DISCLAIMER

Although each program has been tested by its contributor, no warranty, express or implied, is made by the contributor or SHARE, as to the accuracy and functioning of the program and related program material, nor shall the fact of distribution constitute any such warranty, and no responsibility is assumed by the contributor or SHARE, in connection therewith.

A1 BC MPDV 1/3 7090 appl...

IDENTIFICATION:

A1 BC MPDV - MULTIPLE PRECISION DIVISION - FORTRAN FUNCTION
Author: John Brillhart - July, 1963
Adapted for Fortran at the Computer Center
University of California Computer Center
Berkeley 4, California

PURPOSE:

Let T = B*Q+R and [|T/B|] = Q

Given: T and B, 2 multiple precision signed words. This program performs the division of T by B returning the quotient Q and the remainder R. Both Q and R are also, in general, multiple precision words. R always has the sign of T; Q is + or - if T and B have the same or opposite signs respectively.

If B is 0, an error indicator is returned.

USAGE:

I = MPDIV(T,LT,B,LB,Q,LQ,R,LR,SUM)

1. T is the name of a FORTRAN II vector containing the dividend.
T(1) contains the least significant word, T(2) the next least significant, etc. The sign of T is taken from the most significant non zero word of the dividend. If for some reason, the various words in T are differently signed, all words in T will be set to the sign of T by MPDIV

2. LT is the length of T. Normally, T(LT) contains the most significant word of T. If T(LT) is 0, LT is reduced, internally only, to the minimum length. LT is a FORTRAN II decrement integer.

3. B is the name of the FORTRAN II vector containing the divisor. The conventions for T apply to B.

4. LB is the length of B. LB is a FORTRAN II decrement integer.

5. Q is the name of the FORTRAN II vector where the quotient will be returned. The same conventions apply. If Q is < 0, all words in Q are negative.

AN CS MP 0N

6. LQ is the length of Q. LQ is a FORTRAN II decrement integer

   Q(LQ) ≠ 0 if Q ≠ 0.
   If Q =± 0, then LQ = 1 and Q(LQ) = ± 0

7. R is the name of the FORTRAN II vector where the remainder will be returned. The same conventions apply.

8. LR is the length of R. LR is a FORTRAN II decrement integer.

   R(LR) ≠ 0 if R ≠ 0.
   If R = ± 0, then LR = 1 and R(LR) = ± 0

9. SUM is ACL (add and carry logical) sum of the LR words in the remainder, discounting the signs in R, if R is negative.

   SUM can be tested to see if T is exactly divisible by B. A Boolean IF statement should be used.
   (The condition for exact divisibility is SUM = 0)

10. I is the value of the function, MPDIV.

    I = 0 if B ≠ 0.
    I = 1 if B = 0. (equivalent to divide check condition).

    In the latter case, Q,R, and SUM are set to +0 with LQ = LR = 1.

Storage Requirements:

T,B,Q,R must appear in dimension statements. None of these arrays can be equivalent. In general, R and Q should be as long as T.

Example: Compute greatest common divisor of two numbers, X and Y.

```
      DIMENSION X(50), Y(50), R1(50), R2(50), R3(50),
     1  GCD(50), Q(50)

      I = MPDIV (X,LX, Y,LY, Q,LQ, R1,LR1, SUM)
      IF (I) 60, 10, 60
B  10 IF (SUM) 20, 70, 20
   20 CALL MPDIV (Y,LY, R1,LR1, Q,LQ, R2,LR2, SUM)
B     IF(SUM) 30, 80, 30
   30 CALL MPDIV (R1,LR1, R2,LR2, Q,LQ, R3,LR3, SUM)
B     IF(SUM) 40, 90, 40
   40 CALL MPDIV (R2,LR2, R3,LR3, Q,LQ, R1, LR1, SUM)
B     IF(SUM) 50, 100, 50
   50 CALL MPDIV (R3,LR3, R1,LR1, Q,LQ, R2,LR2, SUM)
B     IF(SUM) 30, 80, 30
```

```
      60  DO 6S I = 1,LX
      65 GCD(I) = X(I)
         LGCD = LX
         GO TO 200

      70  DO 75 I = 1,LY
      75 GCD(I) = Y(I)
         LGCD = LY
         GO TO 200

      80  DO 85 I = 1, LR1
      85 GCD(I) = R1(I)
         LGCD = LR1
         GO TO 200

      90  DO 95 I = 1, LR2
      95 GCD(I) = R2(I)
         LGCD = LR2
         GO TO 200

     100  DO 105 I = 1, LR3
     105 GCD(I) = R3(I)
         LGCD = LR3

     200 CONTINUE
```

In the above example, if Y = 0, then MPDIV returns I = 1. The convention here is to set the GCD to X when Y = 0. (merely to simplify the example).

LGCD is the length of the greatest common divisor.

After the first call to MPDIV, the logic of the program makes division by 0 impossible. Hence MPDIV is simply called as a subroutine.

MULTI-PRECISION

Ira L. Wright, Systems Engineer
2911 Cedar Springs Road
Dallas, Texas 75219

June 12, 1964

MULTI-PRECISION

Ira L. Wright

June 12, 1964

Direct Inquiries to: Ira L. Wright
2911 Cedar Springs Road
Dallas, Texas 75219
LA 6-7651

The IBM 7040/7044 multi-precision arithmetic package is for the purpose of executing floating binary point calculations, using "N" words of core for each data element. "N" is supplied at execution time.

The first word of the data element is the binary exponent. The next N-1 words are for the mantissa.

An accumulator, multiplier-quotient, and storage register of "N" words each is supplied to the package by the user.

The instructions included will be CHS, SSP, CLA, CLS, STO, STQ, LDQ, FAD, FSB, FDP, FMP, UFA, UFS and UFM. Instructions not recognized by the package are executed by the IBM 7040/7044.

MULTI-PRECISION

A. Source Decks

DECK KEY	# CARDS	CODE col. 78-80
Sequence in 73-77		
1. Set up routine - MAP	116	SET
2. Effective y - MAP	55	EFY
3. Enter - MAP	129	INP
4. Transfer Operations - MAP	162	TRN
5. Add Operations - MAP	488	ADD
6. Multiply Operations - MAP	252	MUL
7. Divide Operation - MAP	398	DIV
8. Test Deck - Fortran	103	TST

B. Object Decks

col. 73-76

Relocatable Binary
Sequence in 77-80

1. Set up	11	SETU
2. Effective y	9	EFFY
3. Enter	22	INPR
4. Transfer Operations	20	TRAN
5. Add Operations	32	FLOA
6. Multiply Operations	18	FLOM
7. Divide Operation	25	FLOD

TAPE KEY

A. LISTINGS	1 TAPE	1 FILE

TABLE OF CONTENTS

MULTI-PRECISION

Ira L. Wright, Systems Engineer
2911 Cedar Springs Road
Dallas, Texas 75219

June 12, 1964

MULTI-PRECISION

Ira L. Wright

June 12, 1964

Direct Inquiries to: Ira L. Wright
2911 Cedar Springs Road
Dallas, Texas 75219
LA 6-7651

The IBM 7040/7044 multi-precision arithmetic package is for the purpose of executing floating binary point calculations, using "N" words of core for each data element. "N" is supplied at execution time.

The first word of the data element is the binary exponent. The next N-1 words are for the mantissa.

An accumulator, multiplier-quotient, and storage register of "N" words each is supplied to the package by the user.

The instructions included will be CHS, SSP, CLA, CLS, STO, STQ, LDQ, FAD, FSB, FDP, FMP, UFA, UFS and UFM. Instructions not recognized by the package are executed by the IBM 7040/7044.

MULTI-PRECISION

I. PROGRAM BRIEF

A. Purpose/Description

This package is a set of subroutines which will calculate a level of precision from 3 to 255. The precision level is given as a data element. There is, as far as the author knows, no other package of this type available.

Multi-precision means arithmetic calculations based on a larger number of words of memory than is allowed by the floating point hardware. Normally, any arithmetic that uses more than one word of memory is considered extended, or multi-precision.

In the precision level of MP, the form that a data item takes is one word for the floating binary exponent and MP-1 words for the binary fraction. That is, the data item is:

$$\underbrace{(.XXXXX\ldots.)_2}_{\text{MP-1 words}} \cdot 2^{\text{exponent}}$$

This package was coded to run in the 7040/7044 operation system and meets the Fortran IV subroutine standards.

B. Method (see Appendix)

C. Restrictions

A maximum precision level of 255 is the limit because of the transmit instruction used. Any executed transfer or instruction which changes the location counter of the hardware causes the interpretation to be stopped. Interpretation can be entered without resetting (SETMP).

Precision level of 2 is not allowed.

D. Accuracy

No rounding is done. Double sizes are carried to hold digits for normalizing.

E. Machine Configuration

A 7040/7044 with extended instruction set is required. Floating point is not required if precision is greater than one.

F. Program Storage Requirements

Total program package storage requirement is 1483.

1.	Set up routine (SETMP)		100
	a. Program	77	
	b. CONST Section	10	
	c. ADDR Section	13	
2.	Interpretation (INMP)		248
	a. Program	217	
	b. CONSTI Section	31	
3.	Calculate effective address (EFFCTY)		52
	a. Program	47	
	b. CONSTE Section	5	
4.	Transmit Data		218
	a. Seven operations	186	
	b. Set addresses & constants(STRNS)	28	
	c. CONSTT Section	4	
5.	Add routine		384
	a. Four operations	291	
	b. Set addresses & constants(SADD)	85	
	c. CONSTA Section	8	
6.	Multiply routine		183
	a. Two operations	131	
	b. Set addresses & constants(SMUL)	48	
	c. CONSTM Section	4	
7.	Divide routine		288
	a. Operation	193	
	b. Set addresses & constants(SDIV)	89	
	c. CONSTD Section	6	

G. Source Language is MAP

H. Execution Times in Milliseconds:

Precision	Set Up	Interpret	Transfer	Add	Mult.	Div.
5	6.533	.667	.770	2.900	4.483	14.917
6	"	"	1.026	3.185	5.650	19.350
7	"	"	1.045	3.483	7.067	30.950
8	"	"	1.053	3.600	8.703	38.342
9	"	"	1.086	4.083	10.500	56.900

I. The check out was by a sample problem as in the supplied test deck. There have been no application runs yet.

J. This program and its documentation were written by an IBM employee. It was developed for a specific purpose and submitted for general distribution to interested parties in the hope that it might prove helpful to other members of the data processing community. The program and its documentation are essentially in the author's original form. IBM serves only as the distribution agency in supplying this program. Questions concerning the use of the program should be directed to the author's attention.

Time to run - 1.5 minutes

II. DETAILED PROGRAM DESCRIPTION

A. Form of the Data

1. Each data element takes MP words of storage, where MP is the precision level 3, 4, ..., or 255.

2. Each data element is of the form:

a. 1st word is the exponent of the Base 2.

b. 2nd word thru MPth word is the fraction. These words must all have the same sign, that of the fraction.

c. That is: (2nd word -MPth word) * $2^{\text{1st word}}$.

B. A Storage Register, an Accumulator, a Multiplier-Quotient, and a Temporary Register are provided outside the package. These registers are addressed by the setup routine, CALL SETMP (REG, MP). Where MP is the precision level and REG is the Storage Register, REM+MP is the Accumulator, REG+2MP is the Multiplier-Quotient, and REG+3MP is the Temporary Register.

C. The first step in a main program is CALL SETMP(REG, MP). This routine calculates all the relative constants to MP and stores them in the "CONST" control section and calculates all the relative addresses to REG and stores them in the "ADDR" control section. Control is then transferred to subroutines STRNS, SADD, SMUL, and SDIV which set the addresses and constants in all the operational subroutines. This routine subsequently returns to the main program.

D. When the required precision, previously set up, is to be used, the operations are preceded by an enter interpretive mode subroutine (CALL INMP). The CALL INMP call sequence is immediately followed by the hardware single precision floating point instructions required to calculate the necessary answer. Some additional instructions are used to facilitate the multi-precision calculations. Therefore, one should code as for single precision using all hardware instructions, but precede the formula evaluating by the CALL INMP. The hardware instructions that are interpreted are:

1.	Non-floating	CLA	CHS
		CLS	SSP
		STO	
		LDQ	
		STQ	
		STZ	
2.	Floating	FAD	
		UFA	
		FSB	
		UFS	
		FMP	
		UFM	
		FDP	
3.	Pseudo-op	ZAC	(PXD 0,0)

E. The instructions to be interpreted are followed by TRA*+1 which causes the interpretation to be terminated. The same thing can be accomplished by any other instruction that changes the hardware location counter register. (See F below). Therefore, each instruction that "loops back" terminates interpretation.

F. Any instruction that is not in the table of interpretable instructions will be executed by the hardware instruction XEC* (Indirect Execute). WARNING: Exact understanding of hardware XEC* execution is necessary to interpret terminating results.

G. Each time the interpretive routine is entered by CALL INMP, the precision level is tested. It is possible, therefore, to change precision during execution. Any change in precision should be followed by a CALL SETMP to re-initialize before CALL INMP. The exception occurs when the precision constant in the interpretive routine is set to one or zero, no initializing necessary, or if reset to original level, no re-initializing.

H. If the data is stored for a given level of precision, it is possible to run for a lower precision without changing data or storage arrangement. Change the precision constant and re-initialize.

I. It is possible to change precision as many times as necessary for the parts of program as required. Several CALL SETMP (REG, MP) could be executed, each with a different REG and MP. Each call sequence would be followed by moving the control sections "CONST" and "ADDR" to some temporary storage.

In order to change precision, it is now necessary to store the control sections and call the set up routines (STRNS, SADD, SMUL, SDIV) required. There should be a set of registers for each precision. If all set up routines are required, the difference between this method and re-initializing with CALL SETMP(REG,MP) is negligible.

III. BLOCK DIAGRAMS

A. Block diagrams are supplied in the 7070 Autochart form in Enclosure C.

B. The label on each box is the same as the label on the 1st instruction that represents the box in the coding.

C. The chart labels are:

SE	Set up Routine
IN	Interpretation Routine
EF	Calculate Effective Y Routine
TR	Transfer Routines
FA	Floating Add and Subtract Setup Routines
AD	Floating Add Routine
FM	Floating Multiply Setup Routine
MP	Floating Multiply Routine
FD	Floating Divide Routine

D. Symbols are:

AR	Accumulator
MQ	Multiplier-Quotient
SR	Storage Register
TEM	Temporary Storage

IV. DETAILED CODING

A. Register Block

A block of four times the precision (MP) of words is supplied to the package by the setup routine (CALL SETMP(REG, MP)), called register (REG). The first location of the block is REG. The block is four registers of the form $(.XXXX...X)_2 \ *2^{exp}$ where exp is the first location of the register and $(.XXXX...X)_2$ is the next MP-1 locations.

1. REG is the first address of the functional storage register.
2. REG+MP is the first address of the functional accumulator.
3. REG*2MP is the first location of the functional multiplier-quotient.
4. REG+3MP is the first location of a temporary register.

B. Control Section "CONST" in Setup Routine

MP	Precision Constant
2P	2*MP
3P	3*MP
4P	4*MP
PM1	MP -1
PM2	MP -2
2PM1	2*MP -1
2PM2	2*MP -2
MPM2	(MP -2) 2's complement
70PM35	70*MP -35

C. Control Section "ADDR" in Setup Routine

S	Storage Register Address
A	Accumulator Address
Q	M/Q Address
T	Temporary Register Address
TPP1	T + 1
AP1	A + 1
SP1	S + 1
APP1	A + MP + 1

AP2PM1	A + 2*MP -1
AP2PM2	A + 2*MP -2
TP1	T + 1
TPPM1	T + MP -1

D. Control Section "CONSTT" in Enter Routine

Table of operations: ZACMAS, CHS, SSP, CLA, CLS, STO, FAD, UFA, FSB, UFS, FMP, UFM, FDP, STQ, STZ. These are the labels used.

E. Control Section "CONSTE" in Effective Y

Z	Temporary Storage for Address
MASTAG	Tag Mask
PXAI	Instruction format
OUTTAG	Mask to eliminate Tag
MASFLG	Flag (indirect address) mask

F. Control Section "CONSTT" in Transfer Routine

CLACON	Transmit Word ** to AC
STOCON	Transmit Word AC to **
LDQCON	Transmit Word ** to MQ
STQCON	Transmit Word MQ to **

G. Control Section "CONSTA" in Add Routine

FADCON	Transmit Word ** to SR:
	: -UNOR
ADDNOR	Normalize Switch : + NOR
ADDCNA	Transmit Word SR to AC
SHFQ	# of word to shift AC right
ADDSON	Signed one
ADDSZE	Signed zero
ADDSHF	# of words to shift AC left for normalizing

H. Control Section "CONSTM" in Multiply Routine

FMPCON	Transmit control ** to SR
MPYNOR	Normalize Switch: + NOR
	: - UNOR

MPYCON	Transmit word SR to AC
MPYSHF	# of words to shift AC left for normalizing

I. Control Section "CONSTD" in Divide Routine

FDPCON	Transmit control ** to SR
FDPZTQ	Transmit control for zero ACC to MQ
FDPCNS	Transmit control Temp to MQ
FDPREM	Remainder
FDPSZE	Signed zero
FDPSON	Signed one

J. Subroutines

1. Set up and Initialization

a. Call Sequence

CALL SET(REG, MP)
REG is the first address of a block of 4 MP words.
MP is the precision code.
"1" or "0" no interpretation.
"3" exponent word
fraction 2 words
...
"MP" exponent word
fraction MP -1 words

b. Routine sets up the "CONST" and "ADDR" control sections.

c. The routine uses the initialization routines "STRNS", "SADD", "SMUL", and "SDIV" described below to set addresses and constants in the operation subroutines.

2. Interpretation Subroutine

a. Call Sequence

CALL INMP

b. Maintains a pseudo-location counter. Symbolic location is "INMP."

c. Begins interpretation with the first instruction (word) following the call sequence and is terminated by an instruction that is executed (XEC*) by the interpretation that changes the hardware location counter. (See WARNING II.F.)

d. By interpretation, the routine uses all the operational subroutines and the effective "y" subroutine described below.

3. Operation Subroutines

"y" is the first word address of the MP block of words representing the data element.

a. Transfer routines

(1) CALL CLAMP(y)
Transfers "y" into the multi-precision accumulator (MP-AC).

(2) CALL CLSMP(y)
Transfers "-y" into the MP-AC.

(3) CALL STOMP(y)
Transfers the MP-AC into "y".

(4) CALL LDQMP(y)
Transfers "y" into the multi-precision MQ (MP-MQ).

(5) CALL STQMP(y)
Transfers the MP-MQ into "y".

(6) CALL ZACMP
Transfers zeroes into the MP-AC.

(7) CALL STZMP(y)
Transfers zeroes into "y".

b. Add routines use the ADD routine

(1) CALL FADMP(y)
The contents of "y" are added to the contents of the MP-AC.

(2) CALL FSBMP(y)
The contents of "y" are subtracted from the contents of the MP-AC.

(3) CALL UFAMP(y)
Same as CALL FADMP(y) with the result unnormalized.

(4) CALL UFSMP(y)
Same as CALL FSBMP(y) with the result unnormalized.

c. Multiply routines use the MPY routine

(1) CALL FMPMP(y)
The contents of "y" are multiplied times the contents of the MP-MQ and the normalized results stored in the MP-AC.

(2) CALL UFMMP(y)
Same as CALL EMPMP(y) with the result unnormalized.

d. Divide routine uses the DVP routine

(1) CALL FDPMP(y)
The contents of the MP-AC are divided by contents of "y". The result is stored in the MP-AC.

4. Operational Set up Routines use the "CONST" and "ADDR" Control Section

a. CALL SADD
Sets the addresses and constants required in all the add routines (see 3.b. and 6.).

b. CALL STRNS
Sets the addresses and constants required in all the transfer routines (see 3.a.).

c. CALL SDIV
Sets the addresses and constants required in the divide routine (see 3.d. and 8.).

d. CALL SMUL
Sets all the addresses and constants required in all the multiply routines (see 3.c. and 7.).

5. Effective address of y

   CALL EFFCTY
   Uses the instruction in the hardware accumulator to calculate the effective address of "y". The index of ACC and flag of ACC is considered and the index of the indirect addressed location is used. The effective address calculated is left in the address of the hardware accumulator.

6. CALL ADD
   Adds the multi-precision storage register (MP-SR) to the MP-AC. The routine is normally not used except by the add routines in 3. above. Therefore, it does not save the index registers and the normalize switch must be previously set. This routine uses control sections "CONST" and "ADDR."

7. CALL MPY
   Multiplies the MP-SR times the MP-MQ and stores the results in the MP-AC. The routine is normally used by the multiply routines in 3. above. Therefore, it does not save the index register and the normalize switch must have previously been set.

8. CALL DVP
   Divides the MP-AC by the MP-SR and stores the result in the MP-AC. The routine is normally used by the divide routine in 3. above. Therefore, it does not save the index registers.

V. PROGRAM MODIFICATION

Many loops are similar for the purpose of speed. Core can be saved by overlapping these loops, maybe 5% to 10%.

VI. SAMPLE PROBLEM

Evaluate an F(S,Q) two different ways.

Sample problem begins with a JOB card and can be run as a stacked job on the 7040/7044 operating system.

VII. ASSEMBLY LISTING (Sample Problem)

A. Program Object Decks, sample problem and sample data are all together with IBSYS control cards to run on 7040 monitor. There is no date card. Deck (composite) begins with a $JOB card.

B. Assembly list of subroutine package is supplied where the assembly was made of all decks (excluding sample problem) as one job.

APPENDIX

Method Development

I. ADD METHOD

A. MP is the precision level.

B. The data elements are A_1 , A_2 , ... , A_{MP} and B_1 , B_2 , ... , B_{MP}

C. A_2 , A_3 , ... , A_{MP} and B_2 , B_3 , ... , B_{MP} have been shifted so that the exponents A_1 and B_1 are equal.

D. Add A_2 , A_3 , ... , A_{MP} and B_2 , B_3 , ... , B_{MP} in the following manner:

$A_{MP} + B_{MP}$ into A_{MP}
$A_{MP-1} + B_{MP-1}$ + Carry into A_{MP-1}
$A_{MP-2} + B_{MP-2}$ + Carry into A_{MP-2}
...
$A_2 + B_2$ + Carry into A_2 ; overflow

1. Overflow is zero
The sign of the first non-zero fraction element is taken as the sign of the fraction and elements of opposite sign are complimented and adjusted. The exponent A_1 and fraction are adjusted where normalizing is required.

2. Overflow is not zero
The exponent A_1 and fraction are adjusted for overflow.

II. MULTIPLY METHOD

A. MP is the precision level.

B. The elements are A_1 , A_2 , ... , A_{MP} and B_1 , B_2 , ... , B_{MP}

C. Multiply A_2 , A_3 , ... , A_{MP} by B_2 , B_3 , ... , B_{MP} in the following manner:

$A_{MP} * B_{MP}$		into A_{2MP-1}
$A_{MP} * B_{MP-1}$	+ carry	into A_{2MP-2}
...		
$A_{MP} * B_2$	+ carry	into A_{MP+1}
	carry	into A_{MP}
$A_{MP-1} * B_{MP} + A_{2MP-2}$		into A_{2MP-2}
$A_{MP-1} * B_{MP-1} + A_{2MP-3}$	+ carry	into A_{2MP-3}
...		
$A_{MP-1} * B_2 + A_{MP}$	+ carry	into A_{MP}
	carry	into A_{MP-1}
...		
$A_2 * B_2 + A_3$	+ carry	into A_3
	carry	into A_2

D. The exponent is $A_1 + B_1$ into A_1 .

E. The fraction and exponent A_1 are adjusted when normalizing.

III. DIVIDE METHOD

A. MP is the precision level.

B. The elements are A_1 , A_2 , ... , A_{MP} and B_1 , B_2 , ... , B_{MP}

C. Calculate A_2 , A_3 , ... , A_{MP}/B_2 , B_3 , ... , B_{MP} in the following manner:

1. Substitute:

$C = A_2$
$D = A_3$, A_4 , ... , A_{MP}
$E = B_2$
$F = B_3$, B_4 , ... , B_{MP}

2. $(E+F)^{-1} = 1/E(1-F/E+F^2/E^2 -F^3/E^3 + \ldots)$
$(C+D)/(E+F)\ (C+D)/E(1-F/E+F^2/E^2 -F^3/E^3 + \ldots)$
$(C+D)/E=Q_1 +R_1/E$
$X_1=(R_1 -Q_1 F)/E$
$(C+D)/(E+F)=Q_1 +X_1-(F/E)X_1+ (F/E)^2 X_1-(F/E)^3 + \ldots$

$X_1=(R_1 - Q_1 F)/E=Q_1 +R_1 /E$

$X_2=(R_2 - Q_2 F)/E$

$(C+D)/(E+F)=Q_1 +Q_2X_1(F/E)+X_2(F/E)^2 -X_2(F/E)^3 +...$

$X_2=(R_2 - Q_2 F)/E =Q_3 +R_3 /E$

...

$X_3=(R_{MP-1} - Q_{MP-1} F)/E=Q_{MP} +R_{MP} /E$

$(C+D)/(E+F)= \sum_{i=1}^{MP} Q_i + R_{MP} /E$

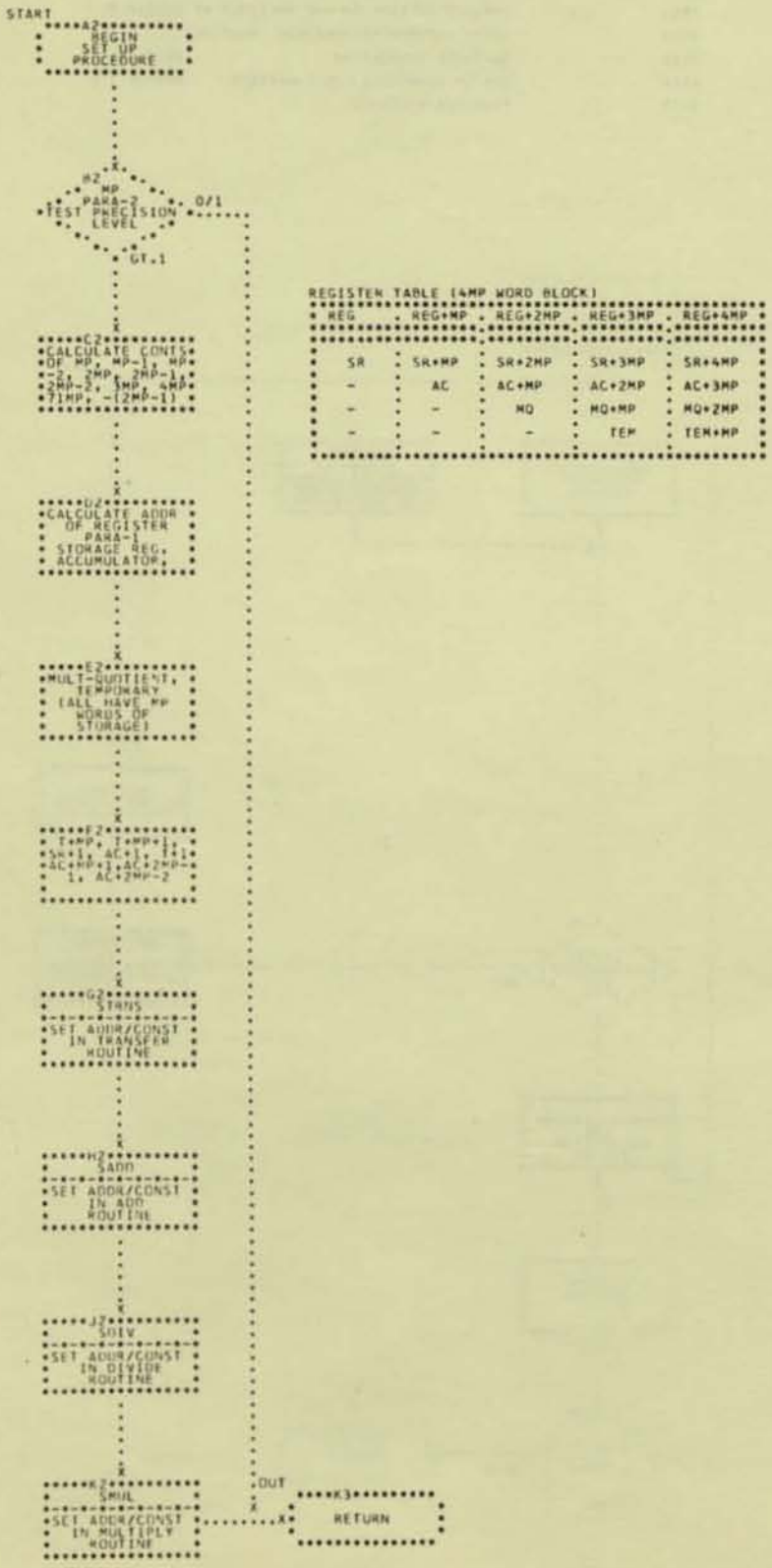
Convergence is assured in MP terms because (F/E) is less than 2^{-35}

Exponent is $A_1 - B_1$ into A_1

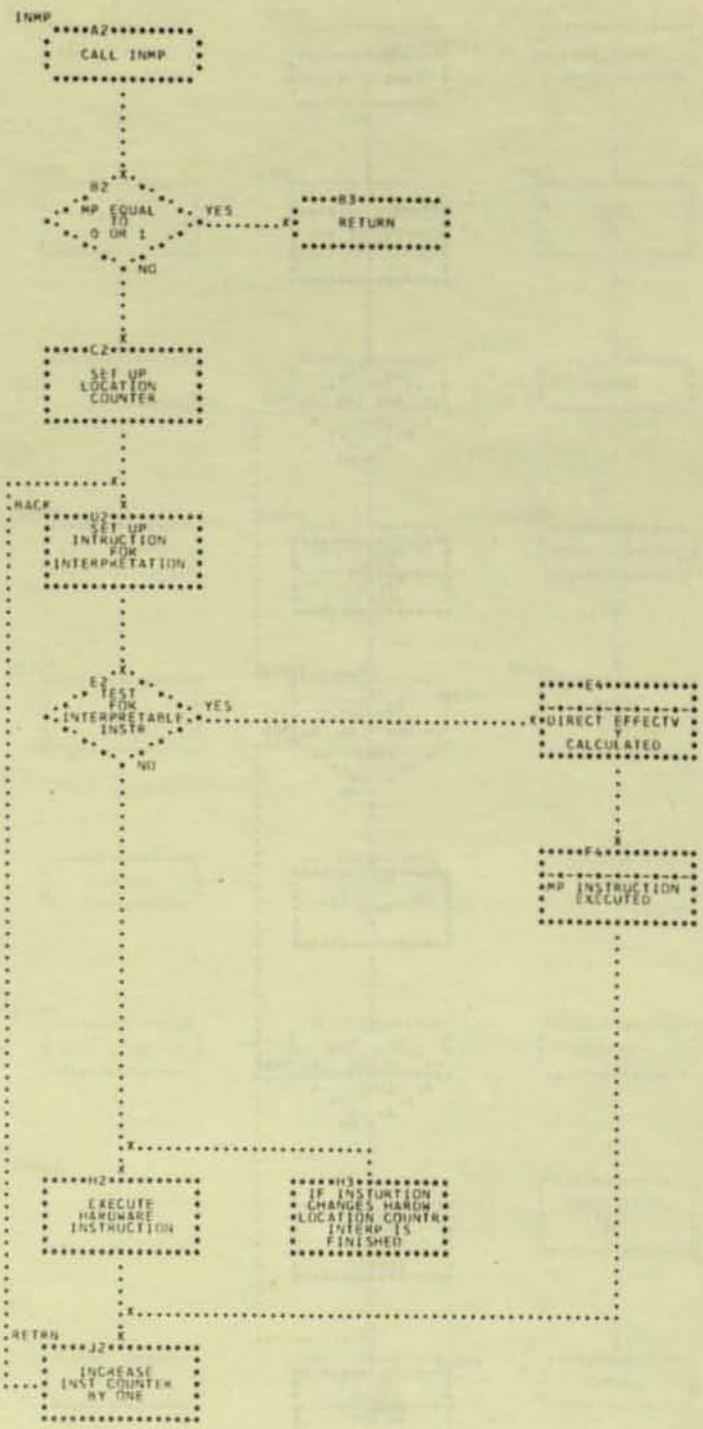
Fraction of quotient is stored in A_2 , A_3 , . . . , A_{MP}

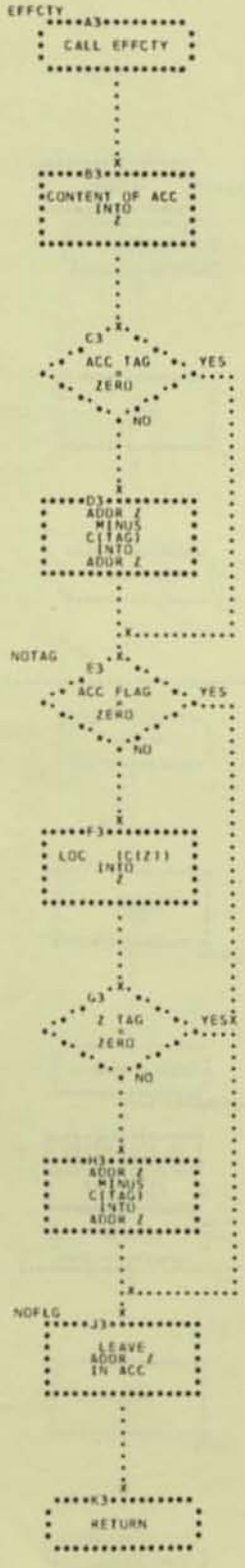
An overflow is possible. If an overflow occurs, the fraction and exponent A_1 are adjusted for normalization.

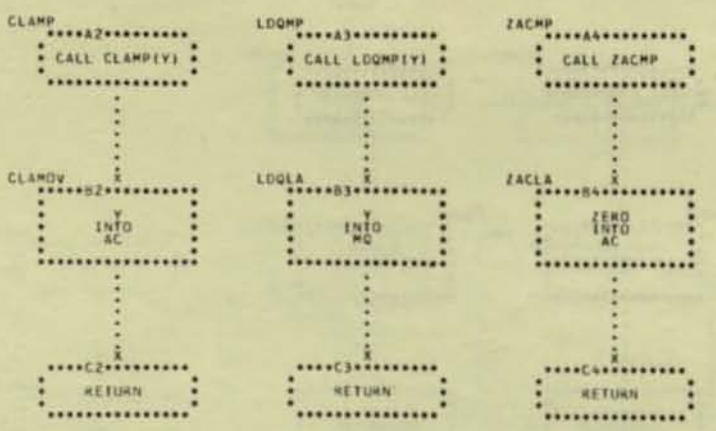
```
MASTER FILE
ADIO            ADD PROCEDURE FROM FAD,UFA,FSB,UFS WRIGHT,I L
EYIO            CALCULATE EFFECTIVE  ADDR OF INST  WRIGHT IL
FAIO            FAD,UFA,FSB,UFS ENTRIES TO ADD
FDIO            MULTI-PRECISION FLOATING DIVIDE    WRIGHT I L
FMIO            FMP,UFM ENTRIES TO THE MULTIPLY RT WRIGHT IL
INIO            BEGIN INTERPRETATION/STOP  ROUTINES
MLIO            MULTIPLY SUBROUTINE                WRIGHT I L
SEIO            SET UP ADDRESSES AND CONSTANTS     WRIGHT I L
TRIO            TRANSFER ROUTINES
```

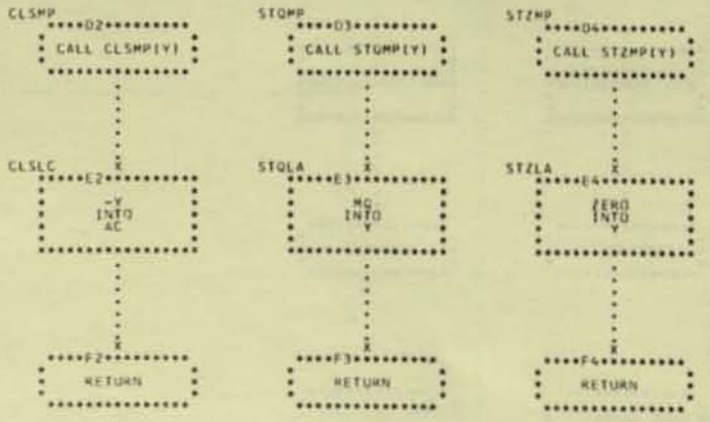

START
A2
BEGIN
SET UP
PROCEDURE
B2
MP
PARA-2
TEST PRECISION
LEVEL
0/1
GT.1
C2
CALCULATE CONTS
OF MP, MP-1, MP
-2, 3MP, 2MP-1,
2MP-2, 5MP, 4MP
7IMP, -(2MP-1)
D2
CALCULATE ADDR
OF REGISTER
PARA-1
STORAGE REG.
ACCUMULATOR,
E2
MULT-QUOTIENT,
TEMPORARY
(ALL HAVE MP
WORDS OF
STORAGE)
F2
I+MP, I+MP+1,
SR+1, AC+1, T+1
AC+MP+1, AC+2MP-
1, AC+2MP-2
G2
STORE
SET ADDR/CONST
IN TRANSFER
ROUTINE
H2
SADD
SET ADDR/CONST
IN ADD
ROUTINE
J2
SDIV
SET ADDR/CONST
IN DIVIDE
ROUTINE
K2
SMUL
SET ADDR/CONST
IN MULTIPLY
ROUTINE
OUT
K3
RETURN
REGISTER TABLE (4MP WORD BLOCK)
REG
REG+MP
REG+2MP
REG+3MP
REG+4MP
SR
SR+MP
SR+2MP
SR+3MP
SR+4MP
-
AC
AC+MP
AC+2MP
AC+3MP
-
-
MQ
MQ+MP
MQ+2MP
-
-
TEM
TEM+MP

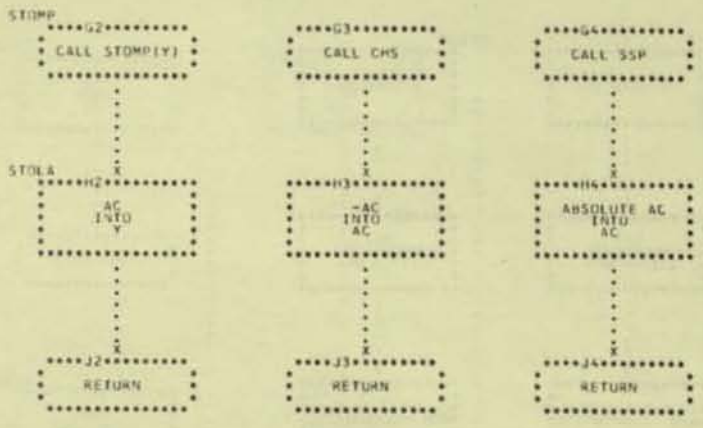
BEGIN INTERPRETATION/STOP ROUTINES




EFFCTY
A3
CALL EFFCTY
B3
CONTENT OF ACC INTO Z
C3
ACC TAG = ZERO
YES
NO
D3
ADDR Z MINUS C(TAG) INTO ADDR Z
NOTAG
E3
ACC FLAG = ZERO
YES
NO
F3
LOC (C(Z)) INTO Z
G3
Z TAG = ZERO
YES
NO
H3
ADDR Z MINUS C(TAG) INTO ADDR Z
NOFLG
J3
LEAVE ADDR Z IN ACC
K3
RETURN


CLAMP
A2
CALL CLAMP(Y)
CLAMOV
B2
Y
INTO
AC
C2
RETURN
LDQMP
A3
CALL LDQMP(Y)
LDQLA
B3
Y
INTO
MQ
C3
RETURN
ZACMP
A4
CALL ZACMP
ZACLA
B4
ZERO
INTO
AC
C4
RETURN


CLSMP
D2
CALL CLSMP(Y)
CLSLC
E2
-Y
INTO
AC
F2
RETURN
STQMP
D3
CALL STQMP(Y)
STQLA
E3
MQ
INTO
Y
F3
RETURN
STZMP
D4
CALL STZMP(Y)
STZLA
E4
ZERO
INTO
Y
F4
RETURN


STOMP
G2
CALL STOMP(Y)
STOLA
H2
AC
INTO
Y
J2
RETURN
G3
CALL CHS
H3
-AC
INTO
AC
J3
RETURN
G4
CALL SSP
H4
ABSOLUTE AC
INTO
AC
J4
RETURN

FAD,UFA,FSB,UFS ENTRIES TO ADD

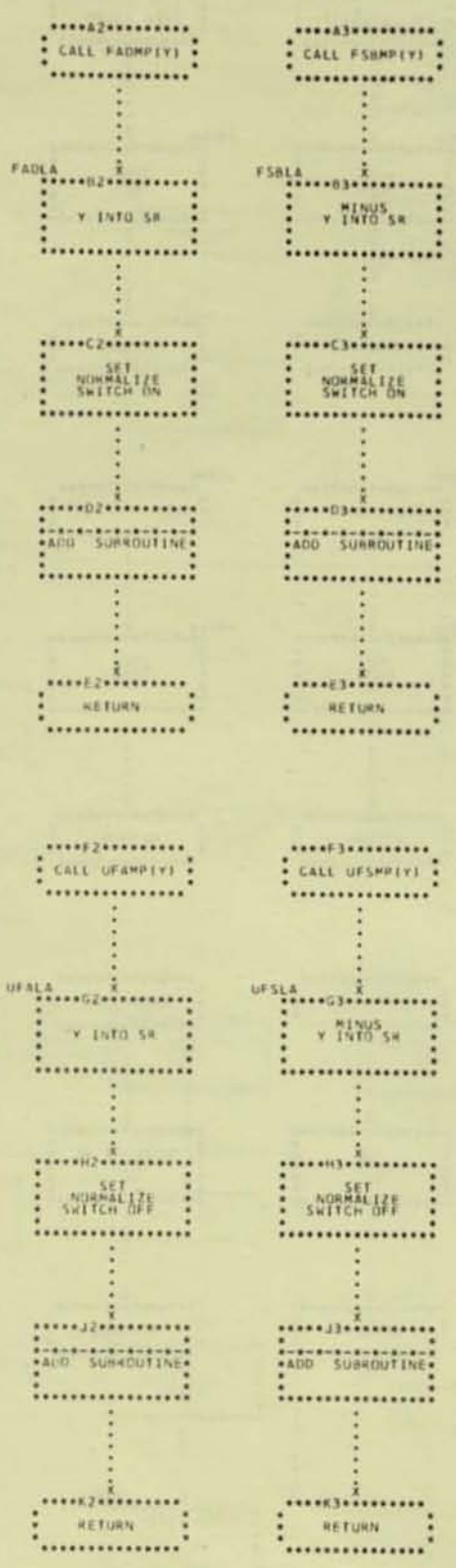
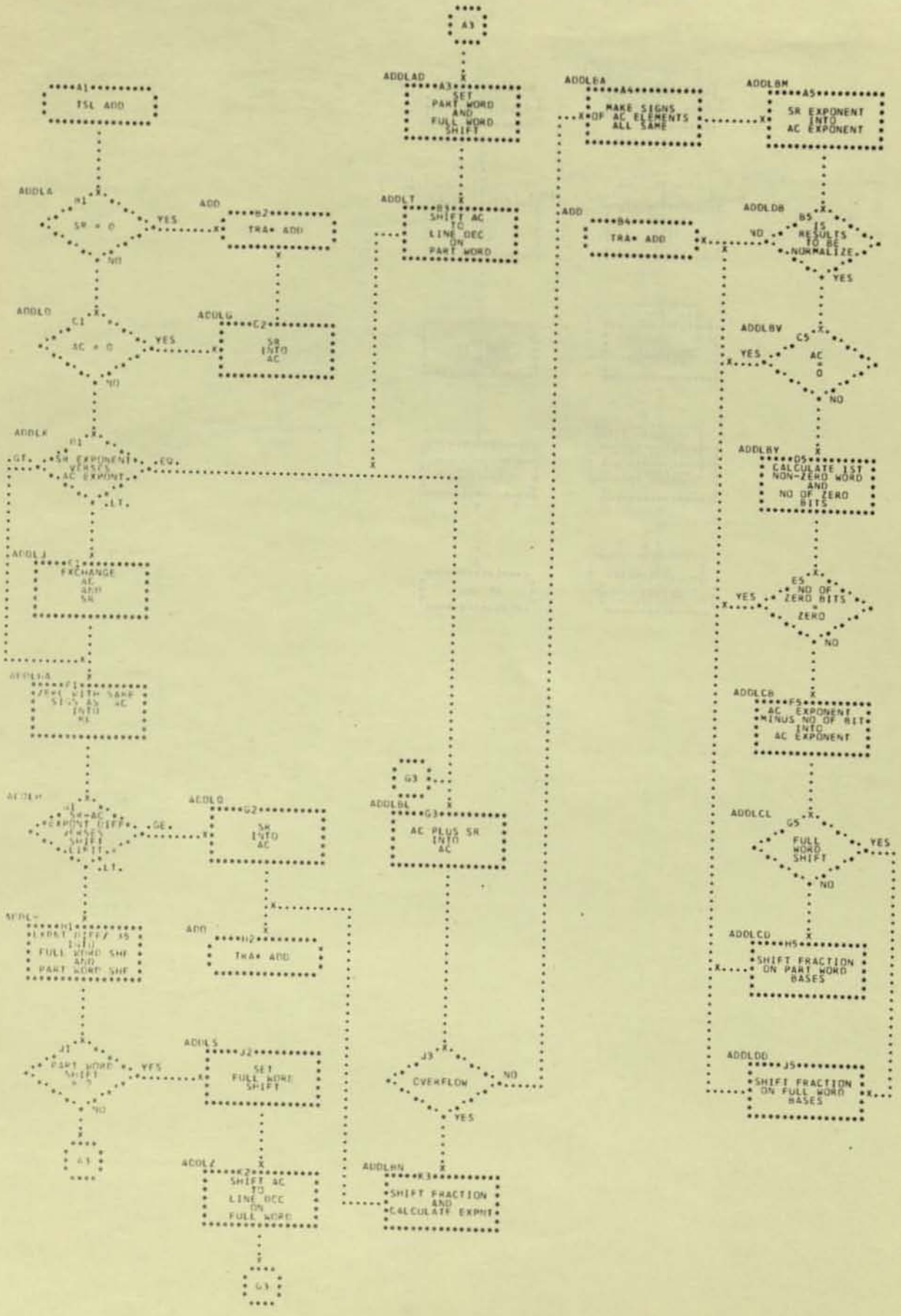
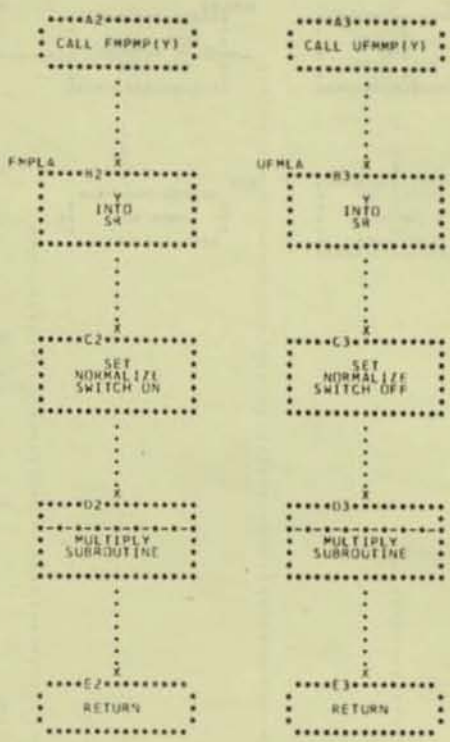


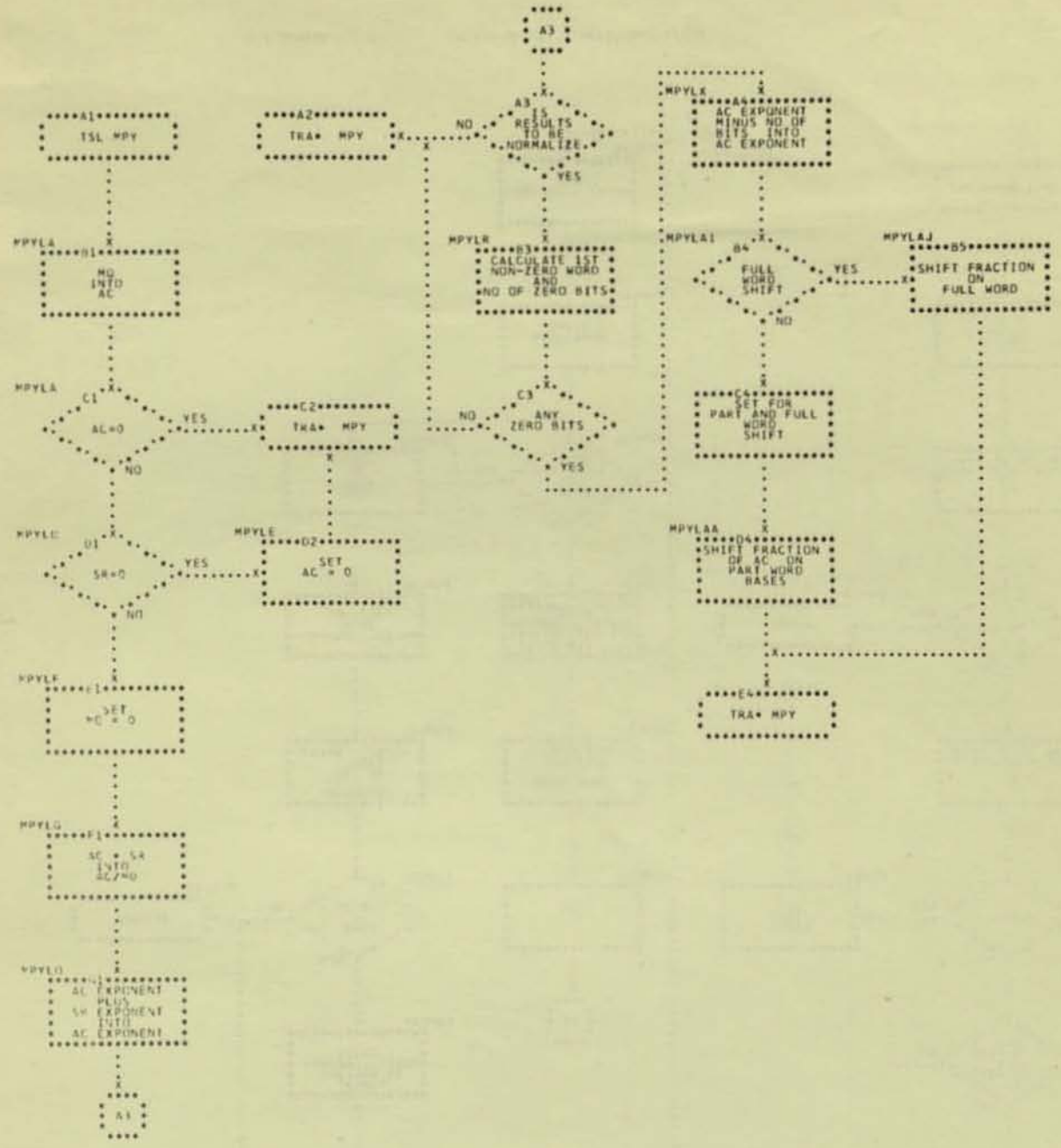
CHART AD

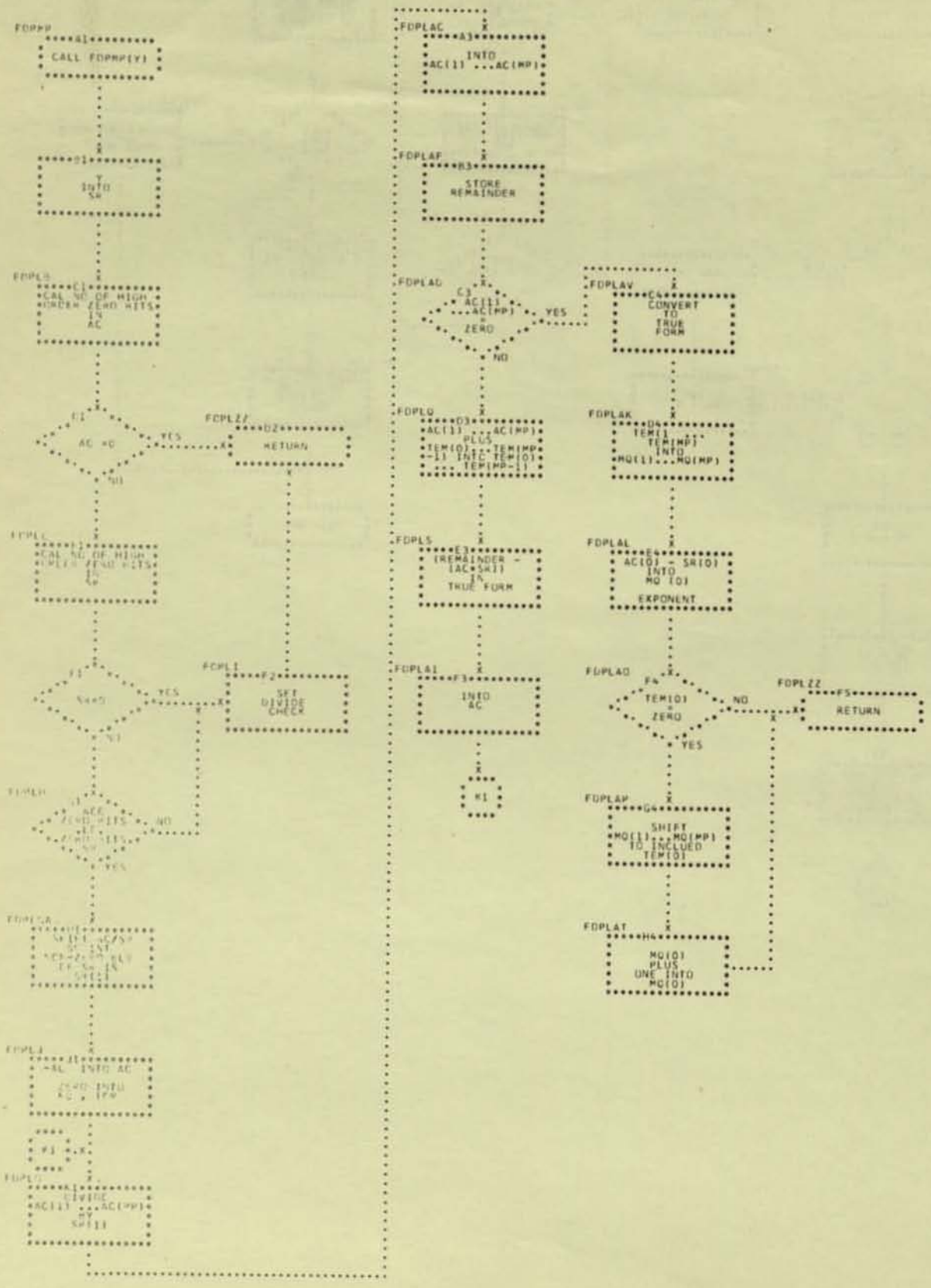
ADD PROCEDURE FROM FAD,UFA,FSB,UFS

WRIGHT, I L




A2
CALL FMPMP(Y)
FMPL&
B2
Y
INTO
SR
C2
SET
NORMALIZE
SWITCH ON
D2
MULTIPLY
SUBROUTINE
E2
RETURN
A3
CALL UFMMP(Y)
UFML&
B3
Y
INTO
SR
C3
SET
NORMALIZE
SWITCH OFF
D3
MULTIPLY
SUBROUTINE
E3
RETURN


A3
A3 IS RESULTS TO BE NORMALIZE
NO
YES
A1
TSL MPY
MPYL4
B1
MQ INTO AC
MPYLA
C1
AC=0
YES
NO
A2
TRA* MPY
C2
TRA* MPY
MPYLC
D1
SR=0
YES
NO
MPYLE
D2
SET AC = 0
MPYLF
E1
SET MQ = 0
MPYLG
F1
SC = SR INTO SC/=0
MPYLO
G1
AC EXPONENT PLUS SR EXPONENT INTO AC EXPONENT
A1
MPYLR
B3
CALCULATE 1ST NON-ZERO WORD AND NO OF ZERO BITS
C3
ANY ZERO BITS
NO
YES
MPYLS
A4
AC EXPONENT MINUS NO OF BITS INTO AC EXPONENT
MPYLAI
B4
FULL WORD SHIFT
YES
NO
MPYLAJ
B5
SHIFT FRACTION ON FULL WORD
C4
SET FOR PART AND FULL WORD SHIFT
MPYLAA
D4
SHIFT FRACTION OF AC ON PART WORD BASES
E4
TRA* MPY


FDPMP
CALL FDPMP(Y)
INTO SW
FDPLB
CAL NO OF HIGH ORDER ZERO BITS IN AC
FDPLZZ
AC =0
YES
RETURN
NO
FDPLC
CAL NO OF HIGH ORDER ZERO BITS IN SW
FDPLZ
SHIFT
DIVIDE CHECK
FDPLH
ACC ZERO BITS
NO
YES
FDPLCA
SHIFT ACC SW
FDPLJ
-AL INTO AC
ZERO INTO AC + MQ
FDPLK
DIVIDE AC(1) ...AC(MP)
SW(1)
FDPLAC
INTO AC(1) ...AC(MP)
FDPLAF
STORE REMAINDER
FDPLAG
C3
AC(1) ...AC(MP)
ZERO
YES
NO
FDPLAV
CONVERT TO TRUE FORM
FDPLQ
AC(1) ...AC(MP) PLUS TEM(0) ...TEM(MP-1) INTO TEM(0) ...TEM(MP-1)
FDPLAK
TEM(1) ...TEM(MP) INTO MQ(1) ...MQ(MP)
FDPLS
(REMAINDER - (AC*SW)) IN TRUE FORM
FDPLAL
AC(0) - SR(0) INTO MQ(0)
EXPONENT
FDPLAI
INTO AC
X1
FDPLAO
TEM(0) ZERO
NO
YES
FDPLZZ
RETURN
FDPLAP
SHIFT MQ(1) ...MQ(MP) TO INCLUED TEM(0)
FDPLAT
MQ(0) PLUS ONE INTO MQ(0)

SHARE library routines for various machines (IBM 360, 7090, 7040)

Multiprecision Floating-Point Add, Subtract, Multiply, Divide, Halve For IBM System/360
360D-40.4.002

DISCLAIMER

This program and its documentation have been contributed to the Program Information Department by an IBM employee and are provided by the IBM Corporation as part of its service to customers. The program and its documentation are essentially in the author's original form and have not been subjected to any formal testing. IBM makes no warranty expressed or implied as to the documentation, function, or performance of this program and the user of the program is expected to make the final evaluation as to the usefulness of the program in his own environment. There is no committed maintenance for the program.

Questions concerning the use of the program should be directed to the author or other designated party. Any changes to the program will be announced in the appropriate Catalog of Programs; however, the changes will not be distributed automatically to users. When such an announcement occurs, users should order only the material (documentation, machine readable or both) as indicated in the appropriate Catalog of Programs.

Multiprecision Arithmetic
Floating-Point Add, Subtract, Multiply, Divide, Halve
For IBM System/360

Howard H. Barnett

May 23, 1968

Direct Inquiries To
Howard H. Barnett
IBM Corporation
711 Hillsborough Street
Raleigh, North Carolina 27603

TABLE OF CONTENTS

PROGRAM ABSTRACT

The subroutine MPA can be used to extend the precision of the floating-point arithmetic operations Add, Subtract, Multiply, Divide and Halve up to thirty hexadecimal digits (approximately 36 decimal digits). The form of the numbers used is long floating-point with fraction extended two more full words. Parameters used to call the subroutine include length (short, long, or multiprecision) and operation (add, subtract, multiply, divide, halve). If length is short or long, the operation (except halve) is performed by use of corresponding (normalized) instructions; if multiprecision, by use of fixed point and logical instructions. At a given point in the user's program the length and operation may be selected according to values previously calculated, and may be changed each time the point is passed.
The floating-point feature is required. The program is written in Assembler language and may be used as a FORTRAN or Assembler language subroutine.

STORAGE REQUIREMENTS

Subroutine MPA 4076 bytes

SUBROUTINE SUMMARY FOR FORTRAN USE

MULTIPRECISION ARITHMETIC
FLOATING-POINT ADD, SUBTRACT, MULTIPLY, DIVIDE, HALVE
FOR IBM SYSTEM/360

Subroutine Name. MPA

Purpose. To extend the precision of the floating-point operations Add, Subtract, Multiply, Divide, Halve up to thirty (30) hexadecimal digits (approximately 36 decimal digits).

Usage. CALL MPA(A, B, C, IOP, L)

Parameter Description.

A - Name of first floating-point variable. This variable may be one, two, or four full-words in length, depending on L.

B - Name of second floating-point variable; same length as A.

C - Name of third (resulting) floating-point variable; same length as A.

IOP - Name of the operation code (has an integer value).
If IOP contains 1, then C is A plus B;
If IOP contains 2, then C is A minus B;
If IOP contains 3, then C is A times B;
If IOP contains 4, then C is A divided by B;
If IOP contains 5, then C is one-half A (for L equal 3 only).

L - Name of the precision length code (has an integer value).
If L contains 1, the operation is Short form;
If L contains 2, the operation is Long form;
If L contains 3 (or greater), the operation is in Multiprecision form. In this form the fraction parts of A, B, C are each extended two more full words to give thirty hexadecimal digits of precision.

General. The variables A, B, C must be aligned on double-word boundaries and each must have a double word following it to store the multiprecision portion. Double-word alignment is necessary for the long-form calculations. The subroutine does not clear the extra words not used in short or long operations.

The Halve operation is not included for the short or long form.

SAMPLE PROBLEM PROGRAM

EVALUATION OF A POLYNOMIAL WITH REAL COEFFICIENTS, USING MULTIPRECISION ARITHMETIC

Description

Purpose. To find the value of a polynomial f(x) having real coefficients, for a given real value X of the argument x.

Polynomial Expression. The polynomial f(x) is expressed in the form

$$f(x) = a_0x^n + a_1x^{n-1} + a_2x^{n-2} + \ldots + a_{n-1}x + a_n$$

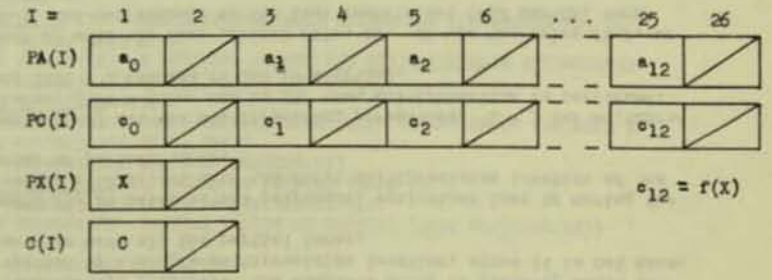
where $2 \leq n \leq 12$ in the present sample problem program, and the coefficients $a_0, a_1, a_2, \ldots, a_n$ are real-valued ("real" is used here in the mathematical sense as opposed to "complex" or "imaginary").

Method. The method of evaluation is a nested iteration of the basic form ax + b, known as "Horner's Rule", or "Synthetic Division". To use this method f(x) is written as

$$f(x) = ((\ldots(((a_0x + a_1)x + a_2)x + a_3)x + \ldots)x + a_{n-1})x + a_n$$

The following paragraphs refer to the listing of the sample problem program at the end of this section.

Storage Layout (in Multiprecision Form)



For each value of the index I, a double word of storage corresponds in the arrays PA and PC. The array PA contains the input values of the coefficients of f(x), in the order and locations shown. The array PC will be filled with the partial values obtained in the evaluation loop of f(x); c_0 is the value a_0; c_1 is the value $c_0X + a_1$; c_2 is the value $c_1X + a_2$; the loop continues until the final value $f(X) = c_{12} = c_{11}X + a_{12}$ is evaluated.

The values c_0 , c_1 , c_2 , ... , c_{11} are saved and displayed mainly to clarify the procedure, but also to suggest a way to obtain the coefficients of the reduced polynomial in case X is a root of $f(x) = 0$.

In the storage layout diagrams for the arrays PA and PC the cross-hatched double words (for I = 2, 4, 6, ... , 26) are the multiprecision extensions of the polynomial coefficients and the partial values. In the arrays PX and C they have similar meanings. In PA the coefficients are stored in the double words with odd values of I, as indicated by a_0 , a_1 , etc. Their extensions are zero at input time. The same is true for the argument X. Each partial value in PC extends from the odd-indexed double word through the following even-indexed double word.

Explanation of Program Statements

Statements 1-5 clear out the arrays PA, PC, PX in order to make sure that no previous values remain to affect the results of the present evaluation. Note that the input values in the arrays PA, PX (polynomial coefficients and value of argument X) are in double-precision form but are used in the polynomial evaluation in multiprecision form (four full words each). For this reason the extension of each of these values must be set to zero to avoid the possibility of non-zero values being picked up from these locations due to their use in a program that was executed before the present program.

Statements 6, 7 read a card containing the degree n of the polynomial to be evaluated. The degree n is denoted by N in the program.

Statements 8-10 check the value of N to make sure that it is in the range from 2 to 12 inclusive. Any other value will cause the program to terminate through statement 9. For this reason the last card of the data deck for this program contains a value of N outside this range.

Statement 11 calculates the value JK equal to 2N - 1 to be used to control reading the input coefficient cards and writing the coefficient values with multiprecision extensions, and to write the initial values of the partial terms (this writing is done to make the output a complete record of the values used and obtained in the evaluation of the polynomial).

Statements 12-14 read the double-precision input coefficients and the value of the argument x. The coefficients are stored in the odd double-precision locations of the array PA, and the argument is stored similarly in PX.

Statements 15-20 write the input values just read in, including their multiprecision extensions: the degree N, the coefficient array PA, and the argument PX.

Statements 21, 22 write the initial values (all zeroes) of the array PC in which the partial terms will be stored.

At this point the data and storage locations are ready for the polynomial evaluation loop. In a program in which the polynomial evaluation is only one of many steps, the above steps would be replaced by calculations to determine the degree, the coefficients, and the argument. Care would still be necessary to set up the multiprecision arrays properly. The complete array PC probably would be replaced by a single multiprecision location, since it is not necessary to save all the partial terms.

Statements 23, 24 initiate the polynomial evaluation loop by moving the leading coefficient into the first multiprecision location of the array of partial terms.

Statements 25-27 set the multiprecision parameters: L = 3 for multiprecision, IMPY = 3 for use as IOP when multiplication is performed, and IADD = 1 for use as IOP in addition.

Statement 28 sets the loop control value JN. JN has the value JK-2, or 2N-1, and corresponds to the last coefficient (and partial term) location.

Statement 29 starts the evaluation loop (which extends through statements 34, 39, 41) by setting the index J equal to 1, to be incremented by 2 until the value JN is exceeded.

Statement 34 calls the subroutine MPA to multiply (note the use of IMPY in place of IOP) the values in the multiprecision locations PC(J) and PX(1). These values are: the partial term obtained in the previous passage through the statements 34, 39, 41; and the argument X. (When J = 1, the partial term has the value of the leading coefficient; see statements 23, 24.) The product is stored in the multiprecision location C(1), which extends through C(2).

Statement 39 calls MPA to add the values in the multiprecision locations C(1) and PA(J+2), and to store the result (a partial term in the polynomial evaluation) in multiprecision location PC(J+2), which extends through PC(J+3).

Statement 41 increments the index J by 2 for the next pass through the loop. If J is not greater than JN, a branch is made to statement 34 and the next partial term is calculated. When the index J exceeds JN, an exit is made from the loop (34, 39, 41). The last partial term calculated is the multiprecision value of the polynomial for the given argument.

Statements 42, 43 write the values of the partial terms stored in the array PC. Only the double-precision part of each term is written.

Statement 44 branches to statement 1 to repeat the above procedure for another polynomial.

Listing of Sample Problem Program

```
C     EVALUATION OF A POLYNOMIAL WITH REAL COEFFICIENTS
C     BY USE OF MULTIPRECISION SUBROUTINE  MPA
C     DEGREE  N  (2 LTE N LTE 12)
C
      DOUBLE PRECISION PA(26),PC(26),PX(2),C(2)
C
C     CLEAR ARRAYS  PA, PC, PX
    1 DO 5 J=1,26
    2 PA(J) = 0.0D0
    3 PC(J) = 0.0D0
    4 PX(1) = 0.0D0
    5 PX(2) = 0.0D0
C
C     READ INPUT FROM CARDS (READ DEGREE N)
    6 READ(1,7) N
    7 FORMAT(I5)
C     TEST FOR EXIT (N LESS THAN 2)
    8 IF(N-2) 9,11,10
    9 CALL EXIT
C     TEST FOR EXIT (N GREATER THAN 12)
   10 IF(N-12) 11,11,9
C     READ INPUT FROM CARDS (READ N-1 COEFFICIENTS)
   11 JK = N+N+1
   12 READ(1,13) (PA(J),J=1,JK,2)
   13 FORMAT(D24.16)
C     READ INPUT FROM CARDS (READ VALUE OF VARIABLE X)
   14 READ(1,13) PX(1)
C
C     WRITE INPUT AND MULTIPRECISION EXTENSIONS FOR ARRAYS PA, PX
   15 WRITE(3,16) N
   16 FORMAT(6H1  N =I5/38H  COEFFS AND MULTIPRECISION EXTENSIONS)
   17 WRITE(3,18) (PA(J),PA(J+1),J=1,JK,2)
   18 FORMAT(2D24.16)
C     WRITE INPUT VALUE AND MULTIPRECISION EXTENSION OF VARIABLE X
   19 WRITE(3,20) PX(1),PX(2)
   20 FORMAT(15H  VALUE OF PX/2D24.16)
C     WRITE INITIAL VALUES OF PARTIAL TERMS PC
   21 WRITE(3,22) (PC(J),PC(J+1),J=1,JK,2)
   22 FORMAT(56H  INITIAL VALUES OF PARTIAL TERMS PC/(2D24.16))
C
C     POLYNOMIAL EVALUATION
C     SET C0 = A0
   23 PC(1) = PA(1)
   24 PC(2) = PA(2)
C     SET PARAMETER L FOR MULTIPRECISION
   25 L = 3
```

```
C     SET PARAMETER  IMPY  FOR MULTIPLICATION
   26 IMPY = 3
C     SET PARAMETER  IADD  FOR ADDITION
   27 IADD = 1
C
C     LOOP TO CALC PARTIAL TERMS IN POLYNOMIAL EVALUATION
   28 JN = JK-2
   29 DO 41 J=1,JN,2
C     CALL MPA FOR MULTIPLICATION
   34 CALL MPA(PC(J),PX(1),C(1),IMPY,L)
C     CALL MPA FOR ADDITION
   39 CALL MPA(C(1),PA(J+2),PC(J+2),IADD,L)
   41 CONTINUE
C     (LAST PARTIAL TERM PC(J+2), PC(J+3) (FOR J=JN) IS POLYNOMIAL VALUE)
C
C     WRITE RESULTS (DOUBLE PRECISION ONLY)
   42 WRITE(3,43) (PC(J),J=1,JK,2)
   43 FORMAT(39H  CALCULATED VALUES OF PARTIAL TERMS PC/(D24.16))
C     DO ANOTHER POLYNOMIAL OR EXIT
   44 GO TO 1
      END
```


DECK KEY

1) Source Deck

Assembler Language
Number of cards in source deck 1152
Sequence-numbered in columns 73-80
Columns 73-75 contain MPA
Columns 76-80 contain 00010, 00020, ... , 11520

One blank separator card 1

2) Sample Problem Deck

FORTRAN Source Deck
Number of cards in FORTRAN Source Deck 67
Sequence-numbered in columns 73-80:
MPA30010, MPA30020, ... , MPA30670

One blank separator card 1

3) Sample Problem Data Deck

Number of cards in Sample Problem Data Deck 61
Sequence-numbered in columns 73-80:
MPA40010, MPA40020, ... , MPA40610

(See Sample Problem Input, page 63)

Note. The above does not include the control cards required for assembling MPA and making it available in object deck or equivalent form for use by the sample problem or other program. An example of actual use of control cards for this purpose under DOS/360 is given in the section "Control Card and Program Deck Sequence".

USER INFORMATION

Detailed Program Description

1) Purpose. This program provides a means to extend the precision of the floating-point operations Add, Subtract, Multiply, Divide and Halve beyond fourteen and up to thirty hexadecimal digits (approximately 17 and up to 36 decimal digits).

2) Advantages. This program makes it possible to perform calculations with greater precision than afforded by the long-form operations Add, Subtract, Multiply, Divide, Halve. This extra precision is necessary when a program is being converted to IBM System/360 from some other system such as IBM 1620 using FORTRAN II, in which the number of digits in floating-point operations may have been specified above 17 and up to 28. This extra precision is necessary for similar programs not yet written, and it cannot be obtained by a combination of two long-form numbers because of the loss of significance due to truncation at the end of the high-order number. The increased precision is obtained at the expense of increased execution time (see paragraph 7) Timing).

3) Method. The program substitutes the use of fixed-point and logical instructions in place of the floating-point operations Add, Subtract, Multiply, Divide, Halve. Several tests are made to avoid such operations as adding or subtracting zero or multiplying by zero, and combining two numbers when one is relatively insignificant or the result is outside the range permitted by the floating-point structure in IBM System/360. Addition and Subtraction are combined by absorbing the operation into the sign of the second variable. Two's complementation is used if necessary to prepare the numbers to be added and to restore the result to true form.

In Multiplication the signs are combined and removed from the fractions, so that no sign consideration is necessary. The fractions are changed in form to four "digits" each, to base 2**32 (see Fig. B). The multiplication procedure is based on one of the methods of multiplying two four-digit decimal integers (see Fig. C).

In Division, the signs are combined as in Multiplication. Repeated subtraction (logical addition of two's complements) of the 1-, 2-, 4-, and 8-multiples of the denominator fraction from the numerator fraction is used to generate the digits of the quotient.

4) Restrictions and Range. No restrictions are considered applicable to this subroutine that do not apply to the short and long forms of the operations considered. The range in magnitude is the same because the seven-bit characteristic is used as in the long and short forms. Each variable must be assigned two consecutive double words of storage.

5) Precision. This subroutine provides up to 36 decimal digits of precision. The variables must be used in double-precision form.

6) Program Requirements. The subroutine uses the standard linkage conventions for passing control and parameters, used by FORTRAN IV under the operating systems DOS/360 and OS/360.

A program using MPA must have its floating-point variables defined in double-precision form, and each such variable which may be used or calculated by MPA must have an extra double-word of storage following it (see Fig. A).

7) Timing.

Approximate maximum timing, in milliseconds, using the timing table for IBM System/360 Model 30 with 1.5 microsecond storage cycle:

Multiprecision Add or Subtract	9.5
Multiprecision Multiply	23.2
Multiprecision Divide	357.
Multiprecision Halve	1.76

Approximate average timing, in milliseconds, under above conditions to perform entry and exit for

CALL MPA(A, B, C, IOP, L)	0.5

Sample Problem

The sample problem is a program written in FORTRAN to evaluate a polynomial of degree between 2 and 12 inclusive, with real coefficients and a real argument. The program reads a card containing the degree N. If N is not between 2 and 12 inclusive then the program terminates; otherwise N 1 cards are read, each containing one coefficient in double precision floating-point form. The coefficients are in the order of decreasing powers of the polynomial variable X. The card containing the value of X is then read and the program evaluates the polynomial by the method of Horner's Rule (Synthetic Division).

The printed output includes the input (one line per card), the initial values (all zero) of the partial terms of the synthetic division evaluation, and the calculated values to sixteen significant digits of the partial terms of the evaluation. The final term of this list is the value of the polynomial. (See the section SAMPLE PROBLEM EXECUTION for further details.)

Setup of Card Decks

The setup of the card decks is given in the section: SAMPLE PROBLEM EXECUTION.

Programming Comments

The subroutine MPA permits in particular the change in length of a calculation (Add, Subtract, Multiply, Divide OR HALVE) at an arbitrary point in the problem program. If a variable has been calculated at one level (length) of precision at a given point in a program and later is calculated at that point (or another point) but at a lower level of precision, then the words of precision not used in the later calculations should be set to zero. If they are not, the values they contain will be considered part of the variable; if the variable is used still later and at a higher level of precision it will appear to have more significant digits than it actually has.

This detail may be handled in several ways. The simplest is to modify MPA to store four full words (two double words) for each result, instead of one full word for Short, two for Long, and four for Multiprecision. This may be done in a revision version of MPA if it does not affect the problem program logic. Another way is to call MPA to do the required clearing by use of zero-valued variables. A better way than this is to use the length L operand to branch to a zero-setting statement.

The use of MPA has the undesirable effect of expanding a FORTRAN program to a single-operation-per-statement program wherever Multiprecision Add, Subtract, Multiply, Divide, or Halve is required. The use of auxiliary statements can be minimized by the use of the appropriate subscripted variables as parameters. The sample FORTRAN program included here indicates this clearly.

Any suggestions to simplify the use of MPA will be welcomed by the author.

SYSTEMS MATERIAL

Program Listing

No listing of the subroutine MPA is included because it may be obtained by the user at the time of assembling the subroutine. A listing of the sample problem program is given earlier in this documentation.

Flowchart Narrative Description

1) Short and Long Operations

In the case of a Short-form or Long-form operation, the appropriate sequence of instructions is executed and the result C is returned to the problem program in the proper length (see Figs. 3, 4).

The Long-form requires the operand to be aligned on a double-word boundary; this condition is considered to be necessary for execution of MPA .

Since MPA requires each operand to occupy two consecutive double-words of storage, a reduction in length will leave a residue in the unused words of storage which may be used later when the length is increased; this will cause incorrect results either in the calculation or in the storage. The user must be careful of this source of error. It may be necessary to modify MPA so that a Multiprecision form of result is always stored.

The Halve operation is not available in MPA for the Short or Long form operation.

2) Addition and Subtraction

In the case of Multiprecision Addition or Subtraction, A and B are first tested for the value zero. This test is made on the left-most word only, on the strength of the assumption that A and B are in normalized form if not zero and contain only zero bits if zero.

If A and B are both zero, the result is zero and is ready for transmission to the problem program (see Fig. 5).

If B is zero and A is not zero, then C is set equal to A. (See Fig. 5)

If B is not zero, the operation Add or Subtract is absorbed into the sign bit of B by not changing the bit if the operation is Add, or by changing the bit if the operation is Subtract. (See Fig. 5)

If B is not zero and A is zero, then C is set equal to B with absorbed operation bit (see Fig. 5).

If B is not zero and A is not zero--the general case--the main section of the Add/Subtract part is entered. (See Figs. 5, 6) In this section the characteristics of A and B are compared. The number having the smaller characteristic has its fraction part shifted right the number of hex digits equal to the difference between the characteristics. However, if this shift number is 30 (decimal) or greater, no shift is made and the result is set equal to the other number (see Figs. 6, 7).

At this point (see Fig. 8) the two characteristics are equal, that is, the fractional parts have been aligned (if necessary) with regard to their hex points, and the operation may be performed.

It may be that the two numbers are equal or equal but opposite in sign. In the latter case the result is set equal to zero (see Figs. 8, 16). However, if the two numbers are equal, the result will be twice the common value. The result is obtained by shifting the fraction and in some cases adjusting the characteristic. If the leading hex digit is between 1 and 7 inclusive, the result is

obtained merely by shifting the fraction part left one bit. If the hex digit is between 8 and F inclusive, the result is obtained by shifting the fraction part right three bits and adding 1 to the characteristic (equivalent to division by 8 and multiplication by 16). This may cause an overflow of the characteristic; if so, the maximum signed value is stored as the result (see Figs. 8, 9).

In general the values of A and B will be unequal. If they have the same sign, the sign and characteristic are stored to be used in the result (see Fig. 10). Then the two fractions are added by a sequence of add-logical instructions, with tests for carry from one portion of the sum to the next higher portion. This is exactly the same as decimal addition of two digits, except that here the "digits" vary, not from 0 through 9, but from 0 through 2**32 -1. For more detail, see Figs. 12, 13.

If A and B have opposite signs and unequal fractions--the most general case--then the fraction of the number with negative sign is two's complemented and an indicator is set to indicate that one of the fractions has been so changed (see Figs. 10, 11). This indicator is used later to decide whether to form the two's complement of the result C (see Fig. 13). (If A and B had the same sign, no two's complementation was done and the indicator was set to record this condition.) Then the two fractions are added by the sequence of add-logical instructions referred to above.

If neither the A nor the B fraction was complemented, the C fraction is stored in preparation for a test of "carry" into the characteristic byte (see Fig. 13). If A or B was complemented and no carry out of word 1 of C occurred, the C fraction is two's complemented and stored: but if a carry-out occurred, the fraction is not two's complemented (see Fig. 13).

If the C fraction does not extend into the characteristic byte, the sign bit and characteristic of the result are combined for later use in normalizing the result C. If the C fraction does extend into the characteristic byte, the C fraction is rounded off by adding hex 8 to the low-order hex digit. The characteristic is increased by one and tested for overflow; if overflow, the result C is set to the maximum signed value and returned to the problem program; if no overflow, the C fraction is shifted right one hex digit. At this point the sign bit and characteristic are combined (see Fig. 14).

The C fraction is normalized by testing the high-order hex digits until a non-zero digit is found. If there are no high-order zeros, the result C is completed by attaching the (sign, characteristic) byte. If there are high-order zeros, the fraction is shifted and the characteristic is reduced by the number of leading zero hex digits. This may result in the C fraction being found insignificant; if so, the C result is set to zero. Otherwise the revised (sign, characteristic) byte is attached to the normalized fraction to give the result C (see Figs. 15, 16).

3) Multiplication

In the case of multiprecision multiplication, if either A or B is zero the result C is set to zero. If neither is zero, the product sign bit is isolated and A and B are from here on considered to be positive. The characteristic for the product is calculated; if it underflows the result is set to zero, but if it overflows the result is set to the maximum value with sign (see Fig. 17).

In the general case, the fraction parts of A and B are "digitized" by clearing the (sign, characteristic) byte and shifting the fraction part so that the three low-order words have leading bit (sign bit) zero and the highest order word has sign bit zero and the next four bits (leading hex digit) also zero. A and B are now in the same form as a four-digit decimal number. The digits of A and B will be referred to from high to low order as A1, A2, A3, A4 and B1, B2, B3, B4 (see Fig. 18).

Multiplication begins by forming the product of A4 and B4. A shift is made so that the low-order word has a sign bit zero. The two words of the product are stored in the "digit" locations C6, C7 (see Fig. 19).

The next multiplication uses A3 and B4, then A4 and B3. Each product is shifted as was the product of A4 and B4 to obtain two "digits" which are added to C5 and C6. The addition is the add-logical type; if a carry into the sign bit occurs, say in C6, the sign is cleared and C5 is increased by one. If a similar carry in C5 occurs, the sign bit is cleared and a one is added to C4. (See Figs. 19, 20.)

Succeeding multiplications are handled as above. The products of A2 and B4; of A3 and B3; and of A4 and B2 are added-logical into C4 and C5, with possible carries from C5 into C4 and from C4 into C3.

The products of A1 and B4; A2 and B3; A3 and B2; and A4 and B1 are added-logical into C3 and C4, with possible carries into C3 and C2.

The products of A1 and B3; A2 and B2; and A3 and B1 are added-logical into C2 and C3, with possible carries into C2 and C1.

The products of A1 and B2; and A2 and B1 are added-logical into C1 and C2, with possible carries into C1 and C0.

Finally, the product of A1 and B1 is added-logical into C0 and C1, with a possible carry into C0. No carry out of C0 is possible because A1 and B1 each has a high-order hex digit value of zero. Thus the product at this point has its two high-order hex digits equal to zero.

The next step is to "compress" the five high-order "digits" of the product back into the form of the fractions of A and B, that is, so that the high-order eight bits--the (sign, characteristic) byte--are zeros and the sign bits of the three low-order "digits" are squeezed out (see Fig. 21). The fifth digit is retained for round-off. The round-off may result in an "overflow" of the fraction into the (sign, characteristic) byte. In this case the characteristic is increased by one; if it overflows then C is set equal to the maximum value with sign. Otherwise the fraction is right-shifted one hex digit (see Fig. 22).

At this point the fraction may have one or more zero-valued high-order hex digits. The fraction is left-shifted one hex digit at a time until the lead hex digit is not zero. The number of these shifts is subtracted from the characteristic.

Finally, the (sign, characteristic) byte is attached to the normalized fraction and the result C is transferred to the designated location in the problem program.

4) Division

The basis of this method of Division is repeated subtraction. The subtraction section of MPA is not used but later refinements of Division may combine the two sections so that there will be less duplication of coding.

The fraction parts of A and B are assumed to be normalized. Various preliminary tests are made to avoid interrupts, possible non-terminating loops, and unnecessary calculations. These tests are for the value zero.

Is the denominator B zero? If B is zero, is the numerator A zero? If A is also zero, the division cannot be performed. This situation requires special consideration. If A is not zero, the value of C should be given the maximum value and the sign of A should be attached (see Fig. 23).

If the denominator B is not zero, is A zero? If A is zero, the result should be zero. If A is not zero, the general case is present and the following procedure is begun (see Fig. 23).

The sign of the quotient C is obtained from the signs of A and B by an exclusive-OR operation (see Fig. 23).

The preliminary value of the characteristic of C is obtained by subtracting the denominator characteristic from the numerator characteristic and adding 64 (decimal) to the difference (because of the excess-64 form of the characteristic). If this value exceeds 127 (decimal) the result should be given the maximum value with the sign as determined above. But if this value is negative, the quotient should be set to zero (see Fig. 24).

In the general case the quotient sign bit and a preliminary value of the characteristic have been determined. Attention is now turned to the fraction portions of A and B. The two fractions are cleared of their respective (sign, characteristic) bytes, leaving two leading hex digits with value 00 in each fraction (see Fig. 24).

The two fractions are compared, down to the least significant pair of digits if necessary. If the fractions are equal, then the quotient fraction is set equal to 100...0 (30 hex digits). The preliminary value of the characteristic is increased by 1: the above fraction has an implied hexadecimal point to the right of the "1" because the fractions are equal, but the fraction part of a floating-point number has an implied hexadecimal point to the left of the leading fraction digit. If the revised characteristic is too large, the final result C is set equal to the maximum value with the sign bit determined earlier. (See Fig. 25)

If the numerator fraction is smaller, the denominator fraction is shifted right one hex digit (four bits) and the characteristic of the result is decreased by 1: the denominator shift must be balanced by increasing the denominator characteristic; this may be done by decreasing the quotient characteristic. If the resulting characteristic is now too small (less than -1), the result C is set to zero. (The characteristic is changed and tested first, to avoid a possibly unnecessary shift of the denominator.) (See Fig. 25)

If the numerator fraction is larger, the situation is the same as after the above case. No change in the characteristic is necessary. (See Fig. 25)

It must be kept in mind that the two fractions may have the same sequence of digits down as far as the lowest (30th) digit. In case the numerator fraction is larger, the resulting quotient fraction will have leading hex digit "1" followed by several zeroes.

The repeated subtraction cycling is to be done now to generate the hex digits of the quotient fraction. To shorten this lengthy process, certain multiples of the denominator fraction are generated and also the two's-complements of these multiples. The multiples are used to compare with the numerator and the two's-complements are used in the repeated subtraction. (See Fig. 26)

The repeated subtraction reduces the numerator fraction, eventually reducing the current leading non-zero digit to zero. The repeated subtraction causes an effective shift to the right of the numerator fraction, although the numerator fraction changes its value after each subtraction since it is the remainder. To provide space for the shifting fraction and to simplify the coding, the numerator fraction has an extra block of four full words of storage reserved for it, following the four words in which the fraction starts. This numerator "pseudo-register" has therefore a length of eight full words. This eight-word block is involved in each subtraction even though each subtraction actually involves only four words. The four "words" involved in a given subtraction start at a particular hex digit which is located within one of the four bytes of a core storage full word. It seems simpler to consider a stationary block of eight words than to keep track of the current position of

the effective leading hex digit of a four-word "block" that is moving a half-byte at a time (see Figs. 27-31).

To use the above approach to the repeated subtraction the denominator fraction is also considered to be eight full words in length. Thus, in the case of the numerator fraction being smaller, the denominator fraction was shifted one hex digit to the right. The original low-order digit is after this shift the high-order digit of the fifth word of the eight-word block containing the denominator.

The denominator fraction in eight-word form is called the "1-multiple" of the denominator and is denoted by B0D. The term "1-multiple" means that B0D is the denominator fraction multiplied by 2**0 (two to the zero power equals one). Similarly three other eight-word quantities are formed: the "2-multiple", B1D, which is (2**1)*B0D equal to 2*B0D; the "4-multiple", B2D, which is (2**2)*B0D equal to 4*B0D; and the "8-multiple", B3D, which is (2**3)*B0D equal to 8*B0D. Whatever may be the lead hex digit of the numerator, subtracting a unique choice of the above multiples B0D, B1D, B2D, B3D of the denominator fraction from the numerator will reduce the numerator so that its lead hex digit is now zero and the remainder (new numerator) is less than B0D. The sum of the corresponding powers of 2: 1, 2, 4, 8, is the hex digit of the quotient.

To find the next hex digit of the quotient, the four multiples B0D, B1D, B2D, B3D are shifted right one hex digit and the process is repeated. The quantity B3D is always compared first, since it is the largest; then B2D, B1D, B0D.

To simplify the subtraction, each of the quantities B0D, B1D, B2D, and B3D has an eight-word two's complement which is added to the eight-word numerator block. These are shifted right one hex digit also before constructing each hex digit of the numerator. Each two's-complement will have leading bits with value 1, propagated by the right shift. Also, the low ends of the two's-complements will have bits with value 0 formed when the two's-complements were formed. These two's complements are named B0DC, B1DC, B2DC, B3DC.

The above shifting, comparing, and two's-complement adding continues until the remainder is zero or until thirty hex digits have been formed. In the first case the digits not yet determined are automatically all zeroes. In the second case, which is more general, the remainder is doubled and compared with the final form of B0D. If B0D is greater, no rounding of the quotient fraction is done. But if B0D is not greater, that is, if the remainder is at least one-half of B0D, a hex value 1 is added to the quotient fraction. This may cause the high-order digit to carry out one bit (a most unlikely event); in this case the quotient is shifted right one hex digit and the characteristic is increased by 1 to compensate.

If the resulting characteristic is greater than or equal to 127 (decimal), the result C is set equal to the maximum value with the sign bit of C attached (see Fig. 31).

If no round-off is necessary, or if it is done without producing too large a characteristic, the quotient fraction is ready to be assembled with the sign and characteristic to yield the result C. The characteristic must first be increased by 1 to account for an implied right shift one hex digit of the quotient fraction, because the quotient fraction as formed has an implied hexadecimal point to the right of the lead hex digit (see Fig. 31).

The above procedure for Multiprecision Division will require many compare instructions, shift instructions, and additions of the eight-word-long two's-complement blocks to the numerator block. It appears to be a procedure that is lengthy in time, although it does not require an excessive amount of storage, nor is the programming logic particularly complicated.

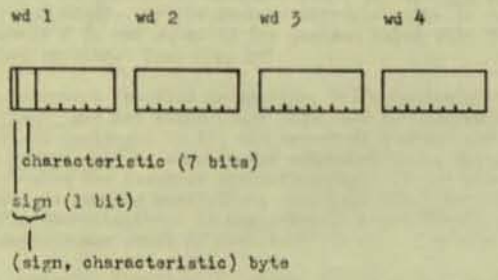
The question arises: Is there a faster method than repeated subtraction? There surely must be. The author will appreciate having his attention directed to any such methods.

Based on the above description of Multiprecision Division, the author will welcome suggestions for improving the procedure and for correcting errors in it that undoubtedly exist.

5) Halving

The Halve operation (for Multiprecision only) is included to improve performance for the special cases of dividing by two or multiplying by one-half. This is the counterpart of the use of addition of a number to itself instead of multiplying by two. Repeated uses of this operation with Add and/or Subtract may shorten execution time, although it may increase the program size.

To perform the Halve operation, the lead hex digit of the fraction is tested. If it is any value except 1 (normalized form is assumed) then the operation is performed by shifting the fraction one bit to the right. If the lead hex digit is 1, the fraction is shifted three bits to the left and the characteristic is decreased by 1 (see Fig. 32).



Short form floating-point:

wd 1 only

Long form floating-point:

wd 1, wd 2

(wd 1 on double-word boundary)

Multiprecision form:

wd 1, wd 2, wd 3, wd 4

(wd 1 on double-word boundary

for consistency only)

Fig. A. Multiprecision Form in Relation to Short, Long Forms

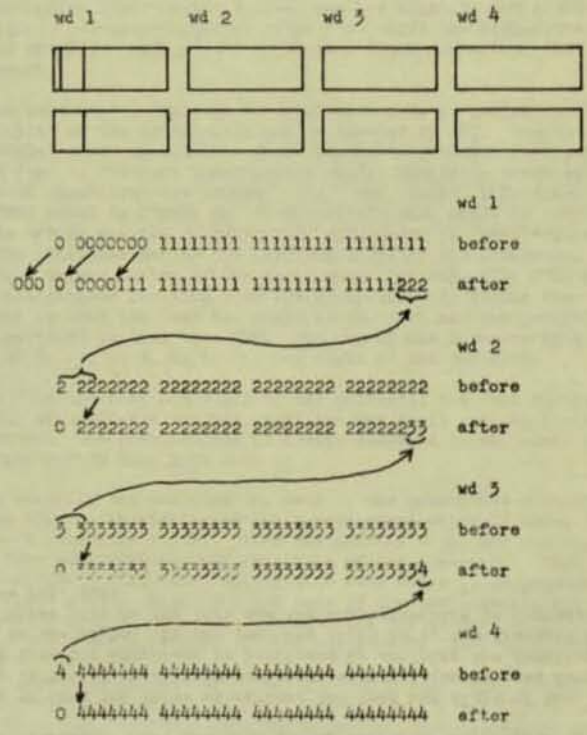


Fig. B. Multiprecision Digitizing Procedure for Multiplication

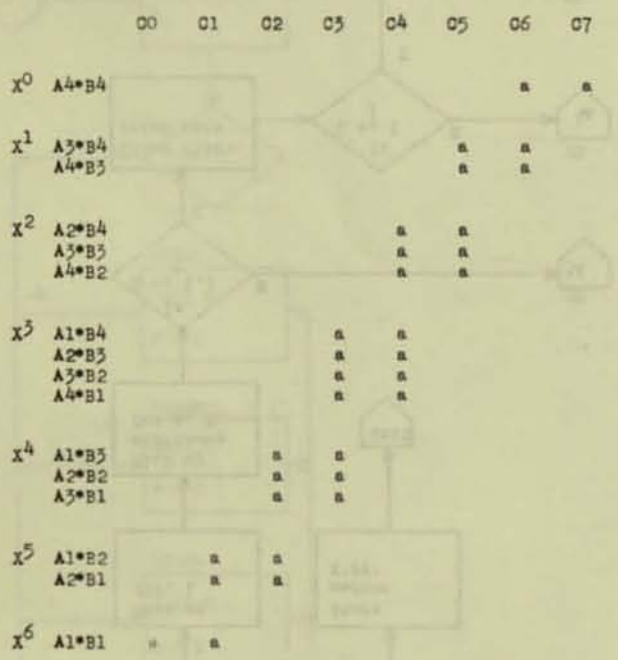


Fig. C. Multiprecision Multiplication by Digit-Product Addition

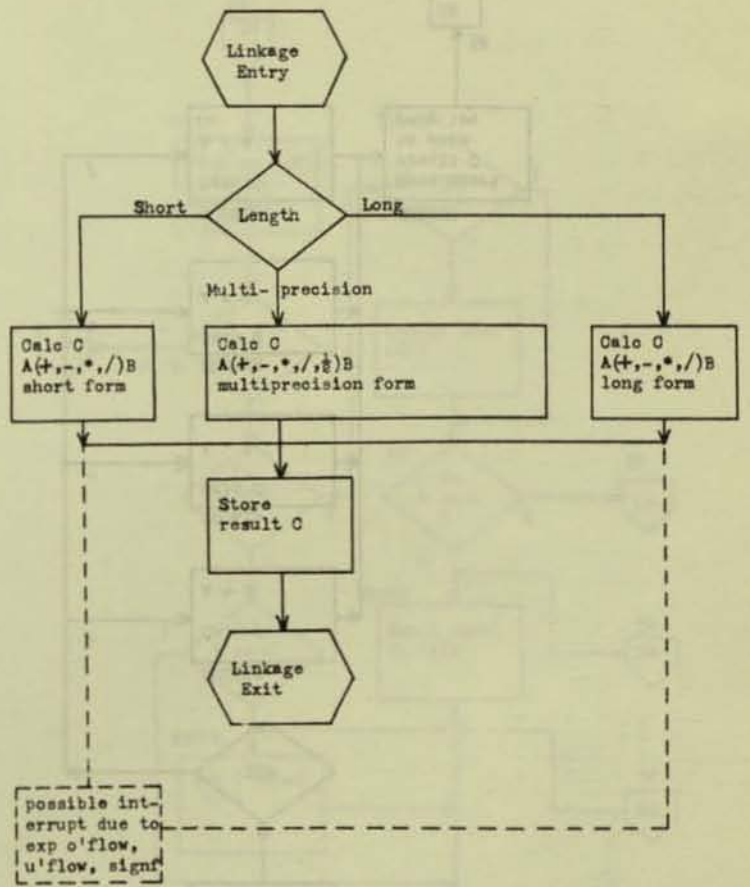


Fig. 1. Basic Flow Diagram of MPA Subroutine

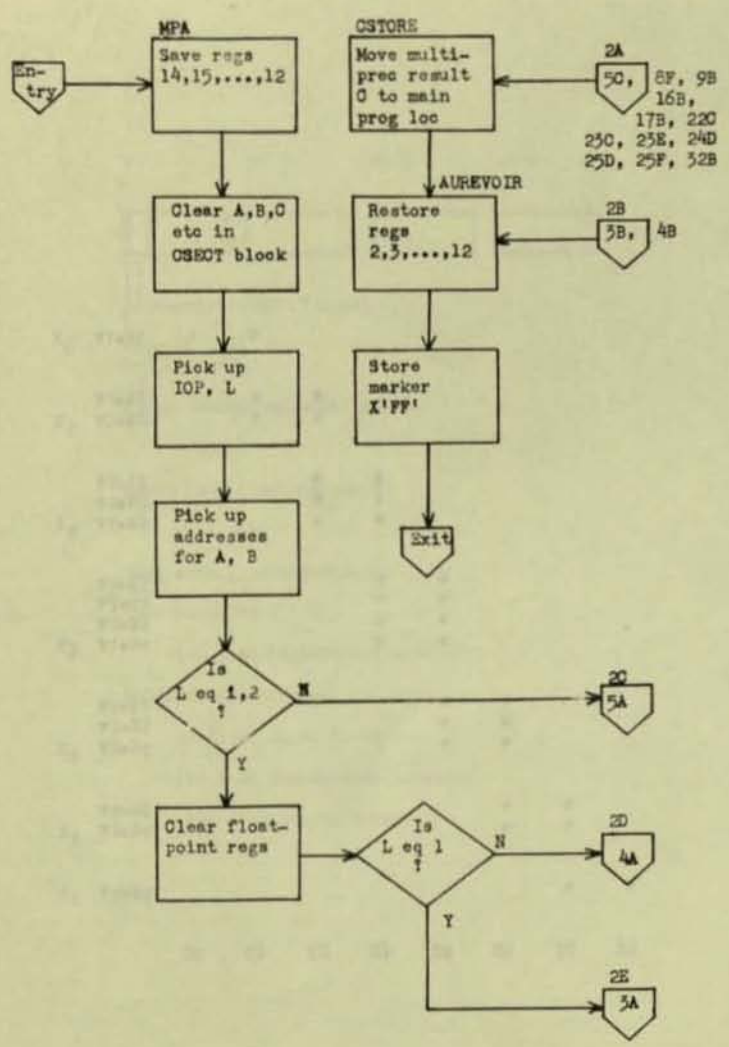


Fig. 2. Entry/Exit Preliminary

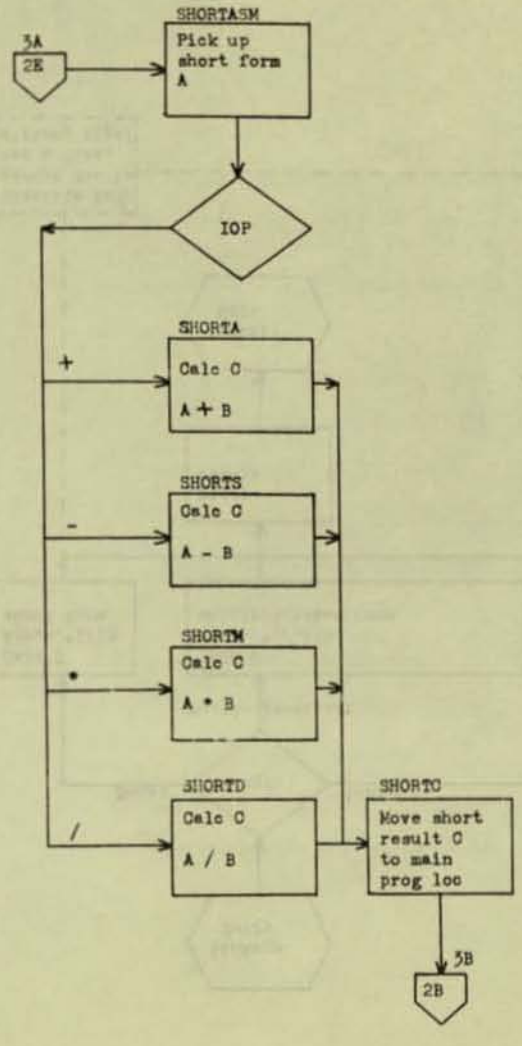


Fig. 3. Short Form Operation

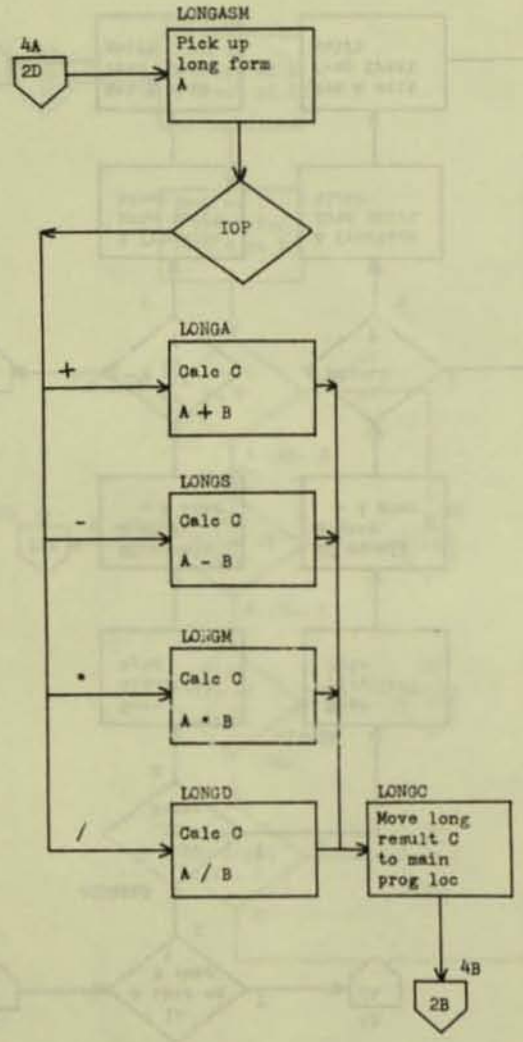


Fig. 4. Long Form Operation

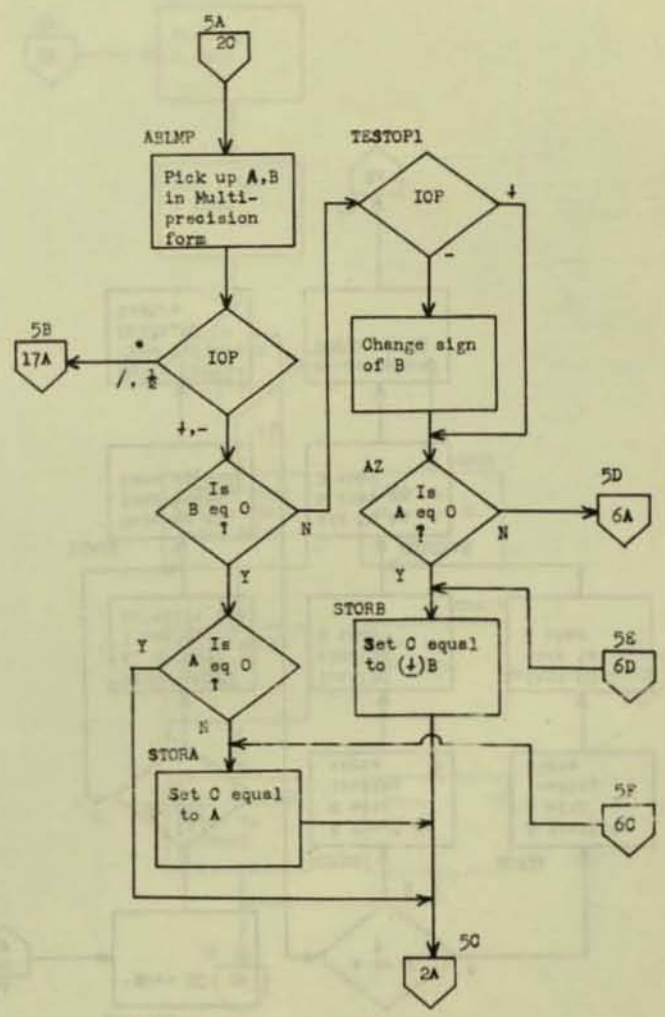


Fig. 5. Multiprecision Add/Subtract Preliminary

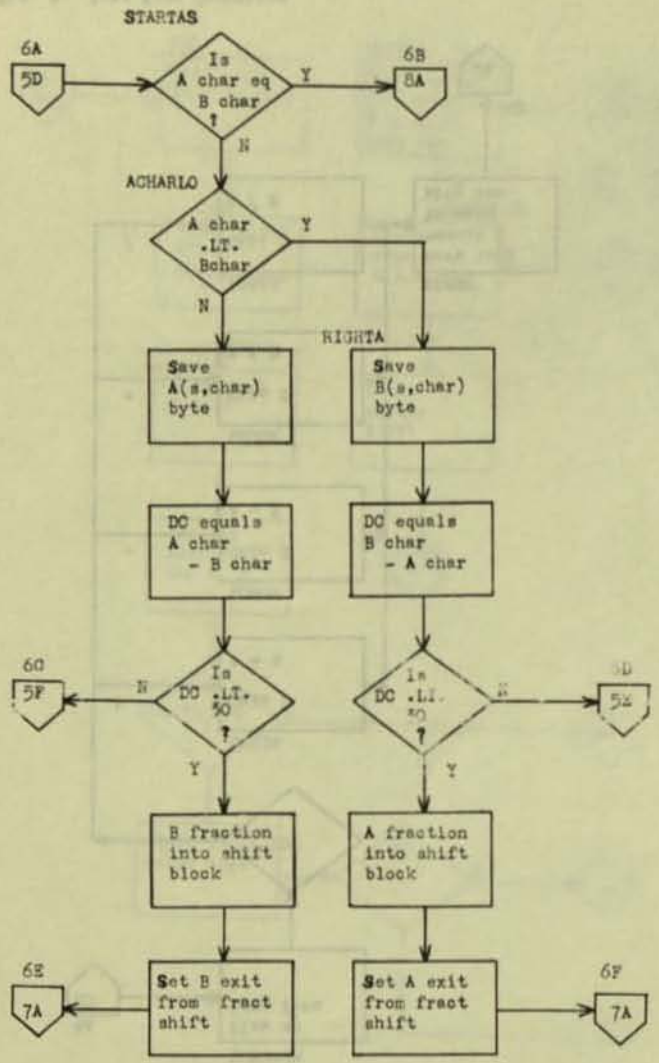


Fig. 6. Multiprecision Add/Subtract Characteristic Test

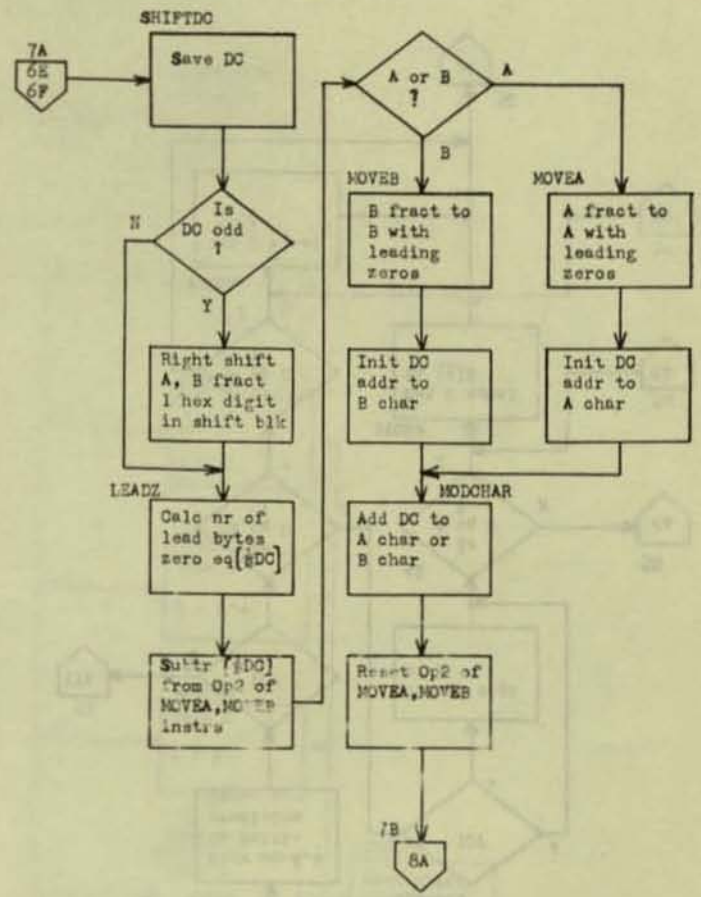


Fig. 7. Multiprecision Add/Subtract Hexadecimal Point Alignment

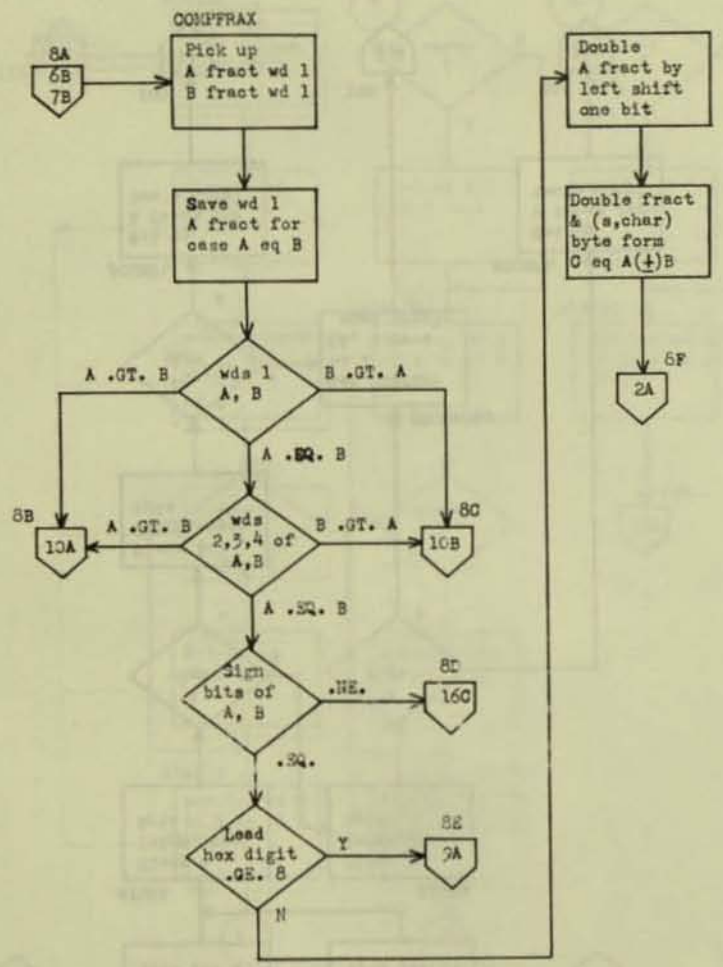


Fig. 8. Multiprecision Add/Subtract Fraction Comparison

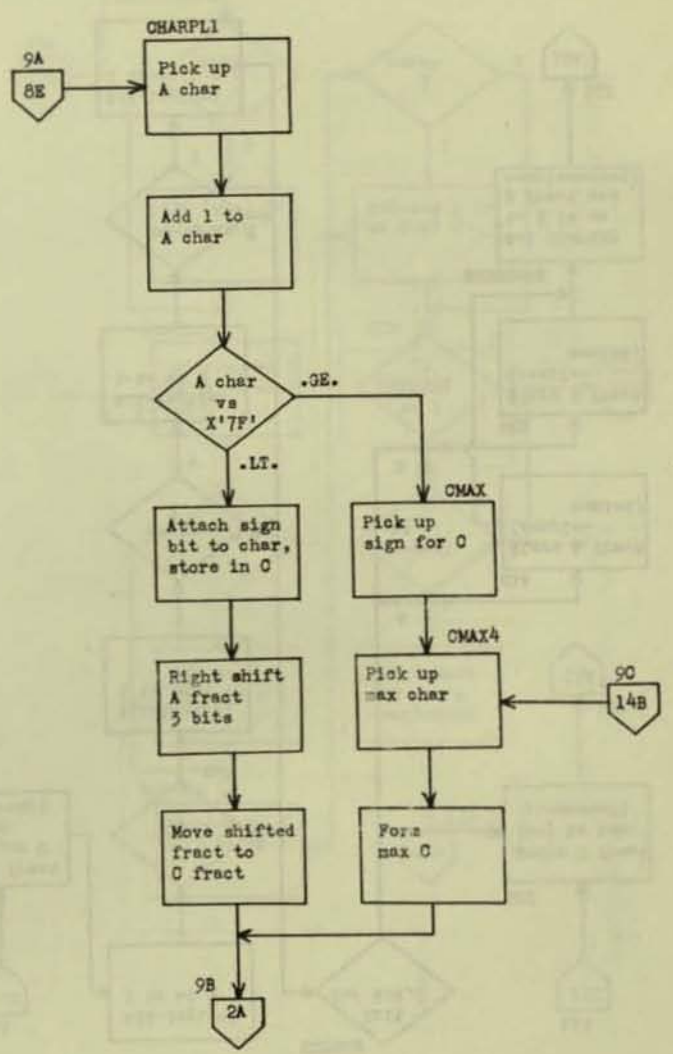


Fig. 9. Multiprecision Add/Subtr Char Adjust, Fraction Shift

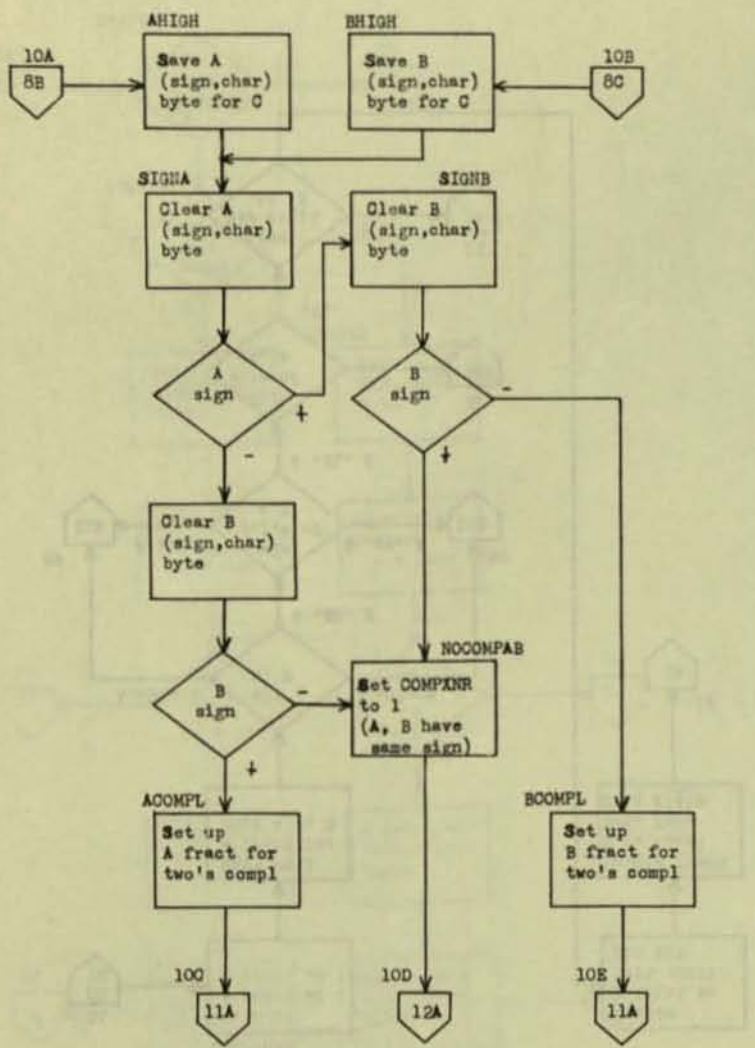


Fig. 10. Multiprecision Add/Subtr Prelim Fraction Complementation

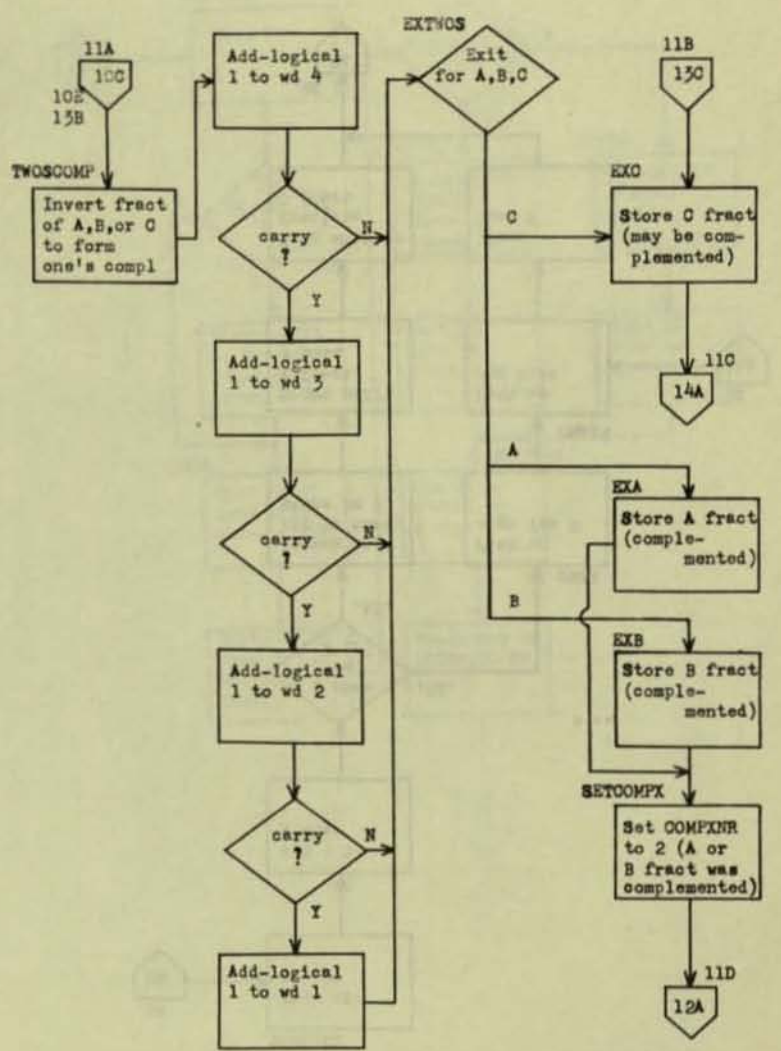


Fig. 11. Multiprecision Add/Subtract Fraction Complementation Sequence

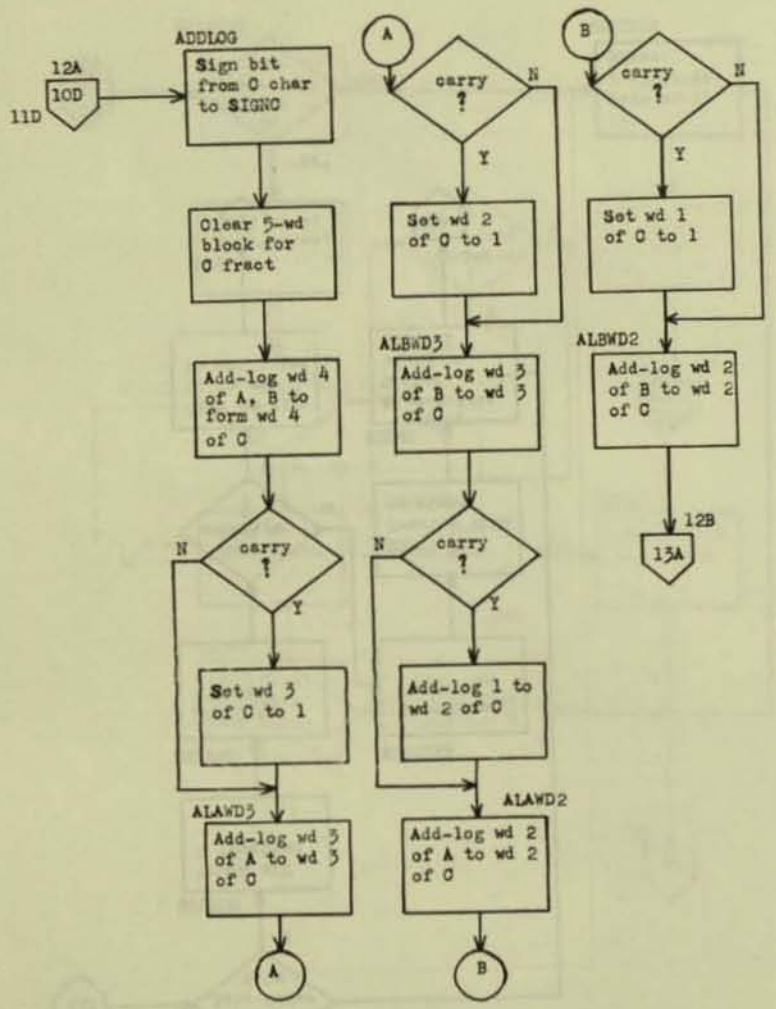


Fig. 12. Multiprec Add/Subtr Fraction Add-logical Sequence, Part 1

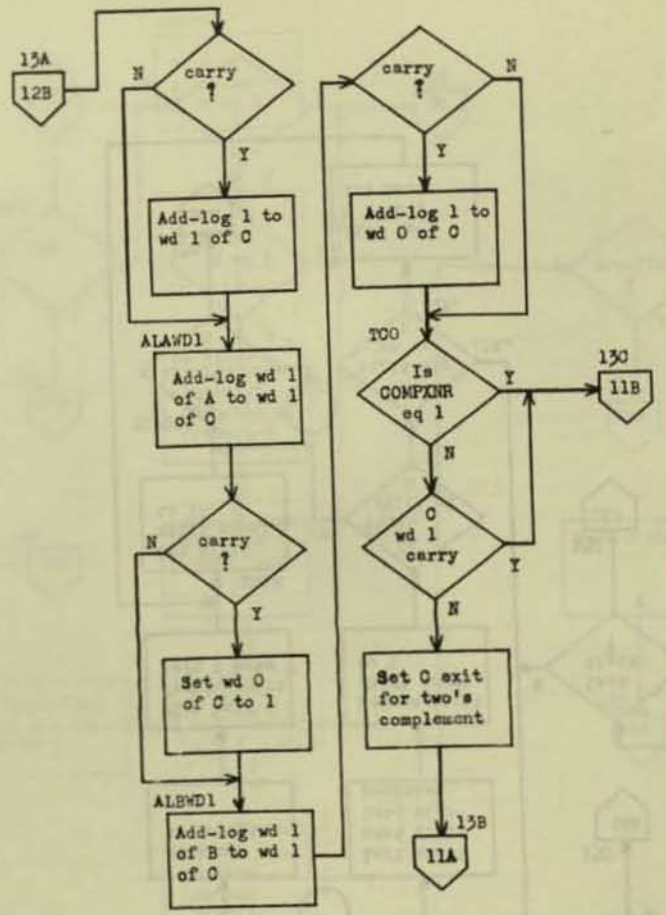


Fig. 13. Multiprec Add/Subtr Fraction Add-logical Sequence, Part 2

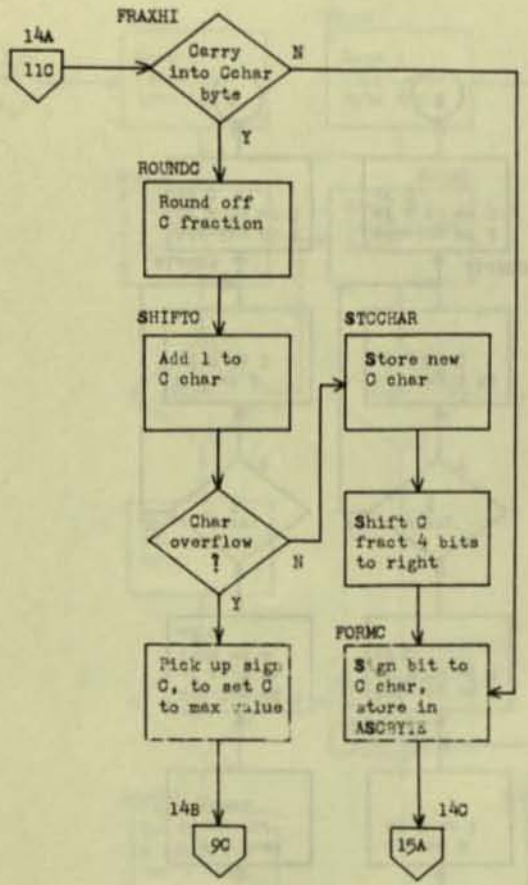


Fig. 14. Multiprec Add/Subtr Add-logical Sequence, Characteristic Test

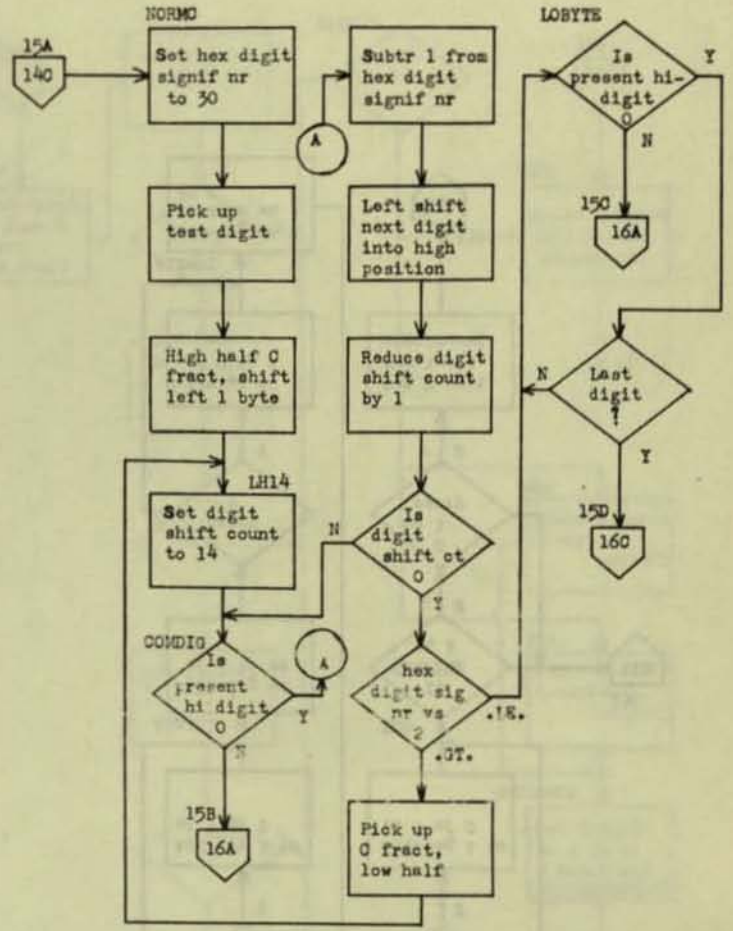


Fig. 15. Multipr Add/Subtr Add-logical Sequence, Normalization, Part 1

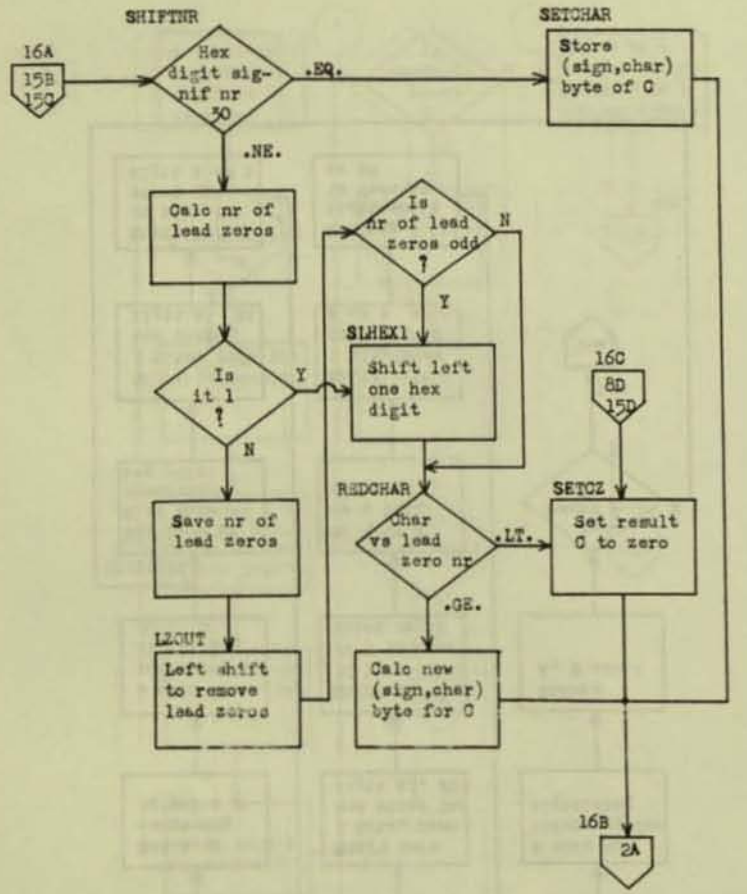


Fig. 16. Multipr Add/Subtr Add-logical Sequence, Normalization, Part 2

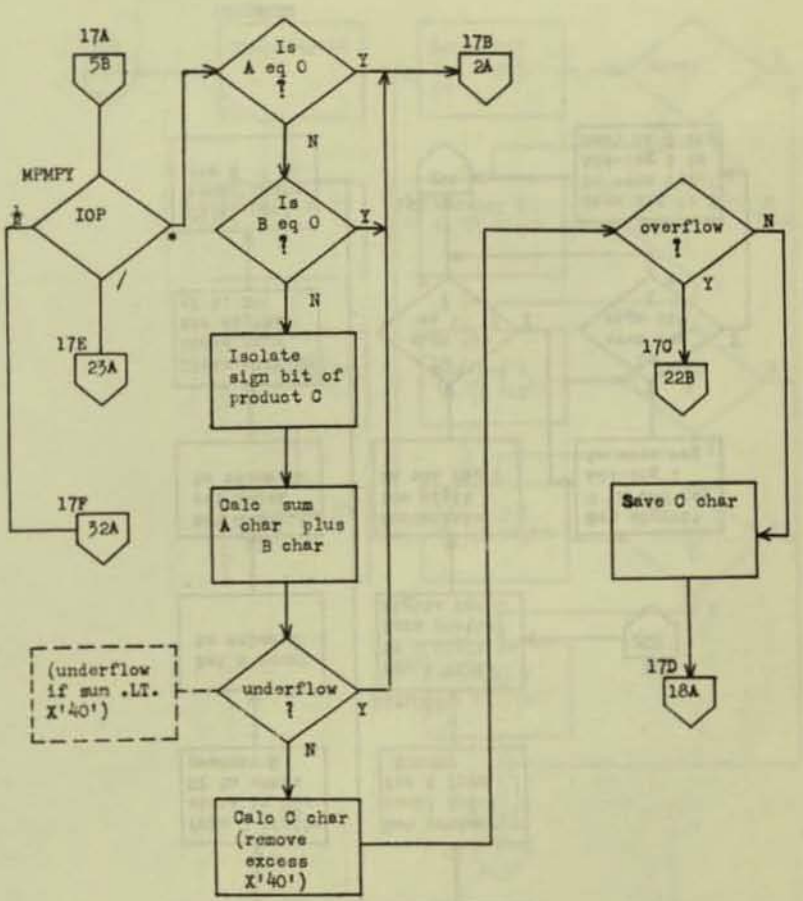


Fig. 17. Multipr Multiply, Prelim and Result-Characteristic Calculation

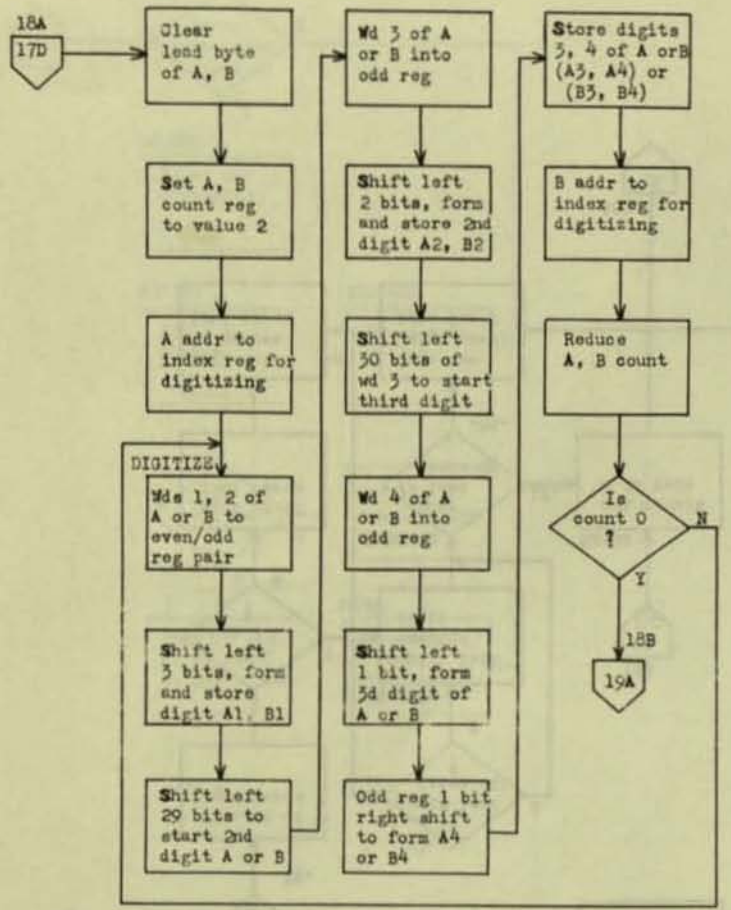


Fig. 18. Multiprecision Multiply, Fraction Digitizing

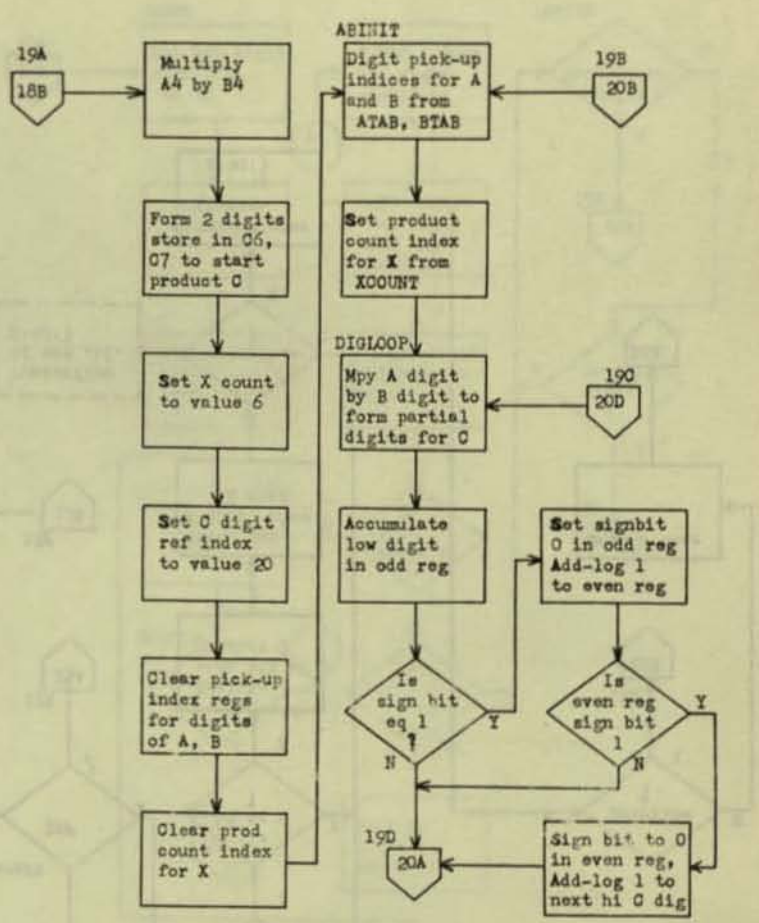


Fig. 19. Multipr Multiply, Digit Multiplication, Accumulation, Part 1

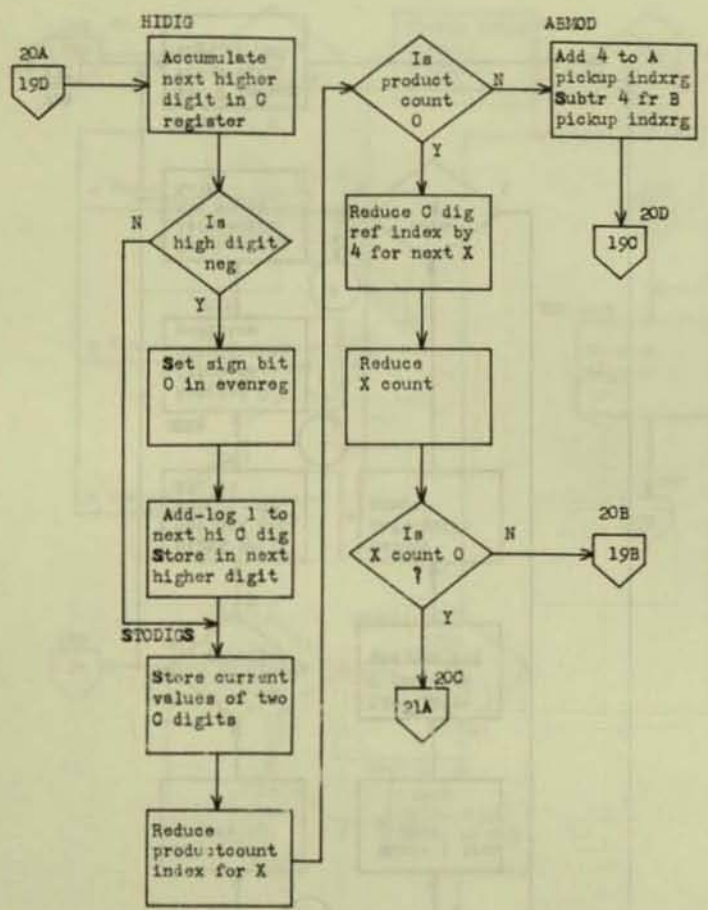


Fig. 20. Multipr Multiply, Digit Multiplication, Accumulation, Part 2

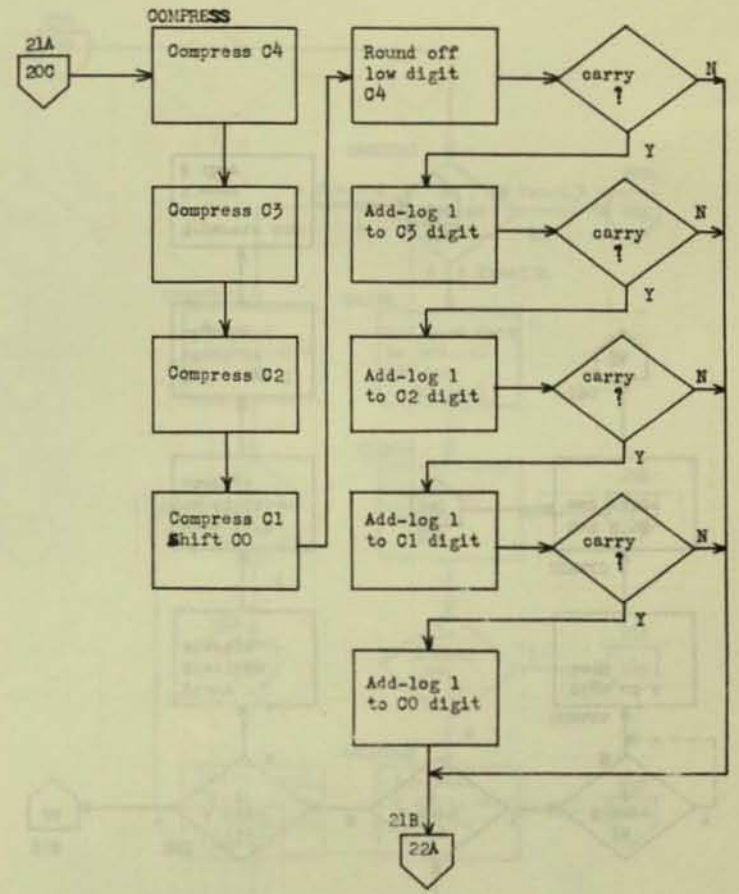


Fig. 21. Multipr Multiply, Product Compression into Fraction Form

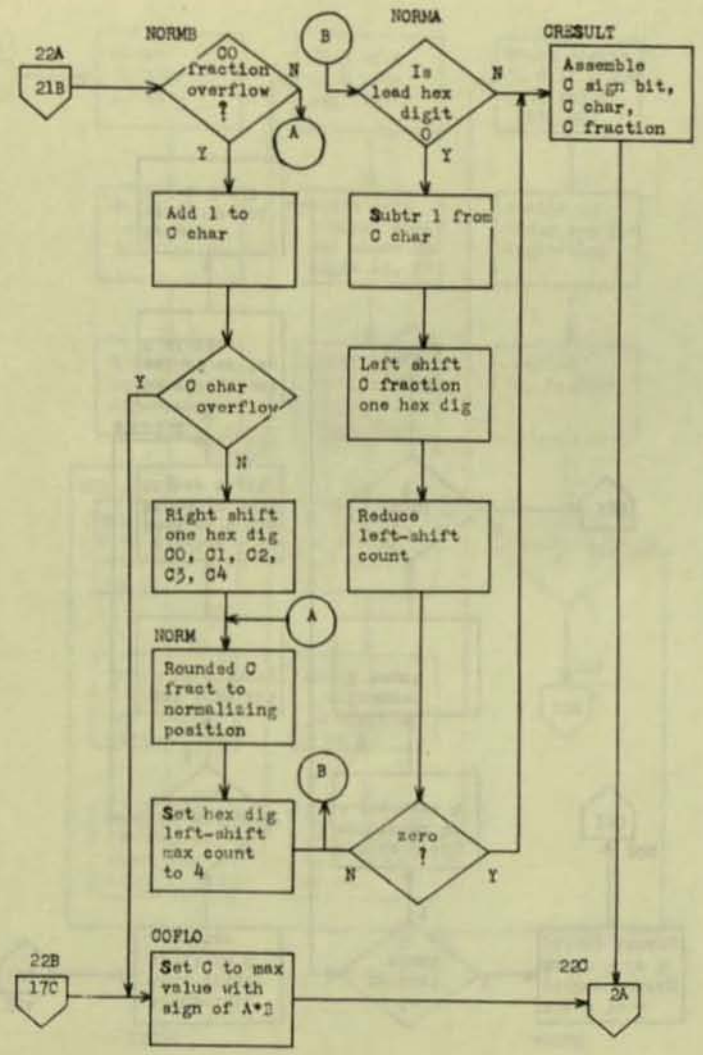


Fig. 22. Multiprecision Multiply, Result Formation

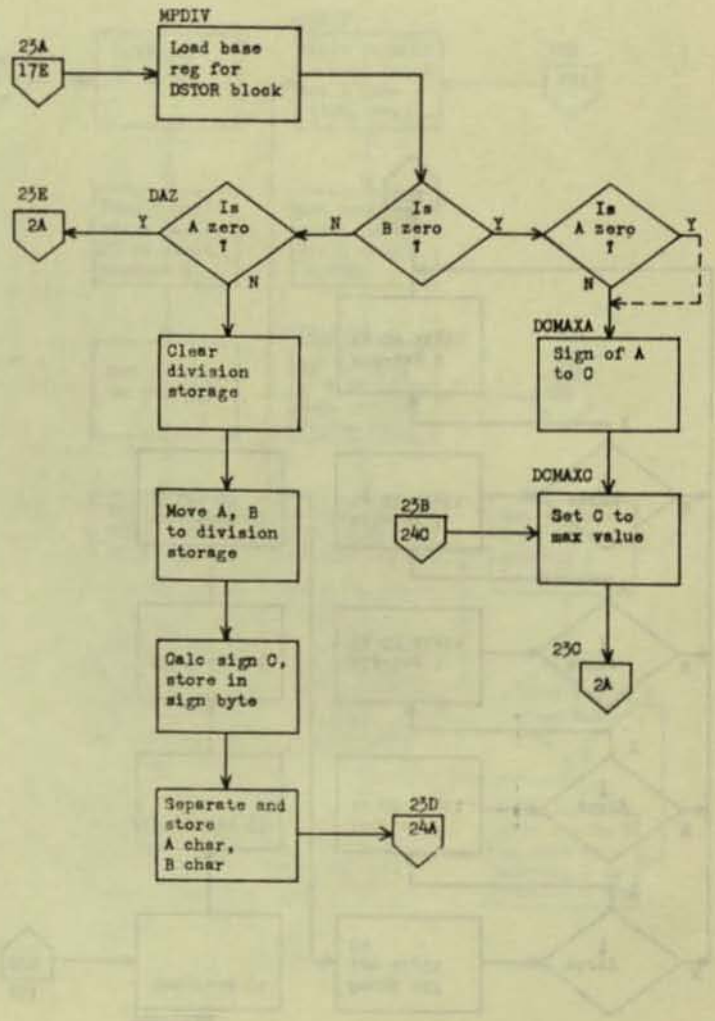


Fig. 23. Multiprecision Division Preliminary

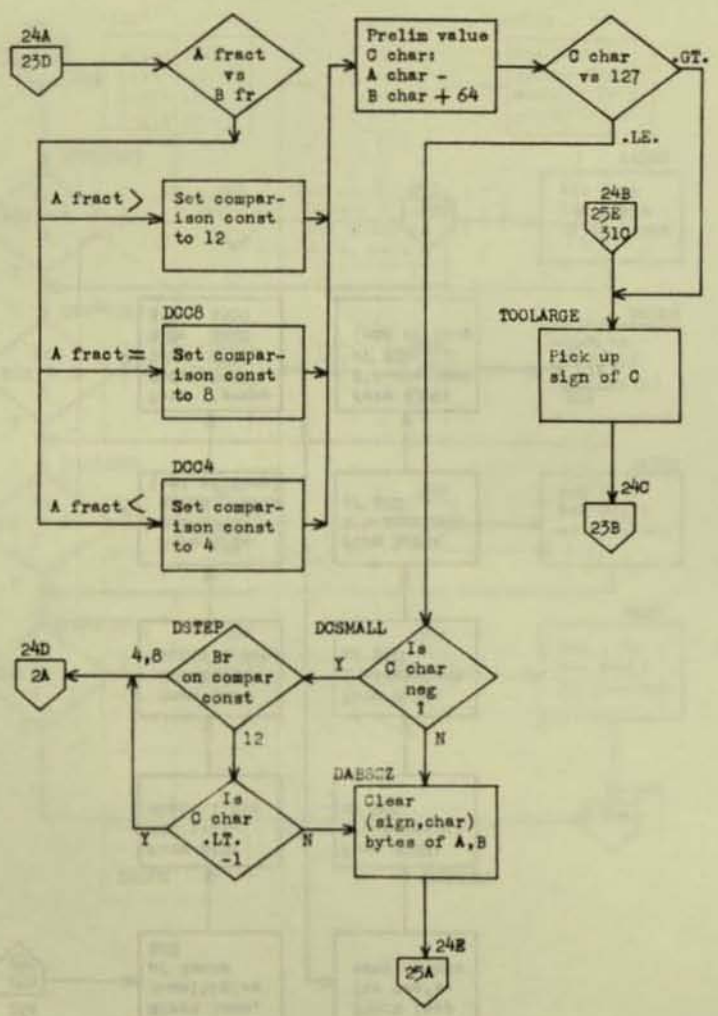


Fig. 24. Multipr Division: Fraction Comparison, Characteristic Calc

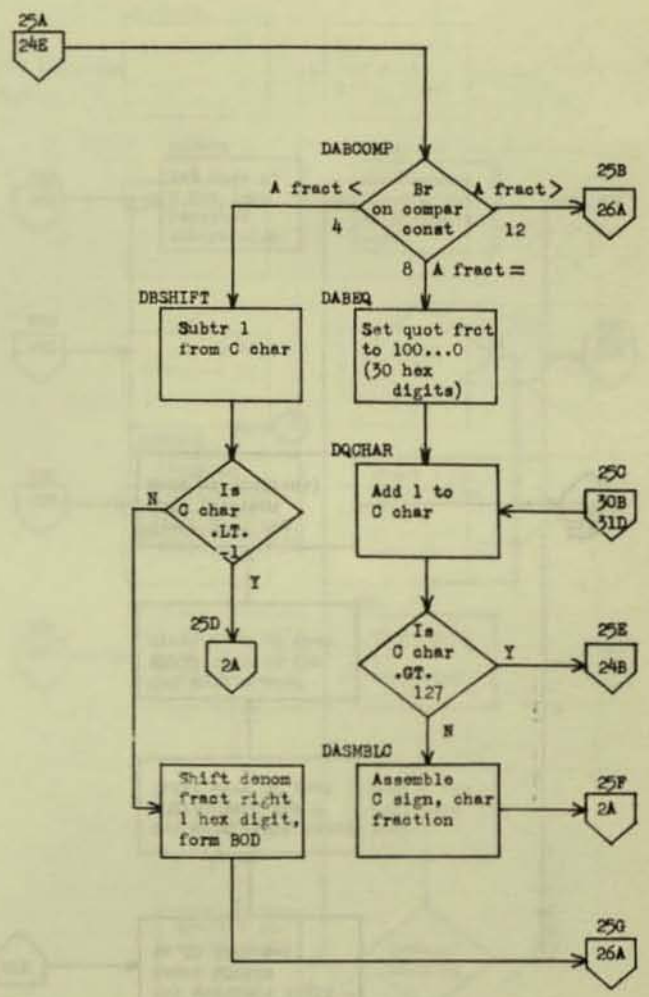


Fig. 25. Multipr Division: Equal Fractions; Denominator Preparation

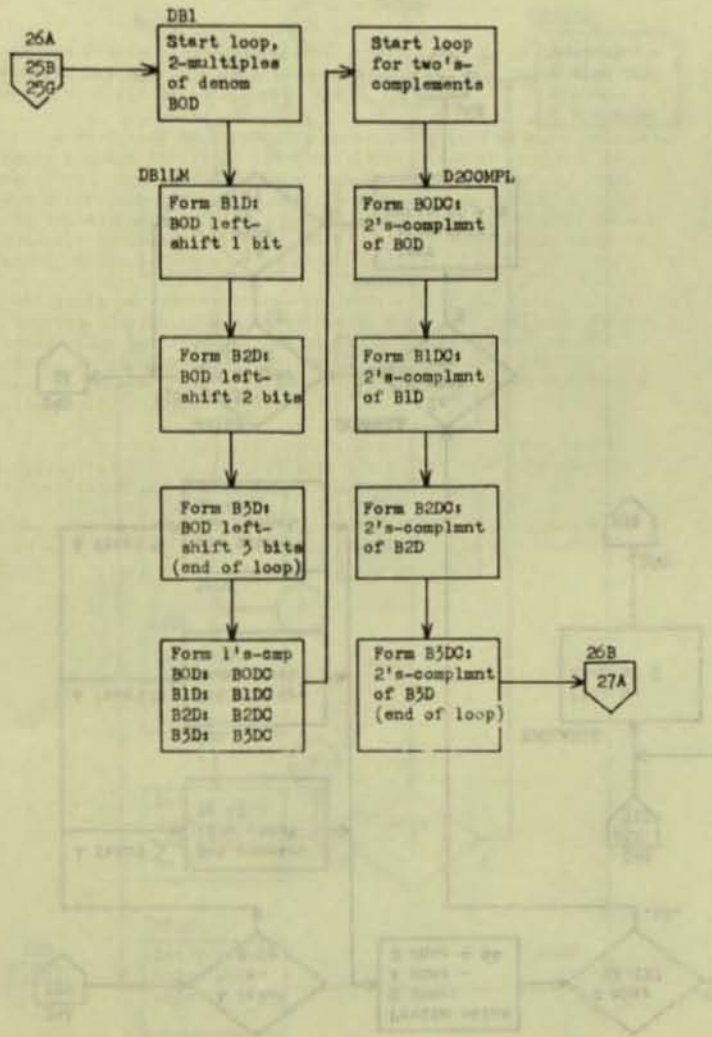


Fig. 26. Multipr Division: Two-multiples, Complements of Denominator

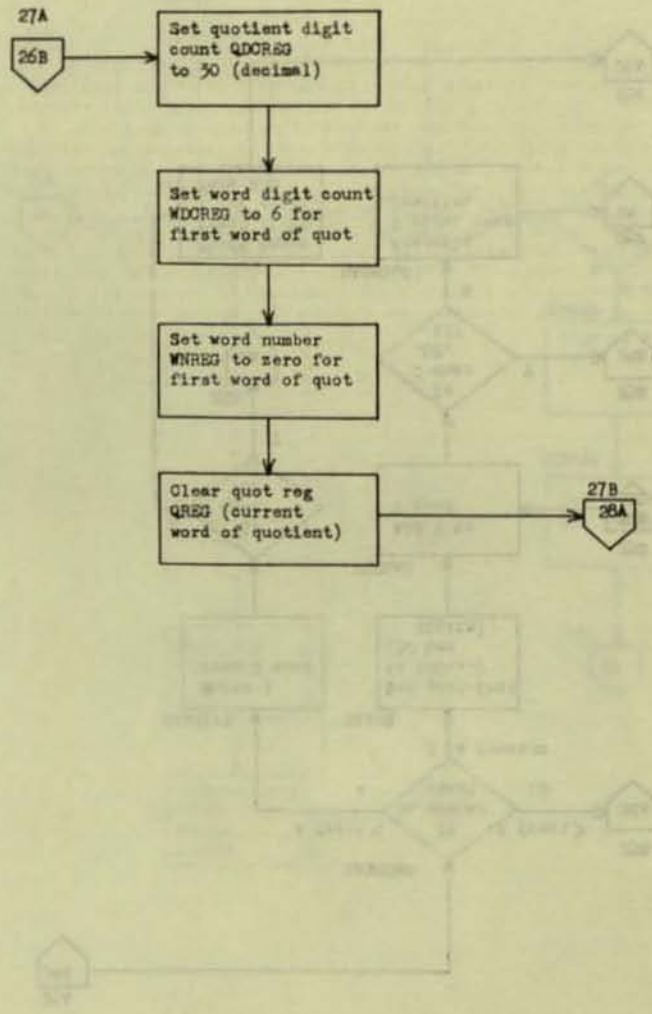


Fig. 27. Multipr Division: Repeated Subtraction Initialization

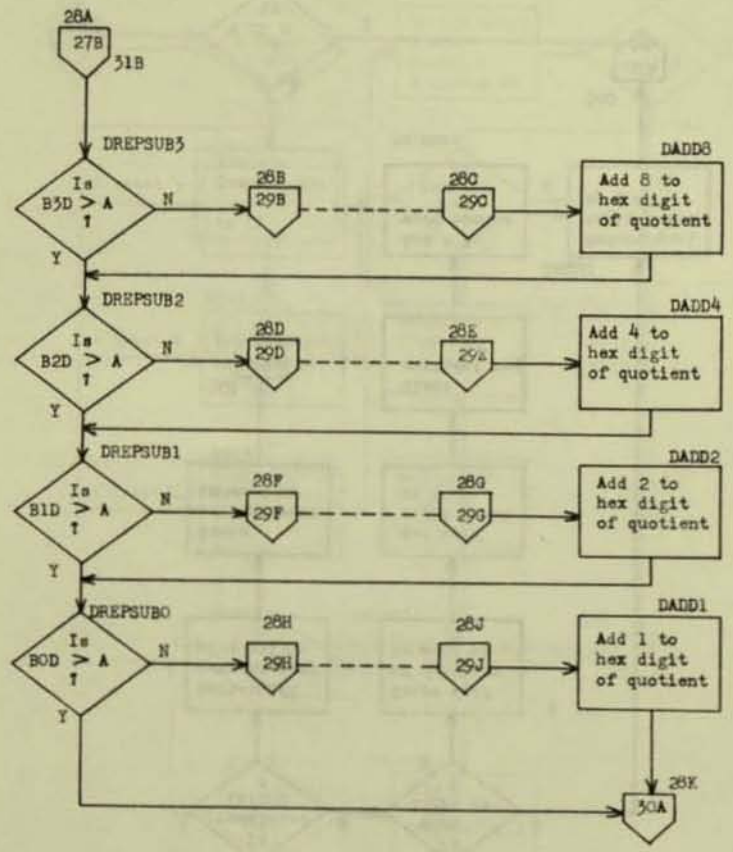


Fig. 28. Multipr Division: Two's-Multiple Selection for Digit Formation

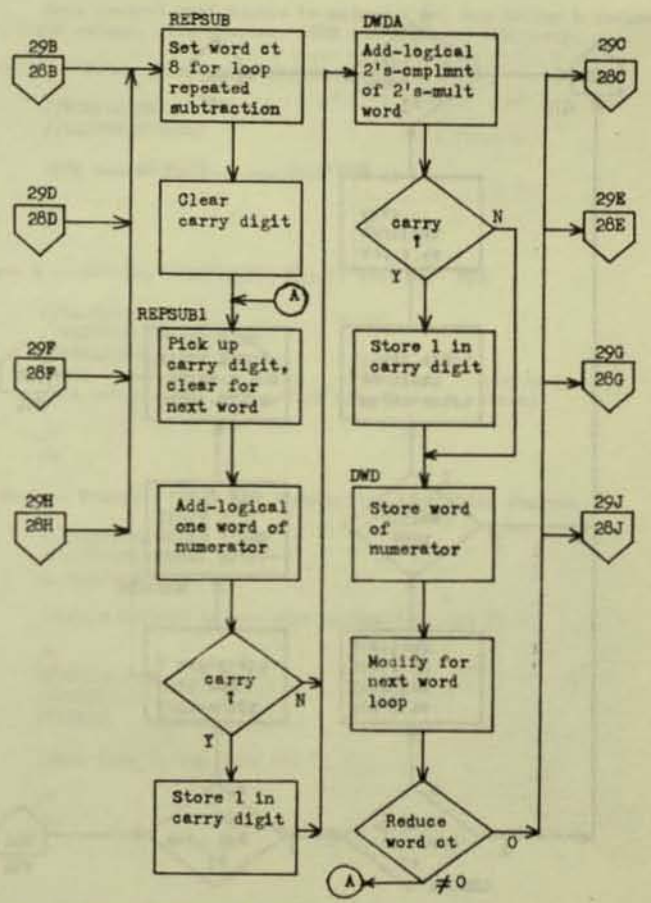


Fig. 29. Multipr Division: Repeated Subtraction Loop

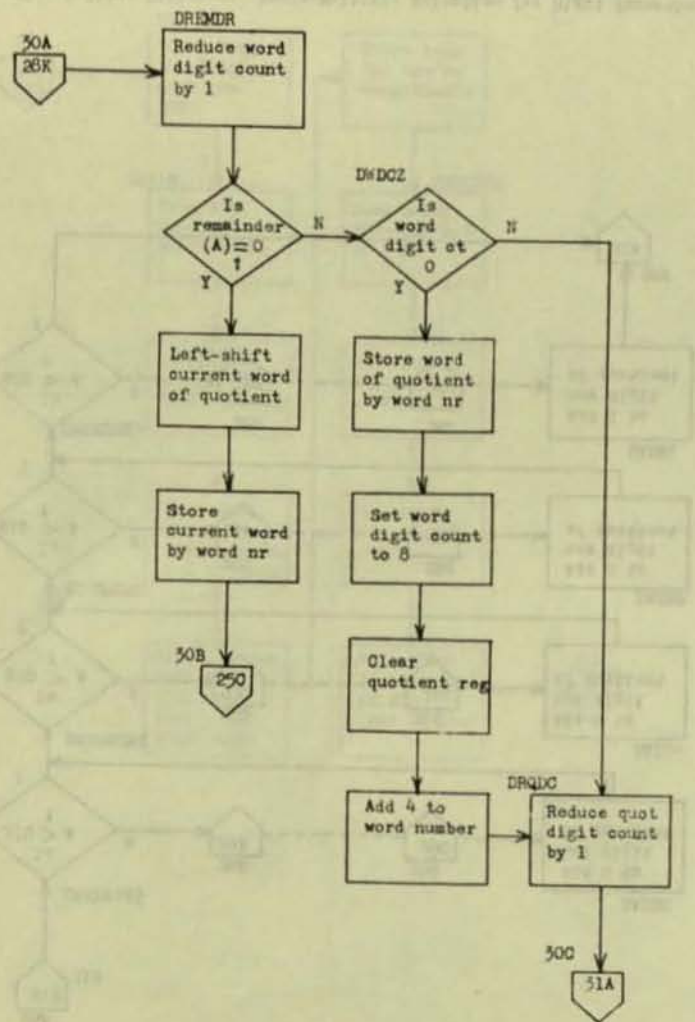


Fig. 30. Multipr Division: Storage of Partial Quotient

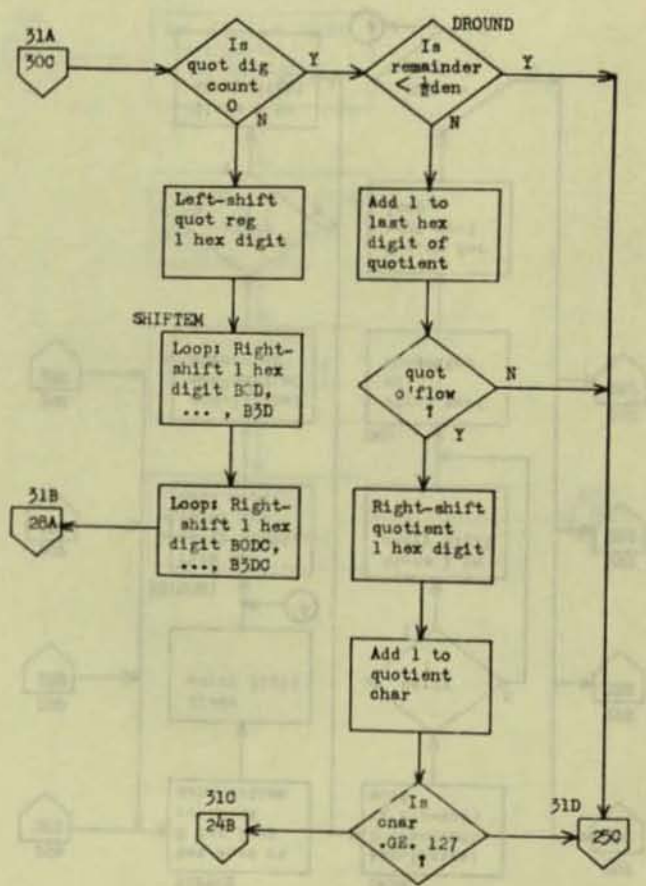


Fig. 31. Multipr Division: Shift for Next Quot Digit, or Round-off

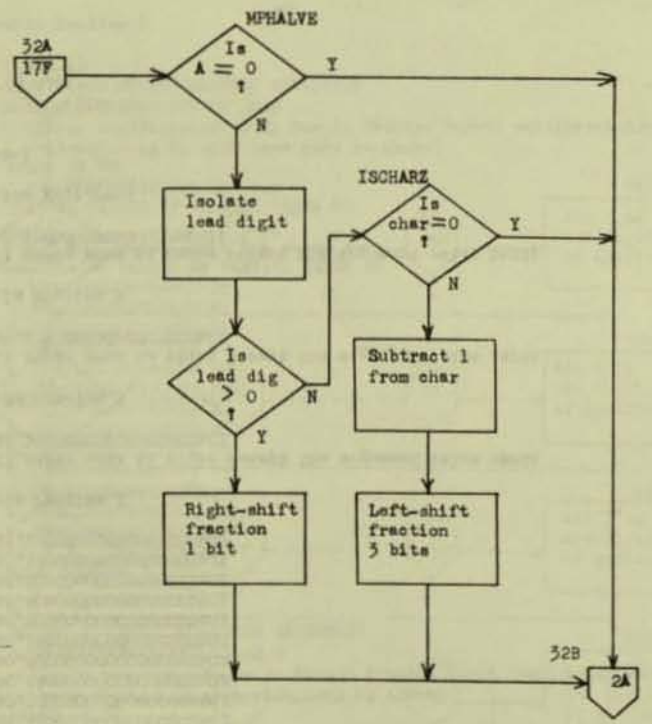


Fig. 32. Multiprecision Halve Operation

SAMPLE PROBLEM EXECUTION

Control Card and Program Deck Sequence

The following sequence of control cards, source and relocatable program decks was used to obtain a successful execution of the sample FORTRAN program on an IBM System/360 Model 30, using DOS/360.

Each control card begins in column 1 and the letter b indicates a blank column. The jobnames ABCD and EFGH are arbitrary.

1) Phase A -- Punch relocatable object deck for MPA

```
//bJOBbABCD
//bEXECbASSEMBLY

(MPA source deck -- see DECK KEY 1) )

/*
/&
```

2) Phase B -- Catalog relocatable object deck for MPA

```
//bJOBbABCD
//bEXECbMAINT
bCATALRbMPA

(MPA relocatable object deck from Phase A, above)

/*
/&
```

3) Phase C -- Compile, link, and Execute Sample FORTRAN Program

```
//bJOBbEFGH
//bOPTIONbLINK
//bEXECbFORTRAN

(sample FORTRAN source deck -- see DECK KEY 2) )

/*
bINCLUDEbMPA
//bEXECbLNKEDT
//bEXEC

(data deck -- see DECK KEY 3) )

/*
/&
```

Sample Problem Input

Each of the four sample problems included contains a data deck of 15 cards. The first card contains the degree N of the polynomial and is read by an I5 format. In each sample N is 12. The next 13 cards each contain a coefficient, read by a D24.16 format. The last card contains the argument value, also read by a D24.16 format. In the listings given below, each line represents the data on a card and b indicates a blank column.

1) Sample Problem 1

```
bbb12
bb0.1000000000000000Db01
bb0.2000000000000000Db01
bb0.4000000000000000Db01
bb0.8000000000000000Db01
bb0.1600000000000000Db02
bb0.3200000000000000Db02
bb0.6400000000000000Db02
bb0.3200000000000000Db02
bb0.1600000000000000Db02
bb0.8000000000000000Db01
bb0.4000000000000000Db01
bb0.2000000000000000Db01
bb0.1000000000000000Db01
bb0.1250000000000000Db00
```

2) Sample Problem 2

All cards same as above except the argument value card:

```
bb0.1000000000000000Db01
```

3) Sample Problem 3

All cards same as above except the argument value card:

```
bb0.1250000000000000Db01
```

4) Sample Problem 4

All cards same as above except the argument value card:

```
b-0.1000000000000000Db01
```

5) Problem Exit Card

```
bbb-1
```


Sample Problem Output

Each of the four sample problems whose input is listed in the preceding section produced approximately a page of output, as listed below for the first problem. One-third of this output is the input printed for the record; the second third is the initial values of the partial terms in the synthetic division, to record that those values were indeed zero. The last third is a listing of the double-precision portions of the partial terms obtained in the synthetic divisior. The last value printed is the value of the given polynomial for the given argument, to 16 decimal digits. In the first and third sample problems this last line includes in parentheses the 17th through 28th digits obtained by an equivalent program using IBM 1620 FORTRAN II. Since each problem uses the same coefficients they are not listed below, but appropriate reference is made to their values as listed in the section Sample Problem Input. Also the initial values of the partial terms in the synthetic division are not all listed but are referred to. The calculated values of the partial terms are listed completely. Leading blanks are not indicated.

1) Sample Problem 1

```
N =   12
COEFFS AND MULTIPRECISION EXTENSION
0.1000000000000000D 01  0.0
  (Other coefficients as in Sample Problem Input; multiprecision
   extension is in each case zero as above)
VALUE OF PX
0.1250000000000000D 00  0.0
INITIAL VALUES OF PARTIAL TERMS PC
0.0                     0.0
  (12 more terms as in the above line)
CALCULATED VALUES OF PARTIAL TERMS PC
0.1000000000000000D 01
0.2125000000000000D 01
0.4265625000000000D 01
0.8533203125000000D 01
0.1706665039062500D 02
0.3413333129882812D 02
0.6826666412353516D 02
0.4053333301544189D 02
0.2106666662693024D 02
0.1063333332538605D 02
0.5329166666045780D 01
0.2666145833525720D 01
0.1333268229165696D 01   (538984775543)
```

2) Sample Problem 2

```
N =  12
COEFFS AND MULTIPRECISION EXTENSION
0.10000000000000000D 01  0.0
  (Other coefficients as in Sample Problem Input; multiprecision
   extension is in each case zero as above)
VALUE OF PX
0.10000000000000000D 01  0.0
INITIAL VALUES OF PARTIAL TERMS PC
0.0                      0.0
  (12 more terms as in the above line)
CALCULATED VALUES OF PARTIAL TERMS PC
0.10000000000000000D 01
0.30000000000000000D 01
0.70000000000000000D 01
0.15000000000000000D 02
0.31000000000000000D 02
0.63000000000000000D 02
0.12700000000000000D 03
0.15900000000000000D 03
0.17500000000000000D 03
0.18300000000000000D 03
0.18700000000000000D 03
0.18900000000000000D 03
0.19000000000000000D 03
```

3) Sample Problem 3

```
N =  12
COEFFS AND MULTIPRECISION EXTENSION
0.10000000000000000D 01  0.0
  (Other coefficients as in Sample Problem Input; multiprecision
   extension is in each case zero as above)
VALUE OF PX
0.12500000000000000D 01  0.0
INITIAL VALUES OF PARTIAL TERMS PC
0.0                      0.0
  (12 more terms as in the above line)
CALCULATED VALUES OF PARTIAL TERMS PC
0.10000000000000000D 01
0.52500000000000000D 01
0.80625000000000000D 01
0.18078125000000000D 02
0.38597656250000000D 02
0.80247070312500000D 02
0.16430883789062500D 03
0.23738604736328125D 03
0.31273559204101562D 03
0.39891569900512695D 03
0.50264462375640869D 03
0.63030577969551086D 03
0.78888222461938858D 03   (803222656250)
```


4) Sample Problem 4

```
N =  12
COEFFS AND MULTIPRECISION EXTENSION
0.10000000000000000D 01  0.0
  (Other coefficients as in Sample Problem Input; multiprecision
   extension is in each case zero as above)
VALUE OF PX
-0.10000000000000000D 01  0.0
INITIAL VALUES OF PARTIAL TERMS PC
0.0                       0.0
  (12 more terms as in the above line)
CALCULATED VALUES OF PARTIAL TERMS PC
0.10000000000000000D 01
0.10000000000000000D 01
0.30000000000000000D 01
0.50000000000000000D 01
0.11000000000000000D 02
0.21000000000000000D 02
0.43000000000000000D 02
-0.11000000000000000D 02
 0.27000000000000000D 02
-0.19000000000000000D 02
 0.25000000000000000D 02
-0.21000000000000000D 02
 0.22000000000000000D 02
```

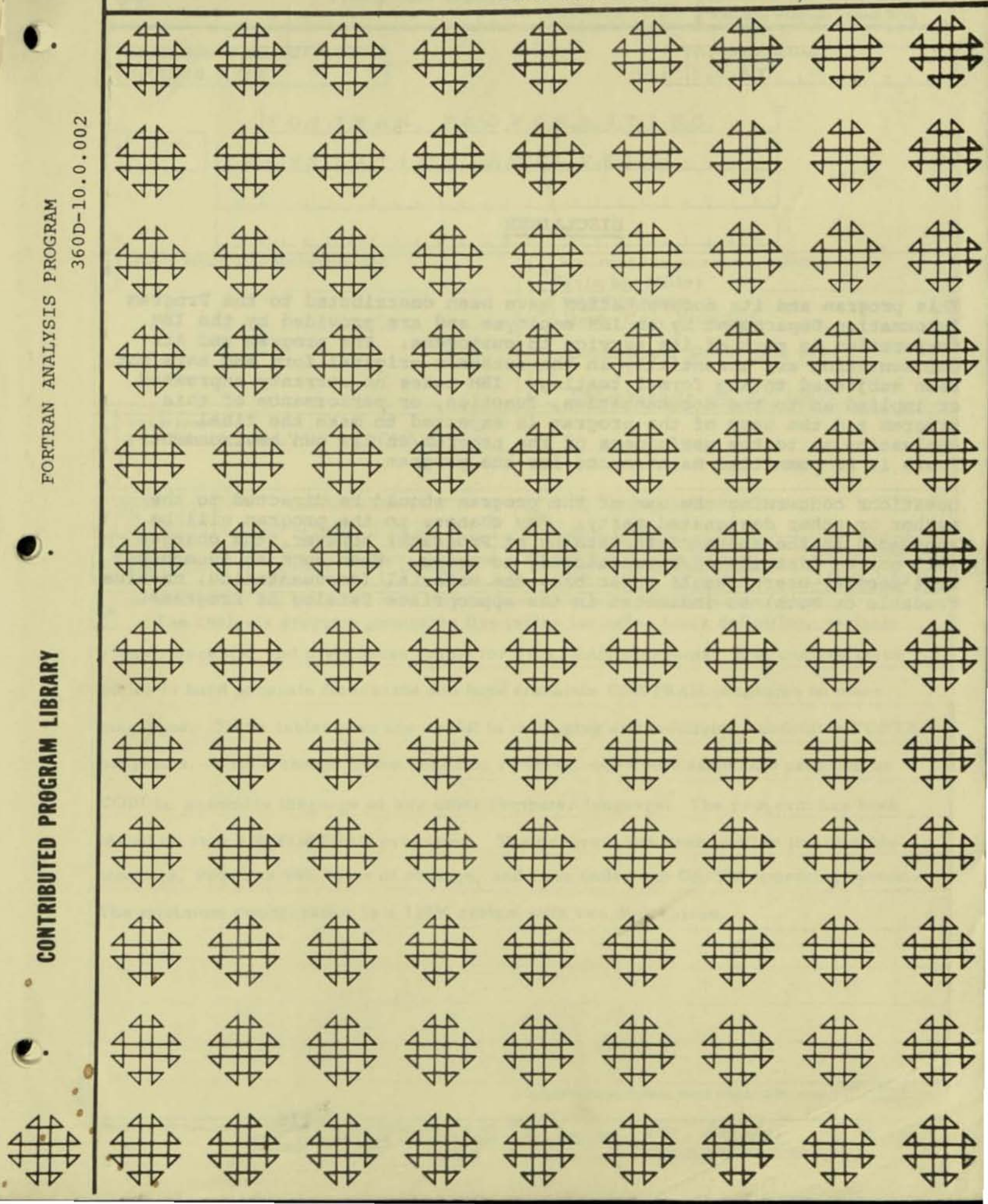
FORTRAN ANALYSIS PROGRAM

Master Copy-
Return to Linda Lorenzetti
B197

360D-10.0.002

FORTRAN ANALYSIS PROGRAM
360D-10.0.002

CONTRIBUTED PROGRAM LIBRARY



DISCLAIMER

This program and its documentation have been contributed to the Program Information Department by an IBM employee and are provided by the IBM Corporation as part of its service to customers. The program and its documentation are essentially in the author's original form and have not been subjected to any formal testing. IBM makes no warranty expressed or implied as to the documentation, function, or performance of this program and the user of the program is expected to make the final evaluation as to the usefulness of the program in his own environment. There is no committed maintenance for the program.

Questions concerning the use of the program should be directed to the author or other designated party. Any changes to the program will be announced in the appropriate Catalog of Programs; however, the changes will not be distributed automatically to users. When such an announcement occurs, users should order only the material (documentation, machine readable or both) as indicated in the appropriate Catalog of Programs.

Program Contribution Form

Type III (IBM Employee)

IBM Corporation
Program Information Department (PID)
40 Saw Mill River Road
Hawthorne, New York 10532, U.S.A.
Attention: Program Control Desk

1 PROGRAM ORDER NUMBER (TO BE FILLED IN BY PID): 360D-10.0.002

2 SYSTEM TYPE (MACHINE): 360D

3 SEARCH KEY: FORTRAN FLOWCHARTING; TRANSLATION ANALYSIS

4 AUTHORS' NAME(S) (IF DIFFERENT THEN SUBMITTER'S)

5 SUBMITTER'S NAME (DIRECT TECHNICAL INQUIRIES TO): Irvin M. Miller

6 SUBMITTER'S ADDRESS: Dept. C40, Bldg 703
South Road, P. O. Box 390
Poughkeepsie, N. Y. 12601

7 TITLE OF PROGRAM: FORTRAN ANALYSIS PROGRAM

8 PRIM.SUB. CODE: 10.0

9 SECONDARY SUBJECT CODES: 12.2 | 00.2 | 04.2 |

10 OPERATING OR MONITOR SYSTEM REQUIRED: OS

11 TYPE OF SUBMITTAL PLEASE CHECK (√): 1 INITIAL PROGRAM ABSTRACT | 2 INITIAL SUBMISSION OF PROGRAM | 3 RESUBMITTAL OF UNANNOUNCED PROG. | 4 CORRECTION TO PROGRAM NUMBER

12 DATE OF SUBMITTAL: 12/31/68

13 ABSTRACT (PLEASE LIMIT TO 150 WORDS. DESCRIBE PROGRAM AND PURPOSE. CLEARLY IDENTIFY MACHINE CONFIGURATION AND SOURCE LANGUAGE.)

The analysis program generates five tables including block definition, variable cross reference, and precedence tables for FORTRAN programs. One can use these tables to hand generate flowcharts and hand translate FORTRAN programs to other languages. These tables also are useful in debugging and modifying executable FORTRAN programs. With a change in the scanning routines, one could adapt this program to COBOL, assembly language or any other computer language. The program has been tested on over 100 FORTRAN programs. The program has been written in assembly language, requires 90K bytes of storage, and runs under any OS/360 operating system. The minimum configuration is a 128K system with two disk drives.

PLEASE ATTACH ADDITIONAL PAGES IF NECESSARY → TOTAL PAGES ATTACHED

120-1424-3

PERMISSION TO PUBLISH: "I HEREBY GIVE IBM PERMISSION TO REPRINT, REPRODUCE AND DISTRIBUTE THIS PROGRAM TO ANYONE."

14 Irvin M. Miller 12/30/68
SIGNATURE OF SUBMITTER AND DATE

TABLE OF CONTENTS

ABSTRACT DETAIL

The Program-Analysis-Program with the FORTRAN scan is designed to provide tables for the translation and debugging of FORTRAN programs into other computer languages. Also, the tables present information which will assist in measuring the degree of parallelism within the FORTRAN program.

A block in this paper is defined as the lines of code between two consecutive entry points or between an entry point and a branch. The object of a branch defines an entry point.

The two basic tables are the Block Definition Table and the Variables Used in Blocks Table. The Block Definition Table delimits the functional blocks of code and lists the transfers of control from each block; the Variables Used in Blocks Table lists the variables and the blocks in which they appear.

The Block Definition Table is complemented by the Branch Cross Reference Table, which lists the transfer of control to each block. The Variables Used in Blocks Table is complemented by the Blocks Referenced By Variables Table, which lists the blocks and the variables appearing in them. The Precedence Diagram Table is a supplementary table which graphically displays the inter-dependencies among the blocks.

Description of Program

The Program-Analysis-Program (FAP) with the FORTRAN scan routines (PASS 1 and PASS 2) is designed to analyze FORTRAN programs and produce as output, five tables, including a Precedence Diagram Table, and a listing of the FORTRAN program with a sequence or statement number for each statement, except JCL and comment cards. At the present, there are three applications of the Program Analysis Program:

1. Manual translation
2. Debugging
3. Program modification

In manual translation, FAP provides tables that one might use in hand translation of his program from one computer language to another.

In the second application, debugging, this program enables the user to be fully aware of the interdependencies of variables, as well as the flow of logic, in the analyzed program. Thus, if the user knows how and when variables are used, and from where blocks of code receive control, then the task of debugging will be easier.

For the third application, if the user wishes to modify his program, the Program-Analysis-Program will indicate to the user the other variables or blocks of coding which may be affected by the change in his program. The user's awareness of the dependencies of variables and blocks of coding in the analyzed program will also reduce the time of debugging these modifications.

Routine PASS 1 processes the source code for a complete FORTRAN program and sets up the intermediate text for PASS 2. PASS 2 then processes each intermediate text record by scanning each record for labels and symbols. For each symbol, a temporary I. D. number is assigned by LABDIC and entered in a utility symbol table. When PASS 2 completes scanning the entire intermediate text, the basic tables, Block Definition and Variables Used in Block, have been built, and the remaining three tables, Branch Cross Reference, Blocks Referenced by Variables, and Precedence Diagram, are built from the first two (see Figure 1 - Systems Flowchart).

The first two tables are built from information passed directly from the scanning routine, PASS 2. The Branch Cross Reference and Block Referenced by Variables are inversions of the first two, respectively. Lastly, the Precedence Diagram Table, a supplementary table, is derived from information concerning the two inversion tables.

The Block Definition Table (routine BDT) defines the functional blocks of code and the transfer of control from those blocks in the analyzed program. The criteria for defining blocks are determined by the scanning routine PASS 2 (see routine PASS 2, page 20). Depending upon the conditions concerning the transfer of control, it is possible for BDT to find both branch and destination points, or either one of the two, within one block. In the case of the destination point, control falls through to the next functional block. These branch and destination points are given in terms of statement numbers, as well as the delimiters of the blocks. However, BDT's complementary table, Branch Cross Reference, uses the resolved block numbers to indicate the transfer of control to a block. The major difference in these two tables is that BDT gives each block and the other blocks to which it transfers control; whereas, the Branch Cross Reference (CROSS routine) gives all blocks which transfer control to each particular block. These two tables would aid the user in debugging because both give the logic flow of the analyzed program.

The second basic table, Variables Used by Block Table, (routine VARUSE), lists all variables according to the block numbers. The Block Referenced by Variables Table (routine BRVBLD), based upon the Variables Used by Block Table, lists for each variable, all blocks in which it appears. If the user wishes to modify the usage of variables in certain functional blocks, referencing one of these two tables would ease the effort in making modification.

The last table to be considered is the Precedence Diagram Table (routine PREC). This table shows the dependencies among blocks, or groups of blocks, based on their branching order which is derived from the Block Definition Table, and on the usage of variables derived from the Block Referenced by Variables Table. The approximate time of execution for each block, using an OS/360 MOD 50, is included in the table. The precedence Diagram Table indicates the degree of parallelism among functional blocks within the FORTRAN program by using the Block Definition and Block Referenced by Variables Tables to determine which blocks are dependent upon other blocks.

In summary, the Program-Analysis-Program with the FORTRAN scan produces five tables to aid the user in debugging and modification of FORTRAN programs, manual translation of programs into other languages, and determination of the degree of parallelism within the FORTRAN program.

CHARACTERISTICS

The following describes the limitations of the program and its execution characteristics.

Table sizes create the following limitations:

BRVBLD Table	1250	Entries (8 bytes per entry)
BRVBLD Table Index	500	Entries (8 bytes per entry)
VARUSE Table	8000	Bytes
VARUSE Table Index	150	Elements (4 bytes per element)
BDT Table	3200	Bytes
BDT Table Index	150	Elements (4 bytes per element)
LABDIC Table	11000	Bytes
LABDIC Table Index	600	Elements (8 bytes per element)
CROSS Table	600	Elements (8 bytes per element)
CROSS Table Index	150	Elements (4 bytes per element)
PREC Table	250	Elements (16 bytes per element)
PREC Table Index	150	Elements (4 bytes per element)

The individual routines are designed so that to change a table size only the DS instruction for that particular table need be changed.

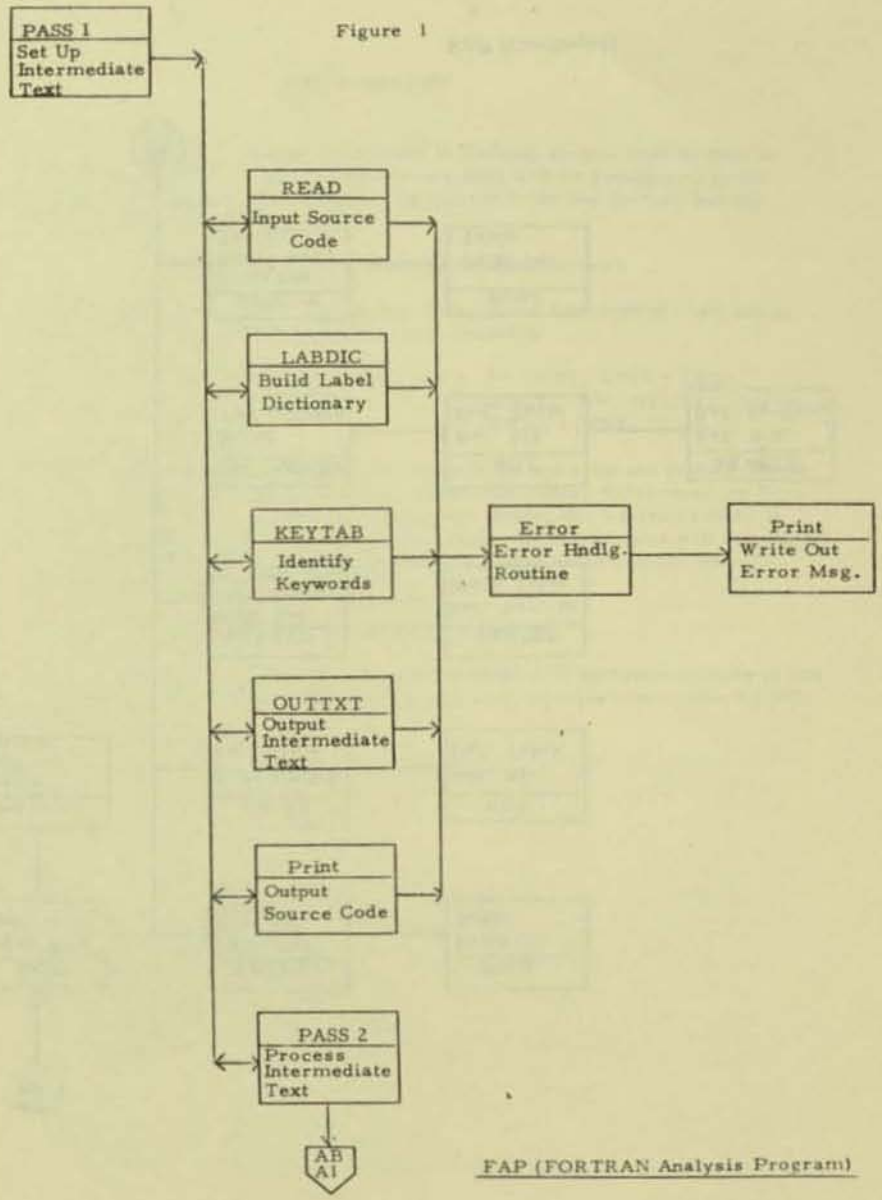
Error messages will be printed for any of the above table routines if overflow occurs. Error messages are also printed when there is an invalid symbol or code passed between two routines or when something to be referenced does not exist. Another type of error message is an abnormal operation message from the I/O routine.

The Keyword Table, which is referenced by the scanning routine, is limited to only basic FORTRAN IV keywords and could easily be altered by the addition or subtraction of entries.

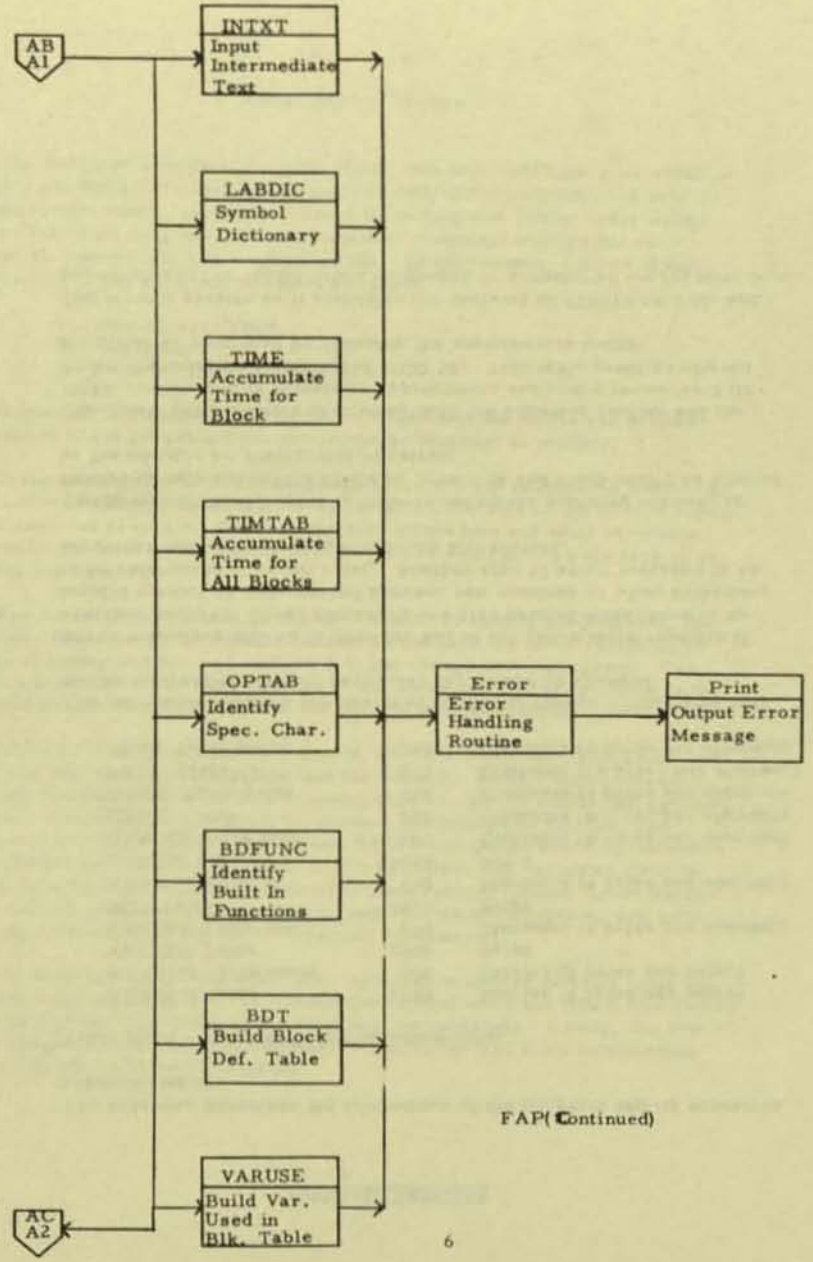
The TIME Table routine is used by both the scanning routine and the PREC routine and is limited to approximate statement times for FORTRAN statements on an OS/360 MOD 50. This table could be altered for different machines by changing the approximate times.

The overall system as it stands is for analysis of FORTRAN code and can be changed to handle other languages by adopting an appropriate scan.

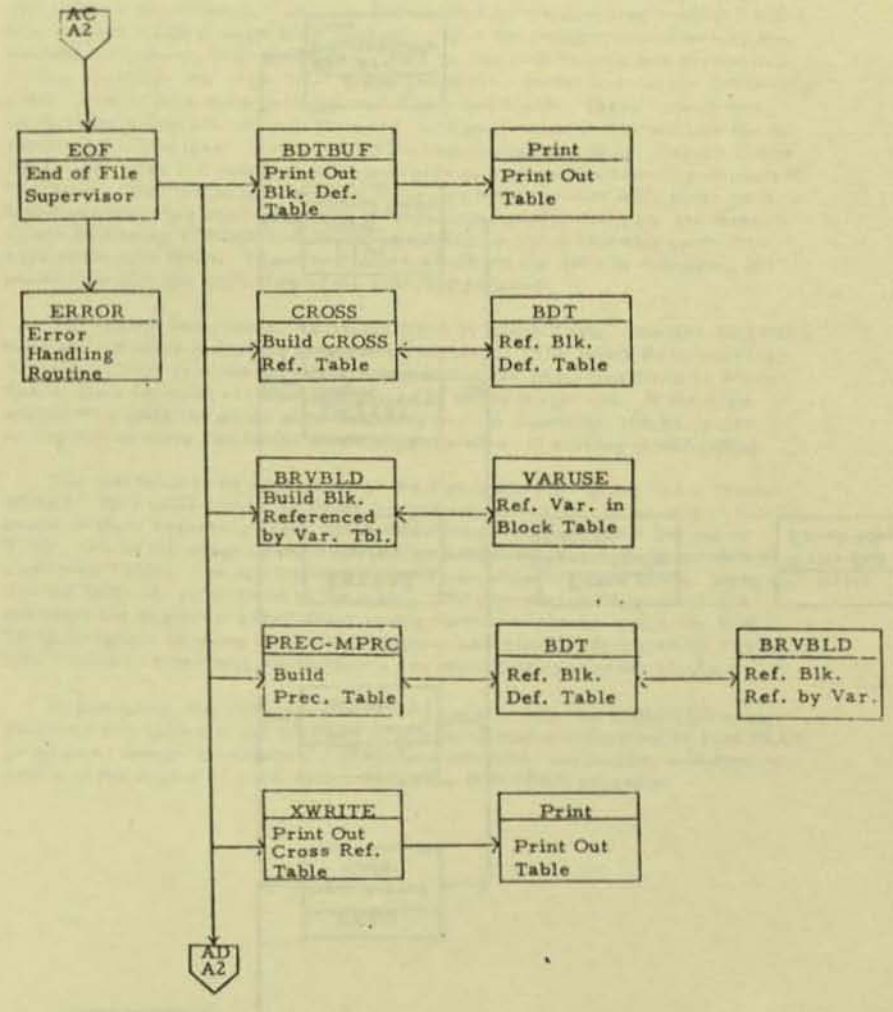
Figure 1



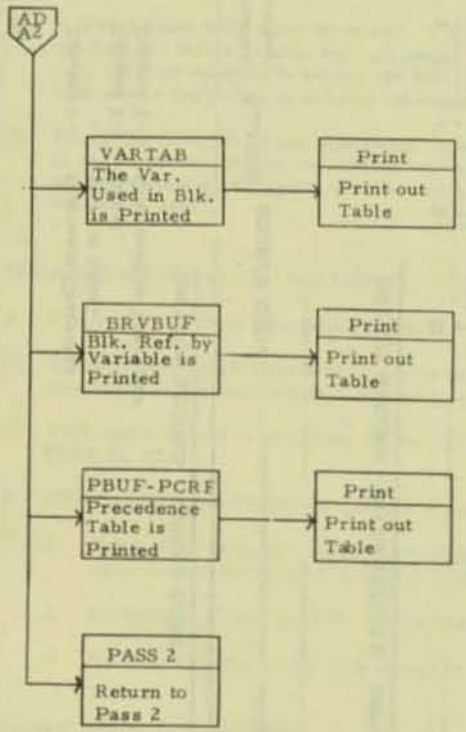
FAP (FORTRAN Analysis Program)



FAP (Continued)



FAP (Continued)



FAP (Continued)

INPUT-OUTPUT

The basic input to the system is Fortran source code written in basic Fortran IV. Only Fortran source code will be recognized by the scanning routines. The input can be entered from any device, but the following JCL must be used:

```
// GO. INPUT     DD ('input device parameters')
```

The system also needs a temporary data set for intermediate text set up by PASS 1. The following JCL must be included.

```
// GO. OUTTXT  DD  DSNAME = && TEXT, UNIT = 2311,
                           DISP = (NEW, DELETE, DELETE),
                           SPACE = (TRK, (1, 1))
```

The main purpose of the system is to build up and print out tables. These tables include the Block Definition Table, the Blocks Referenced By Variables Table, the Branch Cross Reference Table, the Variables Used In Block Table, and the Precedence Diagram. All these tables will be printed out in a formatted listing along with the Fortran source code. The JCL necessary for this procedure is

```
// GO. OUTPUT   DD  SYSOUT = A
```

The system has been designed to make I/O operation as easy to use as possible. These 3 cards are the only ones necessary to handle all I/O.

This program has been designed to run under OS/360, PCP, MFT, or MVT. In order to run under other systems, all routines dealing with INPUT/OUTPUT would have to be changed. These three routines comprise the only significant changes that would have to be made.

If the user desires, other programming languages could be analyzed using parts of this system and parts of user supplied programs. The changes would be the following:

1. Scan Routines - To scan other languages and interface correctly with all other routines
2. Scan Utility Routines - These include the Keyword Table (KEYTAB), Operator Table (OPTAB), Time Tables (TIMTAB), and Build-in - Function Tables (BDFUNC). These would have to be changed to aid the new scan routines.

Because the TIME routines reflect approximations to Fortran IV basic language on the MOD 50, these would have to be updated if execution is to be done on a different machine.

Operating Instructions

To use the program the following must be done:

I Ready operating system, OS/360

A. A disk labeled llllll must be online. If a different label is desired, the fourth, sixth, and eleventh cards of the PID tape must be changed to reflect the label available on the disk where the program is to be cataloged.

B. Punch the first file of the tape using a program equivalent to the one described on the section tape key. There should be approximately 810cards. Remove the blank cards at the end of the deck.

II Cataloging and execution of test cases.

A. Place punched deck into card reader.

B. A disk labeled llllll should be online or its equivalent with two cards in the punched deck replaced to reflect the change.

C. Disk space should be available for the output of &&TEXT (TRK, (1, 1)).

D. START RDR, and START to execute three programs.

1. Purge FAP from the disk labeled llllll. (This is for the user in case he decides to change FAP)

2. Linkedit and catalog FAP, and execute it on a test program.

3. Execute cataloged FAP on Fortran IV test case.

III. General operating instructions.

A. Once the program has been cataloged the following JCL is used:

```
//  NAME        JOB     , NAME, MSGLEVEL = 1
//  JOBLIB      DD      DSNAME = FORTAP, DISP = OLD
//  STEP        EXEC    PGM = FAP
//  GO. OUTTXT  DD      DSNAME = && TEXT, UNIT = 2311,    X
//                      SPACE =( TRK, (1, 1)), DISP =(, DELETE)
//  GO. OUTPUT  DD      SYSOUT = A
//  GO. INPUT   DD      *
```

Source deck to be analyzed.

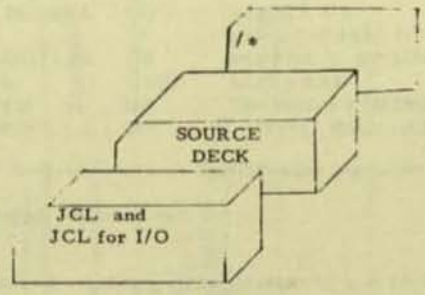
```
/*
```


The names used mean:

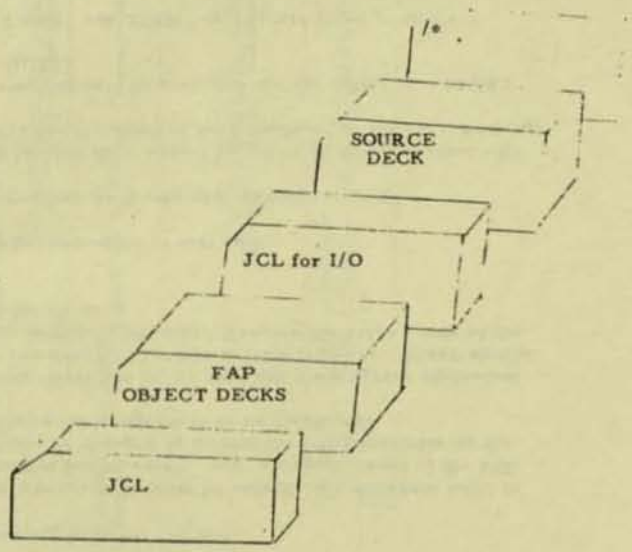
FORTAP	JOBLIB NAME for the analysis program
FAP	program name
&& TEXT	name of auxiliary output
OUTTXT	intermediate output set
OUTPUT	print data set from FAP
INPUT	input data set

IV System Configuration

A. System 360
B. Minimum memory size 128K (with overlaying, can be used with 64K)
C. Operating system
D. Card reader
E. Card punch
F. Printer
G. Standard carriage control tape



Use of Catalogued FORTRAN Analysis Program



Use of Uncatalogued FORTRAN Analysis Program

TAPE KEY

The tape supplied with this documentation consists of two files where the records in each file have a block size of 1600 and a logical record length of 80; in other words, the card images are blocked 20 to a record.

The tape may be punched by using the following Job Control Deck:

```
//PUNCH     JOB
//          EXEC    PGM = IEBPTPCH
//SYSPRINT  DD      SYSOUT = A
//SYSUT1    DD      UNIT = 2400, LABEL = (, NL),                X
//          DD      VOLUME = SER = INPUT, DCB =                 X
                    (RECFM = FB, LRECL = 80,                    X
                    BLKSIZE = 1600)
//SYSUT2    DD      UNIT = 2540
//SYSIN     DD      *
          PUNCH
/*
```

File 1	41 Blocked Records	No. of Cards
I.	JCL for purging FORTAP from disk labeled llllll.	7
II. A.	JCL for cataloging analyzer FORTAP (FAP) on disk labeled llllll.	6
B.	Object decks (27) comprising analyzer.	573
C.	JCL for defining I/O of analyzer and test deck	36
III. A.	JCL for executing cataloged analyzer	7
B.	Test program deck	180

T/M

File 2	406 Blocked Records	No. of Cards
I.	Source decks for analyzer	8118

T/M
T/M

File 1 is the only one which need be punched. It contains three jobs, for scratching, cataloging, and testing the analyzer.

File 2 contains the source decks for the analyzer including JCL so that the source can be assembled immediately after punching.

Description of Routine

On the following pages is a description of each one of the routines in the Fortran Analysis Program. The relationship of the routines and description of tables are described for each routine. Flowcharts following the description of the routines summarize the basic logic of these routines. For a more detailed description, the user is referred to the source programs included on the magnetic tape associated with this documentation.

ROUTINE PASS 1

1. FUNCTION:

PASS 1 is the control routine of the FORTRAN analysis program. It initiates and coordinates the action of all the other routines.

2. ENTRANCE:

As the control routine, PASS 1 is entered from the operating system.

3. OPERATION:

PASS 1 processes all source code for a complete FORTRAN program and sets up the intermediate text for PASS 2. It scans the FORTRAN source code for labels and keywords, removes blanks and translates all characters into EBCDIC representation.

Each FORTRAN statement is given a sequence number except JCL and comment cards. In order to process all labels, PASS 1 communicates with the label dictionary. PASS 1 also sets up status information in the intermediate text. It does this by communicating with Keytab.

All input and output is handled with external routines. The READ routine handles the FORTRAN source code. The PRINT routine prints out the FORTRAN source along with the sequence number and the OUTTXT routine writes out the intermediate text on a temporary data set.

4. EXITS:

When an end card is reached control is passed to PASS 2.

When end of file is reached control is returned to the SUPERVISOR routine.

5. SUBROUTINES CALLED:

1.	LABDIC -	To process labels
2.	KEYTAB -	To process symbols
3.	READ -	To read in source code
4.	PRINT -	To print out source code
5.	OUTTXT -	To write out intermediate text
6.	PASS 2 -	To process the intermediate text.

6. LIMITATIONS:

PASS 1 can scan only basic FORTRAN source code.

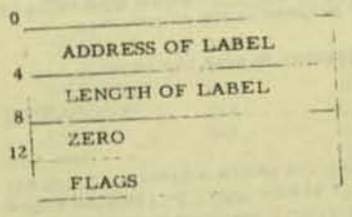
7. PARAMETERS:

To PASS 1 - PASS 1 receives no parameters
From PASS 1 - PASS 1 sends parameters to all subroutines except PASS 2.

(See Figure 2)

FIGURE 2

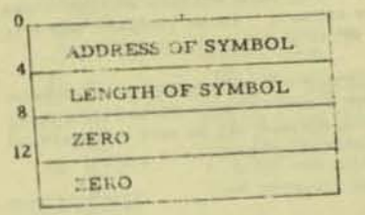
To LABDIC:



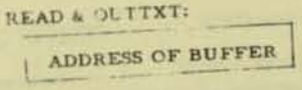
FLAGS

Bit 25 - DO LABEL
Bit 26 - IF, GO TO LABEL

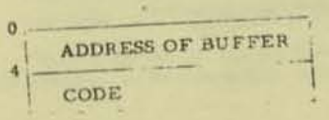
To KEYTAB:



To READ & OUTTXT:



To PRINT:



CODE

4 - NEW PAGE
0 - NEW LINE

SUBROUTINE PASS 2

1. FUNCTION:

PASS 2 is the major scan routine of the program. It scans and analyzes the intermediate text set up by PASS 1.

2. ENTRANCE:

PASS 2 is entered when PASS 1 has finished setting up the intermediate text for one complete FORTRAN program.

3. OPERATION:

PASS 2 processes each intermediate text record. It scans each record for symbols and labels and sends the appropriate information to the necessary table building routines. For symbols, and entry is built in the symbol table and the variables used in a block table(VARUSE). For block definition labels and branch points an entry is made in the block definition table(BDT). PASS 2 also makes entries into the time accumulation tables(TIMTAB and TIME).

4. EXITS:

When the intermediate text is completely processed PASS 2 exits to the end of file supervisor(EOF).

5. SUBROUTINES CALLED:

1.	INTXT	-	To read in intermediate text
2.	LABDIC	-	To process labels and symbols
3.	TIME	-	To accumulate time for block of code
4.	TIMTAB	-	To accumulate total time for each block of code
5.	OPTAB	-	To recognize legal and illegal special characters
6.	BDFUNC	-	To recognize legal FORTRAN built in functions
7.	VARUSE	-	To build up variables used in block table
8.	BDT	-	To build up block definition table
9.	EOF	-	To control printing of all tables
10.	LOGIC	-	To determine logical IF

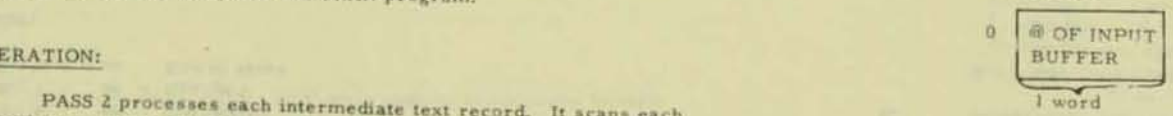
6. LIMITATIONS:

PASS 2 can be used only in conjunction with the intermediate text set up by PASS 1.

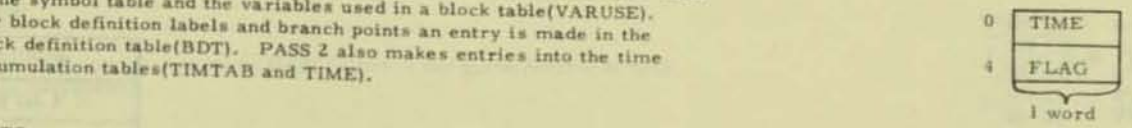
7. PARAMETERS:

To PASS 2 - None
From PASS 2 -

To INTXT:

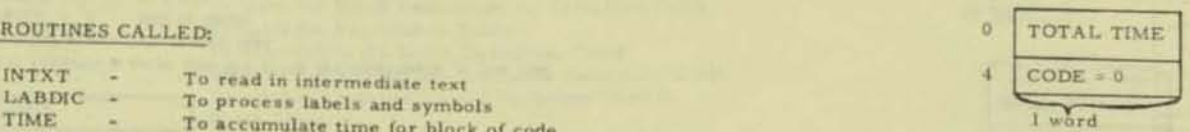


To TIME:

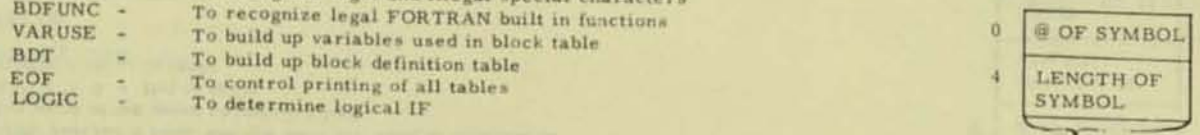


1 Bit Flag - on - Send total time for block to PASS 2 in 1st word of the given parameter list.

To TIMTAB:



To BDFUNC:



BDFUNC returns a code and the time for built in function in the 2nd word of the given parameter list.

CODE = 0 - Not in table
CODE = 4 - In table

To OPTAB:

0	@ OF CHARACTER
4	ZERO

OPTAB returns a code and the time for operation in the 2nd word of the given parameter list.

CODE = 0 - Not in table
CODE = 4 - In table

To VARUSE:

0	CODE = 0
4	@ OF I. D.
8	FLAGS

FLAGS:

Bit 5 - on - End of program
Bit 6 - on - Unchanged
Bit 7 - on - Changed
Bit 8 - on - End of block

To BDT:

0	FLAGS
4	SEQ. NUMBER

FLAGS:

Bit 29 - on - Fall through sequence number
Bit 30 - on - End of program sequence number
Bit 31 - on - Branch point sequence number
Bit 32 - on - End of block sequence number if all
if all are off - Start block sequence number

To LABDIC:

@	@ OF SYMBOL
	LENGTH OF SYMBOL
	ZERO
	FLAGS
	@ OF RETURN

1 word

FLAGS:

Bit On	Meaning
Bit 23	Information requested on ID
Bit 24	On-symbol, Off-Label
Bit 25	Do statement
Bit 26	If, go to statement
Bit 27	Array
Bit 28	Self defining function
Bit 29	Information requested
Bit 30	Should have been defined
Bit 31	Not found

8. Criteria for block definition:

1. Any statement that transfers control or branches to another statement is considered the end of a block.

2. Any statement with a label and receives control from a branch is considered to be the beginning of a block.

SUBROUTINE EOF

1. FUNCTION:

The function of this routine is to coordinate all activity when the program being analyzed has been read in and scanned.

2. ENTRANCE:

The entry point is EOF. Control will be given to this point by PASS 2 when the end of file condition occurs.

3. OPERATION:

Upon receiving control, EOF gives control sequentially to each subroutine which has to build, print, or reinitialize a table.

4. EXITS:

EOF returns control to PASS 2 after it has given control to each of its subroutines.

5. SUBROUTINES CALLED:

BDTBUF	-	To print the Block Definition Table.
CROSS	-	To build the Cross Reference Table.
BRVBLD	-	To build the Block Referenced by Variables Table.
PREC	-	To build the Precedence Table.
BDT	-	To reinitialize the Block Definition Table
XWRITE	-	To print and reinitialize the Cross Reference Table.
VARTAB	-	To print and reinitialize the Variables Used in Block Table.
BRVBUF	-	To print the Blocks Referenced by Variables Table.
PBUF	-	To print and reinitialize the Precedence Table.

6. PARAMETERS:

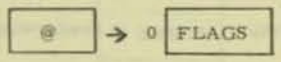
To CROSS

Register 1

Bit 31 is turned on to indicate that the table should be built.

To BDT:

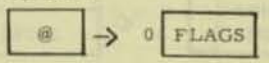
Register 1



FLAGS: Bits 28 and 31 are turned on to indicate the the table is to be reinitialized.

To BRVBLD:

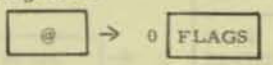
Register 1



FLAGS: Bits 27, 28, and 29 are turned on to indicate that the table is to be built.

To PREC:

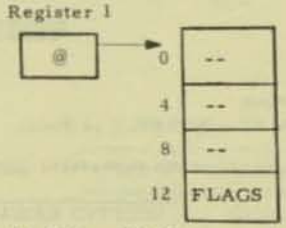
Register 1



FLAGS: No bits are turned on to indiacate that the table is to be built.

To LABDIC:

Register 1



FLAGS: Bit 22 is turned on to indicate that the table should be initialized.

SUBROUTINE BDT

1. FUNCTION:

BDT builds and references a block definition table for the particular routine being analyzed by the scan routines.

2. ENTRANCE:

BDT is either entered from the PASS 2 routine with information to build the table, by a routine that wants to reference the table, or by the EOF routine to set a switch so the tables will be reinitiated when the next job is to be executed.

3. OPERATION:

On the initial call from the PASS 2 routine BDT initializes its thumb-index table and its block definition table, the address of the first block of the table is put in the thumb-index table and the start point of the first block is put in the block definition table. PASS 2 then passes one piece of information at a time with a code that characterizes a start, last, destination, or branch point of a block. The BDT routine, by use of a decision table, puts the information in its appropriate position in the table.

When PASS 2 is finished the block definition table is completely built and can be printed out by BDTBUF referencing it. This is done by passing the block number and a reference code to BDT until a zero is returned.

After the table has been printed out BDT gets control to resolve the sequence number destination and branch points to the block numbers they represent.

After the table has been resolved it can be referenced by both CROSS and PREC. These two routines can find out to which blocks a particular block branchs by continuously passing the block number to BDT, along with the reference code, until a zero is returned.

At the close of the entire program analysis job the EOF routine gives BDT control to enable its table initializing routine for the start of the next job.

4. EXIT:

BDT returns to its caller except when an error occurs and it then exits to the ERROR routine.

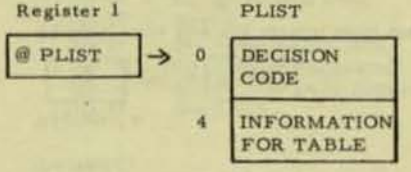
5. SUBROUTINES CALLED:

BDT references the following routine:

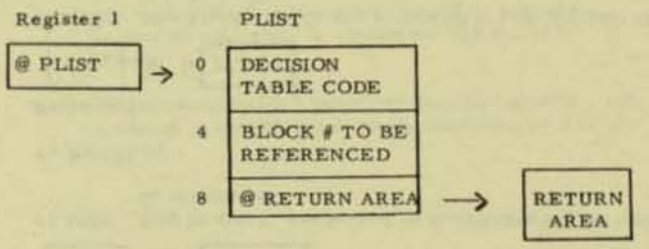
a.) ERROR - To put appropriate error message in buffer.

6. PARAMETERS:

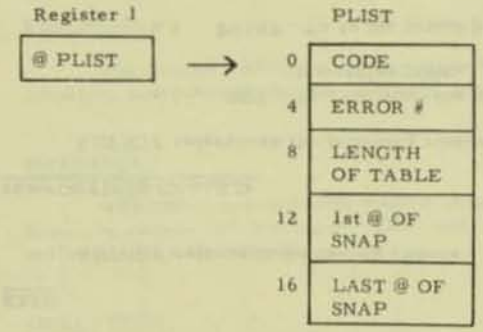
a.) Parameters received from the PASS 2 routine.



b.) Parameters received from referencing routine.



c.) Parameters to ERROR routine.



CODE: 0 → Return
4 → Abort

LAST @ OF SNAP: If 0 then no snap taken.

7. LIMITATIONS:

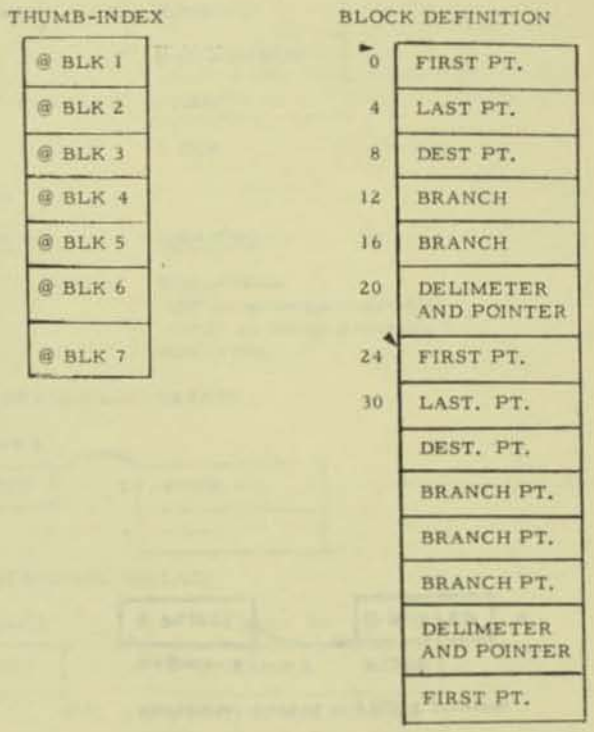
The defined size of the thumb-index table and block definition table is a limitation but can be increased by just changing the appropriate table defining statement.

Error messages are printed out for the following conditions:

a.) Thumb-index table overflow (Error 5)
b.) Block definition table overflow (Error 6)
c.) Invalid decision table code received from caller (Errors 14 and 39)
d.) Referencing an invalid block number (Error 24)

8. Tables:

Block definition and thumb-index tables



Decision Table Codes

CODE	ACTION BY BDT
0	START OF BLOCK
1	END OF BLOCK
2	RESOLVE TABLE
3	BRANCH POINT
4	REFERENCE RESOLVED TABLE
5	REFERENCE UNRESOLVED TABLE
6	END OF SCAN (EOF)
7	DESTINATION POINT
8	REINITIALIZE

SUBROUTINE BDTBUF

1. FUNCTION:

BDTBUF loads a buffer area with information on a block obtained from the BDT routine.

2. ENTRANCE:

BDTBUF is called by the EOF routine when the block definition table is to be printed out.

3. OPERATION:

On initial control of the BDTBUF routine the title and the headings to the block definition table are printed.

BDTBUF then requests information on the blocks from the BDT routine by passing the address of a parameter list, that contains a decision table code and the particular block to be referenced, to the BDT routine. The blocks are processed sequentially from block 1 and when BDTBUF receives all information on the particular block the address of the buffer is passed to the print routine in a parameter list.

When the entire table has been processed, control is passed to the resolve aspect of the BDT routine to ready the table for further reference. Control is then returned to the EOF routine.

4. EXIT:

BDTBUF returns to its calling routine.

5. SUBROUTINES CALLED:

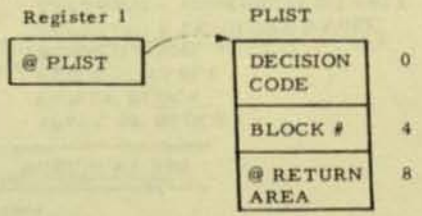
BDTBUF references the following routines:

a.) BDT - To reference the block defintion table and to resolve table.

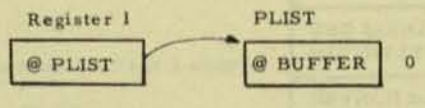
b.) PRINT - To do the actual printing of the buffer.

6. PARAMETERS:

a.) Parameters passed to BDT routine:



b.) Parameter passed to PRINT routine:



SUBROUTINE VARUSE

1. FUNCTION:

The routine VARUSE performs three functions building, referencing, and reinitializing the variables used by block table.

2. ENTRANCE:

VARUSE is entered from PASS 2 when the table is to be built, from any routine referencing the table, and from VARTAB when the table is to be reinitialized.

3. OPERATION:

VARUSE is called by PASS 2 when a variable is encountered in the input stream, and called by BRVBLD and VARTAB when requesting information from the table. Also, VARUSE is called by VARTAB when end of file occurs to reinitialize its table and work areas.

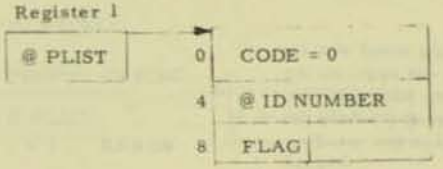
When control is passed to VARUSE, one of the following codes is received:

1. Code 0 - When initially called by PASS 2, VARUSE initializes its table and thumb index to 0. At all other times, VARUSE makes an entry in the Variable Used by Block Table according to the information passed from PASS 2, and the address of each block in the table is entered in the thumb index.
2. Code 4 - VARUSE withdraws one entry, consisting of the ID number, its status, and block number, from the table and places it in a parameter list (see figure 3).
3. Code 8 - VARUSE reinitializes a one-time switch causing the table and thumb index to be reinitialized to 0.

4. PARAMETERS:

From PASS 2:

Register 1

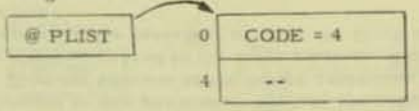


FLAGS:

BIT ON	MEANING
0	End of block
1	"IN" or unchanged variable
2	"OUT" or changed variable
3	End of file

From BRVBLD and VARTAB:

Register 1



To BRVBLD and VARTAB:

Register 1

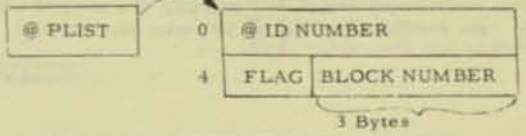


Figure 3

FLAG BYTE = 0: Variable unchanged
= 1: Variable changed
= 2: Variable unchanged and changed in same block

5. EXIT:

VARUSE exits to the calling routine.

6. SUBROUTINES CALLED:

ERROR - to print out an error message and supply an abend dump in case abnormal conditions arise.

7. LIMITATIONS:

The sizes of the table and thumb-index are limitations, but can be increased by changing the DS statements. In case one of these conditions arise, error messages #3 and #4 will be printed out, respectively, and an abend dump will be taken.

8. TABLES:

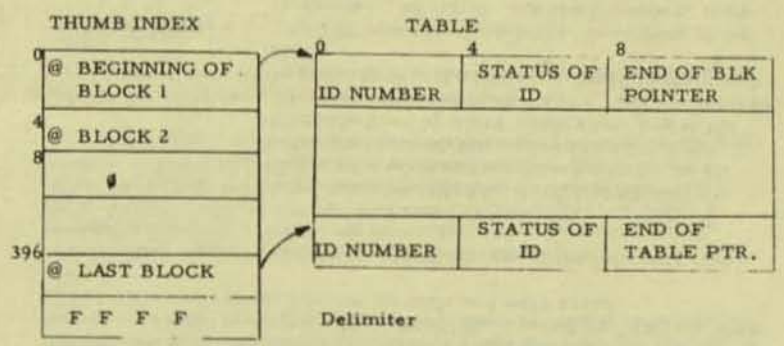


FIGURE 4

NOTE:

An ID number only appears once in a particular block and there may be more than one variable and its status in a block, but only one end of block pointer.

SUBROUTINE VARTAB

1. FUNCTION:

This routine fills a 120 byte buffer area with information from the variables used by block table to be printed out.

2. ENTRANCE:

VARTAB is called by the EOF routine when the variables used by block table is to be printed out and reinitialized.

3. OPERATION:

Initially, the title and headings are printed out. VARTAB then passes a code of 4 to VARUSE requesting information from VARUSE's table. VARTAB repeatedly calls VARUSE for information concerning a block until end of block occurs. In this case, VARTAB passes the address of a buffer containing the block number and all the variables not changed in this particular block to the print routine to be printed out. Then the address of the buffer containing all variables changed in the block is passed to the print routine. This entire process is repeated until the variables used by the block table has been completely printed out.

4. EXIT:

VARTAB returns to the calling routine.

5. SUBROUTINES CALLED:

During execution, VARTAB references the following subroutines.

a.) VARUSE - To reference its table and pass information to its caller.

b.) PRINT - To print out the contents of the buffer area.

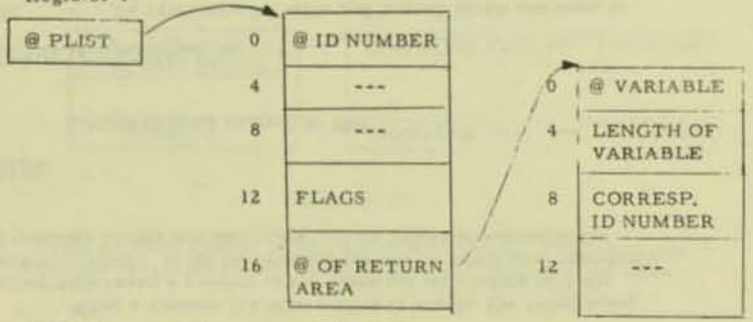
c.) ERROR - To print out any error message to correspond to the type of error and call an abend dump.

d.) LABDIC - To retrieve the corresponding variable for the given ID number.

6. PARAMETERS:

To LABDIC:

Register 1



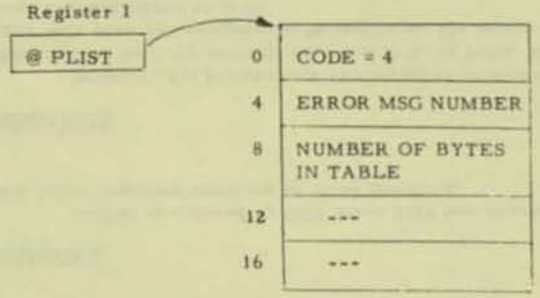
From LABDIC:

FLAGS:

Bits on	Meaning:
23 and 24	Referencing symbol table

To ERROR:

Register 1



CODE = 4 for an abend dump

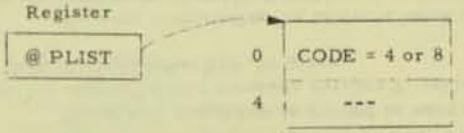
To PRINT:

Register 1

1 word

To VARUSE:

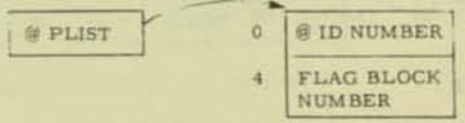
Register



CODE 4 = Reference table CODE 8 = To reinitialize table

From VARUSE:

Register 1



FLAGS: Bit = 0: VARIABLE unchanged
= 1: VARIABLE changed
= 2: VARIABLE used and changed in same block

SUBROUTINE CROSS

1. FUNCTION:

CROSS builds and references a Branch Cross-Reference Table consisting of threads of functional block numbers.

2. ENTRANCE:

CROSS is entered by EOF after BDT has resolved its table from sequence numbers to block numbers.

3. OPERATION:

Initially, the parameters are tested to determine if the table is to be built or referenced. If it is to be built, which can occur only once per program to be analyzed, the tables and work areas are initialized to zero.

Information for building is obtained from BDT. Control must be passed and returned for each transfer block in the program being analyzed. CROSS passes block numbers, in ascending order from one, to BDT, which returns a transfer block number. When the build phase is complete, control is returned to EOF.

When a request for information is made, the referencing section addresses a pointer in the index corresponding with the block requested. If the same block number is requested repeatedly, all transfer blocks for that block will be returned sequentially.

4. EXITS:

CROSS returns control to EOF.

5. SUBROUTINES CALLED:

BDT - To reference its table and branch block numbers to build the Cross-Reference Table.
ERROR - To print an error message and take an ABEND dump.

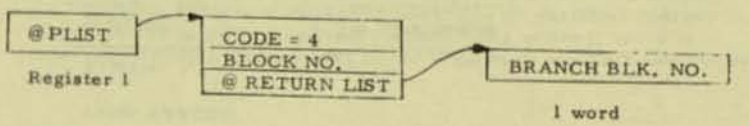
6. PARAMETERS:

To CROSS:

FLAGS:

0 - REFERENCE TABLE
1 - BUILD TABLE

To BDT:



7. LIMITATIONS:

The major limitation in CROSS is table size. If table overflow occurs, error message number 9, table overflow, or number 10, index overflow, is given.

If, when referencing the table, a block number larger than the largest existing block number is requested, error message number 21 is given.

8. TABLES:

INDEX

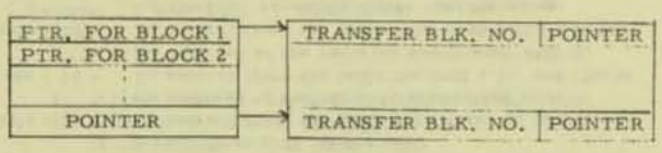


FIGURE 5

Table pointers connect blocks which pass control to the block whose number corresponds with the position of the pointer in the index.

SUBROUTINE XWRITE

1. FUNCTION:

XWRITE is designed to prepare the Branch Cross-Reference Table for printing.

2. ENTRANCE:

XWRITE is entered from EOF after CROSS has completed the building of the table.

3. OPERATIONS:

XWRITE references CROSS by passing a block number repeatedly until all transfer blocks have been returned. Each transfer block number is placed consecutively in a blanked out buffer area. PRINT is then called and the area is printed. This is done until the table is completely printed.

4. EXITS:

Control is returned to EOF.

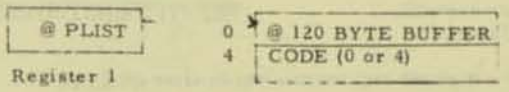
5. SUBROUTINES CALLED:

PRINT: To print out buffer area.
CROSS: To reference the Branch Cross-Reference Table.

6. PARAMETERS:

To XWRITE: None

To PRINT:



SUBROUTINE BRVBLD

1. FUNCTION:

This routine enters or extracts information from the Block referenced by variables table which contains I.D.'s and Blocks which contain the variable associated with I.D.

2. ENTRANCE:

This routine is given control at BRVBLD by three routines:

a) EOF Supervisor - gives control with code of 28 in order that the table is built.

b) BRVBUF - gives control with code of 0, 4, 8, 12, 16, or 20 in order to get information from table.

c) MPRC - gives control with code of 0, 4, or 8 in order to get information from table.

3. OPERATION:

The operation performed by this routine depends on the code passed to it.

a) Code = 0 - To reference the table and extract the first Block number for the next I.D. in the table.
b) Code = 4 - To extract from the table the next Block number for the I.D. presently under consideration.
c) Code = 8 - To extract from the table the first Block number for the first I. D. in the table.
d) Code = 12 - To extract from the table the first I.D. and obtain the address of the variable associated with it.
e) Code = 16 - To extract from the table the next I.D. and obtain the address of the variable associated with it.
f) Code = 20 - To extract from the table, the first Block number for the I.D. presently under consideration.
g) Code = 28 - To request information from VARTAB to build the table.

BRVBLD will put the information it extracts into a parameter list and return control to the routine that called it.

4. EXITS:

BRVBLD exits to the routine that called it.

5. SUBROUTINES CALLED:

VARUSE - To give information necessary to build the Block Referenced by Variables Table.
ERROR - To print out error message and will request either an abend or a snap dump.
LABDIC - To give the address of a variable name and its length. The user must supply this routine with the address of the I.D. associated with the variable.

6. PARAMETERS:

From EOF:

Register 1 1 word

FLAGS:

If bits 27, 28, and 29 are on, BRVBLD builds the table.

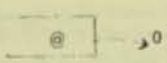
To VARUSE:

Register 1

FLAGS:

Bit 29 - on- to request information for building the table.

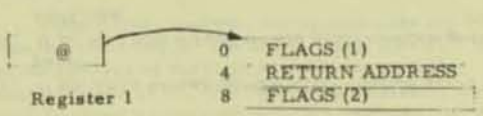
From VARUSE:



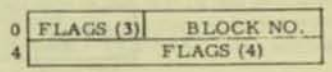
FLAGS:

1. If bit 7 is on, the variable is changed in that block.
2. If no bits in the first byte are on then the variable is not changed in that block.

From PREC:



To PREC:

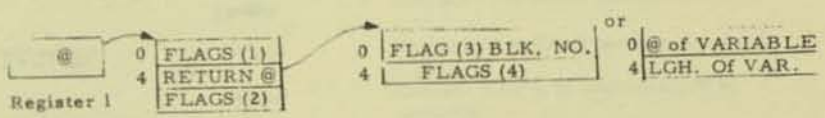


FLAGS	BITS TURNED ON	MEANING
(1)	29	Extract next block number for the same I. D.
	28	Extract first block number for the first I. D.
	none	Extract first block number for next I. D.
(2)	none	Ignore the flag field of the Block numbers in the table.

FLAGS	BITS TURNED ON	MEANING
(3)	none	This block does not change the value of the variable.
	7	This block does change the value of the variable.
(4)	none	There is no more information in the table.
	29	There is still information left in the table.

From BRVBUF:

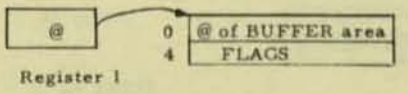
To BRVBUF:



FLAGS	BITS TURNED ON	MEANING
(1)	29	Extract next Block number for same I. D.
	28	Extract first block number for first I. D.
	27, 29	Extract first block number for same I. D.
	28, 29	Extract first I. D. and obtain the address of associated variable.
	27	Extract next I. D. and obtain the address of associated variable.
(2)	0, 31	Extract a Block number only if it changes the value of the variable.

FLAGS	BITS TURNED ON	MEANING
(2)	0	Extract a block number only if it does not change the value of the variable.
(3)	7	Block passed, changes the value of the variable.
	none	Block passed, does not change the value of the variable.
(4)	none	No more information in the table.
	29	More information in the table.

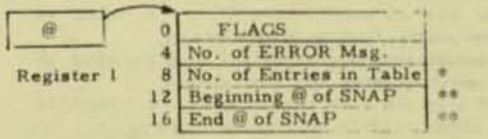
To PRINT:



FLAGS:

1. If bit 29 is turned on the print routine skips to a new page.
2. If no bits are turned on the print routine goes to the next line.

To ERROR:

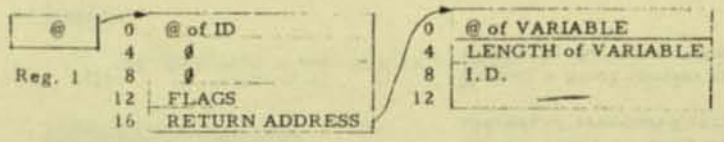


* For table overflow messages only.
** This word is ∅ if no snap is desired.

FLAGS:

1. If no bits are on, no dump will be taken.
2. If bit 29 is on, ERROR will request an abend dump.

To LABDIC: From LABDIC:



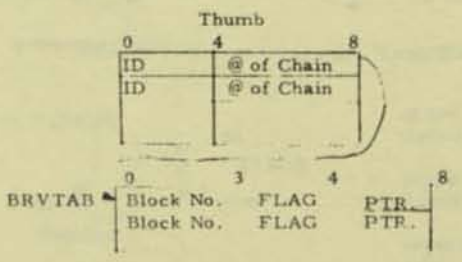
FLAGS:

Bits 23 and 24 are turned on to indicate that the @ of a variable is requested.

7. LIMITATIONS:

This routine can handle only Block numbers less than 9,999. If this occurs, error message 35 will be printed out. Initially, the thumb index is allocated 4000 bytes and the table is allocated 10,000 bytes. If either overflow, an error message numbers 2 and 1, respectively, are printed out. If more storage is needed the only change that has to be made is the DS card for the particular table.

8. TABLES:



FLAG: If no bits are turned on the block does not change the value of the variable. If the low order bit is on the block does change the value of the variable.

SUBROUTINE BRVBUF

1. FUNCTION:

The function of this routine is to request the information in the Block Referenced by Variables table and put it into printable form in a 120-byte buffer area so that it can be printed out.

2. ENTRANCE:

This routine is entered at BRVBUF from the EOF supervisor routine. It receives control at this point only once and this occurs when the table is to be printed out.

3. OPERATION:

Upon getting control, the routine uses the PRINT routine to print out a heading for the table. Next BRVBUF uses BRVBLD to get the address of the first variable in the table. This is done by passing a code of 12. BRVBUF then requests, one at a time, all the block numbers associated with this variable that do not change its value. Next it requests the block numbers that do change the value after each group of block numbers has been passed a code of 0 is returned to signify that this is the last of the particular block numbers. BRVBUF then calls PRINT to print out the buffer which contains the variable and block numbers. This takes place until all the information in the table has been printed out.

4. SUBROUTINES CALLED:

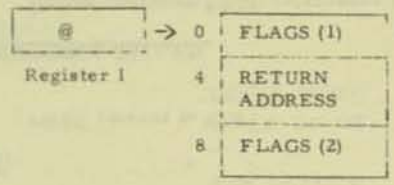
PRINT	-	Prints out the 120-byte buffer area, the address of which is passed from the calling routine.
ERROR	-	Prints out an error message, the number of which is passed from the calling routine.
BRVBLD	-	Extracts information from its table and passes it to the calling routine.

5. EXIT:

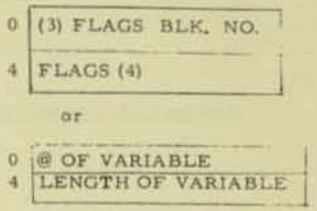
BRVBUF exits to EOF after the table has been completely printed.

6. PARAMETERS:

To BRVBLD:

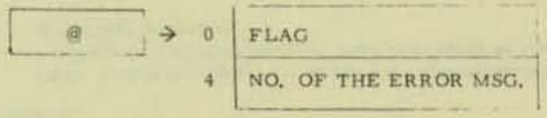


From BRVBLD:



FLAGS:	Bits turned on	Meaning
(1)	29	Extract next Block number for same I. D.
	28	Extract first Block number for first I. D.
	27, 29	Extract first Block number for same I. D.
	28, 29	Extract first I. D. and obtain @ of variable.
	27	Extract next I. D. and obtain @ of variable.
(2)	0, 31	Extract a block number only if it changes the value of the variable.
	0	Extract a block number only if it does not change the value of the variable
(3)	None	Block number does not change the value of the variable.
	7	Block number does change the value of the variable.
(4)	None	No more information in table.
	29	More information in table.

To ERROR:



8	NO. OF ENTRIES IN TABLE	*
12	BEGINNING ADDRESS OF SNAP	**
16	END ADDRESS OF SNAP	**

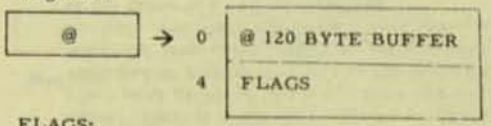
* Used only if error is for table overflow.
** If no snap is desired this word is zero.

FLAGS:

1. If no bits are turned on control is returned to the calling routine after message is printed.
2. If bit 29 is turned on an abend dump is taken.

To Print:

Regiser 1



FLAGS:

If no bits are turned on; Print Buffer on next line.
If bit 29 is turned on: Skip to top of newpage.

7. LIMITATIONS:

This routine can handle variable names of up to 8 characters in length. The block numbers cannot be greater than 9,999. If it occurs this condtion will be discovered in the build routine. The only error message for this routine is the one for an empty table. If this occurs the error probably will be found in the passing of parameter lists.

SUBROUTINE PREC

1. FUNCTION:

PREC builds the first part of the precedence table by moving into the table the number of each block of code according to the "branching order" of the test program.

2. ENTRANCE:

PREC is entered first by a call from EOF (end-of-file supervisor) and second by a call from PCRF (precedence table referencing routine).

3. OPERATION:

PREC initializes the thumb-index and table to zero.

Routine PREC is executed only once for each precedence diagram. PREC builds the first part of the precedence table by inserting a block into the table and then inserting the branch points of that block into the table on a "lower level." This process continues with PREC calling BDT (block-definition-table) until every branch point has been entered into the table (named PRTAB). Duplicate branch points are not entered into the table. Hence, when part 1 is completed, PRTAB contains the number of every block of code in the test program in an order based on the priority of their branching execution. A delimiter is moved into the table and the thumb-index following the last element in both.

The thumb-index PRTNDX is built with each element pointing to the beginning of a chained "level" in the table.

When PREC is finished building the table and thumb-index, it passes control to MPRC to build the final form of the precedence table. When MPRC has finished processing it passes control back to PREC, which then unconditionally branches to EOF.

4. EXIT:

PREC returns to EOF.

5. SUBROUTINES CALLED:

During execution PREC references subroutines:

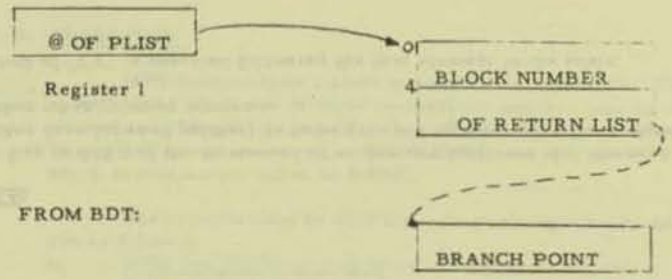
A.) BDT - to receive block numbers and their branch points.
B.) MPRC - to modify first stage of precedence table.

6. LIMITATIONS:

At present, the thumb-index PRTNDX has storage for 150 elements and the table PRTAB has storage for 250 elements. Hence, there cannot be more than 250 blocks of code in a test program. If there are, an abend-dump will be given with a "table overflow" error message 11 printed out. If there are more than 150 levels required by the precedence diagram, a "thumb-index overflow" error message 12 will be given with an abend dump.

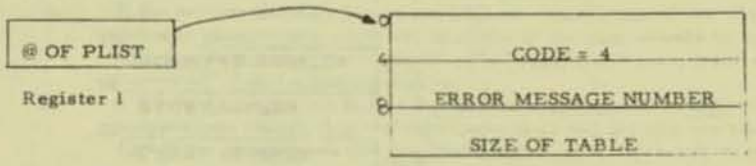
7. PARAMETERS:

TO BDT:

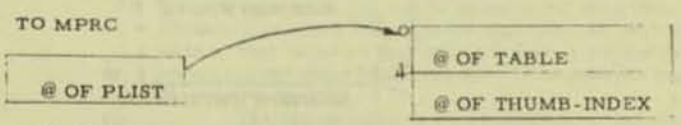


FROM BDT:

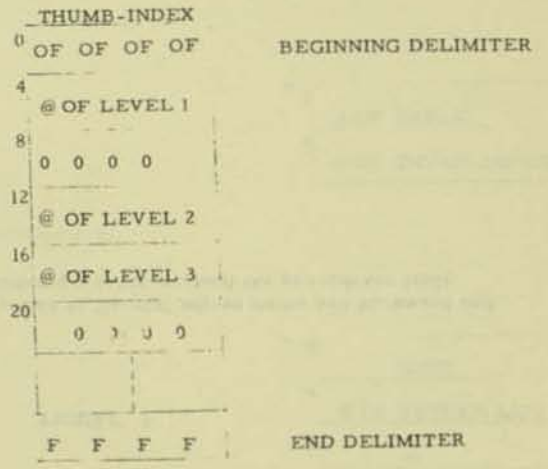
TO ERROR:



TO MPRC

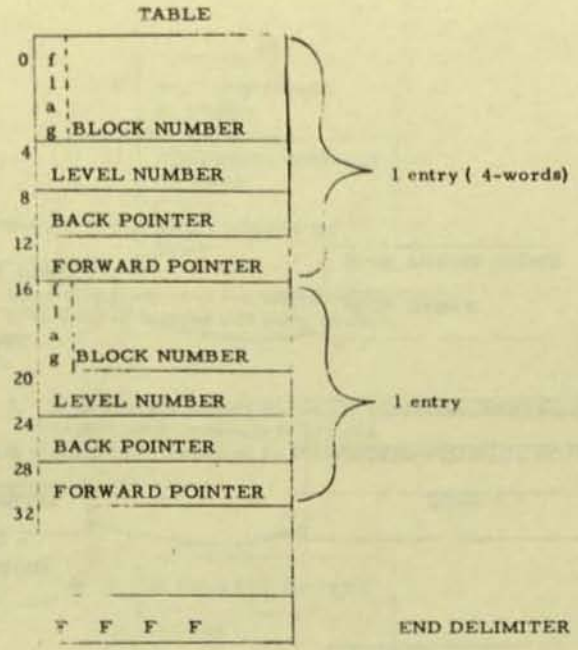


FROM EOF:



NOTE:

There may be an element of zeroes between 2 elements containing addresses. Each element is one full-word. A word of "1's" is inserted following the last element containing an address.



NOTE:

The flag in BIT 1 of the first word of each entry indicates that the next block number (see forward pointer) is to be grouped together with the pointing block number when printing occurs.

A word of "1's" is inserted following the last element in the table.

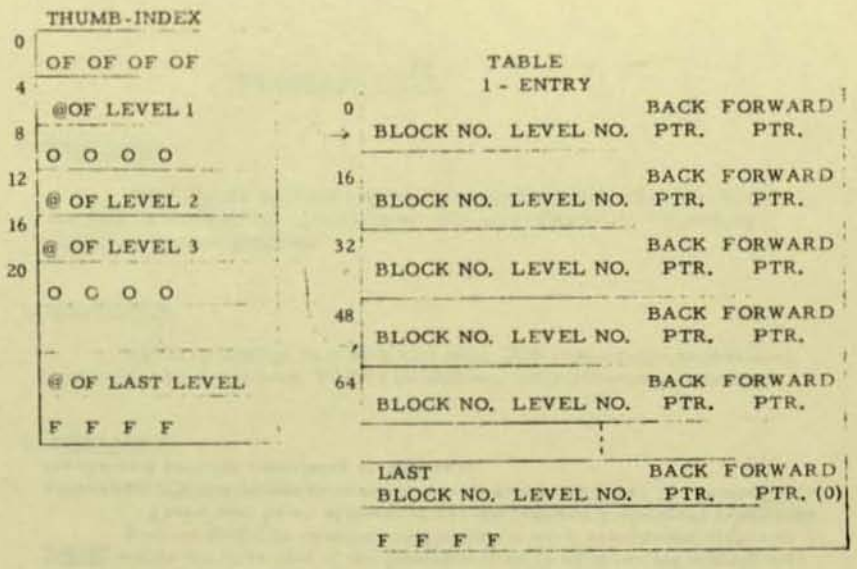


FIGURE 6

NOTE:

Thumb-index points to "levels" whose items are backward and forward chained and areinterspersedthroughout the precedence table.

SUBROUTINE MPRC

1. FUNCTION:

MPRC is a routine which modifies the precedence table begun in PREC.

2. ENTRANCE:

MPRC is entered by a call from PREC.

3. OPERATION:

MPRC establishes 2 block numbers as comparands. The "level" in PRTAB on which one of these comparands appears may or may not be altered according to the use of the variables in that block of code (comparand). The variables are examined by BRVBLD (block-referenced-by-variables), which is continually called by MPRC.

The criteria used by MPRC for determining any changes in PRTAB are as follows:

a. If the two blocks of code being compared do not change a variable, or, if the two comparands have no variables in common, then the two comparands are independent of each other and may be executed simultaneously. This is indicated by placing the block numbers on the same "level" in PRTAB if they are not already in that position.

b. If the two comparands have a variable in common and if one of the comparands changes that variable, then one of the comparands is moved (if necessary) to a "lower level." The "rightmost" comparand shall be "lowered" if this situation occurs.

c. If the two comparands have a variable in common and if both of the comparands change that variable then one of two actions are taken:

1.) If the comparands are not on the same level, then they remain in their position in PRTAB.

2.) If the comparands are on the same level, then their forward and backward pointers are adjusted such that the two comparands are next to each other on that level. A flag is set for the print routine which causes these block numbers to be grouped together in the final printed output.

The thumb index is also modified, if necessary, during MPRC. When all the block numbers of all "levels" in PRTAB have been examined for variables, and all the resulting modifications in PRTAB have been made, then PRTAB is ready for printing.

4. EXIT:

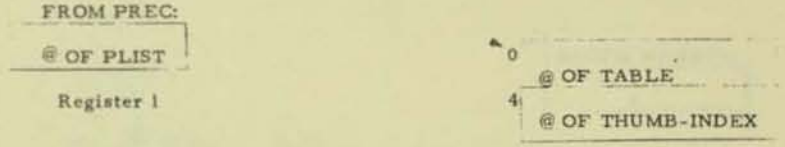
MPRC unconditionally branches back to PREC

5. SUBROUTINES CALLED:

A.) BRVBLD - to determine whether or not a variable has been changed in a block of code and to modify PRTAB accordingly.

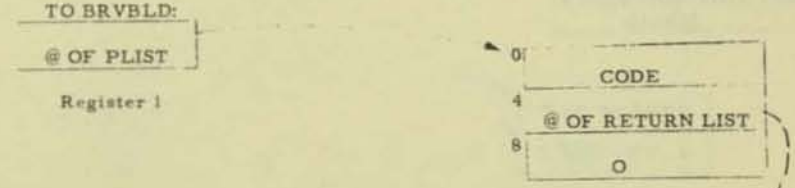
6. PARAMETERS:

FROM PREC:



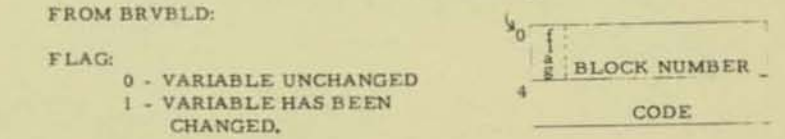
TO PREC: No parameters.

TO BRVBLD:



CODES: 0 - Pass back first block number of next row.
4 - Pass back next block number of current row.
8 - Pass back first block number of first row.

FROM BRVBLD:



FLAG:
0 - VARIABLE UNCHANGED
1 - VARIABLE HAS BEEN CHANGED.

CODE: 0 - More information exists in table.
4 - Block-referenced-by-variables table has been completely searched.

SUBROUTINE PCRF

1. FUNCTION:

PCRF references the precedence table for the buffer routine PBUF.

2. ENTRANCE:

PCRF is entered by a call from PBUF.

3. OPERATION:

When PBUF gets control, it calls the precedence table PREC for the address of the thumb-index. It then traces each level of the table by using the forward pointers to get to each separate block number in the level. It then returns to PBUF with the block number and certain flags.

When one level is printed, PCRF uses the thumb-index to print out each succeeding level in PRTAB.

4. EXIT:

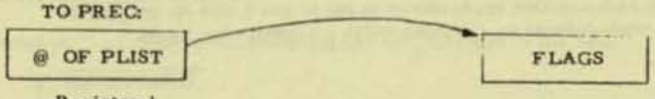
PCRF unconditionally branches back to PBUF.

5. SUBROUTINES CALLED:

PREC: To receive address of the thumb-index.

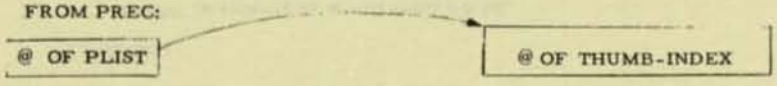
6. PARAMETERS:

TO PREC:



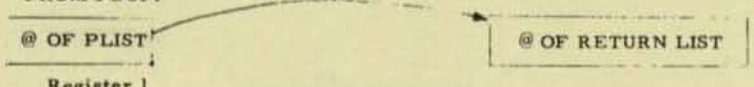
Register 1

FROM PREC:



Register 1

FROM PBUF:



Register 1

RETURN LIST

TO PBUF:

CODES:

0 - This is last block of level.
4 - There are no more levels to be printed.
8 - There are still more block numbers on this level.

SUBROUTINE PBUF

1. FUNCTION:

PBUF sets up a 120-byte buffer area with information from the precedence table and calls PRINT to print out the buffer.

2. ENTRANCE:

PBUF is entered by a call from EOF when all other tables have been printed.

3. OPERATION:

When PBUF is called by EOF, it calls PRINT to print out a title followed by a heading and explanatory notes. Then it calls the precedence table referencing routine PCRF to receive the first block number to be placed in the buffer.

Then it passes the block number to TIMTAB to receive the accumulated execution time for that block. PBUF then moves them into the buffer. If the block number is the last block number for that level in the precedence table, then PBUF calls PRINT to print the buffer; otherwise, it returns to PCRF for the next block number of that level. If the buffer is filled before the end of a table level is reached, then the next printed line will be single spaced, signifying it as a continuation line for that table level.

When all levels in the table have been printed, PBUF returns to EOF.

4. EXIT:

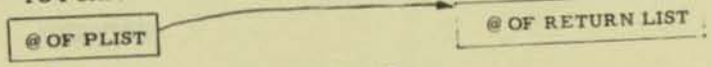
PBUF returns to EOF.

5. SUBROUTINES CALLED:

PCRF - to reference the precedence table which is to be printed.
TIMTAB - to receive the approximate execution time for a block of code.
PRINT - to print out the 120-byte buffer area defined in PBUF.

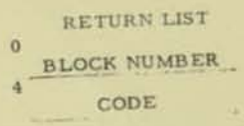
6. PARAMETERS:

TO PCRF:



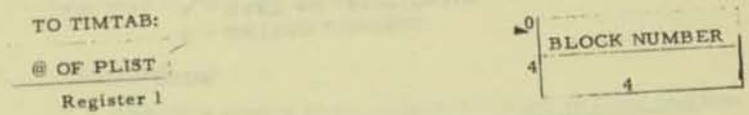
Register 1

FROM PCRF:



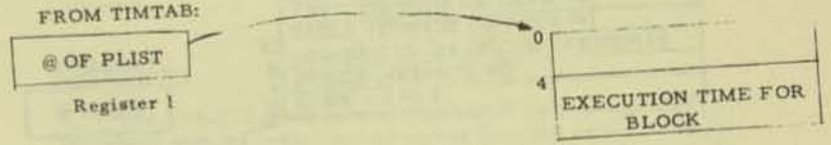
CODE:
- 0 - This is last block of a level.
- 4 - There are no more levels to be printed.
- 8 - There are still more block numbers on this level.

TO TIMTAB:



Register 1

FROM TIMTAB:



Register 1

TO PRINT:

Register 1

7. LIMITATIONS:

The maximum execution time for a block of code should not exceed 9,999 milliseconds, not including any looping within the block. A note explaining this situation is printed with the precedence table.

SUBROUTINE LABDIC

1. FUNCTION:

LABDIC builds and references a symbol dictionary consisting of labels and variable names.

2. ENTRANCE:

LABDIC is entered by PASS 1 to insert labels, by PASS 2 to insert and use symbols and to use labels, and by VARTAB and BRVBUF to resolve I.D. numbers to symbols.

3. OPERATION:

When LABDIC is first called, its tables and work areas are initialized to zero. The routine can either store or retrieve information. Its action is determined by the flag field in the parameter list. If the table is to be built, a symbol is placed in the table area, referenced with an I.D. number in the index, and if required, it returns information to the caller. If the table is to be referenced, the index is searched for a matching I.D. and the address and length of the corresponding symbol are returned.

This routine allows the use of uniform I.D. numbers in the place of variable type, variable length symbols.

4. EXIT:

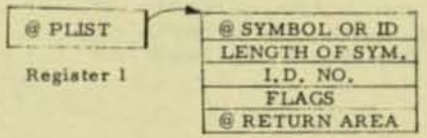
LABDIC returns control to the calling routine.

5. SUBROUTINES CALLED:

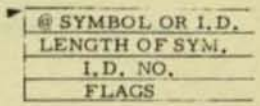
ERROR: To print an error message and take an ABEND dump.

6. PARAMETERS:

TO LABDIC:



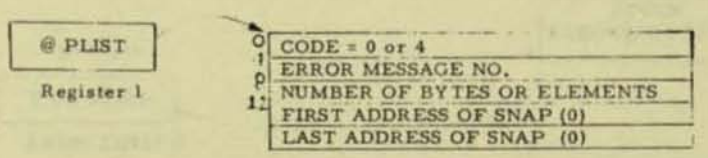
From LABDIC:



FLAGS: Bits 0-31

Bit On	Meaning
22	Re-initialize re-initialization switch
23	Referencing (OFF-Building)
24	Symbol (OFF-Label)
25	DO Statement - End of Block
26	IF or GO TO Statement
27	Array Definition
28	Built-in-Function
29	Information Requested
30	Should have been previously defined
31	Not Found (return information only)

TO ERROR:



CODE:

0 - RETURN CONTROL
4 - TAKE AN ABEND DUMP

7. LIMITATIONS:

The major limitation of this routine is table size. If the defined table size is insufficient, execution will cease. Error messages number 7, index overflow, or number 8, dictionary overflow, will be given.

When retrieving information, a check is made on the I.D. number in the dictionary. If it doesn't correspond with the one in the index, error message number 16 is given.

If the I.D. number requested when referencing is not in the index, error message number 15, is given.

When PASS 2 is building and using the table, the "Should have been previously defined" bit may be on. If that symbol is not found, error message number 36 is given.

8. TABLES:

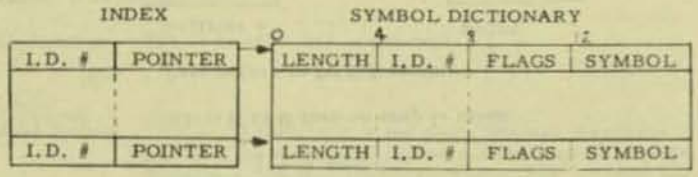


FIGURE 7

Index Pointer: Points to beginning of table element with corresponding I.D. number. If it is zero, no table entry exists.

Length: Length of symbol

Flags: Same as parameter list flags in section 6.

Symbol: EBCDIC representation; variable length.

Symbols are padded to the right with zeros if they do not end on a full-word boundary. All table elements start on full word boundaries.

SUBROUTINE ERROR

1. FUNCTION:

ERROR puts the appropriate error message in the buffer to be printed by PRINT.

2. ENTRANCE:

ERROR is entered by any routine that wants an error message recorded.

3. OPERATION:

The ERROR routine puts into the buffer the message that is associated with the error number passed from the caller. Control is either returned to the caller or the job terminated. If the job is to be terminated an abend dump is supplied. If control is to be returned a snap dump may be specified by the caller.

4. EXIT:

ERROR either returns to its calling routine or terminates the job.

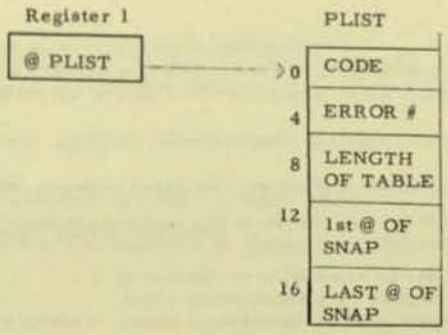
5. SUBROUTINES CALLED:

ERROR references the following subroutine:

a.) PRINT - to do the actual printing of the buffer.

6. PARAMETERS:

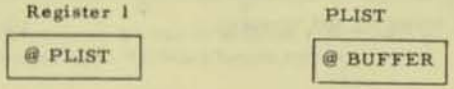
a.) Parameters received from calling routine:



CODE: 0 ⟶ return
4 ⟶ abort

If last @ is Ø then no snap is taken.

b.) Parameters to PRINT routine:



7. LIMITATIONS:

Error messages are limited to 119 bytes.

ERROR MESSAGES FOR FAP

1. Table allocation is _____ entries (8 bytes per entry) for the Blks Ref By Var. table dictionary. It is insufficient.

   Explanation: The thumb-index table for the Block Referenced By Variables build routine has overflowed.

2. Table allocation is _____ entries (8 bytes per entry) for the Blks Ref By Var. Table. It is insufficient.

   Explanation: The Block Referenced By Variables Table has overflowed.

3. Table allocation is _____ bytes for the Var. Used In Blk Table. It is insufficient.

   Explanation: The Variables Used In Block Table has overflowed.

4. Table allocation is _____ elements (4 bytes per element) for Var. Used In Blk table dictionary. It is insufficient.

   Explanation: The thumb-index table for the Var. Used In Blk build routine has overflowed.

5. Table allocation is _____ elements (4 bytes per element) in the blk. def. table dictionary. It is insufficient.

   Explanation: The thumb-index table for the Block Definition Table routine has overflowed.

6. Table allocation is _____ bytes in the Block Definition Table. It is insufficient.

   Explanation: The Block Definition Table has overflowed.

7. Table allocation is _____ elements (12 bytes per element) in the Symbol Table dictionary. It is insufficient.

   Explanation: The Symbol Table index freespace pointer is at, or past the end of the index.

8. Table allocation is _____ bytes in the Symbol Table. It is insufficient.

   Explanation: The symbol table freespace pointer is at or past the end of the table, therefore, overflow has occurred.

9. Table allocation is _____ elements (8 bytes per element) for the X-Ref table. It is insufficient.

Explanation: The Cross Reference Table freespac pointer is at, or past the end of the table. Overflow has occurred.

10. Table allocation is _____ elements (4 bytes per element) for X-Ref dictionary. It is insufficient.

Explanation: The index pointer is at or past the end of the table. The pointer is dependent on the branch block number, which is used to address the correct element in the index. If this condition arises the branch block number is greater than the number of elements allowed in the dictionary table.

11. Table allocation is _____ elements (16 bytes per element) for Precedence table. It is insufficient.

Explanation: The Precedence table has overflowed.

12. Table allocation is _____ elements (4 bytes per element) for Precedence table dictionary. It is insufficient.

Explanation: The thumb-index table for the Precedence diagram routine has overflowed.

13. Table allocation is _____ elements (4 bytes per element) for the Time Table. It is insufficient.

Explanation: The Time table has overflowed.

1-13.

User Action: To correct the error in the above thirteen messages increase the storage allocation for the particular table. To do this only the "DS" card that defines the table need be changed. All other considerations will be taken care of by the program itself.

14. Invalid function code received from scan routine. Job terminated.

Explanation: The code received from Pass 2 that BDT uses in its decision table operation is less than 0 or greater than 8 and therefore invalid.

User Action: Check parameters passed from Pass 2 at time of abend. Change code in first word of parameter list so it is valid.

15. The I.D. presented to the symbol table is invalid.

Explanation: The index was completely searched and the I.D. was not found.

User Action: Check the I.D. number being requested. Also, check the index contents.

16. The I.D. in the symbol table does not correspond with the search I.D. presented.

Explanation: The I.D. in the table element which is pointed to by the index element whose I.D. matches the search I.D., does not correspond with that index I.D.

User Action: Check table and index contents for incorrect information.

17. Incorrect variable received from Symbol-Label dictionary.

Explanation: If the I.D. number passed to LABDIC is not the same as the I.D. in the return list from LABDIC, the corresponding variable is incorrect.

User Action: Check the I.D. number and corresponding variable in LABDIC.

18. Invalid code from calling routine.

Explanation: The code received by the Variables Used in Block Table routine from the calling routine is not 0, 4, or 8.

User Action: Correct the code passed by calling routine.

19. Abnormal I/O operation for source code input - check device.

Explanation: Data management signified an irrecoverable I/O error. The program will not function properly.

20. Improper end of file reached on intermediate text input - end card may not exist.

21. Information requested from X-Ref table does not exist.

Explanation: The block number requested is greater than the highest block number entered.

User Action: Check why that block number was requested.

22. Block number to be referenced does not exist in table. Job terminated.

Explanation: BDT is to reference a block that does not exist.

User Action: Check why caller asked for that block.

23. The code passed to the referencing routine of the Blocks Referenced By Variables table is invalid.

Explanation: The codes passed to the referencing routine are 0, 4, 8, 12, 16, 20. If a code is less than 28 and not one of the above numbers, this message will be printed.

User Action: Test parameter lists coming from VARUSE, MPRC and BRVBUF.

24. Invalid error message number passed to Error message routine.

User Action: Determine routine that passed error number and correct.

25. I/O operation error on intermediate text input - check device.

Explanation: Data management signified an irrecoverable I/O error. The program will not function properly.

26. Abnormal I/O operation on intermediate text output device.

Explanation: Same as 25.

User Action: Check device.

27. Opening of data set for system output routine unsuccessful, check JCL.

28. Opening of data set for system input routine unsuccessful, check JCL.

29. Opening of data set for intermediate text input routine unsuccessful, check JCL.

30. Illegal parameter received by PRINT routine for carriage control.

User Action: Check parameters passed to PRINT routine.

31. Improper setting of switch in pass 2, investigate.

Explanation: Error in switching technique used in determining the routine to receive control.

32. Illegal special character.

Explanation: Character in Fortran source code is not included in standard Fortran character set.

User Action: Program could have scanned a blank when it should not have.

33. Incorrect flag received by VARUSE from PASS 2.

Explanation: Incorrect flag bit turned on, causing VARUSE to end the VARIABLES USED BY BLOCK TABLE, but before the table can be reinitialized, another entry is to be made in the table.

User Action: Check the flag bits passed from calling routine to VARUSE.

34. Beginning of thumb-index reached. Search for level-address is in error.

Explanation: Used as prevention for looping when searching through the thumb-index table. In MPRC. It is issued only if the first element of the thumb-index is reached.

User Action: Check instructions and registers working with level numbers in MPRC.

35. The BLOCK REFERENCED BY VARIABLES ROUTINE is not equipped to handle block numbers greater than 9999.

36. SYMBOL TABLE index does not correspond with the table element.

Explanation: When building with a label that is already in the table the corresponding index element does not exist. This is determined when searching the index to insert the I.D.

User Action: Check the index and table to see if the entries corresponding, if not, determine which routine caused the error.

37. The BLOCK REFERENCED BY VARIABLES TABLE has no information in it. Possible error.

Explanation: This message appears when BRVBUF goes to print the BLOCK REFERENCED BY VARIABLES TABLE and finds that there is no information in it.

User Action: It is possible that this is a legitimate situation but highly improbable. User should check parameters from BRVBLD to BRVBUF and from VARUSE to BRVBLD.

38. Illegal block number received by TIME TABLE from PBUF.

Explanation: PRECEDENCE ROUTINE passed block number that is not in TIME TABLE.

User Action: Check parameter list to TIME.

39. Decision table code received from caller is negative. This is invalid.

Explanation: Function code in first word of parameter list passed to BDT is invalid.

User Action: Check the development of the parameter list by caller.

SUBROUTINE KEYTAB

1. FUNCTION:

KEYTAB searches the basic FORTRAN keyword table and determines if the symbol given by PASS 1 is a keyword. It then returns information to PASS 1.

2. ENTRANCE:

KEYTAB is entered when PASS 1 comes across a symbol as the first field in the source code it is scanning.

3. OPERATION:

The symbol given to KEYTAB by PASS 1 is compared to all the keywords in the appropriate keyword table. These tables are broken down according to length. If the symbol is found, information concerning the keyword is returned to PASS 1. If it is not found a return code is given to PASS 1.

4. EXITS:

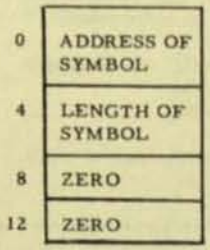
When the operation is complete, control is returned to PASS 1.

5. LIMITATIONS:

The keywords used are those supplied by basic FORTRAN IV.

6. PARAMETERS:

To KEYTAB: A 4-word parameter list (see figure 8).



1 word

FIGURE 8

KEYTAB returns to PASS 1 status information in the third and fourth words of the parameter list.

1. TIME
2. FLAGS - Status information for intermediate text. Consists of eigth flags, one for each type of FORTRAN statement.
3. CODE - Returned in second word of parameter list.
   0 - Not found
   4 - Found

SUBROUTINE OPTAB

1. FUNCTION:

OPTAB searches the FORTRAN special character table and determines if the character supplies by PASS 2 is a FORTRAN special character. It then retrieves the time and returns to PASS 2.

2. ENTRANCE:

OPTAB is entered whenever PASS 2 comes across a special character in its assignment statement scan.

3. OPERATION:

When OPTAB receives control the operator table is searched to see if the character is in the table. If it is the time and a return code are returned to PASS 2. If it is not, a return code is returned to PASS 2 so signifying.

4. EXITS:

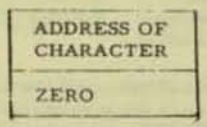
When OPTAB has completed its operation, control is returned to PASS 2.

6. PARAMETERS:

To OPTAB - A one word parameter list which points to the special character.

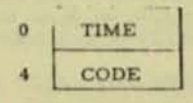
From OPTAB - A two word return list for the time and a return code.

To OPTAB:



1 word

From OPTAB:



CODE = 0 - Not found
CODE = 4 - Found

SUBROUTINE BDFUNC

1. FUNCTION:

BDFUNC searches the FORTRAN built in function table and determines if the symbol given by PASS 2 is a built in function. It then returns to PASS 2 information pertaining to the symbol.

2. ENTRANCE:

BDFUNC is entered whenever PASS 2 comes across a variable on the right side of an equal sign in an assignment, and this variable is not in the symbol dictionary.

3. OPERATION:

When BDFUNC receives control the following takes place:

The length of the symbol is determined and the proper table is searched for the symbol given.
If it is found, the time for that function is retrieved and returned to PASS 2.
If it is not in the table, a code signifying this is returned to PASS 2.

4. EXITS:

When the operation is complete control is returned to PASS 2.

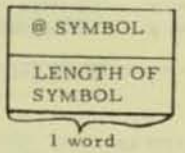
6. LIMITATIONS:

In its present form, BDFUNC handles only build-in functions now specified by IBM basic FORTRAN IV.

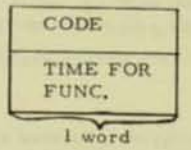
7. PARAMETERS:

To BDFUNC - PASS 2 supplies a 2 word parameter list.
From BDFUNC - Returns to PASS 2 a one word parameter list - which is the second word of PASS 2's supplied list and a return code.

To BDFUNC:



From BDFUNC:



CODE = Not in table
CODE = In table

LOGIC

1. FUNCTION:

LOGIC searches a table of LOGICAL IF COMPARATORS to determine if the symbol is a logical IF designator.

2. ENTRANCE:

LOGIC is entered whenever PASS 2 is processing an IF statement.

3. OPERATION:

When LOGIC receives control, a special table is searched to see if the element is in the table. If it is, a code is sent to the calling routine to designate a LOGICAL IF is being processed. If not, another code is passed.

4. EXITS:

When LOGIC has completed its operation, control is returned to PASS 2.

5. PARAMETERS:

0	@ELEMENT
4	LENGTH
8	CODE

CODE = 0 not a logical IF
4 Logical if

SUBROUTINE TIMTAB

1. FUNCTION:

TIMTAB builds up a table by block number of total time for each block of code and then supplies these times (microseconds) to the precedence diagram (PBUF).

2. ENTRANCE:

TIMTAB is entered from both PASS 2 and PBUF. PASS 2 sends the total time of execution per block to TIMTAB and PBUF asks for the total time by block number for output purposes.

3. OPERATION:

When TIMTAB receives control the following takes place:

1. The code is tested to determine if PASS 2 is making an entry or PBUF is referencing the table.
   Code = 0 - Enter time
   Code = 4 - Reference table
2. If referencing the table, the entry for the given block number is retrieved and returned to PBUF.
3. If entering an element into the table, the time is put into the next available space of the table and the index is updated.

4. EXITS:

On completion of the operation control is returned to the calling routine.

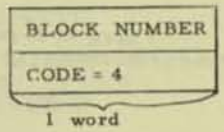
6. LIMITATIONS:

The time for each block must be entered sequentially by block number. Only 125 entries can be handled at the present time (Error - 13).

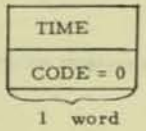
7. PARAMETERS:

To TIMTAB: A two word parameter list from both PBUF and PASS 2.
From TIMTAB: A one word return to PBUF giving the time for the given block number.

From PBUF:



From PASS2:



SUBROUTINE TIME

1. FUNCTION:

TIME accumulates the time for each block of code that PASS 2 processes.

2. ENTRANCE:

TIME is entered when PASS 2 comes across an operation that would involve CPU time in execution. It is also entered when PASS 2 signifies the end of a block.

3. OPERATION:

When TIME receives control the following takes place:

The time sent by PASS 2 is added to the accumulated time already received from PASS 2. The end of block bit is checked to determine if it is end of block. If it is, the total time is passed back to PASS 2 and the time is reset to 0.

4. EXITS:

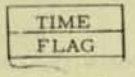
Control is returned to PASS 2 on completion of operation.

6. PARAMETERS:

To TIME: TIME receives a 2 word parameter list from PASS 2
1) First word contains the time to be accumulated.
2) High order bit-on-end of block.

From TIME: TIME returns to PASS 2 with the total time for the block in the first word of the parameter list supplied by PASS 2.

To TIME:



FLAG Bit 0 - on - end of block

SUBROUTINE INTXT

1. FUNCTION:

INTXT reads in an intermediate text record into the user supplied buffer area.

2. ENTRANCE:

INTXT is entered from PASS 2 which processes the intermediate text.

3. OPERATION:

When INTXT receives control the following takes place:

A first time switch is checked to see if the input data set has been opened.
If it has not the data set is opened and the switch is turned on.
The record is read into the user supplied input buffer.
End of file is checked and if it exists the input data set is closed.

4. EXITS:

1) INTXT exits to PASS 2 on normal I/O operation.
2) INTXT exits to ERROR on abnormal I/O operation.

5. SUBROUTINES CALLED:

1) ERROR - To take a dump of storage and to write out the appropriate error message.

6. LIMITATIONS:

INTXT can only be used in conjection with the intermediate text set up by PASS 1.

7. PARAMETERS:

A one word parameter which points to the user supplied buffer area.

SUBROUTINE OUTTXT

1. FUNCTION:

OUTTXT writes out all intermediate text for PASS 1.

2. ENTRANCE:

OUTTXT is entered from PASS 1 which sets up the intermediate text.

3. OPERATION:

When OUTTXT receives control the following takes place:

A first time switch is checked to see if the output data set has been opened. If it has not, the data set is opened and the switch is turned on. The output record in the user supplied buffer is moved to the output buffer of OUTTXT. The record is checked to see if it is an end card. If it is, the data set is closed and the switch is turned off.

4. EXITS:

OUTTXT returns control to PASS 1 on completion of the I/O operation.

5. LIMITATIONS:

OUTTXT can only be used in conjection with the intermediate text set up by PASS 1.

6. PARAMETERS:

To OUTTXT: A 1-word parameter list which points to the user supplied output buffer.

SUBROUTINE READ

1. FUNCTION:

This routine reads the FORTRAN source code into an input buffer for PASS 1.

2. ENTRANCE:

Read is entered from PASS 1.

3. OPERATION:

The READ routine opens the data set the first time it is called. To insure that it is not opened again a first time switch is turned on, and checked each time it is called. The record is read into the READ routine's buffer area and transferred to the user supplied buffer. When end of file is reached a code of 0 is returned to PASS 1.

4. EXITS:

READ exits to PASS 1 on normal I/O operation.
READ exits to ERROR on abnormal I/O operations.

5. SUBROUTINES CALLED:

ERROR - To take a dump of storage and write out an error message.

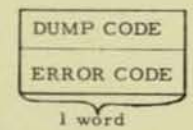
6. LIMITATIONS:

Read can handle only records of length 80 characters.

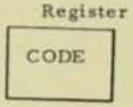
7. PARAMETERS:

To READ - A pointer to PASS 1's buffer area.
From READ - See figure 9

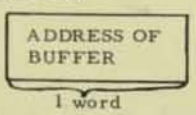
To ERROR:



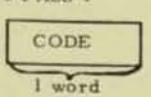
Register 15



To READ:



To PASS 1



0 - End of file
4 - Record processed

Figure 9

SUBROUTINE PRINT

1. FUNCTION:

PRINT writes out all output for the system and controls spacing and new page control.

2. ENTRANCE:

PRINT is entered from all routines that want output to be written out. These routines include PASS 1, BRVBUF, XWRITE, VARTAB, PBUF, and ERROR.

3. OPERATION:

PRINT writes out a 120 byte record that is given to it by the calling routine. PRINT also controls all spacing with 50 lines to a page. New pages can be requested by the calling routine with the proper code.

4. EXITS:

When the record has been printed, control is returned to the calling routine. If an error condition occurs control is passed to the error handling routine.

5. LIMITATIONS:

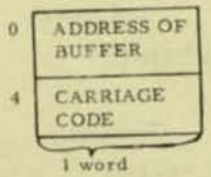
PRINT can handle only records of 120 bytes in length. A wrong carriage code would result in an error (ERROR - 30).

6. PARAMETERS:

To PRINT - A 2 word parameter list - see figure 10
From PRINT - A 2 word parameter list to error - see figure 10

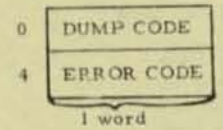
Figure 10

To PRINT:



CODE = 0 - New line
CODE = 4 - New page

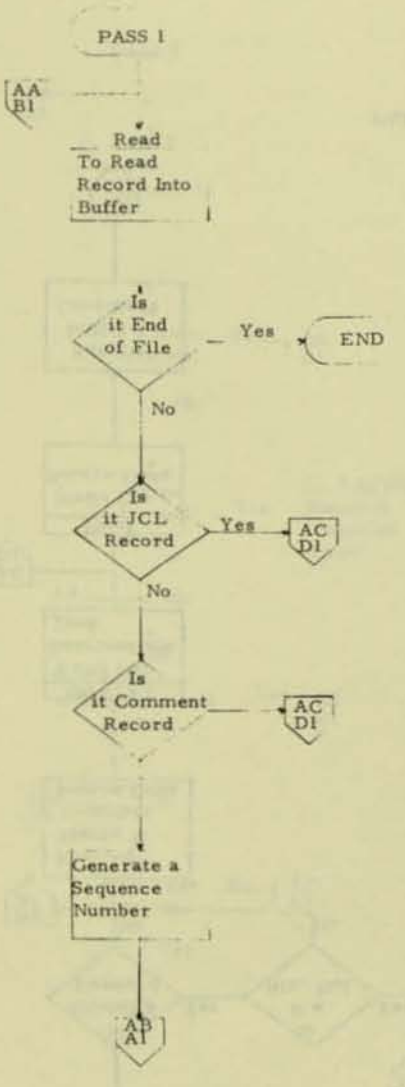
From PRINT:



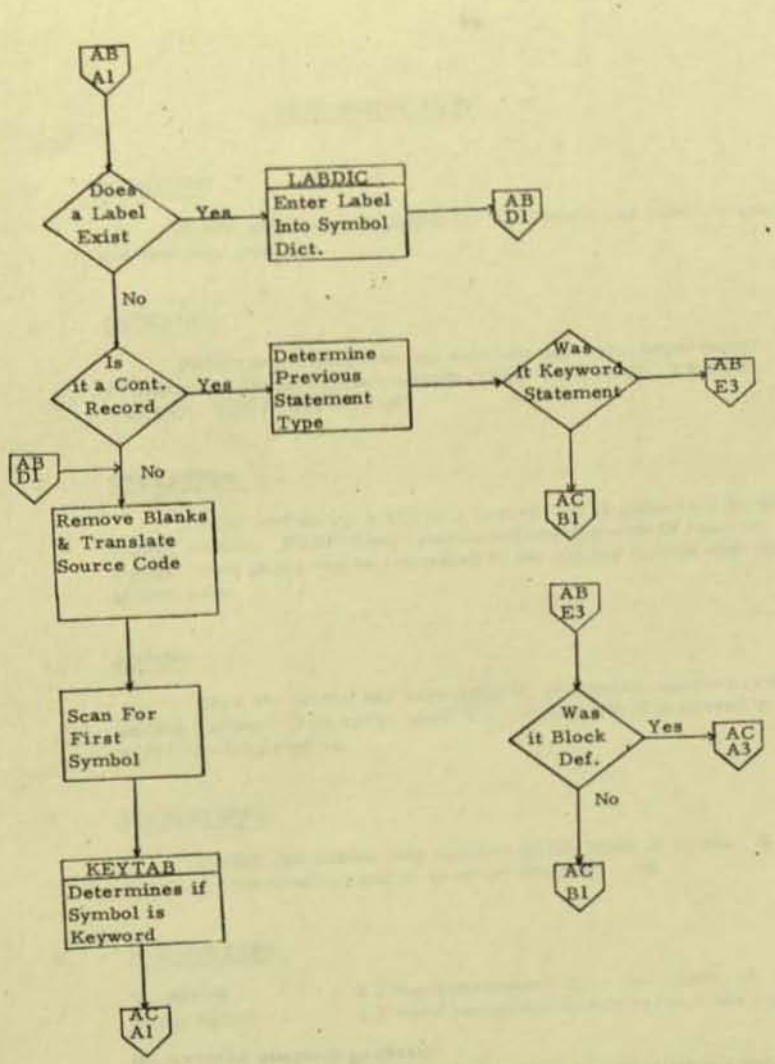
DUMP CODE = 4
ERROR CODE = 30 - Page control code invalid

Flowcharts

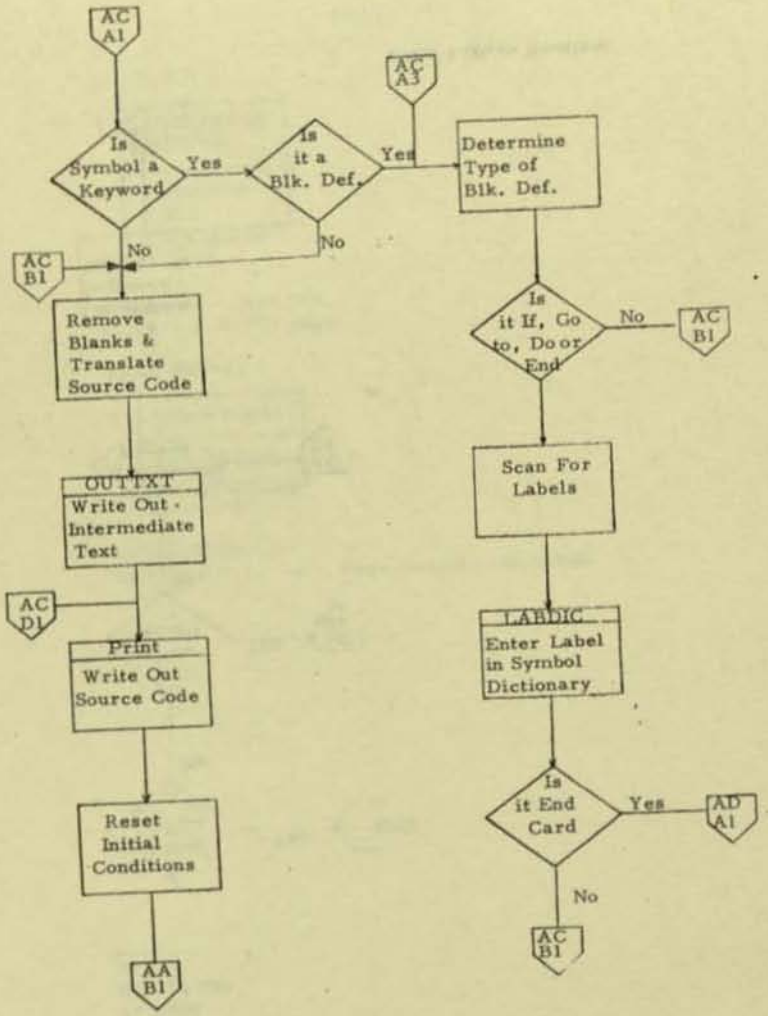
On the following pages are flowcharts of the routines comprising the Fortran Analysis Program.



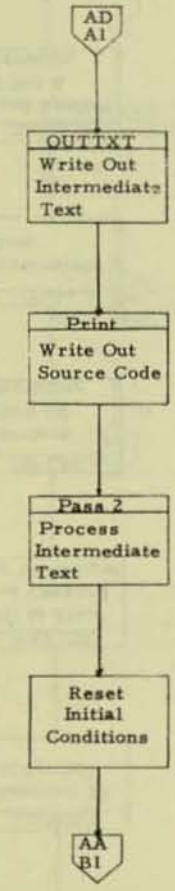
PASS 1 (Scan Routine)



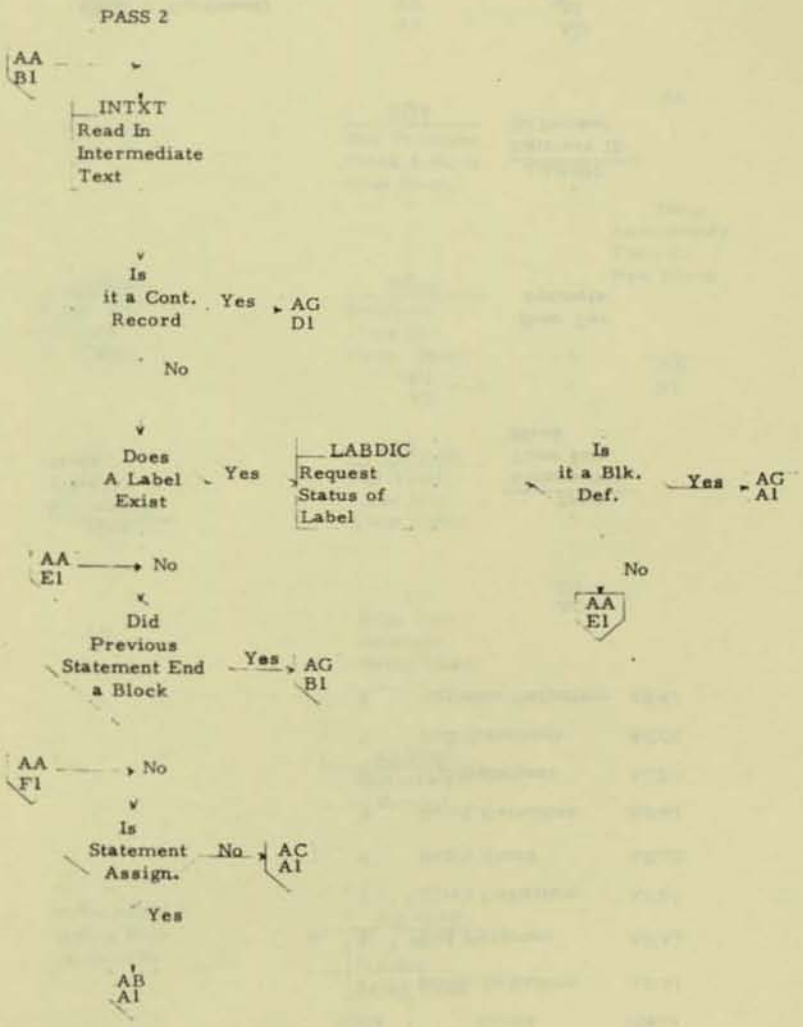
PASS 1 (Continued)



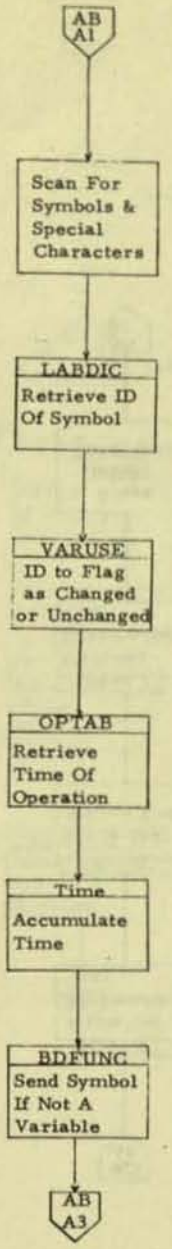
PASS 1 (Continued)

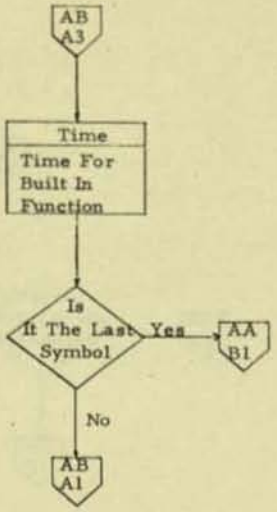


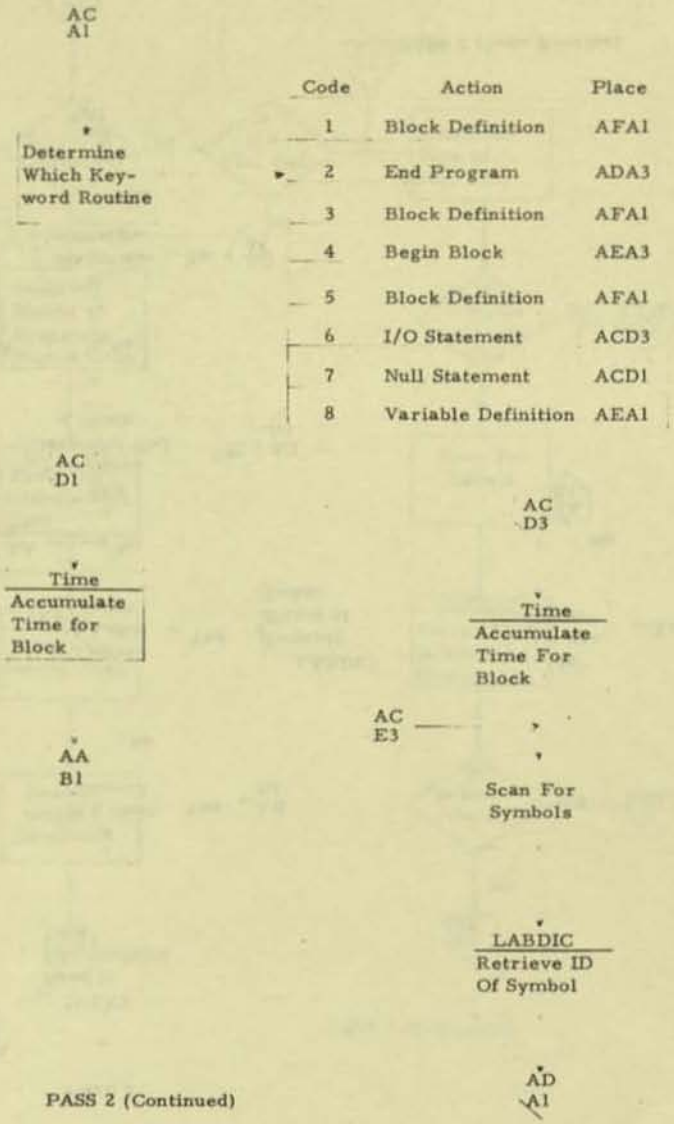
PASS 1 (Continued)

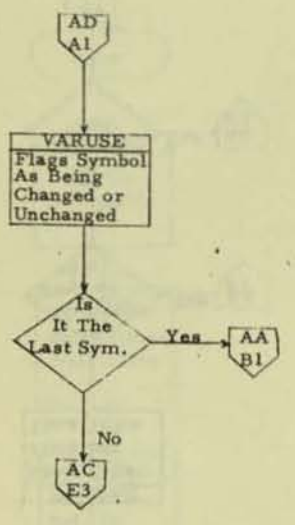


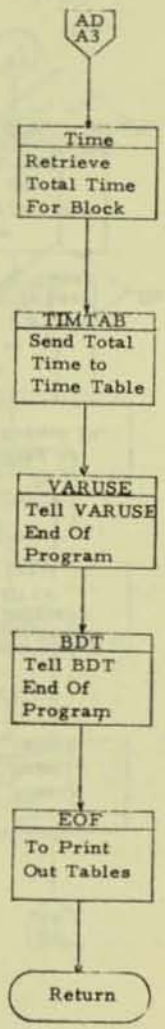
PASS 2 (Scan Routine)


AB
A1
Scan For
Symbols &
Special
Characters
LABDIC
Retrieve ID
Of Symbol
VARUSE
ID to Flag
as Changed
or Unchanged
OPTAB
Retrieve
Time Of
Operation
Time
Accumulate
Time
BDFUNC
Send Symbol
If Not A
Variable
AB
A3

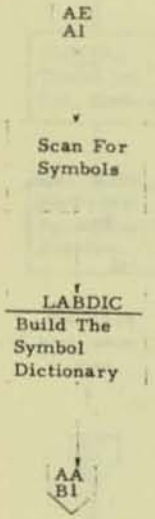

AB
A3
Time
Time For
Built In
Function
Is
It The Last
Symbol
Yes
AA
B1
No
AB
A1

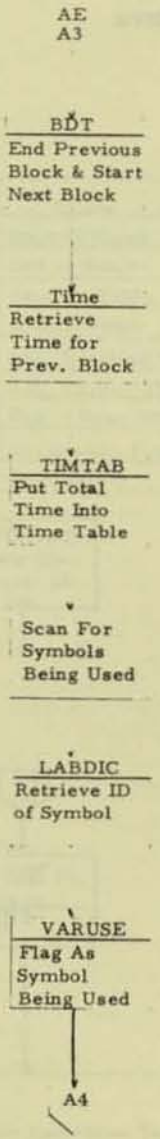

AC
A1
Determine
Which Key-
word Routine
Code
Action
Place
1
Block Definition
AFA1
2
End Program
ADA3
3
Block Definition
AFA1
4
Begin Block
AEA3
5
Block Definition
AFA1
6
I/O Statement
ACD3
7
Null Statement
ACD1
8
Variable Definition
AEA1
AC
D1
Time
Accumulate
Time for
Block
AA
B1
AC
D3
Time
Accumulate
Time For
Block
AC
E3
Scan For
Symbols
LABDIC
Retrieve ID
Of Symbol
AD
A1

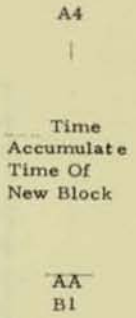




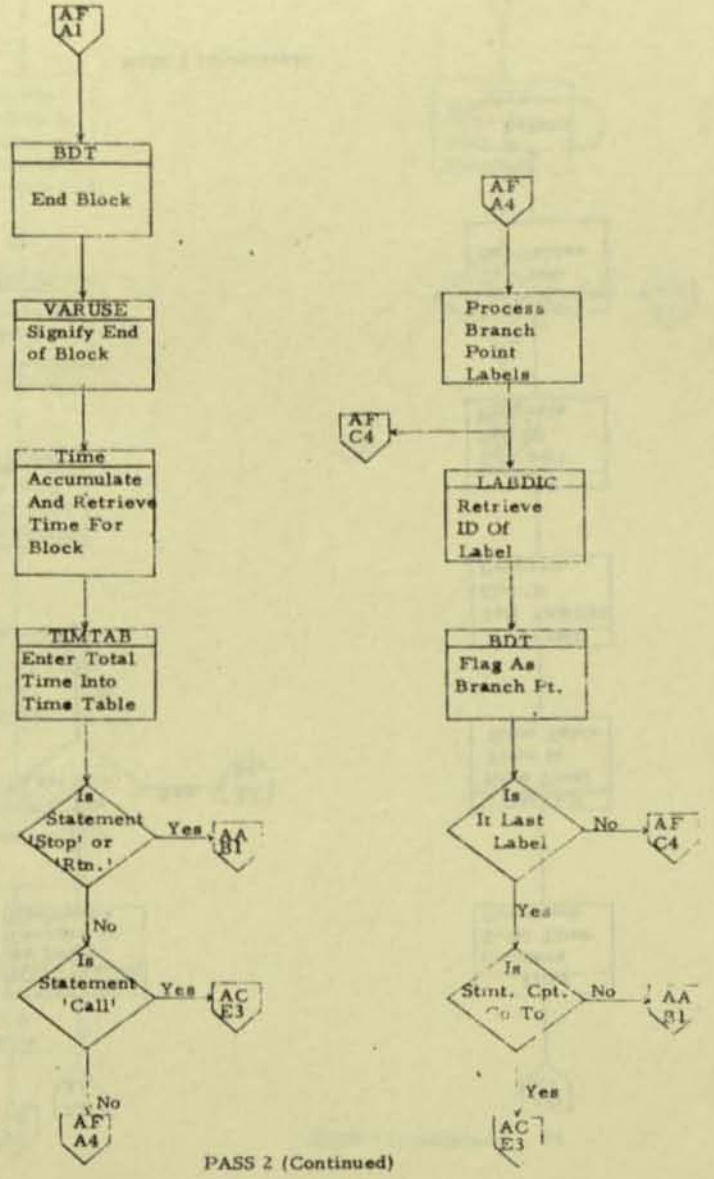
PASS 2 (Continued)



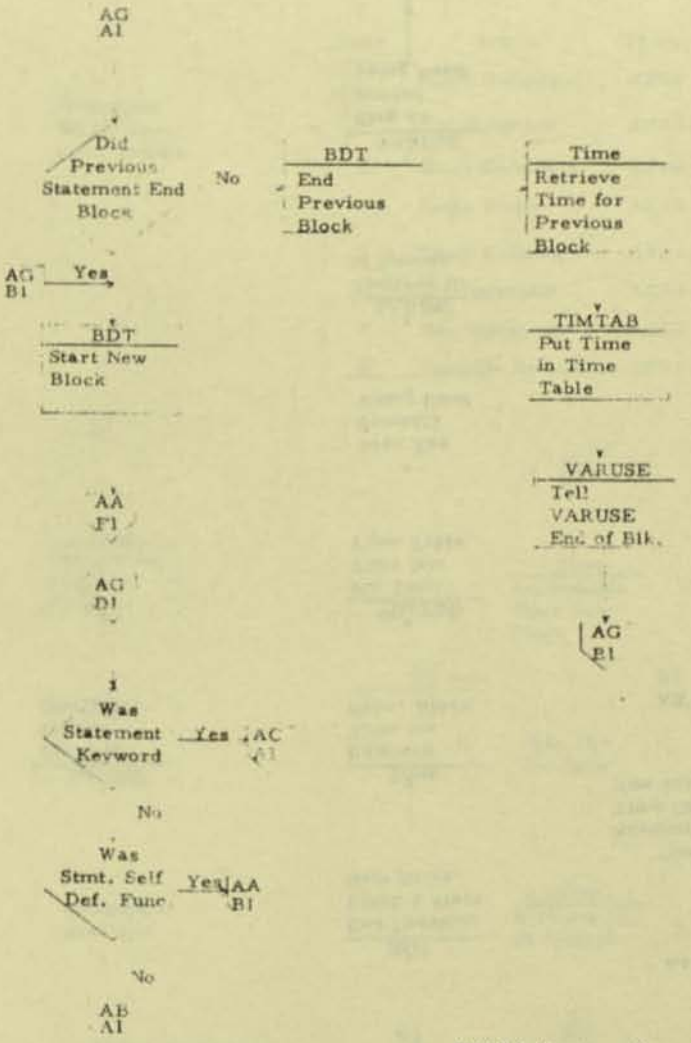




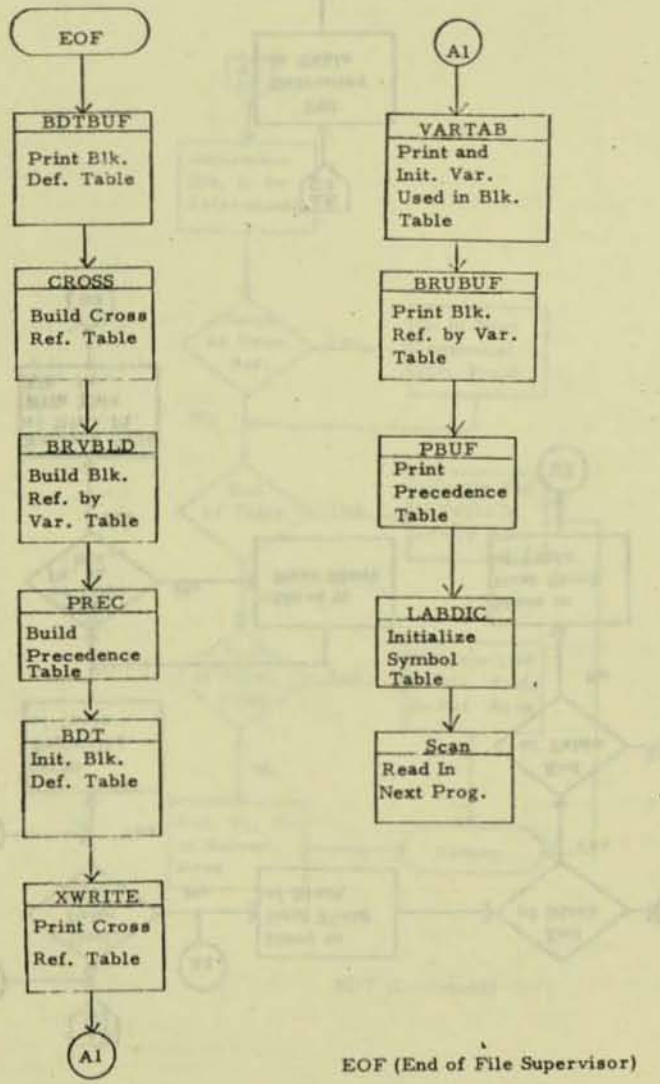
PASS 2 (Continued)



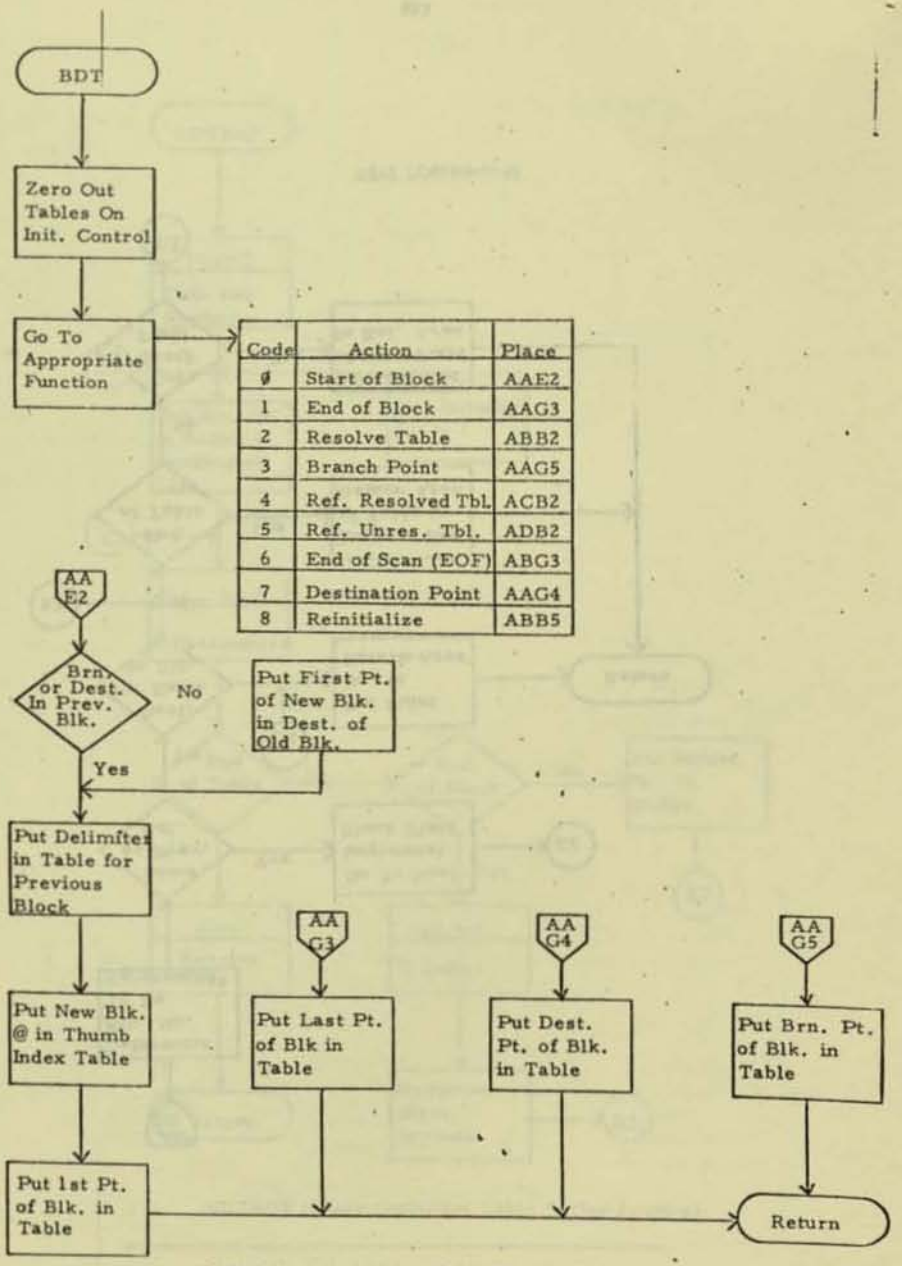
PASS 2 (Continued)



PASS 2 (Continued)

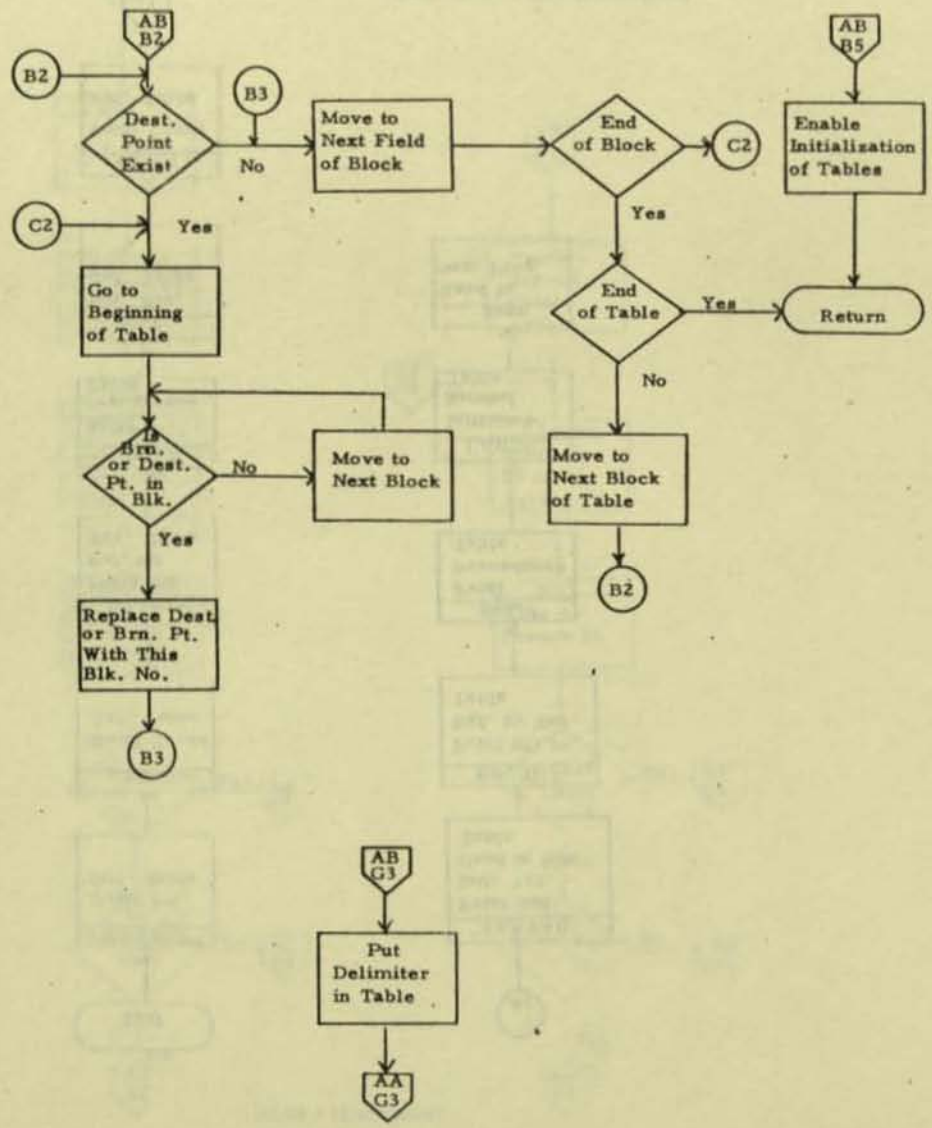


EOF (End of File Supervisor)

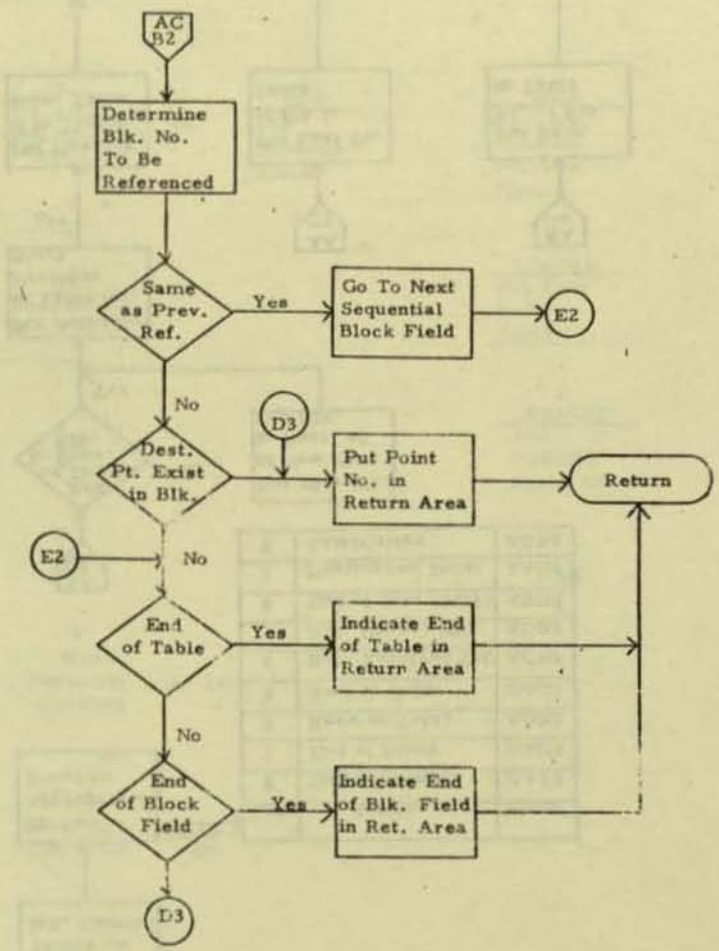


Code	Action	Place
0	Start of Block	AAE2
1	End of Block	AAG3
2	Resolve Table	ABB2
3	Branch Point	AAG5
4	Ref. Resolved Tbl.	ACB2
5	Ref. Unres. Tbl.	ADB2
6	End of Scan (EOF)	ABG3
7	Destination Point	AAG4
8	Reinitialize	ABB5

BDT (Block Definition Table Routine)

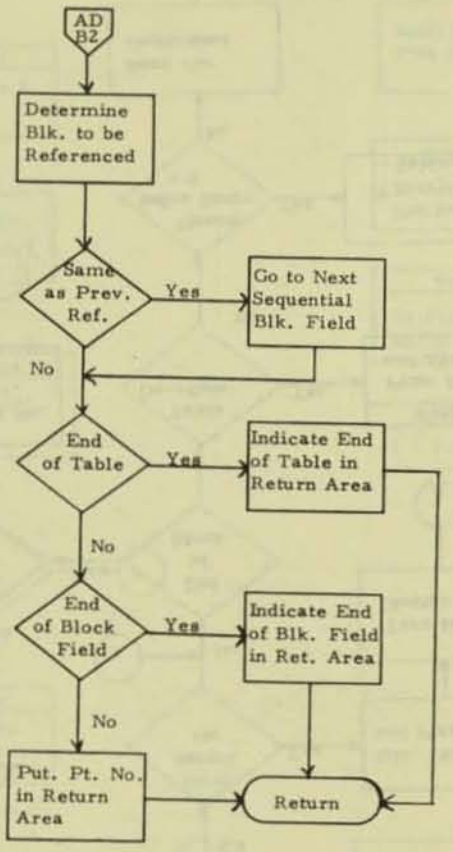


BDT (Continued) 102

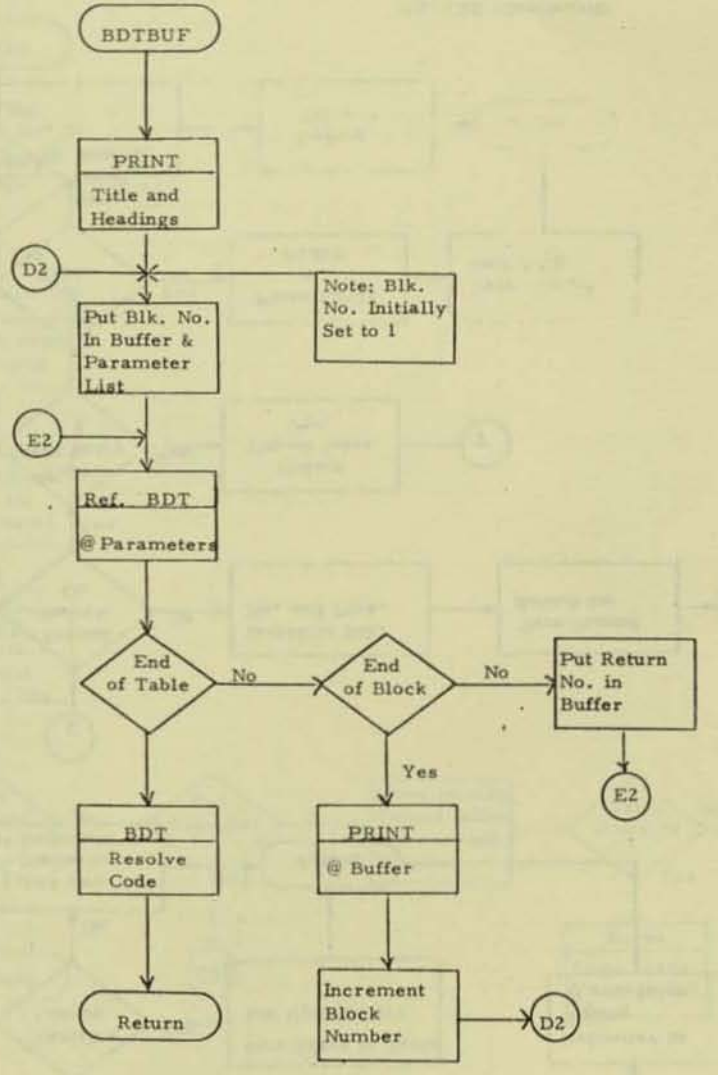


BDT (Continued)

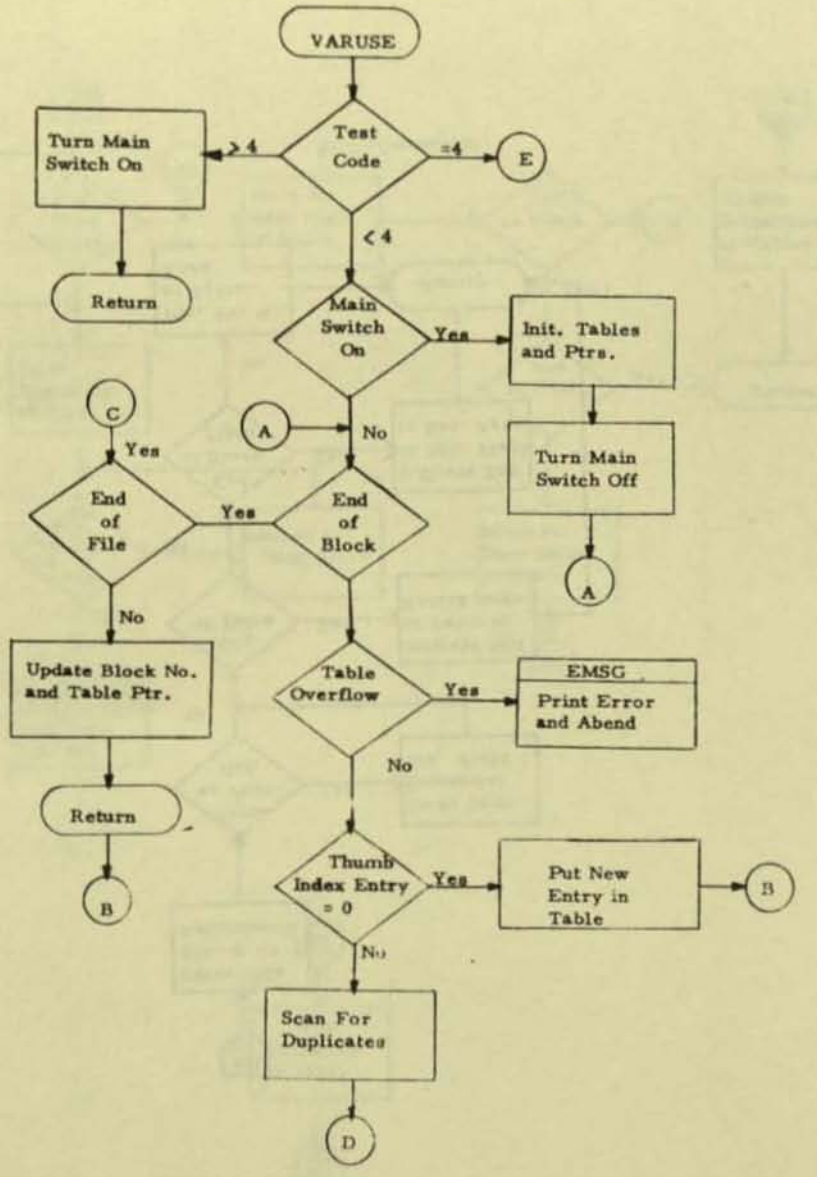
103



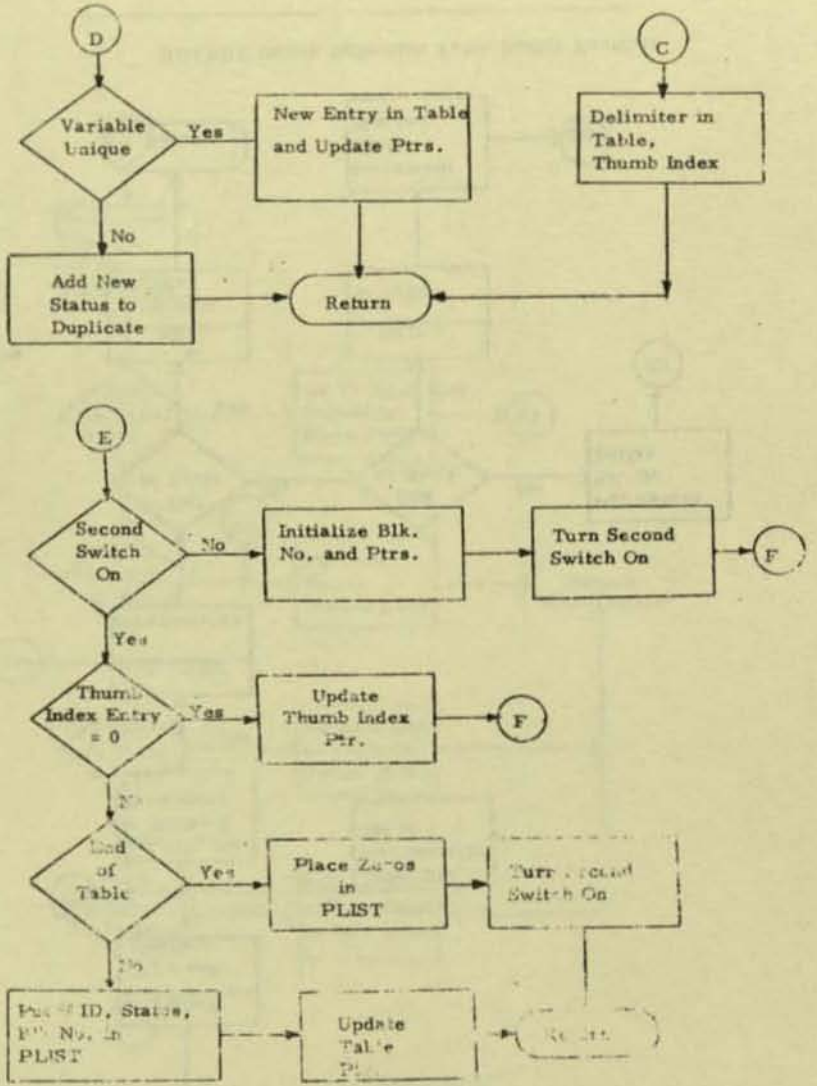
BDT (Continued)



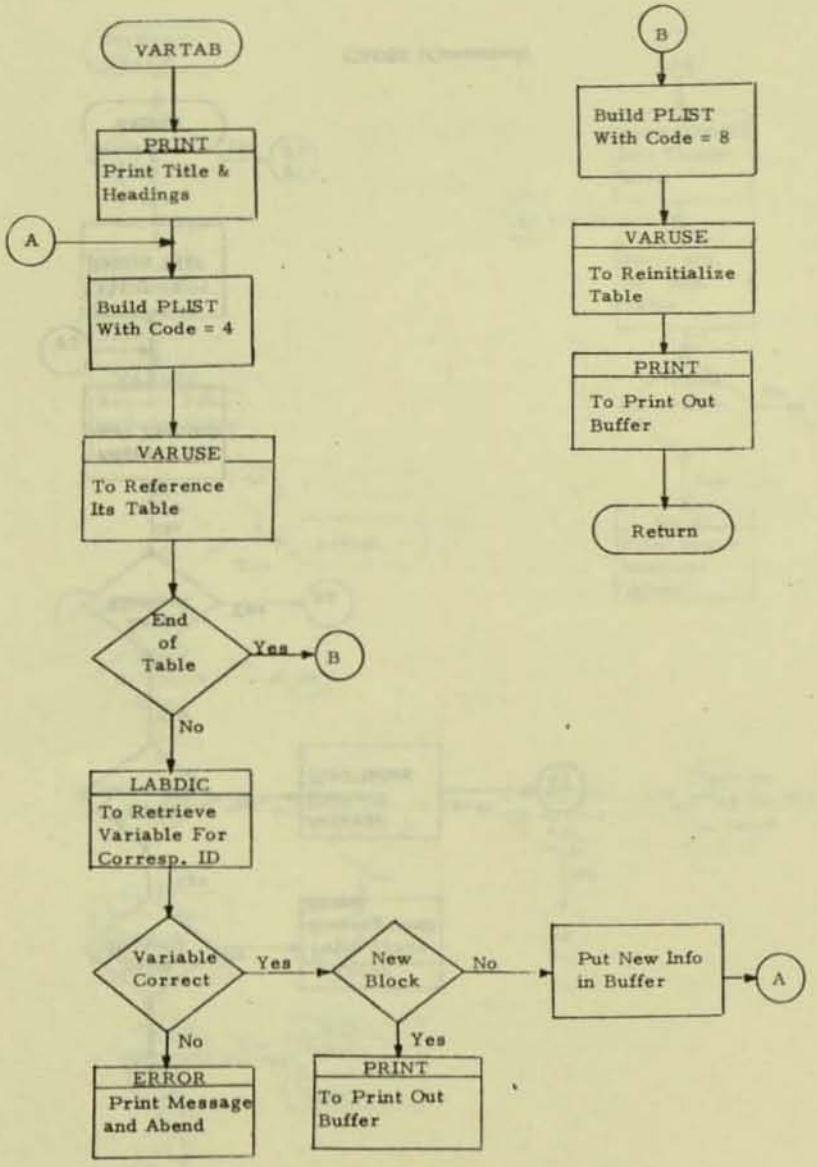
BDTBUF (Block Definition Table Buffer Routine)



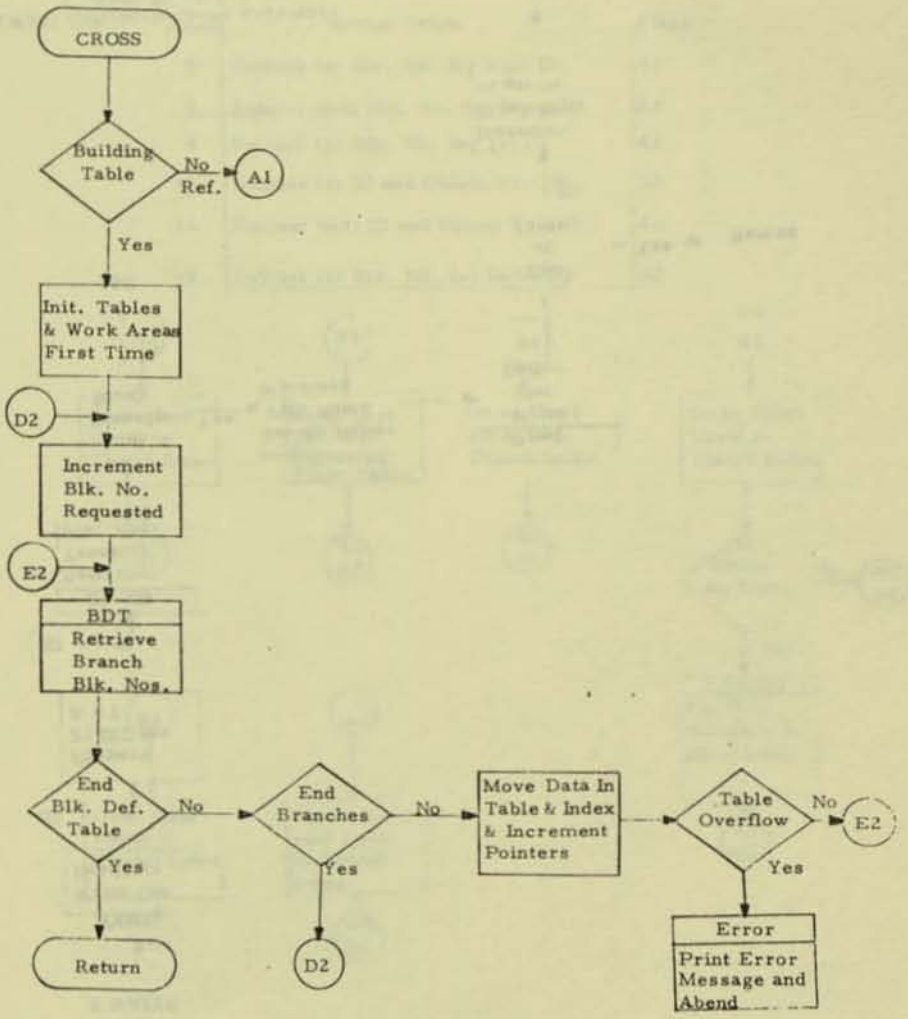
VARUSE (Build, Reference, Variables Used by Block Table)



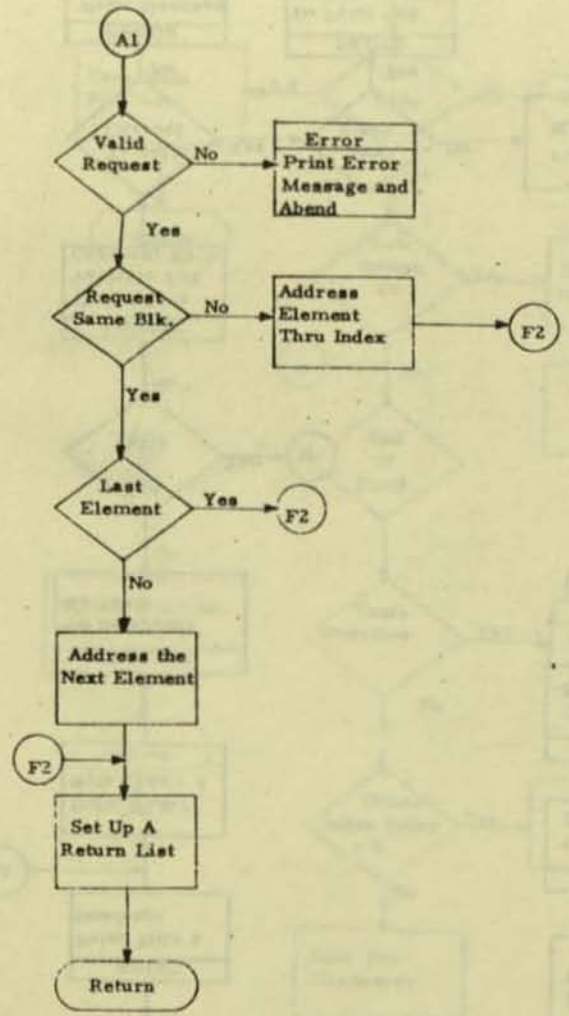
VARUSE (Continued)



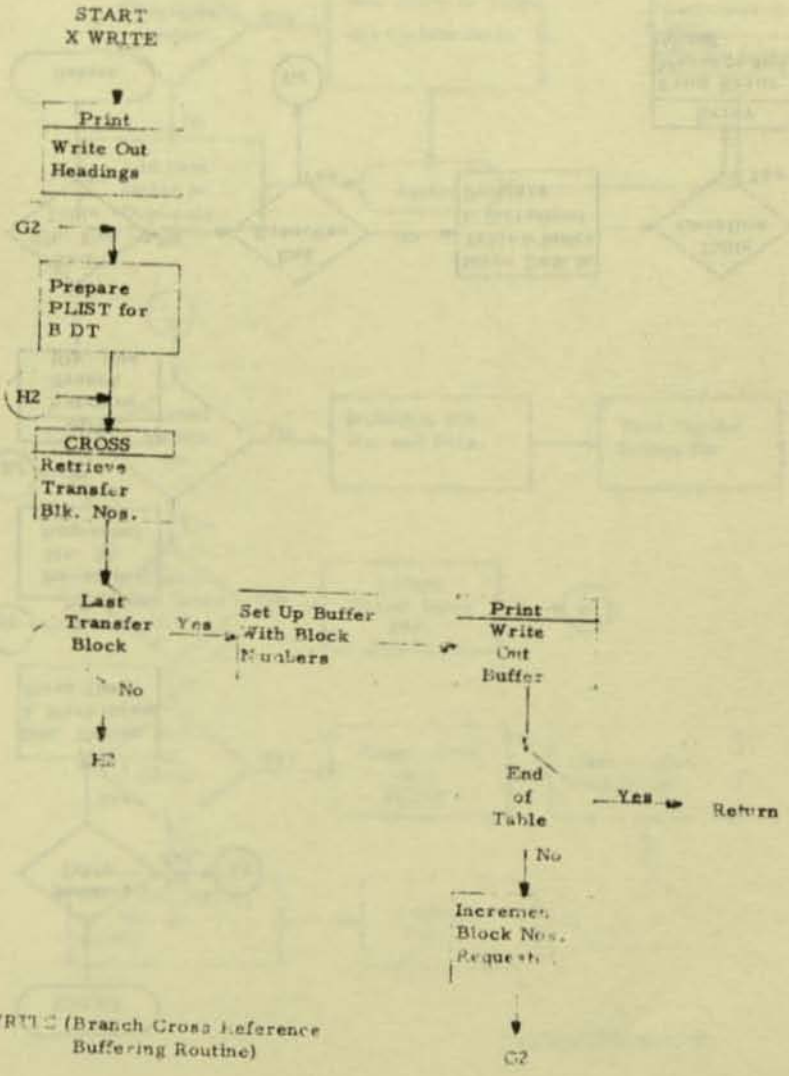
VARTAB (Variables Used By Block Buffer Routine)



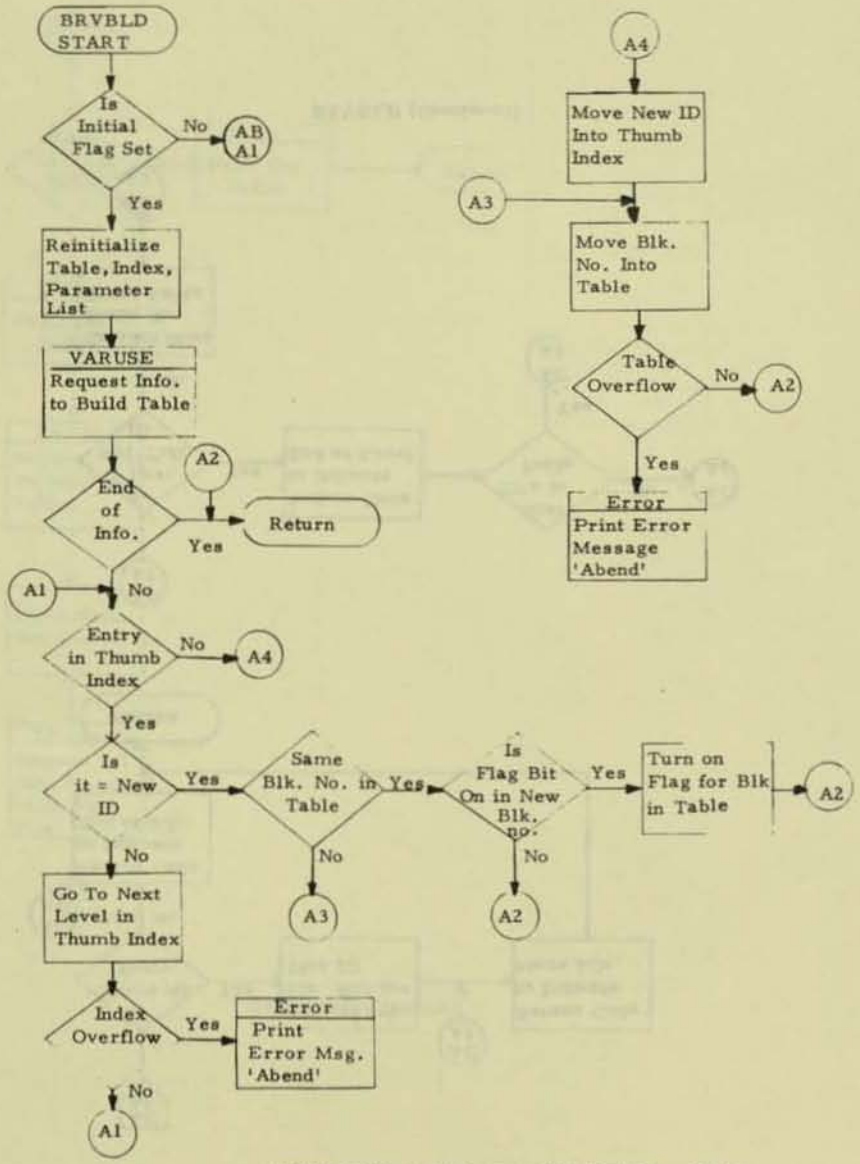
CROSS (Branch Cross-Reference Table)



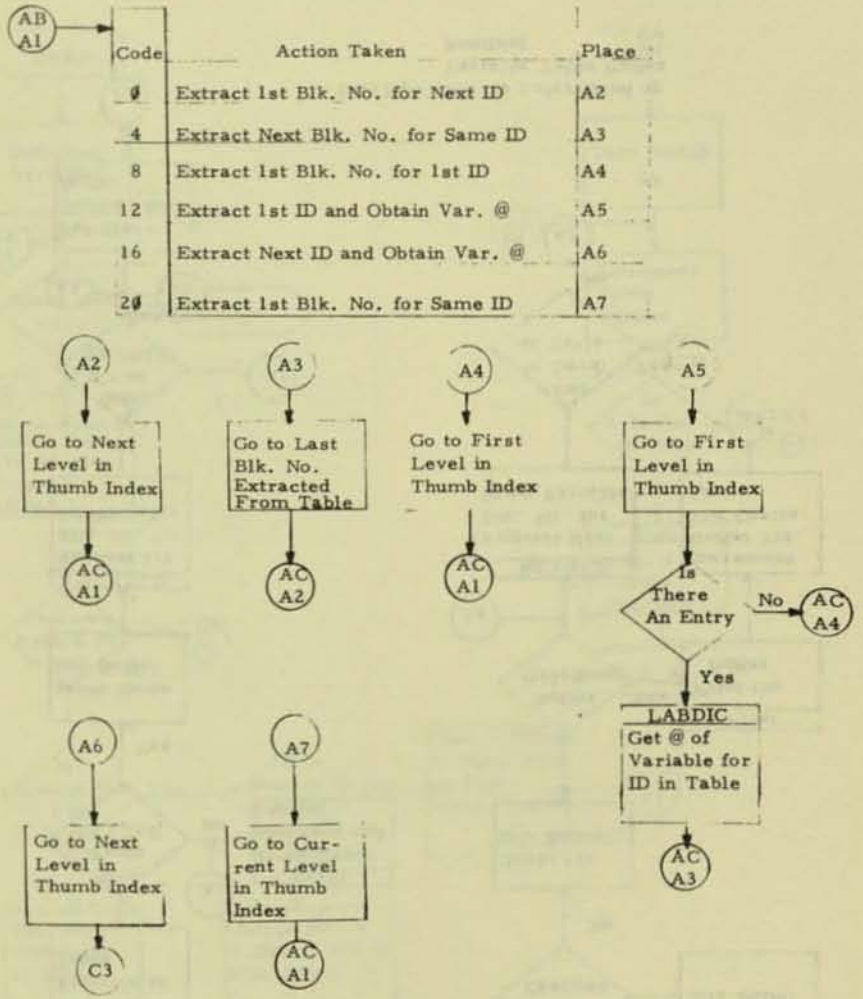
CROSS (Continued)



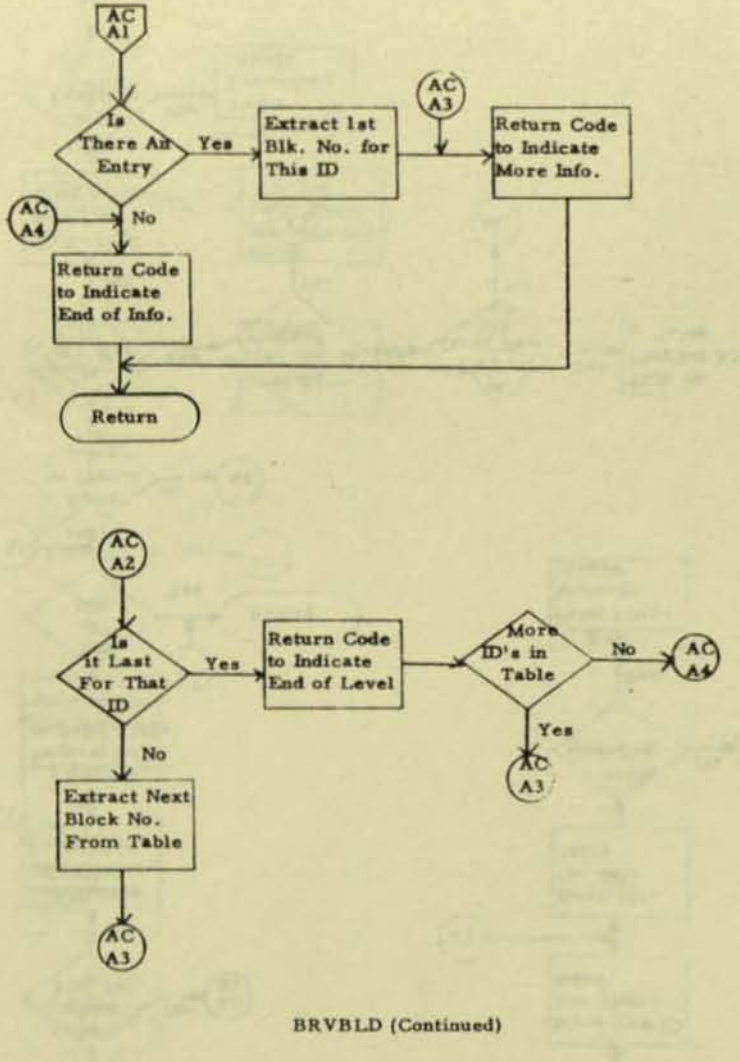
XWRITE (Branch Cross Reference Buffering Routine)



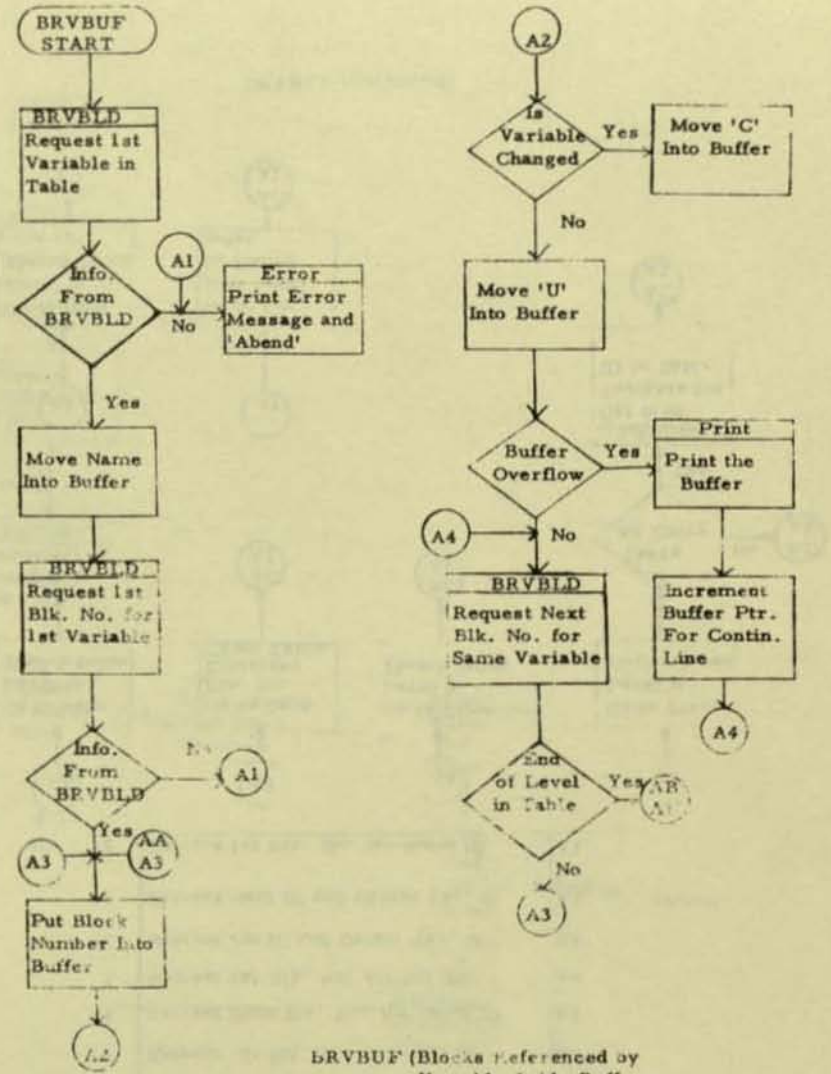
BRVBLD (Block Referenced by Variables Table - Build and Reference Routine)



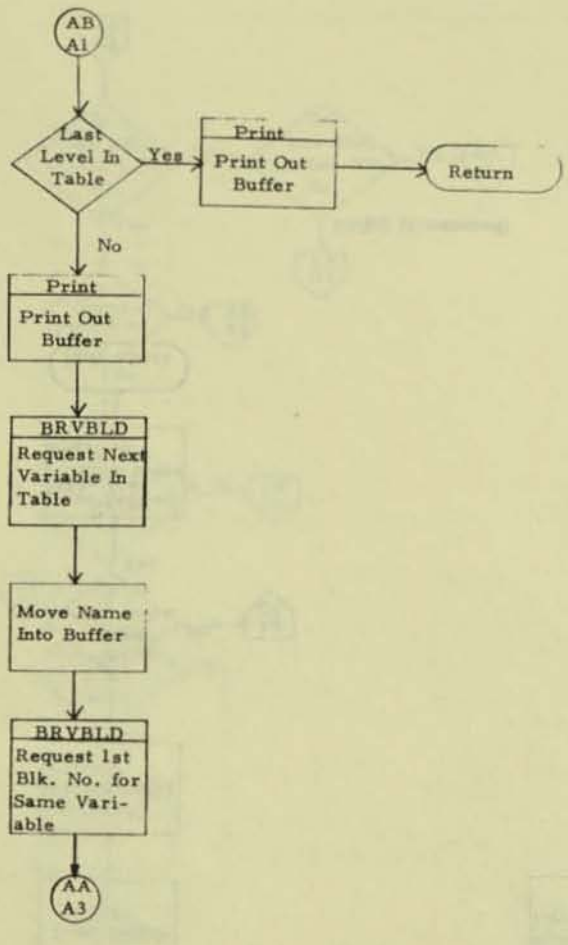
BRVBLD (Continued)



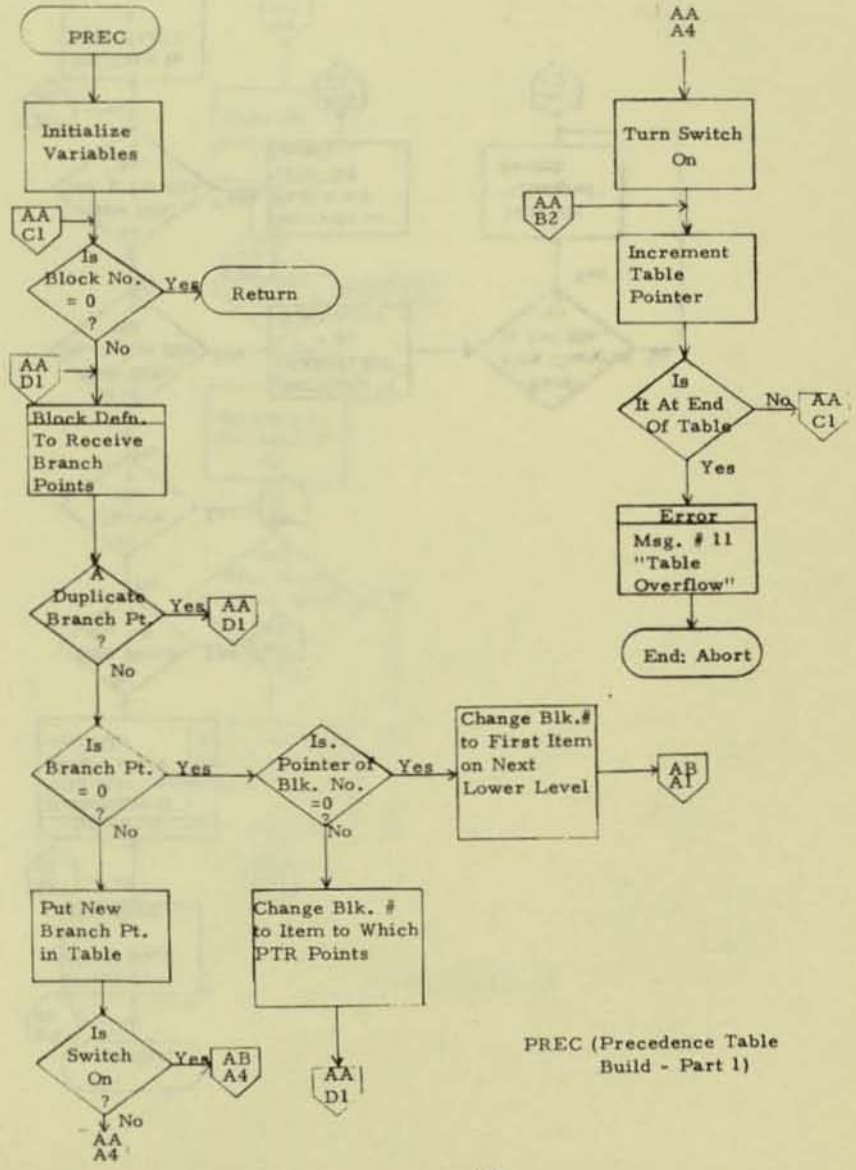
BRVBLD (Continued)



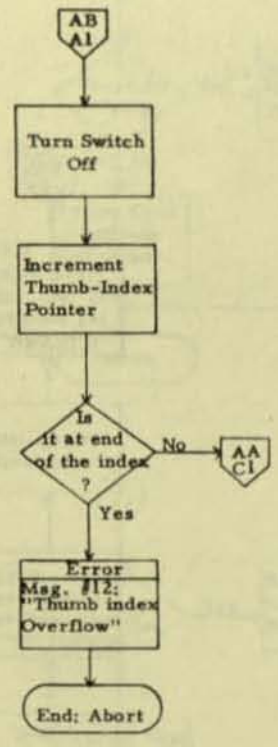
bRVBUF (Blocks Referenced by Variable Table Buffer Routine)

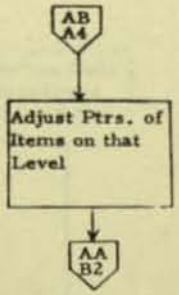


BRVBUF (Continued)

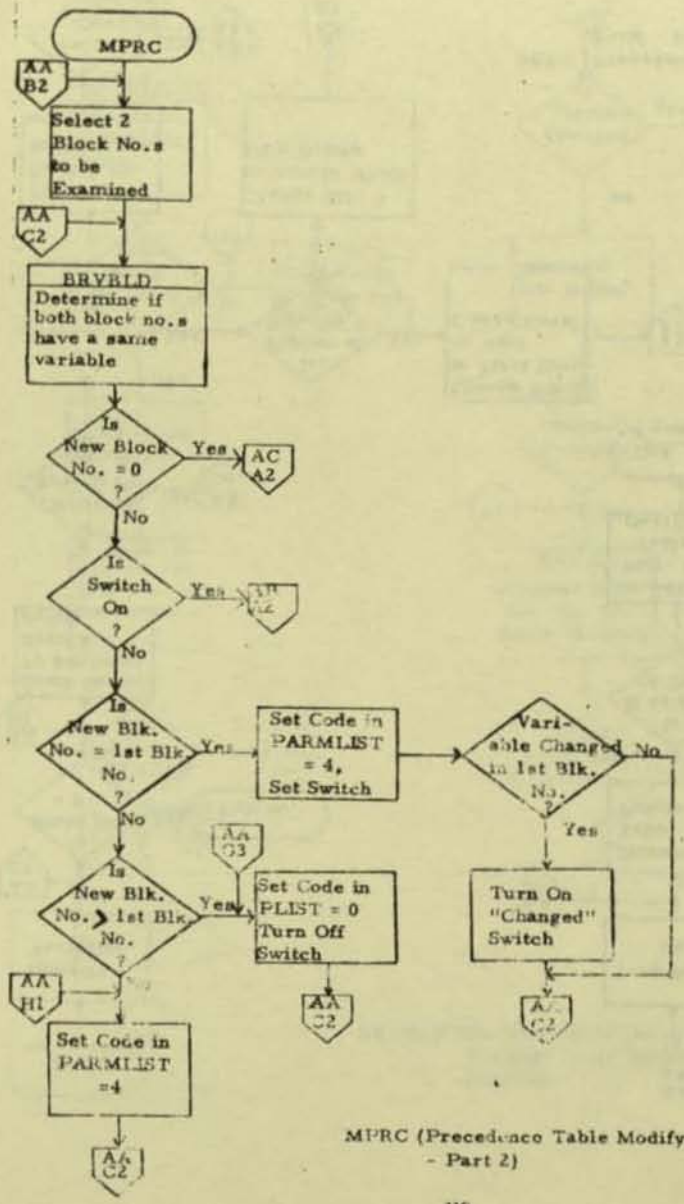


PREC (Precedence Table Build - Part 1)

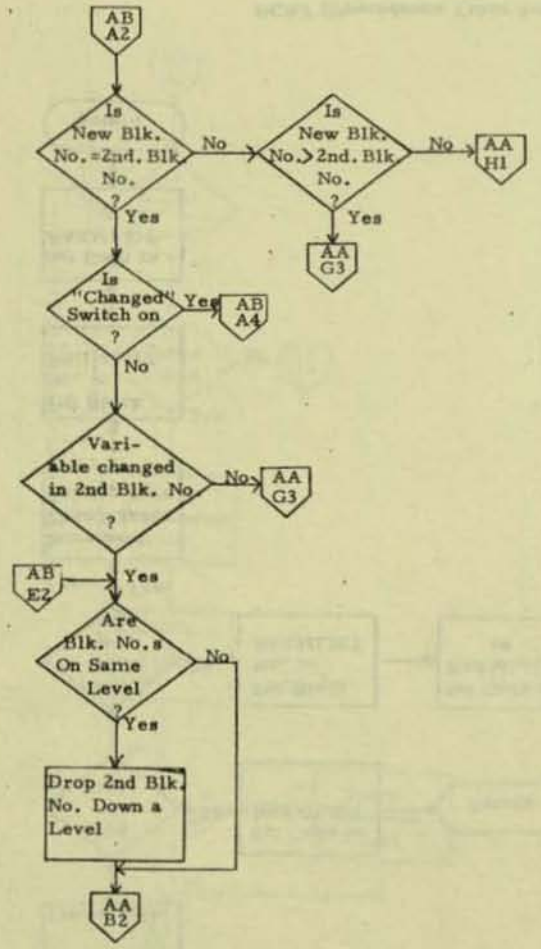


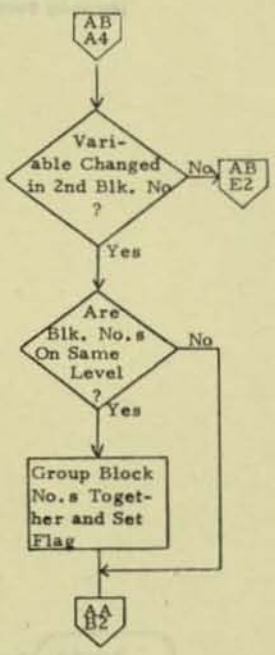


PREC (Continued)

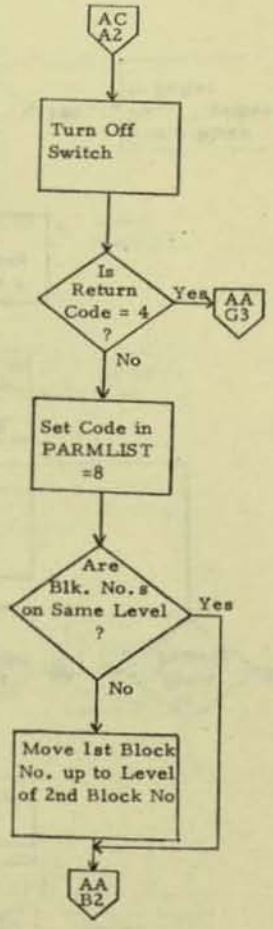


MPRC (Precedence Table Modifying Routine - Part 2)

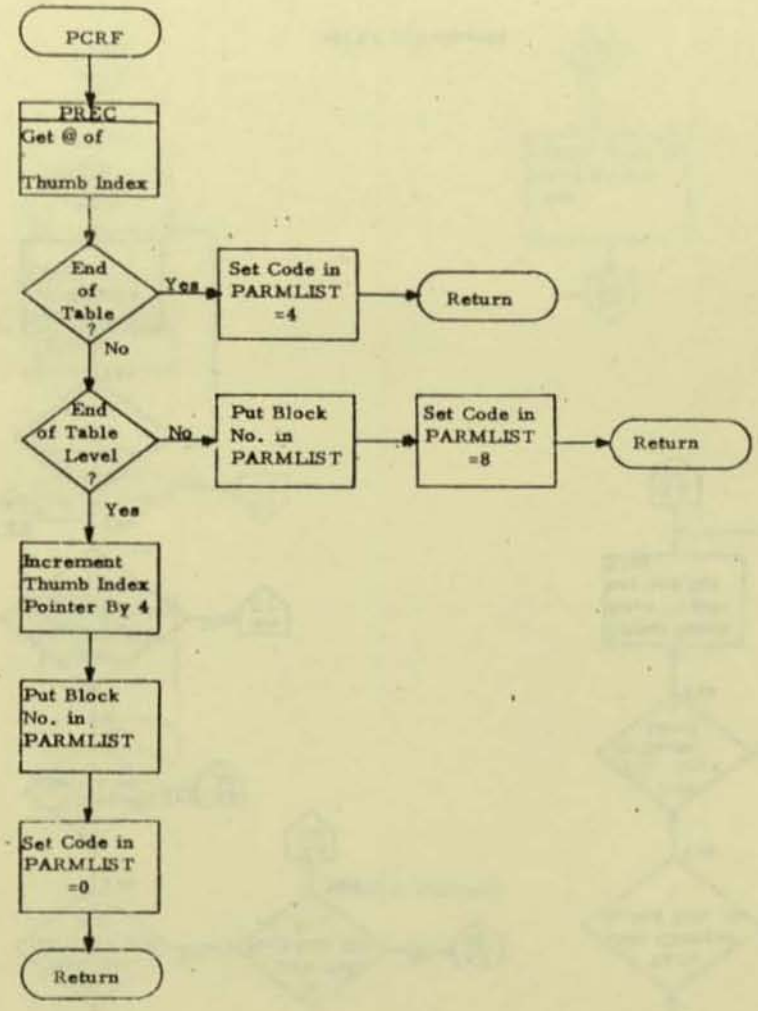




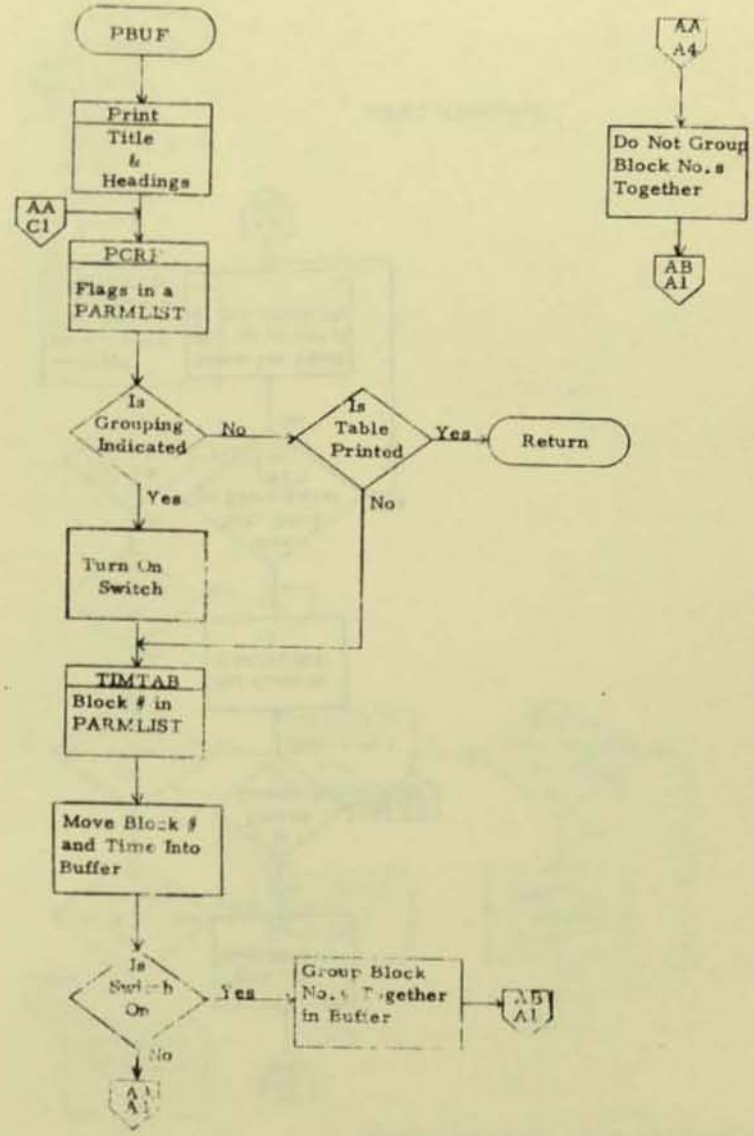
MPRC (Continued)



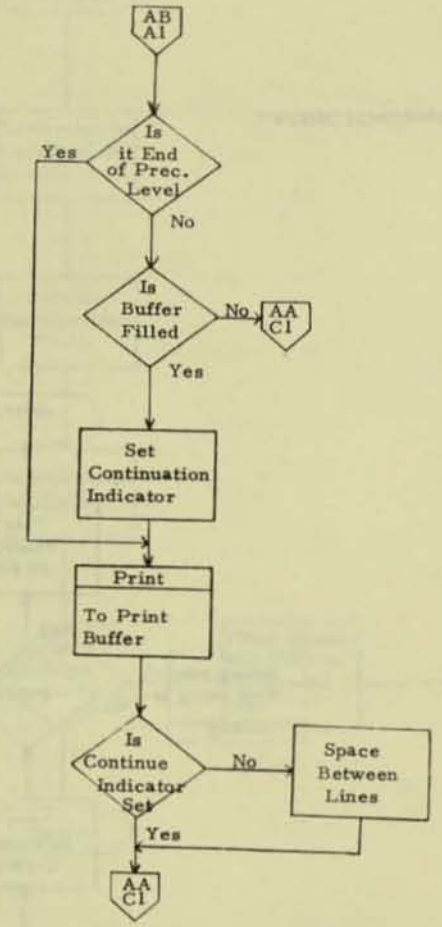
MPRC (Continued)



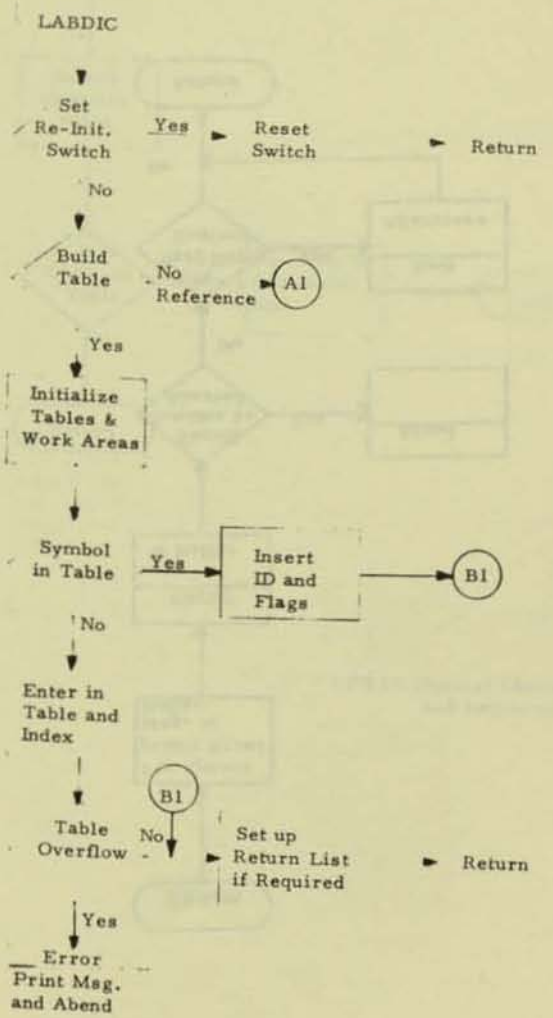
PCRF (Precedence Table Referencing Routine)

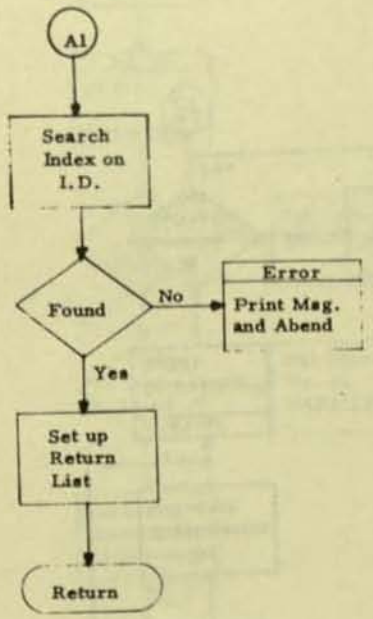


PBUF (Precedence Table Buffering Routine)

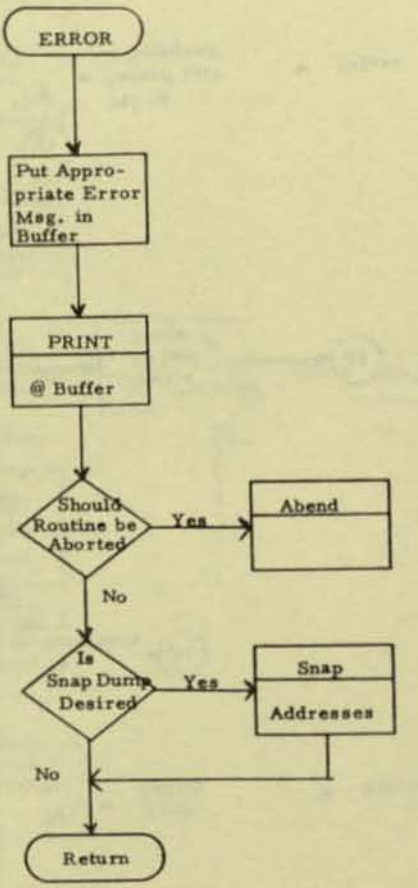


PBUF (Continued)

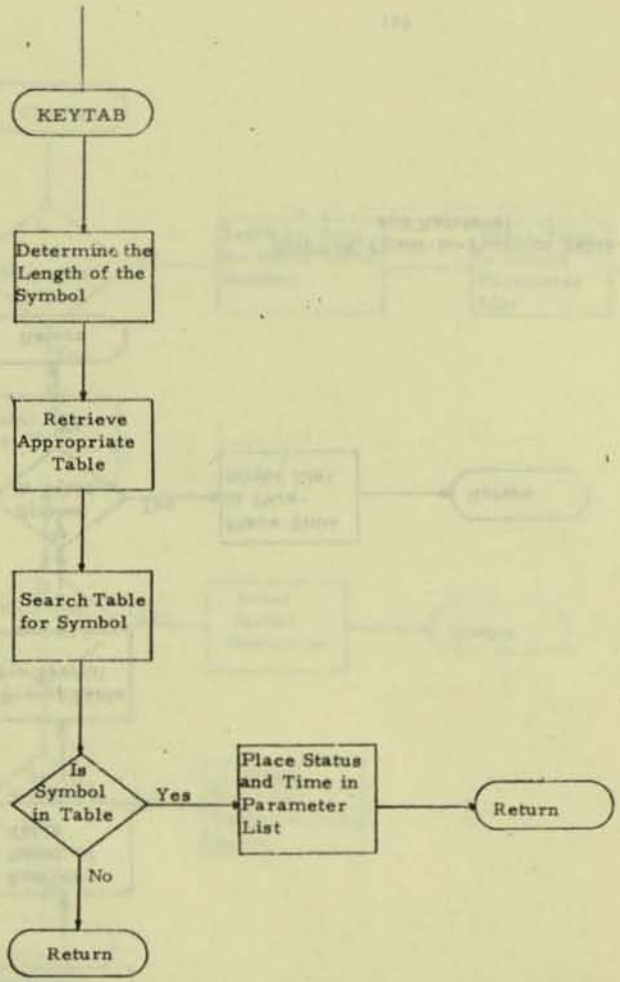




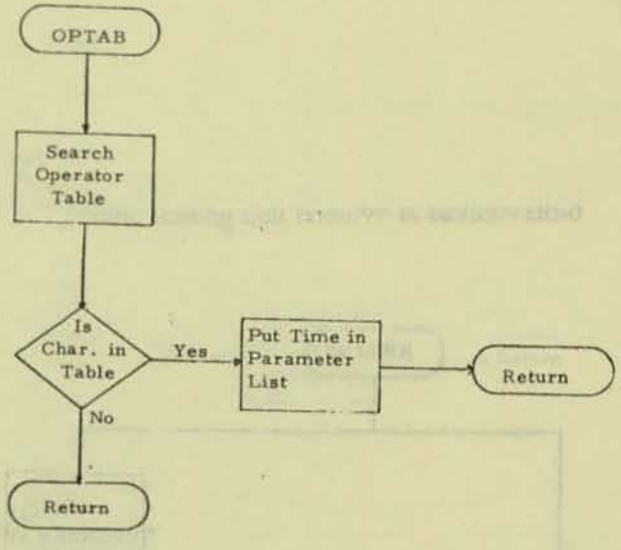
LABDIC (Continued)



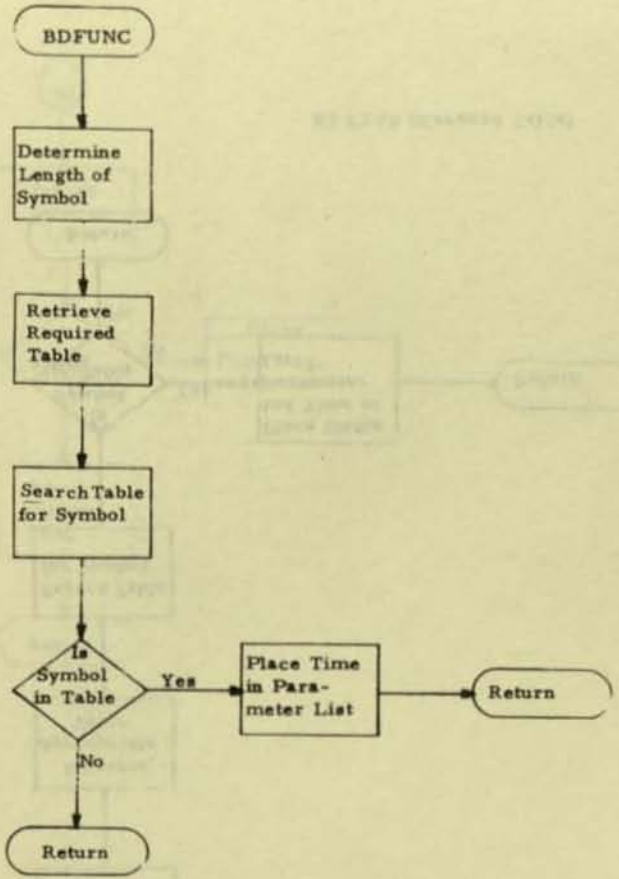
ERROR (Error Message Routine)



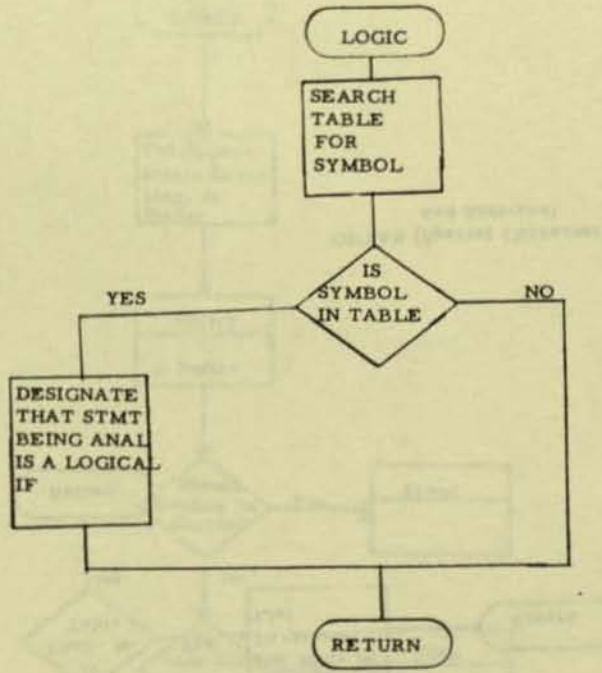
KEYTAB (Keyword Table)



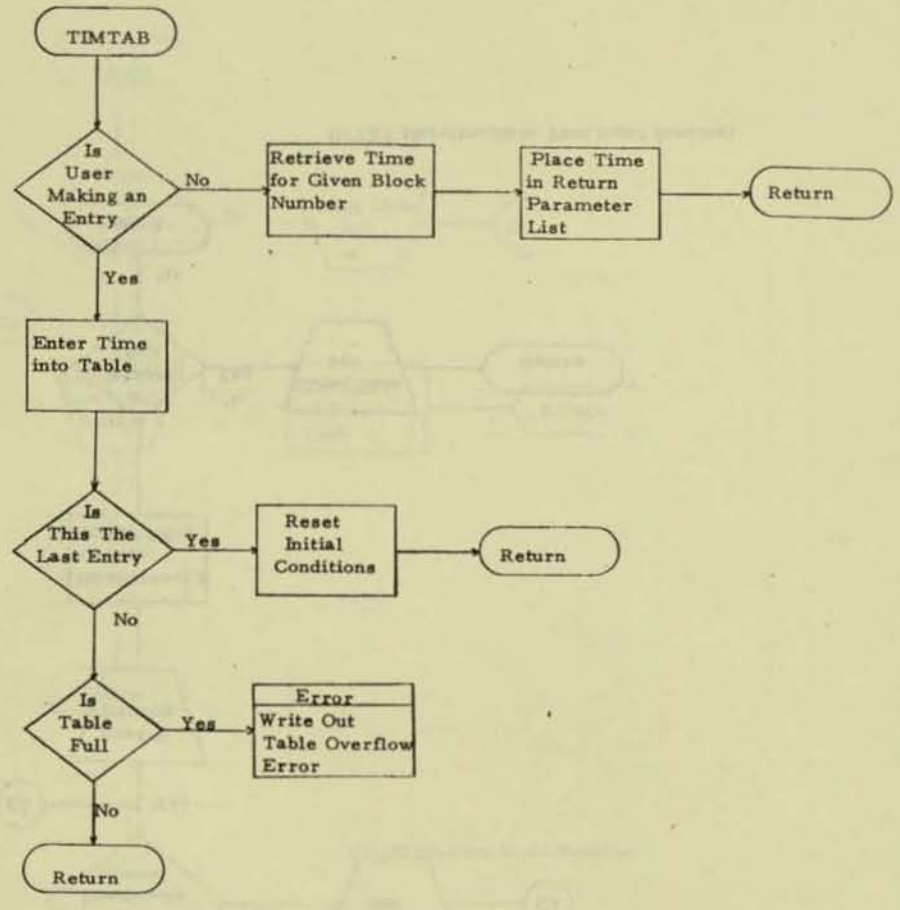
OPTAB (Special Character Table - Search and Retrieve)



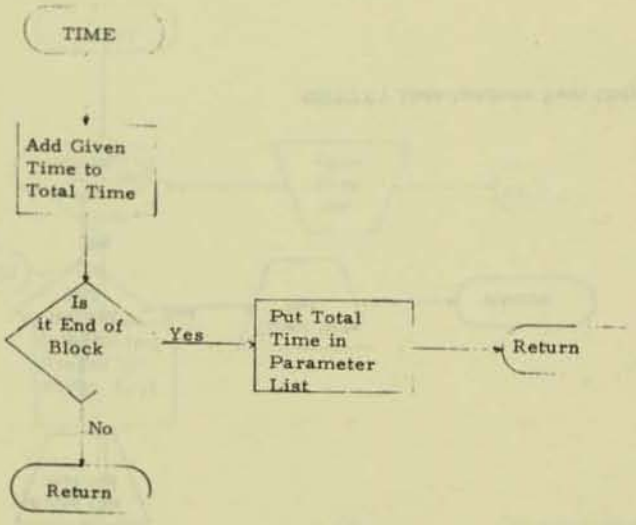
BDFUNC (Built-In-Function Table - Search and Retrieve)



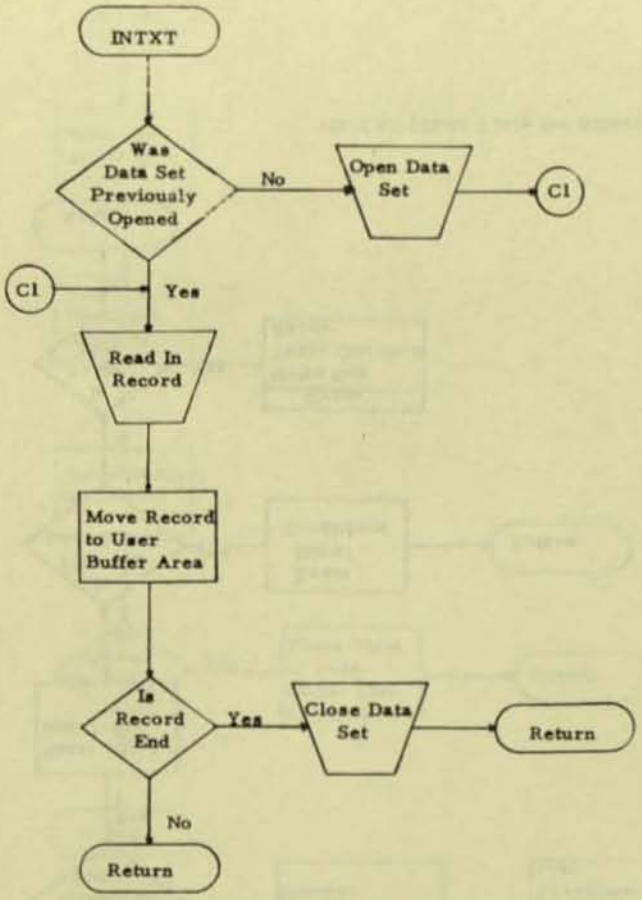
LOGIC (CHECK FOR LOGICAL IF DESIGNATION)



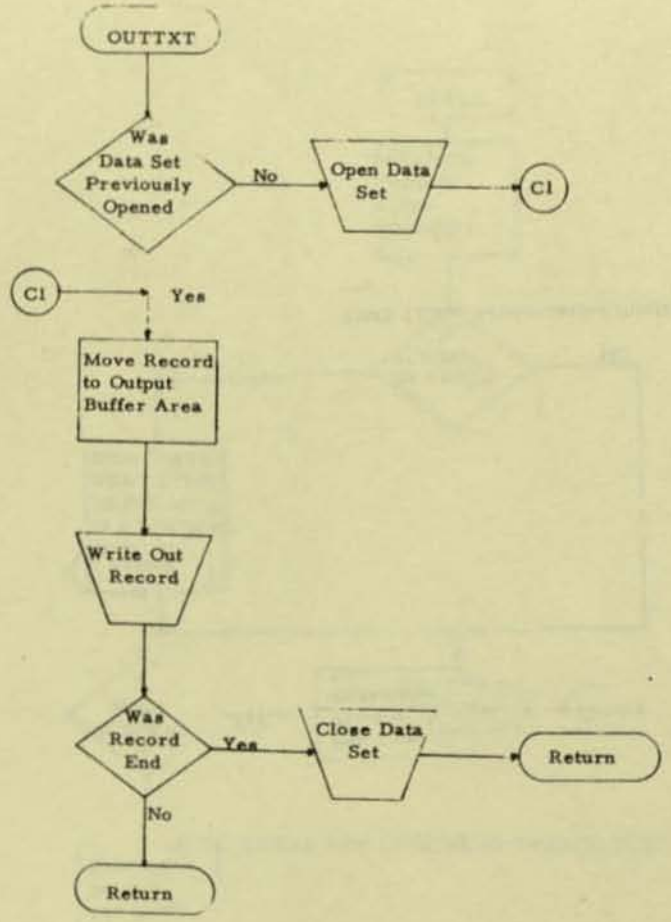
TIMTAB (Time Table for Blocks Routine)



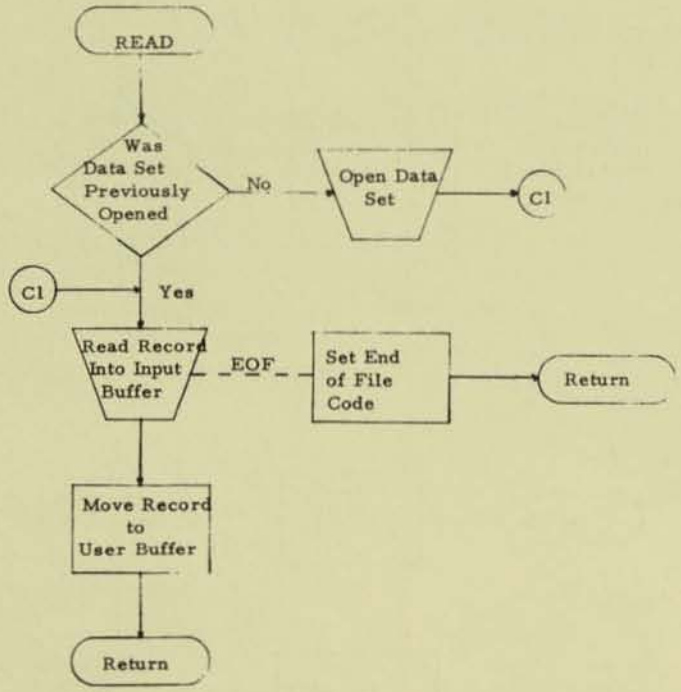
TIME (Time Accumulation Routine)



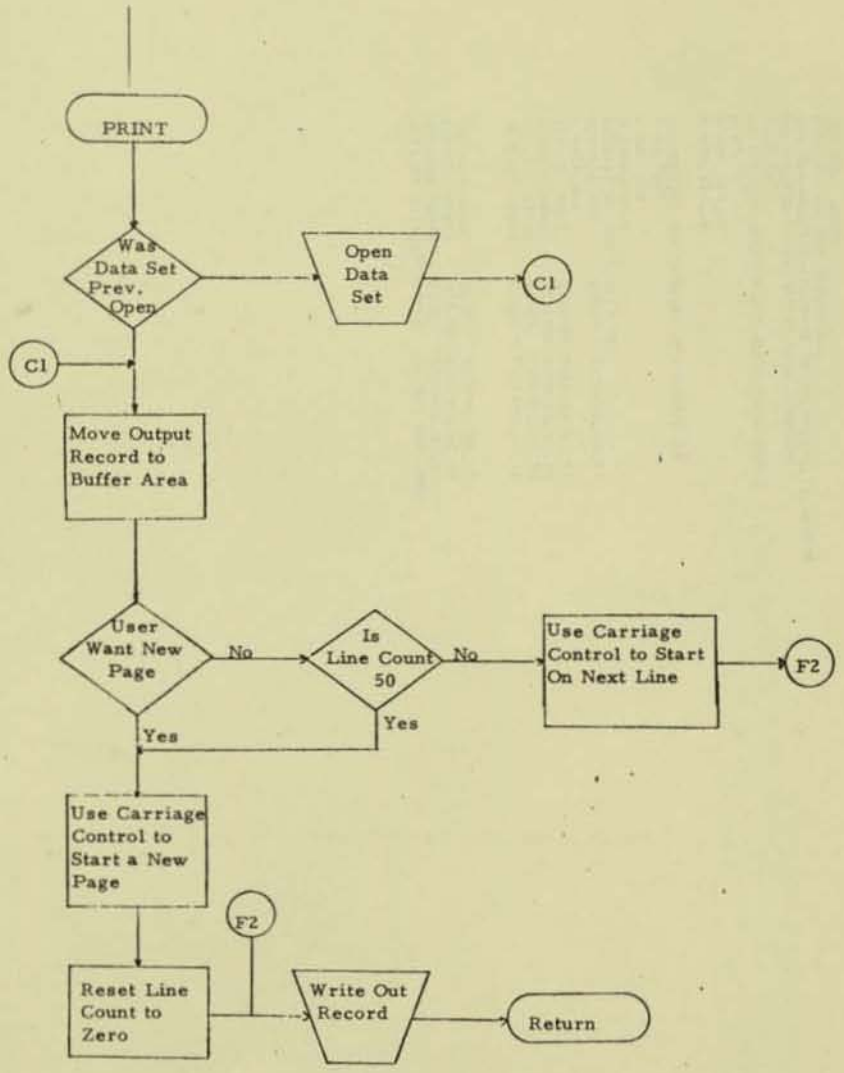
INTXT (Intermediate Text Input Routine)



OUTTXT (Intermediate Test Output Routine)



READ (System Input Routine)



PRINT (System Output Routine)

The output from the three test jobs is on the pages following the typewriter output.

```
IEC101A  READY
IEE537I LOG NOT SUPPORTED.
set date=00.361,clock=09.34.00,q=(191,f),auto=n
00 IEC107D E 191,MVTLIB,SYS1.SYSJOBQE
r 00,'U'
00 IEF423A SPECIFY JOB QUEUE PARAMETERS
r 00,'U'
s wtr,00d,,h
s wtr,00e,,a

00 IEC123D 00E SPECIFY UCS PARAMETER
r 00,'QN'
s rdr,00c
d johnarms,t
IEF161I READER CLOSED 00C
IEF404I RDR      ENDED   TIME=09.40.07
s init.one
IEF403I INIT     STARTED TIME=09.40.24
IEF403I IRV01    STARTED TIME=09.40.27
IEF404I IRV01    ENDED   TIME=09.40.35
IEF403I IRV02    STARTED TIME=09.40.37


IEF404I IRV02    ENDED   TIME=09.41.52
IEF403I IRV03    STARTED TIME=09.41.54
IEF404I IRV03    ENDED   TIME=09.42.59
IEF429I INITIATOR 'ONE' WAITING FOR WORK
```



```
//IRV01  JOB   (EXPG),IMMILLER,MSGLEVEL=1,REGION=90K
//STEP   EXEC  PGM=IEHPROGM
//SYSPRINT DD  SYSOUT=A
//DD DD VOL=SER=111111,DISP=OLD,UNIT=2311
//SYSIN  DD    *
IEF236I ALLOC. FOR IRV01    STEP
IEF237I SYSPRINT ON 191
IEF237I DD       ON 191
IEF237I SYS14    ON 191
```

SYSTEM SUPPORT UTILITIES ---- IEHPROGM PAGE 0001

```
 SCRATCH DSNAME=FORTAP,VOL=2311=111111,PURGE
 IEH207I  STATUS OF USERS REQUEST TO SCRATCH DATA SET FORTAP
          VOLUME I.D.      ACTION TAKEN      REASON FOR TAKING THIS ACTION          ERROR
            111111           NONE            DATA SET OR MEMBER NOT FOUND           ***
END OF ERROR ANALYSIS LISTING  ... UNUSUAL END


UTILITY END
```

141

```
IEF285I   SYS68355.T124943.SV000.I4V01.R0030001      SYSOUT
IEF285I   VOL SER NOS= 111111.
IEF285I   SYS68355.T124943.RV000.IRV01.R0000107      KEPT
IEF285I   VOL SER NOS= 111111.
IEF285I   SYS68355.T124943.RV000.IRV01.S0000003      SYSIN
IEF285I   VOL SER NOS= 111111.
IEF285I   SYS68355.T124943.RV000.IRV01.S0000003      DELETED
IEF285I   VOL SER NOS= 111111.
```

142

```
//IKVO2  JOB   (IXPG1),IMMILLER,MSGLEVEL=1,REGION=90K
//STEP   EXEC  IKFCBL
XXLKED     EXEC  PGM=IEWL,PARM='XREF,LIST,LET,NCAL',REGION=96K          00000010
XXSYSPRINT DD SYSOUT=A                                                  00000020
XXSYSLIN   DD DDNAME=SYSIN                                              00000030
//LKED.SYSLMOD DD DSNAME=FORTAP(FAP),DISP=(NEW,CATLG),                  *
//              LABEL=EXPDT=69350,UNIT=7311,VOL=SER=111111,             *
//              SPACE=(CYL,(1,1,1))
X/SYSLMOD  DD DSNAME=&&GOSET(GO),SPACE=(1024,(50,20,1)),                X00000040
XX            UNIT=SYSDA,DISP=(MOD,PASS)                                00000050
XXSYSUT1   DD UNIT=(SYSDA,SEP=(SYSLMOD,SYSLIN)),                        X00000060
XX            SPACE=(1024,(200,20))                                     00000070
//LKED.SYSIN   DD *
IEF236I ALLOC. FOR IKVO2    LKED    STEP
IEF237I SYSPRINT ON 192
IEF237I SYSLIN   ON 191
IEF237I SYSLMOD  ON 191
IEF237I SYSUT1   ON 194
```



```
IEW0000 LEVEL LINKAGE EDITOR OPTIONS SPECIFIED XREF,LIST,LET,NCAL        DEFAULT OPTION(S) USED
        VARIABLE OPTIONS USED - SIZE=(42160,4192)
```

CROSS REFERENCE TABLE

CONTROL SECTION NAME	ORIGIN	LENGTH	ENTRY NAME	LOCATION	NAME	LOCATION	NAME	LOCATION	NAME	LOCATION
PASS1	00	8C8								
PASS2	8C8	830								
EOF	13F8	104								
BDT	1500	1310								
BDTBUF	2810	2CC								
VARUSE	2AE0	2798								
VARTAB	5278	4E0								
			LAST	5278						
CROSS	5758	1810								
XWRITE	6F68	268								
BRVBLD	7100	388C								
BRVBUF	AD90	590								
PREC	8320	1488								
MPRC	C7A8	5EC								
PCRF	CD98	16C								
PBUF	CF08	60A								
LABDIC	D518	36FC								
ERROR	10C18	D94								
KEYTAB	11980	348								
OPTAB	11CF8	F4								
BDFUNC	11DF0	2AF								
LOGIC	120A0	F0								
TIMTAB	12190	30C								
TIME	124A0	98								
INTXT	12538	10C								
OUTTXT	12718	1C4								
READ	128E0	1C0								
PRINT	12AA0	21C								

LOCATION	REFERS TO SYMBOL	IN CONTROL SECTION	LOCATION	REFERS TO SYMBOL	IN CONTROL SECTION
880	OUTTXT	OUTTXT	884	READ	READ
888	PRINT	PRINT	88C	LABDIC	LABDIC
8C0	KEYTAB	KEYTAB	8C4	PASS2	PASS2
1384	INTXT	INTXT	1388	LABDIC	LABDIC
138C	EOF	EOF	1390	TIME	TIME
1394	TIMTAB	TIMTAB	1398	VARUSE	VARUSE
139C	BDT	BDT	13A0	ERROR	ERROR
13A4	OPTAB	OPTAB	13A8	BDFUNC	BDFUNC
13AC	LOGIC	LOGIC	1404	BDTBUF	BDTBUF
14D8	CROSS	CROSS	140C	BRVBLD	BRVBLD
14E0	PREC	PREC	14E4	BDT	BDT

```
LOCATION  REFERS TO SYMBOL  IN CONTROL SECTION        LOCATION  REFERS TO SYMBOL  IN CONTROL SECTION

   14E8          XWAITF          XWAITF                 14EC          VARTAB          VARTAB
   14F0          BRVBUF          BRVBUF                 14F4          PBUF            PBUF
   14F8          LABDIC          LABDIC                 1894          ERROR           ERROR
   2980          PRINT           PRINT                  2984          BDT             BDT
   3014          LABDIC          LABDIC                 30C4          ERROR           ERROR
   5744          ERROR           ERROR                  5740          LABDIC          LABDIC
   5580          VARUSE          VARUSE                 5688          PRINT           PRINT
   5988          ERROR           ERROR                  59C4          BDT             BDT
   7084          CROSS           CROSS                  708C          ERROR           ERROR
   7088          PRINT           PRINT                  7548          ERROR           ERROR
   75A4          LABDIC          LABDIC                 75AC          VARUSE          VARUSE
   821C          BRVBLD          BRVBLD                 8218          PRINT           PRINT
   8220          ERROR           ERROR                  8528          MPRC            MPRC
   852C          BDT             BDT                    8530          ERROR           ERROR
   C000          BRVBLD          BRVBLD                 C004          ERROR           ERROR
   CE6C          PREC            PREC                   00F8          PRINT           PRINT
   00FC          PCRF            PCRF                   D100          TIMTAB          TIMTAB
   D49C          ERROR           ERROR                 10DA0          PRINT           PRINT
  1225C          ERROR           ERROR                 12680          ERROR           ERROR
  12830          ERROR           ERROR                 12A9C          ERROR           ERROR
  12C10          ERROR           ERROR
ENTRY ADDRESS    00
TOTAL LENGTH   12CC0

****EAP        OOBS NOT EXIST BUT HAS BEEN ADDED TO DATA SET
```



```
IEF285I   SYS68355.T124943.SV000.IRV02.R0000004          SYSOUT
IEF285I   VOL SER NOS= MVTLIB.
IEF285I   SYS68355.T124943.RV000.IRV02.S0000006          SYSIN
IEF285I   VOL SER NOS= 111111.
IEF285I   SYS68355.T124943.RV000.IRV02.S0000006          DELETED
IEF285I   VOL SER NOS= 111111.
IEF285I   FORTAP                                         CATALOGED
IEF285I   VOL SER NOS= 111111.
IEF285I   SYS68355.T124943.RV000.IRV07.*0000005          DELETED
IEF285I   VOL SER NOS= MVTRES.
XAGO EXEC PGM=*.LKED,SYSLMOD,COND=(4,LT,LKED)                              00000040
//GO.SYSUDUMP DD SYSOUT=A
//GO.OUTPUT  DD SYSOUT=A
//GO.OUTTXT  DD  DSNAME=&&TST,UNIT=2311,DISP=(NEW,DELETE,DELETE),      *
//              SPACE=(300,(80,50))
//GO.INPUT DD *
IEF236I ALLOC. FOR IRV02    GO        STEP
IEF237I PGM=*.DD ON 191
IEF237I SYSUDUMP ON 191
IEF237I OUTPUT   ON 193
IEF237I OUTTXT   ON 191
IEF237I INPUT    ON 191
```

```
SEQUENCE   LABEL   SOURCE CODE STATEMENTS

   31             SUBROUTINE CIRCLE(IA,NS,IS,RAD)
0012              DIMENSION IA(64,3,4)
0013              IS=0
0014              AS=2.*RAD
0015              IF(AS-64.)1,1,3
0016            1 NS=AS
0017              CX=32.
0018              CY=32.
0019              CALL ARC(NS,CX,CY,RAD,IA,1)
0020              GO TO 10
0021            3 IS=1
0022              NS=64
0023              DO 4 J=1,4
0024              GO TO(5,6,7,8),J
0025            5 CX=RAD
0026              CY=RAD
0027              GO TO 9
0028            6 CX=64.-RAD
0029              CY=RAD
0030              GO TO 9
0031            7 CX=64.-RAD
0032              CY=64.-RAD
0033              GO TO 9
0034            8 CX=RAD
0035              CY=64.-RAD
0036            9 CALL ARC(NS,CX,CY,RAD,IA,J)
0037            4 CONTINUE
0038           10 RETURN
0039              END
```


LABEL DEFINITION TABLE

ENTRY	LOC	LAST	DEST	BRANCH
1	0001	0005		0006 0011
2	0006	0009	0010	
3	0013	0013		
4	0011	0012	0013	
5	0013	0014		0015 0018 0021 0024
6	0015	0017		0026
7	0018	0020		0026
8	0021	0023		0026
9	0024	0025	0026	
10	0026	0026	0027	
11	0027	0027	0028	
12	0028	0028	0029	
13	0029	0029		

SEQUENCE LABEL SOURCE CODE STATEMENTS


```
IEF285I   FORTAP                                                  KEPT
IEF285I   VOL SER NOS= 111111.
IEF285I   SYS68355.T124943.SV000.1RV02.R0000007                   SYSOUT
IEF285I   VOL SER NOS= 111111.
IEF285I   SYS68355.T124943.SV000.1RV02.R0000008                   SYSOUT
IEF285I   VOL SER NOS= MVTRES.
IEF285I   SYS68355.T124943.RV000.1RV02.TST                        DELETED
IEF285I   VOL SER NOS= 111111.
IEF285I   SYS68355.T124943.RV000.1RV02.S0000009                   SYSIN
IEF285I   VOL SER NOS= 111111.
IEF285I   SYS68355.T124943.RV000.1RV02.S0000009                   DELETED
IEF285I   VOL SER NOS= 111111.
```

```
//IRVOX JOB (EXPG),IMMILLER,MSGLEVEL=1,REGION=40K
//JOBLIB DD   DSNAME=FORTAP,DISP=OLD
//STEP   EXEC PGM=FAP
//GO.OUTTXT  DD  DSNAME=&&TST,UNIT=2311,DISP=(NEW,DELETE,DELETE),
//           SPACE=(400,(80,50))
//GO.OUTPUT  DD  SYSOUT=A
//GO.INPUT DD *
IEF236I ALLOC. FOR IRVOX    STEP
IEF237I JOBLIB    ON 191
IEF237I OUTTXT    ON 192
IEF237I OUTPUT    ON 193
IEF237I INPUT     ON 191
```


FORTRAN ANALYSIS

```
SEQUENCE   LABEL   SOURCE CODE STATEMENTS

0001               SUBROUTINE COSTEQA,QB,C,IS)
0002               DIMENSION PS(3),HS(3)
0003               COMMON PS,HS,XMATA,XMATB,XMAC,XLAA,XMACA,XMACB,XLABA,XLABB
0004               COMMON FA,FAB,F2
0004          C
0004          C    CALCULATES COST OF PRODUCING QA AND QB
0004          C
0004          C        QA=QUANTITY OF A PRODUCED
0004          C        QB=QUANTITY OF B PRODUCED
0004          C        C=COST TO PRODUCE A AND B
0004          C        XMAC=NUMBER OF MACHINES
0004          C        XMATA=AMOUNT OF RAW MATERIAL IN A
0004          C        XMATB=AMOUNT OF RAW MATERIAL IN B
0004          C        XLABA=AMOUNT OF LABOR REQUIRED TO PRODUCE A
0004          C        XLABB=AMOUNT OF LABOR REQUIRED TO PRODUCE B
0004          C        HS=NUMBER OF HOURS IN A SHIFT
0004          C        PS=PAY SCALE IN A SHIFT
0004          C        FA=ADMINISTRATIVE COST TO PROLE A ONLY
0004          C        FAB=ADMINISTRATIVE COST TO PRODUCE B
0004          C        F2=ADMINISTRATIVE COST FOR SECOND SHIFT
0004          C        HSMA=MACHINE HOURS REQUIRED TO PRODUCE QA
0004          C        HSMB=MACHINE HOURS REQUIRED TO PRODUCE QB
0004          C        XM1=MACHINE HOURS AVAILABLE ON FIRST SHIFT
0004          C        XM2=MACHINE HOURS AVAILABLE ON SECOND SHIFT
0004          C        XM3=MACHINE HOURS AVAILABLE ON SECOND SHIFT OVERTIME
0004          C        XLA1=NUMBER OF MEN NEEDED FOR A ON FIRST SHIFT
0004          C        XLB1=NUMBER OF MEN NEEDED FOR B ON FIRST SHIFT
0004          C        XLB2=NUMBER OF MEN NEEDED FOR B ON SECOND SHIFT
0004          C        XLA2=NUMBER OF MEN NEEDED FOR A ON SECOND SHIFT
0004          C        HMA=NUMBER OF MACHINE HOURS TO PRODUCE A
0004          C        HMB=NUMBER OF MACHINE HOURS TO PRODUCE A AND B
0004          C        XLAB1=NUMBER OF MEN ON FIRST SHIFT
0004          C        XLAB2=NUMBER OF MEN ON SECOND SHIFT
0004          C        XLAB3=NUMBER OF MEN ON SECOND SHIFT OVERTIME
0004          C
0004          C
0004          C        COST OF MATERIALS
0004          C
0005               C=QA*XMATA+QB*XMATB
0005          C
0006               IS=1
0007               XLA1=0.
0008               XLA2=0.
0009               XLB1=0.
0010               XLB2=0.
0011               XLAB3=0.
0012               HSMA=QA*XMACA
0013               HSMB=QB*XMACB
0014               RA=XLABA/XMACA
0015               RB=XLABB/XMACB
0016               XM1=XMAC*HS(1)
0017               XM2=XMAC*HS(2)
```

```
0018            XM3=XMAC*HS(3)
0019            HMA=HSMA-XM1
0019      C
0019      C
0019      C         TEST FOR MACHINE AVAILABILITY
0019      C
0019      C         IS THERE ENOUGH MACHINES TO PRODUCE A ON FIRST SHIFT
0019      C
0020            IF(HMA)1,1,2
0021          1 HMB=HSMB+HMA
0022            XLA1=HSMA*RA/HS(1)
0022      C
0022      C         ARE THERE ENOUGH MACHINES TO PRODUCE B ON FIRST SHIFT
0022      C
0023            IF(HMB)3,3,4
0024          2 HMA=HMA-XM2
0025            XLA1=XMAC*RA
0025      C
0025      C         ARE THERE ENOUGH MACH TO PRODUCE A ON FIRST AND SECOND SHIFTS
0025      C
0026            IF(HMA)5,5,6
0027          4 HMA=HMA-XM2
0028            XLB1=(XMAC-HSMA/HS(1))*RB
0029            XM1=HSMA
0030          5 HMB=HSMB+HMA
0031            IF(XLB1)16,16,17
0032         16 XLA2=(HSMA-XM1)*RA/HS(2)
0032      C
0032      C         ARE THERE ENOUGH MACHINES TO PRODUCE B ON SECOND SHIFT
0032      C
0033         17 IF(HMB)7,7,8
0034          6 HMA=HMA-XM3
0035            XLA2=XMAC*RA
0035      C
0035      C         ARE THERE ENOUGH MACHINES TO PRODUCE A ON ALL SHIFTS
0035      C
0036            IF(HMA)9,9,10
0037          8 HMA=HMA-XM3
0038            XLB2=(XMAC-(HSMA-XM1)/HS(2))*RB
0039          9 HMB=HSMB+HMA
0039      C
0039      C         ARE THERE ENOUGH MACHINES TO PRODUCE B ON THIRD SHIFT
0039      C
0040            IF(HMB)11,11,10
0040      C
0040      C
0040      C         TEST FOR LABOR AVAILABILITY
0040      C
0040      C
0040      C         LABOR FOR FIRST SHIFT
0040      C
0041          3 XLABN=(HSMA*RA+HSMB*RB)/HS(1)
0041      C     IGNORE LABOR FORCE
0042            XLAB=XLABN
0043         12 C=C+XLAB*HS(1)*PS(1)
0044         21 IF(QB)10,14,15

                                    157

0045         14 C=C+FA
0046            GO TO 26
0047         15 C=C+FAB
0048            GO TO 26
0048      C
0048      C         LABOR FOR FIRST AND SECOND SHIFTS AND POSSIBLY THIRD
0048      C
0049          7 C=C+F2
0050            XLABN=XLA1+XLB1
0051         22 XLAB2=(HSMA*RA+HSMB*RB-(XLABN*HS(1)))/HS(2)
0052            XLAB1=XLABN
0053            XLAB3=0.
0054            GO TO 20
0054      C
0054      C         LABOR FOR ALL THREE SHIFTS
0054      C
0055         11 C=C+F2
0056            XLABN=XLA1+XLB1
0057            XLAB2=XLA2+XLB2
0058            XLAB1=XLABN
0059            XLAB3=(HSMA*RA+HSMB*RB-(XLABN*HS(1)+XLAB2*HS(2)))/HS(3)
0060         20 C=C+XLAB1*HS(1)*PS(1)+XLAB2*HS(2)*PS(2)+XLAB3*HS(3)*PS(3)
0061            GO TO 21
0061      C
0061      C
0062         10 IS=0
0063            C=0.
0064         26 RETURN
0065            END
```

BLOCK DEFINITION TABLE

BLOCK #	1ST	LAST	DEST	BRANCH		
1	0001	0020		0021	0024	
2	0021	0023		0027	0041	
3	0024	0026		0030	0034	
4	0027	0029	0030			
5	0030	0031		0032	0033	
6	0032	0032	0033			
7	0033	0033		0037	0049	
8	0034	0036		0039	0062	
9	0037	0038	0039			
10	0039	0040		0055	0062	
11	0041	0043	0044			
12	0044	0044		0045	0047	0062
13	0045	0046		0064		
14	0047	0048		0064		
15	0049	0054		0060		
16	0055	0059	0060			
17	0060	0061		0044		
18	0062	0063	0064			
19	0064	0064	0065			
20	0065	0065				

BRANCH CROSS-REFERENCE TABLE

BLOCK NO.	TRANSFER BLOCKS		
1			
2	0001		
3	0001		
4	0002		
5	0003	0004	
6	0005		
7	0005	0006	
8	0003		
9	0007		
10	0008	0009	
11	0002		
12	0011	0017	
13	0012		
14	0012		
15	0007		
16	0010		
17	0015	0016	
18	0008	0010	0012
19	0013	0014	0018
20	0019		

VARIABLES USED BY BLOCK TABLE

BLOCK NUMBER	VARIABLE STATUS									
1	IN	HMA	HS	HSMA	QA	QB	XLADA	XLABN	XMAC	XMAC3
		XMACB	XMATA	XMATB	XM1					
	OUT	C	HMA	HSMA	HSMB	IS	RA	RB	XLAB3	XLA1
		XLA2	XLB1	XLB2	XM1	XM2	XM3			
2	IN	HMA	HMB	HS	HSMA	HSMB	RA			
	OUT	HMB	XLA1							
3	IN	HMA	RA	XMAC	XM2					
	OUT	HMA	XLA1							
4	IN	HMA	HS	HSMA	RB	XMAC	XM2			
	OUT	HMA	XLB1	XM1						
5	IN	HMA	HSMB	XLB1						
	OUT	HMB								
6	IN	HS	HSMA	RA	XM1					
	OUT	XLA2								
7	IN	HMB								
	OUT	NO VARIABLES USED								
8	IN	HMA	RA	XMAC	XM3					
	OUT	HMA	XLA2							
9	IN	HMA	HS	HSMA	RB	XMAC	XM1	XM3		
	OUT	HMA	XLB2							
10	IN	HMA	HMB	HSMB						
	OUT	HMB								
11	IN	C	HS	HSMA	HSMB	PS	RA	RB	XLAB	XLABN
	OUT	C	XLAB	XLABN						
12	IN	QB								
	OUT	NO VARIABLES USED								
13	IN	C	PA							
	OUT	C								
14	IN	C	PAB							
	OUT	C								
15	IN	C	P2	HS	HSMA	HSMB	RA	RB	XLABN	XLA1
		XLB1								
	OUT	C	XLABN	XLAB1	XLAB2	XLAB3				
16	IN	C	P2	HS	HSMA	HSMB	RA	RB	XLABN	XLAB2
		XLA1	XLA2	XLB1	XLB2					
	OUT	C	XLABN	XLAB1	XLAB2	XLAB3				

17	IN	C	HS	PS	XLAB1	XLAB2	XLAB3
	OUT	C					
18	IN	NO VARIABLES USED					
	OUT	C	IS				

IN - VARIABLE REFERENCED IN BLOCK.
OUT - VARIABLE CHANGED OR DEFINED IN BLOCK.

BLOCK REFERENCED BY VARIABLES TABLE

VARIABLE		BLOCK NUMBER
C	U	NONE
	C	0001 0011 0013 0014 0015 0016 0017 0018
FA	U	0013
	C	NONE
FAB	U	0014
	C	NONE
F2	U	0015 0016
	C	NONE
HMA	U	0002 0005 0010
	C	0001 0003 0004 0008 0009
HMR	U	0007
	C	0002 0005 0010
HS	U	0001 0002 0004 0006 0009 0011 0015 0016 0017
	C	NONE
HSMA	U	0002 0004 0006 0009 0011 0015 0016
	C	0001
HSMB	U	0002 0005 0010 0011 0015 0016
	C	0001
IS	U	NONE
	C	0001 0018
P5	U	0011 0017
	C	NONE
QA	U	0001
	C	NONE
QB	U	0001 0012
	C	NONE
RA	U	0002 0003 0006 0009 0011 0015 0016
	C	0001
RB	U	0004 0009 0011 0015 0016
	C	0001
XLA1	U	NONE
	C	0011
XLA1A	U	0001
	C	NONE

XLA1B	U	0001
	C	NONE
XLA2	U	NONE
	C	0011 0015 0016
XLA3I	U	0017
	C	0015 0016
XLA3J	U	0017
	C	0015 0016
XLA3K	U	0017
	C	0001 0015 0016
XLA1	U	0015 0016
	C	0001 0002 0003
XLA2	U	0016
	C	0001 0006 0009
XL3I	U	0005 0015 0016
	C	0001 0004
XLM2	U	0016
	C	0001 0009
XMAC	U	0001 0003 0004 0008 0009
	C	NONE
XMACA	U	0001
	C	NONE
XMACB	U	0001
	C	NONE
XMATA	U	0001
	C	NONE
XMATB	U	0001
	C	NONE
XM1	U	0006 0009
	C	0001 0004
XM2	U	0003 0004
	C	0001
XM3	U	0008 0009
	C	0001

* C MEANS THAT THE VALUE OF THE VARIABLE IS CHANGED IN THE FOLLOWING BLOCKS.
* U MEANS THAT THE VALUE OF THE VARIABLE IS NOT CHANGED IN THE FOLLOWING BLOCKS.

BLK#-TIME PRECEDENCE DIAGRAM.

0001-000582 0012-000040 0019-000030 0020-000000

0002-000156 0011-000262

0004-000142 0003-000116 0008-000116 0007-000040 0018-000020

0006-000106 0009-000148

0013-000036 0016-000346 0015-000284 0014-000036

0017-000308 0010-000066 0005-000066

NOTES:
TIME IS IN MICROSECONDS.
IF TIME IS 999999, THEN IT MAY BE LESS THAN ACTUAL TIME FOR THAT BLOCK OF CODE.
EXECUTION TIME DOES NOT INCLUDE LOOPING WITHIN THAT BLOCK OF CODE.
BLOCK NUMBERS GROUPED TOGETHER HAVE CHANGED THE SAME VARIABLES. THEY SHOULD BE EXAMINED [illegible]

FORTRAN ANALYSIS

```
SEQUENCE   LABEL   SOURCE CODE STATEMENTS

0001               SUBROUTINE MESAGE(MN,NC)
0002               DIMENSION AMES(18,100),IMP(100,2)
0003               DATA ENDM/'ENDM'/,ENDA/'ENDA'/
0003         C
0003         C     DISPLAYS MESSAGE ON 2250
0003         C
0003         C         MN=MESSAGE NUMBER
0004         C         NC=NUMBER OF CHARACTERS (IF NOT ZERO)
0005         C         IPT=NUMBER OF MESSAGES
0006         C         LIN=MESSAGE NUMBER IN AMES
0007         C         MD=IMP(*,1)=POINTER TO MESSAGE IN AMES
0008         C         ML=IMP(*,2)=NUMBER OF LINES IN MESSAGE
0009         C         MC=NUMBER OF CHARACTERS IN MESSAGE
0009         C
0009         C
0004             1 FORMAT(1H44)
0005               IF(MN)2,2,8
0006             2 IO=5
0007               LIN=0
0008               IPT=0
0009             3 IMP(IPT+1,1)=LIN+1
0010               NL=-1
0011             4 LIN=LIN+1
0012               NL=NL+1
0013               CALL IOM(AMES(1,LIN))
0014               IF(AMES(1,LIN)-ENDM)6,5,6
0015             5 LIN=LIN-1
0016               IPT=IPT+1
0017               IMP(IPT,2)=NL
0018               GO TO 3
0019             6 IF(AMES(1,LIN)-ENDA)4,7,4
0020             7 LIN=LIN-1
0021               GO TO 11
0022             8 MD=IMP(MN,1)
0023               ML=IMP(MN,2)
0024               IF(NC)10,10,9
0025             9 CALL LABEL(AMES(1,MD),NC)
0026               GO TO 11
0027            10 MC=ML*72
0028               CALL LABEL(AMES(1,MD),MC)
0029            11 RETURN
0030               END
```

BLOCK #	1ST	LAST	DEST	BRANCH	
1	0001	0005		0006	0022
2	0006	0008	0009		
3	0009	0010	0011		
4	0011	0013	0014		
5	0014	0014		0015	0019
6	0015	0018		0009	
7	0019	0019		0011	0020
8	0020	0021		0029	
9	0022	0024		0025	0027
10	0025	0025	0026		
11	0026	0026		0029	
12	0027	0028	0029		
13	0029	0029	0030		
14	0030	0030			

BRANCH CROSS-REFERENCE TABLE

BLOCK NO.	TRANSFER BLOCKS		
1			
2	0001		
3	0002	0006	
4	0003	0007	
5	0004		
6	0005		
7	0005		
8	0007		
9	0001		
10	0009		
11	0010		
12	0009		
13	0008	0011	0012
14	0013		

VARIABLES USED BY BLOCK TABLE

BLOCK NUMBER	VARIABLE STATUS			
1	IN	MN		
	OUT	NO VARIABLES USED		
2	IN	NO VARIABLES USED		
	OUT	IO	IPT	LIN
3	IN	IPT	LIN	
	OUT	IMP	NL	
4	IN	LIN	NL	
	OUT	AMES	LIN	NL
5	IN	AMES	ENDM	LIN
	OUT	NO VARIABLES USED		
6	IN	IPT	LIN	NL
	OUT	IMP	IPT	LIN
7	IN	AMES	ENDA	LIN
	OUT	NO VARIABLES USED		
8	IN	LIN		
	OUT	LIN		
9	IN	IMP	MN	NC
	OUT	MD	ML	
10	IN	NO VARIABLES USED		
	OUT	AMES	MD	NC
12	IN	ML		
	OUT	AMES	MC	MD

IN - VARIABLE REFERENCED IN BLOCK.
OUT - VARIABLE CHANGED OR DEFINED IN BLOCK.

BLOCK REFERENCED BY VARIABLES TABLE

VARIABLE		BLOCK NUMBER			
AMES	U	0005	0007		
	C	0004	0010	0012	
ENDA	U	0007			
	C	NONE			
ENDM	U	0005			
	C	NONE			
IMP	U	0009			
	C	0003	0006		
IO	U	NONE			
	C	0002			
IPT	U	0003			
	C	0002	000A		
LIN	U	0003	0005	0007	
	C	0002	0004	000A	0008
MC	U	NONE			
	C	0012			
MD	U	NONE			
	C	0009	0010	0012	
ML	U	0012			
	C	0009			
MN	U	0001	0009		
	C	NONE			
NC	U	0009			
	C	0010			
NL	U	0006			
	C	0003	0004		

* C MEANS THAT THE VALUE OF THE VARIABLE IS CHANGED IN THE FOLLOWING BLOCKS.
* U MEANS THAT THE VALUE OF THE VARIABLE IS NOT CHANGED IN THE FOLLOWING BLOCKS.

PRECEDENCE DIAGRAM.

BLKB-TIME

0001-000040 0002-000030 0009-000060 0011-000010 0013-000030 0014-000000

0003-000068 0010-001000 0012-001050

0004-001052

0005-000040

0006-000072

0008-000036

0007-000040

NOTES:
TIME IS IN MICROSECONDS.
IF TIME IS 999999, THEN IT MAY BE LESS THAN ACTUAL TIME FOR THAT BLOCK OF CODE.
EXECUTION TIME DOES NOT INCLUDE LOOPING WITHIN THAT BLOCK OF CODE.
BLOCK NUMBERS GROUPED TOGETHER HAVE CHANGED THE SAME VARIABLES. THEY SHOULD BE EXAMINED MORE THOROUGHLY.

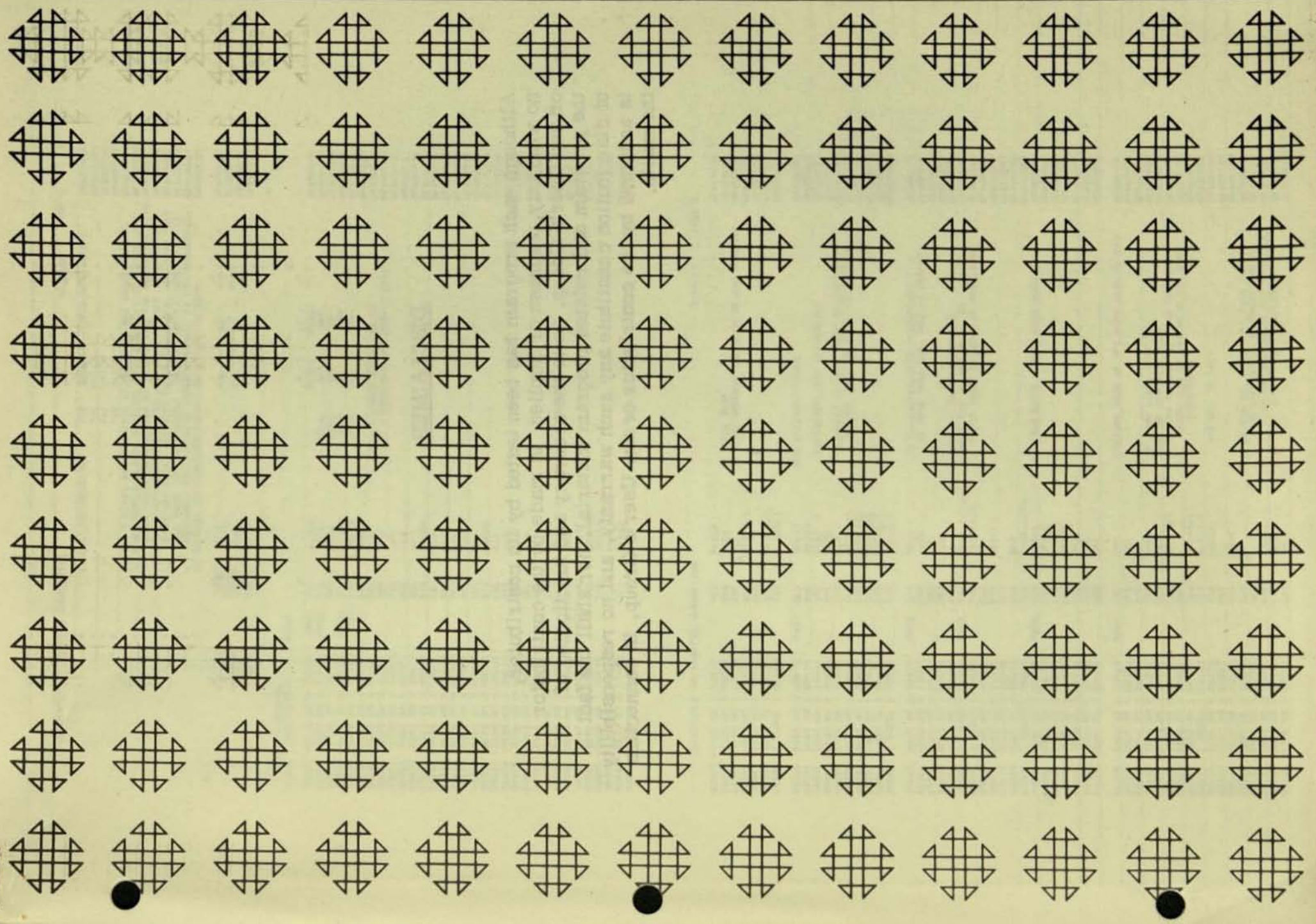
FORTRAN ANALYSIS

SEQUENCE LABEL SOURCE CODE STATEMENTS

```
IEF285I   FORTAP                                                  PASSED
IEF285I   VOL SER NOS= 111111.
IEF285I   SYS08355.T124943.RV000.IRV03.TST                        DELETED
IEF285I   VOL SER NOS= MVTLIB.
IEF285I   SYS08355.T124943.SV000.IRV03.R0000010                   SYSOUT
IEF285I   VOL SER NOS= MVTRES.
IEF285I   SYS08355.T124943.RV000.IRV03.S0000011                   SYSIN
IEF285I   VOL SER NOS= 111111.
IEF285I   SYS08355.T124943.RV000.IRV03.S0000011                   DELETED
IEF285I   VOL SER NOS= 111111.
IEF285I   FORTAP                                                  KEPT
IEF285I   VOL SER NOS= 111111.
```


Acknowledgments

I would like to thank the programming experience group which included R. Laciano, E. Miles, D. Nadel, C. Owens, L. Stultz, and A. Marotta for developing, coding, debugging, and documenting the FORTRAN Analysis Program.



DISCLAIMER

Although each program has been tested by its contributor, no warranty, express or implied, is made by the contributor or any User's Group, as to the accuracy and functioning of the program and related program material, nor shall the fact of distribution constitute any such warranty, and no responsibility is assumed by the contributor or any User's Group, in connection therewith.

```
                    *       1. CALLING SEQUENCE     TSX   $QFAD,4  (QFSB,QFMP,QFDP)     QFAD0004
                    *                               PZE   OP1,T                        QFAD0005
                    *                               PZE   OP2,T                        QFAD0006
                    *                               PZE   ANS,T                        QFAD0007
                    *       2. THE 3 CODES ARE POINTERS TO OPERANDS AND ANSWER CELLS.  QFAD0008
                    *       3. A QUADRUPLE NUMBER CONSISTS OF 4 CONSECUTIVE WORDS, A 3 QFAD0009
                    *          WORD MANTISSA FOLLOWED BY A WORD FOR THE CHARACTERISTIC.QFAD0010
                    *          ONLY NORMALIZED NUMBERS ARE LEGAL.   THE 3 WORDS OF    QFAD0011
                    *          MANTISSA CARRY 105 BITS OF SIGNIFICANCE, AND THEY MUST QFAD0012
                    *          ALL HAVE THE SAME SIGN.   CHARACTERISTIC WORD MUST BE  QFAD0013
                    *          UNSIGNED AND THE BASE CHARACTERISTIC IS 0200000000000. QFAD0014
                    *       4. ILLEGAL COMPUTATION CAUSES AN AUTOMATIC DUMP.          QFAD0015

                00005       ENTRY   QFAD                                              QFAD0017
                00004       ENTRY   QFSB                                              QFAD0018
                00002       ENTRY   QFMP                                              QFAD0019
                00001       ENTRY   QFDP                                              QFAD0020

  TRANSFER VECTOR
00000  246444476060    DUMP

00001 -0625 00 0 00464  QFDP  STL   FLAG2                  FLAG1   FLAG2              QFAD0022
00002 -0625 00 0 00463  QFMP  STL   FLAG1          QFAD    OFF     OFF                QFAD0023
00003  0020 00 0 00005        TRA   QFAD           QFSB    OFF     ON                 QFAD0024
00004 -0625 00 0 00464  QFSB  STL   FLAG2          QFMP    ON      OFF                QFAD0025
00005  0634 00 1 00456  QFAD  SXA   XR1,1          QFDP    ON      ON                 QFAD0026
00006  0634 00 4 00457        SXA   XR4,4                                              QFAD0027
00007 -0500 00 4 00001        CAL   1,4       COMPUTE CHECK ADDRESSES OF               QFAD0028
00010  0625 00 0 00011        STT   *+1         THE 2 OPERANDS AND ANSWER CELLS        QFAD0029
00011  0756 00 0 00000        PCA   ,**                                                QFAD0030
00012  0361 00 4 00001        ACL   1,4                                                QFAD0031
00013  0361 00 0 00477        ACL   =4                                                 QFAD0032
00014  0621 00 0 00032        STA   CAOP1                                              QFAD0033
00015 -0500 00 4 00002        CAL   2,4                                                QFAD0034
00016  0625 00 0 00017        STT   *+1                                                QFAD0035
00017  0756 00 0 00000        PCA   ,**                                                QFAD0036
00020  0361 00 4 00002        ACL   2,4                                                QFAD0037
00021  0361 00 0 00477        ACL   =4                                                 QFAD0038
00022  0621 00 0 00033        STA   CAOP2                                              QFAD0039
00023 -0500 00 4 00003        CAL   3,4                                                QFAD0040
00024  0625 00 0 00025        STT   *+1                                                QFAD0041
00025  0756 00 0 00000        PCA   ,**                                                QFAD0042
00026  0361 00 4 00003        ACL   3,4                                                QFAD0043
00027  0361 00 0 00477        ACL   =4                                                 QFAD0044
00030  0621 00 0 00454        STA   CAANS                                              QFAD0045
00031  0774 00 1 00004        AXT   4,1       PULL IN BOTH OPERANDS                    QFAD0046
00032  0500 00 1 00000  CAOP1 CLA   **,1                                               QFAD0047
00033  0560 00 1 00000  CAOP2 LDQ   **,1                                               QFAD0048
00034  0601 00 1 00471        STO   OP1+4,1                                            QFAD0049
00035 -0600 00 1 00475        STQ   OP2+4,1                                            QFAD0050
00036  2 00001 1 00032        TIX   *-4,1,1                                            QFAD0051
00037  0520 00 0 00463        ZET   FLAG1                                              QFAD0052
00040  0020 00 0 00240        TRA   FMDP      IF FMP OR FDP, SKIP                      QFAD0053
```



```
00041 -0520 00 0 00464        NZT   FLAG2                                              QFAD0055
00042  0020 00 0 00050        TRA   FAD1      IF FAD, SKIP                             QFAD0056
00043  0774 00 1 00003        AXT   3,1       IF FSB, REVERSE THE SIGN OF 2ND OPERAND  QFAD0057
00044  0500 00 1 00474        CLA   OP2+3,1                                            QFAD0058
00045  0760 00 0 00002        CHS                                                      QFAD0059
00046  0601 00 1 00474        STO   OP2+3,1                                            QFAD0060
00047  2 00001 1 00044        TIX   *-3,1,1                                            QFAD0061

00050  0500 00 0 00470  FAD1  CLA   OP1+3     FAD SECTION STARTS                       QFAD0063
00051  0402 00 0 00474        SUB   OP2+3                                              QFAD0064
00052  0601 00 0 00475        STO   TEST        CHARACTERISTIC DIFFERENCE              QFAD0065
00053  0120 00 0 00062        TPL   FAD2                                               QFAD0066
00054  0774 00 1 00004        AXT   4,1       ORGANIZE SO THAT CHAR OF OP1 IS          QFAD0067
00055  0500 00 1 00471        CLA   OP1+4,1     GREATER THAN OR EQUAL TO CHAR OF OP2    QFAD0068
00056  0560 00 1 00475        LDQ   OP2+4,1                                            QFAD0069
00057  0601 00 1 00475        STO   OP2+4,1                                            QFAD0070
00060 -0600 00 1 00471        STQ   OP1+4,1                                            QFAD0071
00061  2 00001 1 00055        TIX   *-4,1,1                                            QFAD0072

00062  0500 00 0 00475  FAD2  CLA   TEST      IF CHARACTERISTIC DIFFERENCE             QFAD0074
00063 -0340 00 0 00501        LAS   =105        IS MORE THAN 104, OP1 IS ANSWER        QFAD0075
00064  0020 00 0 00452        TRA   FIN                                                QFAD0076
00065  0020 00 0 00452        TRA   FIN                                                QFAD0077
00066  0737 00 1 00000        PAC   ,1        XR2 = - CHAR DIFFERENCE                  QFAD0078
00067  3 77735 1 00076  FAD3  TXH   FAD4,1,-35 SHIFT OP2 TO THE RIGHT BY -XR2,         QFAD0079
00070 -3 00000 1 00105        TXL   FAD5,1,0    FIRST SHIFT BY WORD IF NECESSARY       QFAD0080
00071  0443 00 0 00471        DLD   OP2                                                QFAD0081
00072 -0603 00 0 00472        DST   OP2+1                                              QFAD0082
00073  0760 00 0 00000        CLM                                                      QFAD0083
00074  0601 00 0 00471        STO   OP2                                                QFAD0084
00075  1 00043 1 00067        TXI   FAD3,1,35                                          QFAD0085
00076  0500 00 0 00472  FAD4  CLA   OP2+1       THEN SHIFT WITHIN EACH WORD            QFAD0086
00077  0560 00 0 00473        LDQ   OP2+2                                              QFAD0087
00100  0765 00 1 00000        LRS   0,1                                                QFAD0088
00101 -0600 00 0 00473        STQ   OP2+2                                              QFAD0089
00102  0443 00 0 00471        DLD   OP2                                                QFAD0090
00103  0765 00 1 00000        LRS   0,1                                                QFAD0091
00104 -0603 00 0 00471        DST   OP2                                                QFAD0092

00105 -0500 00 0 00465  FAD5  CAL   OP1       COMPARE SIGN OF OP1 WITH SIGN OF OP2     QFAD0094
00106  0322 00 0 00471        ERA   OP2                                                QFAD0095
00107 -0760 00 0 00001        PBT                                                      QFAD0096
00110  0020 00 0 00202        TRA   FAD9      IF SAME SIGN, SKIP                       QFAD0097
00111  0774 00 1 00003        AXT   3,1       CASE OF OPPOSITE SIGN                    QFAD0098
00112 -0500 00 1 00474        CAL   OP2+3,1     INVERT SIGN OF OP2 AND REPLACE         QFAD0099
00113  0760 00 0 00006        COM               MAGNITUDE OF OP2 WITH ITS ONES         QFAD0100
00114  0602 00 1 00474        SLW   OP2+3,1       COMPLEMENT                           QFAD0101
00115  2 00001 1 00112        TIX   *-3,1,1                                            QFAD0102
00116  0074 00 4 00207        TSX   ADD,4       ADD OP1 TO THIS                        QFAD0103
00117 -0760 00 0 00001        PBT                                                      QFAD0104
00120  0020 00 0 00133        TRA   FAD6                                               QFAD0105
00121  0500 00 0 00476        CLA   =1        IF PBT ON, /OP1/ GRT THAN /OP2/          QFAD0106
00122  0763 00 0 00000        LLS   0           ADD LAST BIT 1 TO MAGNITUDE OF         QFAD0107
00123  0400 00 0 00467        ADD   OP1+2         SUM TO COMPENSATE FOR SHORTAGE       QFAD0108
00124  0601 00 0 00467        STO   OP1+2           DUE TO ONES COMPLEMENT             QFAD0109
00125  0765 00 0 00043        LRS   35                                                 QFAD0110
```

```
00126  0400 00 0 00466        ADD    OP1+1                                          QFAD0111
00127  0765 00 0 00043        LRS    35                                             QFAD0112
00130  0400 00 0 00465        ADD    OP1              FINAL P-CARRY IS NOT POSSIBLE QFAD0113
00131 -0603 00 0 00465        DST    OP1                                            QFAD0114
00132  0020 00 0 00140        TRA    FAD7                                           QFAD0115
00133  0774 00 1 00003  FAD6  AXT    3,1          IF PBT OFF, /OP1/ LESS OR EQUAL /OP2/ QFAD0116
00134 -0500 00 1 00470        CAL    OP1+3,1        INVERT SIGN OF SUM AND REPLACE  QFAD0117
00135  0760 00 0 00006        COM                    MAGNITUDE OF SUM WITH ITS ONES QFAD0118
00136  0602 00 1 00470        SLW    OP1+3,1           COMPLEMENT                   QFAD0119
00137  2 00001 1 00134        TIX    *-3,1,1                                        QFAD0120

00140  0774 00 1 00003  FAD7  AXT    3,1          NORMALIZE THE SUM IF CASE OF OPPOSITE QFAD0122
00141  0520 00 0 00465        ZET    OP1                                      SIGN  QFAD0123
00142  0020 00 0 00161        TRA    FAD8           FIRST NORMALIZE BY WORD         QFAD0124
00143  0500 00 0 00466        CLA    OP1+1                                          QFAD0125
00144  0560 00 0 00467        LDQ    OP1+2                                          QFAD0126
00145 -0603 00 0 00465        DST    OP1                                            QFAD0127
00146  0760 00 0 00000        CLM                                                   QFAD0128
00147  0601 00 0 00467        STO    OP1+2                                          QFAD0129
00150  0500 00 0 00470        CLA    OP1+3                                          QFAD0130
00151  0402 00 0 00500        SUB    =35                                            QFAD0131
00152 -0120 00 0 00155        TMI    ZRANS          UNDERFLOW UNLIKELY UNLESS 0-0   QFAD0132
00153  0601 00 0 00470        STO    OP1+3                                          QFAD0133
00154  2 00001 1 00141        TIX    FAD7+1,1,1                                     QFAD0134
00155  0774 00 1 00004  ZRANS AXT    4,1          IF UNDERFLOW OR CLEAN ZERO,     QFAD0135
00156  0600 00 1 00471        STZ    OP1+4,1        GIVE + ZERO AS ANSWER           QFAD0136
00157  2 00001 1 00156        TIX    *-1,1,1                                        QFAD0137
00160  0020 00 0 00452        TRA    FIN                                            QFAD0138
00161  0774 00 1 00043  FAD8  AXT    35,1           FURTHER NORMALIZE BY BITS       QFAD0139
00162  0443 00 0 00465        DLD    OP1                                            QFAD0140
00163  0763 00 0 00001        LLS    1                                              QFAD0141
00164 -0760 00 0 00001        PBT                                                   QFAD0142
00165  1 77777 1 00163        TXI    *-2,1,-1                                       QFAD0143
00166  0765 00 0 00001        LRS    1                                              QFAD0144
00167  0601 00 0 00465        STO    OP1                                            QFAD0145
00170  0500 00 0 00466        CLA    OP1+1                                          QFAD0146
00171  0560 00 0 00467        LDQ    OP1+2                                          QFAD0147
00172  0763 00 1 00043        LLS    35,1                                           QFAD0148
00173 -0603 00 0 00466        DST    OP1+1                                          QFAD0149
00174  0754 00 1 00000        PXA    ,1                                             QFAD0150
00175  0400 00 0 00470        ADD    OP1+3                                          QFAD0151
00176  0402 00 0 00500        SUB    =35                                            QFAD0152
00177 -0120 00 0 00155        TMI    ZRANS                                          QFAD0153
00200  0601 00 0 00470        STO    OP1+3                                          QFAD0154
00201  0020 00 0 00452        TRA    FIN                                            QFAD0155

00202  0074 00 4 00207  FAD9  TSX    ADD,4          CASE OF SAME SIGN               QFAD0157
00203 -0760 00 0 00001        PBT                                                   QFAD0158
00204  0020 00 0 00452        TRA    FIN              IF NO CARRY, SKIP            QFAD0159
00205  0074 00 4 00222        TSX    LRS1,4           IF THERE IS A CARRY,         QFAD0160
00206  0020 00 0 00452        TRA    FIN                GIVE RIGHT SHIFT ADJUSTMENT QFAD0161
```



```
00207  0500 00 0 00467  ADD   CLA    OP1+2          INTEGER ADDITION SUBROUTINE    QFAD0163
00210  0400 00 0 00473        ADD    OP2+2                                          QFAD0164
00211  0765 00 0 00043        LRS    35                                             QFAD0165
00212 -0600 00 0 00467        STO    OP1+2                                          QFAD0166
00213  0400 00 0 00466        ADD    OP1+1                                          QFAD0167
00214  0400 00 0 00472        ADD    OP2+1                                          QFAD0168
00215  0765 00 0 00043        LRS    35                                             QFAD0169
00216  0400 00 0 00465        ADD    OP1                                            QFAD0170
00217  0400 00 0 00471        ADD    OP2                                            QFAD0171
00220 -0603 00 0 00465        DST    OP1                                            QFAD0172
00221  0020 00 4 00001        TRA    1,4                                            QFAD0173

00222  0765 00 0 00001  LRS1  LRS    1              SUBROUTINE TO SHIFT            QFAD0175
00223  0601 00 0 00465        STO    OP1              MANTISSA TO RIGHT BY 1       QFAD0176
00224  0500 00 0 00466        CLA    OP1+1              AND TO ADD 1 TO CHARACTERISTIC QFAD0177
00225 -0600 00 0 00466        STO    OP1+1                                          QFAD0178
00226  0560 00 0 00467        LDQ    OP1+2                                          QFAD0179
00227  0765 00 0 00001        LRS    1                                              QFAD0180
00230 -0600 00 0 00467        STO    OP1+2                                          QFAD0181
00231  0500 00 0 00470        CLA    OP1+3                                          QFAD0182
00232  0400 00 0 00476        ADD    =1                                             QFAD0183
00233 -0760 00 0 00001        PBT                                                   QFAD0184
00234  0020 00 0 00236        TRA    *+2                                            QFAD0185
00235  0074 00 4 00000        CALL   DUMP             IF OVERFLOW, GIVE DUMP        QFAD0186
00236  0601 00 0 00470        STO    OP1+3                                          QFAD0187
00237  0020 00 4 00001        TRA    1,4                                            QFAD0188

00240 -0520 00 0 00464  FMDP  NZT    FLAG2          FMP OR FDP                     QFAD0190
00241  0020 00 0 00353        TRA    FMP1             IF FMP, SKIP                 QFAD0191

00242  0500 00 0 00471        CLA    OP2            FDP SECTION STARTS             QFAD0193
00243 -0100 00 0 00245        TNZ    *+2                                           QFAD0194
00244  0074 00 4 00000        CALL   DUMP             AVOID DIVISION BY ZERO       QFAD0195
00245  0560 00 0 00465        LDQ    OP1                                            QFAD0196
00246  0760 00 0 00003        SSP                     COMPARE /OP1/ AND /OP2/       QFAD0197
00247  0765 00 0 00000        LRS    0                                              QFAD0198
00250  0040 00 0 00253        TLQ    *+3                                            QFAD0199
00251  0443 00 0 00465        DLD    OP1              IF /OP1/ GRT OR EQUAL /OP2/,  QFAD0200
00252  0074 00 4 00222        TSX    LRS1,4             SHIFT RIGHT AND SCALE OP1   QFAD0201
00253  0500 00 0 00470        CLA    OP1+3            COMPUTE PROVISIONAL CHARACTERISTIC QFAD0202
00254  0400 00 0 00502        ADD    =0200000000000   OF ANSWER = CHAR1-CHAR2+BASE  QFAD0203
00255  0402 00 0 00474        SUB    OP2+3              THIS MAY BE 1 TOO HIGH      QFAD0204
00256 -0120 00 0 00155        TMI    ZRANS              IF UNDERFLOW, GIVE +0 AS ANSWER QFAD0205
00257 -0760 00 0 00001        PBT                                                   QFAD0206
00260  0020 00 0 00262        TRA    *+2                                            QFAD0207
00261  0074 00 4 00000        CALL   DUMP             IF OVERFLOW, GIVE DUMP        QFAD0208
00262  0601 00 0 00470        STO    OP1+3                                          QFAD0209

00263  0443 00 0 00465        DLD    OP1              A1+A2+A3   A1+A2+A3    B2+B3  B2+B3  QFAD0211
00264  0100 00 0 00155        TZE    ZRANS            -------- = --------*(1------*(-----1**2) QFAD0212
W 00265  0220 00 0 00471      DVH    OP2              B1+B2+B3      B1         B1     B1   QFAD0213
00266 -0600 00 0 00465        STO    OP1                                            QFAD0214
00267  0560 00 0 00467        LDQ    OP1+2                                          QFAD0215
W 00270  0220 00 0 00471      DVH    OP2                                            QFAD0216
00271 -0600 00 0 00466        STO    OP1+1                                          QFAD0217
```

```
M 00272  0220 00 0 00471        DVH   OP2                                          QFAD0218
  00273 -0600 00 0 00467        STQ   OP1+2          (A1+A2+A3)/B1 IS READY       QFAD0219

  00274  0774 00 1 00000        AXT   0,1            XR1 FOR EXTRA DEFICIT ON     QFAD0221
  00275  0500 00 0 00472        CLA   OP2+1            MOST SIGNIF PART OF 2ND FACTOR QFAD0222
  00276  0765 00 0 00000        LRS   0                                           QFAD0223
  00277  0760 00 0 00003        SSP                                               QFAD0224
  00300 -0400 00 0 00471        SBM   OP2                                         QFAD0225
  00301 -0120 00 0 00304        TMI   **+3             IF /B1/ GRT THAN /B2*2**35/, SKIP QFAD0226
  00302  0763 00 0 00000        LLS   0                IF /B1/ LESS OR EQUAL /B2*2**35/, QFAD0227
  00303  1 00001 1 00305        TXI   **+2,1,1           SUBTRACT B1*2**-35 FROM B2  QFAD0228
  00304  0500 00 0 00472        CLA   OP2+1              AND REMEMBER QUOTIENT CARRY IN XR1 QFAD0229
  00305  0560 00 0 00473        LDQ   OP2+2                                       QFAD0230
M 00306  0220 00 0 00471        DVH   OP2                                         QFAD0231
  00307 -0600 00 0 00472        STQ   OP2+1          WRITE (B2+B3)/B1 = X1+X2+X3 = Y2+X3 QFAD0232
M 00310  0220 00 0 00471        DVH   OP2              X1 WHICH IS 0 OR 1 IS IN XR1 QFAD0233
  00311 -0600 00 0 00473        STQ   OP2+2              X2, X3 ARE IN OP2+1, OP2+2 QFAD0234
  00312  0560 00 0 00472        LDQ   OP2+1                                       QFAD0235
  00313  0200 00 0 00472        MPY   OP2+1                                       QFAD0236
  00314  0600 00 0 00475        STZ   TEST                                        QFAD0237
  00315 -3 00000 1 00324        TXL   FDP1,1,0         IF X1=0, Y2**2=X2**2, SKIP QFAD0238
  00316  0400 00 0 00472        ADD   OP2+1            IF X1=1, Y2**2=1+2*X2+X2**2 QFAD0239
  00317  0400 00 0 00472        ADD   OP2+1                                       QFAD0240
  00320  0765 00 0 00043        LRS   35                                          QFAD0241
  00321  0400 00 0 00476        ADD   =1                                          QFAD0242
  00322  0601 00 0 00475        STO   TEST                                        QFAD0243
  00323  0131 00 0 00000        XCA                                               QFAD0244

  00324  0601 00 0 00474  FDP1  STO   OP2+3          Y2**2 IN TEST AND OP2+3      QFAD0246
  00325  0754 00 1 00000        PXA   ,1                                          QFAD0247
  00326  0760 00 0 00006        COM                  TAKE TWOS COMPLEMENT OF X1+X2+X3 QFAD0248
  00327  0601 00 0 00471        STO   OP2                                         QFAD0249
  00330  0500 00 0 00472        CLA   OP2+1                                       QFAD0250
  00331  0760 00 0 00006        COM                                               QFAD0251
  00332  0601 00 0 00472        STO   OP2+1                                       QFAD0252
  00333  0500 00 0 00473        CLA   OP2+2                                       QFAD0253
  00334  0322 00 0 00503        ERA   =0377777777777   THIS COM KEEPS PBT OFF     QFAD0254
  00335  0400 00 0 00476        ADD   =1               WE WANT TO COME UP CLEAN IF B2=B3=0 QFAD0255
  00336  0400 00 0 00474        ADD   OP2+3          ADD Y2**2 TO THE RESULT      QFAD0256
  00337  0765 00 0 00043        LRS   35                                          QFAD0257
  00340 -0600 00 0 00473        STQ   OP2+2                                       QFAD0258
  00341  0400 00 0 00472        ADD   OP2+1                                       QFAD0259
  00342  0400 00 0 00475        ADD   TEST                                        QFAD0260
  00343  0765 00 0 00043        LRS   35                                          QFAD0261
  00344  0400 00 0 00471        ADD   OP2                                         QFAD0262
  00345 -0760 00 0 00001        PBT                  FINAL CARRY MEANS 2ND FACTOR =1, QFAD0263
  00346  0020 00 0 00350        TRA   **+2             SO, ANSWER IS READY IN OP1. QFAD0264
  00347  0020 00 0 00452        TRA   FIN                (THIS OCCURS IF B2=B3=0)  QFAD0265

  00350 -0603 00 0 00471        DST   OP2            NOW MANT OF 2ND FACTOR IS READY IN OP2 QFAD0267
  00351  0500 00 0 00502        CLA   =0200000000000                              QFAD0268
  00352  0601 00 0 00474        STO   OP2+3                                       QFAD0269
```



```
  00353  0500 00 0 00470  FMP1  CLA   OP1+3          FMP SECTION STARTS           QFAD0271
  00354  0400 00 0 00474        ADD   OP2+3            ADD EXPONENTS              QFAD0272
  00355  0402 00 0 00502        SUB   =0200000000000                              QFAD0273
  00356 -0120 00 0 00155        TMI   ZRANS          IF UNDERFLOW, GIVE +0        QFAD0274
  00357 -0760 00 0 00001        PBT                                               QFAD0275
  00360  0020 00 0 00362        TRA   **+2                                        QFAD0276
  00361  0074 00 4 00000        CALL  DUMP           IF OVERFLOW, GIVE DUMP       QFAD0277
  00362  0601 00 0 00470        STO   OP1+3          PROVISIONAL CHARACTERISTIC READY QFAD0278
  00363  0560 00 0 00467        LDQ   OP1+2                                       QFAD0279
  00364 -0200 00 0 00471        MPR   OP2            (A1+A2+A3)*(B1+B2+B3)        QFAD0280
  00365  0601 00 0 00467        STO   OP1+2                  = A1*B1              QFAD0281
  00366  0560 00 0 00473        LDQ   OP2+2                    +(A1*B2+A2*B1)     QFAD0282
  00367 -0200 00 0 00465        MPR   OP1                        +(A1*B3+A2*B2+A3*B1) QFAD0283
  00370  0601 00 0 00473        STO   OP2+2                                       QFAD0284
  00371  0560 00 0 00466        LDQ   OP1+1                                       QFAD0285
  00372 -0200 00 0 00472        MPR   OP2+1                                       QFAD0286
  00373  0400 00 0 00467        ADD   OP1+2                                       QFAD0287
  00374  0400 00 0 00473        ADD   OP2+2                                       QFAD0288
  00375  0765 00 0 00043        LRS   35                                          QFAD0289
  00376  0601 00 0 00475        STO   TEST                                        QFAD0290
  00377 -0600 00 0 00467        STQ   OP1+2          LOWEST RUNG ACCOUNTED        QFAD0291
  00400  0560 00 0 00472        LDQ   OP2+1                                       QFAD0292
  00401  0200 00 0 00465        MPY   OP1                                         QFAD0293
  00402 -0603 00 0 00472        DST   OP2+1                                       QFAD0294
  00403  0560 00 0 00466        LDQ   OP1+1                                       QFAD0295
  00404  0200 00 0 00471        MPY   OP2                                         QFAD0296
  00405  0601 00 0 00466        STO   OP1+1                                       QFAD0297
  00406  0131 00 0 00000        XCA                                               QFAD0298
  00407  0400 00 0 00473        ADD   OP2+2                                       QFAD0299
  00410  0400 00 0 00467        ADD   OP1+2                                       QFAD0300
  00411  0765 00 0 00043        LRS   35                                          QFAD0301
  00412 -0600 00 0 00467        STQ   OP1+2          LOWEST PART READY            QFAD0302
  00413  0400 00 0 00475        ADD   TEST                                        QFAD0303
  00414  0400 00 0 00472        ADD   OP2+1                                       QFAD0304
  00415  0400 00 0 00466        ADD   OP1+1                                       QFAD0305
  00416  0765 00 0 00043        LRS   35                                          QFAD0306
  00417 -0603 00 0 00472        DST   OP2+1          MIDDLE RUNG ACCOUNTED        QFAD0307
  00420  0560 00 0 00471        LDQ   OP2                                         QFAD0308
  00421  0200 00 0 00465        MPY   OP1                                         QFAD0309
  00422  0601 00 0 00465        STO   OP1                                         QFAD0310
  00423  0131 00 0 00000        XCA                                               QFAD0311
  00424  0400 00 0 00473        ADD   OP2+2                                       QFAD0312
  00425  0765 00 0 00043        LRS   35                                          QFAD0313
  00426  0400 00 0 00472        ADD   OP2+1                                       QFAD0314
  00427  0400 00 0 00465        ADD   OP1                                         QFAD0315
  00430 -0603 00 0 00465        DST   OP1            ALL 3 PARTS ADDED            QFAD0316

  00431  0763 00 0 00001        LLS   1                                           QFAD0318
  00432 -0760 00 0 00001        PBT                  TEST FOR NORMALIZATION       QFAD0319
  00433  0020 00 0 00435        TRA   **+2                                        QFAD0320
  00434  0020 00 0 00452        TRA   FIN              IF ALREADY NORMAL, SKIP    QFAD0321
  00435  0763 00 0 00001        LLS   1                                           QFAD0322
  00436 -0760 00 0 00001        PBT                                               QFAD0323
  00437  0020 00 0 00155        TRA   ZRANS            IF 2 LEADING 0, GIVE +0    QFAD0324
  00440  0771 00 0 00001        ARS   1                IF 1 LEADING 0, NORMALIZE BY LLS 1 QFAD0325
  00441  0601 00 0 00465        STO   OP1                                         QFAD0326
```

```
00442  0500 00 0 00466          CLA     OP1+1                                  QFAD0327
00443  0560 00 0 00467          LDQ     OP1+2                                  QFAD0328
00444  0763 00 0 00001          LLS     1                                      QFAD0329
00445 -0603 00 0 00466          DST     OP1+1                                  QFAD0330
00446  0500 00 0 00470          CLA     OP1+3                                  QFAD0331
00447  0402 00 0 00476          SUB     =1                                     QFAD0332
00450 -0120 00 0 00155          TMI     ZRANS           IF UNDERFLOW, GIVE +0  QFAD0333
00451  0601 00 0 00470          STO     OP1+3                                  QFAD0334

00452  0774 00 1 00004   FIN    AXT     4,1             ALL DONE, REPORT THE ANSWER  QFAD0336
00453  0500 00 1 00471          CLA     OP1+4,1                                QFAD0337
00454  0601 00 1 00000   CAANS  STO     **,1                                   QFAD0338
00455  2 00001 1 00453          TIX     *-2,1,1                                QFAD0339
00456  0774 00 1 00000   XR1    AXT     **,1            RETURN                 QFAD0340
00457  0774 00 4 00000   XR4    AXT     **,4                                   QFAD0341
00460  0600 00 0 00463          STZ     FLAG1                                  QFAD0342
00461  0600 00 0 00464          STZ     FLAG2                                  QFAD0343
00462  0020 00 4 00004          TRA     4,4                                    QFAD0344

00463  0 00000 0 00000   FLAG1  PZE                                             QFAD0346
00464  0 00000 0 00000   FLAG2  PZE                                             QFAD0347
00465                           EVEN                                           QFAD0348
00465                    OP1    BSS     4                                      QFAD0349
00471                    OP2    BSS     4                                      QFAD0350
00475                    TEST   BSS     1                                      QFAD0351
                                END                                            QFAD0352

  LITERALS
00476  000000000001
00477  000000000004
00500  000000000043
00501  000000000151
00502  200000000000
00503  377777777777
```

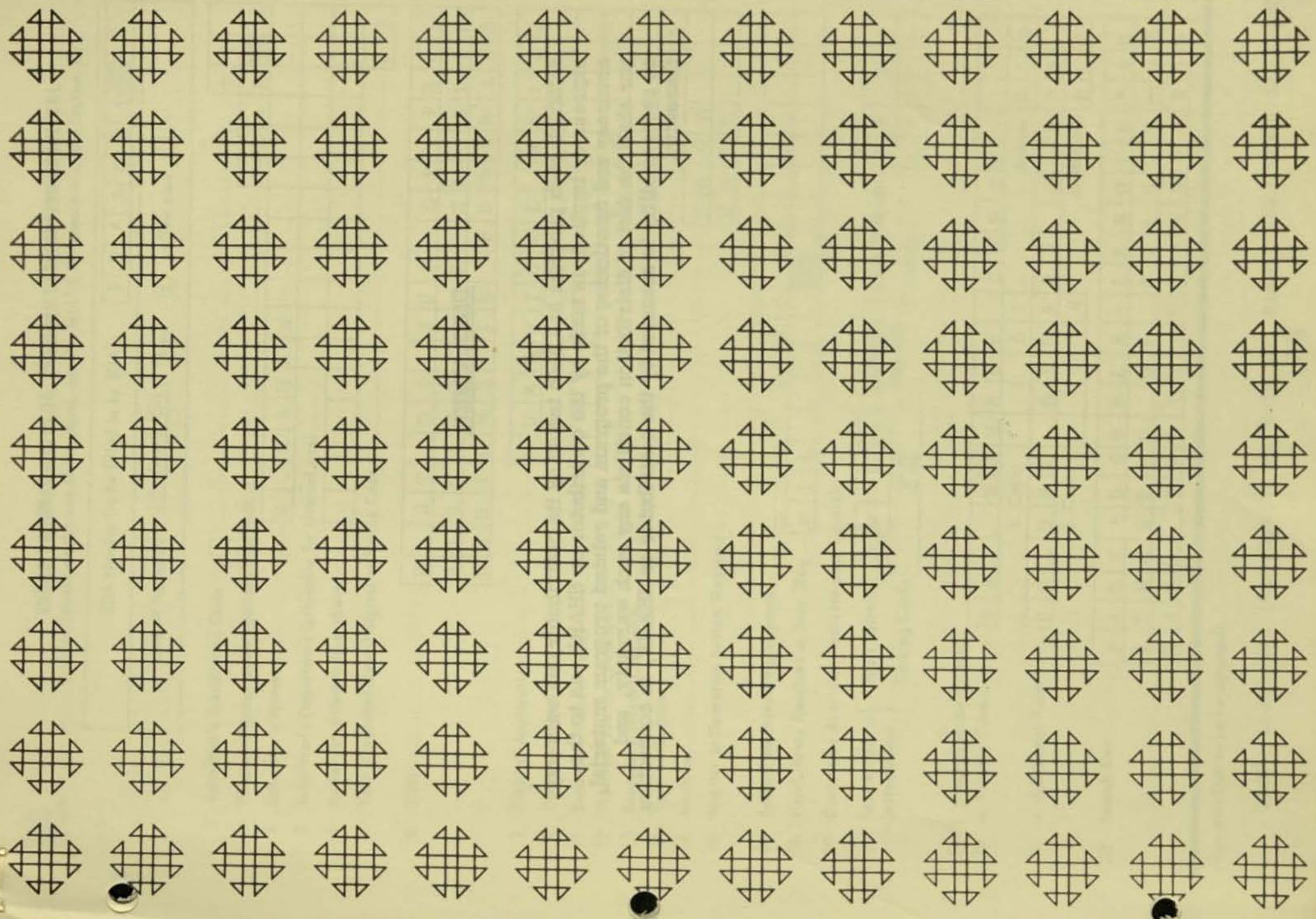

504 IS THE FIRST LOCATION NOT USED BY THIS PROGRAM

REFERENCES TO PROGRAMMER DEFINED SYMBOLS

```
207    ADD     116,  202
452    FIN      64,   65,  160,  201,  204,  206,  347,  434
465    OP1      34,   50,   55,   60,  105,  123,  124,  126,  130,  131,  134,  136,  141,  143,  144,  145,  147
               150,  153,  156,  162,  167,  170,  171,  173,  175,  200,  207,  212,  213,  216,  220,  223,  224
               225,  226,  230,  231,  236,  245,  251,  253,  262,  263,  266,  267,  271,  273,  353,  362,  363
               365,  367,  371,  373,  377,  401,  403,  405,  410,  412,  415,  421,  422,  427,  430,  441,  442
               443,  445,  446,  451,  453
471    OP2      35,   44,   46,   51,   56,   57,   71,   72,   74,   76,   77,  101,  102,  104,  106,  112,  114
               210,  214,  217,  242,  255,  265,  270,  272,  275,  300,  304,  305,  306,  307,  310,  311,  312
               313,  316,  317,  324,  327,  330,  332,  333,  336,  340,  341,  344,  350,  352,  354,  364,  366
               370,  372,  374,  400,  402,  404,  407,  414,  417,  420,  424,  426
456    XR1      9
457    XR4      6
  0    DUMP    235,  244,  261,  361
 50    FAD1     42
 62    FAD2     53
 67    FAD3     75
 76    FAD4     67
105    FAD5     70
133    FAD6    120
140    FAD7    132,  154
161    FAD8    142
202    FAD9    110
324    FDP1    315
240    FMDP     40
353    FMP1    241
222    LRS1    205,  252
  5    QFAD      0,    3
  1    QFDP      0
  2    QFMP      0
  4    QFSB      0
475    TEST     52,   62,  314,  322,  342,  376,  413
454    CAANS    30
 32    CAOP1    14
 33    CAOP2    22
463    FLAG1     2,   37,  460
464    FLAG2     1,    4,   41,  240,  461
155    ZRANS   152,  177,  256,  264,  356,  437,  450
```

FLAGS IN ABOVE ASSEMBLY.
LINES-PREEXECUTION 1094
L=1094 I=11.17, O=11.17, U=.51 MINS. DATE =08/24/66

* * * * * * * * * END OF JOB 1249.00 AT 11.20, 0026 PAGES PRINTED. * * * * * * * * *



DISCLAIMER

Although each program has been tested by its contributor, no warranty, express or implied, is made by the contributor or SHARE, as to the accuracy and functioning of the program and related program material, nor shall the fact of distribution constitute any such warranty, and no responsibility is assumed by the contributor or SHARE, in connection therewith.

Input Field Number

SHARE PROGRAM CATALOG, PROGRAM DESCRIPTION SUBMITTAL

(For details on the use of this form, see the SHARE Reference Manual, Section 06)

Card Columns

1 SDA Number (to be filled in by SDA): SDA3021 — A13-21

Date of Submittal: 15 Jan./64 New [X] or Revision [ ]

2	Submitter's Installation Code	TY	ALL CARDS 1-3
3	Program Number or Designation (and Suffix)	DLAF	4-8
4	Submitter's Name	W. KAHAN	A22-38
5	Submitter's Department (primarily for internal use)		A39-50
6	Author (if different from above)		A51-67
7	Year Completed (last 2 digits) or Status Code	63	A68-69
8	Title	FAST DP SUM OF PROD UCTS ETC. OF SP EXPR ESSIONS IN FORTRAN 2	Title Card O12-68
9	Field of Application	NUMERICAL ANAL.	B12-26
10	Primary Subject Code	A1	B27-29
11	Secondary Subject Codes		B30-41
12	Principal Source Language	FAP	B42-48
13	Secondary Source Language		B49-55
14	Type of Routine	SR	B56-57
15	Machine	7090	B58-64
16	Monitor or Operating System Required	FMS2	B65-71
17	Special Machine Requirements	NIL	C12-23
18	Non-Library Routines or Subr. Req.	NIL	C24-39

19 Documents Available (indicate page counts):

Sent to all Installations — Short Write-up: SW 2

Catalog Cards: 52

Orderable from SDA — Long Write-up: PA 10 — C40-49

Listing: LS 2 — C50-54

20-21 Program Material Avail.

20 a. Primary Form: FAP SOURCE CARDS — D12-31

b. Count: 67 c. Medium: BCD — D32-39

21 a. Additional Form: COLUMN BINARY CARDS — D40-59

b. Count: 4 c. Medium: BIN — D60-67

22 Search Key: FAST *DOUBLE-LENGTH *ARITHMETIC ON SINGLE-LENGTH FORTRAN 2 NUMBERS — Search Card S12-74

Remarks (not to be keypunched)

DLAF obsoletes (because it extends and simplifies) DLAP, SD #1480.

SHARE PROGRAM CATALOG, PROGRAM DESCRIPTION SUBMITTAL (Continued)

Abstract (Cards 10-99, Columns 12-72)

PURPOSE - To provide facilities, analogous to those on a desk calculator, which in FORTRAN 2 permit double-precision accumulation of products of single-precision numbers. Some additional arithmetic operations are provided; all are outlined in the following table in which D represents the contents of a double-precision pseudo-accumulator and X and Y are single-precision expressions.

FORTRAN STATEMENT	SYMBOLIC DEFINITION
CALL DLADF(X)	D = D+X
CALL DLMPF(X)	D = D*X
CALL DLDVF(X)	D = D/X
CALL DLADDF(X,Y)	D = D+(X,Y)
CALL DLADDF(X*Y)	D = D+(X*Y)
CALL DLADDF(X-Y)	D = D+(X-Y)
CALL DLSETF(X,Y)	D = (X,Y)
CALL DLSETF(X*Y)	D = X*Y
CALL DLSETF(X-Y)	D = X-Y

The value of D , rounded to single-precision, can be obtained after each arithmetic operation above if the word CALL is replaced by, say, Z = . The double-precision value of D can be stored via CALL DLSTO(A,B) which sets (A,B) = D .

SPECIAL FEATURES - These programs are faster and use less storage than the double-precision statements in the FORTRAN 2 system, and may be more convenient to use. A timing table and several examples (listed below) are provided in the PA.

cont'd..... (Please attach additional pages, if necessary)

Pages Attached: keypunchable abstract 1
Non-keypunchable short write-up nil

Signature of Submitter W. Kahan Date 15 Jan /64

Signature of Installation Addressee B. A. Worsley

Printed in U.S.A. 328-1926

LIBRARY TAPE SUBPROGRAMS "DLAD F", "DLMP F", "DLDV F", "DLADD F", "DLSET F", "DLST@".

PURPOSE: To provide facilities, analogous to those on a desk calculator, which in FORTRAN II permit double-precision accumulation of products of single-precision numbers, and further arithmetic operations useful in matrix calculations and other areas of Numerical Analysis. To use these subprograms may be more convenient than to use FORTRAN II's (or FORTRAN IV's) double-precision arithmetic statements; besides, DLAD F etc. require less storage and are faster.

HOW TO USE DLADF(X) etc.:

These subprograms perform double-precision arithmetic with the aid of an internal double-length accumulator whose value will be denoted by either DLA or (DLA1,DLA2) . Here DLA1 is the most significant and DLA2 the least significant part of DLA .

In what follows X and Y are single-length floating point expressions, and (X@Y) stands for one of

(X,Y) , (X*Y) , (X+Y) or (X-Y) , but not (X/Y) .

Here (X,Y) denotes again the double-length number whose most significant part is X , least significant Y . (X*Y) denotes the double-length product of the single length numbers X and Y , and (X+Y) is the double-length result of adding/subtracting X and Y ; whereas in FORTRAN these results would be truncated to single-length (27 bits), they are treated as double-length (54-bit) numbers whenever they appear as arguments of DLADD F or DLSET F .

In the table below, DLA+X represents the double-length sum obtained by adding the single-length X to DLA . Similarly for DLA*X and DLA/X . A and B are names of single-length floating point variables. A,B,X and Y may be subscripted as usual.

FORTRAN Statement	Symbolic Definition
CALL DLADF(X)	DLA = DLA + X
CALL DLMPF(X)	DLA = DLA * X
CALL DLDVF(X)	DLA = DLA/X
CALL DLADDF(X@Y)	DLA = DLA+(X@Y)
CALL DLSETF(X@Y)	DLA = (X@Y)
CALL DLST@(A,B)	A = DLA1 and B = DLA2

The first five subprograms (all but DLST@) may be used as functions in single-length arithmetic expressions, in which case their value is obtained by rounding the latest value of DLA to single-length. For example, the statement

Z = DLADDF(X*Y)

is symbolically equivalent to the two statements

```
DLA = DLA + X*Y        to double-precision
Z   = FRNF(DLA1,DLA2)  .                    precision)
```

(FRN F is described in SHARE DISTRIBUTION NO.1502; it rounds to single
However, note that the only way to alter or extract the value of DLA is to use the six subprograms DLAD F etc. as described above. Direct references in FORTRAN to DLA , DLA1 or DLA2 are useless. For example, a statement like

```
CALL DLSETF(0.,0.)    or    CALL DLSETF(0,0)
```

is the only way to clear DLA ; the more general statement

```
CALL DLSETF(X,0)
```

sets the double-length value of DLA equal to the single-length (perhaps truncated) value of the expression X . The statement

```
CALL DLSETF(A,B)
```

would normally be used only with values of A and B that had previously appeared in some statement

```
CALL DLST⊙(A,B)  .
```

Note that the value of A in the last statement equals the truncated value of DLA ; the statement

```
A = DLADDF(0,0)
```

would be used to set A equal to the rounded value of DLA .

In (X⊙Y) there must be no doubt about the identity of X, ⊙ and Y . For example, (P*(Q-R)*S) is ambiguous, but
((P*(Q-R))*S) has X = P*(Q-R) , ⊙ = * , Y = S ;
(P*((Q-R)*S)) has X = P , ⊙ = * , Y = (Q-R)*S ; and
(P*Q*S-P*R*S) has X = P*Q*S , ⊙ = - , Y = P*R*S .

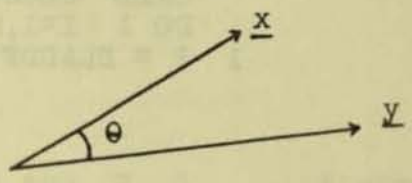
Because X and Y are truncated to single-length, these three expressions (X⊙Y) may all have different values. It helps to remember that when dealing with single-length floating point arguments the 7090 automatically produces double-length sums, differences and products and only truncates these when they are themselves to be used as arguments in further calculations. Since automatically produced quotients are just single-length, we cannot let (X⊙Y) be (X/Y) ; for double-length quotients we must use DLDV F .

Note: DLDVF(0.0) = ±1.585E38 with the correct sign, sets DLA to this same value, and records ⊙VERFL⊙W .

EXAMPLES: (1) $S = \sum_{1}^{N} X_i$ for large N

(2) $P = \underline{x}^*\underline{y}$ (scalar product of two vectors)

(3) $C = \cos\theta = \underline{x}^*\underline{y} \,/\, \sqrt{\underline{x}^*\underline{x}\ \underline{y}^*\underline{y}}$



(4) $S2 = \sin^2\theta$

(5) Residual $\underline{r} = \underline{b}-A\underline{x}$

(6) Solve ill-conditioned equations $a_1x+b_1y = c_1$
$a_2x+b_2y = c_2$

(7) Solve quadratic $ax^2+bx+c = 0$ with near-equal roots

(8) Evaluate Polynomial by nested multiplication

(9) Large Matrix multiplication

(10) Rayleigh Quotients

(11) Straight line fit by least squares

In all examples except the last, the stored values of the arguments are assumed to be exact, and it is supposed that there is a good reason to calculate with extra precision.

Example 1 $S = \sum_{i=1}^{N} X_i$ for large N

```
      CALL  DLSETF(0,0)
      DO 1  I=1,N
    1 S = DLADF(X(I))
```

This would be useful, for example, in numerical quadrature or infinite series.

4

Example 2 Scalar product $P = x^*y = \sum_{i=1}^{N} X_i Y_i$

```
      CALL  DLSETF(0,0)
      DO 1  I=1,N
    1 P = DLADDF(X(I)*Y(I))
```

Example 3 $C = \cos\theta = \underline{x}^*\underline{y} \,/\, \sqrt{\underline{x}^*\underline{x}\ \underline{y}^*\underline{y}}$

$$= \sum_{1}^{N} X_i Y_i / \left(\sum_{1}^{N} X_i^2 \sum_{1}^{N} Y_j^2\right)^{\frac{1}{2}}$$

```
      XX = 0.0
      YY = 0.0
      CALL  DLSETF(0,0)
      DO 1  I=1,N
      C = DLADDF(X(I)*Y(I))
      XX = FRNF(XX+FRNF(X(I)*X(I)))
    1 YY = FRNF(YY+FRNF(Y(I)*Y(I)))
      C = C/SQRTF(XX*YY)
```

This program will yield C = cos θ to within a few units in its last place, except if overflow or underflow occurs. There is no need to use DLADD F to calculate XX and YY , because there can be no cancellation there, unless N is very large in which case despite the use of FRN F there may be an error as large as N/2 units in the last place. Such an event is most unlikely.

Example 4 $S2 = \sin^2\theta$, where θ was defined above.

Here we cannot use S2 = 1.0-C*C because if S2 is small it must be the inaccurate result of cancellation between 1.0 and a value of C*C very near 1.0 . Therefore, we use Lagrange's identity instead:

$$\sin^2\theta = \tfrac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} (X_i Y_j - X_j Y_i)^2 \,/\, \left(\sum_{1}^{N} X_i^2 \sum_{1}^{N} Y_j^2\right)$$

The double sum $\tfrac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N}$ can be replaced by $\sum_{i=2}^{N} \sum_{j=1}^{i-1}$

because of the symmetry in the indices i and j , provided N > 1 .

```
      XX = FRNF(X(1)*X(1))
      YY = FRNF(Y(1)*Y(1))
      S2 = 0.0
      DO 2  I=2,N
      DO 1  J1=2,I
      CALL  DLSETF(X(I)*Y(J1-1))
      XY = DLADDF(-X(J1-1)*Y(I))
    1 S2 = FRNF(S2+FRNF(XY*XY))
      XX = FRNF(XX+FRNF(X(I)*X(I)))
    2 YY = FRNF(YY+FRNF(Y(I)*Y(I)))
      S2 = S2/(XX*YY)
```

Example 5 Residual vector $\underline{r} = \underline{b}-A\underline{x}$

i.e. $r_i = b_i - \sum_{j=1}^{N} a_{ij}x_j \qquad i=1,2,\ldots,M$

If $\underline{x}$ nearly satisfies $A\underline{x} = \underline{b}$ then $\underline{r}$ will be the result of cancellation, and unless DLADD F were used the rounding errors in $\underline{r}$'s calculation could swamp $\underline{r}$.

```
      DO 1  I=1,M
      CALL  DLSETF(B(I),0)
      DO 1  J=1,N
    1 R(I) = DLADDF(-A(I,J)*X(J))
```

Example 6 Solve the ill-conditioned equations

$$a_1x+b_1y = c_1 \quad , \quad a_2x+b_2y = c_2 \quad .$$

Although the coefficients a,b,c may be exact, the solution

$$x = (c_1b_2-b_1c_2)/D \quad , \quad y = (a_1c_2-a_2c_1)/D$$

cannot be calculated accurately unless the determinant

$$D = a_1b_2-a_2b_1$$

is calculated carefully.

```
      CALL  DLSETF(A1*B2)
      D = DLADDF(-A2*B1)
      CALL  DLSETF(C1*B2)
      X = DLADDF(-C2*B1)/D
      CALL  DLSETF(A1*C2)
      Y = DLADDF(-A2*C1)/D
```

Example 7 Solve the quadratic $ax^2+bx+c = 0$.

If the roots are nearly equal the discriminant b^2-4ac will be so small that it could be obscured by rounding errors unless DLADD F were used. Note that 0.5*B is calculated exactly in our machine because it uses binary arithmetic.

```
      X = -0.5*B
      CALL  DLSETF(X*X)
      Y = SQRTF(DLADDF(-A*C))
      IF (Y)  1,2,2
1     X = X/A
      Y = Y/A
      complex roots are  X±iY
      .....
2     R1 = FRNF(X+SIGNF(Y,X))/A
      R2 = C/(A*R1)
      real roots are  R1, R2
      .....
```

The use of DLADD F etc. in this example is pointless if A,B and C are inaccurate themselves.

Example 8 $P = a_0x^N+a_1x^{N-1}+a_2x^{N-1}+\ldots+a_{N-1}x+a_N$, $N > 0$

Extra accuracy is needed when x is near a root of $P = 0$. The following program uses nested multiplication;

$$P = (..((a_0x+a_1)x+a_2)x+ \ldots + a_{N-1})x+a_N$$

```
      CALL  DLSETF(A(0),0)
      DO 1  I=1,N
      CALL  DLMPF(X)
1     P = DLADF(A(I))
```

In Newton's iteration one would need $P1 = dP/dx$ in order to calculate the new value of x

i.e. $x-P/P1$.

Here accuracy in P is far more important than in P1 , which can be calculated easily by extending the recurrence used above thus:

```
      P = DLSETF(A(0),0)
      P1 = 0.0
      DO 1  I=1,N
      P1 = FRNF(P+FRNF(X*P1))
      CALL  DLMPF(X)
1     P = DLADF(A(I))
```

7

Example 9 Large Matrix Multiplication

$$A = B*C \; ; \quad \text{i.e.} \quad a_{ij} = \sum_{k=1}^{M} b_{ik}c_{kj}$$

for $i=1,2,\ldots,L$, $j=1,2,\ldots,N$.

This program is worthwhile only if M is very large.

```
      DO 2  I=1,L
      DO 2  J=1,N
      CALL  DLSETF(0,0)
      DO 1  K=1,M
    1 CALL  DLADDF(B(I,K)*C(K,J))
    2 A(I,J) = DLADDF(0,0)
```

(A(I,J) = ... is put into statement 2 instead of statement 1 in order to achieve more efficient indexing in the object program.)

Example 10 Rayleigh Quotients

If A is a symmetric N x N matrix ($a_{ij} = a_{ji}$) , α is an approximation to an eigenvalue of A , and $\underline{v}$ is an approximation to the corresponding eigenvector, then the error in α and in $\underline{v}$ can be determined with the aid of the residual $\underline{w} = (A-\alpha I)\underline{v}$, $\epsilon^2 = \underline{w}*\underline{w}/\underline{v}*\underline{v}$, and the Rayleigh quotient $Q = \underline{v}*\underline{w}/\underline{v}*\underline{v}$. These are both usually very small if α and $\underline{v}$ are good approximations.

```
      Q = 0.0
      Q2 = 0.0
      EPS2 = 0.0
      VV = 0.0
      DO 2  I=1,N
      VV = FRNF(VV+FRNF(V(I)*V(I)))
      CALL  DLSETF(-ALPHA*V(I))
      DO 1  J=1,N
    1 W =  DLADDF(A(I,J)*V(J))
      EPS2 = FRNF(EPS2+FRNF(W*W))
      CALL  DLMPF(V(I))
      CALL  DLADDF(Q,Q2)
    2 CALL  DLST⊙(Q,Q2)
      EPS2 = EPS2/VV
      Q = FRNF(Q,Q2)/VV
```

Now there is certainly an eigenvalue λ of A that satisfies

$$(\lambda - \alpha - Q)^2 \leq \epsilon^2 - Q^2 \quad ,$$

and if there are no other eigenvalues of A within $\pm D$, say, of the value $(\alpha+Q)$, then λ satisfies

$$|\lambda-\alpha-Q| \leqslant (\epsilon^2-Q^2)/D \quad ,$$

which makes $(\alpha+Q)$ a much better approximation to λ than was α.

In the more general problem with

$$A\underline{v} \doteq \alpha B\underline{v} \quad ,$$

where B is symmetric and positive definite, we would want

$$\underline{w} = (A-\alpha B)\underline{v}$$

$$Q = \underline{v}^*\underline{w}/\underline{v}^*B\underline{v}$$

(The value of ϵ is much harder to calculate).

```
      Q = 0.0
      Q2 = 0.0
     VBV = 0.0
     VBV2 = 0.0
      DO 3  I=1,N
      CALL  DLSETF(0,0)
      DO 1  J=1,N
    1 CALL  DLADDF(B(I,J)*V(J))
      CALL  DLST@(BV,BV2)
      CALL  DLMPF(-ALPHA)
      DO 2  J=1,N
    2 CALL  DLADDF(A(I,J)*V(J))
      CALL  DLMPF(V(I))
      CALL  DLADDF(Q,Q2)
      CALL  DLST@(Q,Q2)
      CALL  DLSETF(BV,BV2)
      CALL  DLMPF(V(I))
      CALL  DLADDF(VBV,VBV2)
    3 CALL  DLST@(VBV,VBV2)
      Q = FRNF(Q,Q2)/VBV
```

Now $(\alpha+Q)$ is a much better approximation to an eigenvalue than was α.

Example 11 Straight line fit by Least Squares

Given a set of N observations (x_i,y_i), we wish to choose a and b to minimize the sum of squares

$$SS = \sum_1^N (y_i-a-bx_i)^2 \quad .$$

For desk calculators the normal equations are put in the form

$$a\sum 1 + b\sum x_i = \sum y_i$$

$$a\sum x_i + b\sum x_i^2 = \sum x_i y_i \quad ,$$

but this is not a good method for an electronic computer. Nonetheless, here is the corresponding program.

```
      DIMENSION SX(2),SY(2),SXX(2),SXY(2),X(N),Y(N)
      CALL  DLSETF(0,0)
      DO 1  I=1,N
    1 CALL  DLADF(X(I))
      CALL  DLST⊙(SX,SX(2))
      CALL  DLSETF(0,0)
      DO 2  I=1,N
    2 CALL  DLADF(Y(I))
      CALL  DLST⊙(SY,SY(2))
      CALL DLSETF(0,0)
      DO 3  I=1,N
    3 CALL  DLADDF(X(I)*Y(I))
      CALL DLST⊙(SXY,SXY(2))
      CALL  DLSETF(0,0)
      DO 4  I=1,N
    4 CALL  DLADDF(X(I)*X(I))
      CALL  DLST⊙(SXX,SXX(2))
D     S1 = FL⊙ATF(N)
D     B = (S1*SXY-SX*SY)/(S1*SXX-SX*SX)
D     A = (SY-B*SX)/S1
```

A better method by far, and one which avoids the use of any double-precision arithmetic, involves the prior transformation to a new origin at the mean of the (x_i,y_i)'s. This better method can also cope with a weighted sum of squares

$$SS = \sum_1^N w_i(y_i-a-bx_i)^2$$

in which all the weights w_i are positive. Let

$$\bar{x} = \sum w_i x_i / \sum w_i \ , \ \Delta x_i = x_i - \bar{x} \ , \ \bar{y} = \sum w_i y_i / \sum w_i \ , \ \Delta y_i = y_i - \bar{y}$$

Then $SS = \sum_1^N w_i(\Delta y_i - \alpha - b\Delta x_i)^2$ where $\alpha = a+b\bar{x}-\bar{y}$.

The normal equations now take the simple form

$$\alpha \sum w_i = 0 \ , \ b\sum w_i(\Delta x_i)^2 = \sum w_i \Delta x_i \Delta y_i \quad , \qquad 11$$

since $\sum w\Delta x_i = \sum w\Delta y_i = 0$.

```
      XBAR = 0.0
      YBAR = 0.0
      SW = 0.0
      DO 1  I=1,N
      SW = FRNF(SW+W(I))
      YBAR = FRNF(YBAR+FRNF(W(I)*Y(I)))
    1 XBAR = FRNF(XBAR+FRNF(W(I)*X(I)))
      XBAR = XBAR/SW
      YBAR = YBAR/SW
      SWXX = 0.0
      SWXY = 0.0
      DO 2  I=1,N
      DX = X(I)-XBAR
      DY = Y(I)-YBAR
      SWXX = FRNF(SWXX+FRNF(W(I)*FRNF(DX*DX)))
    2 SWXY = FRNF(SWXY+FRNF(W(I)*FRNF(DX*DY)))
      B = SWXY/SWXX
      A = YBAR-B*XBAR
```

In both methods, if the sum of squares SS is very small, it might be calculated with the aid of DLAD F etc. Thus:

```
      SS = 0.0
      DO 5  I=1,N
      CALL  DLSETF(A-Y(I))
    5 SS = FRNF(SS+FRNF(W(I)*DLADDF(B*X(I))**2))
```

TIMING: The first column of numbers below gives the average time in 7090 cycles (2.17 μsec. each) to execute the corresponding reference to DLAD F etc. The time spent in the calling sequence in the user's program is included. The second column gives the number of cycles that would be spent to perform a corresponding single-precision operation, and the third column does the same for FORTRAN II double-precision.

Cycles for	Dlad F etc.	S-P	D-P
DLADF(X)	38½	10½	55
DLMPF(X)	62½	15	101
DLMPF(0)	44½	6	—
DLDVF(X)	64	17	117
DLADDF(X,Y)	50	10½	55
DLADDF(0,0)	17	10½	—
DLADDF(X+Y)	54½	17	94
DLADDF(X−Y)	59	21½	148
DLSETF(X,Y)	18	4	16
DLSETF(X+Y)	22½	10½	55
DLSETF(X−Y)	27	15	101
DLST@(A,B)	17	4	16

DISCLAIMER

Although each program has been tested by its contributor, no warranty, express or implied, is made by the contributor or SHARE, as to the accuracy and functioning of the program and related program material, nor shall the fact of distribution constitute any such warranty, and no responsibility is assumed by the contributor or SHARE, in connection therewith.

LS-1/2 TY-DLAF

```
*       FAP     DLAD F, DLMP F, DLDV F, DLADD F, DLSET F, DLSTO                 DLADF   0
        COUNT   66                                                              DLADF   1
        ENTRY   DLAD            DLADF(X)        DLA=DLA+X                       DLADF   2
        ENTRY   DLMP            DLMPF(X)        DLA=DLA*X                       DLADF   3
        ENTRY   DLDV            DLDVF(X)        DLA=DLA/X                       DLADF   4
        ENTRY   DLADD           DLADDF(X..Y)    DLA=DLA+(X..Y)                  DLADF   5
        ENTRY   DLSET           DLSETF(X..Y)    DLA=(X..Y)                      DLADF   6
        ENTRY   DLSTO           DLSTO(A1,A2)    (A1,A2)=DLA                     DLADF   7
DLAD    FAD     DLA             (AC,MQ)=DLA1+X                                  DLADF   8
STO     STO     DLA                                                             DLADF   9
        XCA                                                                     DLADF  10
        TRA     UFA                                                             DLADF  11
DLMP    XCA                                                                     DLADF  12
        STQ     TEMP                                                            DLADF  13
        FMP     DLA2                                                            DLADF  14
        STO     DLA2            =X*DLA2                                         DLADF  15
        LDQ     TEMP                                                            DLADF  16
        FMP     DLA             (AC,MQ)=X*DLA1                                  DLADF  17
        TRA     STO                                                             DLADF  18
DLDV    TZE     OVFLO                                                           DLADF  19
        STO     TEMP            =X                                              DLADF  20
        CLA     DLA                                                             DLADF  21
        FDH     TEMP                                                            DLADF  22
        STQ     DLA             =(DLA1/X)1                                      DLADF  23
        UFA     DLA2                                                            DLADF  24
        FDH     TEMP                                                            DLADF  25
        XCA                                                                     DLADF  26
        TRA     FAD                                                             DLADF  27
DLADD   TZE     ZERO                                                            DLADF  28
        LRS     0                                                               DLADF  29
        STQ     TEMP                                                            DLADF  30
        FAD     DLA                                                             DLADF  31
        STO     DLA                                                             DLADF  32
        XCA                                                                     DLADF  33
        UFA     TEMP                                                            DLADF  34
UFA     UFA     DLA2                                                            DLADF  35
FAD     FAD     DLA             (AC,MQ)=NEW DLA                                 DLADF  36
DLAST   STO     DLA             STORE NEW DLA                                   DLADF  37
        STQ     DLA2                                                            DLADF  38
        FRN                                                                     DLADF  39
        TRA     1,4                                                             DLADF  40
DLSET   LRS     0                                                               DLADF  41
        TRA     DLAST                                                           DLADF  42
DLSTO   CLA     DLA                                                             DLADF  43
        LDQ     DLA2                                                            DLADF  44
        STO*    1,4                                                             DLADF  45
        STQ*    2,4                                                             DLADF  46
        TRA     3,4                                                             DLADF  47
ZERO    CLA     DLA                                                             DLADF  48
        LDQ     DLA2                                                            DLADF  49
        FRN                                                                     DLADF  50
        TRA     1,4                                                             DLADF  51
OVFLO   XCA                                                                     DLADF  52
        CLA     DLA                                                             DLADF  53
        TQP     *+2                                                             DLADF  54
        CHS                                                                     DLADF  55
        ORA     INFY                                                            DLADF  56
        STO     OVIND                                                           DLADF  57
        TRA     DLAST                                                           DLADF  58
INFY    OCT     377777777777                                                    DLADF  59
DLA2    PZE     0                                                               DLADF  60
```

25-2/2 TY-DLAF

```
DLA     PZE     0               54 LOCATIONS + 77777            DLADF 61
        COMMON  -1                                              DLADF 62
OVIND   COMMON  1                                               DLADF 63
        COMMON  -206                                            DLADF 64
TEMP    COMMON  1               V. KAHAN    SEPT.  1963         DLADF 65
        END                                                     DLADF 66
```

DISCLAIMER

This program and its documentation have been contributed to the Program Information Department by an IBM employee and are provided by the IBM Corporation as part of its service to customers. The program and its documentation are essentially in the author's original form and have not been subjected to any formal testing. IBM makes no warranty expressed or implied as to the documentation, function, or performance of this program and the user of the program is expected to make the final evaluation as to the usefulness of the program in his own environment. There is no committed maintenance for the program.

Questions concerning the use of the program should be directed to the author or other designated party. Any changes to the program will be announced in the appropriate Catalog of Programs; however, the changes will not be distributed automatically to users. When such an announcement occurs, users should order only the material (documentation, machine readable or both) as indicated in the appropriate Catalog of Programs.

MACRO ASSEMBLER

For

S/360 MODEL 44

By

MARVIN ROSS

TOM PETERSON

ART REIFF

JAMES WIRTH

September 5, 1967

Direct inquires to:

Tom Peterson

IBM Corporation
2930 West Imperial Highway
Inglewood, California 90303

5194

TABLE OF CONTENTS

MAGNETIC TAPE KEY

This volume contains one tape mark at the end of the material. It was created using the UPDATE feature of the 44/BPS Assembler (see IBM System/360 Model 44 Programming System: Assembler Language, Form C28-6811 and Guide to System Use, Form C28-6812) Density - 800 BPI, 9 track recording, Blocked 2000 bytes per block.

Basic Material

First Assembly -	Pass 1 of assembler
	Identification Col. 73-80 AS100000
	7634 records - 80 characters per record
Second Assembly -	Pass 2 of assembler
	Identification col. 73-80 AS200000
	5040 records - 80 characters per record
Third Assembly -	Decoder
	Identification col. 73-80 DEC10000
	1513 records - 80 characters per record
Fourth Assembly -	Encoder
	Identification col. 73-80 ENC10000
	1440 records - 80 characters per record
Fifth Assembly -	Library reader
	Identification col. 73-80 LIBR1000
	253 records - 80 characters per record
Sixth Assembly -	Macro library utility
	Identification col. 73-80 MCU10000
	787 records - 80 characters per record

ABSTRACT

The macro assembler for 360/44 is designed to provide macro capabilities for the model 44 operating system. The macro package is written in assembler language. It is composed of six programs; 1) Assembler phase one, 2) Assembler phase two, 3) Decoder, 4) Encoder, 5) Macro library routine and 6) Library read routine. Assembler phase one is the main routine. It calls and controls all other routines. The macro language is a subset of OS/360 macro language. This package also has library capabilities for system and user macro libraries.

CONDITIONAL ASSEMBLY AND MACRO FACILITIES IN THE 44PS ASSEMBLER LANGUAGE

SECTION 1: INTRODUCTION TO THE MACRO FACILITIES

The basic macro definition language is an extension of the System/360 Model 44 assembler language. The language described in this publication can be used to facilitate the writing of an assembler language program.

Conditional assembly allows one to specify assembler language statements which may or may not be assembled, depending upon conditions evaluated at assembly time. Conditional assembly statements are used to define, set, change, and test values during the course of the assembly itself.

The conditional assembly instructions may be used to vary the sequence of statements generated for each occurrence of a macro-instruction. Conditional assembly instructions may also be used outside macro-definitions, i.e., among the assembler language statements in the program.

The macro facilities provide the programmer with a convenient way to write a macro-definition that can be used to generate a desired sequence of machine instructions and certain assembler instructions many times in one or more programs.

This macro-definition is written only once, and a single statement, a macro-instruction statement, is written each time a programmer wants to generate the desired sequence of statements.

This facility simplifies the coding of programs, reduces the chance of programming errors, and ensures that standard sequences of statements are used to accomplish desired functions.

THE MACRO-INSTRUCTION STATEMENT

A macro-instruction statement (also called a macro-instruction) is a source program statement used to provide information for generating machine and assembler instructions from a macro-definition. The generated instructions are source statements which are then processed by the assembler program.

THE MACRO-DEFINITION

Before a macro-instruction can be assembled, a macro-definition must be available to the assembler.

A macro-definition is a set of statements that provide the assembler with:

1. The name entry, mnemonic operation code, and the form of the macro-instruction operand, and
2. The sequence of statements the assembler uses when the macro-instruction appears in the source program.

Every macro-definition consists of a macro-definition header statement, a macro-instruction prototype statement, a sequence of model statements, MEXIT, MNOTE, or conditional assembly instructions, and a macro-definition trailer statement.

The macro-definition header and trailer statements denote the beginning and end, respectively, of a macro-definition.

The macro-instruction prototype statement specifies the name entry, mnemonic operation code, and the macro-instruction operand.

The model statements contained in a macro-definition may be used by the assembler to generate machine instructions and certain assembler instructions that replace each occurrence of the macro-instruction.

The MEXIT instruction can be used to terminate processing of a macro-definition.

The MNOTE instruction can be used to generate a message.

The conditional assembly instructions may be used to vary the sequence of statements generated for each occurrence of a macro instruction. Conditional assembly instructions may also be used outside macro-definitions, i.e., among the assembler language statements in the program.

THE MACRO LIBRARY

The same macro-definition may be made available to more than one source program by placing the macro-definitions that can be used by all the assembler language programs in an installation. Once a macro-definition has been placed in the library it may be used by writing a corresponding macro-instruction in a source program. The procedure for placing macro-definitions in the library is described in section 9.

VARYING THE GENERATED STATEMENTS

Each time a macro instruction appears in the source program, it is replaced by the same sequence of assembler language statements. Conditional assembly instructions, however, may be used to vary the number and format of the generated statements.

VARIABLE SYMBOLS

A variable symbol is a type of symbol that is assigned various values by either the programmer or the assembler. Thus, variable symbols allow different values to be assigned to one symbol. When the assembler uses

a macro-definition to determine what statements are to replace a macro-instruction, variable symbols in the model statements are replaced with the current values assigned to them.

A variable symbol is written as an ampersand followed by from one to seven letters and/or digits, the first of which must be a letter.

Types of Variable Symbols

There are three types of variable symbols: symbolic parameters, system variable symbol, and SET symbols. The SET symbols are further broken down into SETA symbols, SETB symbols, and SETC symbols. The three types of variable symbols differ in how they are assigned values.

Assigning Values to Variable Symbols

Symbolic parameters are assigned values by the programmer each time he writes a macro-instruction.

The system variable symbol, &SYSNDX, is assigned a value by the assembler each time it processes a macro-instruction.

SET symbols are assigned values by the programmer by means of conditional assembly instructions.

SET Symbols

The values assigned to SET symbols in one macro-definition may be used in other macro-definitions.

Symbolic parameters and the system variable, &SYSNDX, symbol are local in nature in that they are only referenced within a macro definition.

SECTION 2: HOW TO PREPARE MACRO-DEFINITIONS

A macro-definition consists of:

1. A macro-definition header statement.
2. A macro-instruction prototype statement.
3. Zero or more model statements, MEXIT, MNOTE, or conditional assembly instructions.
4. A macro-definition trailer statement.

Except for MEXIT, MNOTE, and conditional assembly instructions, this section of the publication describes the statements that may be used to prepare macro-definitions. Conditional assembly instructions are described in Section 4. MEXIT and MNOTE instructions are described in Section 5.

Macro-definitions in a source program must appear before all other statements in the main program (except MACLIB cards). No instruction may appear between macro-definitions if there is more than one definition in the source program.

The ICTL instruction cannot be used to alter the normal format of the macro component statements. In writing a macro definition, the begin column is column 1; the end column is column 71; and the continue column is column 16. Continuation of one card to the next is indicated by a non-blank character in column 72. Continuation cards may only be used in macro prorotype statements and source macro-instructions.

MACRO -- MACRO-DEFINITION HEADER

The macro-definition header statement denotes the beginning of a macro-definition. It must be the first statement in every macro-definition.

The form of this statement is:

Name	Operation	Operand
Not used, must not be present	MACRO	Not used, must not be present

MEND -- MACRO-DEFINITION TRAILER

The macro-definition trailer statement denotes the end of a macro-definition. It must be the last statement in every macro-definition. The form of this statement is:

Name	Operation	Operand
Sequence symbol or blank	MEND	Not used, must not be present

MACRO-INSTRUCTION PROTOTYPE

The macro-instruction prototype statement (also called the prototype statement) specifies the name entry, mnemonic operation code, and the form of all macro-instructions that refer to the macro-definition. It must be the second statement of every macro-definition. The typical form of this statement is:

Name	Operation	Operand
A symbolic parameter or not used	A symbol	Zero to 49 symbolic parameters, separated by commas

The symbolic parameters are used in the macro-definition to represent the name entry and operands of the corresponding macro-instruction. A

description of symbolic parameters appears following Model Statements.

The name entry of the prototype statement may be unused or it may contain a symbolic parameter.

The symbol in the operation entry is the mnemonic operation code that must appear in all macro-instructions that refer to this macro-definition. The mnemonic operation code may be the same as the mnemonic operation code of another macro-definition in the source program or of a machine instruction or assembler instruction. In this case the last definition will prevail.

The operand entry may contain zero to 49 symbolic parameters separated by commas.

The following is a prototype statement.

Name	Operation	Operand
&NAME	MOVE	&TO,&FROM

PROTOTYPE STATEMENT FORMAT

The macro instruction prototype statement can be written in a format similar to the format used for other assembler language statements. To allow for the inclusion of up to 49 symbolic parameters in the prototype statement of a macro definition, as many continuation cards as required may be used. As in the normal assembler language statement format, the name field, if used, must begin in column 1, and the operation field followed by at least one blank must appear on the first card of the statement. The other rules are:

1. If the symbolic parameters in the operand field extend up to the end column, column 71, and if column 72 contains a nonblank character, the symbolic parameters can be continued in column 16 of the next card. A single symbolic parameter can

be split between two cards.

2. A blank following a symbolic parameter signifies the end of all symbolic parameters.

3. Comments can appear after the blank that indicates the end of all symbolic parameters, up to and including column 71. If column 72 contains a nonblank character, the comment can be continued in one additional card, beginning in column 16, or any column thereafter.

MODEL STATEMENTS

Model statements are the macro-definition statements from which the desired sequences of machine instructions and certain assembler instructions are generated. Zero or more model statements may follow the prototype statement. A model statement consists of one to four entries. They are, from left to right, the name, operation, operand, and comments entries.

The name entry may be unused, or it may contain an ordinary symbol, a sequence symbol or a variable symbol, depending on the particular statement. (.* may not be substituted in the begin column of a model statement.)

The operation entry may contain any machine, assembler, or macro instruction mnemonic operation code, or it may contain a variable symbol. Variable symbols may not be used to generate START, MACRO, MEND, MEXIT, SETA, SETB, SETC, AIF, AGO, ANOP, or macro-instruction mnemonic operation codes.

Variable symbols may not be used outside of macro-definitions to generate mnemonic operation codes.

The operand entry may contain ordinary symbols or variable symbols. After substitution, the operand must not extend beyond column 71, and model

statement fields must follow the rules for paired apostrophes, ampersands, and blanks. Sequence symbols must appear in the operand entry of AGO and AIF instructions.

The comments entry may contain any combination of characters. Substitution by the use of variable symbols is not allowed.

The comment field is condensed so that all extraneous blanks are omitted.

SYMBOLIC PARAMETERS

A symbolic parameter is a type of variable symbol consisting of an ampersand followed by one to seven letters and/or numbers, the first of which must be a letter. Symbolic parameters appear in prototype and model statements. They are assigned values by the programmer when he writes a macro-instruction. The programmer may vary statements that are generated for each occurrence of a macro-instruction by varying the values assigned to symbolic parameters.

The following are valid symbolic parameters:

&READER	&LOOP2
&A23456	&N
&X4F2	&S4

The following are invalid symbolic parameters:

CARDAREA	(first character is not an ampersand)
&256B	(first character after ampersand is not a letter)
&AREA2456	(more than seven characters after the ampersand)
&BCD(34)	(contains a special character other than initial ampersand)
&IN AREA	(contains a special character i.e., blank, other than initial ampersand)

The following is an example of a macro-definition. Note that the symbolic parameters in the model statements appear in the prototype statement.

	Name	Operation	Operand
Header		MACRO	&TO,
Prototype	&NAME	MOVE	&TO, &FROM
Model	&NAME	ST	2,SAVE
Model		L	2,&FROM
Model		ST	2,&TO
Model		L	2,SAVE
Trailer		Mend	

Symbolic parameters in model statements are replaced by the characters of the macro-instruction operand that correspond to the symbolic parameters.

In the following example the characters HERE, FIELDA, and FIELDB of the MOVE macro-instruction correspond to the symbolic parameters &NAME, &TO, and &FROM, respectively, of the MOVE prototype statement.

Name	Operation	Operand
HERE	MOVE	FIELDA,FIELDB

Any occurrence of the symbolic parameters &NAME, &TO, and &FROM in a model statement will be replaced by the characters HERE, FIELDA, and FIELDB, respectively. If the preceding macro-instruction was used in a source program, the following assembler language statements would be generated:

Name	Operation	Operand
HERE	ST	2,SAVE
	L	2,FIELDB
	ST	2,FIELDA
	L	2,SAVE

The example below illustrates another use of the MOVE macro-instruction using different operands than those that appear in the preceding example.

	Name	Operation	Operand
Macro	LABEL	MOVE	IN,OUT
Generated	LABEL	ST	2,SAVE
Generated		L	2,OUT
Generated		ST	2,IN
Generated		L	2,SAVE

If a symbolic parameter appears in the comments field of a model statement, it is not replaced by the corresponding characters of the macro-instruction.

Concatenating Symbolic Parameters with Other Characters or Other Symbolic Parameters

Concatenation is the process of linking or joining together in a sequence, with a specified order. To concatenate is to join together in a specified order.

If a symbolic parameter in a model statement is immediately preceded or followed by other characters or another symbolic parameter, the characters that correspond to the symbolic parameter are combined, in the order given, in the generated statement, with the other characters or the characters that correspond to the other symbolic parameter. This process is called concatenation.

The macro-definition, macro-instruction, and generated statements in the following example illustrate these rules.

	Name	Operation	Operand
Header		MACRO	
Prototype	&NAME	MOVE	&TY,&P,&TO,&FROM
Model	&NAME	ST&TY	2,SAVEAREA
Model		L&TY	2,&P&FROM

Model		ST&TY	2,&P&TO
Model		L&TY	2,SAVEAREA
Trailer		MEND	
Macro	HERE	MOVE	D,FIELD,A,B
Generated	HERE	STD	2,SAVEAREA
Generated		LD	2,FIELDB
Generated		STD	2,FIELDA
Generated		LD	2,SAVEAREA

The symbolic parameter &TY is used in each of the four model statements to vary the mnemonic operation code of each of the generated statements. The character D in the macro-instruction corresponds to symbolic parameter &TY. Since &TY is preceded by other characters (i.e., ST and L) in the model statements, the character that corresponds to &TY (i.e., D) is concatenated with the other characters to form the operation fields of the generated statements.

The symbolic parameters &P, &TO, and &FROM are used in two of the model statements to vary part of the operand fields of the corresponding generated statements. The characters FIELD, A, and B correspond to the symbolic parameters &P, &TO, and &FROM, respectively. Since &P is followed by &FROM in the second model statement, the characters that correspond to them (i.e., FIELD and B) are concantenated to form part of the operand field of the second generated statement. Similarly, FIELD and A are concatenated to form part of the operand field of the third generated statement.

If the programmer wishes to concatenate a symbolic parameter with a letter, digit, or period following the symbolic parameter he must immediately follow the symbolic parameter with a period. A period is optional if the symbolic parameter is to be concatenated with another symbolic parameter, variable symbol, or a special character other than a period.

If a symbolic parameter is immediately followed by a period, then the symbolic parameter and the period are replaced by the characters that

correspond to the symbolic parameter. A period that immediately follows a symbolic parameter does not appear in the generated statement.

The following macro-definition, macro-instruction, and generated statements illustrate these rules.

	Name	Operation	Operand
Header		MACRO	
Prototype	&NAME	MOVE	&P,&S,&R1,&R2
Model	&NAME	ST	&R1,&S.(&R2)
Model		L	&R1,&P.B
Model		ST	&R1,&P.A
Model		L	&R1,&S.(&R2)
Trailer		MEND	
Macro	HERE	MOVE	FIELD,SAVE,2,4
Generated	HERE	ST	2,SAVE(4)
Generated		L	2,FIELDB
Generated		ST	2,FIELDA
Generated		L	2,SAVE(4)

The symbolic parameter &P is used in the second and third model statements to vary part of the operand field of each of the corresponding generated statements. The characters FIELD of the macro-instruction correspond to &P. Since &P is to be concatenated with a letter (i.e., B and A) in each of the statements, a period immediately follows &P in each of the model statements. The period does not appear in the generated statements.

Similarly, symbolic parameter &S is used in the first and fourth model statements to vary the operand fields of the corresponding generated statements. &S is followed by a period in each of the model statements, this is optional because it is to be concatenated with a left parenthesis, a special character. The period does not appear in the generated statements.

Comments Statements

A model statement may be a comments statement. A comments statement consists of an asterisk in the begin column, followed by comments. The comments statement is used by the assembler to generate an assembler language comments statement, just as other model statements are used by the assembler to generate assembler language statements.

The programmer may also write comments statements in a macro-definition which are not to be generated. These statements must have a period in the begin column, immediately followed by an asterisk and the comments.

The first statement in the following example will be used by the assembler to generate a comments statement; the second statement will not.

Name	Operation	Operand
* This statement will be generated		
.* This one will not be generated		

The use of variable symbols for substitution in comments statements is not allowed. The .* of a comment statement, cannot be created by substitution for a variable symbol.

SECTION 3: HOW TO WRITE MACRO-INSTRUCTIONS

The typical form of a macro-instruction is:

Name	Operation	Operand
A symbol sequence symbol, or not used	Mnemonic Operation Code	Zero to 49 operands, separated by commas.

The name entry of the macro-instruction may contain a symbol. The symbol will not be defined in the generation process unless a symbolic parameter appears in the name entry of the prototype and the same parameter appears in the name entry of a generated model statement.

The operation entry contains the mnemonic operation code of the macro-instruction. The mnemonic operation code must be the same as the mnemonic operation code of a macro-definition in the source program or in the macro library.

The macro-definition with the same mnemonic operation code is used by the assembler to process the macro-instruction. If a macro-definition in the source program and one in the macro library have the same mnemonic operation code, the macro-definition in the source program is used.

The placement and order of the operands in the macro-instruction may be determined by the placement and order of the symbolic parameters in the operand entry of the prototype statement.

MACRO-INSTRUCTION OPERANDS

Any combination of up to 255 characters may be used as a macro-instruction

operand provided that the following rules concerning apostrophes, commas, and blanks are observed.

Paried Apostrophes:

An operand may contain a sequence of characters which is enclosed within single apostrophes. (The sequence of characters itself may contain pairs of apostrophes). The single apostrophes, which enclose the sequence of characters, are called paired apostrophes. The sequence and its paired apostrophes must be immediately preceded by the letter C which may be preceded by a numeric self defining term.

Note: Apostrophes need not be paired unless the sequence of characters is immediately preceded by the letter C or nC where n is a numeric self defining term.

Commas:

A comma indicates the end of an operand, unless it is placed between paired apostrophes preceded by the character C. The following examples illustrate this rule.

```
C'AB,C'
4C'AB,C'
```

Blanks:

A blank indicates the end of the operand entry, unless it is placed between paired apostrophes which are preceded by the letter C. The following examples illustrate this rule.

```
C'A B C'
9C'A B'
```

The following are valid macro-instruction operands:

SYMBOL	A+2
123	XYZ3
X'189A'	ML)O(N
*	=F'4096'

5A)B	AB&&9
C'TEN = 10'	C'PARENTHESIS IS)'
C'COMMA IS,'	2C'APOSTROPHE IS '''

The following are invalid macro-instruction operands:

(15 B)	(blank not placed between paired apostrophes)
'ONE' IS '1'	(blank not placed between paired apostrophes)

An operand is substituted 'as is' wherever called for within model statements. Care should be taken such that the statement generated, after all necessary substitutions and concatenations are made, conforms to the assembler language conventions.

STATEMENT FORM

Macro-instructions must be written using the same form that can be used to write prototype statements. The statement format is described in Section 2 under the subsection "Macro-Instruction Prototype."

OMITTED OPERANDS

If an operand that appears in the prototype statement is omitted from the macro-instruction, then the comma that would have separated it from the next operand must be present. If the last operand(s) is omitted from a macro-instruction, then the comma(s) separating the last operand(s) from the next previous operand may be omitted.

The following example shows a macro-instruction preceded by its corresponding prototype statement. The macro-instruction operands that correspond to the third and sixth operands of the prototype statement are omitted in this example.

Name	Operation	Operand
	EXAMPLE	&A,&B,&C,&D, &E,&F
	EXAMPLE	17,*+4,,AREA, FIELD(6)

If the symbolic parameter that corresponds to an omitted operand is used in a model statement, a null character value (not a blank) replaces the symbolic parameter in the generated statement, i.e., in effect the symbolic parameter is removed.

For example, the first statement below is a model statement that contains the symbolic parameter &C. If the operand that corresponds to &C was omitted from the macro-instruction, the second statement below would be generated from the model statement.

Name	Operation	Operand
	AH	7,THERE&C.25
	AH	7,THERE25

INNER MACRO-INSTRUCTIONS

A macro-instruction may be used as a model statement in a macro-definition. Macro-instructions used as model statements are called inner macro-instructions.

A macro-instruction that is not used as a model statement is referred to as an outer macro-instruction.

Any symbolic parameters used in an inner macro-instruction are replaced by the corresponding operands of the outer macro-instruction.

The macro-definition corresponding to an inner macro-instruction is used to generate the statements that replace the inner macro-instruction.

For example, the following macro definition is given.

Name	Operation	Operand
	MACRO	
	ADD	&N1,&N2,&N3,&REG, &AREA
	L	&REG,&N1
	A	&REG,&N2
	A	&REG,&N3
	ST	&REG,&AREA
	MEND	

The preceding ADD macro is used as an inner macro in the following example.

Name	Operation	Operand
	MACRO	
	COMP	&AREA,&R1,&R2,&V1, &V2,&V3,&NA
	SR	&R1,&R2
	C	&R1,&AREA
	BNE	&NA
	ADD	&V1,&V2,&V3,12,&AREA
&NA	A	&R1,&AREA
	MEND	
	COMP	CHECK,10,11,X,Y,Z, CHNG
	SR	10,11
	C	10,CHECK
	BNE	CHNG
	ADD	X,Y,Z,12,CHECK
	L	12,X
	A	12,Y
	A	12,Z
	ST	12,CHECK
CHNG	A	10,CHECK

Note that the instructions generated in the preceding example by the macro instruction ADD, used as an inner macro in the macro definition COMP, are identical to the instructions that would be generated by the macro ADD is the following macro instruction were given.

Name	Operation	Operand
	ADD	X,Y,Z,12,CHECK
	L	12,X
	A	12,Y
	A	12,Z
	ST	12,CHECK

Note: An ampersand that is part of a symbolic parameter is not considered in determining whether a macro-instruction operand contains an even number of consecutive ampersands.

LEVELS OF MACRO-INSTRUCTIONS

A macro-definition that corresponds to an outer macro-instruction may contain any number of inner macro-instructions. The outer macro-instruction is called a first level macro-instruction. Each of the inner macro-instructions is called a second level macro-instruction.

The macro-definition that corresponds to a second level macro-instruction may contain any number of inner macro-instructions. These macro-instructions are called third level macro-instructions, etc.

The number of levels of macro-instructions that may be used depends upon the complexity of the macro-definition and the amount of storage available.

Note: Macro definitions containing duplicate prototype operation codes as inner macro instructions may result in a loop during processing. For example:

```
MACRO
ZOP     &A,&B,&C
LR      &A,&B
ST      &A,&C
ZOP     &A,&B
SR      &A,&B
ST      &A,&C+4
MEND
```

```
MACRO
ZOP     &A,&B,&C
LR      &A,&B
ST      &A,&C
ZOPQ    &A,&B,&C  (where ZOPQ contains
                   ZOP as inner macro)
MEND
```

The results of such a loop are undertermined. Therefore a macro should not call itself or a macro of which it is an inner macro.

SECTION 4: HOW TO WRITE CONDITIONAL ASSEMBLY INSTRUCTIONS

The conditional assembly instructions allow the programmer to: (1) define and assign values to SET symbols that can be used to vary parts of generated statements, and (2) vary the sequence of generated statements. Thus, the programmer can use these instructions to generate many different sequences of statements from the same macro-definition.

There are six conditional assembly instructions. They are:

SETA	SETC	AGO
SETB	AIF	ANOP

The primary use of the conditional assembly instructions is in macro-definitions. However, all of them may be used in an assembler language source program.

Where the use of an instruction outside macro-definitions, the difference is described in the subsequent text.

The SETA, SETB, and SETC instructions may be used to assign arithmetic, binary, and character values, respectively, to SET symbols. The SETB instruction is described after the SETA and SETC instructions, because the operand of the SETB instruction is a combination of the operands of the SETA and SETC instructions.

The AIF, AGO, and ANOP instructions may be used in conjunction with sequence symbols to vary the sequence in which statements are assembled. The programmer can test attributes assigned by the assembler to symbols or macro-instruction operands to determine which statements are to be processed.

Examples illustrating the use of conditional assembly instructions are included throughout this section. A chart summarizing the elements that can be used in each instruction appears at the end of this section.

SET SYMBOLS

SET symbols are one type of variable symbol. The symbolic parameters discussed in Section 2 are another type of variable symbol. Set symbols differ from symbolic parameters in three ways: (1) where they can be used in an assembler language source program, (2) how they are assigned values, and (3) how the values assigned to them can be changed.

Symbolic parameters can only be used in macro-definitions, whereas SET symbols are assigned values by SETA, SETB, and SETC conditional assembly instructions.

Each symbolic parameter is assigned a single value for one use of a macro-definition, whereas the values assigned to each SETA, SETB, and SETC symbol are not so restricted.

Defining SET Symbols

SET symbols need not be defined by the programmer before they are used, all undefined SET symbols are assigned to value of zero. When a SET symbol is defined it is assigned an initial value. SET symbols may be assigned new values by means of the SETA, SETB, and SETC instructions. A SET symbol can be redefined at any time.

Using Variable Symbols

The SETA, SETB, and SETC instructions may be used to change the values a assigned to previously defined SET symbols. When a SET symbol appears in a statement, the current value of the SET symbol (i.e., the last value

assigned to it) replaces the SET symbol in the statement. However, if the statement is a conditional assembly statement ~~or another SET statement~~ no replacement occurs.

For example:

```
        &A      SETA    1
        &B      SETC    'A'
        &C      SETC    '&B&A'
                L       &A,&C
```

The following will appear on the generated listing.

```
        &A      SETA    1
        &B      SETC    'A'
        &C      SETC    '&A&B'
                L       1,A1
```

If &A is a symbolic parameter, and the corresponding characters of the macro-instruction are the symbol HERE, then HERE replaces each occurrence of &A in the macro-definition. However, if &A is a SET symbol as in the example above, the value assigned to &A can be changed, and a different value can replace various occurrences of &A in the macro difinition.

The same variable symbol may not be used as a symbolic parameter and as a SET symbol in the same macro-definition.

The following illustrates this rule.

Name	Operation	Operand
&NAME	MOVE	&TO,&FROM

If the statement above is prototype statement, then &NAME, &TO, and &FROM may not be used as SET symbols in the macro-definition.

The same variable symbol may be used as different types of SET symbols in the same macro-definition. Similarly, the same variable symbol may be used as different types of SET symbols outside macro-definitions.

For example, if &A is a SETA symbol in a macro-definition, it can be redefined as a SETC symbol in that definition. Similarly, if &A is a SETA symbol outside macro-definitions, it can be redefined as a SETC symbol outside macro-definitions.

SET symbols are treated somewhat like ordinary symbols; that is, they are placed in the symbol table of the assembler. Once defined, they can be referenced throughout the assembly. Therefore care should be exercised in using the same SET symbols inside and outside macro definitions.

For example, if &A is a variable symbol (either SET symbol or symbolic parameter) in one macro-definition, it can be used as a variable symbol (either SET symbol or symbolic parameter) in another definition. Similarly, if &A is a variable symbol (SET symbol or symbolic parameter) in a macro-definition, it can be used as a SET symbol outside macro-definitions. However, unless redefined the previous value is assumed.

All variable symbols may be concatenated with other characters in the same way as symbolic parameters. The rules for concatenation are in Section 2 under the subsection Model Statements.

Variable symbols in macro-instructions are replaced by the character values assigned to them, immediately prior to the start of processing the definition.

Note: SETA and SETB values are converted to character values before substitution takes place.

ATTRIBUTES

The assembler assigns attributes to macro-instruction operands and to symbols in the program. These attributes may be referred to only in conditional assembly instructions.

There are two kinds of attributes. They are: type (T') and length (L')

If an outer macro-instruction operand is a symbol before substitution, then the attributes of the operand are the same as the corresponding attributes of the symbol. The symbol must appear in the name entry of an assembler language statement or in the operand entry of an EXTRN statement in the program.

If an inner macro-instruction operand is a symbolic parameter, then attributes of the operand are the same as the attributes of the corresponding outer macro-instruction operand.

Each attribute has a notation associated with it. The notations are:

Attribute	Notation
Type	T'
Length	L'

All the attributes of macro-instruction operands may be referred to in conditional assembly instructions within macro-definitions or outside definitions. Symbols appearing in the name entry of generated statements are also assigned attributes.

The programmer may refer to an attribute in the following ways:

1. In a statement that is outside macro-definitions, he may write the notation for the attribute immediately followed by a symbol. (E.g., T'NAME refers to the type attribute of the symbol NAME.)

2. In a statement that is in a macro-definition, he may write the notation for the attribute immediately followed by a symbolic parameter. (E.g., L'&NAME refers to the length attribute of the characters in the macro-instruction that correspond to symbolic parameter &NAME).

Type Attribute (T')

The type attribute of a macro-instruction operand or a symbol is a letter.

The programmer may refer to a type attribute in the operand of a SETC instruction, or in character relations in the operands of SETB or AIF instruction, or in other instructions where use of the character is valid.

The following letters are used for symbols that name DC and DS statements and for outer macro-instruction operands that are symbols that name DC or DS statements.

A	A-type address constant, implied length, aligned.
C	Character constant.
D	Long floating-point constant, implied length, aligned.
E	Short floating-point constant, implied length, aligned.
F	Full-word fixed-point constant, implied length, aligned.
H	Half-word fixed-point constant, implied length, aligned.
X	Hexadecimal constant.

The following letters are used for symbols (and outer macro-instruction operands that are symbols) that name statements other than DC or DS statement, or that appear in the operand field of an EXTRN statement.

I	Machine instruction
J	Control section name
T	External symbol
W	CCW assembler instruction

The following letters are used for inner and outer macro-instruction operands only.

N	Self-defining term
O	Omitted operand

N is assigned to all symbolic parameters whose first character is numberic or first 2 characters are, C', F', E', D', H', X'. In the case of C' or NC' where N is a digit, paired apostrophes are expected.

The letter U (Undefined) is used for inner and outer macro-instruction operands that cannot be assigned any of the above letters. The type attribute of all literals appearing as macro-instruction operands is N.

Length Attribute (L')

The length attribute of macro-instruction operands and symbols is a numeric value.

The length attribute of a symbol (or of a macro-instruction operand that is a symbol) is as described below.

TYPE	LENGTH		TYPE	LENGTH
A	4		H	2
C	Length of constant		X	Length of constant
D	8		SETA	3
E	4		SETB	1
F	4		SETC	0 - 4

Reference must not be made to the length attributes of symbols or macro-instruction operands whose type attributes are the letters N, O, or T.

SEQUENCE SYMBOLS

The name entry of a statement may contain a sequence symbol. Sequence symbols provide the programmer with the ability to vary the sequence in which statements are processed by the assembler.

A sequence symbol is used in the operand entry of an AIF or AGO statement to refer to the statement named by the sequence symbol.

A sequence symbol may be used in the name entry of any statement that does not contain a symbol or SET symbol, except a prototype statement, or a MACRO instruction.

A sequence symbol consists of a period followed by one through sevel letters and/or digits, the first of which must be a letter.

The following are valid sequence symbols:

.READER	.A23456
.LOOP2	.X4F2
.N	.S4

The following are invalid sequence symbols:

CARDAREA	(first character is not a period)
.246B	(first character after period is not a letter)
.AREA2456	(more than seven characters after period)
.BCD%84	(contains a special character other than initial period)
.IN AREA	(contains a special character, i.e., blank, other than initial period)

If a sequence symbol appears in the name entry of a macro-instruction, and the corresponding prototype statement contains a symbolic parameter in the name entry, the sequence symbol does not replace the symbolic parameter wherever it is used in the macro-definition.

The following example illustrates this rule.

	Name	Operation	Operand
		MACRO	
1	&NAME	MOVE	&TO,&FROM
2	&NAME	ST	2,SAVEAREA
		L	2,&FROM
		ST	2,&TO
		L	2,SAVEAREA
		MEND	
3	.SYM	MOVE	FIELDA,FIELDB
4		ST	2,SAVEAREA
		L	2,FIELDB
		ST	2,FIELDA
		L	2,SAVEAREA

The symbolic parameter &NAME is used in the name entry of the prototype statement (statement 1) and the first model statement (statement 2). In the macro-instruction (statement 3), a sequence symbol (.SYM) corresponds to the symbolic parameter &NAME. &NAME is not replaced by .SYM, and, therefore, the generated statement (statement 4) does not contain a name entry.

SETA -- SET ARITHMETIC

The SETA instruction may be used to assign an arithmetic value to a SETA symbol. The form of this instruction is:

Name	Operation	Operand
A SETA symbol	SETA	A SETA arithmetic expression

The expression in the operand entry is evaluated as a signed 24-bit arithmetic in the name entry. The minimum and maximum allowable values of the expression are -2^{23} and $+2^{23}-1$, respectively.

The expression may consist of one term or an arithmetic combination of up to three terms. The terms that may be used alone or in combination with each other are self-defining terms, variable symbols, and the length attribute.

Note: A SETC variable symbol may not appear in a SETA expression.

The arithmetic operators that may be used to combine the terms of an expression are + (addition), - (subtraction), *(multiplication), and / (division).

An expression may not contain two terms or two operators in succession, nor may it begin with an operator or exceed three terms.

The following are valid operand fields of SETA instructions:

```
&AREA+X'2D'          L'&HERE+32
&BETA*10             29
```

The following are invalid operand fields of SETA instructions:

```
&AREAX'C'            (two terms in succession)
&FIELD+-             (two operators in succession)
-&DELTA*2            (begins with an operator)
*+32                 (begins with an operator; 2 operators in
                      sucession)
```



```
NAME/15              (NAME is not a valid term)
```

Note: All SETA and SETB variable symbols used in the operand field must have been previously defined.

EVALUATION OF ARITHMETIC EXPRESSIONS

The procedure used to evaluate the arithmetic expression in the operand of a SETA instruction is the same as that used to evaluate arithmetic expressions in assembler language statements. The only difference between the two types of arithmetic expressions is the terms that are allowed in each expression.

The following evaluation procedure is used:

1. Each term is given its numerical value.
2. The arithmetic operations are performed moving from left to right. However, multiplication and/or division are performed before addition and subtraction.
3. The computed result is the value assigned to the SETA symbol in the name entry.

The arithmetic expression in the operand entry of a SETA instruction may contain one or more sequences of arithmetically combined terms that are enclosed in parentheses. A sequence of parenthesized terms may not appear within another parenthesized sequence. One level of parenthesis may be used in a SETA operand.

The following are examples of SETA instruction operands that contain parenthesized sequences of terms.

```
(L'&HERE+32)*29
&EXIT/(&ENTRY-4)
```

The parenthesized portion or portions of an arithmetic expression are evaluated before the rest of the terms in the expression are evaluated.

The SETA arithmetic expression can only have one level of parentheses.

Using SETA Symbols

The arithmetic value assigned to a SETA symbol is substituted for the SETA symbol when it is used in an arithmetic relation. If the SETA symbol is not used in an arithmetic expression, the arithmetic value is completely converted to an unsigned interger, with leading zeros removed. If the value is zero, it is converted to a single zero.

The following example illustrates this rule:

	Name	Operation	Operand
		MACRO	
	&NAME	MOVE	&TO,&FROM
1	&A	SETA	10
2	&B	SETA	12
3	&C	SETA	&A-&B
4	&D	SETA	&A+&C
	&NAME	ST	2,SAVEAREA
5		L	2,&FROM&C
6		ST	2,&TO&D
		L	2,SAVEAREA
		MEND	
	HERE	MOVE	FIELDA,FIELDB
	HERE	ST	2,SAVEAREA
		L	2,FIELDB2
		ST	2,FIELDA8
		L	2,SAVEAREA

Statements 1 and 2 assign to the SETA symbols &A and &B the arithmetic values +10 and +12, respectively. Therefore, statement 3 assigns the SETA symbol &C the arithmetic value -2. When &C is used in statement 5, the arithmetic value -2 is converted to the unsigned integer 2. When &C is used in statement 4, however, the arithmetic value -2 is used. Therefore, &D is assigned the arithmetic value +8. When &D is used in statement 6, the arithmetic value +8 is converted to the unsigned integer 8.

The following example shows how the value assigned to a SETA symbol may be changed in a macro-definition.

	Name	Operation	Operand
		MACRO	
	&NAME	MOVE	&TO&FROM
1	&A	SETA	5
	&NAME	ST	2,SAVEAREA
2		L	2,&FROM&A
3	&A	SETA	8
4		ST	2,&TO&A
		L	2SAVEAREA
		MEND	
	HERE	MOVE	FIELDA,FIELDB
	HERE	ST	2,SAVEAREA
		L	2,FIELDB5
		ST	2,FIELDA8
		L	2,SAVEAREA

Statement 1 assigns the arithmetic value +5 to SETA symbol &A. In statement 2, &A is converted to the unsigned integer 5. Statement 3

assigns the arithmetic value +8 to &A. In statement 4, therefore, &A is converted to the unsigned integer 8, instead of 5.

SETC -- SET CHARACTER

The SETC instruction is used to assign a character value to a SETC symbol. The form of this instruction is:

Name	Operation	Operand
A SETC symbol	SETC	One operand, of the form described below

The operand may consist of the type attribute, a character expression, or a substring notation. A SETA symbol may appear in the operand of a SETC statement. The result is the character representation of the decimal value, unsigned, with leading zeros removed. If the value is zero, one decimal zero is used.

TYPE ATTRIBUTE

The character value assigned to a SETC symbol may be a type attribute. If the type attribute is used, it must appear alone in the operand field. The following example assigns to the SETC symbol &TYPE the letter that is the type attribute of the macro-instruction operand that corresponds to the symbolic parameter &ABC.

Name	Operation	Operand
&TYPE	SETC	T'&ABC

CHARACTER EXPRESSION

A character expression consists of any combination of characters enclosed in apostrophes. The character value enclosed in apostrophes in the operand field is assigned to the SETC symbol in the name entry. The maximum length character value that can be assigned to a SETC symbol is four characters. If a value greater than 4 is specified, the leftmost 4 characters will be used.

EVALUATION OF CHARACTER EXPRESSIONS: The following statement assigns the character value AB%4 to the SETC symbol &ALPHA:

Name	Operation	Operand
&ALPHA	SETC	'AB%4'

Two apostrophes must be used to represent a apostrophe that is part of a character expression.

The following statement assigns the character value L'SY to the SETC symbol &LENGTH.

Name	Operation	Operand
&LENGTH	SETC	'L''SY'

Variable symbols may be concatenated with other characters in the operand field of an SETC instruction according to the general rules for concatenating variable symbols with other characters except a period is not removed inside the quote marks.

If &ALPHA has been assigned the character value AB, the following statement may be used to assign the character value STAB to the variable symbol

&GAMMA.

Name	Operation	Operand
&GAMMA	SETC	'ST&ALPHA'

Two ampersands must be used to represent an ampersand that is not part of a variable symbol. They are replaced by a single ampersand.

The following statement assigns the character value HAL& to the SETC symbol &AND.

Name	Operation	Operand
&AND	SETC	'HAL&&'

SUBSTRING NOTATION

The character value assigned to a SETC symbol may be a substring character value. Substring character values permit the programmer to assign part of a variable symbol to a SETC symbol. This permits the examination of the variable symbol to the character level.

If the programmer wants to assign part of a variable symbol to a SETC symbol, he must indicate to the assembler in the operand of a SETC instruction: (1) the variable symbol itself, and (2) the part of the variable symbol he wants to assign to the SETC symbol. The concatenation of (1) and (2) in the operand of a SETC instruction is called a substring notation.

Substring notation consists of a variable symbol, immediately followed by two arithmetic expressions that are separated from each other by a comma and are enclosed in parentheses. These parentheses do not count when determining the number of levels of parentheses. The two arithmetic expressions may be any expression that is allowed in the operand of a SETA instruction.

The first expression indicates the first character (in the variable symbol) that is to be assigned to the SETC symbol in the name entry. The second expression indicates the number of consecutive characters in the variable symbol (starting with the character indicated by the first expression) that are to be assigned to the SETC symbol. If a substring specifies more characters than are in the variable symbol, the number of available characters will be supplied.

The maximum size variable symbol the substring character value can be chosen from is 255 characters.

The following are valid substring notations:

```
'&ALPHA'(2,4)
'&AB4'(&AREA+2,1)
'&RST'(6,&A)
'&GAMMA'(&A,&AREA-2)
```

The following are invalid substring notations:

'&BETA' (4,4)	(blanks between variable symbol and arithmetic expressions)
'&ALPHA' (8 &FIELD*2)	(arithmetic expressions not separated by a comma)
'&BETA' 4,6	(arithmetic expressions not enclosed in parentheses)
'&ALPHA' (2,4)(1,1)	(double substring notation is not permitted)

Using SETC Symbols

The character value assigned to a SETC symbol is substituted for the SETC symbol when it is used in the name, operation, or operand of a statement.

For example, consider the following macro-definition, macro-instruction, and generated statements.

	Name	Operation	Operand
		MACRO	
	&NAME	MOVE	&TO,&FROM
1	&PREFIX	SETC	'FIEL'
	&NAME	ST	2,SAVEAREA
2		L	2,&PREFIX&FROM
3		ST	2,&PREFIX&TO
		L	2,SAVEAREA
		MEND	
	HERE	MOVE	A,B
	HERE	ST	2,SAVEAREA
		L	2,FIELB
		ST	2,FIELA
		L	2,SAVEAREA

Statement 1 assigns the character value FIEL to the SETC symbol &PREFIX. In statements 2 and 3, &PREFIX is replaced by FIEL.

The following example shows how the value assigned to a SETC symbol may be changed in a macro-definition.

	Name	Operation	Operand
		MACRO	
	&NAME	MOVE	&TO,&FROM
1	&PREFIX	SETC	'FIEL'
	&NAME	ST	2,SAVEAREA
2		L	2,&PREFIX&FROM
3	&PREFIX	SETC	'AREA'
4		ST	2,&PREFIX&TO
		L	2,SAVEAREA
		MEND	
	HERE	MOVE	A,B
	HERE	ST	2,SAVEAREA
		L	2,FIELB
		ST	2,AREAA
		L	2,SAVEAREA

Statement 1 assigns the character value FIEL to the SETC symbol &PREFIX. Therefore, &PREFIX is replaced by FIEL in statement 2. Statement 3 assigns the character value AREA to &PREFIX. Therefore, &PREFIX is replaced by AREA, instead of FIEL, in statement 4.

The following example illustrates the use of a substring notation as the operand field of a SETC instruction.

	Name	Operation	Operand
		MACRO	
	&NAME	MOVE	&TO,&FROM
1	&PREFIX	SETC	'&TO'(1,4)
	&NAME	ST	2,SAVEAREA

2		L	2,&PREFIX&FROM
		ST	2,&TO
		L	2SAVEAREA
		MEND	
	HERE	MOVE	FIELDA,B
	HERE	ST	2,SAVEAREA
		L	2,FIELB
		ST	2,FIELA
		L	2, SAVEAREA

Statement 1 assigns the substring character value FIEL (the first four characters corresponding to symbolic parameter &TO) to the SETC symbol &PREFIX. Therefore, FIEL replaces &PREFIX in statement 2.

SETB -- SET BINARY

The SETB instruction may be used to assign the binary value 0 on 1 to a SETB symbol. The form of this instruction is:

Name	Operation	Operand
A SETB symbol	SETB	A 0 or a 1,(0) or (1), or a logical expression enclosed in parentheses

The operand may contain a 0 or a 1 or a logical expression enclosed in parentheses. (No explicit boolean zeros or ones are allowed in parentheses other than in the form (0) or (1).) A logical expression is evaluated to determine if it is true or false; the SETB symbol in the name entry is then assigned the binary value 1 or 0 corresponding to true or false, respectively.

Note: The parentheses enclosing a logical expression count toward the parenthesis level limit.

A logical expression consists of one term or a logical combination of terms. The terms that may be used alone or in combination with each other are arithmetic relations, character relations, and SETB symbols.

An arithmetic relation consists of two arithmetic expressions connected by a relational operator. A character relation consists of two character strings connected by a relational operator. The relational operators are EQ (equal), NE (not equal), LT (less than), GT (greater than), LE (less than or equal), and GE (greater than or equal).

Any expression that may be used in the operand of a SETA instruction, may be used as an arithmetic expression in the operand of a SETB instruction. Anything that may be used in the operand of a SETC instruction, except substring notation, may be used as a character string in the operand of a SETB instruction. The maximum size of the character values that can be compared is 4 characters. If the two character values are of unequal length, then the shorter one will always compare less than the longer one, regardless of the characters present.

The relational operator must be immediately preceded and followed by at least one blank.

A relation enclosed in parentheses must not be separated from the parentheses by any blanks. If a self defining term is used, they must be decimal, character, or hexadecimal, or e.g., 8, C'A', or X'03', respectively.

The following are valid operand fields of SETB instructions:

```
1
(&AREA+2 GT 29)
(C'AB%4' EQ &ALPHA)
```

(T'&ABC NE T'&XYZ)
(T'&P12 EQ C'F')

The following are invalid operand fields of SETB instructions:

&B	(not enclosed in parentheses)
(T'&P12 EQ C'F' &B)	(two terms in succession)
(T'&P12 EQ 'F')	(must have character modifier C'F')

Evaluation of Logical Expressions

The following procedure is used to evaluate a logical expression in the operand field of a SETB instruction:

1. Each term is evaluated

2. The two terms are compared according to the relational operator.

3. The value of 1 or 0 (true or false) is assigned to the SETB symbol in the name field.

Using SETB Symbols

The logical value assigned to a SETB symbol is used for the SETB symbol appearing in the operand of an AIF instruction or another SET instruction.

If a SETB symbol is used in the operand of a SETA instruction, or in arithmetic relations in the operands of AIF and SETB instructions, the binary values 1 (true) and 0 (false) are converted to the arithmetic values +1 and +0, respectively.

If a SETB symbol is used in the operand of a SETC instruction, in character relations in the operands of AIF and SETB instructions, or in any other statement, the binary values 1 (true) and 0 (false), are converted to the character values 1 and 0, respectively.

The following example illustrates these rules. It is assumed that L'&TO EQ 4 is true, and &T EQ 0 is false.

	Name	Operation	Operand
		MACRO	
	&NAME	MOVE	&TO,&FROM
1	&B1	SETB	(L'&TO EQ 4)
2	&B2	SETB	(L'&T EQ 0)
3	&A1	SETC	'&B1'
4	&C1	SETC	'&B2'
	&NAME	ST	2,SAVEAREA
		L	2,&FROM&A1
		ST	2,&TO&C1
		L	2,SAVEAREA
		MEND	
	HERE	MOVE	FIELDA,FIELDB
	HERE	ST	2,SAVEAREA
		L	2,FIELDB1
		ST	2,FIELDA0
		L	2,SAVEAREA

Because the operand of statement 1 is true, &B1 is assigned the binary value 1. Therefore, the character value +1 is substituted for &B1 in statement 3. Because the operand of statement 2 is false, &B2 is assigned the binary value 0. Therefore the character value 0 is substituted for &B2 in statement 4.

AIF -- CONDITIONAL BRANCH

The AIF instruction is used to alter conditionally the sequence in which source program statements are processed by the assembler. The typical form of this instruction is:

Name	Operation	Operand
A sequence symbol or not used	AIF	A logical expression enclosed in parentheses, immediately followed by a sequence symbol

Any logical expression that may be used in the operand of a SETB instruction may be used in the operand of a SETB instruction may be used in the operand of an AIF instruction. However, the forms

AIF (0), sequence symbol and

AIF (1), sequence symbol

are invalid. The sequence symbol in the operand must immediately follow the closing parenthesis of the logical expression. AIF operand entries must not contain explicit boolean zeros or ones.

Note: The parentheses enclosing the logical expression do not count toward the level limit.

The logical expression in the operand is evaluated to determine if it is true or false. If the expression is true, the statement named by the sequence symbol in the operand is the next statement processed by the assembler. If the expression is false, the next sequential statement is processed by the assembler.

The statement named by the sequence symbol may precede or follow the AIF instruction in macros. However sequence symbols must follow the AIF instruction if used outside of macros.

If an AIF instruction is in a macro-definition, then the sequence symbol in the operand must appear in the name entry of a statement in the definition. If an AIF instruction appears outside macro-definitions, then the sequence symbol in the operand must appear in the name entry of a statement outside macro-definitions.

The following are valid operands of AIF instructions:

```
(&AREA+X'2D' GT 29).READER
(T'&P12 EQ 'F').THERE
```

The following are invalid operands of AIF instructions:

(NAME EQ 4).Q	(NAME is not a self defining term or variable symbol)
(T'&ABC NE T'&XYZ)	(no sequence symbol)
.X4F2	(no logical expression)
(T'&ABC NE T'&XYZ) .X4F2	(blanks between logical expression and sequence symbol)

The following macro-definition may be used to generate the statements needed to move a full-word fixed-point number from one storage area to another. The statements will be generated only if the type attribute of both storage areas is the letter F.

	Name	Operation	Operand
		MACRO	
	&N	MOVE	&T,&F
1		AIF	(T'&T NE T'&F).END

```
2                AIF     (T'&T NE C'F'). END
3       &N       ST      2,SAVEAREA
                 L       2,&F
                 ST      2,&T
                 L       2,SAVEAREA
4       .END     MEND
```

The logical expression in the operand of statement 1 has the value true if the type attributes of the two macro-instruction operands are not equal. If the type attributes are equal, the expression has the logical value false.

Therefore, if the type attributes are not equal, statement 4 (the statement named by the sequence symbol .END) is the next statement processed by the assembler. If the type attributes are equal, statement 2 (the next sequential statement) is processed.

The logical expression in the operand of statement 2 has the value true is the type attribute of the first macro-instruction operand is not the letter F. If the type attribute is the letter F, the expression has the logical value false.

Therefore, if the type attribute is not the letter F, statement 4 (the statement named by the sequence symbol .END) is the next statement processed by the assembler. If the type attribute is the letter F, statement 3 (the next sequential statement) is processed.

Note: C'F' in statement 2 maybe replaced by decimal 198 or X'C6' and the result would be the same.

AGO -- UNCONDITIONAL BRANCH

The AGO instruction is used to unconditionally alter the sequence in which source program statements are processed by the assembler. The typical form of this instruction is:

Name	Operation	Operand
A sequence symbol or not used	AGO	A sequence symbol

The statement named by the sequence symbol in the operand is the next statement processed by the assembler.

The statement named by the sequence symbol may precede (in macros only) or follow the AGO instruction.

If an AGO instruction is part of a macro-definition, then the sequence symbol in the operand must appear in the name entry of a statement that is in that definition. If an AGO instruction appears outside macro-definitions, then the sequence symbol in the operand must appear in the name entry of a statement outside macro-definitions.

The following example illustrates the use of the AGO instruction.

```
         Name      Operation   Operand
                   MACRO
         &NAME     MOVE        &T,&F
1                  AIF         (T'&T EQ C'F').FIRST
2                  AGO         .END
3        .FIRST    AIF         (T'&T NE T'&F).END
         &NAME     ST          2,SAVEAREA
                   L           2,&F
                   ST          2,&T
                   L           2,SAVEAREA
4        .END      MEND
```

Statement 1 is used to determine if the type attribute of the first macro-instruction operand is the letter F. If the type attribute is the letter F, statement 3 is the next statement processed by the assembler. If the type attribute is not the letter F, statement 2 is the next statement processed by the assembler.

Statement 2 is used to indicate to the assembler that the next statement to be processed is statement 4 (the statement named by sequence symbol .END).

ANOP -- ASSEMBLY NO OPERATION

The ANOP instruction facilitates conditional and unconditional branching to statements named by symbols or variable symbols.

The typical form of this instruction is:

Name	Operation	Operand
A sequence symbol	ANOP	Not used, must not be present

If the programmer wants to use an AIF or AGO instruction to branch to another statement, he must place a sequence symbol in the name entry of the statement to which he wants to branch. However, if the programmer has already entered a symbol or variable symbol in the name entry of that statement, he cannot place a sequence symbol in the name entry. Instead, the programmer must place an ANOP instruction before the statement and then branch to the ANOP instruction. This has the same effect as branching to the statement immediately after the ANOP instruction.

The following example illustrates the use of the ANOP instruction.

	Name	Operation	Operand
		MACRO	
	&NAME	MOVE	&T,&F
1		AIF	(T'&T EQ C'F').FTYPE
2	&TYPE	SETC	'E'
3	.FTYPE	ANOP	
4	&NAME	ST&TYPE	2,SAVEAREA
		L&TYPE	2,&F
		ST&TYPE	2,&T
		L&TYPE	2,SAVEAREA
		MEND	

Statement 1 is used to determine if the type attribute of the first macro-instruction operand is the letter F. If the type attribute is not the letter F, statement 2 is the next statement processed by the assembler. If the type attribute is the letter F, statement 4 should be processed next. However, since there is a variable symbol (&NAME) in the name field of statement 4, the required sequence symbol (.FTYPE) cannot be placed in the name field. Therefore, an ANOP instruction (statement 3) must be placed before statement 4.

Then, if the type attribute of the first operand is the letter F, the next statement processed by the assembler is the statement named by sequence symbol .FTYPE. The value of &TYPE remains unchanged because the SETC instruction is not processed. Since .FTYPE names an ANOP instruction, the next statement processed by the assembler is statement 4, the statement following the ANOP instruction.

CONDITIONAL ASSEMBLY ELEMENTS

The following chart summarizes the elements that can be used in each conditional assembly instruction. Each row in this chart indicates which

elements can be used in a single conditional assembly instruction. Each column is used to indicate the conditional assembly instructions in which a particular element can be used.

	Variable Symbols				Attributes	
		SET Symbols				
	S. P.	SETA	SETB	SETC	T'	L'
SETA	O	N,O	N,O	N		O
SETB	O	N,O	N,O	N,O	O^1	O^2
SETC	O	N,O	N,O	N,O	O	
AIF	O	O	O	O	O^1	O^2
AGO						
ANOP						

^{1}Only in character relations
^{2}Only in arithmetic relations

Abbreviations

N is Name

O is Operand

S.P. is Symbolic Parameter

The intersection of a column and a row indicates whether an element can be used in an instruction, and if so, in what fields of the instruction the element can be used. For example, the intersection of the first row and the first column of the chart indicates that symbolic parameters can be used in the operand field of SETA instructions.

SECTION 5: ADDITIONAL FEATURES

The additional features of the assembler language allow the programmer to:

1. Terminate processing of a macro-definition.
2. Generate error messages.
3. Use a system variable symbol.

MEXIT -- MACRO-DEFINITION EXIT

The MEXIT instruction is used to indicate to the assembler that it should terminate processing of a macro-definition. The typical form of this instruction is:

Name	Operation	Operand
A sequence symbol or not used	MEXIT	Not used, must not be present

The MEXIT instruction may only be used in a macro-definition.

If the MEXIT instruction occurs in a macro-definition corresponding to an outer macro-instruction, the next statement processed by the assembler is the next statement outside macro-definitions.

If the MEXIT instruction occurs in a macro-definition corresponding to a second or third level macro-instruction, the next statement processed by the assembler is the next statement after the second or third level macro-instruction in the macro-definition, respectively.

MEXIT should not be confused with MEND. MEND indicates the end of a macro-definition. MEND must be the last statement of every macro-definition, including those that contain one or more MEXIT instructions.

The following example illustrates the use of the MEXIT instruction.

	Name	Operation	Operand
		MACRO	
	&NAME	MOVE	&T,&F
1		AIF	(T'&T EQ C'F').OK
2		MEXIT	
3	.OK	ANOP	
	&NAME	ST	2,SAVEAREA
		L	2,&F
		ST	2,&T
		L	2,SAVEAREA
		MEND	

Statement 1 is used to determine if the type attribute of the first macro-instruction operand is the letter F. If the type attribute is the letter F, the assembler processes the remainder of the macro-definition starting with statement 3. If the type attribute is not the letter F, the next statement processed by the assembler is statement 2. Statement 2 indicates to the assembler that it is to terminate processing of the macro-definition.

MNOTE STATEMENT

The MNOTE instruction may be used to generate a message and to indicate what error severity code, if any, is to be associated with the message. The typical form of this instruction is:

Name	Operation	Operand
A sequence symbol or not used	MNOTE	See examples below.

The operand entry of the MNOTE assembler-instruction may be written in one of the following forms:

1. severity-code, 'message'
2. ,'message'
3. *,'message'

For 2 and 3 above, the severity code is assumed to be one and zero respectively.

The MNOTE instruction may only be used in a macro-definition. Variable symbols may not be used to generate the MNOTE mnemonic operation code, or the severity code indicator.

The resulting severity code indicator may be a decimal integer 0 to 3, blank, or an asterisk. The integers indicate the severity of the error. (0 is the least severe; 3 is the most severe). If the serverity code indicator is blank or omitted, 1 is assumed. If the severity code is an asterisk, the MNOTE is not considered an error message, and the message is considered a comment. Messages can be generated with substitution using variable symbols.

The MNOTE statement appears in the listing with a statement number at the point where it was generated. If the severity code indicator was an integer or a blank, this statement number is placed in a list of statement numbers of MNOTE and other error statements near the end of the assembly listing. If the severity code is an asterisk, the statement number is not placed in this list.

The message portion of the MNOTE operand is enclosed in apostrophes, two apostrophes need not be used to represent a single apostrophe. Any variable symbols used in the message operand are replaced by values assigned to them. Two ampersands need not be used to represe.. a single ampersand

that is not part of a variable symbol.

The following example illustrates the use of the MNOTE instruction.

	Name	Operation	Operand
		MACRO	
	&NAME	MOVE	&T,&F
1		AIF	(T'&T NE T'&F).M1
2		AIF	(T'&T NE C'F').M2
3	&NAME	ST	2,SAVEAREA
		L	2,&F
		ST	2,&T
		L	2,SAVEAREA
4		MNOTE	*,'MOVE GENERATED'
		MEXIT	
5	.M1	MNOTE	3,'TYPE NOT SAME'
		MEXIT	
6	.M2	MNOTE	3,'TYPE NOT F'
		MEND	

Statement 1 is used to determine if the type attributes of both macro-instruction operands are the same. If they are, statement 2 is the next statement processed by the assembler. If they are not, statement 5 is the next statement processed by the assembler. Statement 5 causes an error message -- 3,TYPE NOT SAME -- to be printed in the source program listing.

Statement 2 is used to determine if the type attribute of the first macro-instruction operand is the letter F. If the type attribute is the letter F, statement 3 is the next statement processed by the assembler. If the attribute is not the letter F, statement 6 is the next statement processed by the assembler. Statement 6 causes an error message -- 3, TYPE NOT F -- to be printed in the source program listing. Statement 4 is an MNOTE which

is not treated as an error message.

SYSTEM VARIABLE SYMBOL

The system variable symbol is assigned a value automatically by the assembler. There is one system variable symbol: &SYSNDX. The system variable symbol may be used in the name, operation and operand entries of statements in macro-definitions, but not in statements outside macro-definitions. It may not be defined as symbolic parameters or SET symbols, nor may it be assigned values by SETA, SETB, and SETC instructions.

&SYSNDX --Macro-Instruction Index

The system variable symbol &SYSNDX may be combined with other characters to create unique names for statements generated from the same model statement.

&SYSNDX is assigned the four-digit number 0001 for the first macro-instruction processed by the assembler, and it is incremented by one for each subsequent inner and outer macro-instruction processed.

If &SYSNDX is used in a model statement, SETC or MNOTE instruction, or a character relation in a SETB or AIF instruction, the value substituted for &SYSNDX is the four-digit number of the macro-instruction being processed, including leading zeros.

If &SYSNDX appears in arithmetic expressions (e.g., in the operand of a SETA instruction), the value used for &SYSNDX is an arithmetic value.

Throughout one use of a macro definition, the value of &SYSNDX may be considered a constant, independent of any inner macro-instruction in that definition.

The example following illustrates these rules. It is assumed that the first macro-instruction processed, OUTER1, is the 106th macro-instruction processed by the assembler.

Statement 7 is the 106th macro-instruction processed. Therefore, &SYSNDX is assigned the number 0106 for that macro-instruction. The number 0106 is substituted for &SYSNDX when it is used in statements 4 and 6. Statement 4 is used to assign the character value 0106 to the SETC symbol &NDXNUM. Statement 6 is used to create the unique name B0106.

	Name	Operation	Operand
		MACRO	
		INNER1	
1	A&SYSNDX	SR	2,5
		CR	2,5
2		BE	B&NDXNUM
3		B	A&SYSNDX
		MEND	
		MACRO	
	&NAME	OUTER1	
4	&NDXNUM	SETC	'&SYSNDX'
	&NAME	SR	2,4
		AR	2,6
5		INNER1	
6	B&SYSNDX	S	2,=F'1000'
		MEND	
7	ALPHA	OUTER1	
8	BETA	OUTER1	
	ALPHA	SR	2,4
		AR	2,6
A0107		SR	2,5
		CR	2,5
		BE	B0106
		B	A0107
B0106		S	2,=F'1000'
BETA		SR	2,4
		AR	2,6
A0109		SR	2,5
		CR	2,5
		BE	B0108
		B	A0109
B0108		S	2,=F'1000'

Statement 5 is the 107th macro-instruction processed. Therefore, &SYSNDX is assigned the number 0107 for that macro-instruction. The number 0107 is substituted for &SYSNDX when it is used in statements 1 and 3. The number 0106 is substituted for the SETC symbol &NDXNUM in statement 2.

Statement 8 is the 108th macro-instruction processed. Therefore, each occurrence of &SYSNDX is replaced by the number 0108. For example, statement 6 is used to create the unique name B0108.

When statement 5 is used to process the 108th macro-instruction, statement 5 becomes the 109th macro-instruction processed. Therefore, each occurrence of &SYSNDX is replaced by the number 0109. For example, statement 1 is used to create the unique name A0109.

THE PARM OPTION

PARMn is an additional parameter to the EXEC statement. It is similar to the SYMBn parameter in that n is a decimal number not to exceed 9999 (the value for n must be greater than 128). The programmer estimates the amount of space required by the DECODER for its parameter table, using the formula below, and inserts the results for n. The system reserves n time two bytes for the table.

$$n = RP + M + \frac{\Sigma C}{2}$$

P = Maximum number of parameter in the largest macro (include all inner macro parameters)

M= Total number of macros used by one call, i.e. the sum of inner macros.

ΣC = Total number of character used in the parameters.

For example:

If macro A has 6 parameters (including the name) and each parameter is 6 characters long and calls a macro B with 5 parameters 10 characters long then n = 67.

Solution:

$$n = 2P + M + \frac{\Sigma C}{2}$$

P = 6 + 5 = 11

M = 2 (The macro A and the inner macro B)

C = 6(6) + 5(10) = 86

$$n = R(11) + R + \frac{86}{2} = 67$$

Note: Compute n for the maximum number of parameter for a single macro call not the sum of all macro calls.

SECTION 6: COMPATABILITY WITH BOS ASSEMBLER

The IBM System/360 Model 44 Macro Assembler Language is a selected subset of the IBM System/360 Basic Operating Systems. The following chart summarizes the major features which are not compatible between the two systems.

FEATURE	BOS LANGUAGE	44PS LANGUAGE
Macro Instruction Statement	1. Positional 2. Keyword 3. Mixed Mode	1. Positional
Macro Library	Library of source statements	Library in encoded format.
System variable symbols	1. &SYSNDX 2. &SYSECT 3. &SYSLIST	1. &SYSNDX
Variable symbol definition	1. Global 2. Local	Need not define prior to use, all are considered global.
Maximum number of symbolic parameters	100	49
Names of Macros used in an assembly	All names must be unique.	Name need not be unique, the latest definition will prevail.
Use of variable symbols.	May generate mnemonic operation codes outside of Macros	May not be used to generate mnemonic operation codes outside of Macros
Macro instruction operands	Up to 127 characters in the operand	Up to 255 characters in the operand
Sublists	Supported	Not supported

Inner Macros		A macro may not call itself or an outer macro.
Attributes	1. Type 2. Length 3. Scaling 4. Integer 5. Count 6. Number	1. Type 2. Length
SETA symbols	Range of value is -2^{31} to $+2^{31}-1$	Range of value is -2^{23} to $+2^{23}-1$
SETC symbols	Maximum value of 8 characters	Maximum value of 4 characters. SETC symbols may not be used in the operand of a SETA.
SETC concatenation	The form to concatenate is &A SETC 'A', 'B'	The period may not be used to concatenate. The only concatenation allow is of the form: &A SETC '&B&C'
Substring	Character strings. may be used.	Only variable symbols may be used.
ACTR	Supported	Not supported
MNOTE	Severity code indicator is 0 - 255	Severity code indicator is 0 to 3.
Subscripting.	Supported	Not supported
AIF	Character relations must be inside apostrophes, i.e. 'F', 'E'	Character relations must be preceded by the letter C, i.e. C'F', C'E'

SECTION 7: DIAGNOSTIC MESSAGES

In addition to 44PS Assembler diagnostics the macro processor issues diagnostic messages. All messages except two originate in the Encoder during creation of the macro skeleton.

When an error is detected in a statement, it is flagged on the listing of the definition. If the erroneous statement is generated during the assembly process it will be flagged again by the assembler. One of the diagnostic messages will follow the erroneous statement. (Figure 1).

Should the Encoder detect an error in the prototype statement of a macro definition, a flag of X with a severity of 2 will appear on the left side of the listing. This flag indicates that the entire definition was flushed by the Encoder with no further error checking.

ENCODER DIAGNOSTICS

The following is a list of the diagnostics messages, their severity and flags, generated by the ENCODER.

Severity	Flag	Message
2	P	* ERROR ***** INVALID CHARACTER *****
2	P	* ERROR ***** ILLEGAL SYMBOL LENGTH *****
2	P	* ERROR ***** INVALID CHARACTER AFTER AMPERSAND *****
2	P	* ERROR ***** UNPAIRED PARENTHESIS *****
2	P	* ERROR ***** UNPAIRED QUOTES *****
2	P	* ERROR ***** INVALID TERMINATOR IN NAME OR OP FIELD *****
2	P	* ERROR ***** MISSING COMMA TERMINATOR *****
2	P	* ERROR ***** ERROR IN SETC OPERAND FIELD *****
2	P	* ERROR ***** ERROR IN SET STATEMENT NAME FIELD *****
2	X	* ERROR ***** ERROR IN PROTOTYPE STATEMENT *****

DECODER DIAGNOSTICS

The DECODER does not check syntax of a generated card. It gives the card to pass one of the assembler for assembly and error checking.

The DECODER does issue two diagnostic messages. When either message occurs, the DECODER terminates processing and returns control to pass one.

The following are the decoder messages:

* ERROR ***** UNDEFINED SEQUENCE SYMBOL IN AIF OR AGO SYMBOL IS (sequence symbol)

This message gives a severity of 2 and a flag of U.

* ERROR ***** DECODER PARAMETER TABLE IS TO SMALL. USE PARM OPTION.

This message gives a severity of 3 and a flag of W.

The PARM option may be used to correct the error. (Section 5).

SECTION 8: BRIEF DESCRIPTION OF THE LOGIC OF THE MACRO PROCESSOR

INTRODUCTION

The 44PS Assembler is a two pass assembler. All Macro processing occurs during pass one of the assembler. The Macro processor consists of two routines; 1) ENCODER and 2) DECODER.

The ENCODER routine is used to create a coded form of all Macro definitions in the source input. These coded forms are called "Macro Skeletons." The "Skeletons" are core resident during pass one of the assembly process.

The DECODER routine is used to interpret the "Macro Skeletons" and create source card images for the assembler when a macro instruction is encountered.

THE ENCODING PROCESS

The ENCODER routine is loaded into core by pass one of the assembler when a macro card (macro definition header) is encountered. This routine resides in core following pass one. (Figure 1)

The ENCODER uses the prototype statement of the definition to build a macro name table (Figure 3) and a symbolic parameter table (Figure 4). Pass one of the assembler is used by the ENCODER for all input and output. The ENCODER scans each input card and makes code substitution where necessary. (A list of codes and an example of thier use are illustrated in Figure 5). When the card scan is completed, the card is returned to pass one where it is treated as a comment card for later listing. This process continues until all definitions are encoded.

THE DECODING PROCESS

The DECODER routine is loaded by pass one of the assembler when an operation code on the source input card is found in the Macro name table. This routine overlays the ENCODER area of core. (Figure 2)

The DECODER uses the Macro instruction to form a table of parameters. The macro instruction card is returned to pass one where it is treated as a comment.

The DECODER proceeds to interpret the "encoded" macro. It builds up a card image which is then given to pass one where it is assembled as a source input card.

Pass one is used by the DECODER for input, output, evaluation of character and arithmetic relations and evaluation of all SET symbols.

When decoding is completed, control returns to pass one.

ERRORS FOUND BY MACRO PROCESSOR

When an error is detected during "encoding" the statement is flagged. An error message is placed in the skeleton immediately following the erroneous card image. This message will appear when the macro is "decoded." The flag will appear on the listed definition. If the erroneous card is the prototype it is flagged and the definition is flushed with no further processing.

The DECODER relies on the assembler for error checking of the generated card images. If an error is detected in the macro instruction the statement is flagged and the macro is not "decoded."

The flag W with a severity of 3 is issued by the DECODER when insufficient core is available for its parameter table. This error can be corrected by using the PARM parameter on the EXEC statement. (See Section 5)

A list of all error messages and their severity is contained in Section 7 of this publication.

CORE REQUIREMENT

The ENCODER requires approximately 4800 bytes of core. The DECODER requires approximately 5400 bytes of core. The macro "skeleton" is of variable length depending upon the number and size of the macro definitions. The macro name table requires 12 bytes for each macro defined. 2000 bytes are reserved after the skeleton for the DECODER's parameter table. (fig. 2) The remaining storage is used for the assembler's symbol, attribute and type and length attribute tables. (fig. 2)

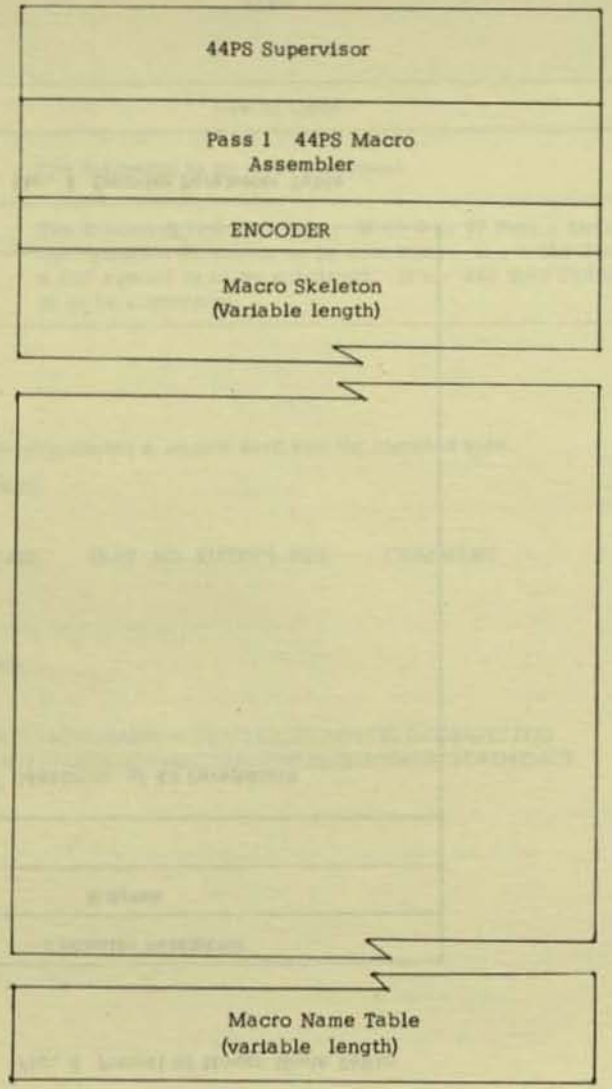


Fig. 1 - Core Allocation During Encoding

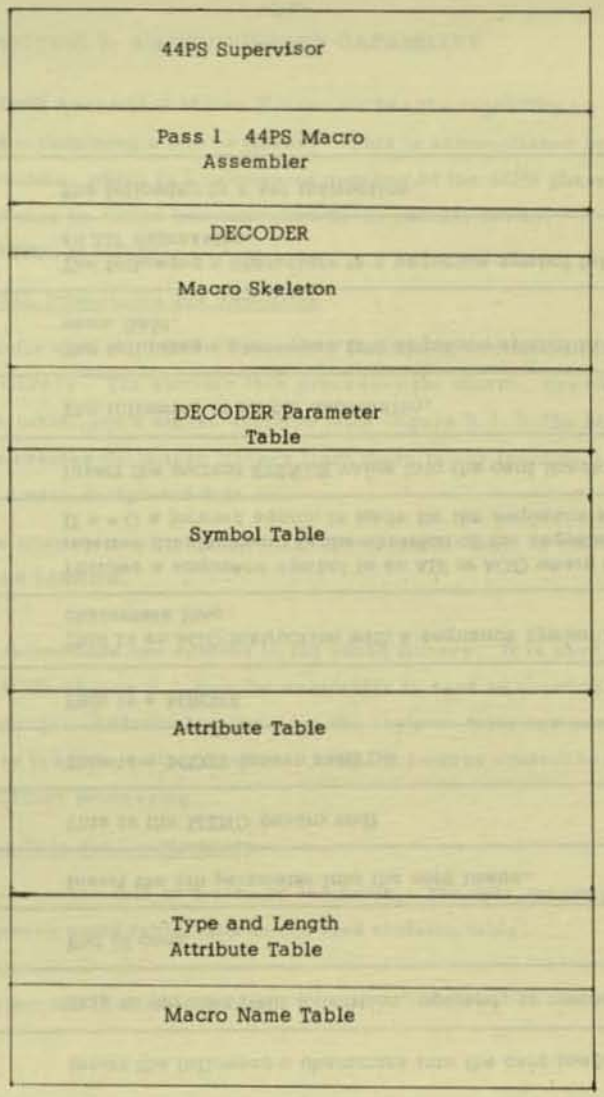


Fig. 2 - Core Allocation After Encoding

Macro Name 8 bytes	4 bytes Relative Displacement

Fig. 3 Format of Macro Name Table

Symbolic Parameter
8 Bytes
Maximum of 49 Parameters

Fig. 4 Encoder Parameter Table

Fig. 5 - Macro Skeleton Codes

Hexadecimal Codes	Use of Code
B0n	Insert the following n characters into the card image.
B1	Skip to the next field (operation, operand, or comment)
B2	End of card
B3n	Insert the nth parameter into the card image.
B4	This is the MEND (macro end)
B5	This is a MEXIT (macro exit)
B6	This is a MNOTE
B7n	This is an AGO instruction with a sequence symbol n characters long.
B8n	Follows a sequence symbol in an AIF or AGO where n is the relative displacement in the skeleton of the sequence symbol. If n = 0 a forward search is made for the sequence symbol.
B9	Insert the current SYSNDX value into the card image.
BA	The following is an AIF expression.
BBn	The following n characters is a sequence symbol in the name field.
BCn	The following n characters is a sequence symbol following an AIF expression.
BD	The following is a set instruction.

Hexadecimal Codes	Use of Code
BE	The following is an error statement.
BFn	The following is a substring. If n= 0 to 49 then n specifies the symbolic parameter to be substrung. If n = 128 then a SET symbol is to be substrung. If n = 255 then SYSNDX is to be substrung.

The following illustrates a source card and its encoded form.

Definition card:

```
.AUD    AIF     (&BT EQ X'F1F2').BUT      COMMENT
```

Encoded card:

```
BB04BC1E4C4B1BAB0014DB00350C2E3B00840C5D840E77DC
6F1B004C6F27D5DBC044BC2E4E2B80000B1B00840C3D6D4D4C5
D5E3B2
```


The model 44PS Assembler Macro Processor has the capability of creating and maintaining a macro library. This is accomplished by the MACUTIL routine, which is a permanent member of the 44PS phase library, and may be called into execution by an execute card:

```
// EXEC MACUTIL
```

9.1 MACUTIL Processing and Execution

MACUTIL calls upon the Macro Processor's encoding routine (ENCODER) to create a library. The encoder then processes the macro, creating a macro name table, and a macro skeleton table (figure 9.1.1). The MACUTIL routine then creates the macro library from these tables (section 9.2), and places it in a user designated data set.

If the macro library routine is updating an existing library, it calls a subroutine called LIBRDR.

LIBRDR is relocatable and resides in the phase library. It is used by either MACUTIL (figure 9.1.2) or the assembler to read an existing library into core storage. Address locations for the skeleton table and name table are passed to it by the calling routine. LIBRDR returns control to the caller for further processing.

9.2 Library Data Set Construction

MACUTIL processes data in blocks of 360 bytes. The data set consists of a header, macro name table, and an encoded skeleton table.

The library header is one block long.

Library Header:

Byte	No. of Bytes	Contents
0-7	8	Library name
8-11	4	Number of macros in library
12-15	4	Length of skeleton table
16-17	2	Last block of the name table
18-19	2	Number of bytes in the first block of the name table
19-20	2	First block of the skeleton table
20-22	2	Last block of the skeleton table
23-24	2	Number of bytes in the last block of the skeleton table
25-36	334	Not used

The length of the name table is dependent upon the number of macros. The table is inverted as will be seen. Each name entry is 12 bytes long. If, for example, the table consisted of one block and two entries, then the format would be as follows:

Name Table:

Byte	Bytes	Contents
360-349	12	Name Entry
348-335	12	Name Entry
334- 0	336	Not used

Name entries have the following format:

Name Entry:

Byte	Contents
349-356	Macro name
357-360	Displacement of the macro skeleton from the beginning of the skeleton table.

The skeleton table is not inverted, and is of variable length (figure 9.2).

9.3 Creating a Library (Data Set)

Since a library is a data set prepared by MACUTIL, it may be treated as any other data set under supervision of the 44PS system.

To create a library, space must be allocated (ALLOC) or accessed (ACCESS) for the data set before the MACUTIL routine is executed. In addition the following cards are necessary:

Library Name Card:

Column	Contents
1-8	library name--a 1 to 8 character name beginning in column 1.
9	blank
XX	a two digit number specifying the SYSUNI index of the device which is to receive the data set

Write Library Card:

Column	Contents
1-8	WRITLIB--keyword specifies the end of all macros for this library

Ex. 1: It is desired to create a library called MACLIB1 on tape unit XXX. The following job control cards might be used:

```
// JOB
// SYSYYY ACCESS MACLIB1, XXX =
// EXEC MACUTIL
MACLIB1  ZZ
         MACRO
         -------
         -------
         MEND
```

```
        MACRO
        -------
        -------
        MEND
    WRITLIB
    /*
    /&
```

SYSYYY is the symbolic unit to be assigned to device XXX. ZZ is its SYSUNI index for the symbolic unit (see IBM System/360 Model 44 Programming System: Guide to System Use, for detailed information on control card usage).

Ex. 2: It is desired to create a library named MACLIB2 on disk XXX. The following job set up will accomplish this.

```
    // JOB
    // SYSYYY ALLOC MACLIB2,XXX='VOLID',BLKCOUNT
    // LABEL 360
    // EXEC MACUTIL
    MACLIB2  ZZ
        MACRO
        -------
        MEND
    WRITLIB
    /*
    /&
```

In this example it is required to specify a block count on the allocate card. MACUTIL processes output in blocks of 360 bytes. Since the block count is variable it may be advantagious to first build the library on tape, or dummy data set. MACUTIL will specify the number of blocks used as part of its output.

9.4 Using The Macro Library

To use the macro library, the following card is necessary:

Library Specification Card

Column	Contents
1-8	MACLIB -- Keyword tells the assembler that there is a library
9	Blank
XXXXXXXX	Library name--eight characters maximum
X	Blank
XX	A two digit number specifying the SYSUNI index of the device where the library is resident

Ex. 3 We wish to use the libraries created in Ex. 1, and Ex. 2 along with new macro definitions:

```
    // JOB
    // SYS003 ACCESS MACLIB1,183=
    // SYS004 ACCESS MACLIB2,OCO='VOLID'
    // EXEC ASSEMBLE
    MACLIB MACLIB1     13
    MACLIB MACLIB2     14
        MACRO
        -------
        -------
        MEND
A       START    0
        ------
        ------
        END      A
/*
/&
```

In the example above, symbolic units SYS003 and SYS004 have been assigned to tape unit 183 and disc unit OCO, respectively. The MACLIB card must immediately follow the // EXEC ASSEMBLE card.

The macro library feature may be used with a source card macro library in conjunction with the UPDATE feature of the Assembler.

Ex. 4: Suppose a tape exists that contains card images of a set of macros definitions (call it MACTAPE). These are sequenced from MAC00010 to MAC00200 and followed by a dummy card which is sequenced 99999999 (these were put on a tape using the copy utility). We wish to merge this set of macros into a program that uses a macro library (MACLIB1), and a new set of macro definitions.

We may accomplish this task if the macro cards in our source deck are sequenced from MAC10000 on; and we use the assembler UPDATE feature. The following control cards may be used:

```
// JOB
// SYS002 ACCESS MACTAPE, 182=
// SYS004 ACCESS MACLIB1, 184=
// SYS003 ACCESS NEWDECK, 183=
// EXEC ASSEMBLE (UPDASMB1)
MACLIB MACLIB1     14
       MACRO            MAC10000
       -------
       MEND             MAC1----
       START
       ------
       END
/*
/&
```

An updated deck will be written as card images on tape 183. Tape 183 will in turn become the input tape to be assembled (Figure 9.4.1).

9.5 Updating and Maintaining a Macro Library

MACUTIL can add, delete, and rename macros in an existing macro library. The following cards are necessary for any library maintenance:

Delete Card:

Column	Contents
1-6	DELETE - keyword, denoting action to be taken by MACUTIL
7	Blank
8-80	name [,name, ...] a list of macros names to delete from the library, each separated by commas

As many DELETE cards as needed are permitted as long as they are contiguous in their order, and they immediately follow the library name card.

Rename Card:

Column	Contents
1-6	RENAME - keyword, denoting action to be taken by MACUTIL
7	Blank
8-80	name,newname b[name,newname b name, ...] ---a list of names to be changed. A comma must separate the new name from the old.

As many RENAME cards as needed are permitted.

Add Card

Column	Contents
1-3	ADD --- keyword denoting action to be taken by MACUTIL.

An add card must be followed by the marco definitions that are being added to the library. The RENAME or ADD cards may be in any order, but must follow all DELETE cards. The WRITLIB card and library name card are required as previously stated.

Ex. 5 Suppose we wish to manipulate a library using all MACUTIL options. The following deck set up will accomplish the task:

```
// JOB
// SYS003 ACCESS MACLIB1, 184=
// EXEC MACUTIL
MACLIB1  13
DELETE   NAMEA, NAMEB
ADD              MACRO
                 -------
                 MEND
RENAME name3,noname, name4, name3
WRITLIB
/*
/&
```

9.6 Error Messages and Recovery

1. OPEN UNSUCCESSFUL - JOB TERMINATED ---a data set is not properly defined. Check the validity of the SYSUNI index and the device status.
2. SYNTAX ERROR ON LIBRARY CONTROL CARD ---a library control card is not properly defined. Check job control set up.
3. NAME CANNOT BE FOUND IN LIBRARY ---macro NAME has already been deleted, or renamed. Check to see if proper library is being processed.
4. ENCODER PHASE CANNOT BE FETCHED ---the ENCODER is not present in the phase library. LNKEDT and KEEP the ENCODER and retry job.

5. INPUT - OUTPUT ERROR - JOB CANCELLED --- an I/O error occurred during job processing, retry the job.
6. MACRO LIBRARY NOT READ - I/O ERROR --- LIBRDR message. The library was not on the volumn requested. Check, and retry.

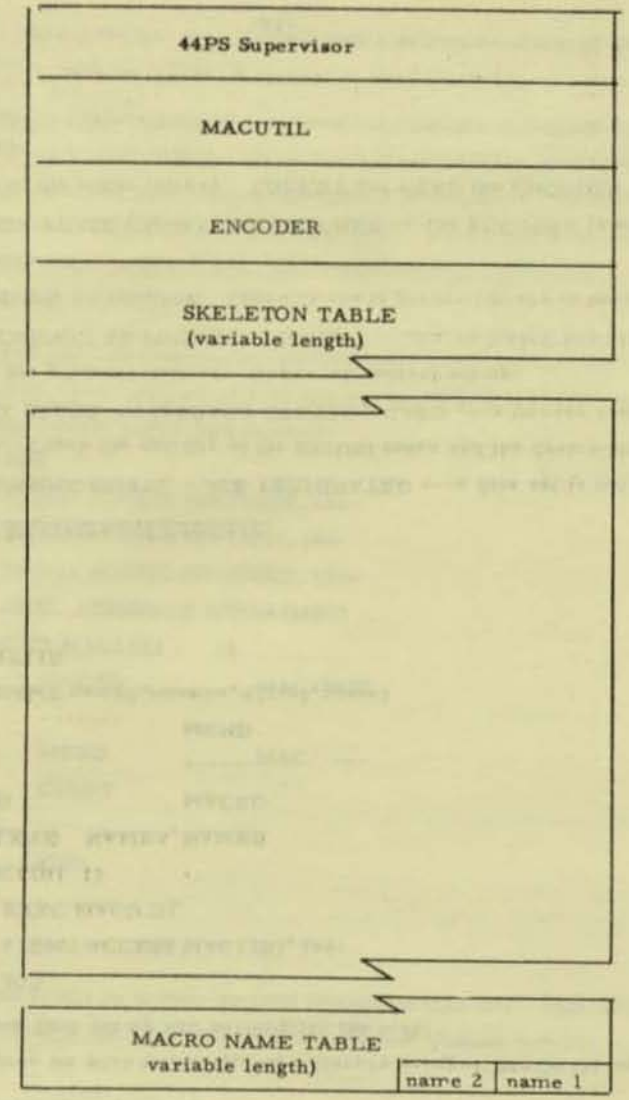


Figure 9.1.1 Core allocation during library creation.

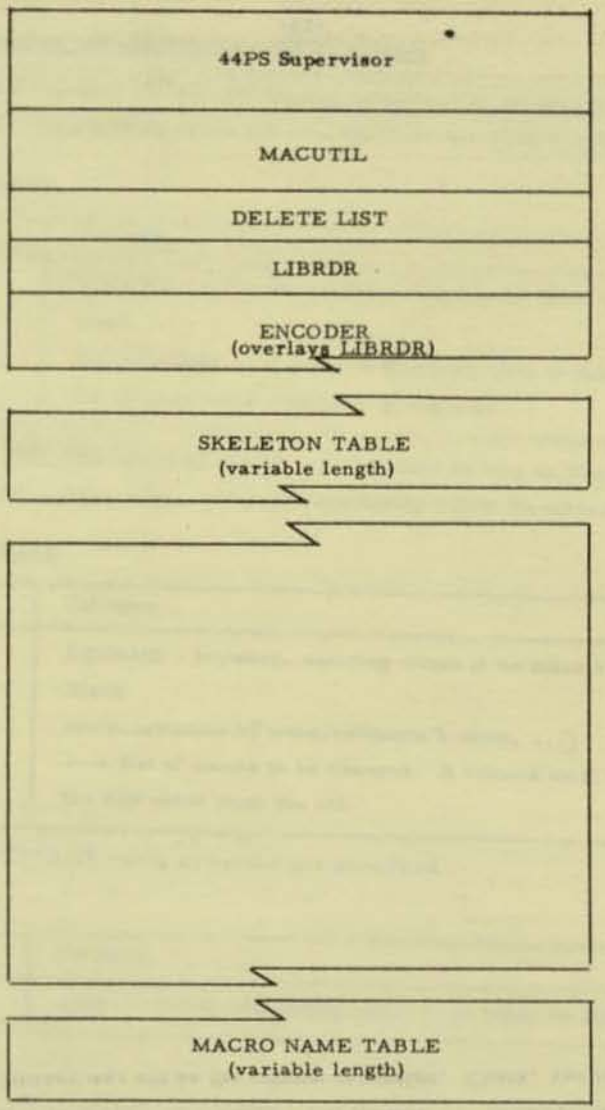


Figure 9.1.2 Core allocation during library update.

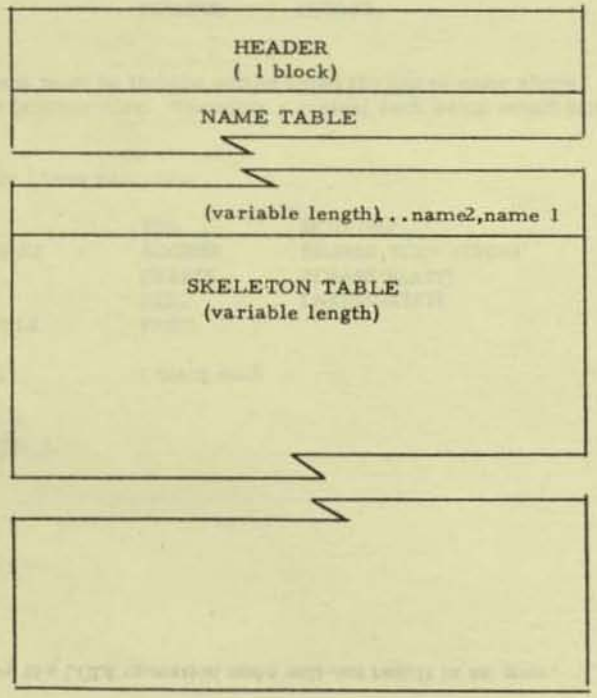


Figure 9.2.1 Library data set.

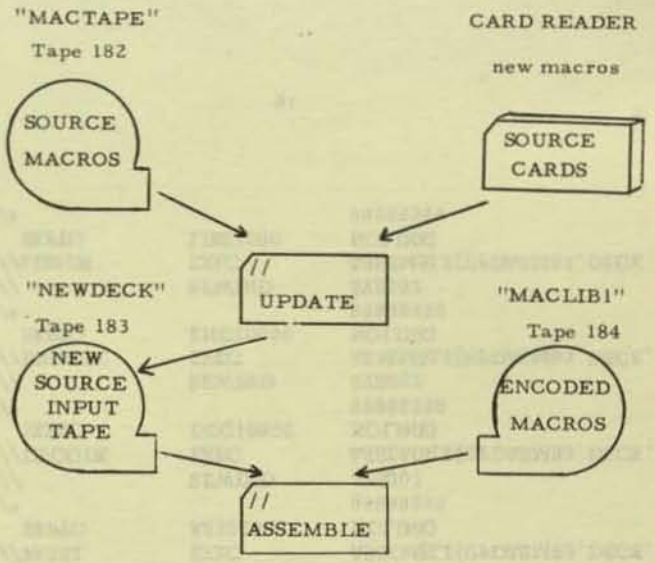


Figure 9.4.1 A source macro library is to be merged with a source deck, containing macro definitions, using the UPDATE feature, and then assembled with an encoded macro library.

SECTION 10: NOTES AND RESTRICTIONS

LOCAL AND GLOBAL OPERATION CODES

If the programmer wishes to remain compatible with other OS/360 assemblers and use local and global definitions, it is recommended that he define macros using local and global operation codes as macro names. These macros need do nothing. An example is as follows:

```
MACRO
LCLA
MEND
```

With the above the LCLA operation code will not result in an error.

This tape is blocked by a factor of 2000/25. The order of data is as follows:

Program	Phase Name	Identification
1) Pass 1	BLAST	Serial number AS1000000
2) Pass 2	BLAZE	Serial number AS2000000
3) Decoder	DECODER	Serial number DEC10000
4) Encoder	ENCODER	Serial number ENC10000
5) Librdr	LIBRDR	Serial number LIBR1000
6) Macutil	MACUTIL	Serial number MCU10000

The control cards need to assemble the data are as follows: (The input tape is assumed to be on 184)

```
//              JOB       NODUMP
//SYS002        ACCESS    TAPE,184=
//              LABEL     2000
//PASS1         EXEC      ASSEMBLE(DECK,NOLINK,
                          UPDASMB3) (serial field)
/*                        99999999
//            REWIND      SYS002
//PASS2         EXEC      ASSEMBLE(UPDASMB3,DECK,
  SKPTO       AS200000    NOLINK)
/*                        99999999
//            REWIND      SYS002
//DECODE        EXEC      ASSEMBLE(UPDASMB3,DECK,
  SKPTO       DEC10000    NOLINK)
/*                        99999999
//            REWIND      SYS002
//ENCODE        EXEC      ASSEMBLE(UPDASMB3,DECK,
  SKPTO       ENC10000    NOLINK)
/*                        99999999
//            REWIND      SYS002
//LIBRDR        EXEC      ASSEMBLE(UPDASMB3,DECK
  SKPTO       LIBR1000    NOLINK)
/*                        99999999
```

```
//              REWIND      SYS002
//MACUTIL       EXEC        ASSEMBLE(UPDASMB3,DECK,
  SKPTO         MCU10000    NOLINK)
/*                          99999999
//              REWIND      SYS002
/&
```

The object deck must be linkage edited using the phase name above. They can reside at problem core. Therefore a typical deck setup would look as follows:

Deck setup for phase assemble:

```
//              JOB         NODUMP
//SYSAB1        ACCESS      SDSABS,0C0='VER003'
//              DELETE      SDSABS(BLAST)
//              EXEC        LNKEDT(KEEP)
MODULE          PASS1

Pass1           Object deck

PHASE BLAST,*
INCLUDE PASS1,L
/*
/&
```

SNAP PROCESSOR (PROTOTYPE)

360D-03.3.010

DISCLAIMER

This program and its documentation have been contributed to the Program Information Department by an IBM customer and are provided by the IBM Corporation as part of its service to customers. The program and its documentation are essentially in the author's original form and have not been subjected to any formal testing. IBM makes no warranty expressed or implied as to the documentation, function, or performance of this program and the user of the program is expected to make the final evaluation as to the usefulness of the program in his own environment. There is no committed maintenance for the program.

Questions concerning the use of the program should be directed to the author or other designated party. Any changes to the program will be announced in the appropriate Catalog of Programs; however, the changes will not be distributed automatically to users. When such an announcement occurs, users should order only the material (documentation, machine readable or both) as indicated in the appropriate Catalog of Programs.

CONTRIBUTED PROGRAM LIBRARY SUBMITTAL FORM
(for IBM S/360, 1130 and 1800)

IBM Corporation
Program Information Department (PID)
40 Saw Mill River Road
Hawthorne, New York 10532, U.S.A.
Attention: Program Control Desk

This form should be completed and submitted with the program package to PID at the address shown above. Standards and instructions for submitting programs are in your *User Group Reference Manual* or the *Contributed Program Submittal Standards Manual* available from PID.

(1) **Program Order Number** *(to be filled in by PID)* 360D-03.3.010

(2) **System Type** *(machine)* S/360

(3) **Search Key** . . . . . /EXECUTES/ SNAP /A/ BAS
IC ENGLISH /FOR/ TEXT PRO
CESSING /FOR/ HUMANITIES L
IBRARIANS EDUCATORS /ETC/

(4) **Name of Author** *(if different than submitter's)* . . . . . William M. Ruhsam
Michael P. Barnett

(5) **Submitter's Name** *(direct technical inquiries to)* . . . . Michael P. Barnett

(6) **Submitter's Address** . . . . . School of Library Service
Columbia University
New York, N. Y. 10027

(7) **Title of Program** . . . . SNAP PROCESSOR (PROTOTYPE)

(8) **Submitter's User Group Affiliation Code and Installation Code** . . . . . S BWY

(9) **Submitter's Own Program Identification and Suffix** *(optional)* . . . . . PIET 0

(10) **Primary Subject Code** 03.3

(11) **Secondary Subject Codes** . . . . . 03.6 06.6 06.7 . .

(12) **Operating or Monitor System Required** . . . . . SEE ABSTRACT

(13) **New or Revision Code** *(if revision, show prior Program Order Number in item 1)* . . . . . N

(14) **Year Completed** 69

(15) **Date of Submittal** 210469

(16) **Documentation** *(number of original pages submitted)* 37

(17) **Abstract** *(should contain sufficient information for a reader to determine the value of the program). Listed on the reverse side of this form are subjects which may serve as a guide for a descriptive abstract.*

CONTRIBUTED PROGRAM LIBRARY SUBMITTAL FORM

Subject Guide

a. Purpose
b. Programming Language used
c. Version and modification level or release number of IBM Programming System used, or program order number for non-IBM authored program used
d. Field of application
e. Type of routine (main program, subroutine, etc.)
f. Specific description of machine requirements
g. Engineering Changes (EC) level of equipment (if pertinent)

ABSTRACT

The SNAP processor executes procedures written in the SNAP language- a "basic English" for librarians, educators, publishers and others to instruct the computer to perform mechanical test processing. (see M.P. Barnett and W.M. Ruhsam. IEEE Trans. EWS. Vol. EWS-11,2,45,1968; M.P.Barnett. Computer Programming in English. Harcourt,Brace & World, May 1969). The SNAP processor is written in FORTRAN IV. When compiled it consists of a root segment of 56K bytes, the translator segment of 45K bytes (that converts SNAP procedures into the numerical SNAPIC code) and the interpreter segment of 48K bytes that can be overlayed with the translator. The minimum requirements for the compiled processor are 128K core storage, card reader, line printer and 1 tape drive (or an equivalent drum or disc). SNAP instructions to punch cards and paper tape, to read and write magnetic tape and direct access devices, and to punch paper tape can be used in SNAP procedures that are run on equipment with appropriate units. The compiled processor can be stored in a language library and used conveniently under OS/360. It has been used extensively for class excercises at Columbia University.

(Please attach additional pages if necessary) . . . . . **Total pages attached** _____

Permission to Publish

"I hereby give anyone permission to reprint, reproduce, and distribute this program to anyone else."

(18) **Signature of Submitter and Date** Murvel P. Barnett

(19) **Signature of Installation Addressee** Kenneth M. King

T4SF

Table of contents

Tape key sheet

The submitted volume is a 9 track unlabelled magnetic tape (external identification CU0463) written in 1600 b. p. i. with two files (of EBCDIC card images, 2000 bytes, i. e. 25 card images, per block), and two tape marks, as follows:

1. File no. 1 (6252 records, with sequence numbers in columns 73-80)* The FORTRAN IV source version of the SNAP processor, consisting of the main program PIET, and the subprogram SCHAR that comprise the root segment, the subprograms COMPL, BACK, CIFST, CNUBR, ECTNUM, ERRPP, FORE, FVERB, IFTYP, KOMPL, NAMES, RLINE, TRANS, that comprise the translator segment, and the subprograms SNAP3, ABLST, ARTHM, BLKLO, COPY1, EDJEK, ERROR, IFRT, PRINT1, PUSH, READI, STORE and TAPCT that comprise the interpreter segment.

2. Tape mark

3. File no. 2 (460 records): A test and demonstration job stream that consists of 17 steps:

   (1) COMPSNAP, that invokes the FORTRAN compiler (IEYFORT) to compile the processor

   (2) LINKSNAP, that invokes the LINK EDITOR (IEWL) to link edit; the DD, INCLUDE, INSERT and OVERLAY statements needed for overlay are included: The load module is retained in temporary storage for the further steps (it could be stored and cataloged in a language library by changing the DSNAME in the SYSLMOD DD statement from & GOSET(MAIN) to the name of the language library)

   (3) TEST01, that invokes the SNAP processor, which has just been link edited, by the statement

   ```
   //TEST01 EXEC PGM=*.LINKSNAP.SYSLMOD
   ```

   to execute the first of the SNAP test and demonstration procedures

   (4)-(17) TEST02 to TEST15 that invoke the SNAP processor in the same way to execute the further SNAP test and demonstration procedures.

4. Tape mark

* The sequence numbers are 100(100)316600(10)316620, 316700(100)382500, 382510, 382600(100)469000, 472900(100)55100, 55110, 551100(100)566600, 566610, 566700(100)628300, 644700(100)644800. A uniform increment will be used in a forthcoming revision.

The file no. 2 can be punched with the standard IBM utility routine IEBPTPCH. This is done at the submitting installation with a deck that is punched as follows:

```
//BASNSH07 JOB (URTAZ03,I,1000,500),M.P.BARNETT,MSGLEVEL=1
/*SETUP DDNAME=SYSUT1,DEVICE=2400-9,ID=(CU0463,,,NL)
//PUNCHJOB EXEC PGM=IEBPTPCH
//SYSPRINT DD SYSOUT=A
//SYSUT1 DD UNIT=2400-9,VOL=(,RETAIN,SER=CU0463),LABEL=(2,NL),
//   DISP=(OLD,PASS),DCB=(RECFM=FB,LRECL=80,BLKSIZE=2000)
//SYSUT2 DD SYSOUT=A
//SYSIN DD *
 PUNCH
/*
```

To punch the file elsewhere: (1) the job card should be changed to give the user's account number and name, (2) the /* SETUP card should be omitted if ASP is not used, (3) the tape I. D. should be changed, to that of the tape on which the processor has been received, in the SYSUT1 statement (and in the /* SETUP statement if ASP is used).

After punching the file, a JOB card should be added at the beginning, a /* SETUP card if ASP is used, the tape I. D. should be changed in the SYSIN statement in the step COMPSNAP, and the cards punched //* REPLACE THIS CARD BY AN END OF DATA SET MARKER at the end of the step LINKSNAP and the fifteen tests should be replaced appropriately (e. g. by /* cards in most installations). These sixteen cards that need replacement are the 23rd, 59th, 83rd, 104th, 138th, 188th, 210th, 227th, 262nd, 290th, 329th, 359th, 385th, 428th, 442nd, and 460th. Some changes also may be needed in space allocation and other details of the JCL cards in the compile and link edit steps. These can be found by inspecting the diagnostic messages if the run fails.

An improved SHARE submittal, that will provide the complete job stream as the second file, the results of the test SNAP procedures as the third file and number the records in the first file with uniform increment is being prepared and will be submitted shortly.

SNAP PROCESSOR (PROTOTYPE) - Program Write-up: April 1969

Contents

1. Introduction

This report describes the structure and operation of the SNAP processor, that executes procedures which are expressed in the SNAP language - a "basic English" for editors, librarians, educators and others to instruct computers to perform mechanical text processing. The language has been summarized in tabular form (1), and it is presented in a book designed for teaching purposes (2).

The SNAP processor is written in FORTRAN IV. It consists of:

(i) a small general control section,

(ii) a translator, that converts a SNAP procedure into a numerical representation in "SNAPIC" code, and stores this in the "procedure table",

(iii) an interpreter, that executes procedures which have been stored this way.

In the standard version of the processor that is operating at present the control section of the processor is bound as a root segment of 56K bytes, and the translator and the interpreter are bound as segments of 45K bytes and 48K bytes respectively, that can be overlayed. The overall core requirement of 104K bytes can be altered easily by changing the size of certain arrays that are described next.

2. The Principal Arrays

These arrays store the coded representations of procedures and of strings, quantities and lists that the SNAP procedures manipulate. The sizes of these arrays impose limits on the length and other features of the SNAP procedures that can be processed. The sizes of several of the arrays are written symbolically, in statements that, for example, test for overflow. Values are given to the symbolic names for the array sizes initially, by PIET. The processor can be modified to use arrays of different sizes accordingly by:

(i) changing the relevant DIMENSION and COMMON statements systematically, at the beginning of PIET and each subroutine,

(ii) changing the relevant initialization statements in PIET.

The principal arrays, their symbolic names and sizes, and their purpose are as follows:

(1) The procedure table, PROCB, symbolic size PTSIZE, standard size 7000. The translator stores the SNAPIC representation of a SNAP procedure in this table. Quotations (as in the instruction PRINT "ABRACADABRA") are embedded, in numerical code, within instructions in PROCB. The interpreter executes the SNAPIC instructions in PROCB. The length of a SNAP procedure is limited by the length of its SNAPIC representation, which cannot exceed the length of PROCB. An empirical estimate of the number of SNAPIC codes needed to represent a SNAP procedure is A+Rx(B+C) where A is the total number of characters in quotations, B is the combined number of unconditional statements and of conditions and action clauses in conditional statements, C is the total number of occurrences of numbers, names, quotations and extract expressions, and R is between 2 and 3 (and often very close to 2).

(2) The string bank, STRBNK, symbolic size TSIZE, standard size 5000. The interpreter stores strings that it forms by COPY, APPEND, DELETE and OVERWRITE instructions, and that it immits by input instructions, in STRBNK, in numerical code. Strings are stored consecutively whenever possible, with provision for chaining as described

at the end of the next section.

(3) The string directory, STRDIR, standard size 50: The interpreter stores pointers in STRDIR to the strings that are given unsubscripted names in a procedure and which are represented in the procedure table or in the string bank, as described in the next section. The name is represented in SNAPIC by the pointer to the appropriate entry in STRDIR.

(4) The number bank, NUMBK, standard size 50: The interpreter stores, in NUMBK, numerical values of quantities that are given unsubscripted names in a procedure. The name of a quantity is represented in SNAPIC by the pointer to the appropriate entry in NUMBK.

(5) The subscripted string directory, ASEG, symbolic size ASEGZ, standard size 500. The interpreter stores, in ASEG, pointers to individual strings that have subscripted names, using the same conventions as the string directory STRDIR. Pointers to successive elements of a list of strings are stored consecutively in ASEG. Chaining tactics are used when the size of a list is changed, as described at the end of the next section.

(6) The string list directory, ALIST, symbolic size 50: The interpreter stores, in ALIST, the pointers to the elements of ASEG that point to the first member of each list of strings. The name of a list of strings is represented in SNAPIC by a pointer to ALIST.

(7) The quantity list directory BLIST, symbolic size 50: The interpreter stores, in BLIST, the pointers to the elements of BSEG that contain the values of the first element of each list of quantities. The name of a list of quantities is represented in SNAPIC by a pointer to BLIST.

(8) The subscripted quantity bank, BSEG, symbolic size BSEGZ, standard size 500: The interpreter stores, in BSEG, the values of individual quantitites with subscripted quantity names. Successive elements are stored consecutively, with provisions for chaining.

(9) The statement buffer, STRING, standard size 1000. This stores a SNAP statement one character per word to be translated into SNAPIC in the procedure table.

(10) The translator card image buffer NCARD, size 80. This stores a card image, one character per word, during the immission of a SNAP procedure (by the subroutine RLINE of the translator). Characters are copied from it into the array STRING.

(11) The word buffer, WORD, standard size 100. This stores a single word (i.e., sequence of non-blank characters) from an input statement, for identification in the translation process. Quotations are not put into WORD.

(12) The name bank, NBANK, standard size 5000. All unsubscripted and generic string and quantity names, and statement labels are stored in NBANK by the translator (the subroutines NAMES and TRANS do this). Pointers to the ends, in NBANK, of the successively encountered names of different kinds are kept in four pairs of 50 word arrays by the subroutine NAMES, and pointers to the ends of the labels are kept in two 100 word arrays by the subroutine TRANS. This allows 100 statement labels. All this information is discarded by the end of the translation process.

(13) The flag list KFLAG, size 20. This transmits flags that are set by SNAP control cards, to request debug information, and for other purposes described in Section 8.

(14) The control list PINT, size 20. This transmits some control information, such as the lengths of certain arrays and error flags.

(15) The heading buffer PNAM, size 80. This was intended to transmit the title of a procedure (from the subroutine FVERB to higher levels of the translator), but the relevant portion of the program is incomplete.

(16) The numerically sequenced EBCDIC code list, COT, size 128. This contains EBCDIC codes for characters that can be punched on the 029, arranged in decreasing numerical order. It is defined and used in the subroutine SCHAR that is called by all three sections of the processor.

(17) The indirect EBCDIC-SNAPIC conversion table, NB, size 128, (in subroutine SCHAR). The 029 character whose EBCDIC code is COT(I) has the SNAPIC code NB(I).

(18) The SNAPIC-EBCDIC conversion table, NCHAR, size 129 (in subroutine SCHAR). The character whose SNAPIC code is J has the EBCDIC code NCHAR(J).

(19) The output buffer, OBUF, standard size 150. EBCDIC output records are formed one at a time in OBUF (by the interpreter subroutine STORE) and written on output devices (by the interpreter subroutines STORE, PRINT1 and TAPCT).

(20) The auxiliary string buffer STRN1, standard size 300. The interpreter processes a string expression, e, that occurs as the left comparand of a string comparison, or in a statement of the form SET...TO e, or SET ... TO THE LENGTH OF e, by writing the string explicitly in STRN1. (This is done by the subroutine STORE.)

(21) The auxiliary string buffer STRN2, standard size 100. The interpreter subroutine STORE processes the right comparand of a string comparison by writing the string that it represents in STRN2.

(22) The substring push down list, KCTAB, standard size 40. When an expression that represents a string in the procedure table is processed by the interpreter, and the SNAPIC code for the name of a string that is defined elsewhere in the procedure table is encountered, a pointer is put in KCTAB, to the position in the current expression at which processing should continue, after the expression that the name references has been processed.

(23) The extract expression pointer tables PTR, PTT1, PTT2, and PIP. These deal with implicitly nested expresssions, and are described in Section 6.

(24,25) The interpreter input record image buffers, ICARD(80), and TRECD(1000). The latter is used for FETCH, and the former for all other input statements.

3. The Internal Representations

The SNAPIC codes for SNAP procedures

The beginning of a SNAP procedure is shown in the procedure table, PROCB, by the integer -1. This is followed by the SNAPIC representations of the successive statements of the procedure. Each of these begins with a negative integer that signifies the kind of statement, and ends with the integer -501, which serves as an end of statement delimiter, and is

denoted symbolically, in the processor, by EOS. Several other negative integers are used as delimiters and precedence codes within SNAPIC instructions, as described below. The symbolic name that is used for each of these is given in brackets, after its value. The symbolic names for frequently used pointers are also given in brackets, when they are mentioned. Most SNAP instructions contain expressions that represent quantities and/or strings. The SNAPIC representations of these are described first, and the representations of the complete instructions then described in turn. The different forms of quantity and string expressions are numbered consecutively for convenient later reference.

Quantity Expressions

(i) The SNAPIC for a number consists of

(a) the integer -108, (NBA),
(b) the number in hexadecimal

(ii) The SNAPIC for an unsubscripted quantity name consists of

(a) the integer -106, (NBR),
(b) the pointer, (N), to the number bank NUMBK

(iii) The SNAPIC for a subscripted quantity name consists of

(a) the integer -107, (NLT),
(b) the pointer, (NL), to BLIST,
(c) an integer (PN) that is equal to the subscript if the subscript is numeric; and is the negative of the pointer to the element of the number bank that contains the value of the subscript, when the subscript is literal.

Numbers and the values of quantities that are given names, are limited at present to the range -65,536 to 65,536.

String Expressions

(iv) The SNAPIC for a quotation (e.g., "ABRACADABRA") consists of

(a) the integer -101, (LD), that serves as a left quote mark,

(b) the SNAPIC codes for the successive characters in the quotation (see Table 1) and
(c) the integer -102, (RD), that serves as a right quote mark.

(v) The SNAPIC for an unsubscripted string name consists of the pointer, (S), to the string directory STRDIR. The element of STRDIR that this identifies contains either (a) a pointer to the beginning of the string in the string bank, if it is currently stored there, or (b) an integer, which when reduced by the size of the string bank, is the pointer to the beginning of the expression that represents the string in the procedure table, when it is currently represented there.

(vi) The SNAPIC for an extract expression consists of

(a) the integer -103, (EXT),
(b) the pointer, (S), to the entry in the string directory for the string in which the extract is defined,

(c) the ordinal, (PF), of the leftmost character of the extract in this string,
(d) the ordinal, (PL), of the rightmost character (this limits extracts to individually named strings - the processor will be expanded in due course to allow extracts of strings that are specified by subscripted names).

(vii) The SNAPIC for a line feed outside a quotation consists of the integer -104, (LFD).

(viii) The SNAPIC for a subscripted string name consists of

(a) the integer -105, (EXL),
(b) the pointer (SL) to ALIST,

(c) an integer (PS) that represents the subscript in a manner analogous to that described for subscripted quantity names.

The SNAPIC representations (i) to (iii) of numbers and quantity names can also be used in certain contexts that require a string expression to stand for the string that expresses the value as a decimal number.

(ix) The SNAPIC codes for string expressions that are joined by the word THEN in a SNAP instruction are written in sequence in the SNAPIC representation of the instruction.

List Expressions

(x) and (xi) The SNAPIC codes for the name of an entire list of strings or quantities are analogous to those for a single element, except that the third code, which represents the subscript of an element, is 0 for the entire list.

The SNAPIC representations of the different kinds of statement are summarized below. Commas separate successive codes and sequences of codes that are designated by letters which are explained after their first use. The first code of each statement is called the operation class code, and the second code usually serves as an operation modifier code, that identifies the operation within a class of related operations.

Output: -2,K,X,-501; where the operation modifier code K is 1,2,3,4,5 for TYPE, PERFORATE, PRINT, PUNCH and WRITE respectively. X stands for a SNAPIC string expression of one of the forms (i) to (iii), or for a sequence of such expressions, that represent the string to be printed.

Call: 3,X,-502,N,-501; where X stands for the SNAPIC of the direct object, as just described, and N stands for a SNAPIC unsubscripted or subscripted string name, of the forms (v) and (viii) respectively, that is the indirect object of the statement.

String Synthesis and Alteration:

COPY: -4,1,X,-502,N,-501
APPEND: -4,2,X,-502,N,-501
DELETE: -4,3,-103,N,m_1,m_2,-502,-501
OVERWRITE: -4,4,q,X,-502,N,m,-501

where N denotes the SNAPIC name for the string to be formed or altered, X denotes the expression for the string to be copied, appended or superimposed; m_1,m_2,m denote SNAPIC numbers or unsubscripted quantity names of the forms (i) and (ii); the values corresponding to m_1 and m_2 point to the ends of the deletion zone, and the value corresponding to m points to the first and last character to be overwritten when q is 1 and 2 respectively.

Input: -5,K,L,-501; where K is 1,2,3 or 4 for REQUEST, READ, FETCH and ACCEPT respectively, and L is an expression of the form (v) or (viii) to immit an individual string, or (x) or (xi) to immit a list of strings or numbers.

Continue and Repeat: -7,P,-501; where P points to the beginning, in the procedure table, of the instruction to which control is transferred.

Binary SET Statements, i.e., SET a TO THE f OF b AND c; where a and b denote unsubscripted or subscripted quantity names and c denotes a quantity name or a number: -8,K,B,C,A,-501; where K is 1.., 8 for SUM, DIFFERENCE, PRODUCT, QUOTIENT, REMAINDER, CEILING, GREATER, LESSER respectively; and A,B,C denote the SNAPIC codes for the three operands a,b,c respectively.

Increase and Decrease statements are represented as the equivalent SET statements.

Unary SET statements: (i.e., SET a TO d): -8,K,D,-502,A,-501; where K is 9 when d denotes a number or a quantity name, or a string expression (that may be a quotation, a string name or an extract, or several such items joined by THEN), in which case D is its SNAPIC representation of one of the forms (i)-(vi), (viii), or (ix) (with the restriction in this last case that the concatenated items are only of the forms (iv), (v), (vi), (viii)).

K is 10 for a list definition of the form SET THE a LIST TO n_1,n_2,...; A is then the SNAPIC representation of the name of the list, of the form (xi) described earlier, and D consists of the elements of the list, represented as SNAPIC numbers. (Exceptionally, when the list contains only one element, K is 9).

K is 11 when the second operand is the length of a string, and D is then the SNAPIC for the expression that represents the string (and which may not contain numbers or quantity names, or names of other strings that are defined directly or indirectly by use of numbers or quantity names - this is an accidental but innocuous restriction).

K is 12 when the second operand is the length of a list, and D is then the SNAPIC for the list name, of the form (x) or (xi) described earlier.

Execute Statements: -9,-501.

Reserve Statements: -10,I,Y,-501, where I denotes an integer that specifies the number of characters or list elements for which space should be reserved, and Y is either a SNAPIC string name of the form (v) or (viii), or a list name of the form (x) or (xi).

Select and Rewind Statements: J,K,-501, where J is -11 and -12 respectively, and K is the relevant data set reference number in the range 1 to 99.

Terminate Statements: -99,-501.

Conditional (IF) Statements: -6,Q,-501,S_1,..S_I,-503,F_1,... F_J,-504; where Q denotes the SNAPIC condition and S_1 and F_J denote success and fail clauses respectively. There must be at least one success clause but there need not be any fail clauses. Any of the unconditional SNAPIC statements described so far may appear as a success or as a fail clause.

Exhaustion of Input: Q is 1,1 or 1,2 for card and other input respectively.

String Comparison: Q has the form 2,X_1,X_2; where X_1 and X_2 denote the SNAPIC codes for the comparand string expressions that may be of the forms allowed for the second operand in a unary SET statement (see above).

Numerical Comparison: Q has the form 3,K,M_1,M_2; where K is 1,2,..,6 for EQUAL, GREATER THAN OR EQUAL, GREATER THAN, LESS THAN OR EQUAL, LESS THAN and UNEQUAL, respectively; M_1 denotes the SNAPIC codes for the left comparand, that may be a number or an unsubscripted or subscripted quantity name, and M_2 denotes the SNAPIC codes for the right comparand that also may be of these forms.

String comparison (of the ..nth and SUBSEQUENT (PRECEDENT).. kinds).

For a condition of the form

THE mth AND SUBSEQUENT CHARACTERS OF x_1 ARE THE SAME AS x_2

Q has the form 4,1,M,X_1,X_2, where M,X_1 and X_2 denote the

SNAPIC codes for the number or unsubscripted quantity name m, and the string expressions x_1 and x_2 (that satisfy the restrictions mentioned above for simple string comparisons).

The second code in Q takes the value 2 for the corresponding condition with PRECEDENT in place of SUBSEQUENT, and the values 3 and 4 for the conditions of the forms

x_1 IS THE SAME AS THE m-th and SUBSEQUENT (PRECEDENT) CHARACTERS OF x_2.

String bank representations: A string is represented in the string bank by the SNAPIC codes for its successive characters. A left delimiter code, -101, is put before the first of these in some circumstances (it is redundant however). A right delimiter code, -102, and an end of block delimiter, -199, are put at the end. When a name is given for the first time to a string that is put into the string bank, consecutive storage locations are used for its successive characters starting at the first available position in the string bank. When the same name is given later to another string, the processor starts to write the new string from the same point as the earlier string with the same name. If the new string is longer, the block in which it is being written is chained into the next available word in the string bank by replacing the right delimiter code -102 before the end of block delimiter, with the pointer to this word. Another right delimiter and end of block delimiter are put at the end of the new string when it is complete.

A DELETE operation is performed at present by replacing each deleted character with a null code, -190. Asterisks are included in SNAPIC strings to connote literal (instead of typographic) interpretation of the characters they precede. These factors are accommodated by the interpreter in character counting operations.

Auxiliary buffer representations: Strings that are comparands and operands of arithmetic statements are represented in these buffers as simple sequences of SNAPIC codes.

Quantity list representation: When a quantity list name is used for the first time in a procedure, the appropriate number of elements are stored in successive words of the subscripted quantity bank starting from the first available position, followed by 0 (to provide space for a chaining pointer later

if necessary) and an end of segment delimiter, -32,767. (If the first mention of the list is in a subscripted name, the length is made equal to the subscript in this name, and the earlier elements simply set to 0.) When a list is redefined with additional elements, or an element whose subscript exceeds its prior length is defined, the processor chains the segment to the next free word by putting the negative of its pointer into the word before the end of segment delimiter. A zero element and end of segment delimiter are put at the end of the revised list.

String list representation: Similar tactics are adopted with the lists of pointers in the subscripted string directory. The end of a list is demarked by a zero entry, followed by the end of segment delimiter, -500 (ASEGZ). A segment is chained by putting the negative of the chaining pointer in place of the zero before the end of segment delimiter. A zero element and an end of segment delimiter are put at the end of the revised list.

4. The Control Section

This consists of

(i) the main program PIET,

(ii) the subroutine SCHAR which is used by the translator to convert EBCDIC code to SNAPIC, and by the interpreter to convert back,

The main program PIET

(i) calls SCHAR for initialization

(ii) calls COMPL to translate a SNAP procedure into SNAPIC

(iii) calls SNAP3 to interpret it

(iv) calls EXIT when appropriate

5. The Overall Operation of the Translator

The translator consists of the subroutine COMPL and its subsidiaries, that are listed below. COMPL is called by the main program PIET. Its overall effect is to immit a SNAP procedure

from an input deck, and to write the SNAPIC representation of this into the procedure table PROCB. Further information is also written into several of the other tables which were described in Section 2.

COMPL first initializes the procedure table and several variables directly and it also initializes some further variables and tables by calling ERRPP and NAMES.

COMPL then progresses through a major cycle of operations that consists of the following three stages:

(i) RLINE is called to obtain a SNAP statement from one or more input records. It does this by reading a record, whenever necessary, and transferring the characters that form a complete statement into the array STRING. If the statement is labelled, the label is stored in the name table, NTAB.

(ii) FVERB is called to identify the type of statement, to set the switching parameter TRA accordingly, to write the operation class code in the procedure table, and to set KSTYP to the operation modifier code. The other literal arguments of FVERB are irrelevant at this point. Several items of information are communicated via common blocks.

(iii) The translation of the statement into SNAPIC is completed by further portions of COMPL, which are reached from statements that branch on the parameter TRA. More of the work is performed by subroutines, BACK, FORE, ECTNUM, CIFST and CNUBR that share access to numerous tables and variables through block common storage.

This cycle is repeated, in the absence of irrecoverable error conditions, until FVERB recognizes an EXECUTE instruction in STRING, and sets the switching parameters accordingly. COMPL then branches to a sequence of instructions that

(iv) calls TRANS to perform the second pass on statement labels, putting pointers to labelled statements in the SNAPIC representation of CONTINUE and REPEAT instructions, in PROCB,

(v) calls NAMES to print the names of string, quantities and lists, for testing, and calls TRANS again, to print the statement labels, and

(vi) returns control to PIET

COMPL also returns control to PIET when an irrecoverable error condition is detected, with the error indicator PINT(3) set to 1.

The direct and indirect subsidiary subroutines of COMPL are RLINE, FVERB, BACK, FORE, ECTNUM, CNUBR, CIFST, TRANS, ERRPP and SCHAR, which have been mentioned already, and IFTYP and KOMPL. The operations that these subroutines perform are described in the paragraphs that follow.

RLINE(XFRST): This puts a SNAP instruction into the common array STRING. It reads a card image, whenever necessary, into the array NCARD of symbolic length MAXCOL. The calling program COMPL initializes the argument XFRST to 0, before it calls RLINE for the first time. RLINE then alters XFRST to 2. RLINE replaces commas and slashes outside a quotation by spaces, and it concatenates material from successive cards in accordance with SNAP line break conventions. RLINE calls BACK to deal with statement labels (v.i.) and ERRPP to print error messages. RLINE sets the common variable KKK to point to the end of the statement in STRING. RLINE puts an EXECUTE statement into STRING if an end of file condition occurs.

FVERB(TRA,CMVC,TIFT,J,BBB,YESNO,KSTYP). This scans the statement in positions 1 through KKK of STRING for a command word. The command verbs of SNAP, and the words IF and OTHERWISE comprise the command words. The limit KKK and the array STRING are transmitted via common. When TIFT is 1, FVERB determines if the first word in STRING is a command word. If it is, FVERB sets TRA to the appropriate value to control the branching in COMPL, sets PROCB(IC) to the operation class code, and KSTYP to the operation modifier code, increases IC by 1, and sets III to point to the last character of the command word in STRING. If a command word is not found, TRA is set to 8.

If TIFT is 2, FVERB scans forward from character III in STRING for a command word, and sets YESNO to 1, 4 and 3 respectively, when it finds a command verb, the word OTHERWISE, and no command word respectively. In the first two cases, it also sets TRA,CMVC to the operation class code, and KSTYP, sets BBB to point to the last non-blank character before the command word, and J to point to the last character of this command word, in STRING.

BACK(TYPP,POSTT,FNAM,POINT): This is used to delimit and identify a name of label that ends at position KKK in STRING. A backward scan is performed and an error return is made if this crosses the position III. The scan limits III and KKK are transmitted via common. TYPP is set by the calling program (e.g., COMPL) to 0, when BACK is called to delimit a name that need not have been encountered previously, and to 1 otherwise (this convention is varied slightly when the name of a list or an element of a list is scanned - see below). POSTT is set by the calling program to 1, 2, 3 and 4 respectively when BACK is called to scan a portion of a statement that should contain (1) the name of a string, or the name of a list of strings (2) a referenced statement label, (3) a bracketed statement label, (4) the name of a quantity, or the name of a list of quantities. BACK either

(i) identifies the name that ends the scan field with one already in the name bank, of the appropriate nature.

(ii) finds that the name is not in the name bank yet, and adds it

(iii) finds that the name is not in the name bank yet and does not add it

In cases (i) and (ii) BACK sets the argument FNAM to 0, and sets the argument POINT and the common variables LLST and NADD as in the scheme that follows. In case (iii) BACK sets FNAM to 1, and LLST, NADD and POINT to 0.

type of name	LLST	NADD	POINT
unsubscripted string name	0	pointer to string directory, STRDIR	0
unsubscripted quantity	0	pointer to number bank, NUMBK	0
name of list of strings	1	pointer to string list directory ALIST	0
element of list of strings	1	pointer to string list directory ALIST	subscript(v.i.)

type of name	LLST	NADD	POINT
name of list of quantities	-1	pointer to quantity list directory BLIST	0
element of list of quantities	-1	pointer to quantity list directory BLIST	subscript (v.i)

When a list element is specified by numerical subscript, (e.g., THE 7-TH GLONK) POINT is set to the value of this subscript. When a literal subscript is used (e.g., THE K-TH GLONK), POINT is set to the negative of the pointer to the relevant quantity name (i.e., K, in THE K-TH GLONK).

When BACK is called with POSTT set to 1 or 4, it scans backwards from position KKK until it reaches an article (A, AN, or THE), or certain other words, left over from some early experiments (THIS, THAT, OF, CALL), or an ordinal (recognized by an ending that consists of a hyphen or apostrophe and two letters. A name of the form THE g LIST is treated as the name of a list (A or AN could appear in place of THE). A name that begins with an ordinal is treated as the name of an element of a list.

When BACK is called with POSTT set to 2 or 3 it transmits the label in STRING to the subroutine TRANS for action after eliding spaces and articles, except the first.

The precise conventions concerning TYPP are as follows. If an unsubscripted name is delimited, it is sought only amongst previously defined string names, when POSTT is 1, and amongst previously defined quantity names when POSTT is 4. If it is not found, and TYPP is 0, it is added to the name bank, as a string name, and as a quantity name respectively in these two cases. If TYPP is 1, and it is not found action (iii) is taken, i.e., FNAM is set to 1. If the name of a list, or an element is delimited, it is sought amongst previously defined string list names, when POSTT is 1, and amongst previously defined quantity list names when POSTT is 4. If it is not found, and TYPP is 0, it is added to the name bank, as a string list name, and as a quantity list name in these two cases respectively. If it is not found, and TYPP is 1 however, it is sought amongst the quantity list names, and string lists names in these two cases. If it is still unfound, it is added to the names of lists of strings in either event, and the variables LIST, NADD and POINT set to the values that represent it with this status, and FNAM set to 0.

FORE: This scans the string expression in positions III through KKK of the array STRING, and puts the SNAPIC equivalent into the procedure table, in accordance with the conventions of Section 2.

ECTNUM(AAA,BBB,XNUM1,XNUM2,N2): This scans the portion of STRING, that is bounded by positions AAA and BBB, for numbers and ordinals. It sets its last three arguments as follows:

N2	kind of expression scanned	XNUM1	XNUM2
1	contains no ordinals or numbers	undefined	
2	extract expression	left boundary	right boundary
	subscripted name	ordinal	
3	a number occurs at the end of the expression, but not within it	this number	
4	at least one number occurs within the expression (and possibly at the end as well)	this number	
5	an m-th AND SUBSEQUENT phrase occurs	m	
6	an m-th AND PRECEDENT phrase occurs	m	

ECTNUM treats a string of characters as a number, if it (i) begins the scan region, or is preceded by a blank, and (ii) ends the scan region or is followed by a blank, and (iii) consists entirely of numerals and perhaps a leading minus sign.

CNUBR(TRAN,TYT): This translates the portion of an arithmetic statement, that follows the opening verb, and occupies positions III through KKK of STRING into SNAPIC in the procedure table. It is called by COMPL with TRAN set to 0 for INCREASE and DECREASE, and to 2 for SET statements; and TYT set to 1, 2 and 9 for these three kinds of statement respectively. III and KKK are transmitted via common.

CIFST: This translates the portion of a conditional statement, that follows the word IF, and occupies positions III through KKK of STRING into SNAPIC in the procedure table. It calls FVERB to find the verb that starts the first action clause, and IFTYP to identify the kind of condition that this delimits. After encoding the condition, it progresses through a cycle that encodes each of the success clauses in turn, and then through a similar cycle for the fail clauses. In these cycles, it calls FVERB to find the next command word to delimit the clause under consideration, and then calls KOMPL to translate this clause. Appropriate delimiters are inserted after the codes for each clause.

IFTYP(BBB,TYPT,JJ1,JJ2,KYPT) scans the portion of STRING that is bounded by the common variable III that points to the end of the word IF, and BBB, that points to the beginning of the first command verb. It sets TYPT to 1, 2, 3 and 4 for input exhaustion, string comparisons that do not contain SUBSEQUENT/PRECEDENT phrases, number comparison, and SUBSEQUENT/PRECEDENT type string comparisons respectively. In this last instance, KYPT is set to -1 when the SUBSEQUENT/PRECEDENT phrase precedes the verb IS (or ARE) and to 1 when it follows. The subroutine sets JJ1 to point to the character preceding the relationship verb (e.g., IS, IS THE SAME AS, IS EQUAL TO), and sets JJ2 to point to its last character. IFTYP also puts the codes that indicate the kind of condition into the procedure table in accordance with the SNAPIC conventions of Section 2. CIFST then branches on the value of TYPT that IFTYP returned.

KOMPL(TRA,KSTYP): This controls the translation of a clause within a conditional statement. Its operation is very similar to COMPL. It is called by CIFST, after FVERB has identified the verb, and set III and KKK to the boundaries of the portion of STRING that contains the rest of the clause. The switching parameter TRA, and the operation modifier code KSTYP that FVERB found are provided as arguments. KOMPL is a replica of the portion of COMPL from the point at which it branches on TRA to the end, with four obvious minor changes that give error returns if EXECUTE, PROCEDURE, CONTROL or inner conditional statements are encountered.

NAMES(TYPP,BRA,FNAM): This subroutine stores the names of strings, quantities, and lists in the name bank NBANK. It also determines whether a name, that a scan just delimited, is in NBANK already. Another subroutine, TRANS also uses NBANK to store statement labels.

NAMES is usually called by a subroutine (e.g., BACK, FORE, etc.) that has delimited the name of a string, or a quantity, or a list, by reference to context, in the array STRING, and copied it backwards into the array WORD, left adjusted. This array, and II, the length of the name plus 1, are transmitted through block common. The calling subroutine sets the first argument TYPP to 0, when the name need not be in NBANK yet, and to 1 when its absence would be an error. The second argument, BRA, controls the operation of NAMES as follows:

BRA	action	BRA	action
1	process an unsubscripted string name	4	process the name of a list of strings
2	process an unsubscripted quantity name	5	process the name of a list of quantities
3	print the contents of NBANK and the associated tables (v.i.)	6	initialize NBANK and the associated tables

NAMES, sets the third argument FNAME to 0 when the name in WORD is already in NBANK, and to 1 otherwise.

NUMK, the writing pointer to NBANK, is initialized by COMPL to 1, and the contents of NBANK are identified further by the following pointers.

kind of name	number in name bank	pointer to beginning of M-th name of this kind	pointer to end of M-th name of this kind
unsubscripted string	NUM	NAMNS(M)	NAMNE(M)
unsubscripted quantity	NUB	NBTAS(M)	NBTAE(M)
list of strings	ANAM	ANAMS(M)	ANAME(M)
list of quantities	BNAM	BNAMS(M)	BNAME(M)

TRANS(POSTT) deals with statement labels. The calling subroutine, (COMPL or BACK) sets the argument POSTT, to direct TRANS as follows:

POSTT=1 a REPEAT or CONTINUE statement is being processed, and account must be taken of its reference to a statement label.

POSTT=2 not now used, but entirely equivalent to 1.

POSTT=3 the bracketed label that precedes a statement has been encountered.

POSTT=4 the first pass of the translation is complete, and transfer points must be put into the SNAPIC codes for instructions that transfer control, which the first pass left incomplete.

POSTT=5 print the transfer tables, for testing purposes.

TRANS puts statement labels into the name bank NBANK, which is also used by the subroutine NAMES. NAM is the number of labels in NBANK; NAMTS(M) and NAMTE(M) point to the ends of the M'th label in NBANK, and NAMIC(M) points to the beginning in the procedure table PROCB, of the statement that is labelled by the M'th label in the name bank, when this is known.

When a referenced label is processed, the words WITH and FROM are bypassed. Spaces are elided when labels are stored in the name bank. When a bracketed statement label is processed by TRANS, a pointer to where it starts, in the procedure table, is put in the array NAMIC. When a statement that refers to a label which has not been encountered yet in brackets is processed, -(600+NAM) is stored as a temporary transfer pointer in the procedure table and in NAMIC. NAM enumerates the labels in the name bank. In the second pass, integers in the procedure table less than -600 are recognized as temporary pointers, and overwritten. Any still unassigned in NAMIC are interpreted as RETURN TO THE BEGINNING.

SCHAR(TYP,A,B): This is called by the main program, and by subroutines in the translator and interpreter, that set TYP to -1, 0 and 1 respectively. When TYP is -1, SCHAR initializes the variables in the common blocks CHARC and NUBTB. When TYP is 0, SCHAR sets B to the SNAPIC equivalent of the EBCDIC code A. When TYP is 1, SCHAR sets B to the EBCDIC equivalent of the SNAPIC code A.

ERRPP(KRROR): This prints error messages that the argument specifies. It initiates an error count when KRROR is 99, and calls EXIT when more than 50 errors have occurred.

6. The overall operation of the interpreter

The interpreter consists of the subroutine SNAP3, and its subsidiaries that are listed below. SNAP3 is called by the main program PIET. Its overall effect is to execute a SNAP procedure that is represented in SNAPIC code in the procedure table PROCB.

SNAP3 begins by printing the SNAPIC representation of the procedure, if this was requested by a control card, and initializes IC, the pointer to the procedure table. It then progresses repeatedly through a major cycle of operations that interprets a SNAPIC instruction. At the beginning of this cycle, IC always points to the operation class code that begins the instruction in the procedure table. The subroutine branches on this code to either (i) initialize, (ii) terminate, (iii) interpret an executable SNAPIC instruction, or (iv) make an error return.

The SNAPIC initialization code makes SNAP3 clear several arrays described in Section 2, and several variables directly, and by calling the subroutine ABLST (see later).

The SNAPIC output, string synthesis and alteration (i.e. COPY, APPEND, DELETE and OVERWRITE), input, arithmetic, and tape control (i.e. SELECT and REWIND) statements are interpreted by calling the subroutines PRINT1, COPY1, READ1, ARTHM and TAPCT respectively. A conditional statement is executed by calling IFRT to test the condition and advance IC to the beginning of the first action clause that should be executed. SNAP3 then deals with this and any further relevant action clauses just as if they were unconditional statements. CALL and RESERVE statements are interpreted by SNAP3 directly, using the subroutine ABLST to deal with storage assignments for list elements. Control transfer statements (REPEAT and CONTINUE) are interpreted by merely by setting IC appropriately. SNAP3 calls ERROR to print error diagnostics, and returns control to the main program PIET when a SNAPIC terminate instruction is executed, or when an end of file condition is encountered by an input instruction, or when an irrecoverable error condition occurs.

The direct and indirect subsidiaries of SNAP3 are ABLST, ARTHM, COPY1, ERROR, IFRT, PRINT1, READ1 and TAPCT, mentioned already; BLKLO, PUSH and STORE, that are used widely by the interpreter: and the rather specialized subroutines EDJEK and VIDCP, called by TAPCT.

Most of the arrays, and the variables associated with them are stored in block common. The processes performed by the subroutines that SNAP3 calls are as follows:

BLKLO: This is called by the subroutines COPY1, READ1 and STORE, to increase the common variable IBLANK to point to the next free word in the string bank, after the free word to which IBLANK points when BLKLO is entered. This simply increases IBLANK by 1, except when space is being reused, and the end of a chained segment has been reached.

PUSH(KNULL): This is called by PRINT1, COPY1, IFRT and ARTHM, when an extract expression in the procedure table is being processed, with IC pointing to the precedence code that begins the expression. KNULL is set to 0 if the expression is valid, and to 1 if the ordinals specified for its ends are in the wrong numerical order. PUSH puts the ordinals of the ends of a valid extract in the push down lists PTT1 and PTT2. The corresponding entry in the array PTR is initialized to zero - its use is explained on the comments cards in the subroutine STORE. The pointer to the end of the extract expression in the procedure table is preserved in the push down list KCTAB (which stacks the current pointer to the procedure table whenever a string name or an extract expression is encountered in interpreting a string expression) and the pointer to this entry in KCTAB is preserved in the push down list PIP, that contains pointers to the entries in KCTAB that pertain to extract expressions only. The variables KC and PI give the current occupancy of the push down lists KCTAB and PIP.

STORE(PRT): This transfers strings of characters between different arrays, under control of its argument, and the common variables PRO, OUT and SWT1. The string is put into the string bank STRBNK, when OUT is 1, and into the output buffer OBUF and the auxiliary buffers STRN1 and STRN2 when OUT is 2 and PRT is 1, 2 and 3 respectively. The writing pointers to the string bank, output buffer and auxiliary buffers are K, LCOL, STK1 and STK2 respectively, and are transmitted via common. When a string is put into the string bank, new space is used if SWT1 is 1, and old space is reused if SWT1 is 0. The nature of the string that is put into the string bank or a buffer is set by PRO in accordance with the scheme that follows. (The pointers I, IC and BFIL are transmitted via common).

PRO	nature of string
1	already resident in the string bank, starting at position I
2	defined in the procedure table, starting at position IC
3	resident in the input buffer NCARD, starting at position I
4	code representing line feed
5	decimal representation of a quantity specified by the unsubscripted name, and whose precedence code is in position IC of the procedure table.
6	decimal representation of a SNAPIC number, similarly located.
7	decimal representation of a quantity specified by a subscripted name, and whose value is in position BFIL of the subscripted quantity bank.
8	comma, to be used as a separator in a list.

READ1: This is called by SNAP3 to read a logical record into the input buffer ICARD (for cards) or TRECD (for tape), translate it to SNAPIC in the buffer NCARD, and then transfer this to the string bank, when the common variable XEOC is 0. SNAP3 initializes XEOC to 0, but its value is changed to 1 when the subroutine IFRT omits a record while testing for an end of file condition. When XEOC is 1, READ1 translates the record that already is in the input buffer (as the result of IFRT) into SNAPIC in the buffer NCARD, and then transfers this to the string bank. The writing pointer to the string bank is the common variable K.

PRINT1: This is called by SNAP3 to emit the logical record represented by the string expression in the procedure table that begins at position IC+1. The output device is specified by the operation modifier code at position IC. The output record is put into the output buffer OBUF first and then emitted.

COPY1: This is called by SNAP3 to put a string into the string bank, for a COPY statement, or to alter a string that is already there for an APPEND, DELETE or OVERWRITE statement. IC points to the operation modifier code in the SNAPIC statement in the procedure table.

IFRT: This is called by SNAP3 to test the condition in an IF statement, and to advance the procedure table pointer IC to the beginning of the first success clause, or to the beginning of the first fail clause, depending on whether the condition is satisfied. The common variable XEOC is set to 1 when IFRT tests for an end of file (see READ1).

ARTHM: This is called by SNAP3 to execute the arithmetic statement in the procedure table, to whose operation modifier code IC points.

ABLST (ANT,KKP,ABTYP,ABTYL,KRR): This is called (i) to initialize the variables and arrays that deal with lists, and (ii) to find the pointer to the entry in the subscripted string directory or the subscripted quantity bank, for a particular element of a list of strings or quantities that is under consideration. The calling program provides values of the first four arguments and some common variables that ABLST modifies. ANT is set to -1 when the pointer to the entry for the last element in a list is needed; to 0 when entries are to be created for entries with subscripts greater than those used previously for a list (or for a new list); and to 1 when the subscript of the element under consideration should be within the range of those of elements already in existence.

KKP is set to the subscript when an element of specified subscript is under consideration, and is irrelevant otherwise.

ABTYP is set to -1 when the pointer to the entry for an existing element of given subscript is sought; to 0 when an element is being defined, and the list may have to be extended to accommodate it; and to 1 when a list is being processed, and the calling program is seeking pointers to the entries for successive elements of the list in turn.

ABTYL is set to -1 to initialize the arrays and variables that deal with lists; and to 0 and 1 respectively when an element of a list of strings or quantities is under attention. The common variables AFIL and BFIL point to the entries in the subscripted string directory and the subscripted quantity bank respectively for the elements of lists of strings and quantities that are under consideration. The calling program sets AFIL or BFIL to point to the entry for the first element of the list (by reference to the list directories ALIST and BLIST) before ABLST is called, and ABLST alters AFIL, or BFIL, to point to the entry for the element of interest.

KNR is set by ABLST to 0 normally, and to 1 when the entry for the last element is found, in a deliberate search for it, or in an unsuccessful search for the entry for an element with a given subscript.

TAPCT(TYP,TOP,TK,RET): This performs I/O operations for SELECT, REWIND, FETCH and WRITE statements, that at present are used for magnetic tape I/O, but which can be extended to disc and other data sets by defining appropriate DATAD or DD macros. The calling program sets TYP to -1 and 1 to initialize and finalize respectively (the other arguments are then irrelevant), and to 0 for all other purposes. TOP is set to 1 to test for an end of file; to 2 to immit a record into the buffer TRECD for a FETCH statement; to 3 to put the EBCDIC character corresponding to the SNAPIC code TK into the image of an output record that is being formed in the output buffer, for a WRITE statement and to write it if it has reached the requisite length; to 4 to write the contents of OBUF, even though its contents have not reached this length; and to 5 and 6 to execute SELECT and REWIND statements for the data set reference number TK. TAPCT sets RET to 0 except when an end of file is sought and found. An end of file during an attempted FETCH causes an error return. TAPCT sets TK to the record length when it reads a record.

7. Control Card Options

Control cards can be included in a SNAP procedure to request certain options that are listed below. A control card is punched with (i) the word CONTROL in columns 1 to 7, (ii) an integer, right adjusted to column 10, that specifies the option, (iii) the letter Y in column 15, and anything that the user wishes in columns 16 et seq, ended by a period. Any character except Y, or a space, in column 15, causes the default option, that is also provided by omitting the card. Only 8 options are provided by the processor at present (and some of these are defunct), but any integer in the range 1 to 20 will plant a flag in an array that is transmitted throughout the processor, and which could be tested by local program changes. The options are listed below, with the default actions in brackets.

Option Code	Action
2	Do (not) print SNAPIC procedure strings.
3	Accept a SNAP (SNAPIC) procedure*
4	Do not (Do) print debug data from translator subroutines
5	Do not (Do) print debug data from interpreter subroutines
6	Process a single (several) SNAP procedure(s)*
7	Special characters in input records will be treated as typographic codes (printing characters)
8	Discard (Retain) trailing blanks in input records

* Applies only to early time shared version.

8. The Prototype SNAP Language

The SNAP language that the prototype SNAP processor accepts differs slightly from the target language that is described

in references (1) and (2). All the elements of the target language, and many further features will be acceptable at a later juncture. The differences that exist at present, and some further details that were not mentioned in the earlier accounts, are summarized below. The information in reference (1) is taken for granted. The differences are so slight, and it is hoped temporary, that it seemed most expedient to present them in this manner rather than to provide an independent self-contained account. The subroutines that are responsible for the details discussed here are cited in brackets.

Spacing and Punctuation: SNAP procedures may be punched in free format, on cards, using an 029 keypunch. Outside a quotation, successive words in a SNAP statement must be separated by either (i) a space, (ii) a comma, or (iii) a card break. Redundant spaces are ignored. The continuation of a word from one card to the next is shown by a hyphen at the breakpoint on the first card. Commas are treated as spaces. (These matters are handled by the subroutine RLINE.) Within a quotation, the conventions described in references (1) and (2) apply, except that (i) the characters > and < are not used as case shift indicators, (ii) the character | is used as a case reversal indicator, within the convention that upper case is assumed at the beginning of each quotation, (iii) the character pair =+ represents a line break with suppression of vertical spacing before the next line is printed. (This is done by the subroutine STORE.) A space or card break must precede a left quote, and a space,card break, or period (in the case of an output statement) must follow a right quote. Space should be left after the period that ends a statement for clarity, but it is not needed by the processor. Students are advised initially to begin each statement on a separate card for clarity. A statement normally is followed (after any intervening spaces) by the first word of the next statement, or by the left bracket that precedes a statement label. Explanatory comments can be put in brackets before a statement, even though these are not referenced as a statement label, but this clutters the name bank in the translation process. The processor (actually RLINE) treats the last bracketed string between the periods that end successive statements as a statement label, and ignores any material between the end of the preceding statement and this label. Comments can be included this way, but it is not recommended.

A procedure can be headed by a sentence that begins with the word PROCEDURE. The processor ignores it. Spaces within a statement label are elided (by RLINE). The word AND may not be used at present before a clause in a conditional (IF) statement.

Names of Strings: One objective of SNAP is to allow almost complete freedom in the choice of names for strings and other objects. An overall SNAP syntax and scanning tactics that will allow this have been designed (reference (2), Chapter 11), but interim measures were adopted in the prototype processor that are somewhat restrictive. The intersection of what is acceptable to the processor and what is easily explained is as follows. A string may be given an unsubscripted name or a subscripted name. An unsubscripted name consists of an article (A,AN,THE) followed by a "word", or several "words" that are separated by spaces and/or card breaks. Each "word" consists of letters and digits, and also may contain apostrophes and hyphens (provided these are not followed by two letters and a space, as in an ordinal). No other characters should be used in a name.

Each word must begin with a letter, and none of the words may be A, AN, CALL, IN, IT, ITS, LIST, OF, THAT, THE, THEN, THIS (because of the subroutine BACK), nor AND (because of the subroutine FORE). Redundant spaces between words in a name are ignored. The article that begins an unsubscripted string name may be changed (as in READ AN ITEM, PRINT THE ITEM). The article in fact can be omitted from the direct object of a SNAP statement once the name has been introduced by a CALL, input, or COPY statement. For example, the following statements work: READ THE X. PRINT X.

The article is needed in the statement that defines the name, to delimit it in a backward scan from the end of the sentence. The article also can be omitted from the name that is defined in a CALL statement whose direct object consists of a quotation or is ended by one, as in CALL "HOCUS POCUS" X. PRINT X. But this is best left unmentioned to students.

A name that is used in the first comparand of a conditional (IF) statement may not contain the word IS or ARE. A name that is used in the second comparand, or in a success or fail clause of a conditional statement may not contain any of the SNAP command words:

ACCEPT, APPEND, CALL, CONTINUE, CONTROL, COPY, DECREASE, DELETE, EXECUTE, FETCH, FORM, IF, INCREASE, LINK, OTHERWISE, OVERWRITE, PERFORATE, PROCEDURE, PUNCH, READ, REPEAT, REQUEST, RESERVE, REWIND, SELECT, SET, TERMINATE, TO FORM, TYPE, WRITE (because of the subroutines CIFST and FVERB) nor begin with any of the condition words or phrases EXHAUSTED, IDENTICAL, EQUAL, LESS THAN, GREATER THAN, UNEQUAL, nor contain the word TO (because of the subroutine IFTYP). The exclusion of some of these words is vestigal. No name that is used in a comparand may contain the word SUBSEQUENT or PRECEDENT (because of IFTYP and CNUBR).

The word pairs LINE FEED and NULL STRING are allowed (by FORE) in string expressions as alternatives to "/" and "", and may not be used as names.

Names of Quantities: These also may be unsubscripted or subscripted. An unsubscripted quantity name consists of one or more words, (that satisfy the restrictions which apply to words that can be used in the names of strings), preceded by an article (A, AN, THE) if the user wishes. The article is not mandatory in any context, and can be changed, and omitted, arbitrarily, when a name is reused.

A quantity name that is used as the second operand of a unary SET statement may not begin with any of the function words SUM, DIFFERENCE, PRODUCT, QUOTIENT, REMAINDER, CEILING, LESSER, GREATER, LENGTH (because of the subroutine CNUBR).

Names of Lists and Elements: An entire list is given a name of the form A g LIST, AN g LIST, or THE g LIST, where g denotes the generic name. The choice of article is irrelevant and it may be varied for the same list within a procedure. The generic name g consists of one or more words that satisfy the same rules as the words that are used in unsubscripted string and quantity names. An individual element of a list is reference by a subscripted string or quantity name, of the form THE m-th g, where m-th denotes a numerical or literal ordinal derived from a decimal integer or a previously defined unsubscripted quantity name, and g denotes a generic name.

The first occurrence of the generic name, g, of the elements of a list of strings, in a procedure, must be in a statement of one of the following kinds:

(i) RESERVE SPACE FOR n ELEMENTS IN THE g LIST

(ii) CALL "..,...,...,.." THE g LIST .

(iii) READ THE g LIST.

(iv) RESERVE SPACE FOR R CHARACTERS IN THE m-th g.

(v) CALL .... THE m-th g.

(vi) COPY .... AND CALL IT THE m-th g.

(vii) READ THE m-th g.

(viii) FETCH input statements corresponding to (iii) and (vii) but with the verbs REQUEST in place of READ.

The words that are underlined in (i) and (iv) are optional. The letters n and R denote integers, and m-th denotes a numerical or literal ordinal. A COPY statement cannot be used at present to define a list, though COPY statements can be used to define individual elements. The processor permits the article THE to be replaced by A or AN in THE g LIST and THE m-th g in these statements, though there seems no reason to do so.

The first occurrence of the generic name, h, of the elements of a list of numbers, in a procedure, must be in a statement of either of the forms

(i) SET THE h LIST TO ...

(ii) SET THE m-th h TO ...

These make h a generic quantity name, even if in the form (i), only one element is specified, (i.e., defining the list to be of length 1). Once h has been declared a generic quantity name by a SET statement of the form (i) or (ii) above it may be used in further SET statements of these forms, and in place of g in RESERVE and input statements of the form displayed earlier for lists of strings. These alter its length automatically, if this is necessary, and treat the list elements as numbers. For example, executing the statements
SET THE ITEM LIST TO 0. READ THE ITEM LIST.
with an input card punched
25,17,304
would make THE 1-ST ITEM, THE 2-ND ITEM and THE 3-RD ITEM be subscripted quantity names, with the values 25,17 and 304.

A RESERVE, CALL, SET or input statement that defines or immits an entire list sets the length accordingly, as mentioned already. A CALL, COPY, SET or input statement that defines or immits a single element extends the length of the list to the value of the ordinal, if this exceeds the prior length, and makes the intervening elements zero or null.

A negative number cannot be included at present in a list that is defined or immitted in its entirety by an input statement (the sign is dropped by the subroutine READ1).

The conventions concerning lists that are given in the present note overlap those in references (1) and (2) considerably, but it seemed advisable to present them here in full.

Statement Labels: A statement label consists of one or more words. Articles (A, AN, THE) and spaces are elided (by RLINE) and ignored so that a label cannot consist of just articles. A label may not contain any of the SNAP command words, if it is referenced in a conditional statement because of the subroutines (CIFST and FVERB) nor begin with the word WITH or FROM (because of the subroutine TRANS).

Numbers: Errors result if a number in a SNAP statement is outside the range -65,536 to +65,536, or if the value of a quantity that has an unsubscripted or a subscripted name goes outside this range (because INTEGER*2 storage is used). The second operand of a binary SET statement (e.g., SET .. TO THE SUM OF ..) must be a quantity name and not a number (because of the subroutine ARTHM). The third operand may be a number or a quantity name. So may the second operand of a unary SET statement.

String Expressions: An extract expression must have the form
THE m-th CHARACTER OF s
or THE m-th THROUGH n-th CHARACTERS OF s
where s denotes an unsubscripted string name. Separate statements may define this name in terms of other kinds of string expression (e.g., subscripted string names, concatenated expressions, etc.).

Numbers and quantity names may not occur in a string expression that is used as a comparand (because of the subroutine IFRT). Neither may they occur in a string expression e that ends a statement of the form
SET ... TO e.
or
SET ... TO THE LENGTH OF e.

(This is because of the subroutine ARTHM). Numbers and quantity names may not occur either in the definitions of string names that are referenced directly or indirectly by string expressions in comparands or SET statements as just described. A string comparand may not contain an implicitly nested extract of depth greater than 2.

Forced line breaks: A slash immediately before a quote mark is ignored at present. It can be made to force a line break by typing an extra slash before the quote mark.

List length expressions: The interpreter is equipped to evaluate the SNAPIC expression for the length of a list. Unfortunately the formation of this from the SNAP expression has not yet been coded (it involves a minor addition to CNUBR).

Paper tape and console i/o: ACCEPT and REQUEST are entirely equivalent to READ, and TYPE and PERFORATE to PRINT at present (because of the subroutines READ1 and PRINT1 respectively).

Conditional copy action: An IF statement may not contain a COPY...CALL... pair of clauses, because of FVERB.

Extracts interpretedas numbers: The preposition IN must be used instead of OF in a statement of the form
SET n TO THE i-th THROUGH k-th CHARACTERS IN s.
where n, i-th, k-th and s stand for a quantity name, two ordinal adjectives, and a string name respectively.

References

(1) M. P. Barnett and W. M. Ruhsam, A Natural Language Programming System for Mechanical Text Processing, IEEE Transactions of Engineering Writing and Speech, Vol. EWS-11, No. 2, 45, 1968.

(2) M. P. Barnett, Computer Programming in English, Harcourt Brace and World, May 1969.

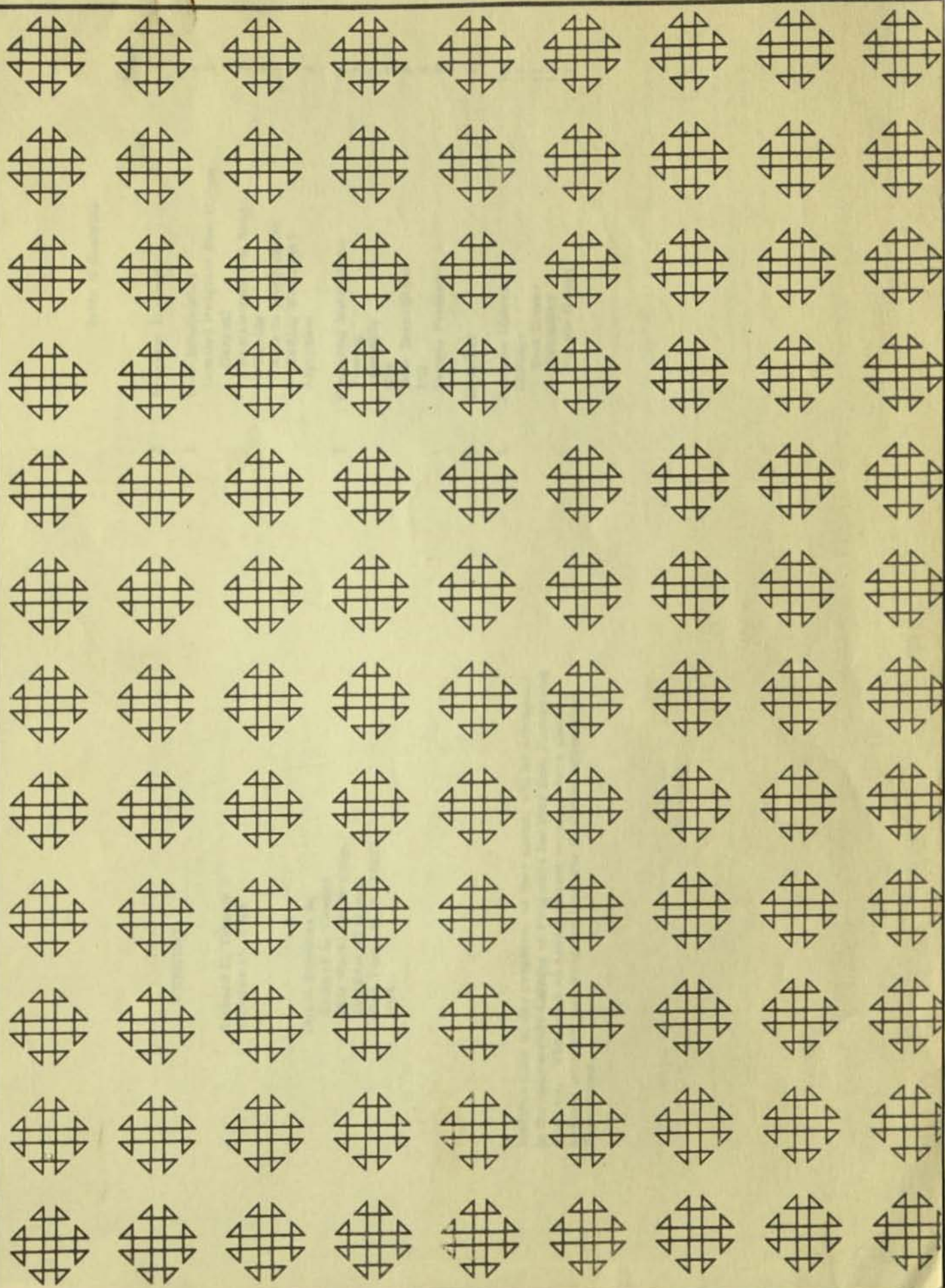
Table 1 Character codes

SNAPIC codes correspond primarily to EBCDIC codes. For conciseness, the SNAPIC codes are associated with the printer graphics that are given on the IBM System/360 Reference Data form no. X20-1703-5, that correspond to the relevant EBCDIC codes.

SNAPIC code	Printer graphics	SNAPIC code	Printer graphics
0	space	50	(less than)
1-9	1-9	51	(logical OR)
10	0	52	!
11-36	A-Z	53	:
37	+	54	(logical NOT)
38	-	55	(logical AND)
39	/	56	%
40	=	57	_
41	.	58	(greater than)
42	)	59	?
43	*	60	:
44	,	61	#
45	(	62	@
46	'	63	"
47	$	64	(lozenge)
48	&	65-90	a-z
49	¢		

360D-03.8.003

WEEKDAY



WEEKDAY

Richard L. Conner
October 15, 1966

Direct Inquiries to:
Richard L. Conner
IBM World Trade Corp.
201 East 42 Street
New York, N. Y. 10017
USA

Modifications to this program, as they occur, will be announced in the appropriate Catalog of Programs for IBM Data Processing Systems. When such an announcement occurs, users should order a complete new program from the Program Information Department.

Table of Contents

Program Abstract

WEEKDAY is a subroutine which will determine the day of the week for any date in the Gregorian calendar from AD October 15, 1582, to AD February 28, 4000, inclusive. It operates on any IBM System/360 that has the Standard and Decimal Instruction sets. It is invoked by the OS/TOS/DOS CALL statement - thus it can be used by programs coded in FORTRAN, COBOL, PL/I and Assembler languages. WEEKDAY occupies fewer than 160 bytes. It was coded in /360 Assembler Language as a serially reusable subroutine.

This program and its documentation were written by an IBM employee. It was developed for a specific purpose and submitted for general distribution to interested parties in the hope that it might prove helpful to other members of the data processing community. The program and its documentation are essentially in the author's original form. IBM serves as the distribution agency in supplying this program. Questions concerning the use of the program should be directed to the author's attention.

User Information

2.1 Detailed Program Description

2.1.1 General

WEEKDAY determines the day of the week for a given date in the Gregorian Calendar. The algorithm and the coding have been carefully planned to take maximum advantage of the IBM System/360 instructions, resulting in very low execution time.

WEEKDAY is coded in 360 DOS/TOS Assembler Language, which is upwards compatible with OS/360 Assembler Language; and it conforms to OS-TOS-DOS subroutine coding conventions.

2.1.2 Restrictions and Range

The Gregorian calendar was nowhere used before October 15, 1582 AD, so an argument representing an earlier date has little meaning. The definition of the Gregorian calendar after February 28, 4000 AD, is uncertain. For this reason, and because users will probably not be concerned with dates so far in the future, WEEKDAY is not designed to work with dates later than this.

WEEKDAY performs no checks for year within the above range, or for valid month or days of months. Invalid arguments will produce indeterminate results.

2.1.3 Timing

WEEKDAY takes about 2.3 milliseconds on a 360/30. Running time is largely independent of the argument date.

2.1.4 Aids to Modification

2.1.4.1 Conversion to reentrant code is possible with the addition of a third parameter specifying an 8-byte work area controlled by the calling program. This work area must begin on a double-word boundary. A free register must be designated for addressing this work area.

2.1.4.2 The argument format can be changed to packed decimal, with the date presented as a discontinuous number: oyyyymmdds. Changes would be required in the code that converts the argument values to binary, and to that which adjusts the year if the month is January or February. Similarly, the result can be returned as a packed decimal value, or through using it as an index, a string of letters can be returned.

2.1.4.3 WEEKDAY can easily be modified for a machine without the Decimal Instruction Set. Only two decimal instructions are used: the adjustment to the year for a month of January or February (SP), and the shift of the year prior to its conversion to binary (ZAP).

2.1.5 Calling WEEKDAY

2.1.5.1 WEEKDAY can be invoked via coding to produce a 360 calling sequence that passes two parameters "by name" - argument location, and result location, in that order. This coding is produced, for example, by the PL/I statement: CALL WEEKDAY (DATE, DAY). An entry point WEKDAY is provided for use by FORTRAN-coded callers, which are restricted to six-character names.

2.1.5.2 WEEKDAY has no error return, and it leaves resister 15 undisturbed.

2.2 Algorithm

The problem can be simplified in two ways:

a) Note that 364 ≡ 0 mod 7. Thus, if every year contained 364 days, any given date would always fall on the same day of the week. WEEKDAY uses an artificial year of 364 days, and applies a correction for the excess day in each Common Year.

b) If the intercalary day in a Leap Year were the last day of the year, calculation to account for it would be simplified. WEEKDAY uses an artificial year that begins on March 1.

These two simplifications are internal to the subroutine, so the user need not concern himself with them. They are, however, the foundation of the algorithm.

A brief review of the rules determining Gregorian Leap Years may be desirable. Here is a Leap Year Decision Table:

Year ≡ 0 mod 4	N	Y	Y	Y
Year ≡ 0 mod 100	-	Y	N	Y
Year ≡ 0 mod 400	-	N	-	Y
Common Year	X	X		
Leap Year			X	X

If these rules are applied, a small secular error in the Gregorian Calendar remains. No convention for correcting this exists; but it has been suggested that if a year is congruent to 0 mod 4000, it be made a Common Year. WEEKDAY ignores the error, because the correction will not be necessary until 4000 AD.

WEEKDAY determines the difference in days, mod 7, between a known date and the argument date by applying corrections for intervening years and intercalary days. The known date is the fictitious one of March 1, 0 AD. If the Gregorian Calendar went back that far, this date would have been a Wednesday. Days of the week are assigned serial numbers ranging from 0 for Sunday, through 6 for Saturday; so the serial number for the above base date is 3.

M(y,d)
D(y,d)
w(y,d)

(a+b) mod x = a mod x + b mod x

A formula for determining the day of the week (w) for a given year (y) month (m), and day of month (d) can be formed by applying the simplifying assumptions a) and b) above, and the Leap Year decision table:

$$w(y,m,d)=(w(0,3,1)+d-1+30n+f(m)+ax_1+bx_2+cx_3+x_4) \bmod 7 \qquad (1)$$

where y is reduced by one if m equals 1 or 2; and
n = number of months since preceding March;
f(m) = number of months with 31 days since preceding March;
$X_1 = \lfloor y/400 \rfloor$, the number of 400-year periods from 0 AD to y;
r_1 = y mod 400, the remainder of the division for X_1;
$X_2 = \lfloor r_1/100 \rfloor$, the number of centuries since the last preceding year that was an integral multiple of 400;
$r_2 = r_1 \bmod 100$, the remainder of the division for X_2;
$X_3 = \lfloor r2/4 \rfloor$, the number of Leap Years since the beginning of the century containing y;
X_4 = r2 mod 4, the number of Common Years between y and the last preceding Leap Year.

The coefficients a, b and c represent the number of days in excess of 364 ($z_i x_i$). The values, which are easily determined with the aid of the Leap Year Decision Table, are

a = 497	z_1 = 400
b = 124	z_2 = 100
c = 5	z_3 = 4

Formula (1) can be rewritten

$$w(y,m,d)=(d+2+30n+f(m)+497x_1+124x_2+5x_3+x_4) \bmod 7 \qquad (2)$$

Further, note that

$$497 \equiv 0 \bmod 7$$
$$124 \equiv 5 \bmod 7$$

so that (2) can be further simplified to

$$w(y,m,d) = (d+2(1+15n)+f(m)+5(x_2+x_3)+x_4) \bmod 7 \qquad (3)$$

Finally, define g(m) = 2(1+15n)+f(m), and construct a table:

m	n	2(1+15n)	+ f(m)		mod 7 = g(m)
1	10	302	6	308	0
2	11	332	7	339	3
3	0	2	0	2	2
4	1	32	1	33	5
5	2	62	1	63	0
6	3	92	2	94	3
7	4	122	2	124	5
8	5	152	3	155	1
9	6	182	4	186	4
10	7	212	4	216	6
11	8	242	5	247	2
12	9	272	5	277	4

Using the tabular values for g(m), formula (3) becomes

$$w(y,m,d) = (d+g(m) + 5(x_2+x_3) + x_4) \bmod 7 \qquad (4)$$

which is the formula implemented in the coding.

Example:

Determine w(1941, 12, 7)

d = 7
g(12) = 4

$x_2 = \lfloor (y \bmod 400)/100 \rfloor = \lfloor 341/100 \rfloor = 3$
$r_2 = r1 \bmod 100 = 41$
$x_3 = \lfloor r2/4 \rfloor = \lfloor 41/4 \rfloor = 10$
$x2+x3 = 13$
$5(x_2+x_3) = 65$
$x_4 = r_2 \bmod 4 = 41 \bmod 4 = 1$
77
mod 7 ≡ 0

Operating Instructions

3.1 Testing

Amalgamate the distributed decks with the appropriate control cards, and run as a job under OS/TOS/DOS. Figure 1 shows the deck setup used with DOS to generate the distributed output.

3.2 Assembly

Amalgamate the WEEKDAY source program (WKDAY in cc73-77) with appropriate OS/DOS/TOS control cards and run as an assembly job.

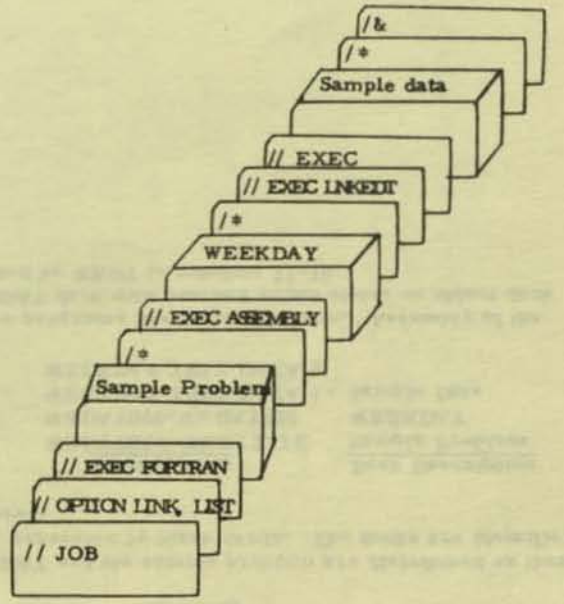
3.3 Use

At the user's option, include WEEKDAY in a job or job step input stream, in source or object form, either directly or as a cataloged entity.

WEEKDAY is invoked by the subroutine linkage appropriate to the calling program, e.g., in an Assembler language main program: CALL WEEKDAY (DATE, DAY). The entry point WEKDAY exists to provide a six-character name as required by FORTRAN.

WEEKDAY operates entirely within the main frame, so no machine setup, volume mounting, etc., are required.

Figure 1. Deck setup for distributed WEEKDAY output
Distributed cards are those with no / in column 1.



3.4 Data Description

The argument to WEEKDAY is an eight-byte zoned decimal field containing year, month and day of month: yyyymmdd. No boundary alignment is necessary. The result returned by the subroutineis a one-byte zoned decimal field containing the day of the week: w. Meaningful values for the argument and result are:

yyyy - 1582-4000
mm - 01-12
dd - 01-31
w - 0-6, with zero representing Sunday.

3.5 I/O

WEEKDAY performs no I/O.

3.6 Sample Problem

The distributed sample problem consists of a FORTRAN-coded driver that reads argument dates and their associated days of the week from FORTRAN data set 1. The driver calls WEEKDAY and exhibits the input and WEEKDAY's results on FORTRAN data set 3. The driver can be compiled by any OS/TOS/DOS FORTRAN compiler.

If the user provides his own sample data, he should punch a non-zero digit in column 41 of the last input card.

Deck Key

4.1 WEEKDAY and the sample problem are distributed as three decks, separated by blank cards. The decks are identified as follows:

CC	Identification	Deck Description	No. of Cards
73-80	WKDYTST0-WKDYTSTE	Sample Problem	15
73-80	WKDAY000-WKDAY820	WEEKDAY	82
62-80	WEEKDAY TEST DATA 1 - WEEKDAY TEST DATA 9	Sample Data	9

The two programs are in source form. Assembly of the WEEKDAY deck with punched output yields an object deck identified by WKDY in columns 73-76.

Systems Material

5.1 The code is straightforward, with no decision points; and the assembly listing is thoroughly annotated. Therefore, a flowchart would be superfluous.

The code evaluates the right-hand side of formula (5) given in the Algorithm section of this writeup. Before this, the elements of the argument are isolated and converted to binary. After the result has been obtain ed, it is converted to zoned format.

The main sections of code are:

Identification-Sequence		Action
WKDAY 320	WKDAY 420	Conversion of argument to binary, collection of D and g(m), and adjustment of y if m equals 1 or 2.
WKDAY 450	WKDAY 520	Calculation and addition of x_4, $5(x_2+x_3)$.
WKDAY 530	WKDAY 570	Calculation, formatting and storing of result.

```
// JOB WDTST
// OPTION LINK, LIST                                                    12.57.10
// EXEC FORTRAN

                        DISK OPERATING SYSTEM/360 FORTRAN   360N-FO-451 10

      C     DRIVER TO TEST WEEKDAY SUBROUTINE                                  WKDYTST0
                  DIMENSION  TEXT (5)                                          WKDYTST1
                  DOUBLE PRECISION  DATE                                       WKDYTST2
      1     WRITE  (3, 101)                                                    WKDYTST3
      101   FORMAT  ('1', T48, 'TEST OF WEEKDAY SUBROUTINE' ///               WKDYTST4
           1                          T44, 'CORRECT', T56, 'WEEKDAY' /       WKDYTST5
           2T2, 'INPUT', T21, 'DATE', T46, 'DAY',    T56, 'ANSWER' //         WKDYTST6
      2     READ (1, 102)  DATE, TEXT, RIGHT, LAST                             WKDYTST7
      102   FORMAT  (A8, 2X, 5A4, 4X, A1, 5X, I1)                             WKDYTST8
            CALL WEKDAY (DATE, ANS)                                          WKDYTST9
            WRITE (3, 103)  DATE, TEXT, RIGHT, ANS                           WKDYTSTA
      103   FORMAT  (T2, A8, T14, 5A4, T47, A1, T59, A1)                      WKDYTSTB
            IF (LAST) 4, 2, 4                                                WKDYTSTC
      4           CALL   EXIT                                                WKDYTSTD
                                      END                                    WKDYTSTE
```



```
      09/14/66              FORTMAIN                                    0002

                                  SCALARS

SYMBOL   LOCATION    SYMBOL   LOCATION    SYMBOL   LOCATION    SYMBOL   LOCATION    SYMBOL   LOCATION
DATE       0050      RIGHT      005C      LAST       00A8      ANS        00AC

                                  ARRAYS

SYMBOL   LOCATION    SYMBOL   LOCATION    SYMBOL   LOCATION    SYMBOL   LOCATION    SYMBOL   LOCATION
TEXT       0070

                              CALLED SUBROUTINES

IJTAAFR     IJTACOM     WEKDAY      IJTFXIT     EXIT

LABEL    LOCATION    LABEL    LOCATION    LABEL    LOCATION    LABEL    LOCATION    LABEL    LOCATION
00001      0078      00101      00B8      00002      00F0      00102      0128      00103      01B4
00004      01AA
        COMPILATION COMPLETE    AMOUNT OF COMMON 000000    AMOUNT OF CORE 000544    ADDRESS BASE TABLE    0180
```

```
// EXEC ASSEMBLY
```

13

```
                                EXTERNAL SYMBOL DICTIONARY                PAGE    1

 SYMBOL   TYPE ID   ADDR   LENGTH LD ID

WEEKDAY   SD   01 000000 000048
WEKDAY    LD      000000         01
```

14

```
  LOC  OBJECT CODE     ADDR1 ADDR2  STMT   SOURCE STATEMENT                                  D022APR66 09/14/66

000000                              2 WEEKDAY  START                                                    WKDAY010
                                    3 ********************************************************************** WKDAY020
                                    4 *                                                                  * WKDAY030
                                    5 *        THIS SEPIALLY-REUSABLE SUBROUTINE DETERMINES THE DAY OF THE  * WKDAY040
                                    6 *   WEEK FOR ANY GREGORIAN DATE FROM OCTOBER 15, 1582, THROUGH FEBRU- * WKDAY050
                                    7 *   ARY 28, 4000.                                                  * WKDAY060
                                    8 *                                                                  * WKDAY070
                                    9 *        CALL  WEEKDAY(DATE,DAY)                                   * WKDAY080
                                   10 *              WHERE DATE NAMES THE ARGUMENT AND DAY NAMES THE ANSWER.* WKDAY090
                                   11 *              THEY HAVE THE FOLLOWING FORMATS, RESPECTIVELY -       * WKDAY100
                                   12 *DATE    DS    0ZL8                                                  * WKDAY110
                                   13 *YEAR    DS    ZL4  YEAR - ANY VALUE 1582-4000                       * WKDAY120
                                   14 *MONTH   DS    ZL2  MONTH - ANY VALUE 01-12                          * WKDAY130
                                   15 *D       DS    ZL2  DAY OF MONTH - ANY VALUE 01-31                   * WKDAY140
                                   16 *DAY     DS    ZL1  DAY OF WEEK - 0(SUNDAY)-6(SATURDAY)              * WKDAY150
                                   17 *                                                                  * WKDAY160
                                   18 ********************************************************************** WKDAY170


                                   20          ENTRY WFKDAY   *** PSEUDONYM FOR FORTRAN CALLERS ***          WKDAY190
000001                             21 CORR     EQU   1    ACCUMULATOR FOR CORRECTIONS                     WKDAY200
000008                             22 BINMTH   EQU   8    MONTH IN BINARY FORM                            WKDAY210
000009                             23 BINYR    EQU   9    YEAR IN BINARY FORM                             WKDAY220
000007                             24 X        EQU   7    ACCUMULATOR FOR INTERMEDIATE VALUE              WKDAY230
000009                             25 QOT      EQU   9    QUOTIENT                                        WKDAY240
00000F                             26 RMNDR    EQU   F    REMAINDER                                       WKDAY250
00000F                             27 DVDND    EQU   RMNDR .   DIVIDEND(OPERAND 1 FOR DIVISION)         WKDAY260

000000                             29          USING *,15      CALLER HAS LOADED GPR15                     WKDAY280
                                   30 WEKDAY   SAVE  (14,12)   SAVE ALL NON-LINKAGE GPR'S                 WKDAY290
                                   31** SYSTEM CONTROL AND BASIC DOS 360N-CL-453 CHANGE LEVEL 1-0
000000 90EC D00C             0000C 32+WEKDAY   STM   14,12,12+48(13)
000000                             33          USING INPUT,3   DATA BASE                                   WKDAY300
000004 5831 0000             00000 34          L     3,0(1)    PICK UP SOURCE ADDRESS                    WKDAY310
```

15


```
  LOC  OBJECT CODE     ADDR1 ADDR2  STMT   SOURCE STATEMENT                                  D022APR66 09/14/66

000008 F231 F07C J00A 0007C 00008   35          PACK  DECMY,INDAY                                         WKDAY320
00000F 4FB0 F07A              0007A 36          CVB   CORR,DECDATE                                       WKDAY330
000012 F211 F07F 1004 0007F 00004   37          PACK  DECMTH,INMTH                                        WKDAY340
000018 4F90 F079              00078 38          CVB   RINMTH,DECDATE                                     WKDAY350
00001C 4199 F07F              0007F 39          IC    RINMTH,FFTBL-1(RINMTH)   FIND FUDGE FACTOR         WKDAY360
000020 1A19                         40          AR    CORR,RINMTH              AND APPLY IT TO CORRECTION WKDAY370
000022 F235 F07C 1003 0007C 00000   41          PACK  DECMY,INMY                                          WKDAY380
000028 F030 F07C F09A 0007C 0009A   42          SP    DECMY,=P'3'     ADJUST YEAR IF JANUARY OR FEBRUARY  WKDAY390
00002E 960F F07E              0007E 43          OI    YRSIGN,15       LEGITIMIZE SIGN FOR ZAP             WKDAY400
000032 FA32 F07C F07C 0007C 0007C   44          ZAP   DECMY,DECYR                                         WKDAY410
000038 4F90 F07A              0007A 45          CVB   RINYR,DECDATE                                      WKDAY420
00003C 1A8A                         46          SR    DVDND,DVDND                                       WKDAY430
00003E 4170 0003              00003 47          LA    X,3  USED TO FIND CENTURY MOD 4                   WKDAY440
000042 5D80 F090              00090 48          D     DVDND,=F'100'  FIND CENTURY                       WKDAY450
000046 1497                         49          NR    QOT,X      CENTURIES MOD 4                        WKDAY460
000048 147A                         50          NR    X,RMNDR    THIS MANY COMMON YEARS                 WKDAY470
00004A 8A40 0002              00002 51          SRA   RMNDR,2          AND                              WKDAY480
00004E 1A9B                         52          AR    QOT,RMNDR              THIS MANY LEAP YEARS       WKDAY490
000050 4C90 F098              00098 53          MH    QOT,=H'5'                                         WKDAY500
000054 1A89                         54          AR    CORR,QOT                                          WKDAY510
000056 1A17                         55          AR    CORR,X                                            WKDAY520
000058 1BAA                         56          SR    CORR-1,CORR-1  PREPARE TO DIVIDE                  WKDAY530
00005A 5D40 F094              00094 57          D     CORR-1,=F'7'                                      WKDAY540
00005E 5831 0004              00004 58          L     3,4(1)     PICK UP SINK ADDRESS                   WKDAY550
000062 42A0 3000              00000 59          STC   CORR-1,DAY                                        WKDAY560
000066 96F0 3000        00000       60          OI    DAY,X'F0'       PLUG IN BLANK ZONE                WKDAY570
00006A 92FF D00C              0000C 61          MVI   12(13),X'FF'    ASSIST DEBUGGING                  WKDAY580
                                   62          RETURN (14,12)  BACK TO CALLER                            WKDAY590
                                   63** SYSTEM CONTROL AND BASIC DOS 360N-CL-453 CHANGE LEVEL 1-0
00006E 98EC D00C              0000C 64+         LM    14,12,12+48(13)
000072 07FE                         65+         BR    14
```

16

```
  LOC  OBJECT CODE     ADDR1 ADDR2  STMT   SOURCE STATEMENT                                  00274PR6A 09/14/AA
                                     67 *              DATA DEFINITIONS                                  WKDAY610
                                     68 *                                                                WKDAY620
000078                               69          DS    0D   ALIGNMENT FOR CVB                           WKDAY630
000078                               70 DECDATE  DS    0PL4      BUCKET FOR PACKED DATE                 WKDAY640
000078 00000000                      71          DC    4XL1'0'                                          WKDAY650
00007C                               72 DECMY    DS    0PL4      MONTH AND YEAR 0YYYYMM$                WKDAY660
00007C                               73 DECYR    DS    PL3       YEAR 0YYYY$                            WKDAY670
00007F                               74          ORG   DECYR+2                                          WKDAY680
00007F                               75 DECMTH   DS    0PL2      MONTH 0MM$                             WKDAY690
00007F                               76 YRSIGN   DS    PL1                                              WKDAY700
00007F                               77          DS    PL1                                              WKDAY710
                                     78 *                                                               WKDAY720
000080 0003020500030501              79 FFTBL    DC    FL1'0,3,2,5,0,3,5,1,4,6,2,4'  FUDGE FACTORS     WKDAY730
                                     80 *                                                               WKDAY740
000000                               81 INPUT    DSECT     SOURCE AND SINK FORMATS                      WKDAY750
000000                               82 DAY      DS    0ZL1      RESULT                                 WKDAY760
000000                               83 DATE     DS    0ZL6      ARGUMENT                               WKDAY770
000000                               84 INMY     DS    0ZL6                                             WKDAY780
000000                               85          DS    ZL4                                              WKDAY790
000004                               86 INMTH    DS    ZL2                                              WKDAY800
000006                               87 INDAY    DS    ZL2                                              WKDAY810
                                     88          END                                                    WKDAY820
000090 00000064                      89                *F'100'
000094 00000007                      90                *F'7'
000098 0005                          91                *H'5'
00009A 3C                            92                *P'3'
```

17


```
SYMBOL    LEN  VALUE  DEFN
BINMTH    00001 000009 0027   0039  0039  0039  0040
BINYR     00001 000009 0021   0045
CURR      00001 000008 0021   0036  0040  0054  0055  0056  0056  0057  0059
DATE      00008 000000 0091
DAY       00001 000000 0082   0059  0060
DECDATE   00008 00007B 0070   003A  003B  0045
DECMTH    00002 00007F 0075   0037
DECMY     00004 00007C 0072   0035  0041  0042  0044
DECYR     00003 00007C 0073   0044  0074
DVDND     00001 000008 0027   004A  004B  0049
FFTBL     00001 000080 0079   0033
INDAY     00002 000006 0087   0035
INMTH     00002 000004 0086   0037
INMY      00006 000000 0084   0041
INPUT     00001 000000 0081   0031
QOT       00001 000000 0025   0049  0052  0051  0054
QWNDR     00001 000008 0026   0027  0050  0051  0052
WEEKDAY   00001 000000 0007
WFKDAY    00004 000000 0032   0020
X         00001 000007 0024   0047  0048  0056  0055
YRSIGN    00001 00007E 0076   0043
```

NO STATEMENTS FLAGGED IN THIS ASSEMBLY

18

```
// EXEC LNKEDT
```



```
JOB  #DTST     09/14/66   DISK LINKAGE EDITOR DIAGNOSTIC OF INPUT

ACTION TAKEN     MAP
LIST   AUTOLINK    IJTAAFR
LIST   AUTOLINK    IJTACOM
LIST   AUTOLINK    IJTACON
LIST   AUTOLINK    IJTFIOS
LIST   AUTOLINK    IJTFXIT
LIST   ENTRY
```

```
09/14/66   PHASE   XFR-AD  LOCORE  HICORE  DSK-AD   ESD TYPE  LABEL     LOADED  REL-FR

        PHASE***   001858  001858  0038CF  24 0 2   CSECT     FORTMAIN  001858  001858

                                                   CSECT     IJTAAFR   001E18  001E18

                                                   CSECT     IJTACOM   001E88  001E88
                                                      ENTRY  IJTSAVE   002404

                                                   CSECT     IJTFXIT   003888  003888
                                                      ENTRY  EXIT      0038AE

                                                   CSECT     WEEKDAY   001D78  001D78
                                                      ENTRY  WEKDAY    001D78

                                                   CSECT     IJTACOM   002718  002718
                                                   *  ENTRY  FCVFI     002718
                                                   *  ENTRY  FCVFD     00271C
                                                   *  ENTRY  FCVEI     002720
                                                   *  ENTRY  FCVED     002724
                                                   *  ENTRY  FCVII     002728
                                                   *  ENTRY  FCVID     00272C
                                                   *  ENTRY  FCVDI     0028C0
                                                   *  ENTRY  FCVDD     002DE0

                                                   CSECT     IJTFIOS   002F80  002F80
                                                      ENTRY  UNITARE   0036DE
                                                   *  ENTRY  DDIOXXF   0034DF
                                                   *  ENTRY  GETUNTF   0032D6
                                                   *  ENTRY  OPENUNE   003324
                                                   *  ENTRY  SFTLGUF   0033F0
                                                   *  ENTRY  CCWNOIE   003680
                                                   *  ENTRY  DSKWTMF   00357C
                                                   *  ENTRY  ASNHUFE   00371C
                                                   *  ENTRY  FILTARE   003610
                                                      ENTRY  IJJCPDI   002F80
```



```
// EXEC
```

TEST OF WEEKDAY SUBROUTINE

INPUT	DATE	CORRECT DAY	WEEKDAY ANSWER
19390223	*** VIP BIRTHDAY ***	4	4
19411207	PEARL HARBOR	0	0
19660816	AUGUST 16, 1966	2	2
19671031	HALLOWE'EN, 1967	2	2
19690317	MARCH 17, 1969	1	1
19681012	COLUMBUS DAY, 1968	6	6
15830301	MARCH 1, 1583	2	2
32000301	MARCH 1, 3200	3	3
35670526	MAY 26, 3567	5	5

EOJ WDTST 12.58.54

```
CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS*.....START JOB 2265....BIN 602 .....J AHERN.................*CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS
CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS*.....START JOB 2265....BIN 602 .....J AHERN.................*CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS
CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS*.....START JOB 2265....BIN 602 .....J AHERN.................*CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS
CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS*.....START JOB 2265....BIN 602 .....J AHERN.................*CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS
CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS*.....START JOB 2265....BIN 602 .....J AHERN.................*CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS
CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS*.....START JOB 2265....BIN 602 .....J AHERN.................*CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS
CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS*.....START JOB 2265....BIN 602 .....J AHERN.................*CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS
CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS*.....START JOB 2265....BIN 602 .....J AHERN.................*CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS
CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS*.....START JOB 2265....BIN 602 .....J AHERN.................*CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS
CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS*.....START JOB 2265....BIN 602 .....J AHERN.................*CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS
CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS*.....START JOB 2265....BIN 602 .....J AHERN.................*CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS
CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS*.....START JOB 2265....BIN 602 .....J AHERN.................*CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS
CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS*.....START JOB 2265....BIN 602 .....J AHERN.................*CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS
CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS*.....START JOB 2265....BIN 602 .....J AHERN.................*CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS
CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS*.....START JOB 2265....BIN 602 .....J AHERN.................*CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS
CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS*.....START JOB 2265....BIN 602 .....J AHERN.................*CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS
CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS*.....START JOB 2265....BIN 602 .....J AHERN.................*CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS
CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS*.....START JOB 2265....BIN 602 .....J AHERN.................*CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS
CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS*.....START JOB 2265....BIN 602 .....J AHERN.................*CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS
CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS*.....START JOB 2265....BIN 602 .....J AHERN.................*CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS
CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS*.....START JOB 2265....BIN 602 .....J AHERN.................*CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS
CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS*.....START JOB 2265....BIN 602 .....J AHERN.................*CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS
CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS*.....START JOB 2265....BIN 602 .....J AHERN.................*CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS
CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS*.....START JOB 2265....BIN 602 .....J AHERN.................*CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS
CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS*.....START JOB 2265....BIN 602 .....J AHERN.................*CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS
CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS*.....START JOB 2265....BIN 602 .....J AHERN.................*CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS
CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS*.....START JOB 2265....BIN 602 .....J AHERN.................*CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS
CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS*.....START JOB 2265....BIN 602 .....J AHERN.................*CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS
CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS*.....START JOB 2265....BIN 602 .....J AHERN.................*CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS
CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS*.....START JOB 2265....BIN 602 .....J AHERN.................*CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS
CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS*.....START JOB 2265....BIN 602 .....J AHERN.................*CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS
CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS*.....START JOB 2265....BIN 602 .....J AHERN.................*CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS
CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS*.....START JOB 2265....BIN 602 .....J AHERN.................*CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS
CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS*.....START JOB 2265....BIN 602 .....J AHERN.................*CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS
CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS*.....START JOB 2265....BIN 602 .....J AHERN.................*CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS
CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS*.....START JOB 2265....BIN 602 .....J AHERN.................*CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS
CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS*.....START JOB 2265....BIN 602 .....J AHERN.................*CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS
CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS*.....START JOB 2265....BIN 602 .....J AHERN.................*CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS
CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS*.....START JOB 2265....BIN 602 .....J AHERN.................*CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS
CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS*.....START JOB 2265....BIN 602 .....J AHERN.................*CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS
CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS*.....START JOB 2265....BIN 602 .....J AHERN.................*CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS
CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS*.....START JOB 2265....BIN 602 .....J AHERN.................*CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS
CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS*.....START JOB 2265....BIN 602 .....J AHERN.................*CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS
CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS*.....START JOB 2265....BIN 602 .....J AHERN.................*CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS
CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS*.....START JOB 2265....BIN 602 .....J AHERN.................*CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS
CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS*.....START JOB 2265....BIN 602 .....J AHERN.................*CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS
CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS*.....START JOB 2265....BIN 602 .....J AHERN.................*CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS
CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS*.....START JOB 2265....BIN 602 .....J AHERN.................*CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS
CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS*.....START JOB 2265....BIN 602 .....J AHERN.................*CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS
CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS*.....START JOB 2265....BIN 602 .....J AHERN.................*CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS
CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS*.....START JOB 2265....BIN 602 .....J AHERN.................*CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS
CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS*.....START JOB 2265....BIN 602 .....J AHERN.................*CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS
CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS*.....START JOB 2265....BIN 602 .....J AHERN.................*CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS
CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS*.....START JOB 2265....BIN 602 .....J AHERN.................*CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS
CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS*.....START JOB 2265....BIN 602 .....J AHERN.................*CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS
CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS*.....START JOB 2265....BIN 602 .....J AHERN.................*CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS
CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS*.....START JOB 2265....BIN 602 .....J AHERN.................*CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS
CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS*.....START JOB 2265....BIN 602 .....J AHERN.................*CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS
CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS*.....START JOB 2265....BIN 602 .....J AHERN.................*CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS
CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS*.....START JOB 2265....BIN 602 .....J AHERN.................*CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS
```

```
//JEALSIT1  JOB  (T200,602,1,2),'J AHERN', MSGLEVEL=1                    JOB2265
C     DRIVER TO TEST WEEKDAY SUBROUTINE                                 WKDYTST0
                DIMENSION  TEXT (5)                                     WKDYTST1
                DOUBLE PRECISION  DATE                                  WKDYTST2
1     WRITE  (3, 101)                                                   WKDYTST3
101   FORMAT  ('1', T48, 'TEST OF WEEKDAY SUBROUTINE' ///               WKDYTST4
     1                          T44, 'CORRECT', T56, 'WEEKDAY' /       WKDYTST5
     2T2, 'INPUT', T21, 'DATE', T46, 'DAY',   T56, 'ANSWER' // )       WKDYTST6
2     READ (1, 102)  DATE, TEXT, RIGHT, LAST                            WKDYTST7
102   FORMAT  (A8, 2X, 5A4, 4X, A1, 5X, I1)                             WKDYTST8
      CALL WEKDAY (DATE, ANS)                                           WKDYTST9
      WRITE (3, 103)  DATE, TEXT, RIGHT, ANS                            WKDYTSTA
103   FORMAT  (T2, A8, T14, 5A4, T47, A1, T59, A1)                      WKDYTSTB
      IF (LAST) 4, 2, 4                                                 WKDYTSTC
4          CALL   EXIT                                                  WKDYTSTD
                                END                                     WKDYTSTE

WKDY      TITLE 'DAY OF WEEK SUBROUTINE - DICK CONNER, 212 MU6-4000 USA' WKDAY000
WEEKDAY   START                                                         WKDAY010
*****************************************************************     WKDAY020
*                                                               *       WKDAY030
*       THIS SERIALLY-REUSABLE SUBROUTINE DETERMINES THE DAY OF THE *   WKDAY040
*   WEEK FOR ANY GREGORIAN DATE FROM OCTOBER 15, 1582, THROUGH FEBRU- *  WKDAY050
*   ARY 28, 4000.                                               *       WKDAY060
*                                                               *       WKDAY070
*       CALL  WEEKDAY(DATE,DAY)                                 *       WKDAY080
*             WHERE DATE NAMES THE ARGUMENT AND DAY NAMES THE ANSWER.*  WKDAY090
*             THEY HAVE THE FOLLOWING FORMATS, RESPECTIVELY -   *       WKDAY100
*DATE     DS    0ZL8                                           *       WKDAY110
*YEAR     DS    ZL4  YEAR - ANY VALUE 1582-4000                *       WKDAY120
*MONTH    DS    ZL2  MONTH - ANY VALUE 01-12                   *       WKDAY130
*D        DS    ZL2  DAY OF MONTH - ANY VALUE 01-31            *       WKDAY140
*DAY      DS    ZL1  DAY OF WEEK - 0(SUNDAY)-6(SATURDAY)       *       WKDAY150
*                                                               *       WKDAY160
*****************************************************************       WKDAY170
         SPACE 3                                                        WKDAY180
         ENTRY WEKDAY   *** PSEUDONYM FOR FORTRAN CALLERS ***           WKDAY190
CORR     EQU   11   ACCUMULATOR FOR CORRECTIONS                         WKDAY200
BINMTH   EQU   9    MONTH IN BINARY FORM                                WKDAY210
BINYR    EQU   9    YEAR IN BINARY FORM                                 WKDAY220
X        EQU   7    ACCUMULATOR FOR INTERMEDIATE VALUE                  WKDAY230
QOT      EQU   9    QUOTIENT                                            WKDAY240
RMNDR    EQU   8    REMAINDER                                           WKDAY250
DVDND    EQU   RMNDR     DIVIDEND(OPERAND 1 FOR DIVISION)               WKDAY260
         SPACE 1                                                        WKDAY270
         USING *,15      CALLER HAS LOADED GPR15                        WKDAY280
WEKDAY   SAVE  (14,12)   SAVE ALL NON-LINKAGE GPR'S                     WKDAY290
         USING INPUT,3   DATA BASE                                      WKDAY300
         L     3,0(1)    PICK UP SOURCE ADDRESS                         WKDAY310
         PACK  DECMY,INDAY                                              WKDAY320
         CVB   CORR,DECDATE                  day of month               WKDAY330
         PACK  DECMTH,INMTH                                             WKDAY340
         CVB   BINMTH,DECDATE                month                      WKDAY350
         IC    BINMTH,FFTBL-1(BINMTH)   FIND FUDGE FACTOR               WKDAY360
         AR    CORR,BINMTH              AND APPLY IT TO CORRECTION      WKDAY370
         PACK  DECMY,INMY                100*year + month               WKDAY380
         SP    DECMY,=P'3'      ADJUST YEAR IF JANUARY OR FEBRUARY      WKDAY390
         OI    YRSIGN,15        LEGITIMIZE SIGN FOR ZAP                 WKDAY400
         ZAP   DECMY,DECYR       year relative to march1                WKDAY410
         CVB   BINYR,DECDATE                                            WKDAY420
         SR    DVDND,DVDND                                              WKDAY430

         LA    X,3  USED TO FIND CENTURY MOD 4                          WKDAY440
         D     DVDND,=F'100'  FIND CENTURY                              WKDAY450
         NR    QOT,X     CENTURIES MOD 4                                WKDAY460
         NR    X,RMNDR   THIS MANY COMMON YEARS                         WKDAY470
         SRA   RMNDR,2        AND                                      WKDAY480
         AR    QOT,RMNDR           THIS MANY LEAP YEARS                 WKDAY490
         MH    QOT,=H'5'                                                WKDAY500
         AR    CORR,QOT                                                 WKDAY510
         AR    CORR,X                                                   WKDAY520
         SR    CORR-1,CORR-1  PREPARE TO DIVIDE                         WKDAY530
         D     CORR-1,=F'7'                                             WKDAY540
         L     3,4(1)    PICK UP SINK ADDRESS                           WKDAY550
         STC   CORR-1,DAY                                               WKDAY560
         OI    DAY,X'F0'      PLUG IN BLANK ZONE                        WKDAY570
         MVI   12(13),X'FF'   ASSIST DEBUGGING                          WKDAY580
         RETURN (14,12)  BACK TO CALLER                                 WKDAY590
         EJECT                                                          WKDAY600
*             DATA DEFINITIONS                                          WKDAY610
         SPACE 3                                                        WKDAY620
         DS    0D   ALIGNMENT FOR CVB                                   WKDAY630
DECDATE  DS    0PL8       BUCKET FOR PACKED DATE                        WKDAY640
         DC    4XL1'0'                                                  WKDAY650
DECMY    DS    0PL4       MONTH AND YEAR 0YYYYMMS                       WKDAY660
DECYR    DS    PL3        YEAR 0YYYYS                                   WKDAY670
         ORG   DECYR+2                                                  WKDAY680
DECMTH   DS    0PL2       MONTH 0MMS                                    WKDAY690
YRSIGN   DS    PL1                                                      WKDAY700
         DS    PL1                                                      WKDAY710
*                                                                       WKDAY720
FFTBL    DC    FL1'0,3,2,5,0,3,5,1,4,6,2,4'  FUDGE FACTORS              WKDAY730
*                                                                       WKDAY740
INPUT    DSECT      SOURCE AND SINK FORMATS                             WKDAY750
DAY      DS    0ZL1       RESULT                                        WKDAY760
DATE     DS    0ZL8       ARGUMENT                                      WKDAY770
INMY     DS    0ZL6                                                     WKDAY780
         DS    ZL4                                                      WKDAY790
INMTH    DS    ZL2                                                      WKDAY800
INDAY    DS    ZL2                                                      WKDAY810
         END                                                            WKDAY820

19390223  *** VIP BIRTHDAY ***     4                  WEEKDAY TEST DATA 1
19411207  PEARL HARBOR             0                  WEEKDAY TEST DATA 2
19660816  AUGUST 16, 1966          2                  WEEKDAY TEST DATA 3
19671031  HALLOWE'EN, 1967         2                  WEEKDAY TEST DATA 4
19690317  MARCH 17, 1969           1                  WEEKDAY TEST DATA 5
19681012  COLUMBUS DAY, 1968       6                  WEEKDAY TEST DATA 6
15830301  MARCH 1, 1583            2                  WEEKDAY TEST DATA 7
32000301  MARCH 1, 3200            3                  WEEKDAY TEST DATA 8
35670526  MAY 26, 3567             5     9            WEEKDAY TEST DATA 9


H A S P  JOB STATISTICS --    110 CARDS READ --      110 LINES PRINTED --      0 CARDS PUNCHED --    0.00 MINUTES EXECUTION TIME
            I/O CALLS                    SVC CALLS        0.00 MINUTES CPU TM    TIME=  21:24:33 DATE=  09/17/68
```

CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS*.....END JOB 2265....BIN 602 .....J AHERN.....*CAMPUS*CAMPUS*CAMPUS*CAMPUS*CAMPUS