option (J[:date])

The /J[:date] option lists a directory of all files stored on the device you specify that were created on or after the date you supply. The default date is the system's current date. The following command lists on the terminal all files on device DT0: that were created on or after 28 January 77.

```
*DT0:/J:28.:JAN:77.
 13-Apr-77
GRAPH .BAK     18 28-Jan-77      DTMNSJ.SYS     91 01-Mar-77
 2 Files, 109 Blocks
 397 Free blocks
```

9.2.9 The Before Option (/K[:date])

The /K[:date] option prints a directory of files created before the date you specify. The default date is the system's current date. The following command lists on the terminal all files stored on device DX1: that were created before 15 March 1977.

```
*DX1:/K:15.:MAR:77.
 13-Apr-77
FORTRA.SAV    191 28-Feb-77      BASIC .SAV     51 25-Feb-77
 2 Files, 242 Blocks
 38 Free blocks
```

9.2.10 The Listing Option (/L)

The /L option lists the directory of the device you specify. The listing contains the current date, all files and their associated creation dates, the number of blocks used by each file, total free blocks on the device (if disk or DECtape), the number of files listed, and the total number of blocks used by the files. File lengths, number of blocks and number of files are indicated as decimal values. For example, the following command lists on the line printer the directory for device DT1:.

```
        *LP:=DX1:/L
```

The line printer output looks like this:

```
 03-MAY-77
DIR   .SAV     16 08-APR-77      DUP   .SAV     17 13-APR-77
ABC   .MAC      4 19-APR-77      AAF   .MAC      2 19-APR-77
PIP   .SAV     16 14-APR-77      MERGF .FOR      6 24-APR-77
 6 FILES, 61 BLOCKS
 419 FREE BLOCKS
```

9.2.11 The Unused Areas Option (/M)

The /M option prints only a directory of unused areas and their size on the device you specify. For example, the following command lists on the terminal all the unused areas on device DK:.

```
*/M
 03-May-77
< UNUSED >     21                < UNUSED >     295
< UNUSED >     16                < UNUSED >     594
 0 Files, 0 Blocks
 926 Free blocks
```

9.2.12 The Summary Option (/N)

The /N option prints a summary of the device directory. The following command lists on the terminal the summary of the directory for device DK:.

```
*/N
 13-Apr-77

    72 Files in segment 1

    72 Files in segment 2

    72 Files in segment 3

    12 Files in segment 4

    16 Available segments, 4 in use

 228 Files, 4141 Blocks
 621 Free blocks
```

9.2.13 The Octal Option (/O)

The /O option is similar to the /L option, but lists the sizes and starting block numbers (if you use /B) of the files in octal. If the device you specify is a magnetic tape or cassette, DIR prints the sequence number in octal. For example, the following command lists on the terminal the directory of device DX0:, with sizes in octal.

```
*DX0:/O
 13-Apr-77 Octal
DXMNSJ.SYS    133 01-Mar-77    TT    .SYS     2 01-Mar-77
LP    .SYS      2 01-Mar-77    DT    .SYS     2 01-Mar-77
EDIT  .SAV     25 01-Mar-77    LINK  .SAV    31 01-Mar-77
DUP   .SAV     21 04-Mar-77    DIR   .SAV    20 15-Mar-77
PIP   .SAV     20 16-Mar-77    MACRO .SAV    56 01-Mar-77
SYSMAC.MAC     33 18-Feb-77
 11 Files, 411 Blocks
 327 Free blocks
```

9.2.14 The Exclude Option (/P)

The /P option lists a directory of all files on a specific device, excluding those that you list. You can specify up to six file specifications.

```
*DX1:*.SAV/P
 03-May-77
ABC   .MAC     4 19-Apr-77    AAF   .MAC     2 19-Apr-77
MERGE .FOR     6 24-Apr-77
 3 Files, 12 Blocks
 419 Free blocks
```

This command lists on the terminal all files on device DX1: except .SAV files.

9.2.15 The Deleted Option (/Q)

The /Q option lists a directory of the device you specify, listing the file names, types, sizes, creation dates, and starting block numbers in decimal of files that have been deleted but whose file name information has not been destroyed. The file names that print represent either tentative files or files that have been deleted. This can be useful in recovering files that have been accidentally deleted. Once you identify the file name and location, you can use DUP to rename the area. See Section 8.2.1 for this procedure.

```
        *DISK.DIR=/Q
```

This command creates a file called DISK.DIR on device DK: that contains directory information about unused areas from device DK:.

Use the monitor TYPE command to read the file:

```
.TYPE DISK.DIR/LOG
 Files copied:
DK:DISK.DIR     to TT:
 03-May-77
EDIT   DEM      21 03-May-77    3843  DUM    .      295 03-May-77   3882
DEMOF1.OBJ      16 26-Apr-77    4179  DISK   .DIR   297 03-May-77   4206
SCOPE .PIC     297 03-May-77    4503
 0 Files, 0 Blocks
 0 Free blocks
```

9.2.16 The Reverse Option (/R)
The /R option lists a directory in the reverse order of the sort you specify with the /A or /S option.

```
*DXO:/S:DAT/R
 13-Apr-77
PIP   .SAV     16 16-Mar-77     LINK  .SAV     25 01-Mar-77
DIR   .SAV     16 15-Mar-77     LP    .SYS      2 01-Mar-77
DUP   .SAV     17 04-Mar-77     MACRO .SAV     46 01-Mar-77
DT    .SYS      2 01-Mar-77     TT    .SYS      2 01-Mar-77
DXMNSJ.SYS     91 01-Mar-77     SYSMAC.MAC     27 18-Feb-77
EDIT  .SAV     21 01-Mar-77
 11 Files, 265 Blocks
 215 Free blocks
```

This command lists on the terminal the directory of device DX0: in reverse chronological order.

9.2.17 The Sort Option (/S[:xxx])
The /S[:xxx] option sorts the directory of the specified device according to a 3-character code you specify with :xxx. Table 9-2 summarizes the codes and their functions.

Table 9-2 Sort Codes

Code	Explanation
DAT	Sorts the directory chronologically by creation date. Files that have the same date are sorted alphabetically by file name and file type.
NAM	Sorts the directory alphabetically by file name. Files that have the same file name are sorted alphabetically by file type (this has the same effect as the /A option).
POS	Lists the files in order by their position on the device. This is the same as using /S with no code.
SIZ	Sorts the directory based on file size in blocks. Files that are the same size are sorted alphabetically by file name and file type.
TYP	Sorts the directory alphabetically by file type. Files that have the same file type are sorted alphabetically by file name.

The following examples illustrate the /S option.

```
*DXO:/S:DAT
 13-Apr-77
SYSMAC.MAC    27 18-Feb-77      MACRO  .SAV    46 01-Mar-77
DT     .SYS    2 01-Mar-77      TT     .SYS     2 01-Mar-77
DXMNSJ.SYS    91 01-Mar-77      DUP    .SAV    17 04-Mar-77
EDIT   .SAV   21 01-Mar-77      DIR    .SAV    16 15-Mar-77
LINK   .SAV   25 01-Mar-77      PIP    .SAV    16 16-Mar-77
LP     .SYS    2 01-Mar-77
 11 Files, 265 Blocks
 215 Free blocks

*DXO:/S:NAM
13-Apr-77
DIR    .SAV   16 15-Mar-77      LP     .SYS     2 01-Mar-77
DT     .SYS    2 01-Mar-77      MACRO  .SAV    46 01-Mar-77
DUP    .SAV   17 04-Mar-77      PIP    .SAV    16 16-Mar-77
DXMNSJ.SYS    91 01-Mar-77      SYSMAC.MAC     27 18-Feb-77
EDIT   .SAV   21 01-Mar-77      TT     .SYS     2 01-Mar-77
LINK   .SAV   25 01-Mar-77
 11 Files, 265 Blocks
 215 Free blocks

*DXO:/S:POS
 13-Apr-77
DXMNSJ.SYS    91 01-Mar-77      DUP    .SAV    17 04-Mar-77
TT     .SYS    2 01-Mar-77      DIR    .SAV    16 15-Mar-77
LP     .SYS    2 01-Mar-77      PIP    .SAV    16 16-Mar-77
DT     .SYS    2 01-Mar-77      MACRO  .SAV    46 01-Mar-77
EDIT   .SAV   21 01-Mar-77      SYSMAC.MAC     27 18-Feb-77
LINK   .SAV   25 01-Mar-77
 11 Files, 265 Blocks
 215 Free blocks

*DXO:/S:TYP
 13-Apr-77
SYSMAC.MAC    27 18-Feb-77      PIP    .SAV    16 16-Mar-77
DIR    .SAV   16 15-Mar-77      DT     .SYS     2 01-Mar-77
DUP    .SAV   17 04-Mar-77      DXMNSJ.SYS     91 01-Mar-77
EDIT   .SAV   21 01-Mar-77      LP     .SYS     2 01-Mar-77
LINK   .SAV   25 01-Mar-77      TT     .SYS     2 01-Mar-77
MACRO  .SAV   46 01-Mar-77
 11 Files, 265 Blocks
 215 Free blocks

*DXO:/S:SIZ
 13-Apr-77
DT     .SYS    2 01-Mar-77      EDIT   .SAV    21 01-Mar-77
LP     .SYS    2 01-Mar-77      LINK   .SAV    25 01-Mar-77
TT     .SYS    2 01-Mar-77      SYSMAC.MAC     27 18-Feb-77
DIR    .SAV   16 15-Mar-77      MACRO  .SAV    46 01-Mar-77
PIP    .SAV   16 16-Mar-77      DXMNSJ.SYS     91 01-Mar-77
DUP    .SAV   17 04-Mar-77
 11 Files, 265 Blocks
 215 Free blocks
```

CHAPTER 10
MACRO-11 PROGRAM ASSEMBLY

This chapter describes how to assemble MACRO-11 programs under the RT-11 operating system, assuming that you have written those programs according to the rules stated in the *PDP-11 MACRO-11 Language Reference Manual*, used associated debugging tools and the linker (see Chapter 11), and understand the RT-11 operating system.

The MACRO-11 assembler operates in two distinct phases, or passes. Chapter 1 of the *PDP-11 MACRO-11 Language Reference Manual* contains a detailed description of the two-pass assembler action.

The assembly output includes any or all of the following items:

1. A binary object file – the machine-readable logical equivalent of the MACRO-11 assembly language source code
2. A listing of the source input file
3. A cross-reference file listing
4. A table of contents listing
5. A symbol table listing

To use the MACRO-11 assembler correctly under RT-11 control, you should understand how to:

1. Initiate and terminate the MACRO-11 assembler (including how to format command strings to specify files MACRO-11 uses during assembly)
2. Assign temporary work files to non-default devices, if necessary
3. Use file specification options to override file control directives in the source program
4. Use the small version of MACRO-11 for PDP-11 systems with 8K memory, if necessary
5. Interpret error messages

The following sections describe these topics.

10.1 INITIATING THE MACRO-11 ASSEMBLER

To call the MACRO-11 assembler from the system device, respond to the system prompt (a dot printed by the keyboard monitor) by typing:

```
R MACRO (RET)
```

When the assembler responds with an asterisk (*), it is ready to accept command string input. (You can also call the assembler using the keyboard monitor MACRO command; see Chapter 4 for a description of this command.)

The assembler now expects a command string consisting of the following items, in sequence

1. Output file specifications
2. An equal sign
3. Input file specifications

Format this command string as follows (punctuation is required where shown):

```
dev:obj,dev:list,dev:cref/s:arg=dev:sourcei, . . . ,dev:sourcen/s:arg (RET)
```

where

dev	is any legal RT-11 device for output; any file-structured device for input
obj	is the file specification of the binary object file that the assembly process produces; the dev for this file should not be TT or LP
list	is the file specification of the assembly and symbol listing that the assembly process produces
cref	is the file specification of the CREF temporary cross-reference file that the assembly process produces. (Omission of device:cref does not preclude a cross-reference listing, however.)
/s:arg	is a set of file specification options and arguments. Section 10.2 describes these options and associated arguments. Before that section, they are omitted from examples.
sourcei	Each sourcei is a file specification for an ASCII MACRO-11 source file or MACRO library file. These files contain the MACRO language programs that you need to assemble. You can specify as many as six source files.

The following command string calls for an assembly that uses one source file plus the system MACRO library to produce an object file BINF.OBJ and a listing. The listing goes directly to the line printer.

```
*DK:BINF.OBJ,LP:=DK:SRC.MAC
```

All output file specifications are optional. The system does not produce an output file unless the command string contains a specification for that file.

The system determines the file type of an output file specification by its position in the command string, as determined by the number of commas in the string. For example, to produce only a listing, and no object file, you must include an empty object specification.

To omit the object file, you must begin the command string with a comma. The following command produces a listing, including cross-reference tables, but not binary object files.

```
*,LP:/C=(source file specification)
```

Notice that you need not include a comma after the final output file specification in the command string.

Table 10-1 lists the default values for each file specification.

10.2 TERMINATING THE MACRO-11 ASSEMBLER

If you have typed R MACRO and received the asterisk prompt but have not yet entered the command string, you can terminate MACRO-11 control by typing CTRL/C once. After you have completed the command string (thus beginning an assembly) you can halt the assembly process at any time by typing CTRL/C twice. This returns control to the system monitor, and a system monitor dot prompt appears on the terminal.

To restart the assembly process, type R MACRO in response to the system monitor prompt. You can also restart using the REENTER command in most cases; however, the RT-11 system does not accept the REENTER command if the assembler is producing a cross-reference listing when you halt the assembly.

Table 10-1 Default File Specification Values

File	Default Device	Default File Name	Default File Type
Object	DK:	Must specify	.OBJ
Listing	Same as for object file	Must specify	.LST
Cref	DK:	Must specify	.TMP
First source	DK:	Must specify	.MAC
Additional source	Same as for preceding source file	Must specify	.MAC
System MACRO Library	System device SY:	SYSMAC	.SML
User MACRO Library	DK: if first file, otherwise same as for preceding source file	Must specify	.MAC

10.3 TEMPORARY WORK FILE

Some assemblies need more symbol table space than available memory can contain. When this occurs the system automatically creates a temporary work file called WRK.TMP to provide extended symbol table space.

The default device for WRK.TMP is DK. To cause the system to assign a different device, enter the following command:

```
.ASSIGN dev: WF
```

The dev parameter is the logical name of a file-structured device. The system assigns WRK.TMP to this device.

10.4 FILE SPECIFICATION OPTIONS

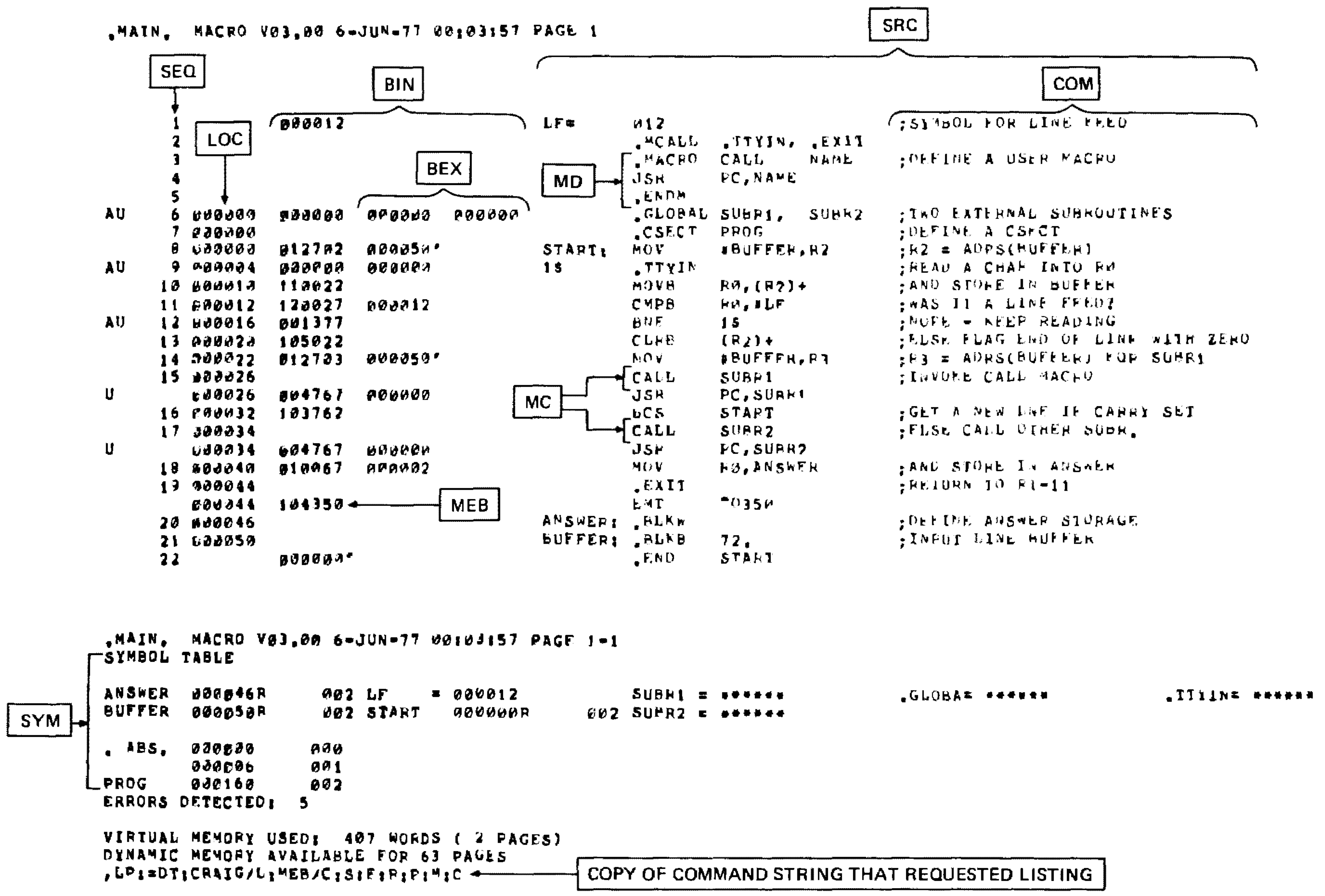
At assembly time you may need to override certain MACRO directives appearing in the source programs. You may also need to direct MACRO-11 on the handling of certain files during assembly. You can satisfy these needs by including special options in the MACRO-11 command string in addition to the file specifications. Table 10-2 lists the options and describes generally the effect of each.

The general format of the MACRO-11 command string is repeated below for your convenience:

```
dev:obj,dev:list,dev:cref/s:arg=dev:source1,. . . ,dev:sourcen/s:arg
```

Table 10-2 File Specification Options

Option	Usage
/L:arg	Listing control, overrides source program directive .LIST
/N:arg	Listing control, overrides source program directive .NLIST
/E:arg	Object file function enabling, overrides source program directive .ENABL
/D:arg	Object file function disabling, overrides source program directive .DSABL
/M	Indicates input file is MACRO library file
/C:arg	Control contents of cross-reference listing
/P:arg	Specifies whether input source file is to be assembled during pass 1 or pass 2


SRC
SEQ
BIN
COM
LOC
BEX
MD
MC
MEB
SYM
COPY OF COMMAND STRING THAT REQUESTED LISTING

```
.MAIN.  MACRO V03.00 6-JUN-77 00:03:57 PAGE 1

      1               000012                       LF=      012                   ;SYMBOL FOR LINE FEED
      2                                                     .MCALL   .TTYIN, .EXIT
      3                                                     .MACRO   CALL     NAME  ;DEFINE A USER MACRO
      4                                                     JSR      PC,NAME
      5                                                     .ENDM
AU    6 000000        000000  000000  000000                .GLOBAL  SUBR1,   SUBR2 ;TWO EXTERNAL SUBROUTINES
      7 000000                                              .CSECT   PROG           ;DEFINE A CSECT
      8 000000        012702  000050'              START:   MOV      #BUFFER,R2     ;R2 = ADRS(BUFFER)
AU    9 000004        000000  000000               1$       .TTYIN                  ;READ A CHAR INTO R0
     10 000010        110022                                MOVB     R0,(R2)+       ;AND STORE IN BUFFER
     11 000012        120027  000012                        CMPB     R0,#LF         ;WAS IT A LINE FEED?
AU   12 000016        001377                                BNE      1$             ;NOPE - KEEP READING
     13 000020        105022                                CLRB     (R2)+          ;ELSE FLAG END OF LINE WITH ZERO
     14 000022        012703  000050'                       MOV      #BUFFER,R3     ;R3 = ADRS(BUFFER) FOR SUBR1
     15 000026                                              CALL     SUBR1          ;INVOKE CALL MACRO
U       000026        004767  000000                        JSR      PC,SUBR1
     16 000032        103762                                BCS      START          ;GET A NEW LNE IF CARRY SET
     17 000034                                              CALL     SUBR2          ;ELSE CALL OTHER SUBR.
U       000034        004767  000000                        JSR      PC,SUBR2
     18 000040        010067  000002                        MOV      R0,ANSWER      ;AND STORE IN ANSWER
     19 000044                                              .EXIT                   ;RETURN TO RT-11
        000044        104350                                EMT      ^O350
     20 000046                                     ANSWER:  .BLKW                   ;DEFINE ANSWER STORAGE
     21 000050                                     BUFFER:  .BLKB    72.            ;INPUT LINE BUFFER
     22               000000'                               .END     START


.MAIN.  MACRO V03.00 6-JUN-77 00:03:57 PAGE 1-1
SYMBOL TABLE

ANSWER  000046R      002 LF     = 000012           SUBR1 = ******          .GLOBA= ******          .TTYIN= ******
BUFFER  000050R      002 START    000000R     002 SUBR2 = ******

. ABS.  000000      000
        000000      001
PROG    000160      002
ERRORS DETECTED:  5

VIRTUAL MEMORY USED:  407 WORDS ( 2 PAGES)
DYNAMIC MEMORY AVAILABLE FOR 63 PAGES
,LP:=DT:CRAIG/L:MEB/C:S:F:R:P:M:C
```

Figure 10-1 Sample Assembly Listing

The /M and /P options affect only the particular source file specification to which they are directly appended in the command string.

Other options are unaffected by their placement in the command string. The /L option, for example, affects the listing file, regardless of where you place it in the command string.

The following subsections describe in detail how to use the several file specification options.

10.4.1 Listing Control Options

Two options, /L:arg and /N:arg, pertain to listing control. By specifying these options with a set of selected arguments (see Table 10-3) you can control the content and format of assembly listings. You can override at assembly time the arguments of .LIST and .NLIST directives in the source program.

Figure 10-1 shows an assembly listing of a small program. This listing shows the more important listing features. It labels each feature with the mnemonic ASCII argument that determines its appearance on the listing; the argument SEQ, for instance, controls the appearance of the source line sequence numbers.

Specifying the /N option with no argument causes the system to list only the symbol table, the table of contents, and error messages.

Specifying the /L option with no arguments causes the system to ignore .LIST and .NLIST directives that have no arguments.

The following example lists binary code throughout the assembly using the 132-column line printer format, and suppresses the symbol table listing.

```
*I,LP:/L:MEB/N:SYM=FILE
```

Table 10-3 Valid Arguments for /L and /N Options

Argument	Default	Controls Listing of
SEQ	list	Source line sequence number
LOC	list	Address location counter
BIN	list	Generated binary code
BEX	list	Binary extensions
SRC	list	Source code
COM	list	Comment
MD	list	Macro definitions, repeat range expansion
MC	list	Macro calls, repeat range expansion
ME	no list	Macro expansions
MEB	no list	Macro expansion binary code
CND	list	Unsatisfied conditionals, .IF and .ENDC statements
LD	no list	List control directives with no arguments
TOC	list	Table of Contents
TTM	no list	132-column line printer format when not specified, terminal mode when specified
SYM	list	Symbol table

10.4.2 Function Control Options

Two options, /E:arg and /D:arg allow you to enable or disable functions at assembly time, and thus influence the form and content of the binary object file. These functions can override ENABL and DSABL directives in the source program.

Table 10-4 summarizes the acceptable /E and /D function arguments, their normal default status, and the functions they control. See Section 5.5.2 for further details of the functions.

Table 10-4 Valid Arguments for /E and /D Options

Argument	Default Mode	Function
ABS	Disable	Allows absolute binary output
AMA	Disable	Assembles all absolute addresses as relative addresses
CDR	Disable	Treats all source information beyond column 72 as commentary
CRF	Enable	Allows cross-reference listing. Disabling this function inhibits CREF output if option /C is active
FPT	Disable	Truncates floating point values (instead of rounding)
GBL	Disable	Treats undefined symbols as globals
LC	Disable	Allows lower case ASCII source input
LSB	Disable	Allows local symbol block
PNC	Enable	Allows binary output
REG	Enable	Allows mnemonic definitions of registers

For example, if you type the following commands the system assembles a file while treating columns 73 through 80 of each source card as commentary.

```
.R PIP
*CARDS.MAC=CR:/A
*^C
.R MACRO
*,LP:=CARDS.MAC/E:CDR
```

Because MACRO-11 is a two-pass assembler, you cannot read the cards directly from the card reader or other non-file structured device. You must use PIP (or the keyboard monitor COPY command) to transfer input to a file-structured device before beginning the assembly.

Use either the function control or listing control option and arguments at assembly time to override corresponding listing or function control directives in the source program. For example, assume that the source program contains the following sequence:

```
.NLIST MEB
   .
   · (MACRO references)
   .
.LIST MEB
```

In this example, you disable the listing of macro expansion binary code for some portion of the code and subsequently resume MEB listing. However, if you indicate /L:MEB in the assembly command string, the system ignores both the .NLIST MEB and the .LIST MEB directives. This enables MEB listing throughout the program.

10.4.3 Macro Library File Designation Option
The /M option is meaningful only if appended to a source file specification. It has no arguments, and it designates its associated source file as a macro library.

If the command string does not include the standard system macro library SYSMAC.SML, the system automatically includes it as the last source file in the command string.

When the assembler encounters an .MCALL directive in the source code, it searches macro libraries according to their order of appearance in the command string. When it locates a macro record whose name matches that given in the .MCALL, it assembles the macro as indicated by that definition. Thus if two or more macro libraries contain definitions of the same macro name, the macro library that appears leftmost in the command string takes precedence.

Consider the following command string:

```
*(output file specification)=ALIB.MAC/M,BLIB.MAC/M,XIZ
```

Assume that each of the two macro libraries, ALIB and BLIB, contain a macro called .BIG, but with different definitions. Then, if source file XIZ contains a macro call .MCALL .BIG, the system includes the definition of .BIG in the program as it appears in the macro library ALIB.

Moreover, if macro library ALIB contains a definition of a macro called .READ, that definition of .READ overrides the standard .READ macro definition in SYSMAC.SML.

10.4.4 Cross-Reference (CREF) Table Generation Option
A cross-reference (CREF) table lists all or a subset of the symbols in a source program, identifying the statements that define and use symbols.

10.4.4.1 Obtaining a Cross-Reference Table – To obtain a CREF table you must include the /C:arg option in the command string. Usually you include the /C:arg option with the assembly listing file specification. You can in fact place it anywhere in the command string.

If the command string does not include a cref file specification, the system automatically generates a temporary file on device DK:. If you need to have a device other than DK: contain the temporary cref file, you must include the dev:cref field in the command string.

If the listing device is magtape or cassette, load the handler for that device before issuing the command string, using the monitor LOAD command (described in Chapter 4).

A complete CREF listing contains the following six sections:

1. A cross reference of program symbols; that is, labels used in the program and symbols followed by a – operator.
2. A cross reference of register equate symbols; that is, symbols defined in the program by the construct:

   symbol–n

   with 0>n>7.

   Normally, these symbols include R0, R1, R2, R3, R4, R5, SP, and PC.

3. A cross reference of MACRO symbols; that is, those symbols defined by .MACRO and .MCALL directives.
4. A cross reference of permanent symbols, that is, all operation mnemonics and assembler directives.
5. A cross reference of program sections. These symbols include the names you specify as operands of .CSECT or .PSECT directives.
6. A cross reference of errors. The system groups and lists all flagged errors from the assembly by error type.

You can include any or all of these six sections on the cross-reference listing by specifying the appropriate arguments with the /C option. These arguments are listed and described in Table 10-5.

Table 10-5 /C Option Arguments

Argument	CREF Section
S	User defined symbols
R	Register symbols
M	MACRO symbolic names
P	Permanent symbols including instructions and directives
C	Control and program sections
E	Error code grouping

NOTE

Specifying /C with no arguments is equivalent to specifying /C:S:M:E. That special case excepted, you must explicitly request each CREF section by including its arguments. No cross-reference file occurs if the /C option is not specified, even if the command string includes a CREF file specification.

10.4.4.2 Handling Cross-Reference Table Files – When you request a cross-reference listing by means of the /C option, you cause the system to generate a temporary file, DK:CREF.TMP.

If device DK: is write-locked or if it contains insufficient free space for the temporary file, you can allocate another device for the file. To allocate another device, specify a third output file in the command string; that is, include a dev:cref specification. (You must still include the /C option to control the form and content of the listing. The dev:cref specification is ignored if the /C option is not also present in the command string.)

The system then uses the dev:cref file instead of DK:CREF.TMP and deletes it automatically after producing the CREF listing.

The following command string causes the system to use RK2:TEMP.TMP as the temporary CREF file.

```
*,LP:,RK2:TEMP.TMP=SOURCE/C
```

Another way to assign an alternative device for the CREF.TMP file is to enter the following command prior to entering R MACRO:

```
.ASSIGN dev:CF
```

This method is preferred if you intend to do several assemblies, as it relieves you from having to include the dev:cref specification in each command string. If you enter the ASSIGN dev: CF command, and later include a cref specification in a command string, the specification in the command string prevails for that assembly only.

The system lists requested cross-reference tables following the MACRO assembly listing. Each table begins on a new page. (Figure 10-2 combines the tables to save space, however.)

The system prints symbols and also symbol values, control sections, and error codes, if applicable, beginning at the left margin of the page. References to each symbol are listed on the same line, left-to-right across the page. The system lists references in the form p-l; where p is the page in which the symbol, control section, or error code appears, and l is the line number on the page.

A number sign (#) next to a reference indicates a symbol definition. An asterisk (*) next to a reference indicates a destructive reference – that is, an operation that alters the contents of the addressed location.

10.4.5 Assembly Pass Option

The /P:arg option is meaningful only if appended to a source input file specification. You must specify either of two arguments with it: 1 or 2.

The specification /P:1 calls for assembly of the file during pass 1 only. Some files consist entirely of code that is completely assembled at the end of pass 1. By specifying /P:1 for these files, you can cause MACRO-11 to skip processing of these files through pass 2. In some cases this procedure can save considerable assembly time.

The specification /P:2 calls for assembly of the file during pass 2 only. (NOTE: Situations where the /P:2 option can be meaningfully employed are unusual.)

10.5 MACRO-11 8K VERSION

A subset version of MACRO-11, with file name MAC8K.SAV, is available for systems with 8K words of memory – that is, systems with insufficient memory to support operation of the full MACRO-11 assembler.

The full assembler (MACRO) requires approximately 10K words of memory, or must be operating on at least a 12K system using the single-job (SJ) monitor.

The subset version (MAC8K) requires approximately 6K words of memory, or must be operating on an 8K system using the baseline SJ monitor.

The subset version differs from the full assembler as follows:

1. All handlers must be resident (that is, loaded) before you call MAC8K.
2. The full assembler prints the input command string at the end of the listing; the subset version does not.
3. The subset version does not recognize the following items:
   a. The operation codes exclusive to PDP-11/45 and PDP-11/70
   b. The Commercial Instruction Set (CIS)
   c. The FLT2 and FLT4 floating point directives
4. The system device is the only available file medium under MAC8K.
5. The subset version does not support the cross-reference file and ignores attempts to obtain such a listing.
6. Assembly times of the subset version are noticeably longer.
7. The subset version operates only under control of the baseline single-job monitor (see the *RT-11 System Generation Manual*).

10.6 MACRO-11 ERROR CODES

The MACRO-11 system prints diagnostic error codes as the first character of a source line on which the assembler detects an error. This error code identifies the type of error; for example, a code of M indicates a multiple definition of a label. Table 10-6 shows the error codes that might appear on an assembly listing. For detailed information on error code interpretation and debugging, see the *MACRO-11 Language Reference Manual.*

```
.MAIN.  MACRO V03.00 6-JUN-77 00:03:57 PAGE S-1
CROSS REFERENCE TABLE (CREF V01-05 )


.GLOBA   1-6
.TTYIN   1-9
ANSWER   1-18*     1-20#
BUFFER   1-8       1-14      1-21#
LF       1-1#      1-11
START    1-9#      1-16      1-22
SUBR1    1-6       1-15
SUBR2    1-6       1-17



.MAIN.  MACRO V03.00 6-JUN-77 00:03:57 PAGE R-1
CROSS REFERENCE TABLE (CREF V01-05 )


PC       1-15*     1-17*
R0       1-10      1-11      1-18
R2       1-9*      1-10*     1-13*
R3       1-14*



.MAIN.  MACRO V03.00 6-JUN-77 00:03:57 PAGE M-1
CROSS REFERENCE TABLE (CREF V01-05 )


.EXIT    1-2#      1-19
.TTYIN   1-2#
CALL     1-3#      1-15      1-17



.MAIN.  MACRO V03.00 6-JUN-77 00:03:57 PAGE P-1
CROSS REFERENCE TABLE (CREF V01-05)


.BLKB    1-21
.BLKW    1-20
.CSECT   1-7
.END     1-22
.MACRO   1-3
.MCALL   1-2
BCS      1-16
BNE      1-12
CLRB     1-13
CMPB     1-11
EMT      1-19
JSR      1-15      1-17
MOV      1-8       1-14      1-18
MOVB     1-10



.MAIN.  MACRO V03.00 6-JUN-77 00:03:57 PAGE C-1
CROSS REFERENCE TABLE (CREF V01-05 )


         0-0
. ABS.   0-0
PROG     1-7



.MAIN.  MACRO V03.00 6-JUN-77 00:03:57 PAGE E-1
CROSS REFERENCE TABLE (CREF V01-05 )


A        1-6       1-9       1-12
U        1-6       1-9       1-12      1-15      1-17
```

Figure 10-2 Cross-Reference Table

Table 10-6 MACRO-11 Error Codes

Error Code	Meaning
A	Addressing or relocation error. This occurs when an instruction operand has an invalid address, or when the definition of a local symbol occurs more than 128 words from the beginning of a local symbol block.
B	Boundary error. The current setting of the location counter would cause the assembly of instruction or word data at an odd memory address. The system increments the location counter by 1 to correct this.
D	Reference to multiple-definition symbol. The program refers to a non-local label that is defined more than once.
E	No END directive. The assembler has reached the end of a source file and found no END directive. The system generates .END and continues.
I	Illegal character detected. The assembler has encountered in the source file a character that is not included in the language character set. The system replaces each illegal character with a ? on the assembly listing and proceeds as if the illegal character were not present.
L	Link buffer overflow. The assembler has encountered an input line greater than 132 characters. In terminal mode the system ignores additional characters.
M	Multiple definition of a label. The source program is attempting to define a label equivalent in the first six characters to a label defined previously.
N	Decimal point missing from decimal number. A number containing the digit 8 or 9 lacks a decimal point.
O	Op-code error. A directive appears in an inappropriate context.
P	Phase error. The definition or value of a label differs from one pass to another, or a local symbol occurs more than once in a local symbol block.
Q	Questionable syntax. This can have any of several causes, as follows: 1. There are missing arguments. 2. The instruction scan is not complete. 3. A line feed or form feed does not immediately follow a carriage return.
R	Register-type error. The source program attempts an invalid reference to a register.
T	Truncation error. A number generates more than 16 significant bits, or an expression generates more than 8 significant bits while a .BYTE directive is active.
U	Undefined symbol. A symbol not defined elsewhere in the program appears as a factor in an expression. The assembler assigns the undefined symbol a constant zero value.
Z	Incompatible instruction (warning). The instruction is not defined for all PDP-11 hardware configurations.

CHAPTER 11
LINKER (LINK)

The RT-11 linker (LINK) converts object modules produced by an RT-11 supported language translator into a format suitable for loading and execution. If you have no previous experience with the linker, read Chapter 12 of the *Introduction to RT-11* for an introductory-level description of the linking process. You can separately assemble a main program and each of its subroutines without assigning an absolute load address at assembly time. The linker processes the object modules of the main program and subroutines to:

- Relocate each object module and assign absolute addresses
- Link the modules by correlating global symbols that are defined in one module and referenced in another
- Create the initial control block for the linked program that the GET, R, RUN, and FRUN commands use
- Create an overlay structure if specified and include the necessary run-time overlay handlers and tables
- Search libraries you specify to locate unresolved globals
- Automatically search a default system library to locate any remaining unresolved globals
- Produce a load map showing the layout of the load module
- Produce a symbol definition file.

The RT-11 linker requires two passes over the input modules. During the first pass it constructs the symbol table, including all program section names and global symbols in the input modules. After it processes all non-library files, the linker scans the library files to resolve undefined globals. It links only those modules that are required into the root segment (that part of the program that is never overlaid). During the final pass, the linker reads the object modules, performs most of the functions listed above, and produces a load module (which is in memory image format for background jobs or for jobs that run in the single-job environment, relocatable format for foreground jobs, and formatted binary for use with the Absolute Loader).

The linker runs in a minimal RT-11 system of 8K words of memory; the linker uses any additional memory to facilitate efficient linking and to extend the size of the symbol table. The linker accepts input from any random-access device on the system; there must be at least one random-access device (disk or DECtape) for memory image or relocatable format output.

11.1 CALLING AND USING THE LINKER

To call the RT-11 linker from the system device, respond to the dot printed by the keyboard monitor by typing:

```
    R LINK(RET)
```

The Command String Interpreter prints an asterisk at the left margin on the console terminal when it is ready to accept a command line. If you enter only a carriage return at this point, the linker prints its current version number.

Type two CTRL/Cs to halt the linker at any time (or a single CTRL/C to halt the linker when it is waiting for console terminal input) and return control to the monitor. To restart the linker, type R LINK or REENTER in response to the monitor's dot.

The first command string you enter in response to the linker's prompt has this syntax:

```
[binout-filespec],[mapout-filespec],[stbout-filespec] = obj-filespec[/option . . .] [, . . . obj-filespec[/option . . .]]
```

where

binout-filespec	represents the device, name and file type to be assigned to the linker's output load module file.
mapout-filespec	represents the device, file name and file type of the load map output file.
stbout-filespec	represents the device, file name and file type of the symbol definition file.
obj-filespec	represents an object module (that can be a library file) to be linked.
/option	is one of the options from Table 11-2.

In each filespec above, the device should be a random access device, with these exceptions: the output device for the load map file can be any RT-11 device, as can the output device for an .LDA file if you use the /L option. If you do not specify a device, the linker uses default device DK:. Note that the linker load map contains lower case characters. Use the SET LP LC command to enable lower case printing if your printer has lower case characters.

If you do not specify an output file, the linker assumes that you do not desire the associated output. For example, if you do not specify the load module and load map (by using a comma in place of each file specification) the linker prints only error messages, if any occur.

Table 11-1 shows the default values for each specification.

Table 11-1 Linker Defaults

	Device	File Name	File Type
Load Module	DK:	none	SAV, REL(/R), LDA(/L)
Map Output	Same as load module	none	MAP
Symbol Definition Output	DK: or same as previous output device	none	STB
Object Module	DK: or same as previous object module	none	OBJ

If you make a syntax error in a command string, the system prints an error message. You can then type a new command string following the asterisk. Similarly, if you specify a nonexistent file, a warning error occurs; control returns to the Command String Interpreter, an asterisk prints and you can enter a new command string.

11.2 OPTIONS SUMMARY

Table 11-2 lists the options associated with the linker. You must precede the letter representing each option by the slash character. Options must appear on the line indicated if you continue the input on more than one line, but you can position them anywhere on the line. (Section 11.8 provides a more detailed explanation of each option.)

Table 11-2 Linker Options

Option Name	Command Line	Section	Explanation
/B:n	first	11.8.1	Changes the bottom address of a program to n (illegal for foreground links).
/C	any but last	11.8.2	Continues input specification on another command line (you can use /C also with /O; do not use /C with the // option).
/E:n	first	11.8.3	Extends a particular program section to a specific value.
/F	first	11.8.4	Instructs the linker to use the default FORTRAN library, FORLIB.OBJ; this option is provided only for compatibility with previous versions of RT-11.
/H:n	first	11.8.5	Specifies the top (highest) address to be used by the relocatable code in the load module.
/I	first	11.8.6	Extracts the global symbols you specify (and their associated object modules) from the library and links them into the load module.
/K:n	first	11.8.7	Inserts the value you specify (the valid range for n is from 1 to 28) into word 56 of block 0 of the image file; this option is provided only for compatibility with the RSTS operating system.
/L	first	11.8.8	Produces a formatted binary output file (illegal for foreground links).
/M or /M:n	first	11.8.9	Causes the linker to prompt you for a global symbol that represents the stack address, or sets the stack address to the value n.
/O:n	any but the first	11.8.10	Indicates that the program is an overlay structure; n specifies the overlay region to which the module is assigned.
/P:n	first	11.8.11	Changes the default amount of space the linker uses for a library routines list.
/R[:n]	first	11.8.12	Produces output in relocatable format and can indicate stack size for a foreground job.
/S	first	11.8.13	Makes the maximum amount of space in memory available for the linker's symbol table. (You only need to use this option when a particular link stream causes a symbol table overflow.)

(Continued on next page)

Table 11-2 (Cont.) Linker Options

Option Name	Command Line	Section	Explanation
/T or /T:n	first	11.8.14	Causes the linker to prompt you for a global symbol that represents the transfer address, or sets the transfer address to the value n.
/U:n	first	11.8.15	Rounds up the section you specify so that the size of the root segment is a whole number multiple of the value you supply (n must be a power of 2).
/W	first	11.8.16	Directs the linker to produce a wide load map listing.
/X		11.8.17	Does not output the bitmap if the code is below 400; this option is provided only for compatibility with the RSTS operating system.
/Y:n	first	11.8.18	Starts a specific program section on a particular address boundary.
/Z:n	first	11.8.19	Sets unused locations in the load module to the value n.
//	first and last	11.8.20	Allows you to specify command string input on additional lines. Do not use this option with /C.

11.3 MEMORY ALLOCATION

The linker allocates the physical memory and address space that the load module requires. The area of memory that the linker allocates for a load module contains the following elements:

- a system communication area
- hardware vectors
- a stack
- a set of named areas called program sections (p-sections).

Section 11.5.2 describes the system communication area.

The stack is an area that a program can use for temporary storage and subroutine linkage. General register 6, the stack pointer (SP), references the stack.

The system communication area, the hardware vectors, and the stack areas are all part of the load module area called the absolute section. The absolute section is often called the ASECT because it is the assembler directive .ASECT that allows information to be stored there. This section appears in the load map with the name .ABS. and is always the first section in the listing. The absolute section (ASECT) normally ends at address 1000 (octal).

A program section is an area of the load module that contains code and/or data; you can reference it by name. The set of attributes associated with each p-section controls the allocation and placement of the section within the load module.

A p-section is the basic unit of memory for a program. It is composed of the following elements:

- a name by which it can be referenced
- a set of attributes that defines its contents, mode of access, allocation, and placement in memory
- a length that determines how much storage is reserved for the p-section.

You create p-sections by using the COMMON statement in FORTRAN, or the .PSECT (or .CSECT) directive in MACRO. You can use the .PSECT (or .CSECT) directive to attach attributes to the section. Note that the attributes that follow the p-section name are not part of the name; only the name itself distinguishes one p-section from another. You should make sure, then, that p-sections of the same name that you want to link together also have the same attribute list. Do this because the linker uses the first appearance of the .PSECT and its attributes throughout the operation. If the linker encounters p-sections with the same name that have different attributes, it prints a warning message.

The linker collects from the input modules scattered references to a p-section and combines them in a single area of the load module. The attributes, which are listed in Table 11-3, control the way the linker collects and places this unit of storage.

Table 11-3 P-section Attributes

Attribute	Value	Explanation
access-code[1]	RW	Read/Write – data can be read from, and written into, the p-section.
	RO	Read Only – data can be read from, but cannot be written into, the p-section.
type-code	D	Data – the p-section contains data.
	I	Instruction – the p-section contains either instructions, or data and instructions.
scope-code	GBL	Global – the p-section name is recognized across overlay segment boundaries. The linker allocates storage for the p-section from references outside the overlay segment.
	LCL	Local – the p-section name is recognized only within each individual overlay segment. The linker allocates storage for the p-section from references within the overlay segment only.
reloc-code	REL	Relocatable – the base address of the p-section is relocated relative to the virtual base address of the program.
	ABS	Absolute – the base address of the p-section is not relocated. It is always 0.
alloc-code	CON	Concatenate – all allocations to a given p-section name are concatenated. The total allocation is the sum of the individual allocations.
	OVR	Overlay – all allocations to a given p-section name overlay each other. The total allocation is the length of the longest individual allocation.

[1]Not used by the linker.

The scope-code and type-code are meaningful only when you define an overlay structure for the program. In an overlaid program, a global section is known throughout the entire program. Object modules contribute to only one global of the same name. If two or more segments contribute to a global section, then the linker allocates that global section in the root segment of the program. In contrast to global sections, local sections are only known within a particular program segment. Because of this, several local sections of the same name can appear in different segments. Thus, several object modules contributing to a local section do so only within each segment. An example of a global section is named COMMON in FORTRAN. An example of a local section is the default blank section for each macro routine.

The alloc-code determines the starting address and length of memory allocated by modules that reference a common p-section. If the alloc-code indicates that such a p-section is to be overlaid, the linker places the allocations from each module starting at the same location in memory. It determines the total size from the length of the longest reference to the p-section. The last input module that stores information in a particular location determines which values the linker stores in the indicated locations of the load module. If the alloc-code indicates that a p-section is to be concatenated, the linker places the allocations from the modules one after the other in the load module; it determines the total allocation from the sum of the lengths of the contributions.

The allocation of memory for a p-section always begins on a word boundary. If the p-section has the D (data) and CON (concatenate) attributes, all storage that subsequent modules contribute is appended to the last byte of the previous allocation. This occurs whether or not that byte is on a word boundary. For a p-section with the I (instruction) and CON attributes, however, all storage that subsequent modules contribute begins at the nearest following word boundary.

The .CSECT directive of MACRO is converted internally by both MACRO and the linker to an equivalent .PSECT with fixed attributes. An unnamed CSECT (blank section) is the same as a blank PSECT with the following attributes: RW, I, LCL, REL, and CON.

A named CSECT is equivalent to a named PSECT with these attributes: RW, I, GBL, REL, and OVR. Table 11-4 shows these sections and their attributes.

The names assigned to p-sections are not considered to be global symbols; you cannot reference them as such. For example:

```
    MOV     #PNAME,R0
```

This statement, where PNAME is the name of a section, is illegal and generates the Undefined global error message if no global symbol of PNAME exists. A symbol can be the same for both a p-section name and a global symbol. The linker treats them separately.

The linker determines the memory allocation of p-sections by the order of occurrence of the p-sections in the input modules. The absolute section (. ABS.) always comes first, followed by the blank section of the input file (if one exists) and the named section. If there is more than one named section, the named sections appear in the same order in which they occur in the input files. For example, the FORTRAN compiler arranges the p-sections in the main program module so that the USR can swap over pure code in low memory rather than over data required by the function making the USR call.

Table 11-4 Section Attributes

	access-code	type-code	scope-code	reloc-code	alloc-code
CSECT	RW	I	LCL	REL	CON
CSECT name	RW	I	GBL	REL	OVR
ASECT	RW	I	GBL	ABS	OVR
COMMON/name/	RW	D	GBL	REL	OVR

11.4 GLOBAL SYMBOLS

Global symbols provide the link, or communication, between object modules. You create global symbols with the .GLOBL or .ENABL GBL assembler directive (or with double colon, ::, or double equal sign, ==). If the global symbol is defined in an object module (as a label using :: or by direct assignment using ==), other object modules can reference it. If the global symbol is not defined in the object module, it is an external symbol and is assumed to be defined in some other object module. If a global symbol is used as a label in a routine, it is often called an entry point. That is, it is an entry point to that subroutine.

As the linker reads the object modules it keeps track of all global symbol definitions and references. It then modifies the instructions and data that reference the global symbols. The linker always prints undefined globals on the console terminal after pass-1. If you request a load map on the terminal, they appear at the end of the load map.

Table 11-5 shows how the linker resolves global references when it creates the load module.

Table 11-5 Global Reference Resolution

Module Name	Global Definition	Global Reference
IN1	B1	A
	B2	L1
		C1
		XXX
IN2	A	B2
	B1	
IN3		B1

In processing the first module, IN1, the linker finds definitions for B1 and B2, and references to A, L1, C1, and XXX. Because no definition currently exists for these references, the linker defers the resolution of these global symbols. In processing the next module, IN2, the linker finds a definition for A that resolves the previous reference, and a reference to B2 that can be immediately resolved.

When all the object modules have been processed, the linker has three unresolved global references remaining: C1, L1, and XXX. A search of the default system library resolves XXX. The global symbols C1 and L1 remain unresolved and are, therefore, listed as undefined global symbols.

The relocatable global symbol, B1, is defined twice and is listed on the terminal as a multiply defined global symbol. The linker uses the first definition of a multiply defined symbol. An absolute global symbol can be defined more than once without being listed as multiply defined as long as each occurrence of the symbol has the same value.

11.5 INPUT AND OUTPUT

Linker input and output is in the form of modules; the linker uses one or more input modules to produce a single output (load) module.

11.5.1 Object Modules

Object files, consisting of one or more object modules, are the input to the linker (the linker ignores files that are not object modules). Object modules are created by an appropriate language translator. The module name item declares the name of the object module. The first six Radix-50 characters of the .TITLE assembler directive are used as the name of the object module. These six characters must be Radix-50 characters (the linker ignores any characters beyond the sixth character). The linker prints the first module name it encounters in the input file stream (normally the main routine of the program) on the second line of the map following .TITLE. It ignores additional module names. The linker reads each object module twice. During the first pass it reads each object module to construct a symbol table and to assign absolute values to the program section names and global symbols. The linker uses the library files to resolve undefined globals. It places their associated object modules in the root. On the second and final pass, the linker reads the object modules, links and relocates the modules and outputs the load module.

11.5.2 Load Module

The primary output of the linker is a load module that you can run under RT-11. The linker creates as a load module a memory image file (SAV) for use under a single-job system or the background job. If you need to execute a program in the foreground, use the /R option to produce a relocatable format (REL) foreground load module. The linker can produce an absolute load module (LDA) if you need to load the module with the Absolute Loader.

The load module for a memory image file is arranged as follows:

Root Segment	Overlay Segments (optional)

For a relocatable image file the load modules are arranged as follows:

Root Segment	Overlay Segments (optional)	Relocation information for root and overlay segments

The first 256-word block of the root segment (main program) contains the memory usage bit map and the locations the linker uses to pass program control parameters. The memory usage bit map outlines the blocks of memory the load module uses; it is located in locations 360 through 377.

The control parameters are located in locations 40 through 50. They contain the following information when the module is loaded:

Address	Information
40	Start address of program
42	Initial setting of SP (stack pointer)
44	Job status word (overlay bit set by LINK)
46	USR swap address (0 implies normal location)
50	Highest memory address in program (high limit)

The linker stores default values in locations 40, 42, and 50, unless you use options to specify otherwise. The /T option affects location 40, for example, and /M affects location 42. You can also use the .ASECT directive to change the defaults. The overlay bit is located in the job status word. LINK automatically sets this bit if the program is overlaid. Otherwise, the linker initially sets location 44 to 0. Location 46 also contains zero unless you specify another value by using the .ASECT directive.

For a foreground link, the following additional parameters contain information:

Address	Information
14, 16	(XM only) BPT trap
20, 22	(XM only) IOT trap
34, 36	TRAP vector
52	Size of root segment in bytes
54	Stack size in bytes (value with /R or default 128)
56	Size of overlay region in bytes
60	Identification that file is in relocatable (REL) format
62	Relative block number for start of relocation information

You can assign initial values to memory locations 0-476 (which include the interrupt vectors and system communication area) by using an .ASECT assembler directive. They appear in block 0 of the load module, but there are restrictions on the use of ASECTs in this region. You should not perform ASECTs of location 54 or of locations 360-377 because the memory usage map is passed in those locations. In addition, for foreground links, ASECTs of words 52-62 are not permitted because additional parameters are passed to the FRUN command in those locations.

You can set with an .ASECT any location that is not restricted, but be careful if you change the system communication area. The program itself must initialize restricted areas, such as the region 360-377. There are no restrictions on ASECTs if the output format is LDA.

Locations in the region 0-476 might not be loaded at execution time even though your program uses an ASECT to initialize them. For background programs, this is because the R, RUN, and GET commands do not load addresses that are protected by the monitor's memory protection map. For foreground programs, the FRUN command loads only locations 14-22 and 34-50. It ignores all other ASECTs. To initialize a location at run time, use the .PROTECT programmed request. If it is successful, follow it by a MOV instruction.

11.5.3 Load Map

If you request, the linker produces a load map following the completion of the initial pass. This map, shown in Figure 11-1, diagrams the layout of memory for the load module.

The load map lists each program section that is included in the linking process. The line for a section includes the name and low address of the section and its size in bytes. The rest of the line lists the program section attributes, as shown in Table 11-3. The remaining columns contain the global symbols found in the section and their values.

The map begins with the version of the linker, followed by the date and time the program was linked. The second line lists the file name of the program, its title (which is determined by the first module name record in the input file), and the first identification record found. The absolute section is always shown first, followed by any non-relocatable symbols. The modules located in the root segment of the load module list next, followed by those modules that were assigned to overlays in order by their region number (see Section 11.6). Any undefined global symbols then list. The map ends with the transfer address (start address) and high limit or relocatable code in both octal bytes and decimal words.

NOTE

The load map does not reflect the absolute addresses for a REL file that you create to run as a foreground job; you must add the base relocation address determined at FRUN time to obtain the absolute addresses. The linker assumes a base address of 1000.

For example, assume the FRUN command is used to run the program CARL:

```
.FRUN CARL/P
Loaded at 127276
```

The /P option causes FRUN to print the load address, which is 127276 in this example. To calculate the actual location in memory of any global in the program, first subtract 1000 from that global's value. (The value 1000 represents the base address assigned by the linker. This offset is not used at load time.) Then add the result to the load address determined with /P. The final result represents the absolute location of the global. For example, the absolute location of TIME (see Figure 11-1) is 127302 (1004-1000+127276=127302).

```
RT-11 LINK  VØ3.Ø1          Load Map           Fri Ø3-Jun-77 18:Ø3:Ø7
CARL   .REL       Title:   DEMOSP  Ident:   VØ1.Ø3

Section  Addr    Size     Global  Value    Global   Value    Global   Value

. ABS.   ØØØØØØ  ØØ1ØØØ    (RW,I,GBL,ABS,OVR)
         ØØ1ØØØ  Ø1ØØ36    (RW,I,LCL,REL,CON)
                          TIMBLK  ØØ1ØØØ   TIME     ØØ1ØØ4   DBLK     ØØ1Ø1Ø
                          LP      ØØ1Ø2Ø   AREA     ØØ1Ø24   START    ØØ1Ø36
                          BUFF    ØØ2Ø22

Transfer address = ØØ1Ø36, High limit = Ø11Ø36 = 2319.  words
```

Figure 11-1 Load Map

11.5.4 Library Files

The RT-11 linker can automatically search libraries. Libraries consist of library files, which are specially formatted files produced by the librarian program (described in Chapter 12) that contain one or more object modules. The object modules provide routines and functions to aid you in meeting specific programming needs. (For example, FORTRAN has a set of modules containing all necessary computational functions – SQRT, SIN, COS, etc.). You can use the librarian to create and update libraries. Then you can easily access routines that you use repeatedly or routines that different programs use. Selected modules from the appropriate library file are linked as needed with your program to produce one load module. Libraries are further described in Section 11.7 and in Chapter 12.

NOTE

Library files that you combine with the monitor COPY command or with the PIP /U or /B option are illegal as input to both the linker and the librarian.

11.6 USING OVERLAYS

The ability of RT-11 to use overlays gives you virtually unlimited memory space for an assembly language or FORTRAN program. A program using overlays can be much larger than would normally fit in the available memory space, since portions of the program (called overlay segments) reside on a backup storage device (disk or DECtape).

The RT-11 overlay scheme is a strict multi-region arrangement; it is not tree-structured. Figure 11-3 diagrams this scheme. The overlay system that you construct from your completed program is composed of a root segment that is always memory resident, the current memory-resident overlay segments, and the overlay segments stored on the backup storage device. The root segment is a required part of every overlay program and contains the transfer address, stack space, impure variables, data, and variables needed by many different segments; it must, therefore, never be overlaid. There is a distinct memory area for each overlay region. The overlay segments are brought into memory as they are needed. A segment consists of a set of modules and program sections. Segments that overlay each other must be logically independent; that is, the components of one segment cannot reference the components of another segment with which it shares address space. In addition to being concerned with the logical independence of the overlay segments, you should also consider the general flow of control within the program. Programs execute most efficiently when the system spends only a small amount of execution time (less than 10 percent) overlaying program segments. Figure 11-2 shows a diagram of an overlay scheme. Some examples of overlaid programs are the linker and the FORTRAN compiler.

Overlay segments that share both the same physical memory location and address space form a region. You specify overlay regions to the linker with the /O option as described in Section 11.8.7. The linker calculates the size of any region to be the size of the largest segment within that region. Thus, to reduce the size of a program (that is, the amount of memory it needs), you should first concentrate on reducing the size of the largest segment in each region. The linker creates the overlay regions and edits the program to produce the desired overlays at run-time. Figure 11-3 shows a listing of a diagram of memory showing a link with an overlay structure. Figure 11-4 shows the run-time overlay handler.

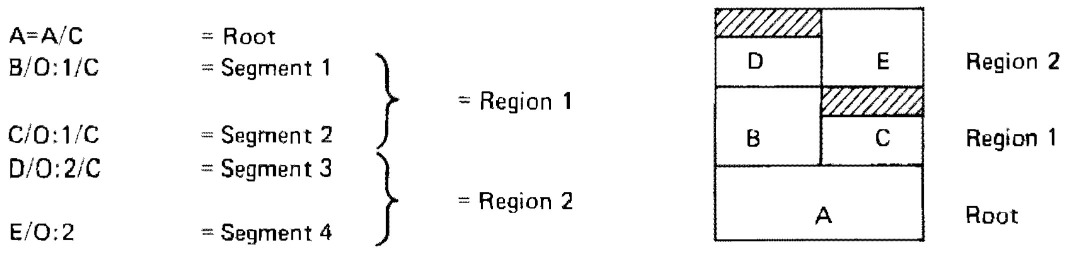


Figure 11-2 Overlay Scheme

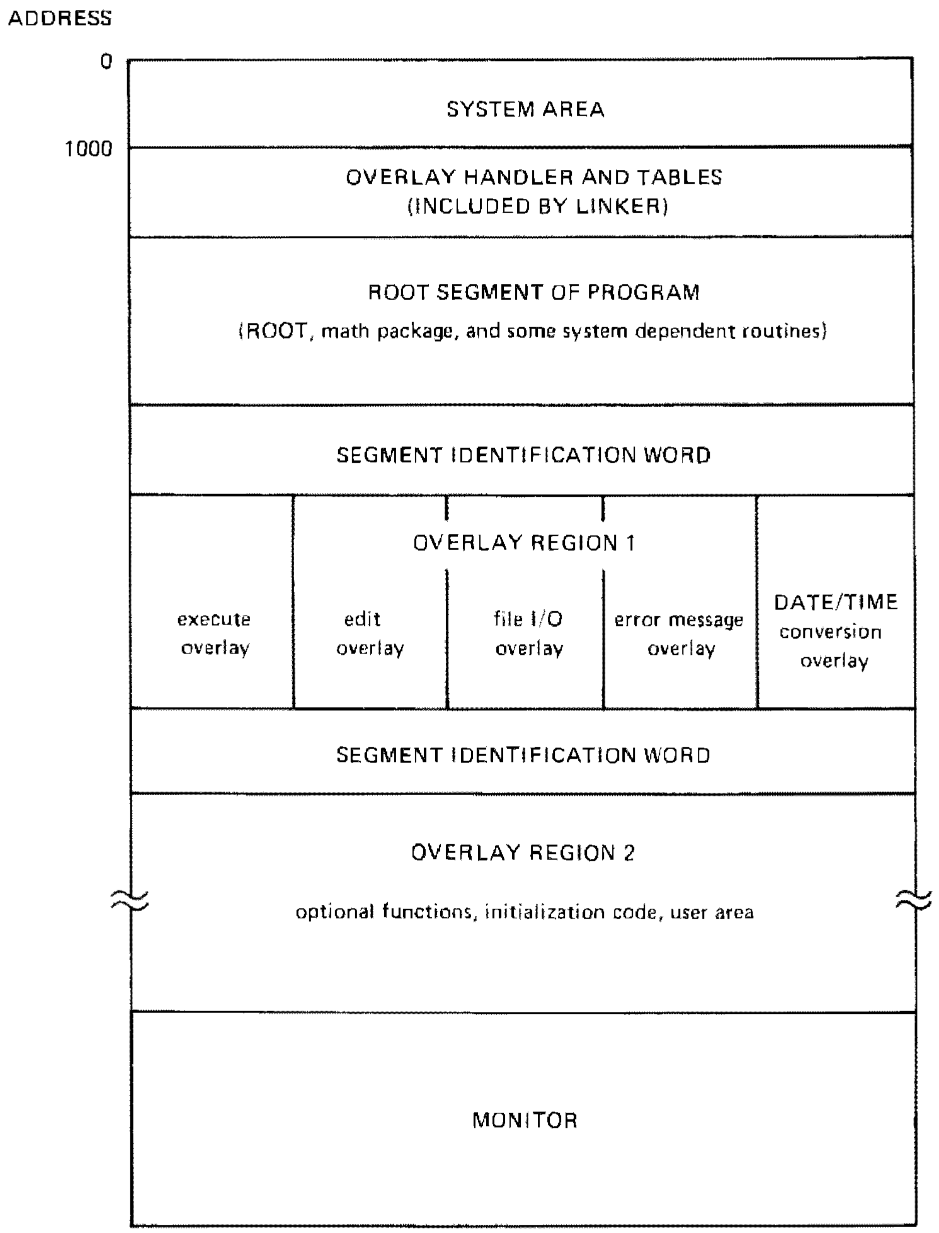

ADDRESS
0
1000
SYSTEM AREA
OVERLAY HANDLER AND TABLES (INCLUDED BY LINKER)
ROOT SEGMENT OF PROGRAM
(ROOT, math package, and some system dependent routines)
SEGMENT IDENTIFICATION WORD
OVERLAY REGION 1
execute overlay
edit overlay
file I/O overlay
error message overlay
DATE/TIME conversion overlay
SEGMENT IDENTIFICATION WORD
OVERLAY REGION 2
optional functions, initialization code, user area
MONITOR

Figure 11-3 Memory Diagram Showing BASIC Link with Overlay Regions

```
        .SBTTL  $OVRH   THE RUN-TIME OVERLAY HANDLER
;THE FOLLOWING CODE IS INCLUDED IN THE USER'S PROGRAM BY THE
;LINKER WHENEVER OVERLAYS ARE REQUESTED BY THE USER.
;THE RUN-TIME OVERLAY HANDLER IS CALLED BY A DUMMY
;SUBROUTINE OF THE FOLLOWING FORM:

;       JSR     R5,$OVRH        ;CALL TO COMMON CODE
;       .WORD   <OVERLAY #>     ;# OF DESIRED SEGMENT
;       .WORD   <ENTRY ADDR>    ;ACTUAL CORE ADDR

;ONE DUMMY ROUTINE OF THE ABOVE FORM IS STORED IN THE RESIDENT PORTION
;OF THE USER'S PROGRAM FOR EACH ENTRY POINT TO AN OVERLAY SEGMENT.
;ALL REFERENCES TO THE ENTRY POINT ARE MODIFIED BY THE LINKER TO INSTEAD
;BE REFERENCES TO THE APPROPRIATE DUMMY ROUTINE.  EACH OVERLAY SEGMENT
;IS CALLED INTO CORE AS A UNIT AND MUST BE CONTIGUOUS IN CORE.  AN
;OVERLAY SEGMENT MAY HAVE ANY NUMBER OF ENTRY POINTS, TO THE LIMITS
;OF CORE MEMORY.  ONLY ONE SEGMENT AT A TIME MAY OCCUPY AN OVERLAY REGION.

.ENABL  LSB
        $OVTAB=1000+$OVRHE-$OVRH
$OVRH:  MOV     R0,-(SP)
        MOV     R1,-(SP)
        MOV     R2,-(SP)
1$:
;       MOV     (R5)+,R0        ;PICK UP OVERLAY NUMBER
        BR      3$              ;FIRST CALL ONLY * * *
        MOV     R0,R1
$OVRHA: ADD     #$OVTAB-6,R1    ;CALC TABLE ADDR
        MOV     (R1)+,R2        ;GET CORE ADDR OF OVERLAY REGION
        CMP     R0,@R2          ;IS OVERLAY ALREADY RESIDENT?
        BEQ     2$              ;YES, BRANCH TO IT
        .READW  17,R2,(R1)+,(R1)+ ;READ FROM OVERLAY FILE
        BCS     5$
2$:     MOV     (SP)+,R2        ;RESTORE USER'S REGS
        MOV     (SP)+,R1
        MOV     (SP)+,R0
        MOV     @R5,R5          ;GET ENTRY ADDRESS
        RTS     R5              ;ENTER OVERLAY ROUTINE AND
                                ;RESTORE USER'S R5

3$:     MOV     #12500,1$       ;RESTORE SWITCH INSTR (MOV (R5)+,R0)
        MOV     (PC)+,R1        ;START ADDR FOR CLEAR OPERATION
$HROOT: .WORD   0               ;HIGH ADDR OF ROOT SEGMENT
        MOV     (PC)+,R2        ;COUNT
$HOVLY: .WORD   0               ;HIGH LIMIT OF OVERLAYS
4$:     CLR     (R1)+           ;CLEAR ALL OVERLAY REGIONS
        CMP     R1,R2
        BLO     4$
        BR      1$              ;AND RETURN TO CALL IN PROGRESS

5$:     EMT     376             ;SYSTEM ERROR 10 (OVERLAY I/O)
        .BYTE   0,373
$OVRHE:
.DSABL  LSB
;OVERLAY SEGMENT TABLE FOLLOWS:
; $OVTAB:       .WORD   <CORE ADDR>,<RELATIVE BLK>,<WORD COUNT>
;THREE WORDS PER ENTRY, ONE ENTRY PER OVERLAY SEGMENT.

;ALSO, THERE IS ONE WORD PREFIXED TO EACH OVERLAY REGION
;THAT IDENTIFIES THE SEGMENT CURRENTLY RESIDENT IN THAT REGION.
```

Figure 11-4 The Run-Time Overlay Handler

You do not need a special code or function call to use overlays. Observe the following rules when you reference parts of your program that might be overlaid.

1. You must make calls or jumps to overlay segments directly to global symbols defined in an instruction p-section (entry points). For example, if ENTER is a global symbol in an overlay segment, the first command is valid, but the second is illegal:

   ```
   JMP ENTER
   JMP ENTER+6
   ```

2. You can use globals defined in an instruction p-section (entry points) of an overlay segment only for transfer of control and not for referencing data within an overlay section. The assembler and linker cannot detect a violation of this rule so they issue no error. However, such a violation can cause the program to use incorrect data. If you reference these global symbols outside of their defining segment, the linker resolves them by using dummy subroutines of four words each in the overlay handler. If such a reference occurs, it is indicated on the load map by a "@" following the symbol.
3. The linker directly resolves global symbols that you define in a data p-section. It is your program's responsibility to load the data into memory before referencing a global symbol defined in a data section. One way to load the data section of an overlay segment is to call an entry point in that segment. This loads the segment if it is not already resident in memory.
4. When you make calls to overlays, the entire return path must be in memory. Observing the following rules accomplishes this:
   a. You can make calls (with expected return) from an overlay segment only to entries in the same segment, the root segment, or an overlay segment with a greater region number.
   b. Calls you make to entries in the same region as the call must be entirely within the same segment, not within another segment in the same region.
   c. You can make jumps (with no expected return) from an overlay segment to any entry in the program. However, jumps should not reference an overlay region whose number is lower than the region from which the last unreturned call was made (for example, if a call was made from region 3, then no jumps should reference regions 1, 2 or 3 until the call has returned).
   d. You can call subroutines in the root segment from overlay segments; in turn, they can call entries from the same overlay segment that called them, or from the root segment, or from another overlay segment with a greater region number. Such subroutines are considered to be part of the overlay segment that called them.
5. You cannot use a p-section (.PSECT or .CSECT) name to pass control to an overlay. It does not load the appropriate segment into memory. (For example, JSR PC,OVSEC is illegal if you use OVSEC as a section name in an overlay.) As stated in 1, above, you must use a global symbol to pass control from one segment to the next. If OVSEC is not a global symbol, the system flags it as an undefined global.
6. Your program cannot use channel 17 (octal) because overlays are read on that channel.
7. You cannot place object modules that are automatically acquired from a library file into overlays. The linker always places those modules in the root segment. However, you can extract modules from a library by using the librarian utility program (see Chapter 12) and then explicitly include them in any segment.
8. You cannot specify library files on the same command line as an overlay. Specify them before you enter any overlay lines.
9. You must specify overlay regions in ascending order. They are read-only. Unlike USR swapping, an overlay handler does not save the segment it is overlaying. Any tables, variables, or instructions that are modified within a given overlay segment are reinitialized to their original values in the SAV or REL file if that segment has been overlaid by another segment. You should place any variables or tables whose values must be maintained across overlays in the root segment.
10. ASECTs at location 1000 or above in an overlay foreground link are illegal; the error message ?LINK-F-Illegal ASECT prints and the link aborts.
11. A global program section that is referenced in more than one segment has its memory allocation in the root segment. This permits common access across the different segments. See Section 11.3.

Note the following information when you write FORTRAN overlays:

1. When you divide a FORTRAN program into a root segment and overlay regions (and subsequently divide each overlay region into overlay segments), you should carefully consider routine placement. Remember that it is illegal to call a routine located in a different overlay segment in the same overlay region, or an overlay region with a lower numeric value (as specified by the linker overlay option, /O:n) from the calling routine. Divide each overlay region into overlay segments that never need to be resident simultaneously. For example, if segments A and B are assigned to region X, they cannot call each other because they occupy the same locations in memory.
2. Place the FORTRAN main program unit in the root segment.
3. In an overlay environment, subroutine calls and function subprogram references can refer only to one of the following:
   a. a FORTRAN library routine (such as ASSIGN, DCOS)
   b. a FORTRAN or assembly language routine contained in the root segment
   c. a FORTRAN or assembly language routine contained in the same overlay segment as the calling routine
   d. a FORTRAN or assembly language routine contained in a segment whose region number is greater than that of the calling routine.
4. In an overlay environment, you must place the COMMON blocks so that they are resident when you reference them. Blank COMMON is always resident because it is always placed in the root segment. You must place all named COMMON either in the root segment or in the segment whose region number is lowest of all the segments that reference the COMMON block. A named COMMON block cannot be referenced by two different segments in the same region unless the COMMON block appears in a segment of a lower region number. The linker automatically places a COMMON block into the root segment if it is referenced by the FORTRAN main program or by a subprogram that is located in the root segment. Otherwise, the linker places a COMMON block in the first segment encountered in the linker command string that references that COMMON block.
5. All COMMON blocks that are initialized with DATA statements must be similarly initialized in the segment in which they are placed.

Refer to the *RT-11/RSTS/E FORTRAN IV User's Guide* for more details.

The ASECT never takes part in overlaying in any way. It is part of the root and is always resident.

The aforementioned sets of rules apply only to communications among the various modules that make up a program. Internally, each module must only observe standard programming rules for the PDP-11 (as described in the *PDP-11 Processor Handbook* and in the *MACRO-11 Language Reference Manual*).

Note that the condition codes set by your program are not preserved on the call across overlay segment boundaries. You can still use the C-bit for error returns.

The linker provides overlay services by including a small resident overlay handler in the same file with your program to be used at program run time. The linker inserts this overlay handler plus some tables into your program beginning at the bottom address. The linker moves your program up in memory by an appropriate amount to make room for the overlay handler and tables, if necessary.

11.7 USING LIBRARIES

You specify libraries in a command string in the same way you specify normal modules; you can include them anywhere in the command string, except in overlay lines. If a global symbol is undefined at the time the linker encounters the library in the input stream, and if a module is included in the library that contains that global definition, then the linker pulls that module from the library and links it into the load image. Only the modules needed to resolve references are pulled from the library; unreferenced modules are not linked.

NOTE

Modules in one library can call modules from another library; however, the libraries must appear in the command string in the order in which they are called. For example, assume module X in library ALIB calls Y from the BLIB library. To correctly resolve all globals, the order of ALIB and BLIB should appear in the command line as:

```
*Z=B,ALIB,BLIB
```

Module B is the root. It calls X from ALIB and brings X into the root. X in turn calls Y which is brought from BLIB into the root.

The linker selectively relocates and links object modules from specific user libraries that were built by the librarian. Figure 11-5 diagrams this general process. During pass-1 the linker processes the input files in the order in which they appear in the input command line. If the linker encounters a library file during pass-1, it makes note of the library in an internal save status block, and then proceeds to the next file. The linker processes only non-library files during the initial phase of pass-1. In the final phase of pass-1 the linker processes only library files. This is when it resolves the undefined globals that were referenced by the non-library files.

The linker processes library files in the order in which they appear in the input command line. The default system library (SYSLIB.OBJ) is always last. The processing steps are as follows:

1. If there are any undefined globals, the linker proceeds to step 2. Otherwise, it skips to step 5.
2. The linker reads as much of the library directory as the input buffer can hold.
3. The linker then searches the entire list of undefined globals for a match with the library directory. It places any globals that match in an internal library module list. If more of the library directory remains to be read, the linker proceeds to step 2.
4. The linker now processes the modules from the library that are associated with the matching undefined globals. If this processing results in new undefined globals that can be resolved by the current library, the linker goes back to step 2.
5. The linker closes the current library and processes the next library file, starting with step 1.

This search method allows modules to appear in any order in the library. You can specify any number of libraries in a link and they can be positioned anywhere, with the exception of forward references between libraries. The default system library, SYSLIB.OBJ, is the last library file the linker searches to resolve any remaining undefined globals.

Some languages, such as FORTRAN, have an Object Time System (OTS) that the linker takes from a library and includes in the final module. The most efficient way to accomplish this is to include these OTS routines (such as NHD, OTSCOM, and V2NS for FORTRAN) in SYSLIB.OBJ.

Libraries are input to the linker in the same way as other input files. Here is a sample LINK command string:

```
*TASK01,LP:=MAIN,MEASUR
```

This causes program MAIN.OBJ to be read from DK: as the first input file. Any undefined symbols generated by program MAIN.OBJ should be satisfied by the library file MEASUR.OBJ specified in the second input file. The linker tries to satisfy any remaining undefined globals from the default library, SYSLIB.OBJ. The load module, TASK01.SAV, is stored on DK: and a load map prints on the line printer.

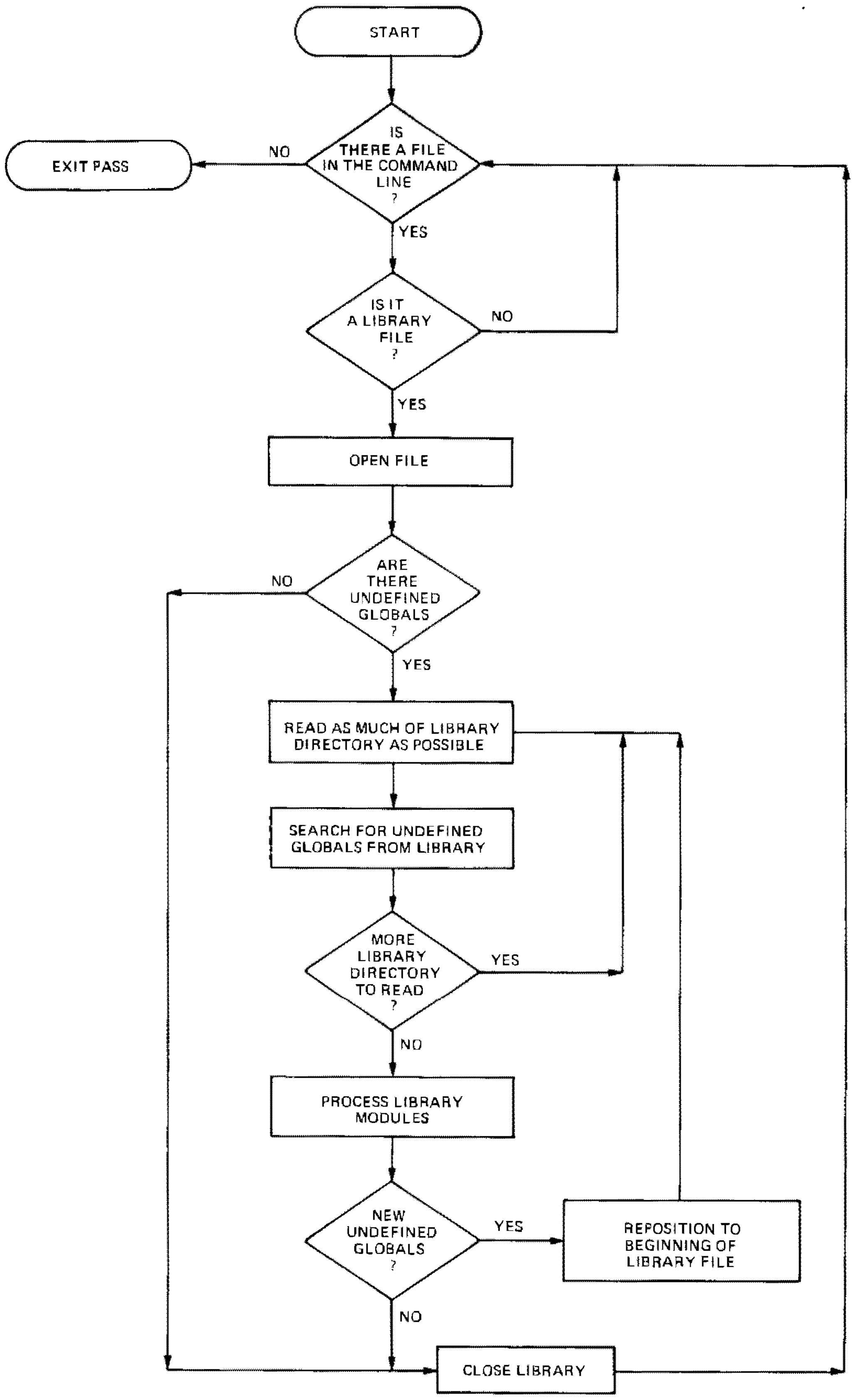

START
IS THERE A FILE IN THE COMMAND LINE ?
NO
EXIT PASS
YES
IS IT A LIBRARY FILE ?
NO
YES
OPEN FILE
ARE THERE UNDEFINED GLOBALS ?
NO
YES
READ AS MUCH OF LIBRARY DIRECTORY AS POSSIBLE
SEARCH FOR UNDEFINED GLOBALS FROM LIBRARY
MORE LIBRARY DIRECTORY TO READ ?
YES
NO
PROCESS LIBRARY MODULES
NEW UNDEFINED GLOBALS ?
YES
REPOSITION TO BEGINNING OF LIBRARY FILE
NO
CLOSE LIBRARY

Figure 11-5 Library Searches

11.8 OPTION DESCRIPTIONS

The options summarized in Table 11-2 are described in detail below.

11.8.1 Bottom Address Option (/B:n)

The /B:n option supplies the lowest address to be used by the relocatable code in the load module. The argument, n, is a 6-digit unsigned octal number that defines the bottom address of the program being linked. If you do not supply a value for n, the linker prints:

```
?LINK-F-/B No value
```

Retype the command, supplying an even octal value.

When you do not specify /B, the linker positions the load module so that the lowest address is location 1000 (octal). If the ASECT size is greater than 1000, the size of ASECT is used.

If you supply more than one /B option during the creation of a load module, the linker uses the first /B option specification. /B is illegal when you are linking to a high address (/H). /B is also illegal with foreground links. These modules are always linked to a bottom address of 1000 (octal).

NOTE

The bottom value must be an unsigned even octal number. If the value is odd, the ?LINK-F-/B odd value error message prints. Reenter the command string specifying an unsigned even octal number as the argument to the /B option.

The following command causes the relocatable code from the input file to be linked starting at location 500 (octal).

```
*OUTPUT,LP:=INPUT/B:500
```

11.8.2 Continue Option (/C) or (//)

The continue option (/C) lets you type additional lines of command string input. Use the /C option at the end of the current line and repeat it on subsequent command lines as often as necessary to specify all the input modules in your program. Do not enter a /C option on the last line of input.

The following command indicates that input is to be continued on the next line; the linker prints an asterisk.

```
*OUTPUT,LP:=INPUT/C
*
```

An alternate way to enter additional lines of input is to use the // option on the first line. The linker continues to accept lines of input until it encounters another // option, which can be either on a line with input file specifications, or on a line by itself. The advantage of using the // option instead of the /C option is that you do not have to type the // option on each continuation line. This example shows how the linker itself is linked:

```
*LINK,LINK=LINK0/B:700/W//
*LNKOV1/O:1
*LNKGSD/O:1
*LNKHDR/O:1
*LNKMAP/O:1
*LNKSAV/O:1
*LNKEM/O:1
*//
```

You cannot use the /C option and the // option together in a link command sequence. That is, if you use // on the first line, you must use // to terminate input on the last line. If you use /C on the first line, use /C on all lines but the last.

11.8.3 Extend Program Section Option (/E:n)

The /E:n option allows you to extend a program section to a specific value. Type the /E:n option at the end of the first command line. After you have typed all input command lines, the linker prompts with:

```
Extend section?
```

Respond with the name of the program section to be extended. The resultant program section size is equal to or greater than the value you specify depending upon the space the object code actually requires. Note that you can extend only one section.

The following example extends section CODE to 100 (octal) blocks.

```
*X,TT:=LK001/E:100
Extend section? CODE
```

11.8.4 Default FORTRAN Library Option (/F)

By indicating the /F option in the command line, you can link the FORTRAN library (FORLIB.OBJ on the system device SY:) with the other object modules you specify. You do not need to specify FORLIB explicitly. For example:

```
*FILE,LP:=AB/F
```

The object module AB.OBJ from DK: and the FORTRAN library SY:FORLIB.OBJ are linked together to form a load module called FILE.SAV.

The linker automatically searches a default system library, SY:SYSLIB.OBJ. The library normally includes the modules that compose FORLIB. The /F option is provided only for compatibility with other versions of RT-11. You should not have to use /F.

11.8.5 Highest Address Option (/H:n)

The /H:n option allows you to specify the top (highest) address to be used by the relocatable code in the load module. The argument, n, represents an unsigned even octal number. If you do not specify n, the linker prints:

```
?LINK-F-/H no value
```

Retype the command, supplying an even octal number to be used as the value.

If you specify an odd value, the linker responds with:

```
?LINK-F-/H odd value
```

Retype the command, supplying an even octal number.

If the value is not large enough to accommodate the relocatable code, the linker prints:

```
?LINK-F-/H value too low
```

Relink the program with a larger value.

The /H option cannot be used with the /R or /Y or /B options.

NOTE

Be careful when you use the /H option. Most RT-11 programs use the free memory above the relocatable code as a dynamic working area for I/O buffers, device handlers, symbol tables, etc. The size of this area differs on different memory configurations. Programs linked to a specific high address might not run in a system with less physical memory because there is less free memory.

11.8.6 Include Option (/I)

The /I option lets you take global symbols from any library and include them in the linking process even when they are not needed to resolve globals. This provides a method for forcing modules that are not called by other modules to be loaded from the library. When you specify the /I option, the linker prints:

```
    Library search?
```

Reply with the list of global symbols to be included in the load module: type a carriage return to enter each symbol in the list. A carriage return alone terminates the list of symbols.

The following example includes the global $SHORT in the load module:

```
    *SCCA=RK1:SCCA/I
    Library search?  $SHORT
    Library search?
```

11.8.7 Memory Size Option (/K:n)

The /K:n option lets you insert a value into word 56 of block 0 of the image file. The argument, n, represents the number of 1K blocks of memory required by the program; n is an integer in the range 1-28. You cannot use the /K option with the /R option. The /K:n option is provided mainly for compatibility with the RSTS operating system. You should not need to use it with RT-11.

11.8.8 LDA Format Option (/L)

The /L option produces an output file in LDA format instead of memory image format. The LDA format file can be output to any device including those that are not block-replaceable, such as paper tape or cassette. It is useful for files that are to be loaded with the Absolute Loader. The default file type .LDA is assigned when you use the /L option. You cannot use the /L option with the overlay option (/O) or the foreground link option (/R). The following example links files IN and IN2 on device DK: and outputs an LDA format file OUT.LDA to the cassette and a load map to the line printer.

```
    *CT:OUT,LP:=IN,IN2/L
```

11.8.9 Modify Stack Address Option (/M[:n])

The stack address, location 42, is the address that contains the initial value for the stack pointer. The /M option lets you specify the stack address. The argument, n, is an even, unsigned 6-digit octal number that defines the stack address. After all input lines have been typed, the linker prints the following message if you have not specified a value for n:

```
    Stack symbol?
```

In this case, specify the global symbol whose value is the stack address. You cannot specify a number. If you specify a nonexistent symbol, an error message prints and the stack address is set to the system default (1000 for SAV files) or to the bottom address if you used /B. If the program's absolute section extends beyond location 1000, the default stack space starts after the largest .ASECT contribution.

Direct assignment (with .ASECT) of the stack address within the program takes precedence over assignment with the /M option. The statements to do this in a MACRO program are as follows:

```
.ASECT
.=42
.WORD   INITSP   ;INITIAL STACK SYMBOL VALUE
.PSECT           ;RETURN TO PREVIOUS SECTION
```

The following example modifies the stack address.

```
*OUTPUT=INPUT/M
Stack symbol? BEG
```

11.8.10 Overlay Option (/O:n)

The /O option segments the load module so that the entire program is not memory resident at one time. This lets you execute programs that are larger than the available memory. The argument, n, is an unsigned octal number (up to six digits in length) specifying the overlay region to which the module is assigned. The /O option must follow (on the same line) the specification of the object modules to which it applies, and only one overlay region can be specified on a command line. Overlay regions cannot be specified on the first command line; that is reserved for the root segment. You must use /C or // for continuation.

You specify co-resident overlay routines (a group of subroutines that occupy the overlay region and segment at the same time) as follows:

```
*OBJA,OBJB,OBJC/O:1/C
*OBJD,OBJE/O:2/C
  .
  .
  .
```

All modules that the linker encounters until the next /O option will be co-resident overlay routines. If you specify, at a later time, the /O option with the same value you used previously (same overlay region), then the linker opens up the corresponding overlay area for a new group of subroutines. The new group of subroutines occupy the same locations in memory as the first group, but not at the same time. For example, if subroutines in object modules R and S are to be in memory together, but are never needed at the same time as T, then the following commands to the linker make R and S occupy the same memory as T (but at different times):

```
*MAIN,LP:=ROOT/C
*R,S/O:1/C
*T/O:1
```

The example shown above can also be written as follows:

```
*MAIN,LP:=ROOT/C
*R/O:1/C
*S/C
*T/O:1
```

The following example establishes two overlay regions.

```
*OUTPUT,LP:=INPUT//
*OBJA/O:1
*OBJB/O:1
*OBJC/O:2
*OBJD/O:2
*//
```

You must specify overlays in ascending order by region number. For example:

```
*A=A/C
*B/O:1/C
*C/O:1/C
*D/O:1/C
*E,F/O:2/C
*G/O:2
```

The following overlay specification is illegal since the overlay regions are not given in ascending numerical order (an error message prints in each case):

```
*X=LIBR0//
*LIBR1/O:1
*LIBR2/O:0
?LINK-W-/O ignored
*//
```

In the above example, the overlay option immediately preceding the error message is ignored.

11.8.11 Library List Size Option (/P:n)

The /P:n option lets you change the amount of space allocated for the library routine list. Normally, the default value allows enough space for your needs. It reserves space for approximately 256 unique library routines, which is the equivalent of specifying /P:256. (decimal) or /P:400 (octal).

The error message ?LINK-F-Library list overflow, increase size with /P indicates that you need to allocate more space for the library routine list. You must relink the program that makes use of the library routines. Use the /P:n option and supply a value for n that is greater than 256.

You can use the /P:n option to correct for symbol table overflow. Specify a value for n that is less than 256. This reduces the space used for the library routine list and increases the space allocated for the symbol table. If the value you choose is too small, the ?LINK-F-Library list overflow, increase size with /P message prints. In the following command, the amount of space for the library routine list is increased to 300 (decimal).

```
*SCCA=RK1:SCCA/P:300.
```

11.8.12 REL Format Option (/R[:n])

The /R[:n] option produces an output file in REL format for use as a foregound job with the FB or XM monitor. You cannot use .REL files with the SJ monitor. The /R option assigns the default file type .REL to the output file. The optional argument, n, represents the amount of stack space to allocate for the foreground job. The default value is 128. (decimal) bytes of stack space. If you also use the /M option, the value or global symbol associated with it overrides the /R value.

The following command links files FILEI.OBJ and NEXT.OBJ and stores the output on DT2: as FILEO.REL. It also prints a load map on the line printer.

```
*DT2:FILEO,LP:=FILEI,NEXT/R:200
```

You cannot use the /B, /H, and /L options with /R since a foreground REL job has a temporary bottom address of 1000 and is always relocated by FRUN. An error message prints if you attempt this. The /K option is also illegal with /R.

11.8.13 Symbol Table Option (/S)

The /S option instructs the linker to allow the largest possible memory area for its symbol table at the expense of input and output buffer space, which makes the linking process slower. You should use the /S option only if an attempt to link a program failed because of symbol table overflow. Often, use of /S allows the program to link.

11.8.14 Transfer Address Option (/T[:n])

The transfer address is the address at which a program starts when you initiate execution with an R, RUN, or FRUN command. It prints on the last line of the load map. The /T option lets you specify the start address of the load module. The argument, n, is a six-digit unsigned octal number that defines the transfer address. If you do not specify n, the following message prints:

```
Transfer symbol?
```

In this case, specify the global symbol whose value is the transfer address of the load module. Terminate your response with a carriage return. You cannot specify a number in answer to this message. If you specify a nonexistent symbol, an error message prints and the transfer address is set to 1 so that the program traps immediately if you attempt to execute it. If the transfer address you specify is odd, the program does not start after loading and control returns to the monitor.

Direct assignment (.ASECT) of the transfer address within the program takes precedence over assignment with the /T option. The transfer address assigned with a /T has precedence over that assigned with an .END assembly directive. To assign the transfer address within a MACRO program, use statements similar to these:

```
                .ASECT
                .=40
                .WORD       START1    ;SYMBOL VALUE FOR TRANSFER ADDRESS
                .PSECT                ;RETURN TO PREVIOUS SECTION
START1:          .
                 .
                 .
                or
START2:          .                    ;SECONDARY STARTING ADDRESS
                 .
                 .
                .END        START2
```

The following example links the files LIBRO.OBJ and ODT.OBJ together and starts execution at ODT's transfer address.

```
*LBRODT,LBRODT=LIBRO,ODT/T/W//
*LIBR1/O:1
*LIBR2/O:1
*LIBR3/O:1
*LIBR4/O:1
*LIBR5/O:1
*LBREM/O:1//
Transfer symbol? O.ODT
*
```

11.8.15 Round Up Option (/U:n)

The /U:n option rounds up the section you specify so that the size of the root segment is a whole number multiple of the value you supply. The argument, n, must be a power of 2. When you specify the /U:n option, the linker prompts:

```
Round section?
```

Reply with the name of the program section to be rounded. The program section must be in the root segment. Note that you can round only one program section. The following example rounds up section CHAR.

```
*LK007,TT:=LK007/U:200
Round section?  CHAR
```

If the program section you specify cannot be found, the linker prints ?LINK-W-Round section not found. The linking process continues with no rounding.

11.8.16 Map Width Option (/W)

The /W option directs the linker to produce a wide load map listing. If you do not specify the /W option, the listing is wide enough for three GLOBAL VALUE columns (normal for paper with 80 columns). If you use the /W command, the listing is six columns wide, which is ideal for a 132 column page.

11.8.17 Bit Map Inhibit Option (/X)

The /X option instructs the linker not to output the bit map if code is below 400. This option is provided only for compatibility with the RSTS operating system. The bit map is stored in locations 360-377 in block 0 of the load module. The linker normally stores the program memory usage bits in these eight words. Each bit represents one 256-word block of memory. This information is used by the R, RUN and GET commands when loading the program; therefore, use care when you use this option.

11.8.18 Boundary Option (/Y:n)

The /Y:n option starts a specific program section on a particular address boundary. The linker generates a whole number multiple of n, the value you specify, for the starting address of the program section. The argument, n, must be a power of 2. The linker extends the size of the previous program section to accommodate the new starting address. When you have entered all the input lines, the linker prompts:

```
Boundary section?
```

Respond with the name of the program section whose starting address you are modifying. Terminate your response with a carriage return. Note that you can specify only one program section for this option. If the program section you specify cannot be found, the linker prints ?LINK-W-Boundary section not found. The linking process continues.

The RT-11 monitors have internal 2-block overlays. The first overlay segment, OVLY0, must start on a disk block boundary:

```
*RKMNSJ.SYS=RKBTSJ,RT11SJ,RKTBSJ,RK/Y:1000
Boundary Section?  OVLY0
```

11.8.19 Zero Option (/Z:n)

The /Z:n option fills unused locations in the load module and places a specific value in these locations. The argument, n, represents the value to be placed in the unused locations. This option can be useful in eliminating random results that occur when the program references uninitialized memory by mistake. The system automatically zeroes unused locations. Use the /Z:n option only when you want to store a value other than zero in unused locations.

11.9 LINKER PROMPTS

Some of the linker operations prompt for more information, such as the names of specific global symbols or sections. The linker issues the prompt after you have entered all the input specifications, but before the actual linking begins. Table 11-6 shows the sequence in which the prompts occur.

Table 11-6 Linker Prompting Sequence

Prompt	Option
Transfer symbol?	/T
Stack symbol?	/M
Extend section?	/E:n
Boundary section?	/Y:n
Round section?	/U:n
Library search?	/I

The library search prompt is last because it can accept more than one symbol and is terminated by a carriage return on a line by itself.

Note that if the command lines are in an indirect file and the linker encounters an end-of-file before the prompting information has been supplied, it prints the prompt messages on the terminal.

The following example shows how the linker prompts for information when you combine options.

```
*LK001=LK001/T/M/E:100/Y:400/U:20/I
Transfer symbol? O.ODT
Stack symbol? ST3
Extend section? CHAR
Boundary section? CODE
Round section? STKSP
Library search? $SHORT
Library search?
*
```

CHAPTER 12
LIBRARIAN (LIBR)

The librarian utility program (LIBR) lets you create, update, modify, list, and maintain object library files. It also lets you create macro library files to use with the V03 MACRO-11 assembler.

A library file is a direct access file (a file that has a directory) that contains one or more modules of the same module type. The librarian organizes the library files so that the linker and MACRO-11 assembler can access them rapidly. Each library contains a library header, library directory (or global symbol or macro name table), and one or more object modules or macro definitions. The object modules in a library file can be routines that are repeatedly used in a program, routines that are used by more than one program, or routines that are related and simply gathered together for convenience. The contents of the library file are determined by your needs. An example of a typical object library file is the default system library that the linker uses, SYSLIB.OBJ. An example of a macro library file is SYSMAC.SML, which MACRO uses automatically to process programmed requests.

You access object modules in a library file from another program by making calls or references to their global symbols; you then link the object modules with the program that uses them, producing a single load module (see Chapter 11).

Consult the *RT-11 Software Support Manual* for more information on the internal data structure of a library file. However, that information is not necessary for your understanding of this chapter.

12.1 CALLING AND USING LIBR

To call the RT-11 librarian from the system device, respond to the dot (.) printed by the keyboard monitor by typing:

```
R LIBR(RET)
```

The Command String Interpreter prints an asterisk at the left margin on the console terminal when it is ready to accept a command line. Chapter 6, Command String Interpreter, describes the general syntax of the command line LIBR accepts.

Type two CTRL/Cs to halt the librarian at any time (or a single CTRL/C to halt the librarian when it is waiting for console terminal input) and return control to the monitor. To restart the librarian, type R LIBR or REENTER in response to the monitor's dot.

Section 12.2 explains how to use the librarian to create and maintain object libraries; Section 12.3 describes how to create macro libraries.

Specify the LIBR command string in the following general format:

```
library-filespec[n],list-filespec[n]=input-filespecs/options
```

where

library-filespec[n]	represents the library file to be created or updated. The optional argument, n, represents the number of blocks to allocate for the output file.

list-filespec [n]	represents a listing file for the library's contents. The optional argument, n, represents the number of blocks to allocate for the listing file.
input-filespec	represents the input object modules (you can specify up to six input files); it can also represent a library file to be updated.
option	represents an option from Table 12-1.

You specify devices and file names in the standard RT-11 command string syntax, with default file types assigned as follows:

File	File Type
list file:	.LST
library file:	.OBJ
input files:	.OBJ

If you do not specify a device, the default device (DK:) is assumed.

Each input file consists of one or more object modules and is stored on a given device under a specific file name and file type. Once you insert an object module into a library file you no longer reference the module by the name of the file of which it was a part; instead, you reference it by its individual module name. You assign this module name with the assembler with either a .TITLE statement in the assembly source program, or with the default name .MAIN. upon absence of a .TITLE statement or the subprogram name for FORTRAN routines. Thus, for example, the input file FORT.OBJ can exist on DT2: and can contain an object module called ABC. Once you insert the module into a library file, reference only ABC (not FORT.OBJ).

The input files normally do not contain main programs but rather subprograms, functions, and subroutines. The library file must never contain a FORTRAN "BLOCK DATA" subprogram; there is no undefined global symbol to cause the linker to load it automatically.

12.2 OPTION COMMANDS AND FUNCTIONS FOR OBJECT LIBRARIES

You maintain object library files by using option commands. Functions that you can perform include object module deletion, insertion and replacement, library file creation, and listing of an object library file's contents.

Table 12-1 summarizes the options available for you to use with RT-11 LIBR for object libraries. The following sections, which are arranged alphabetically by option, describe the options in greater detail.

Table 12-1 LIBR Object Options

Option	Command Line	Section	Meaning
/C	any but last	12.2.1	Command continuation; allows you to type the input specification on more than one line.
/D	first	12.2.4	Delete; deletes modules that you specify from a library file.
/E	first	12.2.5	Extract; extracts a module from a library and stores it in an .OBJ file.
/G	first	12.2.6	Global deletion; deletes global symbols that you specify from the library directory.
/N	first	12.2.7	Names; includes the module names in the directory.

(Continued on next page)

Table 12-1 (Cont.) LIBR Object Options

Option	Command Line	Section	Meaning
/P	first	12.2.8	P-section names; includes the program section names in the directory.
/R	first	12.2.9	Replace; replaces modules in a library file. This option must follow the file specification to which it applies.
/U	first	12.2.10	Update; inserts and replaces modules in a library file. This option must follow the file specification to which it applies.
/W	first	12.2.11	Indicates wide format for the listing file.
//	first and last	12.2.1	Command continuation; allows you to type the input specification on more than one line.

There is no option to indicate module insertion. If you do not specify an option, the librarian automatically inserts modules into the library file.

12.2.1 Command Continuation Options (/C and //)

You must use a continuation option whenever there is not enough room to enter a command string on one line. The maximum number of input files that you can enter on one line is six; you can use the /C option or the // option to enter more. Type the /C option at the end of the current line and repeat it at the end of subsequent command lines as often as necessary, so long as memory is available; if you exceed memory, an error message prints. Each continuation line after the first command line can contain only input file specifications (and no other options). Do not specify a /C option on the last line of input. If you use the // option, type it at the end of the first input line and again at the end of the last input line.

The following example creates a library file on the default device (DK:) under the file name ALIB.OBJ; it also creates a listing of the library file's contents as LIBLST.LST (also on the default device). The file names of the input modules are MAIN.OBJ, TEST.OBJ, FXN.OBJ, and TRACK.OBJ, all from DT1:.

```
*ALIB,LIBLST=DT1:MAIN,TEST,FXN/C
*DT1:TRACK
```

The next example creates a library file on the default device (DK:) under the name BLIB.OBJ. It does not produce a listing. Input files are MAIN.OBJ from the default device, TEST.OBJ from RK1:, FXN.OBJ from RK0:, and TRACK.OBJ from DT1:.

```
*BLIB=MAIN//
*RK1:TEST
*RK0:FXN
*DT1:TRACK//
```

Another way of writing this command line is:

```
*BLIB=MAIN,RK1:TEST,RK0:FXN//
*DT1:TRACK
*//
```

12.2.2 Creating a Library File

To create a library file, specify a file name on the output side of a command line.

The following example creates a new library called NEWLIB.OBJ on the default device (DK:). The modules that make up this library file are in the files FIRST.OBJ and SECOND.OBJ, both on the default device.

```
*NEWLIB=FIRST,SECOND
```

12.2.3 Inserting Modules into a Library

Whenever you specify an input file without specifying an associated option, the librarian inserts the modules in the file into the library file you name on the output side of the command string. You can specify any number of input files. If you include section names (if you use /P) in the global symbol table and if you attempt to insert a file that contains a global symbol or PSECT (or CSECT) having the same name as a global symbol or PSECT already existing in the library file, the librarian prints a warning message. The librarian does, however, update the library file, ignore the global symbol or section name in error, and return control to the CSI. You can then enter another command string.

Although you can insert object modules that exist under the same name (as assigned by the .TITLE statement), this practice is not recommended because of the difficulty and ambiguity involved when you need to update these modules (Sections 12.2.2.9 and 12.2.2.10 describe replacing and updating).

NOTE

The librarian performs module insertion, replacement, deletion, merge, and update concurrently with creating the library file. Therefore, you must indicate the library file to which the operation is directed on both the input and output sides of the command line, since effectively the librarian creates a "new" output library file each time it performs one of those operations. You must specify the library file first in the input field.

The following command line inserts the modules included in the files FA.OBJ, FB.OBJ, and FC.OBJ on DT1: into a library file named DXYNEW.OBJ on the default device. The resulting library also includes the contents of library DXY.OBJ.

```
*DXYNEW=DXY,DT1:FA,FB,FC
```

The next command line inserts the modules contained in files THIRD.OBJ and FOURTH.OBJ into the library NEWLIB.OBJ.

```
*NEWLIB,LIST=NEWLIB,THIRD,FOURTH
```

Note that the resulting library contains the original library plus some new modules. Note also that the resulting library replaces the original library because the same name was used in this example for the input and output library.

12.2.4 Delete Option (/D)

The /D option deletes modules and all their associated global symbols from the library.

When you use the /D option, the librarian prompts:

```
Module name?
```

Respond with the name of the module to be deleted followed by a carriage return; continue until you have entered all modules to be deleted. Type a carriage return immediately after the Module name? message to terminate input and initiate execution of the command line.

The following example deletes the modules SGN and TAN from the library file TRAP.OBJ on DT3:.

```
*DT3:TRAP=DT3:TRAP/D
Module name? SGN
Module name? TAN
Module name?
```

The next example deletes the module FIRST from the library LIBFIL.OBJ; all modules in the file ABC.OBJ replace old modules of the same name in the library; it also inserts the modules in the file DEF.OBJ into the library.

```
*LIBFIL=LIBFIL/D,ABC/R,DEF
Module name? FIRST
Module name?
```

In the following example, the librarian deletes two modules of the same name from the library file LIBFIL.OBJ.

```
*LIBFIL=LIBFIL/D
Module name? X
Module name? X
Module name?
```

12.2.5 Extract Option (/E)

The /E option allows you to extract an object module from a library file and place it in an .OBJ file.

When you specify the /E option, the librarian prints:

```
Global?
```

Respond with the name of the object module to be extracted. If you specify a global name, the librarian extracts the entire module of which that global is a part.

You cannot use the /E option on the same command line with any other option.

The following example extracts the ATAN routine from the FORTRAN library, SYSLIB.OBJ, and stores it in a file called ATAN.OBJ on DX1:.

```
*DX1:ATAN=SYSLIB/E
Global? ATAN
Global?
```

The next example extracts the $PRINT routine from SYSLIB.OBJ and stores it on DM1: as PRINT.OBJ.

```
*DM1:PRINT=SYSLIB/E
Global? $PRINT
Global?
```

The extract option is particularly useful if you need to use a routine in only one overlay segment. Normally, all modules that the linker acquires automatically from a library go into the root segment. To circumvent this, you can extract a routine with /E, then link it into an overlay segment instead of into the root segment.

12.2.6 Delete Global Option (/G)

The /G option lets you delete a specific global symbol from a library file's directory.

When you use the /G option, the librarian prints:

```
Global?
```

Respond with the name of the global symbol to be deleted followed by a carriage return; continue until you have entered all globals to be deleted. Type a carriage return immediately after the Global? message to terminate input and initiate execution of the command line.

The following command instructs the librarian to delete the global symbols NAMEA and NAMEB from the directory found in the library file ROLL.OBJ on DK:.

```
*ROLL=ROLL/G
Global? NAMEA
Global? NAMEB
Global?
```

The librarian deletes globals only from the directory (and not from the library itself). Whenever you update a library file all globals that you previously deleted are restored unless you use the /G option again to delete them. This feature lets you recover if you inadvertently delete the wrong global.

12.2.7 Include Module Names Option (/N)

The librarian does not include module names in the directory unless you use the /N option on the first line of the command. The linker loads modules from libraries based on undefined globals, not on module names. The linker also provides equivalent functions by using global symbols and not module names. Normally, then, it is a waste of space and a performance compromise to include module names in the directory.

If you do not include module names in the directory, the MODULE column of the directory listing is blank unless the module requires a continuation line to print all its globals. A plus (+) sign in the MODULE column indicates continued lines. The /N option is useful mainly when you create a temporary library in order to obtain a directory listing.

If the library does not have module names in its directory, you must create a new library to include the module names. The following example illustrates how to do this. It creates a temporary new library from the current library (by specifying the null device for output) and lists its directory on the terminal. The current library OLDLIB remains unchanged.

```
*NL:TEMP,TT:=OLDLIB/N
RT-11 LIBRARIAN V03.00   TUE 03-MAY-77 20:36:41
TEMP                     TUE 03-MAY-77 20:36:40

MODULE          GLOBALS         GLOBALS         GLOBALS

IRAD50          IRAD50          RAD50
JMUL            JMUL
LEN             LEN
SUBSTR          SUBSTR
JADD            JADD
JCMP            JCMP
```

12.2.8 Include P-section Names Option (/P)

The librarian does not include program section names in the directory unless you use the /P option on the first line of the command. The linker does not use section names to load routines from libraries; including the names can decrease linker performance. Including program section names also causes a conflict in the library directory and subsequent searches, since the librarian treats section names and global symbols identically.

This option is provided for compatibility with RT-11V2C. DIGITAL recommends that you avoid using it with RT-11V03.

12.2.9 Replace Option (/R)

Use the /R option to replace modules in a library file. The /R option replaces existing modules in the library file you specify as output with the modules of the same names contained in the file(s) you specify as input. In the command string, enter the input library file before the files used in the replacement operation.

If an old module does not exist under the same name as an input module, or if you specify the /R option on a library file, the librarian prints an error message preceded by the module name, and ignores the replace command. /R must follow each input file name containing modules for replacement.

The following command line indicates that the modules in the file INB.OBJ are to replace existing modules of the same names in the library file TFIL.OBJ. The object modules in the files INA.OBJ and INC.OBJ are to be added to TFIL. All files are to be stored on the default device DK:.

```
*TFIL=TFIL,INA,INB/R,INC
```

The same operation occurs in the next command as in the preceding example, except that this updated library file is assigned the new name XFIL.

```
*XFIL=TFIL,INA,INB/R,INC
```

12.2.10 Update Option (/U)

The /U option lets you update a library file by combining the insert and replace functions. If the object modules that compose an input file in the command line already exist in the library file, the librarian replaces the old modules in the library file with the new modules in the input file. If the object modules do not already exist in the library file, the librarian inserts those modules into the library. (Note that some of the error messages that might occur with separate insert and replace functions do not print when you use the update function.) /U must follow each input file that contains modules to be updated. Specify the input library file before the input files in the command line.

The following command line instructs the librarian to update the library file BALIB.OBJ on the default device. First the modules in FOLT.OBJ and BART.OBJ replace old modules of the same names in the library file, or if none already exist under the same names, the modules are inserted. The modules from the file TAL.OBJ are then inserted; an error message prints if the name of the module in TAL.OBJ already exists.

```
*BALIB=BALIB,FOLT/U,TAL,BART/U
```

In the next example, there are two object modules of the same name (X) in both Z and XLIB; these are first deleted from XLIB. This ensures that both the modules called X in file Z are correctly placed into the library. Globals SEC1 and SEC2 are also deleted from the directory but automatically return the next time the library XLIB.OBJ is updated.

```
*XLIB=XLIB/D,Z/U/G
Module name?  X
Module name?  X
Module name?
Global?  SEC1
Global?  SEC2
Global?
```

12.2.11 Wide Option (/W)

The /W option gives you a wider listing if you request a listing file. The wider listing has six GLOBAL columns instead of three, as in the normal listing. This is useful if you list the directory on a line printer or a terminal that has 132 columns.

12.2.12 Listing the Directory of a Library File

You can request a listing of the contents of a library file (the global symbol table) by indicating both the library file and a list file in the command line. Since a library file is not being created or updated, you do not need to indicate the file name on the output side of the command line; however, you must use a comma to designate a null output library file.

The command syntax is as follows:

```
*,LP:=library-filespec
```

or

```
*,list-filespec=library-filespec
```

where

library-filespec	represents the existing library file.
LP:	indicates that the listing is to be sent directly to the line printer (or terminal, if you use TT:).
list-filespec	represents a list file of the library file's contents.

The following command outputs to DT2: as LIST.LST, a listing of the contents of the library file LIBFIL.OBJ on the default device.

```
*,DT2:LIST=LIBFIL
```

The next command sends to the line printer a listing of all modules in the library file FLIB.OBJ, which is stored on the default device.

```
*,LP:=FLIB
```

Here is a sample section of a large directory listing:

```
*,TT:=SYSLIB
RT-11 LIBRARIAN V03.00   TUE 03-MAY-77 21:01:01
SYSLIB                   TUE 03-MAY-77 20:59:47

MODULE        GLOBALS        GLOBALS        GLOBALS

              DCO$           ECO$           FCO$
+             GCO$           RCI$
              DIC$IS         DIC$MS         DIC$FS
+             DIC$SS         $DIVC          $DVC
              ADD$IS         ADD$MS         ADD$FS
+             ADD$SS         SUD$IS         SUD$MS
+             SUD$FS         SUD$SS         $ADD
```

The first line of the listing file shows the version of the librarian that was used and the current date and time. The second line prints the library file name and the date and time the library was created. Module names are not included in this example. Each line in the rest of the listing shows only the globals that appear in a particular module. If a module contains more global symbol names than can print on one line, a new line will be started with a plus (+) sign in column 1 to indicate continuation.

12.2.13 Merging Library Files

You can merge two or more library files under one file name by specifying in a single command line all the library files to be merged. The librarian does not delete the individual library files following the merge unless the output file name is identical to one of the input file names.

The command syntax is as follows:

```
*library-filespec=input-filespecs
```

where

library-filespec	represents the library file that will contain all the merged files. (If a library file already exists under this name, you must also indicate it in the input side of the command line so that it is included in the merge).
input-filespec	represents a library file to be merged.

Thus, the following command combines library files MAIN.OBJ, TRIG.OBJ, STP.OBJ, and BAC.OBJ under the existing library file name MAIN.OBJ; all files are on the default device DK:. Note that this replaces the old contents of MAIN.OBJ.

```
*MAIN=MAIN,TRIG,STP,BAC
```

The next command creates a library file named FORT.OBJ and merges existing library files A.OBJ, B.OBJ, and C.OBJ under the file name FORT.OBJ.

```
*FORT=A,B,C
```

NOTE

Library files that you combine using PIP are illegal as input to both the librarian and the linker.

12.2.14 Combining Library Option Functions

You can request two or more library functions in the same command line, with the exception of the /E option, which cannot be specified on the same command line with any other option. The librarian performs functions (and issues appropriate prompts) in the following order:

1. /C or //
2. /D
3. /G
4. /U
5. /R
6. Insertions
7. Listing

Here is an example that combines options:

```
*FILE,LP:=FILE/D,MODX,MODY/R
Module name? XYZ
Module name? A
Module name?
```

The librarian performs the functions in this example in order, as follows:

1. Deletes modules XYZ and A from the library file FILE.OBJ.
2. Replaces any duplicate of the modules in the file MODY.OBJ.
3. Inserts the modules in the file MODX.OBJ.
4. Lists the directory of FILE.OBJ on the line printer.

12.3 OPTION COMMANDS AND FUNCTIONS FOR MACRO LIBRARIES

The librarian lets you create macro libraries. A macro library works with the V03 MACRO-11 assembler to reduce macro search time.

The .MACRO directive produces the entries in the library directory (macro names). LIBR does not maintain a directory listing file for macro libraries; you can print the ASCII input file to list the macros in the library.

The default input and output file type for macro files is .MAC.

Be careful not to give the library file the same name as one of the input files. This deletes the input file when the library is created.

Table 12-2 summarizes the options you can use with macro libraries. The options are explained in detail in the following two sections.

Table 12-2 LIBR Macro Options

Options	Command Line	Section	Meaning
/C	any but last	12.3.1	Command continuation; allows you to type the input specification on more than one line.
/M[:n]	first	12.3.2	Macro; creates a macro library from the ASCII input file containing .MACRO directives.
//	first and last	12.3.1	Command continuation; allows you to type the input specification on more than one line.

12.3.1 Command Continuation Options (/C or //)

These options are the same for macro libraries as for object libraries. See Section 12.2.1.

12.3.2 Macro Option (/M[:n])

The /M[:n] option creates a macro library file from an ASCII input file that contains .MACRO directives. The optional argument, n, determines the amount of space to allocate for the macro name directory. Remember that n is interpreted as an octal number; you must follow n by a decimal point (n.) to indicate a decimal number. Each 64 macros occupy one block of library directory space. The default value for n is 128, enough space for 128 macros, which will use 2 blocks for the macro name table.

The command syntax is as follows:

```
*library-filespec=input-filespec/M[:n]
```

where

library-filespec	represents the macro library to be created.

input-filespec	represents the ASCII input file that contains .MACRO definitions.
/M[:n]	is the macro option.

The continuation options (/C or //) are the only options you can use with the macro option.

The following example creates the macro library SYSMAC.SML from the ASCII input file SYSMAC.MAC. Both files are on device DK:.

```
*SYSMAC.SML=SYSMAC/M
```

CHAPTER 13
DUMP

DUMP is the RT-11 program that prints on the console or lineprinter, or writes to a file all or any part of a file in octal words, octal bytes, ASCII characters, and/or Radix-50 characters. DUMP is particularly useful for examining directories and files that contain binary data.

13.1 CALLING AND USING DUMP

To call the DUMP program from the system device, respond to the dot (.) printed by the keyboard monitor by typing:

```
R DUMP(RET)
```

The Command String Interpreter prints an asterisk at the left margin on the console terminal when it is ready to accept a command line. If you respond to the asterisk by typing only a carriage return, DUMP prints its current version number.

You can type CTRL/C to halt DUMP and return control to the monitor when DUMP is waiting for input from the console terminal. You must type two CTRL/Cs to abort DUMP at any other time. To restart DUMP, type R DUMP or REENTER and a carriage return in response to the monitor's dot. Chapter 6, Command String Interpreter, describes the general syntax of the command line that DUMP accepts. If you do not specify an output file, the listing prints on the line printer. If you do not specify a file type for an output file, the system uses .DMP.

13.2 DUMP OPTIONS

Table 13-1 summarizes the options that are valid for DUMP.

Table 13-1 DUMP Options

Option	Explanation
/B	Outputs octal bytes.
/E:n	Ends output at block number n, where n is an octal block number.
/G	Ignores input errors.
/N	Suppresses ASCII output.
/O:n	Outputs only block number n, where n is an octal block number. With the /O option, you can dump only one block for each command line.
/S:n	Starts output with block number n, where n is an octal block number. For random access devices, n cannot be greater than the number of blocks in the file.
/T	Defines a tape as non-RT-11 file-structured.
/W	Outputs octal words (the default mode).
/X	Outputs Radix-50 characters.

ASCII characters are always dumped unless you type /N.

If you specify an input file name, the block numbers (n) you supply are relative to the beginning of that file. If you do not specify a file name; that is, if you are dumping a device, the block numbers are the absolute (physical) block numbers on that device. Remember that the first block of any file or device is block 0.

DUMP handles operations that involve magtape and cassette differently from operations involving random access devices.

If you dump an RT-11 file-structured tape and specify only a device name in the input specification, DUMP reads only as far as the logical end-of-tape. Logical end-of-tape is indicated by an end-of-file label followed by two tape marks. For non-file-structured tape, logical end-of-tape is indicated by two consecutive tape marks. If you dump a cassette and specify only the device name in the input specification, the results are unpredictable. For magtape dumps, tape mark messages appear in the output listing as DUMP encounters them on the tape.

If you use /S:n with magtape, n can be any positive value. However, an error can occur if n is greater than the number of blocks written on the tape. For example, if a tape has 100 written blocks and n is 110, an error can occur if DUMP accesses past the 100th block. If you specify /E:n, DUMP reads the tape from its starting position (block 0, unless you specify otherwise) to block number n or to logical end-of-tape, whichever comes first.

13.3 EXAMPLES

This section includes sample DUMP commands and the listings they produce.

The following command string directs DUMP to print in octal words information contained in block 1 of the file DMPX.SAV stored on device DK:.

```
*DMPX.SAV/O:1

DMPX.SAV/O:1
BLOCK NUMBER 00001
000/ 012700 000030 000261 006101 106100 110024 012700 000206 *@...1.A.@...@...*
020/ 006301 001403 106100 103774 000766 000207 052140 023364 *A...@.\.V...@TT&*
040/ 022030 021424 015326 023747 000000 023747 006766 021401 *.S.#V.G'..G'V..#*
060/ 050500 062745 177400 177001 051042 040505 020104 042515 *@QEE...¯"READ ME*
100/ 021041 000000 000377 000001 000376 001000 000400 177777 *!"......¯.......*
120/ 000000 000000 000000 000000 000000 000000 000000 000000 *................*
140/ 000000 000000 000000 000000 000000 000000 000000 000000 *................*
160/ 000000 000000 000000 000000 000000 000000 000000 000000 *................*
200/ 000000 000000 000000 000000 000000 000000 000000 000000 *................*
220/ 000000 000000 000000 000000 000000 000000 000000 000000 *................*
240/ 000000 000000 000000 000000 000000 000000 000000 000000 *................*
260/ 000000 000000 000000 000000 000000 000000 000000 000000 *................*
300/ 000000 000000 000000 000000 000000 000000 000000 000000 *................*
320/ 000000 000000 000000 000000 000000 000000 000000 000000 *................*
340/ 000000 000000 000000 000000 000000 000000 000000 000000 *................*
360/ 000000 000000 000000 000000 000000 000000 000000 000000 *................*
400/ 000000 000000 000000 000000 000000 000000 000000 000000 *................*
420/ 000000 000000 000000 000000 000000 000000 000000 000000 *................*
440/ 000000 000000 000000 000000 000000 000000 000000 000000 *................*
460/ 000000 000000 000000 000000 000000 000000 000000 000000 *................*
500/ 000000 000000 000000 000000 000000 000000 000000 000000 *................*
520/ 000000 000000 000000 000000 000000 000000 000000 000000 *................*
540/ 000000 000000 000000 000000 000000 000000 000000 000000 *................*
560/ 000000 000000 000000 000000 000000 000000 000000 000000 *................*
600/ 000000 000000 000000 000000 000000 000000 000000 000000 *................*
620/ 000000 000000 000000 000000 000000 000000 000000 000000 *................*
640/ 000000 000000 000000 000000 000000 000000 000000 000000 *................*
660/ 000000 000000 000000 000000 000000 000000 000000 000000 *................*
700/ 000000 000000 000000 000000 000000 000000 000000 000000 *................*
720/ 000000 000000 000000 000000 000000 000000 000000 000000 *................*
740/ 000000 000000 000000 000000 000000 000000 000000 000000 *................*
760/ 000000 000000 000000 000000 000000 000000 000000 000000 *................*
```

In the printout above, the heading shows which block of the file follows. The numbers in the leftmost column indicate the byte offset from the beginning of the block. Remember that these are all octal values and that there are two bytes per word. The octal words that were dumped appear in the next eight columns. The rightmost column contains the ASCII equivalent of each octal word. DUMP substitutes a dot (.) for non-printing codes, such as those for control characters.

The next command dumps block 1 of file PIP.SAV. The /N option suppresses ASCII output.

```
*PIP.SAV/N/O:1

PIP.SAV/N/O:1
BLOCK NUMBER  00001
000/ 100101 000000 000000 000002 000001 100102 000000 000000
020/ 000001 000002 100103 000000 000000 000000 000000 000104
040/ 000000 177352 002000 000004 100107 000000 000000 000000
060/ 000000 100113 000000 000000 006002 000020 100115 000000
100/ 000000 002000 000040 100116 000000 000000 000500 000200
120/ 100117 000000 000000 000300 000400 100120 000000 000000
140/ 002000 001000 100121 000000 000000 000000 000000 000122
160/ 000000 177602 001164 002000 100123 000000 000000 000000
200/ 000000 100124 000000 000000 000000 000000 100125 000000
220/ 000000 000020 004000 100127 000000 000000 000000 000000
240/ 100130 000000 000000 000000 000000 100131 000000 000000
260/ 000000 000000 000000 100115 000000 000000 002600 000100
300/ 004000 000001 000000 000100 000000 000000 000000 000000
320/ 000000 000000 000000 000000 000000 000000 000000 000000
340/ 000000 000000 000000 000000 000000 000000 000000 000000
360/ 000000 000000 000000 000000 000000 000000 000000 000000
400/ 000000 000000 000000 000000 000000 000000 000000 000000
420/ 000000 000000 000000 000000 000000 000000 000000 000000
440/ 000000 000000 000000 000000 000000 000000 000000 000000
460/ 000000 000000 000000 000000 000000 000000 000000 000000
500/ 000000 000000 000000 000000 000000 000000 000000 000000
520/ 000000 000000 000000 000000 000000 000000 000000 000000
540/ 000000 000000 000000 000000 000000 000000 000000 000000
560/ 000000 000000 000000 000000 000000 000000 000000 000000
600/ 000000 000000 000000 000000 000000 000000 000000 000000
620/ 000000 000000 000000 000000 000000 000000 000000 000000
640/ 000000 000000 000000 000000 000000 000000 000000 000000
660/ 000000 000000 000000 000000 000000 000000 000000 000000
700/ 000000 000000 000000 000000 000000 000000 000000 000000
720/ 000000 000000 000000 000000 000000 000000 000000 000000
740/ 000000 000000 000000 000000 003054 002543 002510 002562
760/ 002314 002407 002426 002342 002446 002614 002676 002177
```

The following command dumps block 1 of SYSMAC.MAC in octal bytes. ASCII equivalents appear underneath each byte.

```
*SYSMAC.MAC/B/O:1

SYSMAC.MAC/B/O:1
BLOCK NUMBER  00001
000/ 040 124 117 040 124 110 105 123 105 040 114 111 103 105 116 123
         T   O       T   H   E   S   E       L   I   C   E   N   S
020/ 105 040 124 105 122 115 123 056 040 124 111 124 114 105 040 124
     E       T   E   R   M   S   .       T   I   T   L   E       T
040/ 117 040 101 116 104 040 117 127 116 105 122 123 110 111 120 040
     O       A   N   D       O   W   N   E   R   S   H   I   P
060/ 117 106 040 124 110 105 040 015 012 073 040 123 117 106 124 127
     O   F       T   H   E       .   .   ;       S   O   F   T   W
100/ 101 122 105 040 123 110 101 114 114 040 101 124 040 101 114 114
     A   R   E       S   H   A   L   L       A   T       A   L   L
```

```
120/ 040 124 111 115 105 123 040 122 105 115 101 111 116 040 111 116
         T   I   M   E   S       R   E   M   A   I   N       I   N
140/ 040 104 111 107 111 124 101 114 056 015 012 073 015 012 073 040
         D   I   G   I   T   A   L   .   .   .   ;   .   .   ;
160/ 124 110 105 040 111 116 106 117 122 115 101 124 111 117 116 040
     T   H   E       I   N   F   O   R   M   A   T   I   O   N
200/ 111 116 040 125 110 111 123 040 123 117 106 124 127 101 122 105
     I   N       T   H   I   S       S   O   F   T   W   A   R   E
220/ 040 111 123 040 123 125 102 112 105 103 124 040 124 117 015 012
         I   S       S   U   B   J   E   C   T       T   O   .   .
240/ 073 040 103 110 101 116 107 105 040 127 111 124 110 117 125 124
     ;       C   H   A   N   G   E       W   I   T   H   O   U   T
260/ 040 116 117 124 111 103 105 040 101 116 104 040 123 110 117 125
         N   O   T   I   C   E       A   N   D       S   H   O   U
300/ 114 104 040 116 117 124 040 102 105 040 103 117 116 123 124 122
     L   D       N   O   T       B   E       C   O   N   S   T   R
320/ 125 105 104 015 012 073 040 101 123 040 101 040 103 117 115 115
     U   E   D   .   .   ;       A   S       A       C   O   M   M
340/ 111 124 115 105 116 124 040 102 131 040 104 111 107 111 124 101
     I   T   M   E   N   T       B   Y       D   I   G   I   T   A
360/ 114 040 105 121 125 111 120 115 105 116 124 040 103 117 122 120
     L       E   Q   U   I   P   M   E   N   T       C   O   R   P
400/ 117 122 101 124 111 117 116 056 015 012 073 015 012 073 040 104
     O   R   A   T   I   O   N   .   .   .   ;   .   .   ;       D
420/ 111 107 111 124 101 114 040 101 123 123 125 115 105 123 040 116
     I   G   I   T   A   L       A   S   S   U   M   E   S       N
440/ 117 040 122 105 123 120 117 116 123 111 102 111 114 111 124 131
     O       R   E   S   P   O   N   S   I   B   I   L   I   T   Y
460/ 040 106 117 122 040 124 110 105 040 125 123 105 015 012 073 040
         F   O   R       T   H   E       U   S   E   .   .   ;
500/ 117 122 040 172 105 114 111 101 102 111 114 111 124 131 040 117
     O   R       R   E   L   I   A   B   I   L   I   T   Y       O
520/ 106 040 111 124 123 040 123 117 106 124 127 101 122 105 040 117
     F       I   T   S       S   O   F   T   W   A   R   E       O
540/ 116 040 105 121 125 111 120 115 105 116 124 015 012 073 040 127
     N       E   Q   U   I   P   M   E   N   T   .   .   ;       W
560/ 110 111 103 110 040 111 123 040 116 117 124 040 123 125 120 120
     H   I   C   H       I   S       N   O   T       S   U   P   P
600/ 114 111 105 104 040 102 131 040 104 111 107 111 124 101 114 056
     L   I   E   D       B   Y       D   I   G   I   T   A   L   .
620/ 015 012 073 015 012 073 040 105 106 054 112 104 054 114 120 054
     .   .   ;   .   .   ;       E   F   ,   J   D   ,   L   P   ,
640/ 102 103 054 104 126 054 103 122 054 110 112 015 012 014 056 115
     B   C   ,   D   V   ,   C   R   ,   H   J   .   .   .   .   M
660/ 101 103 122 117 040 056 056 126 061 056 056 015 012 056 115 103
     A   C   R   O       .   .   V   1   .   .   .   .   .   M   C
700/ 101 114 114 011 056 056 056 103 115 060 054 056 056 056 103 115
     A   L   L   .   .   .   .   C   M   0   ,   .   .   .   C   M
720/ 061 054 056 056 056 103 115 062 054 056 056 056 103 115 063 054
     1   ,   .   .   .   C   M   2   ,   .   .   .   C   M   3   ,
740/ 056 056 056 103 115 064 054 056 056 056 103 115 065 054 056 056
     .   .   .   C   M   4   ,   .   .   .   C   M   5   ,   .   .
760/ 056 103 115 066 015 012 056 056 056 126 061 075 061 056 015 012
     .   C   M   6   .   .   .   .   .   V   1   =   1   .   .   .
```

The last example shows block 6 (the directory) of device RK0:. The output is in octal words with Radix-50 equivalents below each word.

```
*RK0:/N/X/O:6

RK0:/N/X/O:6
BLOCK NUMBER  00006
000/ 000020 000004 000004 000000 000046 002000 071105 055202
        P      D      D               8     YX    RKM    NSJ
020/ 075273 000130 000015 010405 002000 071105 054162 075273
     SYS       BH      M   B.7      YX    RKM    NFB    SYS
040/ 000141 000015 010405 002000 071105 055515 075273 000150
       BQ       M   B.7      YX    RKM    NXM    SYS     BX
```

```
060/ 000015 010405 002000 015425 055202 075273 000132 000015
       M    B,7     YX    DMM    NSJ    SYS     BJ      M
100/ 010405 002000 015425 054162 075273 000143 000015 010405
     B,7     YX    DMM    NFB    SYS     BS      M    B,7
120/ 002000 015425 055515 075273 000152 000015 010405 002000
      YX    DMM    NXM    SYS     BZ      M    B,7     YX
140/ 016315 055202 075273 000130 000015 010405 002000 016315
     DXM    NSJ    SYS     BH      M    B,7     YX    DXM
160/ 054162 075273 000141 000015 010405 002000 016315 055515
     NFB    SYS     BQ      M    B,7     YX    DXM    NXM
200/ 075273 000141 000015 010405 002000 016055 055202 075273
     SYS     BQ      M    B,7     YX    DTM    NSJ    SYS
220/ 000130 000015 010405 002000 016055 054162 075273 000141
      BH      M    B,7     YX    DTM    NFB    SYS     BQ
240/ 000015 010405 002000 016055 055515 075273 000150 000015
       M    B,7     YX    DTM    NXM    SYS     BX      M
260/ 010405 002000 016005 055202 075273 000130 000015 010405
     B,7     YX    DSM    NSJ    SYS     BH      M    B,7
300/ 002000 016005 054162 075273 000141 000015 010405 002000
      YX    DSM    NFB    SYS     BQ      M    B,7     YX
320/ 016005 055515 075273 000150 000015 010405 002000 015615
     DSM    NXM    SYS     BX      M    B,7     YX    DPM
340/ 055202 075273 000130 000015 010405 002000 015615 054162
     NSJ    SYS     BH      M    B,7     YX    DPM    NFB
360/ 075273 000141 000015 010405 002000 015615 055515 075273
     SYS     BQ      M    B,7     YX    DPM    NXM    SYS
400/ 000150 000015 010405 002000 070575 055202 075273 000130
      BX      M    B,7     YX    RFM    NSJ    SYS     BH
420/ 000015 010405 002000 070575 054162 075273 000141 000015
       M    B,7     YX    RFM    NFB    SYS     BQ      M
440/ 010405 002000 070575 055515 075273 000150 000015 010405
     B,7     YX    RFM    NXM    SYS     BX      M    B,7
460/ 002000 071105 056573 075273 000123 000015 010405 002000
      YX    RKM    NBK    SYS     BC      M    B,7     YX
500/ 016315 056573 075273 000123 000015 010405 002000 016040
     DXM    NBK    SYS     BC      M    B,7     YX    DT
520/ 000000 075273 000002 000015 010405 002000 015600 000000
            SYS      B      M    B,7     YX    DP
540/ 075273 000002 000015 010405 002000 016300 000000 075273
     SYS      B      M    B,7     YX    DX            SYS
560/ 000003 000015 010405 002000 070560 000000 075273 000002
       C      M    B,7     YX    RF            SYS      B
600/ 000015 010405 002000 071070 000000 075273 000002 000015
       M    B,7     YX    RK            SYS      B      M
620/ 010405 002000 015410 000000 075273 000004 000015 010405
     B,7     YX    DM            SYS      D      M    B,7
640/ 002000 015770 000000 075273 000002 000015 010405 002000
      YX    DS            SYS      B      M    B,7     YX
660/ 100040 000000 075273 000002 000015 010405 002000 046600
     TT            SYS      B      M    B,7     YX    LP
700/ 000000 075273 000002 000015 010405 002000 012620 000000
            SYS      B      M    B,7     YX    CR
720/ 075273 000003 000015 010405 002000 052140 000000 075273
     SYS      C      M    B,7     YX    MT            SYS
740/ 000010 000015 010405 002000 051510 000000 075273 000011
       H      M    B,7     YX    MM            SYS      I
760/ 000015 010405 002000 054540 000000 075273 000002 000015
       M    B,7     YX    NL            SYS      B      M
```

CHAPTER 14 FILEX

The file exchange program (FILEX) is a general file transfer program that converts files from one format to another so that you can use them with various operating systems. You can initiate transfers between any block-replaceable RT-11 directory-structured device and any device listed in Table 14-1.

Table 14-1 Legal FILEX Devices

Device	Valid as Input	Valid as Output
PDP-11 DOS/BATCH DECtape	X	X
DOS/BATCH Disk	X	
RSTS DECtape	X	X
DECsystem-10 DECtape	X	
Interchange Diskette	X	X

FILEX does not support magtape or cassette in any operation.

Section 4.2 of this manual describes how to use wildcards. You can use wildcards in the FILEX command string. However, you can not use embedded wildcards in any file name or file type. For example, the following line represents a valid file specification.

```
**.MAC
```

The next line is an illegal file specification for FILEX.

```
*T%ST.MAC
```

14.1 FILE FORMATS

FILEX can transfer files created by four different operating systems: RT-11, DECsystem-10, universal interchange format (IBM) and DOS/BATCH (PDP-11 Disk Operating System). You can use the following three data formats in a transfer: ASCII, image, and packed image. ASCII files conform to the American Standard Code for Information Interchange in which each character is represented by a 7-bit code. In ASCII mode, FILEX deletes null and rubout characters, as well as parity bits.

Because the file structure and data formats for each system vary, options are needed in the command line to indicate the file structures and the data formats involved in the transfer. These options are discussed in Section 14.3. FILEX assumes that all devices are RT-11 structured. You can use options from Table 14-2 to indicate otherwise.

14.2 CALLING AND USING FILEX

To call FILEX from the system device, respond to the dot (.) printed by the keyboard monitor by typing:

```
R FILEX (RET)
```

The Command String Interpreter prints an asterisk at the left margin of the terminal and waits for you to enter a command. If you enter only a carriage return at this point, the current version number of FILEX prints on the terminal.

Type two CTRL/Cs to halt FILEX at any time (or a single CTRL/C to halt FILEX when it is waiting for console terminal input) and return control to the monitor. To restart FILEX, type R FILEX or REENTER in response to the monitor's dot.

14.3 FILEX OPTIONS

Table 14-2 lists the options that initiate various FILEX operations. The table is divided into three sections: transfer options, operation options and file structure options. Transfer options direct FILEX to copy data in a certain mode. The three transfer modes are: ASCII, image, and packed image. Operation options perform another function in addition to the data transfer. These additional functions include deleting files, producing directory listings and zeroing device directories. File structure options indicate the formats of devices that are involved in a transfer. These formats are DOS/BATCH or RSTS, DECsystem-10, and interchange. FILEX accepts one transfer option and one operation option in a single command. You can specify one device option for each file involved in the transfer. The device options (/S, /T, and /U) must appear following the device and file name to which they apply; other options can appear anywhere in the command line. These options are explained in more detail in the following sections.

14.3.1 Transferring Files Between RT-11 and DOS/BATCH (or RSTS)

You can transfer files between block-replaceable devices used by RT-11 and the PDP-11 DOS/BATCH system. Input from DOS/BATCH can be either disk or DECtape. You can use both linked and contiguous files.

If the input device is a DOS/BATCH disk, you should specify a DOS/BATCH user identification code (UIC). The UIC is of the form [nnn,nnn], where nnn represents an octal integer less than or equal to 377. The first part of the code represents a user-group number; the second is the individual user number. The initial default value is [1,1]. The UIC you supply will be the default for all future transfers. If you do not specify a UIC, FILEX will use the current default UIC. Note that the square brackets [ ] are part of the UIC; you must type them if you specify a UIC.

Output to DOS/BATCH is limited to DECtape only. You do not need a UIC in a command line when you are accessing only DECtape. Individual users do not "own" files on DECtape under DOS. However, no error occurs if you do use a UIC. DECtape used under the RSTS system is legal as both input and output, since its format is identical to DOS/BATCH DECtape. You can use any valid RT-11 file storage device for either input or output in the transfer. The RT-11 device DK: is assumed if you do not indicate a device.

An RT-11 DECtape can hold more information than a DOS/BATCH or RSTS DECtape. Be careful when you copy files from a full RT-11 tape to a DOS DECtape. Some information might not transfer. In this case, an error message prints and the transfer does not complete.

When a transfer from an RT-11 device to a DOS DECtape occurs, the block size of the file can increase. However, if the file is later transferred back to an RT-11 device, the block size does not decrease.

Table 14-2 FILEX Options

Transfer Options	Explanation
/A	Indicates a character-by-character ASCII transfer in which FILEX deletes rubouts and nulls. If you use /U with /A, FILEX ignores all sector boundaries on the diskette. If you use /T with /A, FILEX assumes that each PDP-10 36-bit word contains five 7-bit ASCII bytes. If you use /U with /A, FILEX assumes that records are to be terminated by a line feed, vertical tab, or form feed. The transfer terminates when a CTRL/Z is encountered. (This feature is included for compatibility with RSTS.) FILEX does not transfer the CTRL/Z.
/I	Performs an image mode transfer. If the input is DOS/BATCH, RSTS, interchange diskette, or RT-11, the transfer is word-for-word. If the input is from DECsystem-10, /I indicates that the file resembles a file created on DECsystem-10 by MACY11, MACX11, or LNKX11 with the /I option. In this case, each PDP-10 36-bit word will contain one PDP-11 8-bit byte in its low-order bits. If input or output is an interchange diskette, FILEX reads and writes four diskette sectors for each RT-11 block.
/P	Performs a packed image mode transfer. If the input is DOS/BATCH, RSTS, or RT-11, the transfer is word-for-word. If the input is from DECsystem-10, /P indicates that the file resembles a file created on DECsystem-10 by MACY11, MACX11, or LNKX11 with the /P option. In this case, each PDP-10 36-bit word will contain four PDP-11 8-bit bytes aligned on bits 0, 8, 18, and 26. This is the default mode. If the input is interchange diskette, the data is assumed to be EBCDIC. If the output is interchange diskette, FILEX writes the data as EBCDIC.
Operation Options	**Explanation**
/D	Deletes the file you specify from the device directory. This option is valid only for DOS/BATCH, RSTS DECtape, and interchange diskette.
/F	Produces a brief listing of the device directory on the terminal. It lists only file names and file types.
/L	Produces a complete listing of the device directory on the console terminal, including file names, block lengths, and creation dates.
/Y	Suppresses the dev:/ZERO ARE YOU SURE? message.
/Z	Zeroes the directory of the device you specify in the proper format. This option is valid only for DOS/BATCH, RSTS DECtape, and interchange diskette.
File Structure Options	**Explanation**
/S	Indicates that the device is a legal DOS/BATCH or RSTS block-replaceable device.
/T	Indicates that the device is a legal DECsystem-10 DECtape.
/U[:n.]	Indicates that the device is an interchange diskette; n. represents the length of each output record, in characters; n. is a decimal integer in the range 1-128. The default value is 80; n. is not valid with an input file specification, or with /A or /I.

To transfer a file from a legal DOS/BATCH block-replaceable device or RSTS DECtape to a legal RT-11 device, use this command syntax:

```
*output-filespec=input-filespec/S[/option]
```

where

output-filespec	represents any valid RT-11 device, file name, and file type (if the device is not file structured, you can omit the file name and file type).
input-filespec	represents the DOS/BATCH or RSTS device, UIC, file name and file type to be transferred. See Table 14-1 for a list of valid devices.
/S	is the option from Table 14-2 that designates a DOS/BATCH or RSTS block-replaceable device. This option must be included in the command line.
/option	is one of the three transfer options from Table 14-2.

To transfer files from an RT-11 storage device to a DOS/BATCH or RSTS DECtape, use this command syntax:

```
*DTn:output-filename/S[/option]=input-filespec
```

where

DTn:output-filename	represents the file name and file type of the file to be created, as well as the DOS/BATCH or RSTS DECtape on which to store the file.
input-filespec	represents the device, file name, and file type of the RT-11 file to be transferred.
/S	is the option from Table 14-2 that designates a DOS/BATCH or RSTS DECtape. This option must be included in the command line.
/option	is one of the three transfer options from Table 14-2.

The following examples illustrate the use of the /S option.

The following command instructs FILEX to transfer a file called SORT.ABC from the RT-11 default device DK: to a DOS/BATCH or RSTS format DECtape on unit DT2. The transfer is done in image mode.

```
*DT2:SORT.ABC/S=SORT.ABC/I
```

The next command allows a file to be transferred from DOS/BATCH (or RSTS) DECtape to the papertape punch under RT-11. The transfer is done in ASCII mode.

```
*PC:=DT2:FIL.TYP/S/A
```

The next command causes the file MACR1.MAC from the DOS/BATCH disk on unit 1, which is stored under the UIC [1,2], to be transferred to the RT-11 device DK:. [1,2] becomes the default UIC for any further DOS/BATCH operations.

```
*DK:*.*=RK1:[1,2]MACR1.MAC/S
```

14.3.2 Transferring Files Between RT-11 and Interchange Diskette

You can transfer files between block-replaceable devices used by RT-11 and interchange format (proposed ANSI format) diskettes. Files are transferred in one of the following three formats: ASCII, image, and packed image (EBCDIC) mode.

A universal diskette consists of 77 tracks (some of which are reserved), each containing 26 sectors numbered from 1 to 26. A sector contains one record of 128 or fewer characters. A record must begin on a sector boundary on an interchange diskette in packed image mode. There must be only one record per sector. If a record does not fill a sector, the remainder is filled with blanks. Since packed image (EBCDIC) mode is inefficient and wastes space, it is only recommended to read or write diskettes that must be compatible with IBM 3741 format.

Image mode provides an exact copy of a file. Nulls, rubouts, and parity are preserved in a transfer. ASCII and image mode perform similar functions; however, for most operations, you should probably use ASCII. Use image mode to transfer data when the parity bit or nulls are significant (i.e., when you are not transferring ASCII data).

Packed image (EBCDIC) mode is generally compatible with IBM 3741 format. (FILEX does not support error mapping of bad sectors and multi-volume files.) Packed image (EBCDIC) is the default mode, so you must use one of the options from Table 14-2 to specify ASCII or image mode. All records of a file must be the same size. You indicate this with the /U:n. option.

NOTE

File types are not normally recognized in interchange format; instead, a single 8-character file name is used. However, in order to provide uniformity throughout RT-11, FILEX has been designed to accept a 6-character file name with a 2-character file type. If you transfer a file from RT-11 to interchange diskette, any 3-character file type is truncated to two characters.

To transfer files from RT-11 format to interchange format, use this command syntax:

```
*output-filespec/U[:n.] [/option]=input-filespec
```

where

output-filespec	represents the device, file name, and file type of the interchange file to be created.
/U[:n.]	is the option from Table 14-2 that designates an interchange diskette. This option must be included in the command line; n. represents the length of each output record, in characters; $1 \leqslant n \leqslant 128$ (default is 80).
/option	is one of the three transfer options from Table 14-2.
input-filespec	represents the device, file name, and file type of the RT-11 file to be transferred.

To transfer files from interchange diskette to RT-11 format, use this command syntax:

```
*output-filespec=input-filespec/U[/option]
```

where

output-filespec	represents the device, file name, and file type of the RT-11 file to be created.
input-filespec	represents the device, file name, and file type of the interchange file to be transferred.
/U	is the option from Table 14-2 that designates an interchange diskette. This option must be included in the command line.
/option	is one of the three transfer options from Table 14-2.

The following command transfers the file IVAN.CAT from RT-11 RK05 unit 2 to the diskette on unit 1. The transfer is done in exact image mode (indicated by /I), ignoring all sector boundaries.

```
*DX1:IVAN.CAT/U/I=RK2:IVAN.CAT
```

The next command instructs FILEX to transfer the file BENMAR.FRM from the RT-11 disk unit 2 to the diskette on unit 0, and rename it KENJOS.JO. The /U option indicates that the format is to be changed from ASCII to the interchange format. There will be one record per sector of 128 or fewer characters. If there are fewer than 128 characters, the remainder of the sector will be filled with spaces.

```
*DX0:KENJOS.JO/U=RK2:BENMAR.FRM
```

The next command transfers the file TYPE.SET from RT-11 diskette unit 0 to the interchange diskette on unit 2. The exchange converts ASCII to interchange format putting a maximum of 7 (indicated by :7.) characters into each sector until the entire record has been transferred. Records in excess of seven characters will be broken up and placed in succeeding sectors on the diskette. New records always begin on a sector boundary; carriage returns and line feeds are discarded. However, if you use /A or /I, FILEX ignores boundary limits and preserves carriage returns and line feeds.

```
*DX2:TYPE.SE/U:7.=RX0:TYPE.SET
```

File TYPE.SET before transfer:

ABCDEFGHIJKLMN

File TYPE.SET after transfer:

ABCDEFG – (spaces up to 128 characters) Sector 1
HIJKLMN – (spaces up to 128 characters) Sector 2

The next command copies file IVAN.CA from the interchange diskette on unit 1 to the RT-11 line printer, treating the input as ASCII characters. Note that once a record has been divided into sectors, it cannot be transferred back to its original large size.

```
*LP:=DX1:IVAN.CA/U/A
```

14.3.3 Transferring Files to RT-11 from DECsystem-10

Files can not be transferred to RT-11 devices from a DECsystem-10 DECtape when a foreground job is running. This restriction is due to the fact that when FILEX reads DECsystem-10 files, it accesses the DECtape control registers directly instead of using the RT-11 DECtape control handler. Output can be to any valid RT-11 device. DECsystem-10 DECtape is the only valid input device. To transfer files from DECsystem-10 format to RT-11 format, use this command syntax:

```
*output-filespec=input-filespec/T[/option]
```

where

output-filespec	represents any valid RT-11 device, file name, and file type (if the device is not file-structured, you can omit the file name and file type).
input-filespec	represents the DECtape unit, file name, and file type of the DECsystem-10 file to be transferred.
/T	is the option from Table 14-2 that signifies a DECsystem-10 DECtape. When you use /T, and especially when you also use /A, the system clock loses time. Examine the time and reset it if necessary with the TIME command.
/option	is one of the three transfer options from Table 14-2.

You can not convert RT-11 files to DECsystem-10 format directly. However, there is a two-step procedure for doing this. First, run RT-11 FILEX and convert the files to DOS formatted DECtape. Then run DECsystem-10 FILEX to read the DOS DECtape.

The following command converts the ASCII file STAND.LIS from DECsystem-10 ASCII format to RT-11 ASCII format and stores it under RT-11 on DECtape 2 as STAND.LIS.

```
*DT2:STAND.LIS=DT1:STAND.LIS/T/A
```

Transfers from DECsystem-10 DECtape to RT-11 DECtape can cause an <UNUSED> block to appear after the file on the RT-11 device. This is a result of the method by which RT-11 handles the increased amount of information on a DECsystem-10 DECtape.

The next command indicates that all files on DECsystem-10 DECtape 0 with the file type .LIS are to be transferred to the RT-11 system device using the same file name and a file type of .NEW. The /P option is the assumed transfer mode.

```
*SY:*.NEW=DT0:*.LIS/T
```

14.3.4 Listing Directories

You can list a directory of any of the block-replaceable devices used in a FILEX transfer. The directory listing prints on the console terminal. The command syntax is:

```
*device:/L/option
```

where

device — represents the block-replaceable device. These are the valid device types:

DOS/BATCH, RSTS	DTn:	or	any disk
DECsystem-10	DTn:		
interchange diskette	DXn:		

/L — is the listing option from Table 14-2. You can substitute /F if you want a brief listing of file names only.

/option	is /S, /T, or /U[:n.]. These are the valid format and option combinations:

DOS/BATCH, RSTS	/S
DECsystem-10	/T
interchange diskette	/U

The following example shows the complete disk directory for UIC[1,7] of the device RK1:. The letter C following the file size on a DOS/BATCH or RSTS directory listing indicates that the file is a contiguous file.

```
*RK1:/L/S[1,7]
BADB      .SYS     1      22-JUL-74
MONLIB    .CIL   175C     22-JUL-74
DU11      .PAL    45      24-JUL-74
VERIFY    .LDA    67C     22-JUL-74
CILUS     .LDA    39      22-JUL-74
```

The next command lists all files with the file type .PAL that are stored on DECtape unit 1.

```
*DT1:*.PAL/L/S
```

The next command produces a brief directory listing of the interchange diskette on unit 0, giving file names only.

```
*DX0:/U/F
```

The following command lists all files on DECsystem-10 formatted DECtape unit 1, regardless of file name or file type; a brief directory is requested (/F) in which only file names print.

```
*DT1:*.*/F/T
```

14.3.5 Deleting Files From DOS/BATCH (RSTS) DECtapes and Interchange Diskettes

Use FILEX to delete files from DOS/BATCH and RSTS formatted DECtapes, and from interchange diskettes.

To delete files, use this command syntax:

```
*filespec/D/option
```

where

filespec	represents the device, file name and file type of the file to be deleted.
/D	is the delete option from Table 14-2.
/option	can be either /S, for DOS/BATCH and RSTS block-replaceable devices, or /U, for interchange diskettes.

The following command deletes all files with the file type .PAL on DECtape unit 0.

```
*DT0:*.PAL/D/S
```

The next command deletes the file TABLE.OBJ from the DECtape on unit 2.

```
*DT2:TABLE.OBJ/D/S
```

The next command deletes all files with an .RN file type from the interchange diskette on unit 0.

```
*DX0:*.RN/D/U
```

You can also use FILEX to initialize the directories of DOS/BATCH and RSTS DECtapes, and interchange diskettes. Use this command syntax:

```
*device:/Z/option [/Y]
```

where

device	represents the DOS/BATCH or RSTS DECtape, or the interchange diskette to be zeroed.
/Z	is the zero option from Table 14-2.
/option	can be either /S, for DOS/BATCH and RSTS DECtapes, or /U, for interchange diskettes.
/Y	inhibits the FILEX verification message.

The following command directs FILEX to initialize the directory of the interchange diskette on unit 0.

```
*DX0:/Z/U
DX0:/Zero are you sure?
```

Respond with a Y for initialization to begin. Any other response aborts the command.

The next command initializes the DECtape on unit 1 in DOS/BATCH (RSTS) format. Note that by using the /Y option you suppress the verification message.

```
*DT1:/Z/S/Y
```

NOTE

An initialized universal diskette's directory has a single file entry, DATA, that reserves the entire diskette. You must delete this file before you can write any new files on this diskette. This arrangement is necessary for IBM compatibility. Do this by using the following command:

```
*DX0:DATA/D/U
```

CHAPTER 15
SOURCE COMPARE (SRCCOM)

The RT-11 source compare program (SRCCOM) compares two ASCII files and lists the differences between them. SRCCOM can either print the results or store them in a file. SRCCOM is particularly useful when you need to compare two similar versions of a source program. A file comparison listing highlights the changes made to a program during an editing session.

15.1 CALLING AND USING SRCCOM

To call SRCCOM from the system device, respond to the dot (.) printed by the keyboard monitor by typing:

```
R SRCCOM (RET)
```

The Command String Interpreter prints an asterisk at the left margin of the terminal and waits for you to enter a command string. If you respond to the asterisk by entering only a carriage return, SRCCOM prints its current version number. The syntax of the command is:

```
[output-filespec=] input-filespec1,input-filespec2 [/option . . . ]
```

where

output-filespec	represents the destination device or file for the listing of differences.
input-filespec1	represents the first file to be compared.
input-filespec2	represents the second file to be compared.
option	is one of the options from Table 15-1.

The console terminal is the default output device. The default file type for input files is .MAC. SRCCOM assigns .DIF as the default file type for output files.

You can type CTRL/C to halt SRCCOM and return control to the monitor when SRCCOM is waiting for input from the console terminal. You must type two CTRL/Cs to abort SRCCOM at any other time. To restart SRCCOM, type R SRCCOM or REENTER and a carriage return in response to the monitor's dot.

SRCCOM examines the two source files line by line, looking for groups of lines that match. When SRCCOM finds a mismatch, it lists the lines from each file that are different. SRCCOM continues to list the differences until a specific number of lines from the first file match the second file. The specific number of lines that constitutes a match is a variable that you can set with the /L:n option.

15.2 SRCCOM OPTIONS

Table 15-1 summarizes the operations you can perform with SRCCOM. You can place these options anywhere in the command string, but it is conventional to place them at the end of the command line.

Table 15-1 SRCCOM Options

Option	Explanation
/B	Compares blank lines; normally, SRCCOM ignores blank lines.
/C	Ignores comments (all text on a line preceded by a semicolon) and spacing (spaces and tabs). A line consisting entirely of a comment is still included in the line count.
/F	Includes form feeds in the output listing; SRCCOM normally compares form feeds, but does not include them in the output listing.
/H	Types on the console terminal a list of options available; this is the "help" text.
/L:n	Specifies the number of lines that determines a match; n is an octal integer in the range 1-310. The default value for n is 3.
/S	Ignores spaces and tabs.

15.3 SRCCOM OUTPUT FORMAT

This section describes the SRCCOM output listing format and explains how to interpret it.

15.3.1 Sample Text

It will be helpful first to look at a sample text file, DEMO.BAK:

```
        FILE1
HERE'S A BOTTLE AND AN HONEST FRIEND!
  WHAT WAD YE WISH FOR MAIR, MAN?
WHA KENS, BEFORE HIS LIFE MAY END,
  WHAT HIS SHAME MAY BE O' CARE, MAN?
THEN CATCH THE MOMENTS AS THEY FLY,
  AND USE THEM AS YE OUGHT, MAN: --
BELIEVE ME, HAPPINESS IS SLY,
  AND COMES NOT AY WHEN SOUGHT, MAN.

                                   --SCOTTISH SONG
```

This file contains two typing errors. In the fourth line of the song, "shame" should be "share". In the seventh line, "sly" should be "shy". Here is a file called DEMO.TXT that has the correct text:

```
        FILE2
HERE'S A BOTTLE AND AN HONEST FRIEND!
  WHAT WAD YE WISH FOR MAIR, MAN?
WHA KENS, BEFORE HIS LIFE MAY END,
  WHAT HIS SHARE MAY BE O' CARE, MAN?
THEN CATCH THE MOMENTS AS THEY FLY,
  AND USE THEM AS YE OUGHT, MAN:--
BELIEVE ME, HAPPINESS IS SHY,
  AND COMES NOT AY WHEN SOUGHT, MAN.

                                   --SCOTTISH SONG
```

15.3.2 Sample Output Listing

SRCCOM lists the differences between the two files. The example below compares the original file, DEMO.BAK, to its edited version, DEMO.TXT:

```
*DEMO.BAK,DEMO.TXT/L:1
1)1                FILE1
2)1                FILE2

1)1           WHAT HIS SHAME MAY BE O' CARE, MAN?
1)          THEN CATCH THE MOMENTS AS THEY FLY,
****
2)1           WHAT HIS SHARE MAY BE O' CARE, MAN?
2)          THEN CATCH THE MOMENTS AS THEY FLY,
**********
1)1         BELIEVE ME, HAPPINESS IS SLY,
1)            AND COMES NOT AY WHEN SOUGHT, MAN.
****
2)1         BELIEVE ME, HAPPINESS IS SHY,
2)            AND COMES NOT AY WHEN SOUGHT, MAN.
**********

%FILES ARE DIFFERENT
```

SRCCOM always prints the first line of each file as identification:

```
1)1                FILE1
2)1                FILE2
```

The numbers at the left margin have the form n)m, where n represents the source file (either 1 or 2) and m represents the page of that file on which the specific line is located.

SRCCOM next prints a blank line and then lists the differences between the two files. The /L:n option was used in this example to set to 1 the number of lines that must agree to constitute a match.

The first three lines of the song are the same in both files, so they do not appear in the listing. The fourth line contains the first discrepancy. SRCCOM prints the fourth line from the first file, followed by the next matching line as a reference.

```
1)1           WHAT HIS SHAME MAY BE O' CARE, MAN?
1)          THEN CATCH THE MOMENTS AS THEY FLY,
****
```

The four asterisks terminate the differences section from the first file.

SRCCOM then prints the fourth line from the second file, again followed by the next matching line as a reference:

```
2)1           WHAT HIS SHARE MAY BE O' CARE, MAN?
2)          THEN CATCH THE MOMENTS AS THEY FLY,
**********
```

The ten asterisks terminate the listing for a particular difference section.

SRCCOM scans the remaining lines in the files in the same manner. When it reaches the end of each file, it prints the %FILES ARE DIFFERENT message on the terminal.

The second example is slightly different. The default value for the /L:n option sets to 3 the number of lines that must agree to constitute a match. The output listing is directed to the file DIFF.TXT on device DK:.

```
*DIFF.TXT=DEMO.BAK,DEMO.TXT

%FILES ARE DIFFERENT
```

The monitor TYPE command lists the information contained in the output file:

```
.TYPE DIFF.TXT

1)1                FILE1
2)1                FILE2

1)1          WHAT HIS SHAME MAY BE O' CARE, MAN?
1)         THEN CATCH THE MOMENTS AS THEY FLY,
1)           AND USE THEM AS YE OUGHT, MAN:--
1)         BELIEVE ME, HAPPINESS IS SLY,
1)           AND COMES NOT AY WHEN SOUGHT, MAN.
1)
1)                         --SCOTTISH SONG
1)
****
2)1          WHAT HIS SHARE MAY BE O' CARE, MAN?
2)         THEN CATCH THE MOMENTS AS THEY FLY,
2)           AND USE THEM AS YE OUGHT, MAN:--
2)         BELIEVE ME, HAPPINESS IS SHY,
2)           AND COMES NOT AY WHEN SOUGHT, MAN.
2)
2)                         --SCOTTISH SONG
2)
**********
```

As in the first example, SRCCOM prints the first line of each file:

```
1)1                FILE1
2)1                FILE2
```

The first three lines of each file are identical and, therefore, constitute a match. Again, the fourth lines differ. SRCCOM prints the fourth line of the first file, followed by the next matching line:

```
1)1          WHAT HIS SHAME MAY BE O' CARE, MAN?
1)         THEN CATCH THE MOMENTS AS THEY FLY,
```

However, SRCCOM did not find a match (three identical lines) before it encountered the next difference. So, the second matching line prints, followed by the next differing line from the first file:

```
1)           AND USE THEM AS YE OUGHT, MAN:--
1)         BELIEVE ME, HAPPINESS IS SLY,
```

Again, the next matching line prints:

```
1)           AND COMES NOT AY WHEN SOUGHT, MAN.
```

The /B option to include blank lines in the comparison was not used in this example. Thus, SRCCOM recognizes only one more line before the end of file. Since the two identical lines do not constitute a match (three are needed) SRCCOM prints the last line as part of the difference section for the first file:

```
1)
1)                                  --SCOTTISH SONG
1)
****
```

In a similar manner, SRCCOM prints a differences section for the second file, ending the listing with the %FILES ARE DIFFERENT message.

NOTE

Regardless of the output specification, the differences message always prints on the terminal. If you compare two files that are identical and specify a file for the output listing, the message NO DIFFERENCES ENCOUNTERED prints on the terminal and SRCCOM does not create an output file.

PART V
ALTERING ASSEMBLED PROGRAMS

This part of the manual consists of the following three chapters: ODT, PATCH, and PAT. The three programs that these chapters describe can help you debug programs and make changes to programs that are already assembled.

Chapter 16 describes the on-line debugging technique (ODT). This program aids you in debugging assembly language programs. With ODT, you can control your program's execution, examine locations in memory and alter their contents, and search the object program for specific words.

Chapter 17 describes the PATCH utility program. PATCH can make code modifications to any RT-11 file. You use PATCH to examine and then change words or bytes in a file. PATCH's checksum feature is particularly useful when you are making a correction or improvement to an existing executable program; it verifies that the changes you make are correct.

Chapter 18 describes the object module patching utility (PAT). This program allows you to patch, or update, code in a relocatable binary object module. PAT accepts a file containing corrections or additional instructions and applies these corrections and additions to the original object module.

CHAPTER 16
ON-LINE DEBUGGING TECHNIQUE (ODT)

RT-11 on-line debugging technique (ODT) is a program (supplied with the system) that aids in debugging assembly language programs. From your terminal, you direct the execution of your program with ODT. ODT performs the following tasks:

- Prints the contents of any location for examination or alteration
- Runs all or any portion of an object program using the breakpoint feature
- Searches the object program for specific bit patterns
- Searches the object program for words that reference a specific word
- Calculates offsets for relative addresses
- Fills a single word, block of words, byte or block of bytes with a designated value.

Make sure you have an assembly listing and a link map available for the program you want to debug with ODT. You can make minor corrections to the program on line during the debugging session, and you can then execute the program under the control of ODT to verify the corrections. If you need to make major changes, such as adding a missing subroutine, note them on the assembly listing and incorporate them in a new assembly.

16.1 CALLING AND USING ODT

ODT is supplied as a relocatable object module. You can link ODT with your program (using the RT-11 linker) for an absolute area in memory and load it with your program. When you link ODT with your program, it is a good idea to link ODT low in memory relative to the program. If you link ODT high in memory, you must be sure that the buffer space for your program is contained within program bounds. Otherwise, if your program uses dynamic buffering, program execution can destroy ODT in memory. Figure 16-1 shows possible relationships between ODT and the program MYPROG in memory.

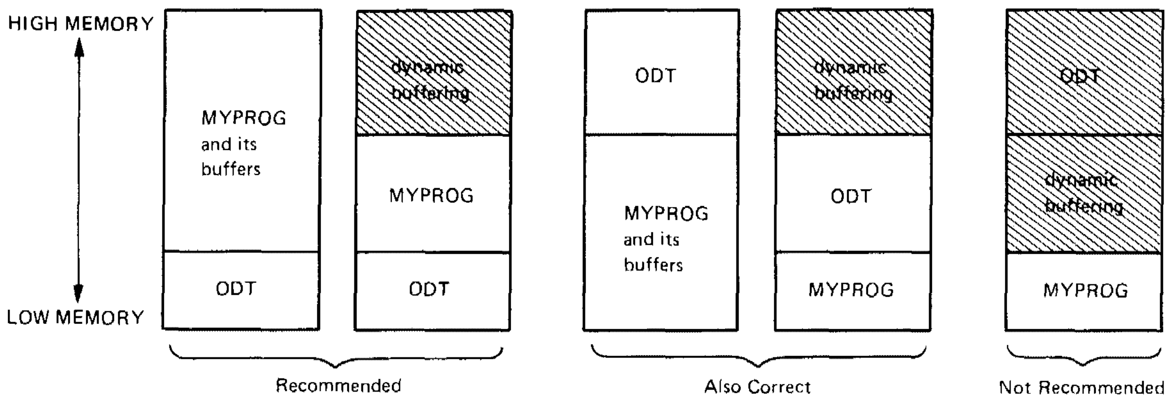


Figure 16-1 Linking ODT with a Program

Once loaded in memory with your program, ODT has three legal start or restart addresses. Use the lowest (O.ODT) for normal entry, retaining the current breakpoi :s. The next (O.ODT+2) is a restart address that clears all breakpoints and reinitializes ODT, thus saving the general registers and clearing the relocation registers. Use the last address (O.ODT+4) to reenter ODT. A reenter saves the processor status and general registers, and removes the breakpoint instructions from your program. ODT prints the bad entry (BE) error message. Breakpoints that were set are reset by the next ;G command. (;P is illegal after a BE message.) The ;G and ;P commands run a program; they are explained in Section 16.3.7.

The system uses as an absolute address the address of the entry point O.ODT shown in the linker load map.

NOTE

If you link ODT with an overlay-structured file, it should reside in the root segment so that it will always be in memory. Remove all breakpoints from the current overlay segment before execution proceeds to another overlay segment. A breakpoint inserted in an overlay is destroyed if it is overlaid during program execution.

The following examples show how to link and load ODT and how to restart ODT.

1. This example links ODT low in memory relative to MYPROG, creating the executable module MYPROG.SAV. Running MYPROG causes ODT to start automatically.

```
.LINK/MAP:TT:/DEBUG  MYPROG
RT-11 LINK  V03.01         Load Map          Mon 09-May-77 18:50:42
MYPROG.SAV        Title:   ODT       Ident:  X01.01

Section   Addr    Size     Global   Value    Global   Value    Global   Value

. ABS.   000000  001000    (RW,I,GBL,ABS,OVR)
         001000  016130    (RW,I,LCL,REL,CON)
                           O.ODT    001222

Transfer address = 001222, High limit = 017130 =  3884. words

.R MYPROG

 ODT  V01.04
*
```

2. This example links MYPROG low in memory relative to ODT and specifies O.ODT as the transfer address. Running MYPROG causes ODT to start automatically. The advantage to this method is that MYPROG is loaded at its normal execution-time address.

```
.LINK/MAP:TT: MYPROG,ODT/TRANSFER
Transfer address? O.ODT
RT-11 LINK  V03.01         Load Map          Mon 09-May-77 18:53:16
MYPROG.SAV        Title:   DEMOSP  Ident:    X01.01

Section   Addr    Size     Global   Value    Global   Value    Global   Value

. ABS.   000000  001000    (RW,I,GBL,ABS,OVR)
         001000  016130    (RW,I,LCL,REL,CON)
                           O.ODT    011260

Transfer address = 011260, High limit = 017130 =  3884. words
```

```
.R MYPROG

 ODT   V01.04
*
```

3. This example is similar to Example 2 above, except that execution does not automatically begin with ODT. When you start the program (MYPROG in this case) you must specify the address of O.ODT as shown in the link map.

```
.LINK/MAP:TT: MYPROG,ODT
RT-11 LINK   V03.01         Load Map          Mon 09-May-77 18:55:03
MYPROG.SAV         Title:   DEMOSP  Ident:    X01.01

Section   Addr    Size     Global   Value    Global   Value    Global   Value

. ABS.   000000  001000    (RW,I,GBL,ABS,OVR)
         001000  016130    (RW,I,LCL,REL,CON)
                           O.ODT    011260

Transfer address = 001036, High limit = 017130 =  3884. words

.GET MYPROG

.START 11260

 ODT   V01.04
*
```

4. This example links ODT with a bottom address of 4000, then loads ODT.SAV and MYPROG.SAV into memory. As in Example 3 above, when you start the program, you must specify the address of O.ODT as shown in the link map.

```
.LINK/MAP:TT: ODT/BOTTOM:4000
RT-11 LINK   V03.01         Load Map          Mon 09-May-77 18:59:42
ODT    .SAV        Title:   ODT     Ident:              /B:004000

Section   Addr    Size     Global   Value    Global   Value    Global   Value

. ABS.   000000  004000    (RW,I,GBL,ABS,OVR)
         004000  006072    (RW,I,LCL,REL,CON)
                           O.ODT    004222

Transfer address = 004222, High limit = 012072 =  2589. words

.GET ODT.SAV

.GET MYPROG.SAV

.START 4222

 ODT   V01.04
*
```

5. You can restart ODT by specifying O.ODT+2 as the start address. This reinitializes ODT and clears all breakpoints.

```
.START 4224

*
```

6. You can reenter ODT by specifying O.ODT+4 as the start address.

```
.START 4226

BE004242
*
```

If ODT is awaiting a command, a CTRL/C from the keyboard calls the RT-11 keyboard monitor. The monitor responds with ^C on the terminal and awaits a command. (You can use the monitor REENTER command to reenter ODT only if your program has set the reenter bit and ODT is linked high in memory relative to the program; otherwise, ODT is reentered at address O.ODT+4 as shown in Example 6 in Section 16.1.)

If you type CTRL/U during a search printout, the search terminates and ODT prints an asterisk.

16.2 RELOCATION

When the assembler produces a relocatable object module, the base address of the module is assumed to be location 000000. The addresses of all program locations, as shown in the assembly listing, are relative to this base address. After you link the module, many of the values and all of the addresses in the program are incremented by a constant whose value is the actual absolute base address of the module after it has been relocated. This constant is called the relocation bias for the module. Since a linked program can contain several relocated modules, each with its own relocation bias, and since, in the process of debugging, these biases have to be subtracted from absolute addresses continually in order to relate relocated code to assembly listings, RT-11 ODT provides automatic relocation.

The basis of automatic relocation is the eight relocation registers, numbered 0 through 7. You can set them to the values of the relocation biases at different times during debugging. Obtain relocation biases by consulting the link map. Once you set a relocation register, ODT uses it to relate relative addresses to absolute addresses. For more information on the exact nature of the relocation process, consult Chapter 11, Linker.

ODT evaluates a relocatable expression as a 16-bit (6-digit octal) number. You can type an expression in any one of the three forms presented in Table 16-1. In this table, the symbol n stands for an integer in the range 0 to 7 inclusive, and the symbol k stands for an octal number up to six digits long, with a maximum value of 177777. If you type more than six digits, ODT takes the last six digits typed, truncated to the low-order 16 bits. k can be preceded by a minus sign, in which case its value is the two's complement of the number typed. For example:

k (number typed)	Values
1	000001
-1	177777
400	000400
-177730	000050
1234567	034567

Section 16.3.13 describes the relocation register commands in greater detail.

Table 16-1 Forms of Relocatable Expressions (r)

Form	Expression	Value of r
A)	k	The value of k.
B)	n,k	The value of k plus the contents of relocation register n. (If the n part of this expression is greater than 7, ODT uses only the last octal digit of n.)
C)	C or C,k or n,C or C,C	Whenever you type the letter C, ODT replaces C with the contents of a special register called the constant register. (This value has the same role as the k or n that it replaces. The constant register is designated by the symbol $C and can be set to any value, as indicated below.)

16.3 COMMANDS AND FUNCTIONS

When ODT starts (as explained in Section 16.1) it indicates readiness to accept commands by printing an asterisk on the left margin of the terminal page. You can issue most of the ODT commands in response to the asterisk. You can examine a word and change it; you can run the object program in its entirety or in segments; you can search memory for specific words or references to them. The discussion below explains these features.

16.3.1 Printout Formats

Normally, when ODT prints addresses, it attempts to print them in relative form (Form B in Table 16-1). ODT looks for the relocation register whose value is closest to, but less than or equal to, the address to be printed. It then represents the address relative to the contents of the relocation register. However, if no relocation register fits the requirement, the address prints in absolute form. Since the relocation registers are initialized to -1 (the highest number), the addresses initially print in absolute form. If you change the contents of any relocation register, it can then, depending on the command, qualify for relative form.

For example, suppose relocation registers 1 and 2 contain 1000 and 1004 respectively, and all other relocation registers contain numbers much higher. In this case, the following sequence might occur (the slash command causes the contents of the location to be printed; the line feed command, LF, accesses the next sequential location):

```
*1000;1R                      sets relocation register 1 to 1000
*1,4;2R                       sets relocation register 2 to 1004
*774/000000 (LF)              opens location 774
000776  /007665 (LF)          opens location 776
1,000000  /000000 (LF)        opens absolute location 1000
1,000002  /000000 (LF)        opens absolute location 1002
2,000000  /000000             opens absolute location 1004
```

The printout format is controlled by the format register, $F. Normally, this register contains 0, in which case ODT prints addresses relatively whenever possible. You can open $F and change its contents to a non-zero value, however. In that case, all addresses print in absolute form (see Section 16.3.4, Accessing Internal Registers).

16.3.2 Opening, Changing, and Closing Locations

An open location is one whose contents ODT prints for examination, making those contents available for change. In a closed location, the contents are no longer available for change. Several commands are used for opening and closing locations.

Any command (except for the slash and backslash commands) that opens a location when another location is already open causes the currently open location to be closed. You can change the contents of an open location by typing the new contents followed by a single character command which requires no argument (i.e., LF, ^, RET, ←, @, >, <).

16.3.2.1 The Slash (/) – One way to open a location is to type its address followed by a slash. For example:

```
*1000/012746
```

This command opens location 1000 for examination and makes it ready to be changed.

If you do not want to change the contents of an open location, press the RETURN key to close the location. ODT prints an asterisk and waits for another command. However, to change the word, simply type the new contents before giving a command to close the location. For example:

```
*1000/012746 012345(RET)
*
```

This command inserts the new value, 012345, in location 1000 and closes the location. ODT prints another asterisk indicating its readiness to accept another command.

Used alone, the slash reopens the last location opened. For example:

```
*1000/012345 2340(RET)
*/002340
```

This command opens location 1000, changes its address to 002340, and then closes the location. ODT prints an asterisk, indicating its readiness to accept another command. The / character reopens the last location opened and verifies its value.

Remember that opening a location while another is open automatically closes the currently open location before opening the new location.

Also note that if you specify an odd-numbered address with a slash, ODT opens the location as a byte and subsequently behaves as if you had typed a backslash (see the following paragraph).

16.3.2.2 The Backslash (\\) – In addition to operating on words, ODT operates on bytes. Typing the address of the byte followed by a backslash character opens the byte. (On the LT33 or LT35 terminal, type \ by pressing the SHIFT key while typing the L key.) This causes ODT to print the byte value at the specified address, to interpret the value as ASCII code, and to print the corresponding character, if possible, on the terminal. (ODT prints a ? when it cannot interpret the ASCII value as a printable character.)

```
*1001\101 =A
```

A backslash typed alone reopens the last open byte. If a word was previously open, the backslash reopens its even byte:

```
*1002/000004 \004 =?
```

16.3.2.3 The LINE FEED Key (LF) – If you type the LINE FEED key when a location is open, ODT closes the open location and opens the next sequential location:

```
*1000/002340(LF)
001002 /012740
```

In this example, the LINE FEED causes ODT to print the address of the next location along with its contents and to wait for further instructions. After the above operation, location 1000 is closed and 1002 is opened. You can modify the open location by typing the new contents.

If a byte location is open, typing a line feed opens the next byte location.

16.3.2.4 The Circumflex or Up-Arrow (^) – If you type the circumflex (or up-arrow) when a location is open (circumflex is produced on an LT33 or LT35 by typing SHIFT/N), ODT closes the open location and opens the previous location. To continue from the example above:

```
*001002/012740 ^
001000 /002340
```

This command closes location 1002 and opens location 1000. You can modify the open location by typing the new contents.

If the opened location is a byte, the circumflex opens the previous byte.

16.3.2.5 The Underline or Back-Arrow (←) – If you type the underline, or back-arrow, (use SHIFT/O on an LT33 or LT35 terminal) to an open word, ODT interprets the contents of the currently open word as an address indexed by the program counter (PC) and opens the addressed location:

```
*1006/000006 _
001016 /000405
```

Notice in this example that the open location, 1006, is indexed by the PC as if it were the operand of an instruction with addressing mode 67 (PC relative mode).

You can make a modification to the opened location before you type a line feed, circumflex, or underline. Also, the new contents of the location will be used for address calculations using the underline command. For example:

```
*100/000222 4(LF)       modifies to 4 and opens next location
000102 /000111 6^       modifies to 6 and opens previous location
000100 /000004 200_     changes to 200 and opens location indexed
000302 /123456          by PC
```

16.3.2.6 Open the Addressed Location (@) – You can use the at (@) symbol (SHIFT/P on the LT33 or LT35 terminal) to optionally modify a location, close it, and then use its contents as the address of the location to open next. For example:

```
*1006/001044 @               opens location 1044 next
001044 /000500

*1006/001044 2100@           modifies to 2100 and opens location
002100 /000167               2100
```

16.3.2.7 Relative Branch Offset (>) – The right-angle bracket, >, optionally modifies a location, closes it, and then uses its low-order byte as a relative branch offset to the next word to be opened. For example:

```
*1032/000407  301>      modifies to 301 and interprets as a
000636 /000010          relative branch
```

Note that 301 is a negative offset (–77). ODT doubles the offset before it adds it to the PC; therefore, 1034+(-176)=636.

16.3.2.8 Return to Previous Sequence (<) – The left-angle bracket, <, lets you optionally modify a location, close it, and then open the next location of the previous sequence that was interrupted by an underline, @, or right-angle bracket command. Note that underline, @, or right-angle bracket causes a sequence change to the open word. If a sequence change has not occurred, the left-angle bracket simply opens the next location as a LINE FEED does. This command operates on both words and bytes.

```
*1032/000407   301>        > causes a sequence change
000636 /000010 <           returns to original sequence
001034 /001040 @           @ causes a sequence change
001040 /000405 \005 = <    < now operates on byte
001035 \002 =? <           < acts like (LF)
001036 \004 =?
```

16.3.3 Accessing General Registers 0-7

Open the program's general registers 0-7 with a command in the following format:

```
$n/
```

The symbol n is an integer in the range 0-7 that represents the desired register. When you open these registers, you can examine them or change their contents by typing in new data as with any addressable location. For example:

```
*$0/000033(RET)           examines register 0 then closes it
*

*$4/000474  464(RET)      opens register 4, changes its contents
*                         to 000464, then closes the register
```

The example above can be verified by typing a slash in response to ODT's asterisk:

```
*/000464
```

You can use the LINE FEED, circumflex, underline or @ command when a register is open.

16.3.4 Accessing Internal Registers

The program's status register contains the condition codes of the most recent operational results and the interrupt priority level of the object program. Open it by typing $S. For example:

```
*$S/000311
```

$S represents the address of the status register. In response to $S in the example above, ODT prints the 16-bit word, of which only the low-order eight bits are meaningful. Bits 0-3 indicate whether a carry, overflow, zero, or negative (in that order) has resulted, and bits 5-7 indicate the interrupt priority level (in the range 0-7) of the object program. (Refer to the *PDP-11 Processor Handbook* for the status register format.)

You can also use the $ to open certain other internal locations listed in Table 16-2.

Table 16-2 Internal Registers

Register	Section	Function
$B	16.3.6	Location of the first word of the breakpoint table
$M	16.3.9	Mask location for specifying which bits are to be examined during a bit pattern search
$P	16.3.15	Location defining the operating priority of ODT
$S	16.3.4	Location containing the condition codes (bits 0-3) and interrupt priority level (bits 5-7)

(Continued on next page)

Table 16-2 (Cont.) Internal Registers

Register	Section	Function
$C	16.3.10	Location of the constant register
$R	16.3.13	Location of relocation register 0, the base of the relocation register table
$F	16.3.1	Location of the format register

16.3.5 Radix-50 Mode (X)

Many PDP-11 system programs employ the Radix-50 mode of packing certain ASCII characters three to a word. You can use Radix-50 mode by specifying the MACRO .RAD50 directive. ODT provides a method for examining and changing memory words packed in this way with the X command.

When you open a word and type the X command, ODT converts the contents of the opened word to its 3-character Radix-50 equivalent and prints these characters on the terminal. You can then type one of the responses from Table 16-3.

Table 16-3 Radix-50 Terminators

Response	Effect
RETURN key (RET)	Closes the currently open location
LINE FEED key (LF)	Closes the currently open location and opens the next one in sequence
Circumflex (^)	Closes the currently open location and opens the previous one in sequence
Any three characters whose octal code is 040 (space) or greater	Converts the three characters into packed Radix-50 format. Legal Radix-50 characters for this response are: . $ Space 0 through 9 A through Z

If you type any other characters, the resulting binary number is unspecified (that is, no error message prints and the result is unpredictable). You must type exactly three characters before ODT resumes its normal mode of operation. After you type the third character, the resulting binary number is available to be stored in the opened location. Do this by closing the location in any one of the ways listed in Table 16-3. For example:

```
*1000/042431  X=KBI CBA(RET)
*1000/011421  X=CBA
```

NOTE

After ODT converts the three characters to binary, the binary number can be interpreted in one of many different ways, depending on the command that follows. For example:

```
*1234/063337  X=PRO  XIT/013704
```

Since the Radix-50 equivalent of XIT is 113574, the final slash in the example causes ODT to open location 113574 if it is a legal address.

16.3.6 Breakpoints

The breakpoint feature helps you monitor the progress of program execution. You can set a breakpoint at any instruction that is not referenced by the program for data. When a breakpoint is set, ODT replaces the contents of the breakpoint location with a BPT trap instruction so that program execution is suspended when a breakpoint is encountered. Then the original contents of the breakpoint location are restored, and ODT regains control.

With ODT, you can set up to eight breakpoints, numbered 0 through 7, at any one time. Set a breakpoint by typing the address of the desired location of the breakpoint followed by ;B. Thus, r;B sets the next available breakpoint at location r. (If all eight breakpoints have been set, ODT ignores the r;B command.) You can set or change specific breakpoints with the r;nB command, where n is the number of the breakpoint. For example:

```
*1020;B     sets breakpoint 0
*1030;B     sets breakpoint 1
*1040;B     sets breakpoint 2
*1032;1B    resets breakpoint 1
*
```

The ;B command removes all breakpoints. Use the ;nB command to remove only one of the breakpoints, where n is the number of the breakpoint. For example:

```
*;2B        removes the third breakpoint
*
```

ODT keeps a table of breakpoints; you can access that table. The $B/command opens the location containing the address of breakpoint 0. The next seven locations contain the addresses of the other breakpoints in order. You can sequentially open them by using the LINE FEED key. For example:

```
*$B/001020(LF)
001136 /001032(LF)
001140 /007070(LF)
001142 /007070(LF)
001144 /007070(LF)
001146 /001046(LF)
001150 /001066(LF)
001152 /007070
```

In this example, breakpoint 0 is set to 1020, breakpoint 1 is set to 1032, breakpoint 5 is set to 1046, and breakpoint 6 is set to 1066. The other breakpoints are not set.

Note that a repeat count in a proceed command (;P) refers only to the breakpoint that ODT most recently encountered. Execution of other breakpoints is determined by their own repeat counts.

16.3.7 Running the Program (r;G and r;P)

ODT controls program execution. There are two commands for running the program: r;G and r;P. The r;G command starts execution (go) and r;P continues (proceed) execution after halting at a breakpoint. For example:

```
*1000;G
```

This command starts execution at location 1000. The program runs until it encounters a breakpoint or until it completes. If it gets caught in an infinite loop, it must be either restarted or reentered as explained in Section 16.1.

Upon execution of either the r;G or r;P command, the general registers 0-6 are set to the values in the locations specified as $0-$6. The processor status register is set to the value in the location specified as $S.

When ODT encounters a breakpoint, execution stops and ODT prints Bn; (where n is the breakpoint number), followed by the address of the breakpoint. You can then examine locations for expected data. For example:

```
*1010;3B          sets breakpoint 3 at location 1010
*1000;G           starts execution at location 1000
B3;001010         stops execution at location 1010
*
```

To continue program execution from the breakpoint, type ;P in response to ODT's last prompt (*).

When you set a breakpoint in a loop, you can allow the program to execute a certain number of times through the loop before ODT recognizes the breakpoint. Set a proceed count by using the k;P command. This command specifies the number of times the breakpoint is to be encountered before ODT suspends program execution (on the kth encounter). The count, k, refers only to the numbered breakpoint that most recently occurred. You can specify a different proceed count for the breakpoint when it is encountered. Thus:

```
B3;001010         halts execution at breakpoint 3
*1026;3B          resets breakpoint 3 at location 1026
*4;P              sets proceed count to 4 and
B3;001026         continues execution; the program loops
*                 through the breakpoint three times and halts on
                  the fourth occurrence of the breakpoint
```

Following the table of breakpoints (as explained in Section 16.3.6) is a table of proceed command repeat counts for each breakpoint. You can inspect these repeat counts by typing $B/ and nine line feeds. The repeat count for breakpoint 0 prints (the first seven line feeds causes the table of breakpoints to be printed; the eighth types the single instruction mode, explained in the next section, and the ninth line feed begins the table of proceed command repeat counts). The repeat counts for breakpoints 1 through 7 and the repeat count for the single-instruction trap follow in sequence. ODT initializes a proceed count to 0 before you assign it a value. After the command has been executed, it is set to -1. Opening any one of these repeat counts provides an alternative way of changing the count. Once the location is open, you can modify its contents in the usual manner by typing the new contents followed by the RETURN key. For example:

```
.
.
.
nnnnnn   /001036(LF)         address of breakpoint 7
nnnnnn   /006630(LF)         single instruction address
nnnnnn   /000000  15(LF)     count for breakpoint 0; changes to 15
nnnnnn   /000000(LF)         count for breakpoint 1
.
.
.
nnnnnn   /000000(LF)         count for breakpoint 7
nnnnnn   /nnnnnn             repeat count for single instruction mode
```

Both the address indicated as the single instruction address and the repeat count for single instruction mode are explained in the following section.

16.3.8 Single Instruction Mode

With this mode, you specify the number of instructions to be executed before ODT suspends the program run. The proceed command, instead of specifying a repeat count for a breakpoint encounter, specifies the number of succeeding instructions to be executed. Note that breakpoints are disabled in single instruction mode. Table 16-4 lists the single instruction mode commands.

Table 16-4 Single Instruction Mode Commands

Command	Explanation
;nS	Enables single instruction mode. (n can be any digit and serves only to distinguish this form from the form ;S). Breakpoints are disabled.
n;P	Proceeds with program run for the next n instructions before reentering ODT. (If n is missing, it is assumed to be 1.) Trapping instructions and associated trap handlers can affect the proceed repeat count (see Section 16.4.2).
;S	Disables single instruction mode.

When the repeat count for single instruction mode is exhausted and the program suspends execution, ODT prints:

```
B8;nnnnnn
```

where nnnnnn is the address of the next instruction to be executed. The $B breakpoint table contains this address following that of breakpoint 7. However, unlike the table entries for breakpoints 0-7, direct modification has no effect.

Similarly, following the repeat count for breakpoint 7 is the repeat count for single instruction mode. You can modify this table entry directly. This is an alternative way of setting the single-instruction mode repeat count. In such a case, ;P implies the argument set in the $B repeat count table rather than an assumed 1.

16.3.9 Searches

With ODT, you can search all or any specific portion of memory for any bit pattern or for references to a particular location.

16.3.9.1 Word Search (r;W) – Before initiating a word search, you must specify the mask and search limits. The location represented by $M specifies the mask of the search. $M/ opens the mask register. The next two sequential locations (opened by LINE FEEDs) contain the lower and upper limits of the search. ODT examines in the search all bits set to 1 in the mask; it ignores other bits.

You must then give the search object and the initiating command, using the r;W command, where r is the search object. When ODT finds a match, (i.e., each bit set to 1 in the search object is set to 1 in the word ODT searches over the mask range) the matching word prints. For example:

```
*$M/000000 177400(LF)         tests high-order eight bits
nnnnnn  /000000  1000(LF)     sets low address limit
nnnnnn  /000000  1040(RET)    sets high address limit
*400;W                        initiates word search
001010  /000770
001034  /000404
*
```

In the above example, nnnnnn is an address internal to ODT; this location varies and is meaningful only for reference purposes. In the first line above, the slash was used to open $M, which now contains 177400; the LINE FEEDs open the next two sequential locations, which now contain the upper and lower limits of the search.

In the search process, ODT performs an exclusive OR (XOR) with the word currently being examined and the search object; the result is ANDed to the mask. If this result is 0, a match has been found and ODT reports it on the terminal. Note that if the mask is 0, all locations within the limits print. This provides a convenient method for dumping all memory locations within given limits using ODT.

Typing CTRL/U during a search printout terminates the search.

16.3.9.2 Effective Address Search (r;E) – ODT provides a search for words that reference a specific location. Open the mask register only to gain access to the low and high limit registers. After specifying the search limits (as explained for the word search), type the command r;E (where r is the effective address) to initiate the search.

Words that are an absolute address (argument *r* itself), a relative address offset, or a relative branch to the effective address print after their addresses. For example:

```
*$M/177400(LF)                          opens mask register only to gain
nnnnnn  /001000  1010(LF)               access to search limits
nnnnnn  /001040  1060(RET)
*1034;E                                 initiates search
001016  /001006                         relative branch
001054  /002767                         relative branch
*1020;E                                 initiates a new search
001022  /177774                         relative address offset
001030  /001020                         absolute address
```

Give particular attention to the reported effective address references. A word can have the specified bit pattern of an effective address without actually being used as one. ODT reports all possible references whether they are actually used or not.

Typing CTRL/U during a search printout terminates the search.

16.3.10 The Constant Register (r;C)

It is often desirable to convert a relocatable address into its value after relocation or to convert a number into its two's complement, and then to store the converted value in one or more places in a program. Use the constant register to perform this and other useful functions.

Typing r;C evaluates the relocatable expression to its 6-digit octal value, prints the value on the terminal, and stores it in the constant register. Invoke the contents of the constant register in subsequent relocatable expressions by typing the letter C. Examples follow:

```
*-4432;C=173346         places the two's complement of 4432 in the
                        constant register

*6632/062701  C(RET)    stores the contents of the constant register
                        in location 6632

*1000;1R                sets relocation register 1 to 1000

*1,4272;C=005272        reprints relative location 4272 as an
                        absolute location and stores it in the
                        constant register
```

16.3.11 Memory Block Initialization (;F and ;I)

Use the constant register with the commands ;F and ;I to set a block of memory to a specific value. While the most common value required is 0, other possibilities are +1, - 1, ASCII space, etc.

When you type the command ;F, ODT stores the contents of the constant register in successive memory words starting at the memory word address you specify in the lower search limit and ending with the address you specify in the upper search limit.

Typing the command ;I stores the low-order eight bits in the constant register in successive bytes of memory starting at the byte address you specify in the lower search limit and ending with the byte address you specify in the upper search limit.

For example, assume relocation register 1 contains 7000, 2 contains 10000, and 3 contains 15000. The following sequence sets word locations 7000-7776 to 0, and byte locations 10000-14777 to ASCII spaces:

```
                                         opens the mask register to gain
*$M/000000 (LF)                          access to search limits
nnnnnn  /000000  1;0(LF)                 sets the lower limit to 7000
nnnnnn  /000000  2;-2(LF)                sets the upper limit to 7776
*0;C=000000                              sets the constant register to zero
*;F                                      sets locations 7000-7776 to zero

*$M/000000 (LF)
nnnnnn /007000  2;0(LF)                  sets the lower limit to 10000
nnnnnn /007776  3;-1(RET)                sets the upper limit to 14777
*40;C=000040                             sets the constant register to 40
                                         (space)
*;I                                      sets the byte locations 10000-14777
*                                        to the value in the low-order 8
                                         bits of the constant register
```

16.3.12 Calculating Offsets (r;O)

Relative addressing and branching involve the use of an offset. An offset is the number of words or bytes forward or backward from the current location to the effective address. During the debugging session it is sometimes necessary to change a relative address or branch reference by replacing one instruction offset with another. ODT calculates the offsets in response to the r;O command.

The command r;O causes ODT to print the 16-bit and 8-bit offsets from the currently open location to address r. For example:

```
*346/000034  414;O  000044  022  22(RET)
*/000022
```

This command opens location 346, calculates and prints the offsets from location 346 to location 414, changes the contents of location 346 to 22 (the 8-bit offset), and verifies the contents of location 346.

The 8-bit offset prints only if it is in the range - 128(decimal) to 127(decimal) and the 16-bit offset is even, as was the case above. In the next example, the offset of a relative branch is calculated and modified so that it branches to itself.

```
*1034/103421  1034;O  177776  377  \021  377(RET)
*/103777
```

Note that the modified low-order byte 377 must be combined with the unmodified high-order byte.

16.3.13 Relocation Register Commands

The use of the relocation registers is described briefly in Section 16.2. At the beginning of a debugging session, it is desirable to preset the registers to the relocation biases of those relocatable modules that will be receiving the most attention. Do this by typing the relocation bias, followed by a semicolon and the specification of relocation registers, as follows:

```
r;nR
```

The symbol r can be any relocatable expression, and n is an integer in the range 0-7. If you omit n, it is assumed to be 0. For example:

```
*1000;5R          puts 1000 into relocation register 5
*5,100;5R         adds 100 to the contents
*                 of relocation register 5
```

Once a relocation register is defined, you can use it to reference relocatable values. For example:

```
*2000;1R          puts 2000 into relocation register 1
*1,2176/002466    examines the contents of location 4176
*1,3712;OB        sets a breakpoint at location 5712
```

Sometimes programs can be relocated to an address below the one at which they were assembled. This could occur with PIC code (position independent code), which is moved without using the linker. In this case, the appropriate relocation bias would be the two's complement of the actual downward displacement. One method for easily evaluating the bias and putting it in the relocation register is illustrated in the following example.

Assume a program was assembled at location 5000 and was moved to location 1000. Then the following sequence enters the two's complement of 4000 in relocation register 1.

```
*1000;1R
*1,-5000;1R
*
```

Relocation registers are initialized to -1 so that unwanted relocation registers never enter into the selection process when ODT searches for the most appropriate register.

To set a relocation register to -1, type ;nR. To set all relocation registers to -1, type ;R.

ODT maintains a table of relocation registers, beginning at the address specified by $R. Opening $R ($R/) opens relocation register 0. Successively typing a LINE FEED opens the other relocation registers in sequence. When a relocation register is opened in this way, you can modify it as you would any other memory location.

16.3.14 The Relocation Calculators, nR and n!

When a location has been opened, it is often desirable to relate the relocated address and the contents of the location back to their relocatable values. To calculate the relocatable address of the opened location relative to a particular relocation bias, type:

```
n!
```

The symbol n specifies the relocation register. This calculator works with opened bytes and words. If you omit n, the relocation register whose contents are closest to, but less than or equal to, the opened location is selected automatically by ODT. In the following example, assume that these conditions are fulfilled by relocation register 3, which contains 2000. Use the following command to find the most likely module that a given opened byte is in:

```
*2500\011  =  !=3,000500
```

To calculate the difference between the contents of the opened location and a relocation register, type:

```
nR
```

The symbol n represents the relocation register. If you omit n, ODT selects the relocation register whose contents are closest to but less than or equal to the contents of the opened location. For example, assume the relocation bias stored in relocation register 1 is 7000:

```
*1,500/11032  1R=1,2032
```

The value 2032 is the content of 1,500, relative to the base 7000. The next example shows the use of both relocation calculators.

If relocation register 1 contains 1000, and relocation register 2 contains 2000, use the following command to calculate the relocatable addresses of location 3000 and its contents relative to 1000 and 2000:

```
*3000/006410  1!=1,002000  2!=2,001000  1R=1,5410  2R=2,4410
```

16.3.15 ODT Priority Level, $P

$P represents a location in ODT that contains the interrupt (or processor) priority level at which ODT operates. If $P contains the value 377, ODT operates at the priority level of the processor at the time ODT is entered. Otherwise, $P can contain a value between 0 and 7 corresponding to the fixed priority at which ODT operates.

To set ODT to the desired priority level, open $P. ODT prints the present contents, which you can then change:

```
*$P/000006  4(RET)  lowers the priority to allow interrupts
*                   from the terminal
```

If you do not change $P, its value is seven.

You must set ODT's priority to 0 if you are using ODT in an FB environment while another job is running.

ODT does not always service breakpoints that are set in routines that run at different priority levels. For example, a program running at a low priority can use a device service routine that operates at a higher priority level. If you set $P low, ODT waits for terminal input at a low priority. If an interrupt occurs from a high priority routine, the breakpoints in the high priority routine are not recognized since they were removed when the earlier breakpoint occurred. That is, interrupts that are set at a priority higher than the one at which ODT is running are serviced, but any breakpoints are not recognized. To avoid this problem, set breakpoints at one priority level at a time. That is, set breakpoints within an interrupt service routine, but not at mainline code level. For a more complete discussion of how the PDP-11 handles priority and interrupts, refer to the processor handbook for your particular machine. ODT disables all breakpoints in the program whenever it gains control. Breakpoints are enabled when ;P and ;G commands are executed. For example:

```
*$P/00007  5
*1000;B
*2000;B
*1000;G
B0;001000
*                  an interrupt occurs and is serviced
```

If a higher level interrupt occurs while ODT is waiting for input, the interrupt is serviced, and no breakpoints are recognized.

16.3.16 ASCII Input and Output (r;nA)

Inspect and change ASCII text by using a command of this syntax:

```
r;nA
```

The symbol r represents a relocatable expression, and n is a character count. If you omit n, it is assumed to be 1. ODT does not check the magnitude of n. ODT prints n characters starting at location r, followed by a carriage return/line feed combination. Table 16-5 lists responses and their effect.

Table 16-5 ASCII Terminators

Response	Effect
RETURN key (RET)	ODT outputs a carriage return/line feed combination followed by an asterisk and waits for another command.
LINE FEED key (LF)	ODT opens the byte following the last byte output.
Up to n characters of text	ODT inserts the text into memory, starting at location r. If you type fewer than n characters, terminate the command by typing CTRL/U. This causes a carriage return/line feed/asterisk combination to print. However, if you type exactly n characters, ODT responds with a carriage return/line feed combination, the address of the next available byte, and then a carriage return/line feed/asterisk combination.

16.4 PROGRAMMING CONSIDERATIONS

Information in this section is not necessary for the efficient use of ODT. However, it does provide a better understanding of how ODT performs some of its functions. In certain difficult debugging situations, this understanding is necessary.

16.4.1 Using ODT with Foreground/Background Jobs

It is possible to use ODT to debug programs written as either background or foreground jobs. ODT does not debug virtual tasks that use extended memory. In the background or under the single-job monitor, you can link ODT with the program as described in Example 1 in Section 16.1. To debug a program in the foreground area, DIGITAL recommends that you run ODT in the background while the program to be debugged is in the foreground. The sequence of commands to do this is as follows:

```
.FRUN PROG/F                 loads the foreground program
LOADED AT nnnnnn             the first address of the job prints
.RUN ODT                     runs ODT in the background
ODT V01.01                   and sets a relocation register
*nnnnnn;0R                   to the start of the job

*$F/000000 0                 clears the format register to enable
*0,nnnnnn;0B                 proper address printing
                             sets a breakpoint

*0;G                         starts the keyboard monitor again

.RESUME                      starts the foreground job
```

The copy of ODT you use must be linked low enough so that it fits in memory along with the foreground job.

NOTE

Since ODT uses its own terminal handler, it cannot be used with the display hardware. If GT ON is in effect, ODT ignores it and directs its input and output only to the console terminal.

If you use ODT in a foreground/background environment while another job is running, set ODT's priority bit to 0 as follows:

```
*$P/000007 0(RET)
```

This puts ODT into the wait state at level 0, not at level 7. If you leave ODT's priority at 7, all interrupts (including clock) are locked out while ODT is waiting for terminal input.

16.4.2 Functional Organization

The internal organization of ODT is almost totally modularized into independent subroutines. The internal structure consists of three major functions: command decoding, command execution, and utility routines.

The command decoder interprets the individual commands, checks for command errors, saves input parameters for use in command execution, and sends control to the appropriate command execution routine.

The command execution routines take parameters saved by the command decoder and use the utility routines to execute the specified command. Command execution routines either return to the command decoder or transfer control to your program.

The utility routines are common routines such as SAVE-RESTORE and I/O. They are used by both the command decoder and the command executers.

16.4.3 Breakpoints

The function of a breakpoint is to give control to ODT whenever a program tries to execute the instruction at the selected address. Upon encountering a breakpoint, you can use all of the ODT commands to examine and modify the program.

When a breakpoint is executed, ODT removes all the breakpoint instructions from the code so that you can examine and alter the locations. ODT then types a message on the terminal in the form Bn;r, where r is the breakpoint address and n is the breakpoint number. ODT automatically restores the breakpoints when execution resumes.

There is a major restriction in the use of breakpoints: the program must not reference the word where a breakpoint was set since ODT altered the word. You should also avoid setting a breakpoint at the location of any instruction that clears the T-bit. For example:

```
MOV #240,177776        ;SET PRIORITY TO LEVEL 5
```

NOTE

Instructions that cause or return from traps (e.g., EMT, RTI) are likely to clear the T-bit, since a new word from the trap vector or the stack is loaded into the status register.

A breakpoint occurs when a trace trap instruction (placed in your program by ODT) is executed. When a breakpoint occurs, ODT operates according to the following algorithm:

1. Sets processor priority to 7 (automatically set by trap instruction).
2. Saves registers and sets up stack.

3. If internal T-bit trap flag is set, goes to step 13.
4. Removes breakpoints.
5. Resets processor priority to ODT's priority or user's priority.
6. Makes sure a breakpoint or single-instruction mode caused the interrupt.
7. If the breakpoint did not cause the interrupt, goes to step 15.
8. Decrements repeat count.
9. Goes to step 18 if non-zero; otherwise resets count to 1.
10. Saves terminal status.
11. Types message about the breakpoint or single-instruction mode interrupt.
12. Goes to command decoder.
13. Clears T-bit in stack and internal T-bit flag.
14. Jumps to the go processor.
15. Saves terminal status.
16. Types BE (bad entry) followed by the address.
17. Clears the T-bit, if set, in the user status and proceeds to the command decoder.
18. Goes to the proceed processor, bypassing the TT restore routine.

Note that steps 1-5 inclusive take approximately 100 microseconds. Interrupts are not permitted at this time, since ODT is running at priority level 7.

ODT processes a proceed (;P) command according to the following algorithm:

1. Checks the proceed for legality.
2. Sets the processor priority to 7.
3. Sets the T-bit flags (internal and user status).
4. Restores the user registers, status, and program counter.
5. Returns control to the user.
6. When the T-bit trap occurs, executes steps 1, 2, 3, 13, and 14 of the breakpoint sequence, restores breakpoints, and resumes normal program execution.

When a breakpoint is placed on an IOT, EMT, TRAP, or any instruction causing a trap, ODT follows this algorithm:

1. When the breakpoint occurs as described above, enters ODT.
2. When ;P is typed, sets the T-bit and executes the IOT, EMT, TRAP, or other trapping instruction.
3. Pushes the current PC and status (with the T-bit included) on the stack.
4. Obtains the new PC and status (no T-bit set) from the respective trap vector.
5. Executes the whole trap service routine without any breakpoints.
6. When an RTI is executed, restores the saved PC and PS (including the T-bit). Executes the instruction following the trap-causing instruction. If this instruction is not another trap-causing instruction, the T-bit trap occurs; reinserts the breakpoints in the user program, or decrements the single-instruction mode repeat count. If the following instruction is a trap-causing instruction, repeats this sequence starting at step 3.

NOTE

Exit from the trap handler must be by means of the RTI instruction. Otherwise, the T-bit is lost. ODT cannot regain control since the breakpoints have not yet been reinserted.

Note that the ;P command is illegal if a breakpoint has not occurred (ODT responds with ?). ;P is legal, however, after any trace trap entry.

The internal breakpoint status words have the following format:

1. The first eight words contain the breakpoint addresses for breakpoints 0-7. (The ninth word contains the address of the next instruction to be executed in single-instruction mode.)
2. The next eight words contain the respective repeat counts. (The following word contains the repeat count for single-instruction mode.)

You can change these words at will, either by using the breakpoint commands or by directly manipulating $B.

When program runaway occurs (that is, when the program is no longer under ODT control, perhaps executing an unexpected part of the program where you did not place a breakpoint) give control to ODT by pressing the HALT key to stop the computer and then restarting ODT (see Section 16.1). ODT prints an asterisk, indicating that it is ready to accept a command.

If the program you are debugging uses the console terminal for input or output, the program can interact with ODT to cause an error since ODT uses the console terminal as well. This interactive error does not occur when you run the program without ODT.

Note the following rules concerning the ODT break routine:

1. If the console terminal interrupt is enabled upon entry to the ODT break routine, and no output interrupt is pending when ODT is entered, ODT generates an unexpected interrupt when returning control to the program.
2. If the interrupt of the console terminal reader (the keyboard) is enabled upon entry to the ODT break routine, and the program is expecting to receive an interrupt to input a character, both the expected interrupt and the character are lost.
3. If the console terminal reader (keyboard) has just read a character into the reader data buffer when the ODT break routine is entered, the expected character in the reader data buffer is lost.

16.4.4 Searches

The word search lets you search for bit patterns in specified sections of memory. Using the $M/ command, specify a mask, a lower search limit ($M+2), and an upper search limit ($M+4). Specify the search object in the search command itself.

The word search compares selected bits (where 1s appear in the mask) in the word and search object. If all of the selected bits are equal, the unmasked word prints.

The search algorithm is as follows:

1. Fetches a word at the current address.
2. XORs (exclusive OR) the word and search object.
3. ANDs the result of step 2 with the mask.
4. If the result of step 3 is 0, types the address of the unmasked word and its contents; otherwise, proceeds to step 5.
5. Adds 2 to the current address. If the current address is greater than the upper limit, types * and returns to the command decoder; otherwise, goes to step 1.

Note that if the mask is 0, ODT prints every word between the limits, since a match occurs every time (i.e., the result of step 3 is always 0).

In the effective address search, ODT interprets every word in the search range as an instruction that is interrogated for a possible direct relationship to the search object. The mask register is opened only to gain access to the search limit registers.

The algorithm for the effective address search is as follows ((X) denotes contents of X, and K denotes the search object):

1. Fetches a word at the current address X.
2. If (X)=K [direct reference], prints contents and goes to step 5.
3. If (X)+X+2=K [indexed by PC], prints contents and goes to step 5.
4. If (X) is a relative branch to K, prints contents.
5. Adds 2 to the current address. If the current address is greater than the upper limit, performs a carriage return/line feed combination and returns to the command decoder; otherwise, goes to step 1.

16.4.5 Terminal Interrupt

Upon entering the TT SAVE routine, ODT follows these steps:

1. Saves the LSR status register (TKS).
2. Clears interrupt enable and maintenance bits in the TKS.
3. Saves the TT status register (TPS).
4. Clears interrupt enable and maintenance bits in the TPS.

To restore the TT:

1. Wait for completion of any I/O from ODT.
2. Restore the TKS.
3. Restore the TPS.

NOTES

1. If the TT printer interrupt is enabled upon entry to the ODT break routine, the following can occur:

    a. If no output interrupt is pending when ODT is entered, an additional interrupt always occurs when ODT returns control to the user.
    b. If an output interrupt is pending upon entry, the expected interrupt occurs when the user regains control.

2. If the TT reader (keyboard) is busy or done, the expected character in the reader data buffer is lost.

3. If the TT reader (keyboard) interrupt is enabled upon entry to the ODT break routine, and a character is pending, the interrupt (as well as the character) is lost.

16.5 ERROR DETECTION

ODT detects two types of error: illegal or unrecognizable command and bad breakpoint entry. ODT does not check for the legality of an address when you command it to open a location for examination or modification. Thus the command:

```
177774/
?MON-F-Trap to 4 003362
```

references nonexistent memory, thereby causing a trap through the vector at location 4. If the program you are debugging with ODT has requested traps through location 4 with the .TRPSET EMT, the program receives control at its TRPSET address.

Typing something other than a legal command causes ODT to ignore the command and to print:

```
(echoes illegal command) ?
*
```

and to wait for another command. Therefore, to cause ODT to ignore a command just typed, type any illegal character (such as 9 or RUBOUT) and the command will be treated as an error and ignored.

ODT suspends program execution whenever it encounters a breakpoint (that is, traps to its breakpoint routine). If the breakpoint routine is entered and no known breakpoint caused the entry, ODT prints:

```
BEnnnnnn
*
```

and waits for another command. BEnnnnnn denotes bad entry from location nnnnnn. A bad entry may be caused by an illegal trace trap instruction, by a T-bit set in the status register, or by a jump to some random location within ODT.

CHAPTER 17
PATCH

You can use the PATCH utility program to make code modifications to any RT-11 file (see Table 3-2 in this manual for a complete list of RT-11 file types). You use PATCH to interrogate and then to change words or bytes in the file.

It is always a good idea to create a backup version of the file you want to patch, because PATCH makes changes directly to the file as you work.

17.1 CALLING AND USING PATCH

To call PATCH from the system device, respond to the dot (.) printed by the keyboard monitor by typing:

```
R PATCH (RET)
```

PATCH then prints:

```
FILE NAME --
*
```

You should enter the name of the file you want to patch according to this general syntax:

```
filespec[/option...]
```

where:

filespec	represents the device, file name, and file type of the file you want to patch.
/option	is one of the options listed in Table 17-1.

If you do not specify a file type, PATCH assumes a .SAV file type.

17.1.1 PATCH Options

Table 17-1 summarizes the options that are valid for PATCH at this point in the opening command.

Table 17-1 PATCH Options

Option	Meaning
/A	Use with a device specification with or without a file specification. Use without a file specification to repair damaged RT-11 directories on directory-structured devices or to patch the bootstrap on disk block 0. Use with a file specification when the file is a source file or has a file type other than .SAV.
/M	Use if the file is an RT-11 monitor file.
/O	Use if the file is an overlay-structured file.
/C	Requires you to enter a checksum. If you make no modifications, PATCH ignores the /C option.
/D	Use if you do not know the checksum for a particular patch. PATCH prints the checksum for that patch. If you make no modifications, PATCH ignores the /D option.

Note that you must enter the complete file specification and accompanying options at this point; they are not legal at any other time. If you enter a carriage return instead of a file specification, however, PATCH prints its current running version number. It then repeats the prompt for a file specification.

After you enter the file specification, PATCH prints another asterisk and waits for commands.

17.1.2 Checksum

The checksum option helps you verify your work. It lets you compare the patch that you make to another patch that is known to be correct. The checksum does not tell you specifically where your error is, but it does tell you that an inconsistency exists.

PATCH can maintain a running total of the value of each command, argument, and character you enter. This total is called the checksum for the patch.

For example, if you receive from DIGITAL a patch to improve your system's performance, the patch contains a checksum value. You should use the /C option in the first PATCH command line, then make the modifications to your file exactly as shown in the DIGITAL patch. When you exit, PATCH asks you for a checksum. Enter the value from the DIGITAL patch. If the checksum you enter and the checksum that PATCH generated when you made your modifications do not match, PATCH prints the ?PATCH-W-CHECKSUM ERROR message. You then know that you made an error in patching your file, and that you need to try again.

17.2 PATCH COMMANDS

Table 17-2 summarizes the PATCH commands. Upper case characters represent PATCH commands; lower case characters represent octal values or ASCII characters. The following sections describe the commands in detail. Section 17.3 provides examples that use PATCH.

17.2.1 Patching a New File (F)

The F command causes PATCH to request you to enter a checksum, or it prints the required checksum (depending upon the options you specify). It also causes PATCH to close the currently open file and to print an asterisk indicating its readiness to accept another command string. No checksum dialogue is invoked if you have not previously specified checksum options (with /D or /C).

17.2.2 Exiting from Patch (E)

The E command causes PATCH to close the currently open file after printing the checksum dialogue according to the options you specify and return control to the RT-11 monitor. As with the F command, the checksum dialogue is by-passed if you have not specified checksum options.

17.2.3 Examining and Changing Locations in the File

For a non-overlay file, you can open a word address (as with ODT) by typing:

```
[relocation register,] offset/
```

PATCH types the contents of the location and waits for you to enter either a new location contents or another command.

For an overlay file, the format is:

```
[segment number:] [relocation register,] offset/
```

Segment number represents the overlay segment number as it is printed on the link map for the file. If you omit the segment number, PATCH assumes the root segment. If you make an error in a command string while patching an overlaid program, you can use CTRL/U to cancel the command. However, PATCH assumes the entire line is incorrect and preserves only the previously set relocation registers. PATCH preserves the segment number only across the ^ and (LF) commands.

Table 17-2 PATCH Commands

Command	Section	Explanation
v;nR	17.2.8	Sets relocation register n to value v.
x;B	17.2.7	Sets the bottom address of the overlay file to the value x.
r,o/	17.2.3	Opens the word location indicated by the contents of relocation register r plus offset o.
r,o\	17.2.3	Opens the byte location indicated by the contents of relocation register r plus offset o.
s:r,o/	17.2.3	Opens the word location indicated by the contents of relocation register r plus offset o in overlay segment s.
s:r,o\	17.2.3	Opens the byte location indicated by the contents of relocation register r plus offset o in overlay segment s.
RET	17.2.3	Closes the currently open word or byte.
LF	17.2.3	Closes the currently open word or byte and opens the next sequential word or byte.
n	17.2.3	Closes the currently open word or byte and opens the previous word or byte.
@	17.2.3	Closes the currently open word and opens the word it addresses.
F	17.2.1	Closes the file currently open and requests a new file specification.
E	17.2.2	Closes the file currently open and returns control to RT-11 monitor.
x;o	17.2.5	Indicates that a value in the overlay handler or its tables is being modified to the value x and that the overlay structure must be re-initialized. A value of 0 is illegal and generates an error message.
&	17.2.6	Indicates that PATCH should add the contents of all subsequently opened locations to the checksum, until it encounters another & symbol.
A	17.2.4	Prints the contents of the opened word or byte as ASCII characters (if a byte is open, one character prints; if a word is open, two characters print).
X	17.2.4	Prints the contents of the opened word as an unpacked Radix-50 word.
C(x[x])	17.2.4	Resets the contents of the opened word or byte to the ASCII value you type (if a byte is open, you must type one character; if a word is open, you must type two characters).
P(xxx)	17.2.4	Resets the contents of the currently opened word to the packed Radix-50 value of the three ASCII characters you type (you must type three characters).

Similarly, you can open a byte address in a file. The format for non-overlay files is:

```
[relocation register,] offset\
```

The format for overlay files is:

```
[segment number:] [relocation register,] offset\
```

Once a location has been opened, you can optionally type in the new contents in the format:

```
[relocation register,] octal value
```

Follow this line by one of the control characters from Table 17-3.

Table 17-3 PATCH Control Characters

Character	Function
RET	Closes the current location by changing contents to the new contents (if any) and awaits additional control input.
LF	Closes the current location by changing its content to the new contents (if any) and opens the next sequential word or byte.
^	Closes the current location by changing its contents to the new contents (if any) and opens the previous word or byte.
@	Closes the current word location and opens the word it addresses (in the same segment if it is an overlay file).

17.2.4 Translating and Indirectly Modifying Locations with a File

After opening a location within a file, you can translate the contents into ASCII characters or into the equivalent of a Radix-50 packed word.

To obtain the ASCII equivalent of the opened location, type the following command after PATCH prints the contents in octal.

```
A
```

PATCH then translates the word or byte into two (or one, if a byte is opened) ASCII characters. In this example, a byte is opened:

```
*1,100\ 102     A = B(LF)
```

PATCH prints only the printable ASCII characters in the opened word or byte (all non-printing characters, such as ASCII codes 0-37, are represented by the ? character). In this example, a word is opened:

```
*1, 100/ 302     A = B?(LF)
```

In the next example, a word is opened, and both ASCII characters are printable:

```
*1, 100/ 33502     A = B7(RET)
```

In these examples, one or both of the characters cannot be printed:

```
*0, 400/ 466      A = 6? (LF)

*1, 202/ 55001      A = ?Z (LF)

*616/ 401      A = ?? (RET)
```

To unpack a Radix-50 word as three ASCII characters, type the following command after PATCH prints the contents of the opened word.

```
X
```

PATCH then unpacks the opened word and prints three ASCII characters.

Note that you must open a word and not a byte.

If the word you open contains an illegal Radix-50 word, PATCH prints ???. If the translated character is not printable, PATCH prints ? in place of it.

Neither the A command nor the X command alters the contents of the open location; however, PATCH updates the checksum to reflect the fact that you have entered a new command.

You can specify the A and X commands in any order on the same command line without altering the contents of the open location. For example,

```
*1, 15022/    50553   X = MAC  A = kQ
```

After examining the location with the A or X command, you can change the location if you wish. For example,

```
*45660/10146   A = F?   X = BX8  12122 (RET) or (LF)
```

If the same location is reopened, the following change appears:

```
*45660/12122
```

You can change the contents of a location to the ASCII code of the value you specify by using the C command. You can use the P command to change a word to the packed Radix-50 word of the three characters you specify. This example changes an open byte to the ASCII code for the letter Z:

```
*1, 115\ 101      C (Z) (RET)
```

Note that PATCH prints the parentheses itself; you type only the character Z.

When reopened, that byte contains the ASCII code for Z:

```
*1, 115\ 132
```

Similarly, PATCH inserts the ASCII code for two ASCII characters into the low order and high order bytes, respectively, of one word. This example changes an open word to the ASCII code for AZ:

```
*0,10116/ 103523  C (AZ) ^
```

If reopened, the location contains the ASCII code for AZ:

```
    *0,10116/ 55101    A = AZ
```

You can examine the same location in more conventional ways, as this example shows:

```
    *0,10116\ 101(LF)
    0,10117\ 132
```

Similarly, you can use the P command to change the contents of an open word to the Radix-50 packed word equivalent of the three ASCII characters you specify. This example changes the Radix-50 word equivalent of SAV to REL:

```
    *2:1,400/ 73376  X = SAV   P (REL)(RET)
```

17.2.5 Setting Values in the Overlay Handler Tables of a Program

Use the ;O command to effect any changes to the overlay handler tables in an overlaid program. For example,

```
    *616/    1043     1100;O
    *
```

This command line increases the size of the referenced overlay region by 35(8) words or 58(10) bytes, to allow room for a patch. The value being modified is a value associated with the overlay handler tables, or a value required by the overlay handler for proper overlay structure initialization. The overlay structure is reinitialized and you can enter commands to modify the new region on the next line. A value of 0 is not permitted with the ;O command. If you omit the preceding argument, or use 0, an error message prints on the terminal.

17.2.6 Including the Old Contents Into the Checksum

Sometimes it is important that the present contents of the locations being changed have known specific values. This is the case when DIGITAL publishes system patches. The & command is designed to aid in implementing system patches. It automatically includes the old contents of an open location into the checksum. This command is a simple switch. The first occurrence of the & turns the switch on, the second turns it off. While the switch is on, the old contents of every location you open and close properly become part of the checksum. To use the & command, type:

```
    &
```

Patch then prints a carriage return-line feed sequence and another * indicating its readiness to accept another command. This switch is then enabled.

If you type the command on a line where a location is currently open, PATCH closes the location and resets the switch. PATCH then prompts with an asterisk indicating that it is ready to accept additional commands.

17.2.7 Setting the Bottom Address

To patch an overlay file, PATCH must know the bottom address at which the program was linked, if it is different from the initial stack pointer. This is the case if the program sets location 42 in an .ASECT. To set the bottom address, type:

```
    bottom address;B
```

You must issue the B command before you open any locations in an overlay for modification.

17.2.8 Setting Relocation Registers

You set the relocation registers 0-7 (as with ODT) with the R command. The R command has the syntax:

relocation value;relocation registerR

Be careful when you type this command string. If you inadvertently substitute a comma (,) for the semicolon (;) in the R command, PATCH does not generate an error message. However, it does not set the value you specify in the relocation register.

Once you set one of the eight relocation registers, the expression:

relocation register,octal number

in a command string will have the value:

relocation value + octal number

17.3 PATCH EXAMPLES

This section consists of two patch examples: one example for a non-overlaid file, and one example for an overlaid file. In each case, the steps that are taken to assemble, link, and patch the files are clearly illustrated.

The following command assembles the MACRO program PROMPT.MAC:

```
.MACRO/LIST PROMPT
ERRORS DETECTED:  0
```

The following listing is produced on the line printer as a result of the assembly. It consists of two parts: 1) the assembly listing of the source program and 2) the symbol table listing.

```
PROMPT.MAC       MACRO V03.00 5-MAY-77 16:54:30 PAGE 1

      1                                                              .TITLE  PROMPT.MAC
      2                                                              .MCALL  .PRINT,.EXIT
      3                                                              .MCALL  .TTYOUT,.TTYIN
      4 000000                                                       .CSECT  HGHSEG
      5                                                              .NLIST  BEX
      6 000000  112700  000052          START:  MOVB    #'*,R0               ; PRINT,...
      7 000004                                  .TTYOUT                      ; ...A PROMPT.
      8 000010                                  .TTYIN                       ; ACCEPT A CHARACTER FROM THE KEYBOARD
      9 000014  122700  000040                  CMPB    #' ,R0               ; IS IT A CONTROL CHARACTER?
     10 000020  101367                          BHI     START                ; YES - MUST BE A MISTAKE.
     11 000022  122700  000057                  CMPB    #'/,R0               ; NO - IS IT A "/"?
     12 000026  001011                          BNE     ERROR                ; NO - REPORT THE ERROR.
     13 000030                                  .TTYIN                       ; YES - GET NEXT.
     14 000034  122700  000126                  CMPB    #'V,R0               ; IS IT A "V" CHARACTER?
     15 000040  001004                          BNE     ERROR                ; NO - REPORT THE ERROR.
     16 000042                                  .PRINT  #MSG                 ; YES - PRINT THE VERSION MESSAGE...
     17 000050                          EXIT:   .EXIT                        ; ...AND THEN EXIT TO THE RT-11 MONITOR.
     18 000052                          ERROR:  .PRINT  #CMDERR              ; PRINT THE "?(RET)(LF)",...
     19 000060  000747                          BR      START                ; ...AND THEN RESTART.
     20 000062     077     000          CMDERR: .ASCIZ  /?/
     21 000064     016     012     106  MSG:    .ASCIZ  <16><12>/FILE  V03.01
     22                                                              .LIST   BEX
     23         000000'                                              .END    START

PROMPT.MAC       MACRO V03.00 5-MAY-77 16:54:30 PAGE 1-1
SYMBOL TABLE

CMDERR  000052R     002 ERROR   000052R     002 EXIT    000050R     002 MSG     000064R     002 START   000000R     002

. ABS.  000000      000
        000000      001
HGHSEG  000107      002
ERRORS DETECTED:  0

VIRTUAL MEMORY USED:  562 WORDS ( 3 PAGES)
DYNAMIC MEMORY AVAILABLE FOR  58 PAGES
DK:PROMPT,LP:PROMPT=DK:PROMPT.MAC/C
```

The next command links file PROMPT.OBJ and produces an executable module called PROMPT.SAV.

```
.LINK/MAP  PROMPT
```

The following listing is produced on the line printer as a result of the link operation.

```
RT-11 LINK  V03.01       LOAD MAP        THU 05-MAY-77 16:55:28
PROMPT.SAV       TITLE:  PROMPT  IDENT:

SECTION  ADDR   SIZE     GLOBAL  VALUE   GLOBAL  VALUE   GLOBAL  VALUE

. ABS.  000000  001000   (RW,I,GBL,ABS,OVR)
HGHSEG  001000  000110   (RW,I,GBL,REL,OVR)

TRANSFER ADDRESS = 001000, HIGH LIMIT = 001110 =  292. WORDS
```

The program PROMPT has an error. On line 21 the characters <16> should really be <15> . The following example uses PATCH to correct the error.

```
.R PATCH

FILE NAME--
*PROMPT/D
*1000;1R
*1,64\  16        15
*E

?PATCH-I-Checksum = 30633
.
```

The example shown above uses the /D option, which requests PATCH to print the checksum when the operation completes. Next, relocation register 1 is set to the transfer address, which the link map shows is 1000. The next command opens relative location 64, which contains the error, as the assembly listing shows at line sequence number 21. The value 15 is substituted for 16 (by typing 15 followed by a carriage return) and the exit command is issued (with E). PATCH then prints the checksum for the operation. It is 30633.

The next example verifies the change just made.

```
.R PATCH

FILE NAME--
*PROMPT
*1000;1R
*1,64\  15
1,65\   12
1,66\   106
1,67\   111
1,70\   114
1,71\   105
1,72\   40
1,73\   40
1,74\   126
1,75\   60
1,76\   63
```

```
1,77\    56
1,100\   60
1,101\   61
1,102\   40
1,103\   40
1,104\   15
1,105\   12
*E
.
```

As before, relocation register 1 is set to 1000 and location 64 is opened. Now it contains the correct value, 15. The rest of the command lines are terminated with a line feed. This closes the open location and opens the next one. This example shows the values from line 21 of the assembly listing (RET) (LF) FILE V03.01 (RET) (LF) as they are stored in memory.

The following commands assemble two MACRO programs: PTCH.MAC, the main program, and OVRLAY.MAC, the overlay.

```
.MACRO/LIST PTCH
ERRORS DETECTED:  0

.MACRO/LIST OVRLAY
ERRORS DETECTED:  0
.
```

The following listings were produced by the two assemblies.

```
PTCH.MAC        MACRO V03.00 5-MAY-77 17:58:35 PAGE 1

     1                                                      .TITLE  PTCH.MAC
     2                                                      .MCALL  .PRINT,.EXIT
     3 000000                                               .CSECT  HGHSEG
     4                                                      .GLOBL  ENTRY,MSG1
     5 000000  000403                    START:  BR     EXIT            ; BRANCH IMMEDIATELY TO CALL OVERLAY,
     6 000002  012700  000016'                   MOV    #MSG,R0         ; ALTERNATIVELY PRINT A MESSAGE
     7 000006                                    .PRINT                 ; DO THE PRINT.
     8 000010  004767  000000G           EXIT:   JSR    PC,ENTRY        ; CALL IN THE OVERLAY.
     9 000014                                    .EXIT                  ; THEN EXIT ON RETURN.
    10                                                      .NLIST  BEX
    11 000016     015     012     124    MSG:    .ASCIZ <15><12>/THIS IS A SUCCESSFUL PATCH/<15><12>
    12 000055     015     012     124    MSG1:   .ASCIZ <15><12>/THIS IS AN OVERLAY PATCH/<15><12>
    13                                                      .LIST   BEX
    14         000000'                                      .END    START

PTCH.MAC        MACRO V03.00 5-MAY-77 17:58:35 PAGE 1-1
SYMBOL TABLE

ENTRY = ****** G        EXIT    000010R     002 MSG     000016R    002 MSG1    000055RG    002 START   000000R    002

. ABS.  000000     000
        000000     001
HGHSEG  000112     002
ERRORS DETECTED:  0

VIRTUAL MEMORY USED:  509 WORDS  ( 2 PAGES)
DYNAMIC MEMORY AVAILABLE FOR  59 PAGES
DK:PTCH,LP:PTCH=DK:PTCH


OVRLAY.MAC      MACRO V03.00 5-MAY-77 17:58:57 PAGE 1

     1                                                      .TITLE  OVRLAY.MAC
     2                                                      .MCALL  .PRINT
     3 000000                                               .CSECT  OVLSEG
     4                                                      .GLOBL  MSG1,ENTRY
     5 000000  000403                 ENTRY:  BR     RETURN             ; BRANCH IMMEDIATELY TO RETURN.
     6 000002  012700  000000G                MOV    #MSG1,R0           ; ALTERNATIVELY PRINT A MESSAGE
     7 000006                                 .PRINT
     8 000010  000207                 RETURN: RTS    PC                 ; THEN RETURN.
     9         000001                                            .END
```

```
OVRLAY.MAC       MACRO V03.00 5-MAY-77 17:58:57 PAGE 1-1
SYMBOL TABLE

ENTRY   000000RG    002 MSG1  = ****** G        RETURN  000010R    002

. ABS.  000000      000
        000000      001
OVLSEG  000012      002
ERRORS DETECTED:  0

VIRTUAL MEMORY USED:  354 WORDS  ( 2 PAGES)
DYNAMIC MEMORY AVAILABLE FOR  59 PAGES
DK:OVRLAY,LP:OVRLAY=DK:OVRLAY
```

The next command links the module PTCH.OBJ and the overlay module OVRLAY.OBJ, producing the executable module ROOT.SAV.

```
.LINK/MAP/PROMPT/EXECUTE:ROOT PTCH
*OVRLAY/O:1
*//

.
```

The following listing is the load map that results from this link.

```
RT-11 LINK  V03.01       LOAD MAP         THU 05-MAY-77 17:59:51
ROOT  .SAV       TITLE:  PTCH.M  IDENT:

SECTION  ADDR   SIZE    GLOBAL  VALUE   GLOBAL  VALUE   GLOBAL  VALUE

. ABS.  000000  001122  (RW,I,GBL,ABS,OVR)
HGHSEG  001122  000112  (RW,I,GBL,REL,OVR)
                        MSG1    001177
 SEGMENT SIZE = 001234 =   334. WORDS
 OVERLAY REGION 000001  SEGMENT 000001
OVLSEG  001236  000012  (RW,I,GBL,REL,OVR)
                        ENTRY @ 001236
 SEGMENT SIZE = 000012 =     5. WORDS


TRANSFER ADDRESS = 001122, HIGH LIMIT = 001250 =  340. WORDS
```

The following example shows how to patch an overlay segment.

```
.R PATCH

FILE NAME--
*ROOT/O/C
*1236;1R
*1:1,0/ 403      240
*E

Checksum?   45475

.R ROOT

THIS IS AN OVERLAY PATCH


.
```

The options /O and /C are used in the file specification. /O indicates that the file is overlaid. /C causes PATCH to verify that the changes are correct by requesting a checksum value, which it compares to the actual checksum value the changes generate internally. The patch for this example was supplied by an experienced user, and the checksum for the correct patch is known to be 45475.

The first command line sets relocation register 1 to the start of the overlay segment to be patched. The load maps show that overlay segment 1 begins at location 1236. The next command opens the first location in overlay segment 1. It contains a branch instruction (403). A no-op instruction is substituted for it (240) followed by a carriage return, and E is used to exit. PATCH then requests the checksum value and 45475 is entered. This matches the checksum that the changes generated internally, so control returns to the monitor and the patch is successful.

The program is executed by typing:

```
    R ROOT (RET)
```

Control branches immediately to the overlay segment. Since the branch instruction at ENTRY: is now inoperative, control passes to the next line and the message prints on the terminal.

NOTE

The linker allocates space for overlay segments in 256-word blocks. Each segment begins on a block boundary. If a particular overlay segment's size is less than a whole number multiple of 256, you can add patches in the free space that exists between the end of that overlay segment and the beginning of the next block. To do this you must first modify the word-count word in the overlay handler table so that the patches you add are included in the size of the overlay segment. Be careful not to patch into the next block, though, because the next overlay segment begins there.

CHAPTER 18
OBJECT MODULE PATCH UTILITY (PAT)

PAT, the RT-11 object module patch utility, allows you to patch, or update, code in a relocatable binary object module. PAT does not permit you to examine the octal contents of an object module; use PATCH (described in Chapter 17) to do that. PAT makes the patch to the object module by means of the procedure outlined in Figure 18-2. One advantage to using PAT is that you can add relatively large patches to an object module without performing any octal calculations. PAT accepts a file containing corrections or additional instructions and applies these corrections and additions to the original object module. You prepare the correction input in MACRO source form and assemble it with the MACRO-11 assembler.

Input to PAT is two files: 1) the original input file and 2) a correction file containing the corrections and additions to the input file. The input file consists of one or more concatenated object modules. You can correct only one of these object modules with a single execution of the PAT utility. The correction file consists of object code that, when linked by the linker, either replaces or appends to the original object module. Output from PAT is the updated input file.

18.1 CALLING AND USING PAT

To call PAT from the system device, respond to the dot (.) printed by the keyboard monitor by typing:

```
R PAT (RET)
```

The Command String Interpreter prints an asterisk at the left margin on the console terminal when it is ready to accept a command line. Chapter 6 describes the general syntax of the command line that PAT accepts.

Type two CTRL/Cs to halt PAT at any time (or a single CTRL/C to halt PAT when it is waiting for console terminal input) and return control to the monitor. To restart PAT, type R PAT in response to the monitor's dot. When PAT executes an operation it returns control to the RT-11 monitor.

Figure 18-1 shows how you use PAT to update a file (FILE1) consisting of three object modules (MOD1, MOD2, and MOD3) by appending a correction file to MOD2. After running PAT, you use the linker to relink the updated module with the rest of the file and to produce a corrected executable program.

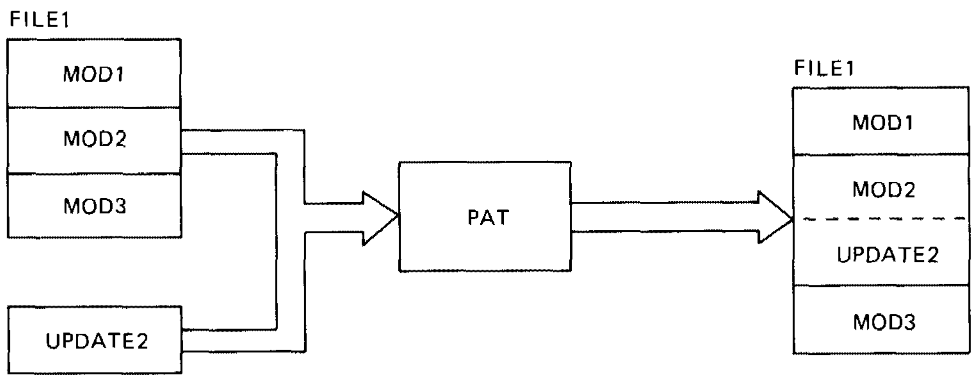


Figure 18-1 Updating a Module Using PAT

There are several steps you must perform to use PAT to update a file. First, create the correction file using a text editor. Then, assemble the correction file to produce an object module. Next, submit the input file and the correction file in object module form to PAT for processing. Finally, link the updated object module, along with the object modules that make up the rest of the file, to resolve global symbols and create an executable program. Figure 18-2 shows the processing steps involved in generating an updated executable file by using PAT.

Specify the PAT command string in the following form:

```
[output-filespec]=input-filespec[/C[:n]],correct-filespec[/C[:n]]
```

where:

output-filespec	is the file specification for the output file. If you do not specify an output file, PAT does not generate one.
input-filespec	is the file specification for the input file. This file can contain one or more concatenated object modules.
correct-filespec	is the file specification for the correction file. This file contains the updates being applied to a single module in the input file.
C	specifies the checksum option for the associated file. This directs PAT to generate an octal value for the sum of all the binary data composing the module in that file. (See Section 18.2.5 for more information on checksums.)
n	specifies an octal value. PAT compares the checksum value it computes for a module with the octal value you specify.

18.2 HOW PAT APPLIES UPDATES

PAT applies updates to a base input module by using the additions and corrections you supply in a correction file. This section describes the PAT input and correction files, gives information on how to create the correction file, and gives examples of how to use PAT.

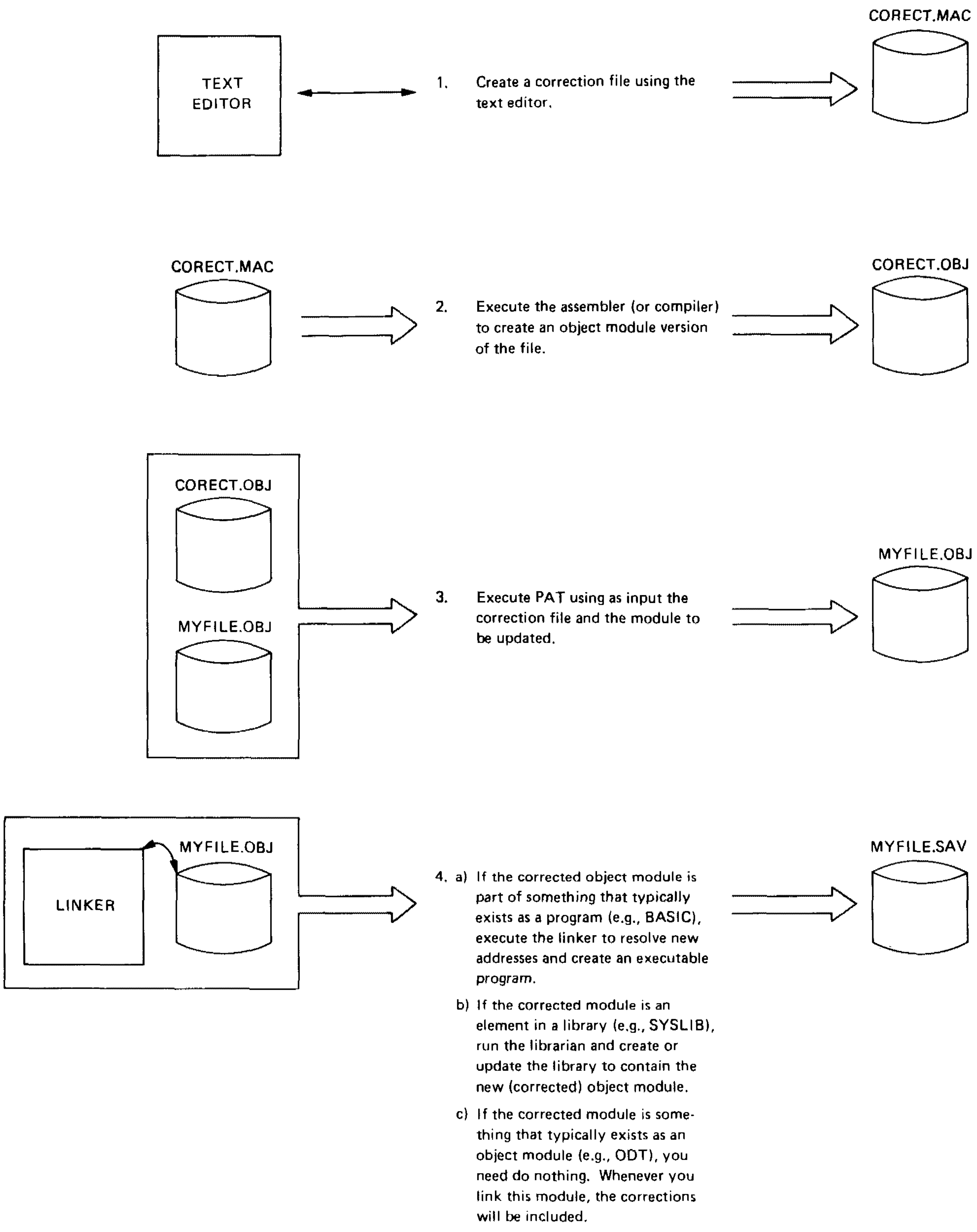
18.2.1 The Input File

The input file is the file to be updated; it is the base for the output file. The input file must be in object module format. When PAT executes, the module in the correction file applies to this file.

18.2.2 The Correction File

The correction file must also be in object module format. It is usually created from a MACRO-11 source file in the following format:

```
.TITLE inputname

.IDENT updatenum

inputline

inputline

  .

  .

  .
```



1. Create a correction file using the text editor.

2. Execute the assembler (or compiler) to create an object module version of the file.

3. Execute PAT using as input the correction file and the module to be updated.

4. a) If the corrected object module is part of something that typically exists as a program (e.g., BASIC), execute the linker to resolve new addresses and create an executable program.

   b) If the corrected module is an element in a library (e.g., SYSLIB), run the librarian and create or update the library to contain the new (corrected) object module.

   c) If the corrected module is something that typically exists as an object module (e.g., ODT), you need do nothing. Whenever you link this module, the corrections will be included.

Figure 18-2 Processing Steps Required to Update a Module Using PAT

where:

inputname	is the name of the module to be corrected by the PAT update. That is, inputname must be the same name as the name on the input file .TITLE directive for a single module in the input file.
updatenum	is any value acceptable to the MACRO-11 assembler. Generally, this value reflects the update version of the file being processed by PAT, as shown in the examples below.
inputline	are lines of input for PAT to use to correct and update the input file.

During execution, PAT adds any new global symbols that are defined in the correction file to the module's symbol table. Duplicate global symbols in the correction file supersede their counterparts in the input file, provided both definitions are relocatable or both are absolute.

A duplicate PSECT or CSECT supersedes the previous PSECT or CSECT, provided:

- both have the same relocatability attribute (ABS or REL):
- both are defined with the same directive (.PSECT or .CSECT).

If PAT encounters duplicate PSECT names, it sets the length attribute for the PSECT to the length of the longer PSECT and appends a new PSECT to the module.

If you specify a transfer address, it supersedes that of the module you are patching.

18.2.3 Creating the Correction File

As shown in Figure 18-2, the first step in using PAT to update an object file is to generate the correction file. Use the editor to build a file that contains these additions and corrections. The correction file must be in object module format before PAT can process it. Assemble the correction file with the MACRO-11 assembler to produce an object module that PATCH can process.

18.2.4 How PAT and the Linker Update Object Modules

The following examples show the source code for an input file and a correction file to be processed by PAT and the linker. The examples show as output a single source file that, if assembled and linked, would produce a binary module equivalent to the file generated by PAT and LINK. Two techniques are described: one is for overlaying lines in a module and the other is for appending a subroutine to a module.

18.2.4.1 Overlaying Lines in a Module – The first example illustrates a technique for overlaying lines in a module by using a patch file. First, PAT appends the correction file to the input file. The linker then executes to replace code within the input file.

The source code for the input file for this example is:

```
        .TITLE  ABC
        .IDENT  /01/
        .ENABL  GBL
ABC::
        MOV     A,C
        JSR     PC,XYZ
        RTS     PC
         .END
```

To add the instruction ADD A,B after the JSR instruction, the following patch source file is included:

```
          .TITLE   ABC
          .IDENT   /01.01/
          .ENABL   GBL
.=.+12
          ADD      A,B
          RTS      PC
          .END
```

The patch source is assembled using MACRO-11 and the resulting object file is input to PAT along with the original object file. The following source code represents the result of PAT processing:

```
          .TITLE   ABC
          .IDENT   /01.01/
          .ENABL   GBL
ABC::
          MOV      A,C
          JSR      PC,XYZ
          RTS      PC
.=ABC
.=.12
          ADD      A,B
          RTS      PC
          .END
```

After the linker processes these files, the load image appears as this source-code representation shows:

```
          .TITLE   ABC
          .IDENT   /01.01/
          .ENABL   GBL
ABC::
          MOV      A,C
          JSR      PC,XYZ
          ADD      A,B
          RTS      PC
          .END
```

The linker uses the .=.+12 in the program counter field to determine where to begin overlaying instructions in the program. The linker overlays the RTS instruction with the patch code:

```
          ADD      A,B
          RTS      PC
```

18.2.4.2 Adding a Subroutine to a Module – The second example illustrates a technique for adding a subroutine to an object module. In many cases, a patch requires that more than a few lines be added to patch the file. A convenient technique for adding new code is to append it to the end of the module in the form of a subroutine. This way, you can insert a JSR instruction to the subroutine at an appropriate location. The JSR directs the program to branch to the new code, execute that code, and then return to in-line processing.

The source code for the input file for the example is:

```
         .TITLE   ABC
         .IDENT   /01/
         .ENABL   GBL
ABC::
         MOV      A,B
         JSR      PC,XYZ
         MOV      C,R0
         RTS      PC
         .END
```

Suppose you wish to add the instructions:

```
         MOV      D,R0
         ASL      R0
```

between

```
         MOV      A,B
```

and

```
         JSR      PC,XYZ
```

The correction file to accomplish this goal is as follows:

```
         .TITLE   ABC
         .IDENT   /01.01/
         .ENABL   GBL
         JSR      PC,PATCH
         NOP
         .PSECT   PATCH
PATCH:
         MOV      A,B
         MOV      D,R0
         ASL      R0
         RTS      PC
         .END
```

PAT appends the correction file to the input file, as in the previous example. The linker then processes the file and generates the following output file:

```
         .TITLE   ABC
         .IDENT   /01.01/
         .ENABL   GBL
ABC::
         JSR      PC,PATCH
         NOP
         JSR      PC,XYZ
         MOV      C,R0
         RTS      PC
         .PSECT   PATCH
```

```
PATCH:
        MOV     A,B
        MOV     D,R0
        ASL     R0
        RTS     PC
        .END
```

In this example, the JSR PC,PATCH and NOP instructions overlay the three-word MOV A,B instruction. (The NOP is included because this is a case where a 2-word instruction replaces a 3-word instruction. NOP is required to maintain alignment.) The linker allocates additional storage for .PSECT PATCH, writes the specified code into this program section, and binds the JSR instruction to the first address in this section. Note that the MOV A,B instruction replaced by the JSR PC,PATCH is the first instruction the PATCH subroutine executes.

18.2.5 Determining and Validating the Contents of a File

Use the checksum option (/C) to determine or validate the contents of a module. The checksum option directs PAT to compute the sum of all binary data composing a file. If you specify the command in the form /C:n, /C directs PAT to compute the checksum and compare that checksum to the value you specify with n.

To determine the checksum of a file, enter the PAT command line with the /C option applied to the appropriate file (the file whose checksum you need to determine). For example:

```
      *=INFILE/C,INFILE.PAT
```

PAT responds to this command with the message:

```
?PAT-W-Input module checksum is nnnnnn
```

PAT generates a similar message when you request the checksum for the correction file.

To validate the changes made to a file, enter the checksum option in the form /C:n. PAT compares the value it computes for the checksum with the value you specify with n. If the two values do not match, PAT displays a message reporting the checksum error:

```
?PAT-W-Input file checksum error
```

or

```
?PAT-W-Correction file checksum error
```

Checksum processing always results in a nonzero value.

Do not confuse this checksum with the record checksum byte.

APPENDIX A
BATCH

RT-11 BATCH is a complete job control language that allows RT-11 to operate unattended. RT-11 BATCH processing is ideally suited to frequently-run production jobs, large and long-running programs, and programs that require little or no interaction with you, the user. Using BATCH, you can prepare your job on any RT-11 input device and leave it for the operator to start and run.

RT-11 BATCH permits you to:

- Execute an RT-11 BATCH stream from any legal RT-11 input device
- Output a log file to any legal RT-11 output device (except magtape or cassette)
- Execute the BATCH stream with the single-job monitor or in the background with the foreground/background monitor or the extended memory monitor
- Generate and support system-independent BATCH language jobs
- Execute RT-11 monitor commands from the BATCH stream.

RT-11 BATCH consists of 1) the BATCH compiler and 2) the BATCH run-time handler. The BATCH compiler reads the batch input stream you create, translates it into a format suitable for the RT-11 BATCH run-time handler, and stores it in a file. The BATCH run-time handler executes this file with the RT-11 monitor. As each command in the batch stream executes, BATCH lists the command, along with any terminal output generated by executing the command, on the BATCH log device.

A.1 HARDWARE AND SOFTWARE REQUIREMENTS TO RUN BATCH

You can run RT-11 BATCH on any single-job system that is configured with at least 12K words of memory. You need a minimum system of 16K words of memory to run BATCH in the background in a foreground/background environment. BATCH can run in any extended memory environment. A line printer, although optional, is highly desirable as the log device.

BATCH uses certain RT-11 system programs to perform its operations. For example, the $BASIC command executes the file BASIC.SAV. Make sure that the following RT-11 programs are on the system device, with exactly the following names, before you run BATCH:

BASIC.SAV	(BASIC users only)
BA.SYS	
BATCH.SAV	
CREF.SAV	(MACRO users only)
SYSLIB.OBJ	(FORTRAN and MACRO users)
FORTRA.SAV	(FORTRAN users only)
LINK.SAV	
MACRO.SAV	(MACRO users only)
PIP.SAV	
DIR.SAV	

A.2 BATCH CONTROL STATEMENT FORMAT

You can use two forms of input to RT-11 BATCH. Generate a file using the RT-11 editor and input it from any RT-11 input device, or input punched cards from the card reader. In both cases, the input consists of BATCH control statements. A BATCH control statement consists of three fields, separated from one another with spaces: 1) command fields, 2) specification fields, and 3) comment fields. The control statement has the syntax:

```
    $command/option      specification/option      [!comment]
```

Each control statement requires a specific combination of command and specification fields and options (see Section A.4). Control statements cannot be longer than 80 characters, excluding multiple spaces, tabs, and comments. You can use a hyphen (-) as a line continuation character to indicate that the control statement is continued on the next line (see Table A-4). Even if you use the line continuation character, the maximum control statement length is still 80 characters.

The following example of a $FORTRAN command illustrates the various fields in a control statement.

```
$FORTRAN/LIST/RUN     PROGA/LIBRARY     PROGB/EXE     !RUN FORTRAN

  command/options            spec fields/options         comment field
```

A.2.1 Command Fields

The command field in a BATCH control statement indicates the operation to be performed. It consists of a command name and certain command field options. Indicate the command field with a $ in the first character position and terminate it with a space, tab, blank, or carriage return.

A.2.1.1 Command Names – The command name must appear first in a BATCH control statement. All BATCH command names have a dollar sign ($) in the first position of the command (for example, $JOB). No intervening spaces are allowed in the command name. BATCH recognizes only two forms of a command name: the full name and an abbreviation consisting of $ and the first three characters of the command name. For example, you can enter the $FORTRAN command as:

```
    $FORTRAN
```

or

```
    $FOR
```

You cannot enter it as:

```
    $FORT
```

or

```
    $FORTR
```

A.2.1.2 Command Field Options – Options that appear in a command field are command qualifiers. Their functions apply to the entire control statement. All option names must begin with a slash (/) that immediately follows the command name. Table A-1 describes the command field options that are legal in BATCH and indicates the commands on which you can use them. Those option characters that appear in square brackets are optional. These are described in greater detail in the sections pertaining to the commands with which you use them.

NOTE

All /NO options are the defaults, except the /WAIT option in the $MOUNT and $DISMOUNT commands and the /OBJECT option in the $LINK command.

Table A-1 Command Field Options

Option	Explanation
/BAN[NER]	Prints the header of the job on the log file. BATCH allows this option only on the $JOB command.
/NOBAN[NER]	Does not print a job header.
/CRE[F]	Produces a cross reference listing during compilation; BATCH allows this option only on the $MACRO command.
/NOCRE[F]	Does not create a cross reference listing.
/DEL[ETE]	Deletes input files after the operation completes. BATCH allows this option on the $COPY and $PRINT commands.
/NODEL[ETE]	Does not delete input files after operation completes.

/DOL[LARS]	The data following this command can have a $ in the first character position of a line. BATCH allows this option on the $CREATE, $DATA, $FORTRAN, and $MACRO commands. BATCH terminates reading data when you use one of the following commands or when it encounters a physical end-of-file on the BATCH input stream:
	`$JOB`
	`$SEQUENCE`
	`$EOD`
	`$EOJ`

/NODOL[LARS]	Following data cannot have a $ in the first character position; a $ in the first character position signifies a BATCH control command.
/LIB[RARY]	Includes the default library in the link operation. BATCH allows this option on the $LINK and $MACRO commands.
/NOLIB[RARY]	Does not include the default library in the link operation.
/LIS[T]	Produces a temporary listing file (see Section A.2.5) on the listing device (LST:) or writes data images on the log device (LOG:). BATCH allows this option on the $BASIC, $CREATE, $DATA, $FORTRAN, $JOB, and $MACRO commands. When you use /LIST on the $JOB command, /LIST sends data lines in the job stream to the log device (LOG:).
/NOLIS[T]	Does not produce a temporary listing file.
/MAP	Produces a temporary link map on the listing device (LST:). BATCH allows this option on the $FORTRAN, $LINK, and $MACRO commands.

(Continued on next page)

Table A-1 (Cont.) Command Field Options

Option	Explanation
/NOMAP	Does not create a MAP file.
/OBJ [ECT]	Produces a temporary object file as output from compilation or assembly (see Section A.2.5). BATCH allows this option on the $FORTRAN, $LINK, and $MACRO commands. When you use /OBJECT on $LINK, BATCH includes temporary files in the link operation.
/NOOBJ [ECT]	Does not produce an object file as output of compilation; with $LINK, does not include temporary files in the link operation.
/RT11	Sets BATCH to operate in RT-11 mode (see Section A.5). BATCH allows this option only on the $JOB command.
/NORT11	Does not set BATCH to operate in RT-11 mode.
/RUN	Links (if necessary) and executes programs compiled since the last "link-and-go" operation or start of job. BATCH allows this option on the $BASIC, $FORTRAN, $LINK, and $MACRO commands.
/NORUN	Does not execute or link and execute the program after performing the specified command.
/TIM [E]	Writes the time of day to the log file when BATCH executes. BATCH allows this option only on the $JOB command.
/NOTIM [E]	Does not write the time of day to the log file.
/UNI [QUE]	Checks for unique spelling of options and keynames (see Section A.4.13). BATCH allows this option only on the $JOB command.
/NOUNI [QUE]	Does not check for unique spelling.
/WAI [T]	Pauses for operator action. BATCH allows this option on the $DISMOUNT, $MESSAGE, and $MOUNT commands.
/NOWAI [T]	Does not pause for operator action.
/WRI [TE]	Indicates that the operator is to WRITE ENABLE a specified device or volume. BATCH allows this option only on the $MOUNT command.
/NOWRI [TE]	Indicates that no writes are allowed or that the specified volume is read-only; informs the operator, who must WRITE LOCK the appropriate device.

A.2.2 Specification Fields

Specification fields immediately follow command fields in a BATCH control statement. Use them to name the devices and files involved in the command. You must separate these fields from the command field, and from each other, by blanks or spaces.

If a specification field contains more than one file to be used in the same operation, separate the fields by a plus (+) sign. For example, to assemble files F1 and F2 to produce an object file F3 and a temporary listing file, type:

```
$MACRO/LIST F1+F2/SOURCE F3/OBJECT
```

If you need to repeat a command for more than one field specification, separate the files by a comma (,). For example, the following command assembles F1 to produce F2, a temporary listing file, and a map file F3. It then assembles F4 and F5 to produce F6 and a temporary listing file.

```
$MACRO/LIST F1/SOURCE F2/OBJECT F3/MAP,F4+F5/SOURCE-
 F6/OBJECT
```

Note that the command field options apply to the entire line, but the specification field options apply only to the field they follow.

Depending on the command you use, specification fields can contain a device specification, file specification, or an arbitrary ASCII string. You can use an appropriate specification field option (see Table A-3) with any of these three items.

A.2.2.1 **Physical Device Names** – Represent each device in an RT-11 BATCH specification field with a standard 2- or 3-character device name. Table 3-1 in Chapter 3 lists each name and its related device. If you do not specify a unit number for devices that have more than one unit, BATCH assumes unit 0.

In addition to the permanent names shown in Table 3-1, you can assign logical device names to devices. A logical device name takes precedence over a physical name, thus providing device independence. With this feature, you do not need to rewrite a program that is coded to use a specific device if the device is unavailable. For example, DK: is normally assigned to the system device, but you can assign that name to diskette unit 1 (DX1:) with an RT-11 monitor ASSIGN command.

You must assign certain logical names prior to running any BATCH job. BATCH uses these logical names as default devices. These names are:

LOG:	BATCH log device (cannot be magtape or cassette)
LST:	Default device for listing files generated by BATCH stream.

The following are not legal device names in RT-11; if you use them, the operator must assign them as logical names with the ASSIGN command. You can use these names in BATCH streams written for other DIGITAL systems.

DF:	Fixed-head disk (RF).
LL:	Line printer with upper case and lower case characters.
M7:	7-track magtape.
M9:	9-track magtape.
PS:	Public storage (DK: as assigned by RT-11).

Refer to Sections 4.3 and A.7.1 for instructions on assigning logical names to devices.

A.2.2.2 **File Specifications** – You can reference files symbolically in a BATCH control statement with a name of up to six alphanumeric characters followed, optionally, by a period and a file type of three alphanumeric characters. Tabs and embedded spaces are not allowed in either the file name or file type. The file type generally indicates the format of a file. It is a good practice to conform to the standard file types for RT-11 BATCH. If you do not specify a file type for an output file, BATCH and most other RT-11 system programs assign appropriate default file types. If you do not specify a file type for an input file, the system searches for that file name with a default file type. Table A-2 lists the standard file types used in RT-11 BATCH.

Table A-2 File Types

File Type	Explanation
.BAS	BASIC source file (BASIC input)
.BAT	BATCH command file
.CTL	BATCH control file generated by the BATCH compiler.
.CTT	BATCH temporary file generated by the BATCH compiler.
.DAT	BASIC or FORTRAN data file
.DIR	Directory listing file
.FOR	FORTRAN IV source file (FORTRAN input)
.LST	Listing file
.LOG	BATCH log file
.MAC	MACRO source file (MACRO or SRCCOM input)
.MAP	Link map output from $LINK operation
.OBJ	Object file output from compilation or assembly
.SOU	Temporary source file
.SAV	$RUNable file or program image output from $LINK

A.2.2.3 **Wildcard Construction** – You can use wildcards in certain BATCH control statements (such as, $COPY, $CREATE, $DELETE, $DIRECTORY, $PRINT). You can use the asterisk as a wildcard to designate the entire file name or file type. See Chapter 4, Section 4.2, for a complete description of the wild card construction.

NOTE

You cannot use embedded wildcards (* or %) in BATCH control statements. However, you can use them in the keyboard monitor commands if you use the RT-11 mode of BATCH.

A.2.2.4 **Specification Field Options** – Specification field options follow file specifications in a BATCH control statement and designate how the file is to be used. These options apply only to the field in which they appear. Option names begin with a slash. The specification field options legal in RT-11 BATCH are listed in Table A-3. Optional characters in the option names are in square brackets.

Table A-3 Specification Field Options

Option	Explanation
/BAS[IC]	BASIC source file
/EXE[CUTABLE]	Indicates the executable program image file to be created as the result of a link operation
/FOR[TRAN]	FORTRAN source file
/INP[UT]	Input file; default if you specify no options
/LIB[RARY]	Library file to be included in link operation (prior to default library)
/LIS[T]	Listing file
/LOG[ICAL]	Indicates that the device is a logical device name; use in $DISMOUNT and $MOUNT commands
/MAC[RO]	MACRO source file
/MAP	Linker map file
/OBJ[ECT]	Object file (output of assembly or compilation)
/OUT[PUT]	Output file
/PHY[SICAL]	Indicates physical device name
/SOU[RCE]	Indicates source file
/VID	Volume identification

A.2.3 Comment Fields

Comment fields, which document a BATCH stream, are identified by an exclamation point (!) appearing anywhere except the first character position in the control statement. BATCH treats any character following the ! and preceding the carriage return/line feed combination as a comment. For example:

```
$RUN PIP       !DELETE FILES ON DK:
```

This command runs the RT-11 system program PIP. BATCH ignores the comment.

You can also include comments as separate comment lines by typing a $ in character position 1, followed immediately by the ! operator and the comment. For example:

```
$!DELETE FILES ON DK:
```

A.2.4 BATCH Character Set

The RT-11 BATCH character set is limited to the 64 upper case characters (ASCII 40 through 137). The current ASCII set is assumed (character 137 is underscore and not left arrow, and character 136 is circumflex, not up-arrow). The BATCH job control language does not support any control characters other than tab, carriage return, and line feed.

Table A-4 shows how BATCH normally interprets certain characters. Character interpretations are different if you use RT-11 mode (see Section A.5).

Table A-4 Character Explanation

Character	Explanation
blank/space	Specification field delimiter. It separates arguments in control statements. BATCH considers any string of consecutive spaces and tabs (except in quoted strings) as a blank (that is, equivalent to a single space).
!	Comment delimiter. The input routine ignores all characters after the exclamation point, up to the carriage return/line feed combination.
"	Passes a text string containing delimiting characters where the normal precedence rules would create the wrong action. For example, use it to include a space in a volume identification (/VID).
$	BATCH control statement recognition character. A dollar sign ($) in the first character position of a BATCH input stream line indicates that the line is a control statement.
.	Delimiter for file type.
-	Indicates line continuation if the character after the hyphen is one of the following: • A carriage return/line feed • Any number of spaces or tabs followed by a carriage return/line feed • A comment delimiter (!) • Spaces followed by a comment delimiter (!). If any other character follows the hyphen, the hyphen is assumed to be a minus sign indicating a negative value in an option.
/	Precedes an option name. An alphanumeric string must immediately follow it.
0-9	Numeric string components.
:	Immediately follows a device name. You can also use it to separate an option name from its value or to separate an option value from its subvalue (you can use : interchangeably with = for this purpose).
A-Z	Alphabetic string components.
=	Separates an option name from a value.
\	Illegal character except when it precedes a directive to the BATCH run-time handler from the operator (see Section A.7.3). (To include \ in an RT-11 mode command, use \\.)
+	File delimiter. Separates multiple files in a single specification field. Also indicates a positive value in options.

(Continued on next page)

Table A-4 (Cont.) Character Explanation

Character	Explanation
,	Separates sets of arguments for which the command is to be repeated.
*	A wildcard in utility command file specifications.
CR/LF	Carriage return/line feed. It indicates end-of-line (or end of logical record) for records in the BATCH input stream.

A.2.5 Temporary Files

When you do not include field specifications in a BATCH command line, BATCH sometimes generates temporary files. For example, you can enter a $FORTRAN command that is followed in the BATCH stream by the FORTRAN source program as:

```
$FORTRAN/RUN/OBJECT/LIST
    FORTRAN source program
$EOD
```

This command generates: 1) a temporary source file from the source statements that follow, 2) a temporary object file, 3) a temporary listing file, and 4) a temporary memory image file.

BATCH sends temporary files to the default device (DK:) or the listing device (LST:) according to their nature. If the device is file-structured, BATCH assigns file names and file types as follows:

nnnmmm.LST	for temporary listing files (sent to LST:)
nnnmmm.MAP	for temporary map files (sent to LST:)
nnnppp.OBJ	for temporary object files (sent to DK:)
nnnppp.SAV	for temporary memory image files (sent to DK:)
nnnppp.SOU	for temporary source files (sent to DK:)

where:

nnn	represents the last three digits of the sequence number assigned to the job by the $SEQUENCE command (see Section A.4.22). Thus, a sequence number of 12345 produces a file name beginning 345. If you do not use the $SEQUENCE command, BATCH sets nnn to 000.
mmm	represents the number of listing (or map) files that BATCH generated since the BATCH run-time handler (BA.SYS) was loaded. The first such file, listing or map, is 000. Each time BATCH generates a new temporary file, it increments the file name by 1. Thus, the second listing file produced under job sequence number 12345 is 345001.LST, and the first map file produced is 345000.MAP.
ppp	represents the number of object, memory image, or source files in the current BATCH run. The first such file (object, memory image, or source) is 000. Each time BATCH generates a new temporary file, it increments the file name by 1. BATCH resets these file names to 000 every time that you run BATCH and after every $LINK, $MACRO, or $FORTRAN command that uses the temporaries.

A.3 GENERAL RULES AND CONVENTIONS

You must adhere to the following general rules and conventions associated with RT-11 BATCH processing.

1. Always place a dollar sign ($) in the first character position of a command line.
2. Each job must have a $JOB and $EOJ command (or card).
3. You can spell out command and option names entirely or you can specify only the first three characters of the command and required characters of the option.
4. Specify wildcard construction (*) only for the utility commands ($COPY, $CREATE, $DELETE, $DIRECTORY, and $PRINT) and for commands that normally accept wildcards in RT-11 mode.
5. Include comments at the end of command lines or in a separate comment line. When you include comments in a command line, place them after the command but precede them by an exclamation mark.
6. Include only 80 characters per control statement (card record), excluding multiple spaces, tabs, and comments.
7. When you omit file specifications from BATCH commands and supply data in the BATCH stream, the system creates a temporary file with a default name (see Section A.2.5).
8. You can use the RT-11 monitor type-ahead feature only with BATCH handler directives (see Section A.7.3) to be inserted into a BATCH program. No other terminal input (except input to a foreground program) can be entered while a BATCH stream is executing.
9. You cannot use an indirect command file to call BATCH.

A.4 BATCH COMMANDS

Place BATCH commands in the input stream to indicate to the system which functions to perform in the job. All BATCH commands have a dollar sign ($) in the first character position (e.g., $JOB). Intervening spaces are not allowed in command names. The command name must always start in the first character position of the line (card column 1).

BATCH commands are presented in alphabetical order in this chapter for ease of reference. However, if you are not familiar with BATCH, read the commands in a functional order as listed in Table A-5. The characters shown in square brackets are optional.

Table A-5 BATCH Commands

Command	Section	Explanation
$SEQ[UENCE]	A.4.22	Assigns an arbitrary identification number to a job.
$JOB	A.4.13	Indicates the start of a job.
$EOJ	A.4.11	Indicates the end of a job.
$MOU[NT]	A.4.18	Signals the operator to mount a volume on a device and optionally assigns a logical device name.
$DIS[MOUNT]	A.4.9	Signals the operator to dismount a volume from a device and deassigns a logical device name.
$FOR[TRAN]	A.4.12	Compiles a FORTRAN source program.
$BAS[IC]	A.4.1	Compiles a BASIC source program.
$MAC[RO]	A.4.16	Assembles a MACRO source program.
$LIB[RARY]	A.4.14	Specifies libraries that BATCH should use in link operations.

(Continued on next page)

Table A-5 (Cont.) BATCH Commands

Command	Section	Explanation
$LIN[K]	A.4.15	Links modules for execution.
$RUN	A.4.21	Causes a program to execute.
$CAL[L]	A.4.2	Transfers control to another BATCH file, executes that BATCH file, and returns to the calling BATCH stream.
$CHA[IN]	A.4.3	Passes control to another BATCH file.
$DAT[A]	A.4.6	Indicates the start of data.
$EOD	A.4.10	Indicates the end of data.
$MES[SAGE]	A.4.17	Issues a message to the operator.
$COP[Y]	A.4.4	Copies files.
$CRE[ATE]	A.4.5	Creates new files from data included in the BATCH stream.
$DEL[ETE]	A.4.7	Deletes files.
$DIR[ECTORY]	A.4.8	Provides a directory of the specified device.
$PRI[NT]	A.4.19	Prints files.
$RT[11]	A.4.20	Specifies that the following lines are RT-11 mode commands.

For each command listed below, the term filespec represents a device name, a file name, and a file type.

It has this form:

```
dev:filnam.typ
```

As a general rule, BATCH assumes device DK: if you omit a device specification.

A.4.1 $BASIC Command

The $BASIC command calls RT-11 single-user BASIC to execute a BASIC source program. The $BASIC command has the following syntax:

```
$BASIC[/option ...] [filespec/option] ] [!comments]
```

where:

/option — indicates an option you can append to the $BASIC command. The options are as follows:

/RUN	indicates that BATCH should execute the source program.
/NORUN	indicates that BATCH should only compile the program, and send error messages to the log file.

/LIST	writes data images that are contained in the job stream to the log file (LOG:).
/NOLIST	writes data images to the log file only if you specify $JOB/LIST.

filespec — indicates the name and type of the source file and the device on which it resides. If you omit the file type, BATCH assumes .BAS. If you omit this specification, the source statements must immediately follow the $BASIC command in the input stream.

Terminate the source program after a $BASIC statement with either a $EOD command or with any other BATCH command that starts with a $ in the first position.

/option — indicates an option that can follow the source file name. BATCH assumes that any file name with no option appended is the name of a source file. This option can have one of the following values (or you can omit it):

/BASIC	indicates that the file name you specify is a BASIC source program.
/SOURCE	performs the same function as /BASIC.
/INPUT	performs the same function as /BASIC.

You can follow the $BASIC command with the source program, legal BASIC commands (such as RUN), or data. The following two BATCH streams, for example, produce the same results.

```
$BASIC              $BASIC/RUN
10 INPUT A          10 INPUT A
20 PRINT A          20 PRINT A
30 END              30 END
RUN                 $DATA
123                 123
$EOD                $EOD
```

A.4.2 $CALL Command

The $CALL command transfers control to another BATCH control file, temporarily suspending execution of the current control file. BATCH executes the called file until it reaches $EOJ or until the job aborts; control then returns to the statement following the $CALL in the originating BATCH control file. You can nest calls up to 31 levels. BATCH includes the log file for the called file in the log file for the originating BATCH program. (See NOTE following the $EOJ command.)

The syntax of the $CALL command is:

```
$CALL filespec[!comments]
```

BATCH does not permit options in the $CALL command. BATCH saves $JOB command options across a $CALL; however, they do not apply to the called BATCH file. If you specify .CTL as the file type, BATCH assumes a precompiled BATCH control file. If you do not specify a file type, BATCH assumes .BAT and compiles the called BATCH stream before execution.

NOTE

If the called program generates temporary files, those files can supersede currently existing temporary files if the two jobs have the same sequence number. For example, consider the following two BATCH streams:

```
$FOR/OBJ A          $FOR/OBJ A
$FOR/OBJ B          $CALL C
$LINK/RUN           $FOR/OBJ B
                    $LINK/RUN
```

The called BATCH file (C.BAT) contains the following:

```
$JOB
$FOR/OBJ A1
$FOR/OBJ B1
$LINK/RUN
$EOJ
```

The temporary object files that C.BAT generates change the behavior of the previous two BATCH statement sequences. The first temporary file created by C.BAT (000000.OBJ) supersedes the temporary file produced by the first $FORTRAN command (000000.OBJ). Avoid this situation by giving the BATCH job C.BAT a unique sequence number (see Section A.4.22).

A.4.3 $CHAIN Command

The $CHAIN command transfers control to a named BATCH control file but does not return to the input stream that executed the $CHAIN command. The syntax of the $CHAIN command is:

```
$CHAIN filespec[!comments]
```

BATCH does not permit options in the $CHAIN command. If you specify .CTL as the file type, BATCH assumes a precompiled BATCH control file. If you do not specify a file type, BATCH assumes .BAT and compiles the chained BATCH stream before execution.

A $EOJ command should always follow the $CHAIN command in the BATCH stream.

NOTE

The values of BATCH run-time variables remain constant across a $CALL, $CHAIN, or return from call. See Section A.5.2.2 for a description of these variables.

Use the $CHAIN command to transfer control to programs that you need to run only once at the end of a BATCH stream. For example, you could use the following BATCH program (PRINT.BAT) to print and then delete all temporary listing files generated during the current BATCH job.

```
$JOB                          !PRINT ALL LIST FILES
$PRINT/DELETE *.LST
$EOJ
```

You could then run PRINT.BAT with the $CHAIN command as follows:

```
$JOB
$MACRO/RUN  A  ALST/LIST
$MACRO/RUN  B  BLST/LIST
$CHAIN PRINT
$EOJ
```

A.4.4 $COPY Command

The $COPY command copies files in image mode from one device to another. You can use the wildcard construction (see Section A.2.2.3) in the input and output file specifications. You can concatenate several input files to form one output file (as long as the output specification does not contain a wildcard). The $COPY command has the following syntax:

```
$COPY[/option] output-filespec [ . . . , output-filespec] /OUTPUT-
   input-filespec[ . . . , input-filespec] [/INPUT] [!comments]
```

where:

/option	indicates options that you can append to the $COPY command.	
	/DELETE	deletes input files after the copy operation.
	/NODELETE	does not delete input files after the copy operation.
output-filespec	represents an output file. You must specify a file type.	
/OUTPUT	indicates that a file specification is for an output file.	
input-filespec	represents a file to be copied. BATCH copies files to the output file in the order that you list them (except when you use wildcards).	
/INPUT	indicates that a file specification is for an input file. If you do not specify an option, BATCH assumes INPUT.	

The following are examples of the $COPY command:

```
$COPY *.BAS/OUTPUT DT1:*.BAS
```

This command copies all files with the file type .BAS from the DECtape on unit 1 to the default storage device DK:.

```
$COPY FILE2.FOR/OUTPUT FILE0.FOR+FILE1.FOR
```

This command merges the input files FILE0.FOR and FILE1.FOR to form one file called FILE2.FOR and stores FILE2.FOR on device DK:.

```
$COPY *.*/OUT DT0:*.FOR, DT1:*.*/OUT DT0:*.*
```

This command copies all files with the file type .FOR from DT0: to DK: and all files on DT0: to DT1:.

A.4.5 $CREATE Command

The $CREATE command generates a file from data records that follow the $CREATE command in the input stream. An error occurs if the data does not immediately follow the $CREATE command. You cannot precede the data records with a $DATA command.

You can follow the $CREATE data with a $EOD command to signify the end of data, or you can use any other BATCH control statement to indicate end of data and initiate a new function. The $CREATE command has the following syntax:

```
$CREATE[/option . . .] filespec [!comments]
```

where:

/option	indicates an option you can append to the $CREATE command. The options are:

/DOLLARS	indicates that the data following this command can have a $ in the first character position of a line.
/NODOLLARS	indicates that a $ cannot be in the first character position of a line.
/LIST	writes data image lines to the log file.
/NOLIST	does not write data image lines to the log file. If you specify $JOB/LIST, BATCH ignores this option.

filespec	represents the file you want to create.

NOTE

If you use the /DOLLARS option, you must follow the last data record with a $EOD command (see Table A-1).

The following is an example of the $CREATE command:

```
$CREATE/LIST PROG.FOR
     FORTRAN source file
$EOD
```

The data records following the $CREATE command become a new file (PROG.FOR) on the default device (DK:). BATCH generates a listing on logical device LOG:.

A.4.6 $DATA Command

Use the $DATA command to include data records in the input stream. Data you include in this manner needs no file name. BATCH transfers the data to the appropriate program as though it were input from the console terminal. For example, you can follow the $RUN command for a particular program by a $DATA command and the data records for the program to process. The data records must be valid data for the program that is to use them.

The $DATA command has the following syntax:

```
$DATA[/option . . .] [!comments]
```

Four options that you can use with the $DATA command are as follows:

/DOLLARS	indicates that the data following this command can have a $ in the first character position of a line.

/NODOLLARS	indicates that a $ cannot be in the first character position of a line.
/LIST	writes data image lines to the log file.
/NOLIST	does not write data images to the log file. If you specify $JOB/LIST, BATCH ignores this option.

NOTE

Any command beginning with a $ normally follows the last data record. However, if you specify $DATA/DOLLARS, you must follow the last data record with $EOD.

The following example shows data entered into a BASIC program (TEST1.BAS).

```
$BASIC/RUN TEST1.BAS
$DATA
25,75,125,146
180,210,520,874
$EOD
```

A.4.6.1 **Using $DATA with FORTRAN Programs** – When you use the $DATA command to provide input to a FORTRAN program, you must insert a CTRL/Z into the BATCH file after the last data line and before $EOD (or before the next BATCH command if you do not use $EOD). This procedure permits FORTRAN to properly detect an end-of-file after it reads the last data line. For example:

```
$FORTRAN/RUN  A.FOR
$DATA
1
2
3
^Z (RET) (LF)
$EOD
$RUN PIP
```

The above program reads three numbers from the input stream and then detects an end-of-file when it attempts to read a fourth number. If you include an END=n statement in your FORTRAN program, statement n gets control when the end-of-file is detected. If the CTRL/Z (RET) (LF) is not present, the program aborts when it reaches $EOD and never executes the END=n statement.

A.4.7 $DELETE Command

Use the $DELETE command to delete files from the device you specify. This command has the syntax:

$DELETE filespec [. . . , filespec] [!comments]

where:

filespec	represents the name of a file to be deleted.

The following example deletes all files named TEST1 on the default device DK:.

```
$DELETE TEST1.*
```

The following example deletes all files with .FOR file types on DT1:, then deletes all files with .MAC file types on DK:.

```
    $DELETE DT1:*.FOR,*.MAC
```

A.4.8 $DIRECTORY Command

The $DIRECTORY command outputs a directory of the device you specify to a listing file. If you do not specify a listing file, the listing goes to the BATCH log file. This command has the syntax:

```
    $DIRECTORY [filespec/LIST] [filespec[ . . . , filespec] ] [/INPUT] [!comments]
```

where:

filespec/LIST	indicates the name of the directory listing file.
filespec/INPUT	indicates the input files to be included in the directory (default).

The following are examples of the $DIRECTORY command:

```
    $DIRECTORY
```

This command outputs a directory of the device DK: to the BATCH log file.

```
    $DIRECTORY FOR.DIR/LIST *.FOR
```

This command creates a directory file (FOR.DIR) on the device DK:. The directory contains the names, lengths, and dates of creation of all FORTRAN source files on the device DK:.

A.4.9 $DISMOUNT Command

The $DISMOUNT command removes the logical device name assigned by a $MOUNT command. When BATCH encounters $DISMOUNT while executing a job, it prints the entire $DISMOUNT command line on the console terminal. This message tells the operator which device to unload. This command has the syntax:

```
    $DISMOUNT[/option] logical-device-name:[/LOGICAL] [!comments]
```

where:

/option — indicates an option you can append to the $DISMOUNT command. The options are:

/WAIT	indicates that the job must pause until the operator enters a response. If you do not specify either /WAIT or /NOWAIT, BATCH assumes /WAIT. BATCH rings a bell at the terminal, prints the physical device name to be dismounted followed by a question mark (?), and waits for a response. (At this point you can enter input to the BATCH handler. See Section A.7.3.)
/NOWAIT	does not pause for operator response, BATCH prints the physical device name to be dismounted.

logical-device-name:	is the logical device name to be deassigned from the physical device.
/LOGICAL	identifies the device specification as a logical device name.

The following example instructs the operator to dismount the physical device with the logical device name OUT: and removes the logical assignment of device OUT:. In this example, OUT: is DT0:. The operator dismounts DT0: and then types a carriage return.

```
$DISMOUNT/WAIT OUT:/LOGICAL
DT0?
```

A.4.10 $EOD Command

The $EOD command indicates the end-of-data record or the end of a source program in the job stream. The syntax of this command is:

```
$EOD [!comments]
```

The $EOD command can signal the end of data associated with any of the following commands:

$BASIC
$CREATE
$DATA
$FORTRAN
$MACRO

In the following example, the $EOD command indicates the end of a source program that is to be compiled, linked, and executed.

```
$FORTRAN/RUN
     source program
$EOD
```

A.4.11 $EOJ Command

The $EOJ command indicates the end of a job. This command must be the last statement in every BATCH job. The command has the following syntax:

```
$EOJ [!comments]
```

If BATCH encounters a $JOB command, a $SEQUENCE command, or a physical end-of-file in the input stream before $EOJ, an error message appears in the log file.

NOTE

Make sure that the $EOJ command is the last line in a .BAT file.

A.4.12 $FORTRAN Command

The $FORTRAN command calls the FORTRAN compiler to compile a source program. Optionally, this command can provide printed listings or list files and can produce a link map in the listing. The $FORTRAN command has the following syntax:

```
$FORTRAN[/option . . .] [source-filespec[/option]] [filespec/OBJECT] [filespec/LIST] -
   [filespec/EXECUTE] [filespec/MAP] [filespec/LIBRARY] [!comments]
```

where:

/option	indicates an option you can append to the $FORTRAN command. The options are as follows:

/RUN	indicates that FORTRAN is to compile the source program, link it with the default library, and execute it. The default library is SYSLIB.OBJ. You can change it with the $LIBRARY command.
/NORUN	compiles the program only.
/OBJECT	produces a temporary object file.
/NOOBJECT	does not produce a temporary object file.
/LIST	produces a list file on the listing device (LST:).
/NOLIST	does not produce a list file.
/MAP	produces a link map on the listing device (LST:).
/NOMAP	does not create a MAP file.
/DOLLARS	indicates that the data following this command can have a $ in the first character position of a line.
/NODOLLARS	indicates that a $ cannot be in the first character position of a line.

source-filespec

indicates the device, file name, and file type of the FORTRAN source file. If you do not specify the file name, the $FORTRAN source statements must immediately follow the $FORTRAN command in the input stream; BATCH generates a temporary source file that it deletes after FORTRAN compiles the temporary source file (see Section A.2.5).

You can terminate the source program included after a $FORTRAN statement by either a $EOD command or by any other BATCH command. If, however, you use dollar signs in the first position in the source program, you must enter the source program with $CREATE/DOLLARS. In this case, you cannot use $FORTRAN/DOLLARS.

/option

represents an option that can have one of the following values:

/FORTRAN	indicates that the file name you specify is a FORTRAN source program. BATCH assumes that any file name with no option appended is the name of a source file.
/SOURCE	performs the same function as /FORTRAN.
/INPUT	performs the same function as /FORTRAN.

filespec/OBJECT

indicates the device, file name, and file type of the object file produced by compilation. The object file remains on the device you specify after the job finishes. You must follow the object file specification, if you include it, by the /OBJECT option.

If you omit the object file specification but specify $FORTRAN/OBJECT, BATCH creates a temporary object file. BATCH includes this temporary file in any $LINK operations that follow it in the job, and deletes it after the link operation.

filespec/LIST	indicates the name you assign to the list file created by the compiler. BATCH does not automatically print the list file if you assign LST: to a file-structured device, but you can list it using the $PRINT command. Follow the list file specification by the /LIST option.
filespec/EXECUTE	indicates the name you assign to a memory image file. Follow the memory image file specification by the /EXECUTE option. If you do not include this field, BATCH generates a temporary memory image file (see Section A.2.5) and then deletes the temporary file.
filespec/MAP	indicates the name you assign to the link map file created by the linker. Follow the map specification by the /MAP option.
filespec/LIBRARY	indicates that BATCH must include the file you specify in the link procedure as a library before SYSLIB.OBJ. The file must be a library file (produced by the RT-11 librarian). Follow the library specification by the /LIBRARY option.

The following are examples of $FORTRAN commands:

```
$FORTRAN/RUN PROGA.FOR
```

This command calls FORTRAN to compile and execute a source program named PROGA.FOR.

```
$FORTRAN/NOOBJ/LIST
    source program
$EOD
```

This command sequence compiles the FORTRAN program but does not produce an object file. BATCH creates a temporary listing file on LST:.

NOTE

See Section A.4.6.1 for instructions on using the $DATA command with FORTRAN programs.

A.4.13 $JOB Command

The $JOB command indicates the beginning of a job. Each job must have its own $JOB command. This command has the following syntax:

```
$JOB[/option . . . ] [!comments]
```

BATCH allows the following options in the $JOB command:

/BANNER	prints a header (a repetition of the $JOB command) on the log file.
/NOBANNER	does not print a job header.
/LIST	writes data image lines that are contained in the job stream to the log file.
/NOLIST	writes data image lines to the log file only when a /LIST option exists on a $BASIC, $CREATE, or $DATA command that has data lines following it.
/RT11	if no $ appears in column 1 when BATCH expects one, BATCH assumes that the line or card is an RT-11 mode command (see Section A.5).

/NORT11	does not process RT-11 mode commands.
/TIME	writes the time of day to the log file when BATCH executes command lines (except $DATA command lines).
/NOTIME	does not write the time of day.
/UNIQUE	checks for unique spelling of options and keynames. When you use this option, you can abbreviate commands and options to the least number of characters that still make their names unique. For example, you can abbreviate the /DOLLARS option to /DO since no other option begins with the characters DO.
/NOUNIQUE	checks only for normal option and keyname spellings.

End each job with a $EOJ command if you want to run it. If an input stream consists of more than one job, BATCH automatically terminates one job when it encounters the $JOB command for the next job. BATCH will never run a job terminated with another $JOB command; an error message will appear in the log.

The following $JOB command writes the time of day to the log file before BATCH executes each command beginning with a $. It also accepts unique abbreviations of BATCH commands and options.

```
$JOB/TIME/UNIQUE
```

A.4.14 $LIBRARY Command

The $LIBRARY command lets you specify a list of library files that will be included in FORTRAN links or with other link operations that have the /LIBRARY option. By default, the list of libraries contains only SYSLIB.OBJ, the RT-11 system library. This command has the syntax:

```
$LIBRARY filespec [!comments]
```

or

```
$LIBRARY filespec+SYSLIB [!comments]
```

where:

filespec	represents a library file; the default file type is .OBJ.
SYSLIB	is the RT-11 system library that you create at system generation.

Libraries are linked in order of their appearance in the $LIBRARY command.

The following example shows two libraries (LIB1.OBJ and LIB2.OBJ) that are included in FORTRAN links before SYSLIB.OBJ.

```
$LIBRARY LIB1.OBJ+LIB2.OBJ+SYSLIB.OBJ
```

A.4.15 $LINK Command

Use the $LINK command to produce memory image files from object files. This command links the files you specify (if any) with all temporary object files created since the last link or link-and-go operation (if any).

Temporary object files are those files you create as a result of a $FORTRAN or $MACRO command without naming an object file (with the /OBJECT option) or suppressing an object file (with the /NOOBJECT option). Create permanent object files by using the /OBJECT option on a $FORTRAN or $MACRO file descriptor.

BATCH links files in the following order:

1. First, it links temporary files in the order in which they were compiled.
2. Then, it links permanent files in the order in which they are specified in the $LINK command.
3. If the $LINK command specifies a library, BATCH links it next, providing that unresolved references remain.
4. If you specify $LINK/LIBRARY, BATCH searches and links the default library list.

The syntax for this command is:

```
$LINK[/option ... ] [filespec/OBJECT] [filespec/LIBRARY] [filespec/MAP] [filespec/EXECUTE] -
    [!comments]
```

where:

/option	indicates an option that you can append to the $LINK command. The options are as follows:

/LIBRARY	includes the RT-11 system library (SYSLIB.OBJ) and any default libraries specified in the $LIBRARY command in this $LINK operation. Use this option when the files being linked do not include any temporary FORTRAN object files. You can also use it when you specify $FORTRAN without the /RUN or /MAP option, but you want to search the default library list for unresolved references.
/NOLIBRARY	does not include the default libraries.
/MAP	produces a temporary load map on the listing device (LST:).
/NOMAP	does not produce a map file.
/OBJECT	includes temporary object files in the link. If you specify neither /OBJECT nor /NOOBJECT, BATCH assumes $LINK/OBJECT.
/NOOBJECT	does not include temporary files in the link.
/RUN	executes the memory image files associated with this $LINK command when the link is complete.
/NORUN	only links the program and does not execute it.

filespec/OBJECT	indicates the name of the object file BATCH must link. If you do not specify /OBJECT, BATCH assumes it as the default.
filespec/LIBRARY	indicates that the file you specify is to be included in the link procedure as a library. The file you specify must be a library file (produced by the RT-11 librarian).
filespec/MAP	indicates the load map file BATCH must create as a result of the $LINK command.
filespec/EXECUTE	indicates the memory image file BATCH must create as a result of the $LINK command.

The following are examples of the $LINK command:

```
    $LINK/RUN
```

This command links all temporary object files created since the last $LINK command or the last $FORTRAN/OBJ or $MACRO/OBJ command.

```
    $LINK/MAP PROG1.OBJ+PROG2.OBJ/OBJ PROGA.SAV/EXE
```

This command links the temporary files and the object files PROG1.OBJ and PROG2.OBJ to form a memory image file named PROGA.SAV. It also creates and outputs a temporary map file.

A.4.16 $MACRO Command

The $MACRO command calls the MACRO assembler to assemble a source program and, optionally, to provide printed listings or list files. You must specify MACRO listing directives, if any, in the source program. You cannot enter them at BATCH command level.

The $MACRO command has the following syntax:

```
    $MACRO[/option . . .] [source-filespec[/option] ] [filespec/OBJECT] [filespec/LIST] -
        [filespec/MAP] [filespec/LIBRARY] [filespec/EXECUTE] [!comments]
```

where:

/option — indicates an option you can append to the $MACRO command. The options are as follows:

/RUN	assembles, links, and runs the source program.
/NORUN	only assembles the source program.
/OBJECT	produces a temporary object file.
/NOOBJECT	does not produce a temporary object file.
/LIST	produces a listing file on the listing device (LST:).
/NOLIST	does not produce a list file.
/CREF	produces a cross reference listing during assembly.
/NOCREF	does not produce a cross reference listing during assembly.
/MAP	produces a link map as part of the listing file on LST:.
/NOMAP	does not create a MAP file.
/DOLLARS	indicates that the data following this command can have a $ in the first character position of a line.
/NODOLLARS	indicates that a $ cannot be in the first character position of a line.
/LIBRARY	includes the default library (SYSLIB.OBJ) in the link operation.

	/NOLIBRARY	does not include the default library in the link operation.
source-filespec		indicates the name of the source file. If you do not specify a file name, the $MACRO source statements must immediately follow the $MACRO command in the input stream.
		You can terminate the source program you include after a $MACRO statement by either a $EOD command or by any other BATCH command. If, however, you include dollar signs in the first position in the source program, you must use the $CREATE/DOLLARS command to enter the source program. In this case, you cannot use $MACRO/DOLLARS.
/option		can have one of the following values:
	/MACRO	indicates that the file name you specify is a MACRO source program. BATCH assumes that any file name with no option appended is the name of a source file.
	/SOURCE	performs the same function as /MACRO.
	/INPUT	performs the same function as /MACRO.
filespec/OBJECT		indicates the name you assign to the object file produced by compilation. The object file remains on the device you specify after the job finishes. If you include an object file specification, follow it with the /OBJECT option.
		If you omit the object file specification but specify $MACRO/OBJECT, BATCH creates a temporary object file. BATCH also includes the temporary object file in any $LINK operations that follow the $MACRO command in the job, and deletes it after the link operation (see Section A.2.5).
filespec/LIST		indicates the name you assign to the list file created by the assembler. BATCH does not automatically print the list file if you assign LST: to a file-structured device, but you can list it using the $PRINT command. The /LIST option must follow the list file specification.
filespec/MAP		indicates the file to which BATCH must output the storage map.
filespec/LIBRARY		indicates that BATCH must include the file you specify in the link procedure as a library. The /LIBRARY option must follow the library file specification.
filespec/EXECUTE		indicates the name you assign to a memory image file. The /EXECUTE option must follow the memory image file specification. If you do not include this field but do use $MACRO/RUN, BATCH generates a temporary memory image file (see Section A.2.5) and runs it.

The following $MACRO command assembles a program named PROGO.MAC and creates a temporary object file and a temporary listing file.

```
$MACRO/LIST/OBJECT PROGO.MAC
```

A.4.17 $MESSAGE Command

Use the $MESSAGE command to issue a message to the operator at the console terminal. It provides a means for the job to communicate with the operator. The $MESSAGE command has the syntax:

$MESSAGE[/option] message [!comments]

where:

/option	indicates an option you can append to the $MESSAGE command. The options are:	
	/WAIT	indicates that the job is to pause until the operator either types a carriage return to continue or enters commands to the BATCH handler followed by a carriage return (see Section A.7.3).
	/NOWAIT	does not pause for operator response.
message	is a string of characters that must fit on one console line. BATCH prints the message on the console.	

For example, if you include the following message in the input stream:

```
$MESSAGE/WAIT MOUNT SCRATCH TAPE ON MT0:
```

The message:

```
MOUNT SCRATCH TAPE ON MT0:
?
```

appears on the console terminal and a bell sounds. The operator mounts the tape and types carriage return to allow further processing of the job. (See Section A.7.3 for operator interaction with BATCH.)

NOTE

BATCH compresses multiple spaces and tabs in BATCH command lines; therefore, attempts to format $MESSAGE output with tabs or spaces do not provide you with the desired results.

A.4.18 $MOUNT Command

The $MOUNT command assigns a logical device name and other characteristics to a physical device. When BATCH encounters $MOUNT during the execution of a job, it prints the entire $MOUNT command line on the console terminal to notify the operator which volume to use.

The $MOUNT command has the syntax:

$MOUNT[/option . . .] physical-device-name: [/PHYSICAL] [/VID=x]
    [logical-device-name:/LOGICAL] [!comments]

where:

/option	indicates an option you can append to the $MOUNT command. The options are:	
	/WAIT	indicates that the job is to pause until the operator enters a response. If you do not specify either /WAIT or /NOWAIT, BATCH assumes /WAIT. BATCH rings a bell, prints the physical device name and a

question mark (?), and waits for a response. (The response can consist of input for the BATCH handler; see Section A.7.3.)

/NOWAIT	does not pause for operator response. BATCH prints the name of the physical device to be mounted.
/WRITE	tells the operator to WRITE ENABLE the volume.
/NOWRITE	tells the operator to WRITE PROTECT the volume.

physical-device-name	is required and specifies the physical device name and an optional unit number followed by a colon (for example, DX1:). If you specify a device name without a unit number, the operator can enter one in response to the question mark printed by the $MOUNT command. If you want the operator to supply a unit number, do not use the /NOWAIT option because it assumes unit 0.
/PHYSICAL	identifies the device specification as a physical unit specification. If you do not specify either /PHYSICAL or /LOGICAL, BATCH assumes /PHYSICAL.
/VID=x /VID="x"	provides volume identification. The volume identification is the name physically attached to the volume. Include it to help the operator locate the volume. Use this option only on the physical device file specification. If x contains spaces, specify it as "x".

NOTE

This volume identification is only a visual check for the operator. Make this volume identification match the visual label on the volume, not the volume identification that you wrote onto the volume at initialization time with the INITIALIZE/VOLUMEID command.

logical-device-name/LOGICAL

is required to identify the logical device name, if any, you assign to the device. The /LOGICAL option must follow the logical device name specification.

The following command instructs the operator to select a DECtape unit and mount DECtape volume BAT01 on that unit, WRITE ENABLED. It informs the operator by printing:

```
$MOUNT/WAIT/WRITE DT:/VID=BAT01 2:/LOGICAL
DT?
```

The operator selects a unit, mounts DECtape volume BAT01, WRITE ENABLED, and responds to the question mark by typing the unit number (such as 1) followed by a carriage return. BATCH assigns logical device name 2 to the physical device (in this case DT1:) and proceeds.

If no unit number response is necessary, as this command shows,

```
$MOUNT/WAIT/WRITE DT1: 2:/LOGICAL
DT1?
```

the operator responds with a carriage return after mounting the DECtape and WRITE ENABLING the device.

A.4.19 $PRINT Command

Use the $PRINT command to print the contents of the files you specify on the listing device (LST:). This command has the syntax:

$PRINT[/option] filespec [. . . ,filespec] [/INPUT] [!comments]

where:

/option	indicates an option you can append to the $PRINT command. The options are:
	/DELETE — deletes input files after printing.
	/NODELETE — does not delete input files after printing.
filespec	represents a file to be printed.
/INPUT	indicates that the file is an input file; BATCH assumes /INPUT if you omit it.

The following command prints a listing of files with file type .MAC that are stored on default device DK:.

```
$PRINT *.MAC
```

The following example creates listing files for the programs A and B, prints the listing files, and then deletes them.

```
$MACRO A.MAC A/LIST
$MACRO B.MAC B/LIST
$PRINT/DELETE A.LST,B.LST
```

A.4.20 $RT11 Command

The $RT11 command allows the BATCH job to communicate directly with the RT-11 system. DIGITAL recommends that you use RT-11 mode if you use BATCH. This command puts BATCH in RT-11 mode until BATCH encounters a line beginning with $. In RT-11 mode, BATCH interprets all data images as commands to the RT-11 monitor, to RT-11 system programs, or to the BATCH run-time system. The $RT11 command has the syntax:

$RT11 [!comments]

See Section A.5 for a complete description of the RT-11 mode.

A.4.21 $RUN Command

The $RUN command executes a program for which a memory image file (.SAV) was previously created. It can also run RT-11 system programs.

The $RUN command has the syntax:

$RUN filespec[!comments]

where:

filespec	represents the file to be executed. If you omit the file type, BATCH assumes .SAV.

For example, you can run DIR to print a directory listing:

```
$RUN DIR
$DATA
LP:=DK:/L
$EOD
```

A.4.22 $SEQUENCE Command

The $SEQUENCE command is an optional command. If you use it, it must immediately precede a $JOB command. The $SEQUENCE command assigns a job an arbitrary identification number. BATCH assigns the last three characters of a sequence number as the first three characters of a temporary listing or object file (see Section A.2.5). If a sequence number is less than three characters long, BATCH fills it with zeroes on the left.

The syntax of this command is:

$SEQUENCE id [!comments]

where:

id	represents an unsigned decimal number that indicates the identification number of a job.

The following are examples of the $SEQUENCE command:

```
$SEQUENCE 3           !SEQUENCE NUMBER IS 003
$JOB

$SEQUENCE 100         !SEQUENCE NUMBER IS 100
$JOB
```

A.4.23 Sample BATCH Stream

The following sample BATCH stream creates a MACRO program, assembles and links that program, and runs the memory image file. It then deletes the object, memory image, and source files it created and prints a directory of DK: showing the files the BATCH stream created.

```
$JOB
$MESSAGE        THIS IS AN EXAMPLE BATCH STREAM
$MESSAGE        NOW CREATE A MACRO PROGRAM
$CREATE/LIST    EXAMPL.MAC
.TITLE  EXAMPL FOR BATCH
        .MCALL  .PRINT,.EXIT
START:  .PRINT  #MESSAG
        .EXIT
MESSAG: .ASCIZ  /EXAMPLE MACRO PROGRAM FOR BATCH/
        .END    START
$EOD
$MACRO  EXAMPL EXAMPL/OBJECT EXAMPL/LIST    !ASSEMBLE
$LINK   EXAMPL EXAMPL/EXECUTE               !AND LINK
$PRINT/DELETE EXAMPL.LST
$MESSAGE        RUN THE MACRO PROGRAM
$RUN    EXAMPL                              !AND EXECUTE
$DELETE EXAMPL.OBJ+EXAMPL.SAV+EXAMPL.MAC
$MESSAGE        PRINT A DIRECTORY
$DIRECTORY      DK:EXAMPL.*
$MESSAGE        END OF THE EXAMPLE BATCH STREAM
$EOJ
```

To run this batch stream, type the following commands at the console. BATCH prints the messages.

```
.LOAD BA,LP
.ASSIGN LP: LOG
.ASSIGN LP: LST
.R BATCH
*EXAMPL
 THIS IS AN EXAMPLE BATCH STREAM
 NOW CREATE A MACRO PROGRAM
 RUN THE MACRO PROGRAM
 PRINT A DIRECTORY
 END OF THE EXAMPLE BATCH STREAM

END BATCH
.
```

The above sample BATCH stream produces the following log file on the line printer:

NOTE

The amount of free core and the directory format are variable.

```
$JOB

$MESSAGE        THIS IS AN EXAMPLE BATCH STREAM

$MESSAGE        NOW CREATE A MACRO PROG.

$CREATE/LIST    EXAMPL.MAC

.TITLE  EXAMPLE FOR BATCH
        .MCALL  .PRINT,.EXIT
START:  .PRINT  #MESSAG
        .EXIT
MESSAG: .ASCIZ  /EXAMPLE MACRO PROGRAM FOR BATCH/
        .EVEN
        .END    START
0

$EOD

$MACRO  EXAMPL EXAMPL/OBJECT EXAMPL/LIST   !ASSEMBLE

ERRORS DETECTED:  0

EXAMPLE FOR BATCH       MACRO V03.00 21-JUN-77 00:05:29 PAGE 1

      1                                 .TITLE  EXAMPLE FOR BATCH
      2                                         .MCALL  .PRINT,.EXIT
      3 000000                          START:  .PRINT  #MESSAG
      4 000006                                  .EXIT
      5 000010    105     130     101   MESSAG: .ASCIZ  /EXAMPLE MACRO PROGRAM FOR BATC
        000013    115     120     114
        000016    105     040     115
        000021    101     103     122
        000024    117     040     120
        000027    122     117     107
        000032    122     101     115
        000035    040     106     117
        000040    122     040     102
        000043    101     124     103
        000046    110     000
      6                                         .EVEN
      7        000000'                          .END    START
```

```
EXAMPLE FOR BATCH       MACRO V03.00 21-JUN-77 00:05:29 PAGE 1-1
SYMBOL TABLE

MESSAG  000010R         START   000000R

. ABS.  000000     000
        000050     001
ERRORS DETECTED:  0

VIRTUAL MEMORY USED:  508 WORDS  ( 2 PAGES)
DYNAMIC MEMORY AVAILABLE FOR  48 PAGES
EXAMPL,EXAMPL=EXAMPL

$LINK   EXAMPL EXAMPL/EXECUTE               !AND LINK

$PRINT/DELETE EXAMPL.LST


$MESSAGE        RUN THE MACRO PROGRAM

$RUN    EXAMPL                                  !AND EXECUTE

EXAMPLE MACRO PROGRAM FOR BATCH

$DELETE EXAMPL.OBJ+EXAMPL.SAV+EXAMPL.MAC


$MESSAGE        PRINT A DIRECTORY

$DIRECTORY      DK:EXAMPL.*

 21-JUN-77
EXAMPL.BAK     2 14-JUN-77      EXAMPL.BAT     2 21-JUN-77
EXAMPL.CTL     3 21-JUN-77
 3 FILES, 7 BLOCKS
 1903 FREE BLOCKS

$MESSAGE        END OF THE EXAMPLE BATCH STREAM

$EOJ
```

A.5 RT-11 MODE

RT-11 mode lets you enter commands to the RT-11 monitor or to system programs, and lets you create BATCl programs. You can enter RT-11 mode with either the $JOB/RT11 command or the $RT11 command. If you ei RT-11 mode with the $JOB/RT11 command, RT-11 mode remains in effect until BATCH encounters the next command. If you enter RT-11 mode with the $RT11 command, RT-11 mode is in effect until BATCH encount a $ in the first position of the command line.

The characters . , $, *, and tab or space appearing in the first position of a line (or card column 1) are control cl acters and indicate the following:

.	command to the RT-11 monitor, such as `.R PIP`
*	data line; any line not intended to go to the RT-11 monitor or to the BATCH run handler, such as a command to the RT-11 PIP program: `*FILE1.DAT/D`

NOTE

BATCH does not pass the * as data to the program. Comment lines (!) cannot appear on data lines as BATCH would consider them as data.

$	BATCH command. It causes an exit from RT-11 mode if you entered RT-11 mode with the $RT11 command. For example:

```
$RT11                        !ENTER RT-11 MODE
.R PIP
*FILE1.DAT/D
$FORTRAN                     !LEAVE RT-11 MODE
```

space/tab	separator to indicate a line directed to the BATCH run-time handler. This separator is indicated by a (TAB) in the following descriptions.

A.5.1 Communicating with RT-11

The most common use of RT-11 mode is to send commands to the RT-11 monitor and to run system programs. For example, you can insert the following commands in the BATCH stream to run PIP and save backup copies of files on DECtape:

```
$RT11
.R PIP
*DT1:*.*=*.FOR
```

You must anticipate and include in the BATCH input stream responses that the called program requires, such as the Y response to DUP's ARE YOU SURE? query. Place a line in your BATCH file consisting of Y and RETURN or use the DUP /Y option to suppress the query. For example:

```
$RT11
.INITIALIZE RK1:
*Y
```

You can communicate directly with the RT-11 monitor by using the keyboard monitor commands that are described in Section 4.3. For example:

```
$RT11
.DELETE/NOQUERY DX1:*.MAC
```

This command deletes all files with a file type of .MAC from device DX1:.

You cannot mix BATCH standard commands with RT-11 mode data lines (lines beginning with an asterisk). For example, the proper way to do a $MOUNT within a sequence of RT-11 mode data commands is:

```
$JOB/RT11
.R MACRO
*A1=A1
*A2=A2
$MOUNT DT0:/PHYSICAL
.R MACRO
*B1=DT:B1
*B2=DT:B2
```

A.5.2 Creating RT-11 Mode BATCH Programs

Advanced system programmers can use RT-11 mode to create BATCH programs. These BATCH programs consist of standard RT-11 mode commands (monitor commands, data lines for input to system programs, etc.) plus special RT-11 mode commands. The BATCH run-time handler interprets these special commands to allow dynamic calculations and conditional execution of the RT-11 mode standard commands. The following can help you create BATCH programs and dynamically control their execution at run-time:

- Labels
- Variable modification:

  1) equating a variable to a constant or character (LET statement)
  2) incrementing the value of a variable by 1
  3) reading a value into a variable
  4) conditional transfers on comparison of variable values with numeric or character values (IF and GOTO statements)

- Commands to control terminal I/O
- Other Control Characters
- Comments

A.5.2.1 **Labels** – You define labels in RT-11 mode to provide a symbolic means of referring to a specific location within a BATCH program. If present, a label must begin in the first character position, must be unique within the first six characters, and must terminate with a colon (:) and a carriage return/line feed combination.

A.5.2.2 **Variables** – A variable in RT-11 mode is a symbol representing a value that can change during program execution. The 26 variables BATCH permits in a BATCH program have the names A-Z; each variable requires one byte of physical storage. You can assign values to variables in a LET statement. You can then test these values by an IF statement to control the direction of program execution.

Assign values to variables with a LET statement of the following form:

```
(TAB) LET x="c
```

where:

x	represents a variable name in the range A-Z.
"c	indicates the ASCII value of a character.

For example:

```
(TAB) LET    A="0
```

This example indicates that the value of variable A is the 7-bit ASCII value of the character 0 (60).

The LET statement can also specify an octal value in the form:

```
(TAB) LET A=n
```

where:

n	represents an 8-bit signed octal value in the range 0-377. Positive numbers range from 0-177; negative numbers range from 200-377 (-200 to -1).

You can use variables to introduce control characters, such as ESCAPE, into a BATCH stream. For example, wherever 'A' appears in the following BATCH stream, BATCH substitutes the contents of variable A (the code for an ESCAPE):

```
$JOB/RT11
   LET   A=33
   !A  IS  AN  ESCAPE
.R EDIT
*EBFILE,MAC'A'A'
*R'A'A'
   !EDIT FILE TO CHANGE THE VERSION NUMBER TO 2
*GVERSION='A'DI2'A'
*EX'A'A'
```

Increment the value of a variable by 1 by placing a percent sign (%) before the variable. For example:

```
TAB %A
```

This command indicates that BATCH must increase the unsigned contents of variable A by 1.

Indicate with an IF statement conditional transfers of control according to the value of a variable. The IF statement has the syntax:

TAB IF(x-"c) label1, label2, label3

or

TAB IF(x-n) label1, label2, label3

where:

x	represents the variable to be tested.
"c	is the ASCII value to be compared with the contents of the variable.
n	is an octal integer in the range 0-377.
label1 label2 label3	represent the names of labels included in the BATCH stream.

When BATCH evaluates the expression (x-"c) or (x-n), the BATCH run-time handler transfers control to:

- label1 if the value of the expression is less than zero.
- label2 if the value of the expression is equal to zero.
- label3 if the value of the expression is greater than zero.

If you omit one of the labels, and the condition is met for the omitted label, control transfers to the line following the IF statement.

NOTE

Since this comparison is a signed byte comparison, 377 is considered to be -1.

The characters + and - allow you to control where BATCH begins searching for label1, label2, and label3. If you precede the label by a minus sign (-), BATCH starts the label search just after the $JOB command. If a plus sign (+) or no sign precedes the label, the label search starts after the IF statement. For example:

```
(TAB)IF(B-"9) -LOOP, LOOP1,
```

This statement transfers program control to the label LOOP following the $JOB command if the contents of variable B are less than the ASCII value of 9. It transfers control to the label LOOP1 following the IF statement if B is equal to ASCII 9. If the contents of variable B are greater than the ASCII value of 9, program control goes to the next BATCH statement in sequence.

The GOTO statement unconditionally transfers program control to a label you specify as the argument of the statement. You can use one of the following three forms of this statement:

(TAB) GOTO label	transfers control to the first occurrence of label that appears after this GOTO statement in the BATCH stream.
(TAB) GOTO +label	same as GOTO label.
(TAB) GOTO -label	transfers control to the first occurrence of label that appears after the $JOB command.

The following GOTO statement transfers control unconditionally to the next label LOOP if such a label appears in the BATCH stream following the GOTO statement.

```
(TAB)GOTO   LOOP
```

NOTE

If BATCH cannot find a label (for example, if you unintentionally omit a minus sign) the BATCH handler searches until it reaches the end of the .CTL file and ends the job.

A.5.2.3 Terminal I/O Control – You can issue commands directly to the BATCH run-time handler to control logging console terminal input and output. If you do not enter any of the following commands, BATCH assumes TTYOUT.

(TAB) NOTTY	does not write terminal input and output to the log file. Comments to the log are still logged.
(TAB) TTYIN	writes only terminal input to the log file.
(TAB) TTYIO	writes terminal input and output to the log file. (You should enter this command if you are using RT-11 mode so that RT-11 mode commands go to the log file.)
(TAB) TTYOUT	writes only terminal output to the log file (default).

A.5.2.4 Other Control Characters – The system permits other control characters in an RT-11 mode command that begins with a period (.) or an asterisk (*). Following are these control characters and their meanings:

'text' command to BATCH run-time handler, where text can be one of the following:

CTY	accepts input from the console terminal; notifies the operator that action is required by ringing a bell and printing a question mark (?).
FF	outputs the current log buffer.

NL	inserts a new line (line feed) in the BATCH stream.
x	inserts the contents of a variable where x is an alphanumeric variable in the range A through Z. It indicates that BATCH should insert the contents of the variable as an ASCII character at this place in the command string.
"message"	directs the message to the console terminal.

The following commands allow the operator to enter the name of a MACRO program to be assembled. The BATCH stream contains:

```
$JOB/RT11
.R MACRO
*'"ENTER MACRO COMMAND STRING"''CTY'
```

The operator receives the following message at the terminal and types a response, followed by carriage return; BATCH processing continues.

```
ENTER MACRO COMMAND STRING
?FILE,FILE=FILE
```

To run the same BATCH file on several systems with different configurations you need to assign a device dynamically. The following RT-11 mode command lets you request that the listing device name be entered by the operator.

```
.ASSIGN '"PLEASE TYPE LST DEVICE NAME"''CTY' LST
```

The operator receives the message and responds with the device to be used as the listing device (DT2:).

```
PLEASE TYPE LST DEVICE NAME
?DT2:
```

A.5.2.5 **Comments** – You can include comments in RT-11 mode as separate comment statements. Include comments by typing a separator followed by a ! and the comment. For example:

```
(TAB)!OPERATOR ACTION IS REQUESTED IN THIS JOB. BE PREPARED.
```

A.5.3 RT-11 Mode Examples

The following are examples of BATCH programs using the RT-11 mode.

This BATCH program assembles, lists, and maps 10 programs with only 12 BATCH commands.

```
$JOB/RT11     !ASSEMBLE, LIST, MAP PROG0 to PROG9
   TTYIO
   !WRITE TERMINAL I/O TO THE LOG FILE
   LET N="0
   !START AT FILE PROG0
LOOP:
.R MACRO
*PROG'N',LOG:/C=PROG'N'/N:TTM
.R LINK
*,LOG:=PROG'N'
   %N
   !INCREMENT VARIABLE N
   IF(N-"9)-LOOP,-LOOP,END
   !TEST FOR END
END:
$EOJ
```

The following program lets you set up a master control stream to run several BATCH jobs with one call to BATCH. First set up a BATCH job (INIT.BAT) that performs a $CHAIN to the master control stream:

```
$JOB/RT11
   LET I='0
   !INITIALIZE INDEX
$CHAIN MASTER          !GO TO MASTER
$EOJ
```

The following is the master control stream (MASTER.BAT) to which INIT chains.

```
$JOB/RT11              !MASTER CONTROL STREAM
   %I
   !BUMP INDEX BY 1
   IF(I-'7),,END
.R BATCH
   !THIS IS A $CHAIN
*JOB'I'
   !RUNS JOB1-JOB7
END:
$MESSAGE END OF BATCH RUN
$EOJ
```

Each job that MASTER.BAT runs must contain the following:

```
$JOB
   !BATCH COMMANDS
$CHAIN MASTER
$EOJ
```

Activate the master control stream by calling BATCH as follows:

```
.R BATCH
*INIT
```

A.6 CREATING BATCH PROGRAMS ON PUNCHED CARDS

To create a BATCH program on punched cards, punch into the cards the commands described in Section A.4. Each command line occupies a single punched card. Only one card, the EOF card, is different from the standard BATCH commands. The EOF (end-of-file) card terminates the list of jobs from the card reader.

To create the EOF card, hold the MULT PCH key on the keypunch keyboard while typing the following characters:

```
- & 0 1 6 7 8 9
```

This procedure produces an EOF card with holes punched in the first column (see Figure A-1).

To run multiple jobs from the card reader, simply combine the jobs into a single card deck. Ensure that each job has its own $JOB and $EOJ card. Then follow the last $EOJ card with two EOF cards.

Although in general, you terminate BATCH jobs on cards by placing two EOF cards after the last $EOJ card, some card readers require that you type \F followed by a carriage return. Put two EOF cards and a blank card in the reader and ensure that the card reader is ready. Note that a small card deck (less than 512 characters) can require more than two EOF cards to terminate the deck.

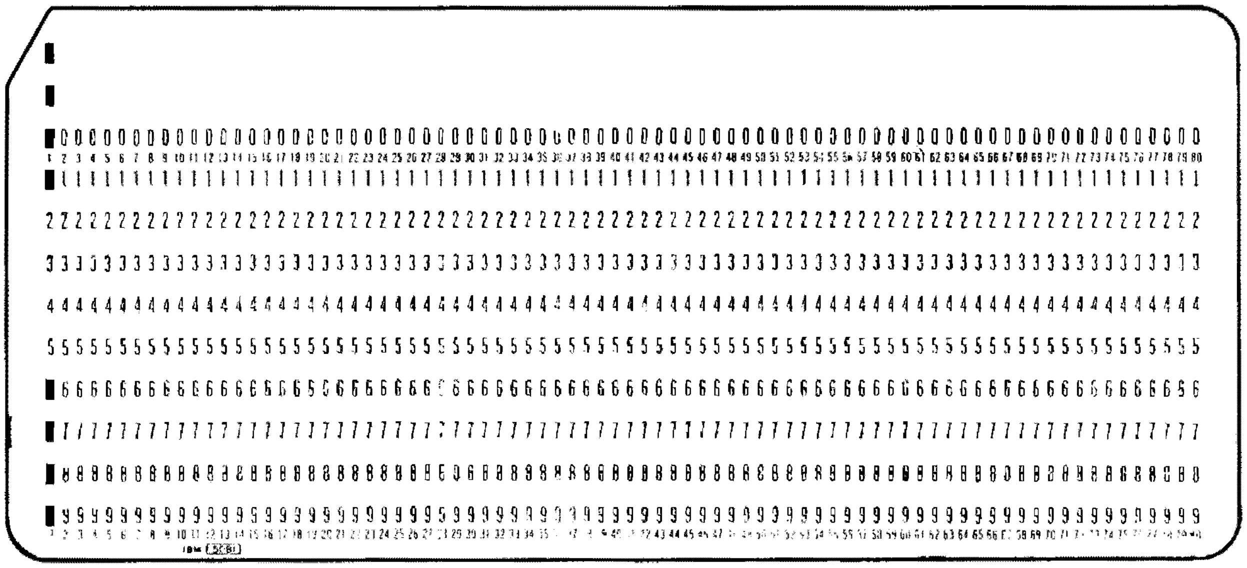


Figure A-1 EOF Card

A.7 OPERATING PROCEDURES

A.7.1 Loading BATCH

After you bootstrap the RT-11 system and enter the date and time, you must make the BATCH run-time handler resident by typing the RT-11 LOAD command as follows:

```
.LOAD BA:
```

You detach and unload the BATCH run-time handler with the /U option in the BATCH compiler command line (see Section A.7.2).

NOTE

If BATCH crashes, you must unload BATCH with the UNLOAD command and then reload BATCH with the LOAD command. This ensures that the BATCH handler is properly initialized when you rerun BATCH.

You must make the BATCH log device resident unless the log device is SY:, or unless it is a device for which the handler is already resident. Load the log device by typing:

```
.LOAD log
```

where:

log	represents the device to which BATCH must write the log file.

For example:

```
.LOAD LP:
```

You can, of course, load device handlers with a single LOAD command. For example:

```
.LOAD BA:,LP:
```

You must then assign the logical device name LOG to the log device. Use the RT-11 monitor ASSIGN command in the form:

```
.ASSIGN log LOG
```

For example, if LP: is the log device, type:

```
.ASSIGN LP: LOG
```

Then assign the logical device name LST: using the RT-11 ASSIGN command in the form:

```
.ASSIGN list-device LST
```

where:

list-device	represents the physical device BATCH must use for listings.

If, for example, you want to produce listings on the line printer, type:

```
.ASSIGN LP: LST
```

NOTE

Do not use the DEASSIGN command with no arguments in a BATCH program since it deassigns the log and list devices, possibly causing the BATCH job to terminate.

You must also make resident the BATCH run-time handler input device (compiler output device). If this device is already resident or is SY:, you do not need to load it. For example, to load the DECtape handler as the input device, type:

```
.LOAD DT:
```

If the input file to the BATCH compiler is on cards, load the card reader handler by typing:

```
.LOAD CR:
```

NOTE

If input is on cards, you must use the RT-11 monitor SET command (before loading the handler) to specify CRLF and NOIMAGE modes. That is, the following command appends a carriage return/line feed combination to each card image.

```
.SET CR: CRLF
```

The following command translates the card by packing card code into ASCII data, one column per byte.

```
.SET CR: NOIMAGE
```

If card images do not properly translate to ASCII, you may have to change the card translation codes by using one of the following commands:

```
.SET CR: CODE=29
```

or

```
.SET CR: CODE=26
```

See Section 4.4.

A.7.2 Running BATCH

When you have loaded all necessary handlers, run the BATCH compiler as follows:

```
.R BATCH
```

BATCH responds by printing an asterisk (*) to indicate its readiness to accept commands. In response to the *, type the output file specifications for the control file followed by an equal sign. Then type the input file specifications for the BATCH file as follows:

[[output-filespec] [,log-filespec] [/option . . .] =] input-filespec [. . . , input-filespec] [/option . . .]

where:

output-filespec	is the BATCH compiler output device and file the BATCH run-time handler must use. The device you specify must be random-access. Your BATCH job should not delete or move this file. Your BATCH job should avoid compressing the system volume with the SQUEEZE command or the DUP /S option. If you omit the output-filespec, BATCH generates a file on the default device DK: with the same name as the first input file but with a .CTL file type. If you do not specify a file type in the output-filespec, BATCH assumes .CTL.
log-filespec	is the log file created by the BATCH run-time handler. If you do not specify a log device, BATCH assumes LOG:. The device name you specify for log-filespec must be the same as you assign to LOG:.

You can change the size of a log file on a file-structured device from the default size of 64 (decimal) blocks. To make this change, enclose the required size in square brackets. For example:

```
*,FILE.LOG[10]=FILE
```

The default file type for the log-filespec is .LOG.

input-filespec	represents an input file. If you do not specify a file type, BATCH assumes .BAT. If you specify a .CTL file, BATCH assumes a precompiled file that must be the only file in the input list.
/option	is an option from the following list:

/N	compiles but does not execute. This option creates a BATCH control file (.CTL), generates an ABORT JOB message at the beginning of the log file, and returns to the RT-11 monitor.
/T:n	if n=0, sets the /NOTIME option as the default on the $JOB command. If n=1, the default option on the $JOB command is /TIME.
/U	indicates that the BATCH compiler must detach the BATCH run-time handler from the RT-11 monitor and unload the handler.

NOTE

You need not specify the RT-11 monitor UNLOAD BA command to actually remove the handler. Specifying /U to BATCH causes the handler to detach and unload.

/X	indicates that the input is a precompiled BATCH program. Use this option when you do not specify the .CTL file type.
(RET)	prints the version number of the BATCH compiler.

The following example calls BATCH to compile and execute the three input files (PROG1.BAT, PROG2.BAT, PROG3.BAT) to generate on DK: the compiler output files, and to generate on LOG: a log file.

```
.R BATCH
*PROG1.BAT,PROG2.BAT,PROG3.BAT
```

The following commands print the version number of BATCH, then compile and run SYBILD.BAT.

```
.R BATCH
*(RET)
BATCH V03.02
*SYBILD
```

The following commands compile PROTO.BAT to create PROTO.CTL but do not run the compiled program.

```
.R BATCH
*PROTO/N
```

Type the following commands to unlink BA.SYS from the monitor and to unload it.

```
.R BATCH
*/U
```

The following commands compile FILE.BAT from magtape to create FILE.CTL on RK1:. They execute the compiled file and create a log file named FILE.LOG (of size 20) on LOG:.

```
.R BATCH
*RK1:FILE,FILE[20]=MT:FILE
```

The following commands execute a precompiled job called FILE.TST.

```
.R BATCH
*FILE.TST/X
```

The following commands execute a precompiled job called FILE.CTL.

```
.R BATCH
*FILE/X
```

The following commands accept input from the card reader to create a file called TEMP.CTL. BATCH stores this file on DK: and executes it.

```
.R BATCH
*CR:
```

The following commands accept input from the card reader to create a file called JOB.CTL. BATCH stores the file on DK: and executes it.

```
.R BATCH
*JOB=CR:
```

A.7.3 Communicating with BATCH Jobs

During the execution of a BATCH stream, BATCH can request the operator to service a peripheral device, to provide information, or to insert a command line into the BATCH stream. The operator does this by typing directives to the BATCH handler on the console terminal.

NOTE

These directives are equivalent to the compiler output that BATCH generates in the .CTL file. The .CTL file is an ASCII file that you can list by using the PRINT or TYPE commands or by running PIP.

These directives have the form:

\dir

where:

dir	represents one of the directives listed in Table A-6.

To use these directives, the operator must get control of the BATCH run-time handler by typing a carriage return on the console terminal. When BATCH executes a command, it acknowledges the carriage return and prints a carriage return/line feed combination at the terminal. The operator can then enter a directive from Table A-6. The most useful directives are marked with an asterisk (*).

Table A-6 Operator Directives to BATCH Run-Time Handler

Directive	Explanation
*\A	Changes the input source to be the console terminal.
*\B	Changes the input source to be the BATCH stream.
*\C	Sends the following characters to the log device.
*\D	Considers the following characters as user data.
*\E	Sends the following characters to the RT-11 monitor.
*\F	Forces the output of the current log block. If this directive is followed by any characters other than another BATCH backslash (\) directive, the BATCH job prints an error message and terminates. BATCH then returns control to the RT-11 monitor.
\Hn	Is the help function to change the logging mode; n specifies the following: 0 logs only .TTYOUT and .PRINT. 1 logs .TTYOUT, .PRINT, and .TTYIN 2 does not log .TTYOUT, .PRINT, and .TTYIN 3 logs only .TTYIN

In this example, the operator must interrupt the BATCH handler to enter information from the console. As a result of a /WAIT or 'CTY' in the BATCH stream, the following message appears at the terminal:

```
    $MESSAGE/WAIT WRITE NECESSARY FILES TO DISK
```

To divert BATCH stream input from the current file to the console terminal, the operator types a \E, enters commands to the RT-11 monitor, then types a \B. Control then returns to the BATCH stream. The following example illustrates this procedure.

```
.R BATCH
*NEXT
 WRITE NECESSARY FILES TO DISK
?\A\E

\ECOPY DT1:FILE.MAC RK:

 FILES COPIED:
DT1:FILE.MAC   TO RK:FILE.MAC

\E\F\B



END BATCH
```

The following BATCH program lets you make frequent edits to a file and list only the edits. First, create a BATCH program that assembles with a listing and then link the file. This BATCH program, called COMPIL.BAT, contains:

```
$JOB/RT11
   TTYIO
   !WRITE TERMINAL I/O TO LOG FILE
.R MACRO
   !CALL THE MACRO ASSEMBLER
*FILE,FILE/C=FILE
$MESSAGE/WAIT OK TO TYPE EDIT COMMANDS
.R LINK
   !CALL THE RT-11 LINKER
*FILE,LOG:=FILE
$EOJ
```

At run-time, you can insert commands into the BATCH stream from the console terminal. These commands search for the section of the listing file that has been edited then lists this section to the log. You must insert the command after the R MACRO command but before the R LINK command. The following example illustrates this procedure.

```
.R BATCH
*COMPIL
 OK TO TYPE EDIT COMMANDS
?\A\E

\ER EDIT

*ERFILE.LST$$
*EWFILE.SEC$$
*PRETRY:$=J$$
*\L$$
RETRY:  0                                 ;HIGH ORDER BIT USED FOR 'RESET IN PROGRESS FLAG
     49 000020  016705  177764                            MOV     RKCQE,R5           ;GET Q P
     50 000024  011502                                    MOV     @R5,R2             ;R2 = BL
     51 000026  016504  000002                            MOV     2(R5),R4           ;R4 = UN
     52 000032  006204                                    ASR     R4                 ;ISOLATE
     53 000034  006204                                    ASR     R4
     54 000036  006204                                    ASR     R4
     55 000040  000304                                    SWAB    R4
     56 000042  042704  017777                    BIC     #^C<160000>,R4
     57 000046  000404                            BR      2$              ;ENTER C
*EX$$

\E\C\B

END BATCH
```

A.7.4 Terminating BATCH

When BATCH terminates normally, it prints the following message and returns control to the RT-11 monitor:

```
END BATCH
```

To abort BATCH while it is executing a BATCH stream, interrupt the BATCH handler by typing a carriage return. When BATCH executes the next command after the carriage return, it prints a carriage return/line feed combination at the console terminal. You then gain control of the system. Type \F followed by a carriage return. The BATCH handler responds with the FE (forced exit) error message and writes the remainder of the log buffer. Control returns to the RT-11 monitor.

Typing two CTRL/Cs interrupts and terminates BATCH immediately. Use two CTRL/Cs when BATCH is in a loop or when a long assembly is running. In these cases, BATCH does not respond promptly (or at all) to your carriage return interrupt.

A.8 DIFFERENCES BETWEEN RT-11 BATCH AND RSX-11D BATCH

Some programmers run their RT-11 BATCH programs under RSX-11D. Note the differences between the two BATCH implementations listed in Table A-7. BATCH programs that run under both systems must be compatible with both RT-11 and RSX-11D BATCH.

Table A-7 Differences Between RT-11 and RSX-11D BATCH

Characteristic	RT-11	RSX-11D
File descriptors	filespec/option	SY:filnam.typ/option
Default listing file type	.LST(or .LIS)	.LIS
Executable file type	.SAV	.EXE
Incompatible commands	$BASIC $CALL $CHAIN $LIBRARY $RT11 $SEQUENCE	$MCR
Incompatible options	$COPY/DELETE $CREATE/DOLLARS $CREATE/LIST $DATA/DOLLARS $DATA/LIST $DIR file/LIST $DISMOUNT/WAIT $DISMOUNT lun:/LOGICAL $FORTRAN/DOLLARS $FORTRAN/MAP	 $DIR file/DIRECTORY

(Continued on next page)

Table A-7 (Cont.) Differences Between RT-11 and RSX-11D BATCH

Characteristic	RT-11	RSX-11D
Incompatible options (Cont.)	$JOB/BANNER $JOB/LIST $JOB/RT11 $JOB/TIME $JOB/UNIQUE $LINK/LIBRARY $LINK/OBJECT $MACRO/CREF $MACRO/DOLLARS $MACRO/LIBRARY $MACRO/MAP $MESSAGE/WAIT $MESSAGE/WRITE $PRINT/DELETE	$JOB/NAME $JOB/LIMIT $JOB/MCR $LINK/MCR
$DATA input	appears as if from input	appears as if from a file named FOR001.DAT
Logical device names	in $MOUNT and $DISMOUNT	logical unit numbers only
$RUN	you must specify file name	RSX11DBAT.EXE is default

APPENDIX B
MONITOR COMMAND ABBREVIATIONS AND SYSTEM PROGRAM EQUIVALENTS

This appendix provides a table of correspondence (Table B-1) between the keyboard monitor commands with their options and the system utility programs with their options. Remember that the syntax you use to issue a keyboard monitor command is different from the syntax that the Command String Interpreter requires for input and output specifications for the system utility programs. Bear in mind that there are many differences between issuing a monitor command and running a utility program. Table B-1 lists all the keyboard monitor commands and options. A dash under the corresponding system program or option column indicates that the command has no real system program equivalent, that the function is inherent in the keyboard monitor, or that the function is the default mode of operation. The minimum abbreviation for each command and option is underlined.

Table B-1 Monitor Command/System Program Equivalents

Monitor Command	Option	System Utility Program	Option
APL		R APL	–
ASSIGN		–	–
B		–	–
BASIC		R BASIC	–
BOOT		DUP	/O
CLOSE		–	–
COMPILE		–	–
	/ALLOCATE:size	–	[n]
	/ALPHABETIZE	DIBOL	/A
	/CODE:type	FORTRAN	/I
	/CROSSREFERENCE[:type[. . . :type]]	MACRO, DIBOL	/C
	/DIAGNOSE	FORTRAN	/B
	/DIBOL	–	–
	/DISABLE:value[. . . :value]	MACRO	/D
	/ENABLE:value[. . . :value]	MACRO	/E
	/EXTEND	FORTRAN	/E
	/FORTRAN	–	–
	/HEADER	FORTRAN	/O
	/I4	FORTRAN	/T
	/LIBRARY	MACRO	/M
	/LINENUMBERS	–	–
	/NOLINENUMBERS	DIBOL,	/O
		FORTRAN	/S
	/LIST[:filespec]	–	–
	/MACRO	–	–
	/OBJECT[:filespec]	–	–
	/NOOBJECT	–	–
	/ONDEBUG	DIBOL, FORTRAN	/D
	/OPTIMIZE[:type]	FORTRAN	/P
	/NOOPTIMIZE[:type]	FORTRAN	/M
	/PASS:1	MACRO	/P

(Continued on next page)

Table B-1 Monitor Command/System Program Equivalents (Cont.)

Monitor Command	Option	System Utility Program	Option
	/RECORD:length	FORTRAN	/R
	/SHOW:value	FORTRAN, MACRO	/L
	/NOSHOW:value	MACRO	/N
	/STATISTICS	FORTRAN	/A
	/SWAP	–	–
	/NOSWAP	FORTRAN	/U
	/UNITS:n	FORTRAN	/N
	/VECTORS	–	–
	/NOVECTORS	FORTRAN	/V
	/WARNINGS	FORTRAN	/W
	/NOWARNINGS	DIBOL	/W
COPY		–	–
	/ALLOCATE:size	–	[n]
	/ASCII	PIP	/A
	/BINARY	PIP	/B
	/BOOT	DUP	/U
	/CONCATENATE	PIP	/U
	/DEVICE	DUP	/I
	/DOS	FILEX	/S
	/EXCLUDE	PIP	/P
	/IGNORE	PIP	/G
	/IMAGE	–	–
	/INTERCHANGE[:size]	FILEX	/U
	/LOG	PIP	/W
	/NOLOG	–	–
	/NEWFILES	PIP	/C
	/OWNER:[nnn, nnn]	FILEX	UIC
	/PACKED	FILEX	/P
	/POSITION:n	PIP	/M
	/PREDELETE	PIP	/O
	/QUERY	PIP	/Q
	/NOQUERY	–	–
	/REPLACE	–	–
	/NOREPLACE	PIP	/N
	/SETDATE	PIP	/T
	/SLOWLY	PIP	/S
	/SYSTEM	PIP	/Y
	/TOPS	FILEX	/T
D		–	–
DATE		–	–
DEASSIGN		–	–
DELETE		–	–
	/DOS	FILEX	/S
	/EXCLUDE	PIP	/P
	/INTERCHANGE	FILEX	/U
	/LOG	PIP	/W

(Continued on next page)

Table B-1 Monitor Command/System Program Equivalents (Cont.)

Monitor Command	Option	System Utility Program	Option
	/NEWFILES	PIP	/C
	/POSITION:n	PIP	/M
	/QUERY	PIP	/Q
	/NOQUERY	–	–
	/SYSTEM	PIP	/Y
DIBOL		R DIBOL	–
	/ALLOCATE:size	–	[n]
	/ALPHABETIZE	DIBOL	/A
	/CROSSREFERENCE	DIBOL	/C
	/LINENUMBERS	–	–
	/NOLINENUMBERS	DIBOL	/O
	/LIST[:filespec]	–	–
	/OBJECT[:filespec]	–	–
	/NOOBJECT	–	–
	/ONDEBUG	DIBOL	/D
	/WARNINGS	–	–
	/NOWARNINGS	DIBOL	/W
DIFFERENCES		R SRCCOM	–
	/ALLOCATE:size	–	[n]
	/BLANKLINES	SRCCOM	/B
	/COMMENTS	–	–
	/NOCOMMENTS	SRCCOM	/C
	/FORMFEED	SRCCOM	/F
	/MATCH:n	SRCCOM	/L
	/OUTPUT:filespec	–	–
	/PRINTER	–	–
	/SPACES	–	–
	/NOSPACES	SRCCOM	/S
	/TERMINAL	–	–
DIRECTORY		R DIR	–
	/ALLOCATE:size	–	[n]
	/ALPHABETIZE	DIR	/A
	/BADBLOCKS	DUP	/K
	/BEFORE[date]	DIR	/K
	/BEGIN	DIR	/G
	/BLOCKS	DIR	/B
	/BRIEF	DIR, FILEX	/F
	/COLUMNS:n	DIR	/C
	/DATE[date]	DIR	/D
	/DELETED	DIR	/Q
	/DOS	FILEX	/S
	/EXCLUDE	DIR	/P
	/FAST	DIR, FILEX	/F
	/FILES	DUP	/F
	/FREE	DIR	/M
	/FULL	DIR	/E
	/INTERCHANGE	FILEX	/U

(Continued on next page)

Table B-1 Monitor Command/System Program Equivalents (Cont.)

Monitor Command	Option	System Utility Program	Option
	/NEWFILES	DIR	/D
	/OCTAL	DIR	/O
	/ORDER[:category]	DIR	/S
	/OUTPUT:filespec	–	–
	/OWNER: [nnn, nnn]	FILEX	UIC
	/POSITION	DIR	/B
	/PRINTER	–	–
	/REVERSE	DIR	/R
	/SINCE[date]	DIR	/J
	/SORT[:category]	DIR	/S
	/SUMMARY	DIR	/N
	/TERMINAL	–	–
	/TOPS	FILEX	/T
	/VOLUMEID	DUP	/V
DUMP		R DUMP	–
	/ALLOCATE:size	–	[n]
	/ASCII	–	–
	/NOASCII	DUMP	/N
	/BYTES	DUMP	/B
	/END:block	DUMP	/E
	/IGNORE	DUMP	/G
	/ONLY:block	DUMP	/O
	/OUTPUT:filespec	–	–
	/PRINTER	–	–
	/RAD50	DUMP	/X
	/START:block	DUMP	/S
	/TERMINAL	–	–
	/WORDS	DUMP	/W
E		–	–
EDIT		EDIT	EB
	/ALLOCATE:size	–	[n]
	/CREATE	EDIT	EW
	/INSPECT	EDIT	ER
	/OUTPUT:filespec	EDIT	EW
EXECUTE		–	–
	/ALLOCATE:size	–	[n]
	/ALPHABETIZE	DIBOL	/A
	/BOTTOM:n	LINK	/B
	/CODE:type	FORTRAN	/I
	/CROSSREFERENCE[:type[. . . :type]]	DIBOL, MACRO	/C
	/DEBUG[:filespec]	–	–
	/DIAGNOSE	FORTRAN	/B
	/DIBOL	–	–
	/DISABLE:value[. . . :value]	MACRO	/D
	/ENABLE:value[. . . :value]	MACRO	/E
	/EXECUTE[:filespec]	–	–
	/EXTEND	FORTRAN	/E

(Continued on next page)

Table B-1 Monitor Command/System Program Equivalents (Cont.)

Monitor Command	Option	System Utility Program	Option
	/FORTRAN	–	–
	/HEADER	FORTRAN	/O
	/I4	FORTRAN	/T
	/LIBRARY	MACRO	/M
	/LINENUMBERS	–	–
	/NOLINENUMBERS	DIBOL,	/O
		FORTRAN	/S
	/LINKLIBRARY:filespec	–	–
	/LIST[:filespec]	–	–
	/MACRO	–	–
	/MAP[:filespec]	–	–
	/OBJECT[:filespec]	–	–
	/ONDEBUG	DIBOL, FORTRAN	/D
	/OPTIMIZE:type	FORTRAN	/P
	/NOOPTIMIZE:type	FORTRAN	/M
	/PASS:1	MACRO	/P
	/RECORD:length	FORTRAN	/R
	/RUN	–	–
	/NORUN	–	–
	/SHOW[:value]	FORTRAN, MACRO	/L
	/NOSHOW:value	MACRO	/N
	/STATISTICS	FORTRAN	/A
	/SWAP	–	–
	/NOSWAP	FORTRAN	/U
	/UNITS:n	FORTRAN	/N
	/VECTORS	–	–
	/NOVECTORS	FORTRAN	/V
	/WARNINGS	FORTRAN	/W
	/NOWARNINGS	DIBOL	/W
	/WIDE	LINK	/W
FOCAL		R FOCAL	–
FORTRAN		R FORTRAN	–
	/ALLOCATE:size	–	[n]
	/CODE:type	FORTRAN	/I
	/DIAGNOSE	FORTRAN	/B
	/EXTEND	FORTRAN	/E
	/HEADER	FORTRAN	/O
	/I4	FORTRAN	/T
	/LINENUMBERS	–	–
	/NOLINENUMBERS	FORTRAN	/S
	/LIST[:filespec]	–	–
	/OBJECT[:filespec]	–	–
	/NOOBJECT	–	–
	/ONDEBUG	FORTRAN	/D
	/OPTIMIZE:type	FORTRAN	/P
	/NOOPTIMIZE:type	FORTRAN	/M
	/RECORD:length	FORTRAN	/R
	/SHOW[:value]	FORTRAN	/L

(Continued on next page)

Table B-1 Monitor Command/System Program Equivalents (Cont.)

Monitor Command	Option	System Utility Program	Option
	/STATISTICS	FORTRAN	/A
	/SWAP	–	–
	/NOSWAP	FORTRAN	/U
	/UNITS:n	FORTRAN	/N
	/VECTORS	–	–
	/NOVECTORS	FORTRAN	/V
	/WARNINGS	FORTRAN	/W
FRUN		–	–
	/N:n	–	–
	/P	–	–
	/T:n	–	–
GET		–	–
GT OFF		–	–
GT ON		–	–
	/L:n	–	–
	/T:n	–	–
HELP		–	–
	/PRINTER	–	–
	/TERMINAL	–	–
INITIALIZE		–	–
	/BADBLOCKS	DUP	/B
	/DOS	FILEX	/S
	/FILE:filespec	–	–
	/INTERCHANGE	FILEX	/U
	/QUERY	–	–
	/NOQUERY	DUP, FILEX	/Y
	/REPLACE[:RETAIN]	DUP	/R
	/SEGMENTS:n	DUP	/N
	/VOLUMEID[:ONLY]	DUP	/V
INSTALL		–	–
LIBRARY		R LIBR	–
	/ALLOCATE:size	–	[n]
	/CREATE	–	–
	/DELETE	LIBR	/D
	/EXTRACT	LIBR	/E
	/INSERT	–	–
	/LIST[:filespec]	–	–
	/MACRO	LIBR	/M
	/OBJECT[:filespec]	–	–
	/NOOBJECT	–	–
	/PROMPT	LIBR	//
	/REMOVE	LIBR	/G
	/REPLACE	LIBR	/R
	/UPDATE	LIBR	/U
LINK		R LINK	–
	/ALLOCATE:size	–	[n]
	/BOTTOM:n	LINK	/B

(Continued on next page)

Table B-1 Monitor Command/System Program Equivalents (Cont.)

Monitor Command	Option	System Utility Program	Option
	/BOUNDARY:value	LINK	/Y
	/DEBUG[:filespec]	–	–
	/EXECUTE[:filespec]	–	–
	/NOEXECUTE	–	–
	/EXTEND:n	LINK	/E
	/FILL:n	LINK	/Z
	/FOREGROUND[:stacksize]	LINK	/R
	/INCLUDE	LINK	/I
	/LDA	LINK	/L
	/LIBRARY:filespec	–	–
	/LINKLIBRARY:filespec	–	–
	/MAP[:filespec]	–	–
	/PROMPT	LINK	//
	/ROUND:n	LINK	/U
	/RUN	–	–
	/SLOWLY	LINK	/S
	/STACK[:n]	LINK	/M
	/TRANSFER[:n]	LINK	/T
	/WIDE	LINK	/W
LOAD		–	–
MACRO		R MACRO	–
	/ALLOCATE:size	–	[n]
	/CROSSREFERENCE[:type[. . . :type]]	MACRO	/C
	/DISABLE:value[. . . :value]	MACRO	/D
	/ENABLE:value[. . . :value]	MACRO	/E
	/LIBRARY	MACRO	/M
	/LIST[:filespec]	–	–
	/OBJECT[:filespec]	–	–
	/NOOBJECT	–	–
	/PASS:1	MACRO	/P
	/SHOW:value	MACRO	/L
	/NOSHOW:value	MACRO	/N
PRINT		–	–
	/COPIES:n	PIP	/K
	/DELETE	PIP	/D
	/LOG	PIP	/W
	/NOLOG	–	–
	/NEWFILES	PIP	/C
	/QUERY	PIP	/Q
R		–	–
REENTER		–	–
REMOVE		–	–
RENAME		–	–
	/LOG	PIP	/W
	/NOLOG	–	–
	/NEWFILES	PIP	/C
	/QUERY	PIP	/Q

(Continued on next page)

Table B-1 Monitor Command/System Program Equivalents (Cont.)

Monitor Command	Option	System Utility Program	Option
	/REPLACE	–	–
	/NOREPLACE	PIP	/N
	/SETDATE	PIP	/T
	/SYSTEM	PIP	/Y
RESET		–	–
RESUME		–	–
RUN		–	–
SAVE		–	–
SET		–	–
SHOW		–	–
SQUEEZE		–	–
	/OUTPUT:filespec	–	–
	/QUERY	–	–
	/NOQUERY	DUP	/Y
START		–	–
SUSPEND		–	–
TIME		–	–
TYPE		–	–
	/COPIES:n	PIP	/K
	/DELETE	PIP	/D
	/LOG	PIP	/W
	/NOLOG	–	–
	/NEWFILES	PIP	/C
	/QUERY	PIP	/Q
UNLOAD		–	–

INDEX

INDEX (Cont.)

INDEX (Cont.)

INDEX (Cont.)

INDEX (Cont.)

INDEX (Cont.)

INDEX (Cont.)

INDEX (Cont.)

INDEX (Cont.)

READER'S COMMENTS

NOTE: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. Problems with software should be reported on a Software Performance Report (SPR) form. If you require a written reply and are eligible to receive one under SPR service, submit your comments on an SPR form.

Did you find errors in this manual? If so, specify by page.

Did you find this manual understandable, usable, and well-organized? Please make suggestions for improvement.

Is there sufficient documentation on associated system programs required for use of the software described in this manual? If not, what material is missing and where should it be placed?

Please indicate the type of user/reader that you most nearly represent.

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Non-programmer interested in computer concepts and capabilities

Name ______________________ Date ______________________

Organization ______________________

Street ______________________

City ______________ State ______________ Zip Code or Country ______________

Please cut along this line.

Fold Here

Do Not Tear - Fold Here and Staple

FIRST CLASS
PERMIT NO. 33
MAYNARD, MASS.

BUSINESS REPLY MAIL
NO POSTAGE STAMP NECESSARY IF MAILED IN THE UNITED STATES

Postage will be paid by:

digital

Software Documentation
146 Main Street ML 5-5/E39
Maynard, Massachusetts 01754

digital

digital equipment corporation

Printed in U.S.A.