Box 50
Folder 12

102749013

T. Litt

PDP-11 COBOL User's Guide

digital

PDP-11

COBOL User's Guide

DEC-11-LCUGA-A-D

Order additional copies as directed on the Software Information page at the back of this document.

digital equipment corporation · maynard. massachusetts

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this manual.

The software described in this document is furnished to the purchaser under a license for use on a single computer system and can be copied (with inclusion of DIGITAL's copyright notice) only for use in such system, except as may otherwise be provided in writing by DIGITAL.

Digital Equipment Corporation assumes no responsibility for the use or reliability of its software on equipment that is not supplied by DIGITAL.

Copyright © 1974 by Digital Equipment Corporation, Maynard, Mass.

The HOW TO OBTAIN SOFTWARE INFORMATION page, located at the back of this document, explains the various services available to DIGITAL software users.

The postage prepaid READER'S COMMENTS form on the last page of this document requests the user's critical evaluation to assist us in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

CDP	DIGITAL	INDAC	PS/8
COMPUTER LAB	DNC	KA10	QUICKPOINT
COMSYST	EDGRIN	LAB-8	RAD-8
COMTEX	EDUSYSTEM	LAB-8/e	RSTS
DDT	FLIP CHIP	LAB-K	RSX
DEC	FOCAL	OMNIBUS	RTM
DECCOMM	GLC-8	OS/8	RT-11
DECTAPE	IDAC	PDP	SABR
DIBOL	IDACS	PHA	TYPESET 8
			UNIBUS

CONTENTS

		Page
CHAPTER 1	INTRODUCTION	1-1
1.1	THE HISTORY OF COBOL	1-1
1.2	THE PDP-11 COBOL COMPILER	1-4
1.3	THE COBOL SOURCE PROGRAM	1-6
1.3.1	The Identification Division	1-6
1.3.2	The Environment Division	1-6
1.3.3	The Data Division	1-7
1.3.4	The Procedure Division	1-7
1.4	THE COBOL UTILITY PROGRAMS	1-8
1.4.1	COBRG	1-8
1.4.2	REFORMAT	1-9
1.4.3	SORT	1-9
CHAPTER 2	TYPICAL USAGE	2-1
2.1	DEVELOPING A PROGRAM	2-1
2.1.1	Punched Cards	2-1
2.1.2	Console	2-2
2.2	REFERENCE FORMATS	2-2
2.2.1	The Conventional Format	2-3
2.2.2	The Terminal Format	2-5
2.3	LIBRARY FACILITY (COPY)	2-9
2.3.1	Creating a COBOL Library File	2-9
2.3.2	The COPY Statement	2-10
2.3.3	The COPY REPLACING Statement	2-14
2.3.3.1	Examples, COPY REPLACING	2-14
2.3.4	The Source Listing	2-16
2.3.4.1	Before the COPY Statement	2-16
2.3.4.2	After the COPY Statement	2-17
2.3.5	Common Errors, Library Facility	2-18
2.4	USING THE COMPILER	2-18
2.4.1	Calling the COBOL Compiler	2-18
2.4.2	COBOL Command String	2-19
2.4.2.1	Object Program File Specification	2-19
2.4.2.2	Listing File Specification	2-21
2.4.2.3	Source Program File Specification	2-21
2.4.2.4	Compiler Switches	2-22
2.4.2.5	Examples of the COBOL Command String	2-23
2.4.2.6	Common Entry Errors, COBOL Command String	2-24
2.5	EXECUTING THE OBJECT CODE	2-24
2.5.1	Compile-and-Go	2-24
2.5.2	Execute only	2-24
2.5.3	Common Errors, Object Code Execution	2-25

		Page
CHAPTER 3	NON-NUMERIC CHARACTER HANDLING	
3.1	INTRODUCTION	3-1
3.2	DATA ORGANIZATION	3-2
3.2.1	Group Items	3-2
3.2.2	Elementary Items	3-2
3.3	SPECIAL CHARACTERS	3-3
3.4	TESTING NON-NUMERIC FIELDS	3-4
3.4.1	Relation Tests	3-4
3.4.1.1	Classes of Data	3-5
3.4.1.2	The Comparison Operation	3-6
3.4.2	Class Tests	3-6
3.5	DATA MOVEMENT	3-7
3.6	THE MOVE STATEMENT	3-8
3.6.1	Group Moves	3-8
3.6.2	Elementary Moves	3-8
3.6.2.1	Edited Moves	3-10
3.6.2.2	JUSTIFIED Moves	3-10
3.6.3	Multiple Receiving Fields	3-11
3.6.4	Subscripted Moves	3-11
3.6.5	Common Errors, MOVE Statement	3-12
3.7	THE STRING STATEMENT	3-12
3.7.1	Multiple Sending Fields	3-13
3.7.2	The POINTER Phrase	3-14
3.7.3	The DELIMITED BY Phrase	3-14
3.7.4	The OVERFLOW Phrase	3-16
3.7.5	Subscripted Fields in STRING Statements	3-18
3.7.6	Common Errors, STRING Statement	3-20
3.8	THE UNSTRING STATEMENT	3-20
3.8.1	Multiple Receiving Fields	3-21
3.8.2	The DELIMITED BY Phrase	3-23
3.8.2.1	Multiple Delimiters	3-27
3.8.3	The COUNT Phrase	3-28
3.8.4	The DELIMITER Phrase	3-29
3.8.5	The POINTER Phrase	3-30
3.8.6	The TALLYING Phrase	3-32
3.8.7	The OVERFLOW Phrase	3-33
3.8.8	Subscripted Fields in UNSTRING Statements	3-34
3.8.9	Common Errors, UNSTRING Statement	3-36
3.9	THE INSPECT STATEMENT	3-36
3.9.1	The BEFORE/AFTER Phrase	3-37
3.9.2	Implicit Redefinition	3-39
3.9.3	The INSPECT Operation	3-41
3.9.3.1	Setting the Scanner	3-42
3.9.3.2	Active/Inactive Arguments	3-42
3.9.3.3	Finding an Argument Match	3-43
3.9.4	Subscripted Fields in INSPECT Statements	3-44
3.9.5	The TALLYING Phrase	3-44
3.9.5.1	The Tally Counter	3-45
3.9.5.2	The Tally Argument	3-45

			Page
	3.9.5.3	The Tally Argument List	3-46
	3.9.5.4	Interference in Tally Argument Lists	3-48
	3.9.6	The REPLACING Phrase	3-52
	3.9.6.1	The Search Argument	3-52
	3.9.6.2	The Replacement Value	3-54
	3.9.6.3	The Replacement Argument	3-54
	3.9.6.4	The Replacement Argument List	3-54
	3.9.6.5	Interference in Replacement Argument Lists	3-56
	3.9.7	Common Errors, INSPECT Statement	3-57
CHAPTER 4		NUMERIC CHARACTER HANDLING	4-1
	4.1	INTRODUCTION	4-1
	4.2	USAGES, DISPLAY/COMP	4-1
	4.2.1	Sign Conventions	4-2
	4.2.2	Illegal Values in Numeric Fields	4-3
	4.3	TESTING NUMERIC FIELDS	4-5
	4.3.1	Relation Tests	4-5
	4.3.2	Sign Tests	4-6
	4.3.3	Class Tests	4-7
	4.4	THE MOVE STATEMENT	4-7
	4.4.1	Group Moves	4-8
	4.4.2	Elementary Numeric Moves	4-8
	4.4.3	Elementary Numeric Edited Moves	4-10
	4.4.4	Common Errors, Numeric MOVE Statements	4-11
	4.5	THE ARITHMETIC STATEMENTS	4-11
	4.5.1	Intermediate Results	4-12
	4.5.2	The ROUNDED Phrase	4-13
	4.5.3	The SIZE ERROR Phrase	4-14
	4.5.4	The GIVING Phrase	4-15
	4.5.5	Multiple Operands in ADD and SUBTRACT Statements	4-15
	4.5.6	The ADD Statement	4-16
	4.5.7	The SUBTRACT Statement	4-16
	4.5.8	The MULTIPLY Statement	4-16
	4.5.9	The DIVIDE Statement	4-17
	4.5.10	Common Errors, Arithmetic Statements	4-18
CHAPTER 5		TABLE HANDLING	5-1
	5.1	INTRODUCTION	5-1
	5.2	DEFINING TABLES	5-1
	5.3	MAPPING TABLE ELEMENTS	5-2
	5.3.1	Initializing Tables	5-6
	5.4	SUBSCRIPTING AND INDEXING	5-7
	5.4.1	Subscripting with Literals	5-8
	5.4.1.1	Operations Performed by the Software	5-9
	5.4.2	Subscripting with Data-names	5-9
	5.4.2.1	Operations Performed by the OTS	5-10
	5.4.3	Subscripting with Indexes	5-10
	5.4.3.1	Operations Performed by the OTS	5-11
	5.4.3.2	Relative Indexing	5-12
	5.4.4	Index Data Items	5-12
	5.4.5	The SET Statement	5-13

			Page
CHAPTER 6		FILE HANDLING	6-1
	6.1	INTRODUCTION	6-1
	6.2	SEQUENTIAL FILE ORGANIZATION	6-2
	6.2.1	Format	6-2
	6.2.2	Record Size	6-3
	6.2.2.1	RECORD CONTAINS Clause	6-4
	6.2.2.2	Print-Controlled Records	6-4
	6.2.3	Record Blocking	6-5
	6.2.4	Device Restrictions	6-6
	6.3	RELATIVE FILE ORGANIZATION	6-6
	6.3.1	Record Size	6-7
	6.3.2	Record Blocking	6-8
	6.3.3	Media Restrictions	6-11
	6.4	CURRENT RECORD AREA	6-11
	6.4.1	SAME RECORD AREA Clause	6-11
	6.5	BUFFERING	6-12
	6.5.1	Buffer Size	6-13
	6.5.2	Double Buffering	6-13
	6.5.3	Sharing Buffer Space Among Files	6-13
	6.6	DEVICES, FILES, FILE-NAMES	6-14
	6.6.1	Using Explicit File-names (VALUE OF ID Clause)	6-16
	6.6.1.1	Switches	6-17
	6.6.2	Device Assignment by ASSIGN Clause	6-17
	6.7	COBOL I/O STATEMENTS	6-18
	6.7.1	Sequential I/O Statements	6-19
	6.7.1.1	Opening Sequential Files	6-19
	6.7.1.2	Reading Sequential Files	6-21
	6.7.1.3	Rewriting Records into Sequential Files	6-22
	6.7.1.4	Writing Sequential Files	6-22
	6.7.1.5	Closing Sequential Files	6-23
	6.7.2	Relative I/O Statements	6-24
	6.7.2.1	Access Modes	6-24
	6.7.2.2	Opening Relative Files	6-25
	6.7.2.3	Reading Relative Files	6-26
	6.7.2.4	Rewriting Records into a Relative File	6-27
	6.7.2.5	Writing Records in a Relative File	6-27
	6.7.2.6	Deleting Records from a Relative File	6-28
	6.7.2.7	Specifying the Next Record to be Read	6-28
	6.7.2.8	Closing Relative Files	6-29
	6.8	COMMUNICATING WITH THE PROGRAM	6-30
	6.8.1	Using the ACCEPT Statement	6-30
	6.8.2	Using the DISPLAY Statement	6-31
	6.9	FILE COMPATIBILITIES -- COBOL, FORTRAN, AND MACRO	6-32
	6.9.1	Writing Files for FORTRAN or MACRO	6-32
	6.9.2	Reading Files Written by FORTRAN or MACRO	6-33
	6.10	PROCESSING I/O ERRORS	6-34
	6.11	RUNTIME FILE I/O ERROR PROCEDURES	6-34
	6.12	RUN-TIME ERROR MESSAGES	6-38

		Page
CHAPTER 7	DEVICES	7-1
7.1	INTRODUCTION	7-1
7.2	DISK	7-2
7.3	DECTAPE	7-2
7.4	MAGNETIC TAPE	7-3
7.5	CARD READER AND LINE PRINTER	7-3
CHAPTER 8	GOOD PROGRAMMING PRACTICES	8-1
8.1	FORMATTING THE SOURCE PROGRAM	8-1
8.2	USE OF PUNCTUATION	8-5
8.3	USE OF THE ALTER VERB	8-5
8.4	USE OF THE PERFORM VERB	8-6
8.5	USE OF LEVEL 88 CONDITION NAMES	8-9
CHAPTER 9	PDP-11 COBOL UTILITY PROGRAMS	9-1
9.1	COBRG (COBOL REPORT GENERATOR)	9-1
9.1.1	Introduction	9-1
9.1.2	The Sample Problem	9-3
9.1.3	The Specification Lines	9-3
9.1.3.1	NAME Specification	9-4
9.1.3.2	INPUT Specifications	9-5
9.1.3.3	OUTPUT Specifications	9-6
9.1.3.4	HEADER Specifications	9-7
9.1.3.5	BREAK Specifications	9-8
9.1.3.6	ACCUMULATOR Specifications	9-10
9.1.3.7	TOTAL Specifications	9-12
9.1.3.8	EMIT Specifications	9-13
9.1.3.9	LIST Specifications	9-14
9.1.4	Output From COBRG	9-15
9.1.5	COBRG Command String	9-15
9.1.5.1	Default Assumptions	9-15
9.1.6	The Sample Program	9-15
9.1.6.1	Specification Lines	9-16
9.1.6.2	Listing File	9-17
9.1.6.3	Source Program	9-18
9.1.6.4	Input File	9-23
9.1.6.5	Printed Report	9-24
9.1.7	Error Messages	9-25
9.2	REFORMAT	9-28
9.2.1	Introduction	9-28
9.2.2	REFORMAT Command String	9-29
9.2.3	Error Messages	9-30
CHAPTER 10	ERROR MESSAGES	10-1
10.1	COMPILER SYSTEM ERRORS	10-1
10.2	DIAGNOSTIC ERROR MESSAGES	10-1
APPENDIX A	THE COBOL FORMATS	A-1
APPENDIX B	LOGICAL UNIT NUMBER (LUN) ASSIGNMENTS	B-1
APPENDIX C	PDP-11 COBOL Field Release Version 1.00 Restrictions	C-1

FOREWORD

The PDP-11 COBOL User's Guide is intended primarily for reference. It assumes that the reader has a working knowledge of the COBOL language. It should not be considered as a tutorial guide for beginning programmers.

This guide describes the COBOL data files, recording modes, file structures, data Formats, and some of the features of this compiler, as well as a complete list of the error messages generated by the compiler and runtime systems, a description of the I-O devices available with the system, some hints on good programming practices, some modern techniques for debugging source programs, and a description of the PDP-11 COBOL utility programs, COBRG and REFORMAT.

Those wishing to learn the COBOL language are referred to the following excellent tutorial manuals:

Farina, Mario V., COBOL Simplified,
New Jersey, Prentice Hall, Inc., 1968.

McCameron, Fritz A., COBOL Logic and Programming,
Homewood, Illinois, Richard D. Irwin, Inc., 1970

McCracken, Daniel D. and Garbassi, Umberto,
A Guide to COBOL Programming, Second Edition,
New York, John Wiley and Sons, Inc., 1970

Note: these publication dates are the latest available. They will all probably be revised to reflect ANS-74 standards.

ACKNOWLEDGMENT

COBOL is an industry language and is not the property of any company or group of companies, or of any organization or group of organizations.

No warranty, expressed or implied, is made by any contributor or by the CODASYL Programming Language Committee as to the accuracy and functioning of the programming system and language. Moreover, no responsibility is assumed by any contributor, or by the committee, in connection therewith.

The authors and copyright holders of the copyrighted material used herein are: FLOW-MATIC (trademark of Sperry Rand Corporation), Programming for the Univac (R) I and II, Data Automation Systems copyrighted 1958, 1959, by Sperry Rand Corporation; IBM Commercial Translator Form No. F28-8013, copyrighted 1959 by IBM; FACT, DSI 27A5260-2760, copyrighted 1960 by Minneapolis-Honeywell.

They have specifically authorized the use of this material, in whole or in part, in the COBOL specifications. Such authorization extends to the reproduction and use of COBOL specifications in programming manuals or similar publications.

Procedures have been established for the maintenance of COBOL. Inquiries concerning the procedures for proposing changes should be directed to the Chairman of the CODASYL Programming Language Committee, P.O. Box 124, Monroeville, Pa. 15146.

CHAPTER 1

INTRODUCTION

COBOL (COmmon Business Oriented Language), a near-English programming language, is designed primarily for business use. The language is an industry/government standard.

The development of the COBOL programming language is a continuing project of the Programming Language Committee of the Conference on Data Systems Languages (CODASYL). Representatives of computer manufacturers and computer users in both industry and governments comprise the committee.

The following section provides a brief history of the development of the CODASYL COBOL language.

1.1 THE HISTORY OF COBOL

In May, 1959, representatives from industry and the U.S. government met in the Pentagon to consider the desirability and feasibility of establishing a common language for the programming of electronic computers for business-type applications. CODASYL developed out of this meeting.

The original COBOL specification resulted from the work of a CODASYL committee. By September, 1959, the committee submitted to CODASYL a language that they considered to be superior to existing languages. They further modified the orginal specification and by December, 1959, COBOL existed as a language that was not identified with any manufacturer, and therefore presented advantages for both government and private industry users. The Programming Language Committee of CODASYL is now responsible for the publication of the COBOL Journal of Development, which comprises the official CODASYL COBOL language specifications.

COBOL-60 -- The first version of COBOL to be presented by the Committee was called COBOL-60. COBOL-60 proved that the concept of a common business oriented lanugage was indeed practical.

COBOL-61 -- The second official version of COBOL, COBOL-61 contained changes in the organization of the Procedure Division. (These changes made COBOL-61 not completely compatible with COBOL-60.) This version was widely implemented and was the basis for many COBOL compilers.

COBOL-61, Extended -- COBOL-61, Extended was generally compatible with COBOL-61 and included the following major modifications:

- Addition of the Sort feature and the Report Writer Feature.
- Modification of the arithmetic statements to include multiple receiving fields.
- Addition of the CORRESPONDING option to the ADD and SUBTRACT statements.

COBOL, Edition 1965 --This version included COBOL-61, Extended plus the following major modifications:

- Addition of options for reading, writing, and processing of mass storage files.
- Addition of table handling features

NOTE

In 1967 the USA Standards Institute (USASI) defined and published (in COBOL Information Bulletin #9) a proposed Standard COBOL. This Standard COBOL which included selected elements of COBOL, Edition 1965, was accepted as the USA Standard COBOL, 1968. The specification of the USA Standard COBOL, 1968 was published as USA Standards document X3.23-1968.

COBOL, 1968 -- In 1968, CODASYL published, in their Journal of Development, a specification for COBOL, 1968. COBOL, 1968 was compatible with COBOL, Edition 1965 and included the following major modifications:

- Addition of inter-program communication.
- Addition of the SUSPEND statement.

- Deletion of the DEFINE, NOTE, and REMARKS statements.
- Revision of the COPY and EXAMINE statements.

COBOL, 1969 -- This version of COBOL is based on COBOL, 1968 with the following modifications:

- Addition of a communication facility that permits input and output with communications devices.
- Addition of the date and time facility.
- Addition of the STRING and UNSTRING statements.
- Addition of the SIGN clause.
- Deletion of the EXAMINE statement and the addition of the more powerful INSPECT statement.
- Deletion of the CONSTANT SECTION of the Data Division.

COBOL, 1970 -- This version of COBOL is based on COBOL, 1969 with the following modifications:

- Addition of the INITIALIZE statement.
- Deletion of the RANGE clause.

NOTE

In the fall of 1969, USASI became the American National Standards Institute (ANSI). In 1971, ANSI began a revision of USA Standard COBOL 1968.

In 1974, ANSI published American National Standard COBOL, 1974 (X3.23-1974). PDP-11 COBOL is based on this ANSI Standard COBOL. This guide refers to ANSI Standard COBOL, 1974 as ANS-74 COBOL.

COBOL, 1973 -- This version of COBOL is based on COBOL, 1970 with further revisions and enhancements.

The PDP-11 COBOL compiler accepts COBOL language elements that are a true subset of ANS-74 COBOL. The PDP-11 COBOL Reference Manual (DEC-11-LCOBA-B-D) and the PDP-11 COBOL Reference Card (2010 00874 3606/Y 14 05) both indicate extensions to the standard with shading. This guide notates all extensions with explanations of the extension, and no shading. However, Appendix A of this guide (COBOL Formats), does indicate extensions with shading.

1.2 THE PDP-11 COBOL COMPILER

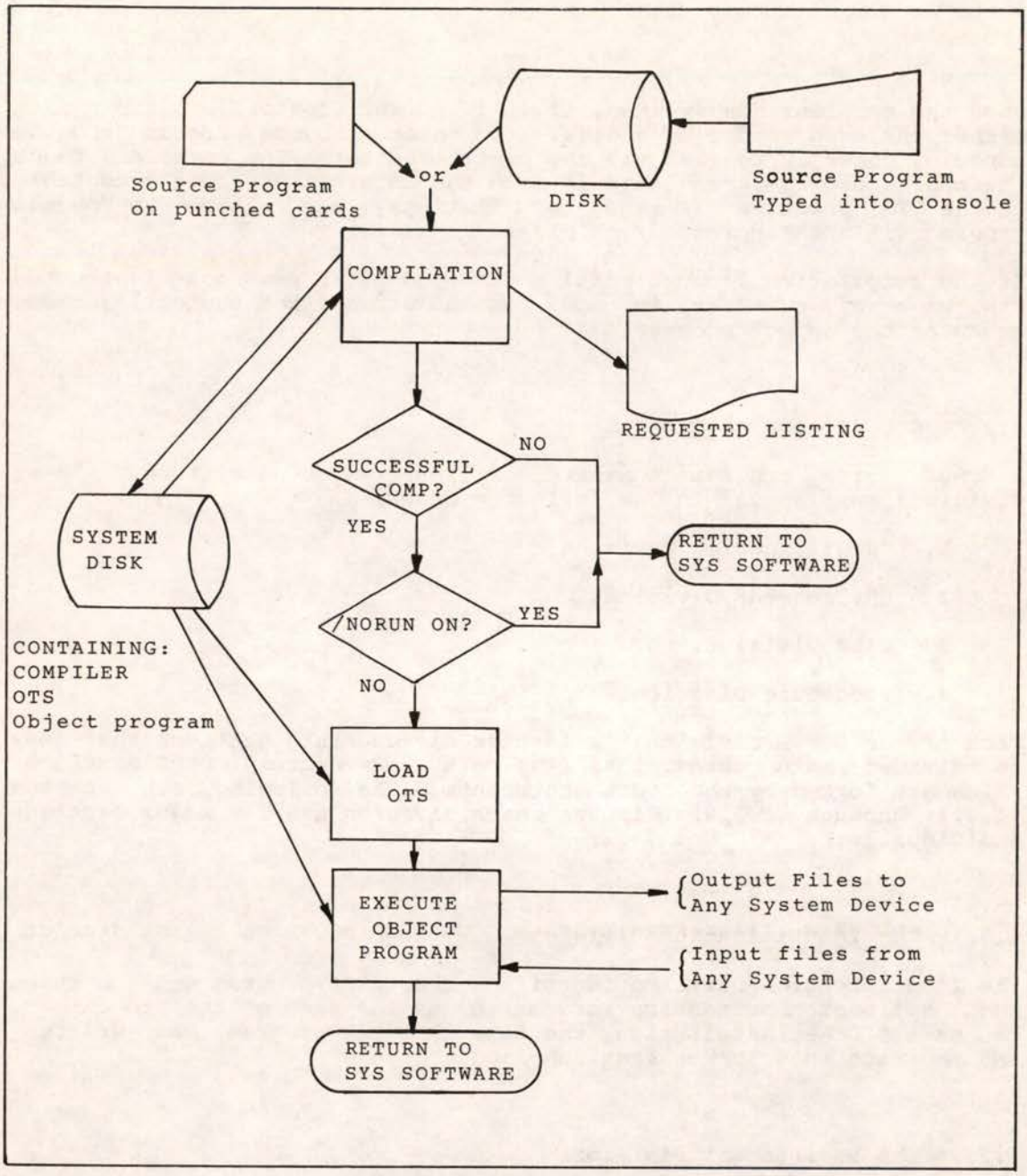
Before a program can be run on a computer, it must be translated from the form that a man can understand (near-English) into a form that the computer can understand (machine language). COBOL compiler systems perform this translation. Since each manufacturer's system has a different machine language, each type of system must have its own COBOL compiler. The COBOL compiler for the PDP-11 system is the PDP-11 COBOL compiler.

The PDP-11 COBOL compiler translates ANS-74 COBOL source programs into an object language that guides the execution of the COBOL object time system. The compiler runs under the supervision of the RSX-11D operating system, and conforms to all of the RSX-11D conventions and restrictions. The compiler itself has the following requirements:

1. A 20K partition.

2. Disk storage space for its work file system and temporary files. (The work file system requires a minimum of 64 blocks and a maximum of 256 blocks. The temporary file and object program file requirements grow in proportion to the size of the source program.)

The PDP-11 COBOL compiler is a compile-and-go system. (However, command string switches can instruct it to compile-only, or to execute a previously compiled COBOL object program.) The following flow chart illustrates the compile-and-go concept. The text following the illustration explains the program flow.



The Compilation/Execution Process

The first step in compiling a COBOL program is to key it into a device for input to the compiler. Start the compiler, which resides on disk, by typing CBL into the console in the MCR mode. Then enter the COBOL command string. (The COBOL command string is discussed in greater detail in Section 2.4 in the next chapter.)

Once the compiler has started, it will accept source programs from either the card reader or a disk. (To enter a source program from the console, enter it to disk via the context editor.) The compiler reads the source program, translates it into the intermediate code acceptable to the OTS, produces all requested listings, and places an object program (if not suppressed) on disk.

If the compilation is successful and the compiler was not instructed otherwise by switches in the command string, the system loads and executes the object program.

1.3 THE COBOL SOURCE PROGRAM

A COBOL source program consists of four major divisions. These divisions must appear in the following order:

1. Identification Division,
2. Environment Division,
3. Data Division,
4. Procedure Division.

Each of the COBOL divisions is further divided into Sections that may be divided into paragraphs. The paragraphs contain COBOL sentences which are formed by the COBOL statements. The following sub-sections (1.3.1 through 1.3.4) discuss each division and its major sections individually.

1.3.1 The Identification Division

The Identification Division identifies the COBOL program and contains such optional documentary information as the name of the programmer, the name of the installation, the date that the program was written, and the date that it was last compiled.

1.3.2 The Environment Division

The Environment Division identifies the hardware configuration of the system that is compiling the program (SOURCE-COMPUTER) and the hardware configuration of the system that is running the program (OBJECT-COMPUTER). The division is divided into the following two major sections:

- Configuration Section -- This required section contains the names of the source and object computers and any mnemonic names that are to be assigned to devices.

- Input-Output Section -- This optional section contains descriptions of all of the external files being manipulated by the program. (The section is required if there are external files.)

1.3.3 The Data Division

The Data Division contains complete descriptions of all of the data to be processed by the program. In this division the programmer must assign a data-name to and describe every data item referred to in the Procedure Division.

The Data Division is divided into the following two optional major sections:

- File Section -- This section contains the descriptions of all of the input and output files and their records.
- Working-Storage Section -- This section contains the descriptions of temporary records and data items.

1.3.4 The Procedure Division

The Procedure Division contains the program's procedural statements - the actual program flow. In this division, the program manipulates the records and data items described in the Data Division.

The Procedure Division may begin with the declaratives, which contain USE procedure declarative sections. (For further information concerning the processing of I/O errors with the USE declaratives see Section 6.10 in this guide.)

The programmer may divide the Procedure Division into paragraphs and sections that each perform a functional procedure.

It is the Procedure Division that makes COBOL's advantages (excellent documentation and programming simplicity) the most apparent. The following sample Procedure Division coding illustrates the documentation capabilities of COBOL programs:

```
PROCEDURE DIVISION.

BEGIN.

    OPEN INPUT TRANSACTION-FILE.

    OPEN OUTPUT MASTER-FILE.
```

```
READ-A-RECORD.
    READ TRANSACTION-FILE NEXT RECORD
         AT END GO TO CLOSE-ROUTINE.
    READ MASTER-FILE NEXT RECORD
         AT END GO TO CLOSE-ROUTINE.
PROCESS-A-RECORD.
    IF TRANS-ACCT-NUMBER IS GREATER THAN MAST-ACCT-NUMBER
         GO TO READ-MAS-RECORD.
    IF TRANS-ACCT-NUMBER IS LESS THAN MAST-ACCT-NUMBER
         GO TO ERROR-ROUTINE.
    .
    .
    .
```

Sample COBOL Procedural Coding

1.4 THE COBOL UTILITY PROGRAMS

PDP-11 COBOL offers three utility programs to aid the user with business-oriented tasks. These programs are:

- COBRG -- Produces report generating COBOL programs.
- REFORMAT -- Converts PDP-11 terminal format COBOL programs into conventional format ANSI COBOL programs.
- SORT -- Sorts data files.

The following sub-sections (1.4.1, 1.4.2, and 1.4.3) describe these utility programs. Chapter 9 in this guide discusses the use of COBRG and REFORMAT in detail. The SORT utility, however, is discussed in a separate manual, The PDP-11 SORT Reference Manual.

1.4.1 COBRG

COBRG generates a report writing source program ready for compilation. The user specifies certain parameters and COBRG builds a COBOL program that produces a tailored report when it is compiled and run.

1.4.2 REFORMAT

PDP-11 COBOL accepts source programs that were coded using either the ANSI 80 column card reference format or the shorter, terminal-oriented PDP-11 reference format. The REFORMAT utility program reads source programs that were coded in the PDP-11 terminal format and converts them to 80 column ANSI compatible source programs. (Section 2.2 in this guide discusses the two reference formats in detail.)

1.4.3 SORT

The SORT utility program sorts data files into ascending or descending order on user-specified keys. Additionally, SORT allows the user to:

1. rearrange, delete, and reformat records in a mass storage file.
2. select the sorting process and input device that best suits the processing environment.
3. create new file indices to access a large data File.
4. batch stream several sorting jobs.
5. define specification files in a fixed format.

CHAPTER 2

TYPICAL USAGE

The other chapters in this guide discuss preparing and coding COBOL source programs so that they may be read into the system for compilation. This chapter discusses how to use those programs; how to get them into the system. It covers the following topics:

- Choice of input medium;
- Choice of reference format;
- Use of the library facility (COPY);
- Use of the compiler and its switches;
- Execution of the object code.

2.1 DEVELOPING A PROGRAM

PDP-11 COBOL provides two media for reading prepared programs into the system where they can be compiled, debugged, and run: punched cards or the computer console.

2.1.1 Punched Cards

Although 80-column punched cards are not as convenient to use as the console, the following techniques can help to simplify their use:

- If the cards are to serve as the program storage medium, use the Conventional reference format (Section 2.2 discusses the reference formats) and utilize the first six columns as line number columns. Increase the numbers by 10 for successive lines and jump by 1000 for successive Divisions or Sections. This not only makes the deck easy to sort if it is dropped, but allows new lines or Sections to be added during debugging or modification.
- Use a control card in the keypunch device to define field positions and to provide skip and duplication.

- Insert comment cards in the deck before Divisions, Sections, subroutines, or paragraphs to help increase readability of the source listing. (A comment card has an asterisk in column 7.)

- If the cards are punched in the Conventional reference format, use columns 73-80 to record program modification information. Information such as your initials and the date of the modification provides an excellent historical record of the modifications to a particular program. (Although this information does not appear on the source listing, it does appear on the cards themselves and on PIP listings.)

If the program is in the Conventional format and the /CVF switch is set ON (compiler switches are discussed later in this chapter), the values in columns 1-6 and 73-80 have no effect on the compiler or the compilation process.

2.1.2 Console

The presence of context editors under RSX and other PDP-11 operating systems makes program preparation, modification, and storage from a console terminal a much simpler task than is generally the case with punched cards.

When creating source programs at the console under RSX, use the context editor to read them into the RSX system. When typing them in under other PDP-11 operating systems, write them on DECtape first and use the Filex (FLX) utility program to read them into the RSX system.

The following suggestion will make editing a source program from a terminal even more convenient:

- Use the terminal reference format (see Section 2.2.2). The terminal format is designed for use at a terminal, and does away with the conventional COBOL fields in columns 1-6 and 73-80. It allows the line to be further shortened by the use of the horizontal tab characters to vertically align similar statements or phrases, and by the use of the carriage return to terminate the line. A typical conventional format line can often be reduced to less than 40 characters in terminal format, with no loss in source text readability.

2.2 REFERENCE FORMATS

PDP-11 COBOL accepts source programs that were coded using either the conventional 80-column card format or the shorter, terminal-oriented PDP-11 terminal format.

The conventional format requires that the source switch be set to /CVF. If the switch is not set, the compiler assumes that the source program is in the terminal format.

2.2.1 The Conventional Format

The conventional format produces source programs that are compatible with the reference format of other COBOL compilers throughout the industry. It is the format specified by the American National Standards Committee for COBOL.

Figure 2-1 illustrates the conventional reference format:

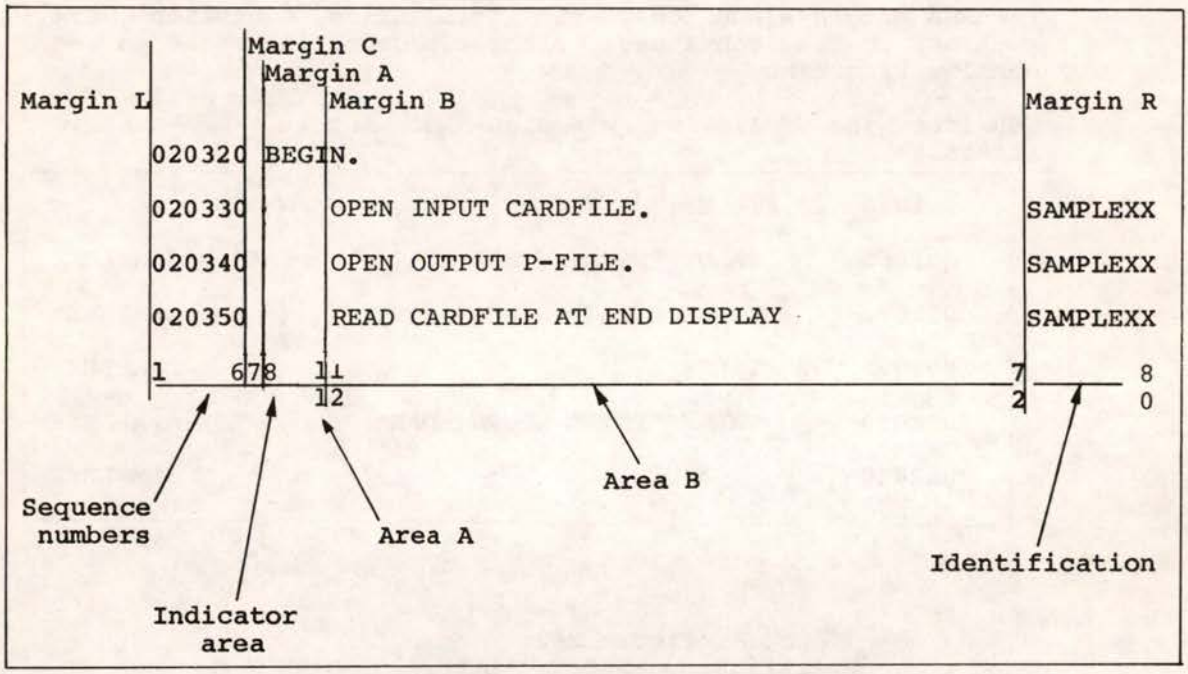


Figure 2-1
Conventional Reference Format Coding

- Character positions 1 through 6 must contain sequence numbers. The PDP-11 COBOL compiler ignores this area; however, to keep the source program compatible with other standard programs, enter sequential numbers in this area.

- Character position 7 is the continuation/comment indicator area. A specific entry (as discussed below) in this position indicates to the compiler that a COBOL word or literal on the preceding line is split and continued on this line, or that this line contains a comment entry.

  To split a word or numeric literal, break the word anywhere on the line, insert a hyphen (-) in this character position on the next line, and resume coding on that line anywhere to the right of character position 11 (Area B). Spaces within

the split (between the last character and margin R of the first line, and between character position 11 and the first character on the continued line) have no effect on the word or numeric literal.

To split non-numeric literals, break the literal at margin R (any spaces at the end of the first line become a part of the literal and will appear when it is printed or typed), insert a hyphen in this character position on the next line, and, beginning with a quotation mark, resume coding anywhere to the right of character position 11 (Area B). If the literal reaches margin R and only the terminating quotation mark remains to be continued, place two quotation marks on the continuation line.

Consider the following examples of split non-numeric literals:

```
001050  02 FILLER
001060       VALUE "NOTE:  THIS ACC      HAS NOSAMPLEXX
001070-      "T BEEN PAID SINCE".            SAMPLEXX
002020  02 FILLER                           SAMPLEXX
002030       VALUE "TOTAL HOURS WORKE      WEEK:SAMPLEXX
002040-        "".                           SAMPLEXX
```

Figure 2-2
Splitting Non-numeric Literals

To break COBOL coding that does not split a word or literal, break the statement between the words and resume coding on the next line. If the break does not split a word or literal, the compiler does not require an indicator.

The line immediately preceding a continuation line must not be a comment line or a blank line.

To create a comment line, place an asterisk or a slash (/) in character position 7 and enter any desired comment to the right of it. An asterisk causes the compiler to ignore the information on the line and print it on the source listing as a comment. A slash causes the compiler to skip to head of form before printing the comment. When a comment requires more than one line, each line must have its own indicator.

```
...
093030      GO TO READ-IN.                              SAMPLEXX
093040* PROGRAM SUBROUTINES ARE NEXT.                   SAMPLEXX
093050* THE FIRST ONE CALCULATES THE NUMBER             SAMPLEXX
093060* OF ITEMS SHOPLIFTED IN ONE WEEK.                SAMPLEXX
093070*                                                 SAMPLEXX
093080 SHOPLIFT.                                        SAMPLEXX
093090          IF SHIFTY-LOOK = BAL-GONE               SAMPLEXX
093100          GO TO COPS-RTN.                         SAMPLEXX
...
```

Figure 2-3
Comment Lines

- Character positions 8 through 11 comprise Area A. All Division names, Section names, and paragraph names must begin in Area A. The Data Division level indicator (FD) and level numbers may begin in either Area A or Area B.

- Character positions 12 through 72 comprise Area B. All of the remaining COBOL entries must begin in Area B.

- Character positions 73 through 80 are reserved for program identification characters. The compiler does not interrogate this field. It may contain any combination of up to eight characters that are convenient for identifying the source program card deck.

Conventional format lines may be shortened by terminating the line with a carriage return character (Octal 015). The compiler treats the carriage return character as a redefinition of margin R.

To eliminate spaces from the lines, use tab characters to position the words on the line. The compiler replaces the tab character (Octal 011) with enough spaces to advance to the next higher tab stop. Tab stops are set at positions 7, 8, 12, 20, 28, 36, 44, 52, 60, 68, and 73.

2.2.2 The Terminal Format

The PDP-11 terminal format produces source programs without the sequence number and identification fields. Since PDP-11 COBOL does not require information in either of these fields, it is more efficient to write programs without using them.

NOTE

Programs written in the PDP-11 terminal format are not compatible with the ANSI 74 reference format. However, the REFORMAT utility program converts terminal format programs to the standard format, quickly and easily. For further information concerning the REFORMAT utility, see Chapter 9, PDP-11 COBOL Utility Programs.

The following sample coding illustrates the terminal reference format:

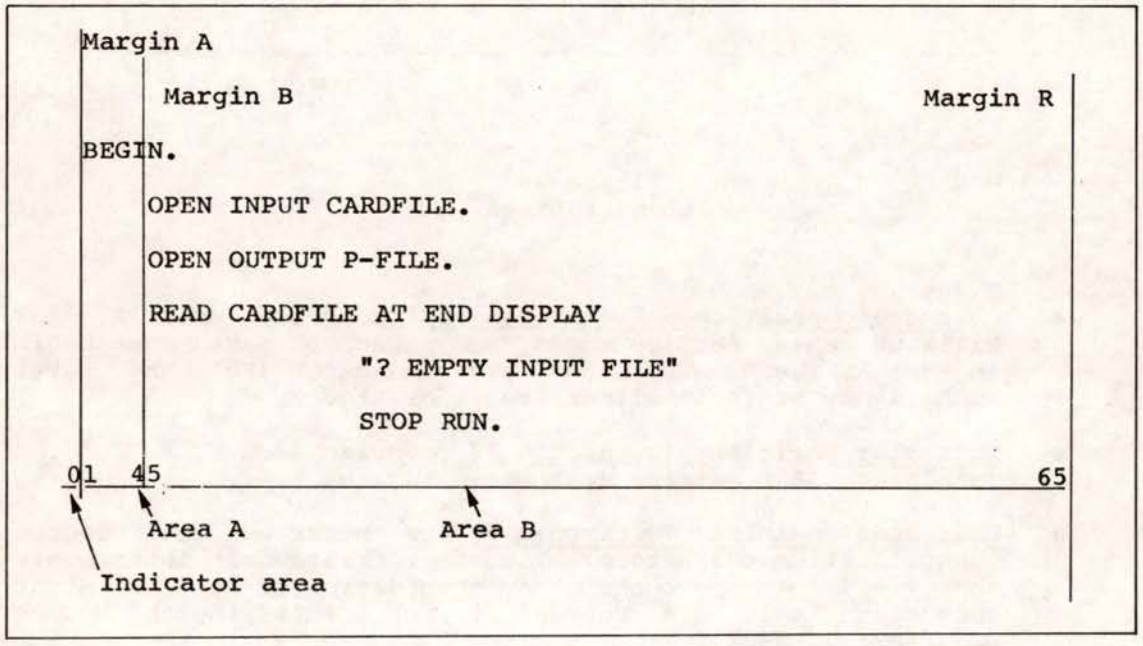


Figure 2-4
Terminal Reference Format Coding

- Character position 0 is the continuation/comment indicator area. A specific entry (as discussed below) in this position indicates to the compiler that a COBOL word or literal on the preceding line is split and continued on this line, or that this line contains a comment entry.

  This position is the first position of a line. When it contains a continuation or comment indicator, it becomes character position zero. The next character position becomes position one and the remaining character positions are sequential from there for a total of 65 character positions (not including position 0).

To split a word or numeric literal, break the word anywhere on the line, enter a hyphen (-) in this position on the next line, and resume coding anywhere to the right of character position four. Spaces within the split (between the last character and position 65 of the first line, and between character position four and the first character on the continued line) have no effect on the word or numeric literal being continued.

To split non-numeric literals, break the literal at character position 65 (any spaces at the end of the first line become a part of the literal and will appear when it is printed or typed), insert a hyphen in this character position on the next line, and, beginning with a quotation mark, resume coding anywhere to the right of character position four. If the literal reaches position 65 and only the terminating quotation mark remains to be continued, place two quotation marks on the continuation line.

Consider the following examples of split non-numeric literals:

```
    02 FILLER
    VALUE "      NAME          ACCOUNT NUMBER              QUAN
-   "TITY ORDERED              UNIT COST TOTAL COST".
    02 FILLER
       VALUE "EMPLOYEE PAY NUMBER       LAST NAME INITIAL
-   "".
```

Figure 2-5
Splitting Non-numeric Literals

To break COBOL coding that does not split a word or literal, simply break off the statement between words and resume coding on the next line. If the break does not split a word or literal, the compiler does not require an indicator.

The line immediately preceding a continuation line must not be a comment line or a blank line.

To create a comment line, place an asterisk or a slash (/) in character position 7 and enter any desired comment to the right of it. An asterisk causes the compiler to ignore the information on the line and print it on the source listing as

a comment. A slash causes the compiler to skip to head of form before printing the comment. When a comment requires more than one line, each line must have its own indicator.

The following sample coding illustrates comment lines in the terminal format.

```
...
       GO TO TOP.
*
* THE TAX CALCULATION SECTION STARTS HERE.
* IT CALCULATES EMPLOYEES TAXES AS FOLLOWS:
* SINGLE 75%
* MARRIED 65%
* OTHERS 59%
*
       IF EMPL-TAX-CODE = SINGLE
           NEXT SENTENCE
...
```

Figure 2-6

Comment Lines

- Character positions 1 through 4 comprise Area A. All Division names, Section names, and paragraph names must begin in Area A. The Data Division level indicator (FD) and level-numbers may begin in either Area A or Area B.

- Character positions 5 through 65 comprise Area B. All of the remaining entries must begin in Area B. A line need not be space filled to character position 65, but may be terminated after the last non-space character by a carriage return.

Tab stops are set at character positions 5, 13, 21, 29, 37, 45, 53, 61, and 66. The compiler replaces the tab character (Octal 011) with enough spaces to advance to the next higher tab stop.

2.3 LIBRARY FACILITY (COPY)

The PDP-11 COBOL library facility provides a means of copying COBOL source language text from a library of source material into a COBOL program during compilation. One COPY statement can place large amounts of library source text into a source program, thus saving repetitious coding. The compiler treats the copied material as though it were part of the program being compiled. However, the material does not physically alter the source program file in any way.

The library facility should not be confused with the COBOL source update facility.

The COBOL library facility has two important benefits for the user:

1. Standardization of File and Coding Conventions

   A typical data file is processed by more than one program, and each processing program must describe the characteristics of that file (file-name, blocking factor, field names, etc.). Often the programs are written by one programmer, then maintained and updated by another programmer, who has to try to understand a program that was written by someone else. Since this situation arises at most sites, a good practice is to design a standardized file description (keeping in mind the programs that process that file) and place it in the library. Then a COPY statement in each processing program can merge it into the program at compile time.

   This technique applies as well to any procedure that is used in many different applications. For example, the library could contain a standardized routine that converts calendar dates to Julian dates to be merged into each source program that uses this function.

2. Time Savings for Initial Coding and Updating

   The COBOL programmer's most time-consuming chore is defining and coding file and record descriptions. If the descriptions existed in the library, a single COPY statement could save the time required for all of those entries.

   Another time-consuming chore arises when a file format must be changed. Typically, when a file format changes, the programmer must change and recompile all of the programs that use that file. If the file description were in a library file, the programmer would have to change only the library file. Of course, the source programs would still have to be recompiled, but would require no individual coding changes.

   Commonly-used Procedure Division subroutines can yield the same time-saving benefits.

2.3.1 Creating a COBOL Library File

Each line of a COBOL library file must be in such a form that, when it merges into the source program, it forms a syntactically correct COBOL clause, phrase, or sentence. It can meet this condition either by being syntactically correct by itself, or by becoming so when it merges with the source program.

The library text must conform to the rules for the COBOL reference format. (See the Library Module in the PDP-11 COBOL Reference Manual.) It may be written in either the conventional (with page-line numbers) or the terminal (without page-line numbers) format. However, the library file and the source programs that it merges into must be in the same format. (A conventional format library file cannot properly merge into a terminal format source program and vice-versa.)

When writing text for library files in the terminal format, place at least four space characters or a tab character before any entry that normally begins in Area B of the COBOL source program. Left justify, without space characters, entries that normally begin in area A or at character position 0.

Since each library file contains all of the source language text to be merged into a source program by one COPY statement, the COPY statement text-name must refer to the library file-name.

To create the library file, either punch it into cards and use PIP to place it on DECtape or disk, or enter it directly on one of these media via the console. RSX-11D provides a text editor (EDIT) for accomplishing this function. There is no method for updating COBOL library files on magtape. The available media for these files are DECtape and disk storage.

2.3.2 The COPY Statement

The COPY statement is a compile-time function that merges a COBOL library file into a COBOL source program.

The statement must begin with the word COPY and end with a period. The compiler logically replaces the COPY statement (including the period) with the library file named by the statement. However, both the COPY statement and the library text material appear in the source listing (see section 2.3.4, The Source Listing). The statement may appear anywhere in a source program that a COBOL word is allowed. The simplest form of the statement follows:

```
COPY text-name.
```

Text-name must specify either a file-name or a non-numeric literal. If a file-name is specified, the compiler assumes standard file specification defaults , and copies the text from the latest version of that file into the source program.

The format for the full File Specification follows:

```
device:[uic]file-name.file-type;version-number
```

This specification defaults as follows:

device:	--	the standard system device or the device specified in the batch JCL;
[user-identification]	--	200,200 unless altered via the RSX SET/UIC command

filename -- (no default)

.file-type -- .LIB;

,version-number -- the latest version.

For example, the following text-name entry copies a library file named JACKS.LIB from the system disk to the source program.

```
COPY JACKS.
```

This text-name entry causes the software to search the system disk for a file named JACKS.LIB. This search takes place in the user's directory only.

If a non-numeric literal is entered, it may specify the full file specification for that file. For example, the following entry also copies the library file JACKS.LIB;3 from DK1 into the source program.

```
COPY "DK1:JACKS.LIB;3".
```

This text-name entry causes the software to search DK1 for version 3 of the file named JACKS.LIB.

Only four situations require the use of the non-numeric literal option to indicate the full file specification for the COPY statement:

1. when the library has a file type other than .LIB;
2. when there is more than one device containing a library file;
3. when the library contains more than one version of the file and you want to copy a version other than the latest; and,
4. when the library file is in another directory.

The following examples use only the file-name option.

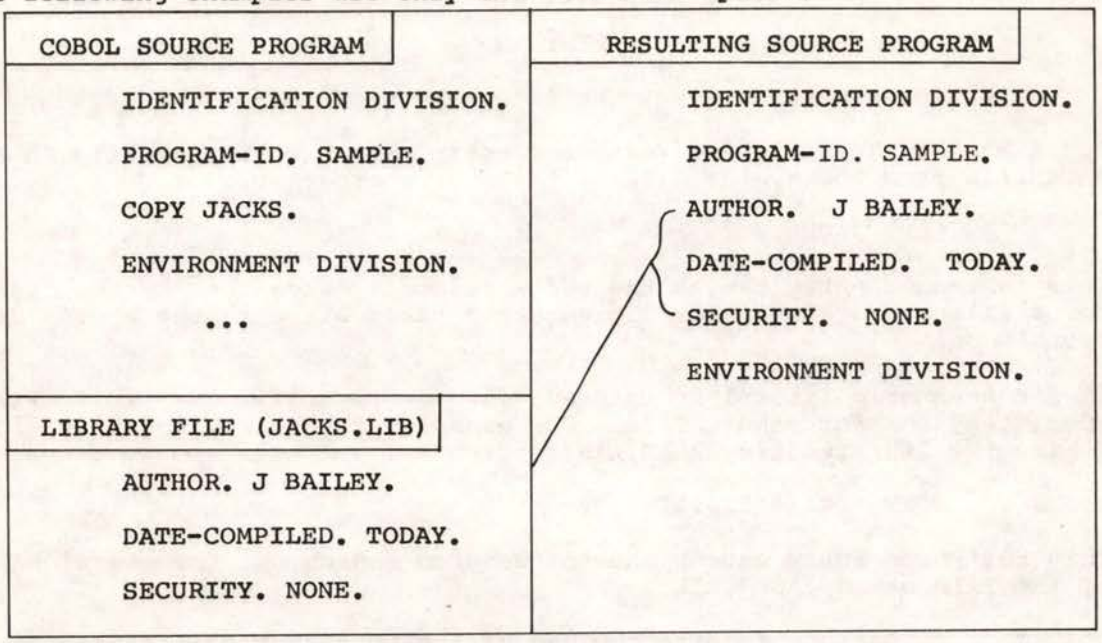
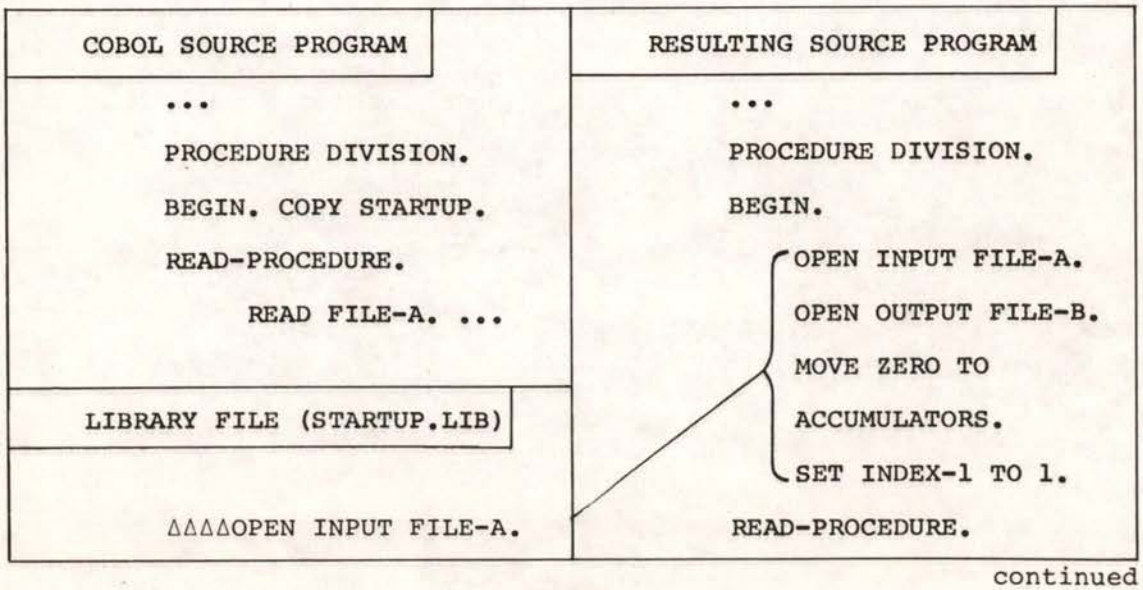


Figure 2-7
Merging a Library File

The library file in Figure 2-8 is formatted (four spaces before each entry) so that when it merges into the source program, each entry begins in Area B. If the four spaces were not there, the text would move into Area A and cause syntax errors (Area A is reserved for division headers, paragraph headers, etc.).



continued

ΔΔΔΔOPEN OUTPUT FILE-B. ΔΔΔΔMOVE ZERO TO ΔΔΔΔACCUMULATORS. ΔΔΔΔSET INDEX-1 TO 1.	READ FILE-A. ...

Figure 2-8
Merging a Library File into Area B

Since the COPY statement may appear anywhere in a source program that a COBOL word is allowed, it can be used in various ways to solve a particular programming problem. For example, if a library file, named DRACULA, contains the single entry "ΔΔΔΔBLOOD-RATE", it could be used in the Data Division as follows:

SOURCE COPY STATEMENT	RESULTING SOURCE STATEMENT
02 COPY DRACULA. PIC 99V99.	02 BLOOD-RATE PIC 99V99.

Figure 2-9
Using the COPY Statement in a Data-description

Or it could be used in the Procedure Division as follows:

SOURCE COPY STATEMENT	RESULTING SOURCE STATEMENT
MULTIPLY 40 BY COPY DRACULA. GIVING PLASMA.	MULTIPLY 40 BY BLOOD-RATE GIVING PLASMA.

Figure 2-10
Using the COPY Statement in a Procedural Statement

The periods terminating the COPY statements in both of these examples are replaced by the test file. No periods appear in the resulting source program unless it is in the text file. (If a source statement needs a period, the text file must have a period at the required point.)

NOTE

The preceding two examples are not recommended uses of the COPY statement. This chapter includes them only to illustrate the mechanics of the PDP-11 COBOL library facility.

2.3.3 The COPY REPLACING Statement

Sometimes it may be necessary to tailor library text material for use in a particular program. (For example, if a data description in the library text has level numbers incremented by 1 -- 01, 02, etc. and you want them incremented by four -- 01, 05, etc.). The COPY statement can replace, during the copying process, all occurrences of a given literal or word with an alternate literal or word. A sample COPY REPLACING statement follows:

```
COPY WALDEN REPLACING 02 BY 05.
```

This sample statement causes the compiler to scan the library file (WALDEN) searching for 02. Wherever it finds a 02, it replaces it with a 05. A match occurs only if the compiler finds a 02 (not just a 0 or just a 2). The REPLACING character string, which may be a literal or a word, must compare equally, character for character, with the entire character string in the library text. The following table shows some successful (YES) and unsuccessful (NO) matches:

Table 2-1
Successful and Unsuccessful Replacing Matches

GIVEN LITERAL OR WORD	LIBRARY TEXT	MATCH?
"ABC"	"ABCD"	NO
HRLY-RATE	HRLY-RATE	YES
1	1	YES
"2"	2	NO
" 15"	"15"	NO
"012"	"12"	NO
012	12	NO
SUBTRACT	SUBTRACT	YES
"012"	"012"	YES

2.3.3.1 Examples, COPY REPLACING

The following examples of the COPY REPLACING statement all refer to this library file:

NEWSBOY (library filename)

```
01 A.
   02 B PIC 99.
   02 C PIC 99 VALUE 2.
   02 D PIC X(5) VALUE "ABCDE".
   02 E PIC 99V99 VALUE 3.75.
   02 F PIC 99 VALUE 02.
```

Example-1

```
COPY NEWSBOY REPLACING B BY X.
```

This statement merges the entire file named NEWSBOY into the source program and changes data-name B to X. It does not change the character "B" in the character string of data-name D because it is part of a non-matching character string. This statement causes the compiler to merge the following text into the source program:

```
01 A.
   02 X PIC 99.
   02 C PIC 99 VALUE 2.
   02 D PIC X(5) VALUE "ABCDE".
   ...
```

Example-2

```
COPY NEWSBOY REPLACING 2 BY 6.
```

This statement merges the entire file named NEWSBOY into the source program and changes the 2 in the entry for data-name C to a 6. It does not change the 02 in the literal entry for data-name F nor any of the 02 level numbers because they contain a non-matching character, 0. This statement causes the compiler to merge the following text into the source program:

```
01 A.
    02 B PIC 99.
    02 C PIC 99 VALUE 6.
    02 D PIC X(5) VALUE "ABCDE".
    02 E PIC 99V99 VALUE 3.75.
    02 F PIC 99 VALUE 02.
```

Example-3

```
COPY NEWSBOY REPLACING 02 BY 63
```

This statement merges the entire file named NEWSBOY into the source program and changes not only the 02 literal entry in data-name E, but also all of the 02 level numbers to 63. Since level number 63 is illegal, this causes the compiler to produce a syntax error. The

replacing process is not sensitive to the syntax of the text. The string of characters in the library may be literals, level-numbers, data-names, etc.; if they match the REPLACING string, character for character, the compiler replaces them. Consider the results of this statement:

```
01 A.

   63 B PIC 99.

   63 C PIC 99 VALUE 2.

   ...

   63 F PIC 99 VALUE 63.
```

Example-4

```
COPY NEWSBOY REPLACING B BY X,
                       "ABCDE" BY "HIJKL",
                       3.75 BY 4.23.
```

This statement illustrates how a single COPY statement can replace more than one literal or word. It causes the compiler to merge all of the following text into the source program:

```
01 A.

   02 X PIC 99.

   02 C PIC 99 VALUE 2.

   02 D PIC X(5) VALUE "HIJKL".

   02 E PIC 99V99 VALUE 4.23.

   02 F PIC 99 VALUE 02.
```

2.3.4 The Source Listing

Depending on how the COPY statement is written, the PDP-11 COBOL compiler lists library text material either before or after the COPY statement.

2.3.4.1 Before the COPY Statement - If other source material (including spaces) follows the COPY statement on the same source line, the compiler lists the library text before the COPY statement. Consider the following illustration:

SOURCE LINE	SOURCE LISTING
...	...
COPY CHANGES. ADD A TO B.	text-line-1
	text-line-2
	text-line-3
	COPY CHANGES. ADD A TO B.

Figure 2-11
Placing the Library Text Before the COPY Statement

The reason for this is that the compiler does not place a source line to an output device until it has scanned the entire line. Therefore, in this case (CHANGES), it takes the following steps, in order:

1. scans the COPY statement;
2. senses more information on the same line;
3. prints the library text;
4. scans the rest of the line; and,
5. prints the entire source line.

This results in a somewhat confusing listing and should be avoided. When the library text follows the COPY statement, it produces a much more readable listing.

2.3.4.2 After the COPY Statement - If the COPY statement terminates its source line, the compiler merges the library text after the COPY statement. Consider the following illustration:

SOURCE LINE	SOURCE LISTING
...	...
COPY CHANGES.	COPY CHANGES.
ADD A TO B.	text-line-1
	text-line-2
	text-line-3
	ADD A TO B.

Figure 2-12
Placing the Library Text After the COPY Statement

In this case, the compiler takes the following steps, in order:

1. it scans the COPY statement;
2. it prints the COPY statement;
3. it prints the library text; and,
4. it prints the next sequential statement.

2.3.5 Common Errors, Library Facility

The most common errors made when using the library facility are:

- Failing to follow the rules for COBOL Reference Format when creating the library file;
- Writing the library file in one format (Conventional or Terminal) and the source program in the other;
- Forgetting to terminate the COPY statement with a period;
- Using data-names in the library file that also appear in the source program, thus causing duplicates;
- Writing the library text so that when it merges into the source program, it becomes syntactically incorrect;
- Merging the wrong library file, either because multiple versions exist or because of misspelling;
- Writing other source material on the same line following the COPY statement, thus causing confusion on the source program listing;
- Forgetting that numeric literals (such as 02, 77, etc.) used in the REPLACING option replace level numbers, picture descriptions, and paragraph or section names, when they find matches.
- Forgetting that a period must appear in the text file if it is to appear in the source program - the period terminating the COPY statement is replaced by the text.

2.4 USING THE COMPILER

This section is intended to help the user execute the COBOL compiler and run-time system.

2.4.1 Calling the COBOL Compiler

The following instructions assume that the user has bootstrapped RSX-11D, and installed and loaded all of the desired device handlers.

If the COBOL compiler was not installed at SYSGEN time, the first step is to install it. To install the compiler, type INS COBOL as shown below (prompting messages, typed by the system, are underlined):

MCR>INS [11,1] COBOL

Then, to start the COBOL compiler, type CBL in MCR mode, as shown below:

MCR>CBL

This starts the compiler, which then types out the following console prompting message, indicating that it is ready to accept a command string:

CBL>

2.4.2 COBOL Command String

The format of the command string follows:

A,B=C/Si ... Sj

A represents the object-program file specification;

B represents the listing-file specification;

C represents the source-program file specification; and,

Si through Sj represent compiler switches.

The subsections below (2.4.2.1 through 2.4.2.4) discuss these specifications and switches in detail, and subsection 2.4.2.5 provides several examples of complete COBOL command strings.

2.4.2.1 Object Program File Specification

The Object Program File specification specifies:

1. the device for the object file
2. the UIC
3. a file-name and file-type
4. the version number of the file

The Object Program File specification is optional. If no object program is required, it may be omitted. If it is omitted, the command string must begin with a comma, and the compiler produces no object program.

When the command string contains an Object Program File Specification, the compiler produces an object program. If the specification does not specify an explicit device, the object program file defaults to the system device. If it does not specify an explicit file-type the compiler assigns a file-type of .OBJ to the file. If it does not specify the version number of the file-type, the system creates a new version; or, if the file does not exist, it creates version 1 of the file.

Consider the following examples of Object Program File Specifications:

COMMAND STRING	RESULT
X,B=C	An object program on the system device called X.OBJ;n (n is the next highest version number).
X;3,B=C	An object program on the system device called X.OBJ;3 (3 is the version number).
DK1:X,B=C	An object program on DK1 called X.OBJ;n (n is the version number).
,B=C	No object program.
X.TST;1,B=C	An object program on the system device called X.TST;1.

Figure 2-13
Sample Object Program File Specifications and Their Results

2.4.2.2 Listing File Specification

The Listing File Specification specifies:

1. the device for a compilation listing of the source file
2. the UIC
3. The file-name and file-type of the listing file
4. the version number of the file

The listing file consists of the user's source program with embedded diagnostics and other information, as requested via compiler switches (see section 2.4.2.4, Compiler Switches). The Listing File Specification can either direct the listing file to the line printer or spool it to another device for subsequent printing. The system assigns a default file-type of .LST and directs the files to the system device, unless otherwise specified. If the specification does not specify the version number of the file, the system creates a new version; or, if the file does not exist, it creates version 1 of the file.

Consider the following example of Listing File Specifications:

COMMAND STRING	RESULT
A,LP:=C	A compilation on the Line Printer.
A,B=C	A compilation listing spooled to the system device and named B.LST.
A,DK1:B=C	A compilation listing spooled to DK1 and named B.LST.
A,B;2=C	A compilation listing spooled to the system device named B.LST, with a version number of 2 (B.LST:2).
A=C	No compilation listing.

Figure 2-14
Sample Listing File Specifications and Their Results

2.4.2.3 Source Program File Specification

The Source Program File Specification specifies:

1. the device containing the source program; and,
2. the UIC
3. the file-name and file-type
4. the version number of the file

This guide recommends the following devices for source programs: disk, DECtape, card reader, magnetic tape. However, source programs can be input from other devices, if the system supports the device. If the specification does not specify an explicit device, the compiler expects the source program to come from the system device. If the

specification does not specify an explicit file-type, the compiler assigns a file-type of .CBL.

Consider the following examples of Source Program File Specifications:

COMMAND STRING	RESULT
A,B=C	The compiler expects a source program file named C.CBL from the system device.
A,B=C.PRG	The compiler expects a source program file named C.PRG from the system device.
A,B=DK1:C	The compiler expects a source program file named C.CBL from DK1.
A,B=CR:	The compiler expects a source program file from the card reader. (A file-name with cards is ignored.)
A,B=MT2:C	The compiler expects a source program file named C.CBL from magnetic tape drive 2.

Figure 2-15
Sample Source Program File Specifications and Their Results

2.4.2.4 Compiler Switches

The PDP-11 compiler provides a series of switches, for use at compile time, that permit the user to tailor the compilation listing to meet particular needs. Other switches are available at execution time that may be interrogated by the object program.

A list of the switches and their meanings follows:

SWITCH	MEANING
/ERR:n	Suppress the printing of diagnostics with a severity number of less than n. The range of n must be $0 \leq n \leq 2$. The switch cannot suppress severity 2 (fatal) diagnostics. (An entry of 2 suppresses the printing of all severity numbers that are less than 2.) The default is /ERR:0.
/ACC:n	Produce an object program only if the source program contains diagnostics with severities equal to or less than n. The range of n must be $0 \leq n \leq 2$. The default is /ACC:1.
/MAP	Produce a Data Division map showing the memory addresses for Data Division entries.
/LOD	Print the starting and ending addresses of the Data and Procedure Division blocks. Indicate the number of PD blocks available and the version

number of the compiler that produced this object program.

/CVF — The source program is in conventional format (i.e., 80-character images with Area A beginning in character position 8).

/NORUN — Compile only. Do not execute. The default is automatic execution after compilation.

/RUN — Execute only. Do not compile. This switch should be used to execute a previously compiled program. (Section 2.5 explains execute-only in greater detail.)

/USW:n...:m — Turn ON switches n through m. The system sets to OFF all of the switches that are not named. The switches n through m must be in the range 1 to 16(10) or 1 to 20(8). Switch numbers that are expressed in decimal must be followed by a decimal point (for example, switch 14 (decimal) should be expressed as /USW:14.). If the decimal point is omitted, the system interprets /USW:14 (decimal) as an octal number and sets switch 12 ON. Each switch in a series must be separated from the other switches by colons. The Special-Names entries in the user's program interrogate switches n through m.

/HELP — Display information on the console concerning how to use the compiler switches.

2.4.2.5 Examples of the COBOL Command String

```
CBL>TESTA,TESTB=TESTC
```

This command string causes the system to compile source program TESTC.CBL, and spool the compiled listing to a file named TESTB.LST. It produces an object program named TESTA.OBJ and, after compilation, loads and executes TESTA.OBJ. The source program (TESTC.CBL) is in terminal format. All files exist or will be created on the system device.

```
CBL>DK1:TESTA.JOB,LP:=DT0:TESTC/CVF/MAP
```

This command string causes the system to compile source program TESTC.CBL. The compiler expects the source program, which is in conventional format, from DECtape 0. It prints the compiled listing and a Data Division map on the line printer. It produces an object program, named TESTA.JOB on DK1, and loads and executes the object program.

```
CBL>TESTA/RUN/LOD
```

This command string causes the system to execute the program TESTA.OBJ from the system device. It produces a Data Division and Procedure Division map.

2.4.2.6 Common Entry Errors, COBOL Command String

The most common errors made when entering the COBOL command string are:

- omitting the /CVF switch for programs that are in the conventional format, thus causing a system error
- leaving the colon off the LP: specification, thus causing the listing to be spooled to a file named LP.LST
- turning ON switches that contradict the File specification parameters. For example, entering the command string A=C/MAP, which does not request a listing file, yet does request a Data Division map
- omitting version numbers from one or more of the File specifications, causing the system to create a new version, or to compile the wrong version.

2.5 EXECUTING THE OBJECT CODE

2.5.1 Compile-and-Go

The PDP-11 COBOL system is a compile-and-go system. It immediately executes any successfully-compiled program, unless specifically suppressed by the command string.

The COBOL object program file specified in the compilation command string contains all of the information necessary for the COBOL run-time system to execute the program. The COBOL system's own loader loads the file and the COBOL run-time system executes it in one sequential operation.

The system provides no method for task building a COBOL program (installing it and running it by name). However, the program can be compiled and stored, then called and run with a separate command string, as discussed in the following section.

2.5.2 Execute Only

A previously compiled PDP-11 COBOL program can be executed at a later time by following this procedure:

1. Compile the program and create an object file, but suppress execution, as shown below:

   CBL>TEST.OBJ=TEST.CBL/NORUN

   This command string causes the COBOL system to place an object program on the system disk named TEST.OBJ and save this object program for later use.

2. Execute the program later by invoking COBOL and requesting a run of the program, as follows:

```
CBL>TEST.OBJ/RUN
```

This command string causes the COBOL system to load and execute the object program, TEST.OBJ.

2.5.3 Common Errors, Object Code Execution

The most common errors encountered while executing COBOL programs are:

- Programs that require too much space to be accommodated by the COBOL system. When it encounters this problem, the system types the following message while in the loading process:

```
CBL-DATA DIVISION TOO LARGE-EXPAND TASK BY nnnnn
```

(The n's specify the additional space required for the larger work area.)

CHAPTER 3

NON-NUMERIC CHARACTER HANDLING

3.1 INTRODUCTION

COBOL programs hold their data in fields whose sizes are described in their source programs. These fields are thus "fixed" during compilation to remain the same size throughout the lifespan of the resulting object program.

The data descriptions of the fields in a COBOL program describe them as belonging to any of three data classes -- alphanumeric, alphabetic, or numeric class. Numeric class data items contain only numeric values, alphabetic class only A-Z and space, but alphanumeric class data items may contain values that are all alphabetic, all numeric, or a mixture of alphabetic bytes, numeric bytes, or, in fact, any character from the ASCII character set.

Further, these 3 classes are subdivided into 5 categories: alphabetic, numeric, numeric edited, alphanumeric edited, and alphanumeric. Every elementary item except for an index data item belongs to one of the classes and further to one of the categories. The class of a group item is treated at object time as alphanumeric regardless of the classes of subordinate elementary items.

For alphabetic and numeric (data items) class and category are synonymous.

An alphabetic field is a field declared to contain only alphabetic (A-Z and space) characters.

An alphanumeric class field that is declared to contain any ASCII character is called an alphanumeric category field.

If the data description of an alphanumeric class field specifies that certain editing operations will be performed on any value that is moved into it, that field is called an alphanumeric, or numeric edited category field.

When reading the following sections of this chapter, this distinction between the class or category of a data item and the actual value that the item contains should always be kept in mind.

Sometimes the text refers to alphabetic, alphanumeric, and alphanumeric edited data items as non-numeric data items. This is to distinguish them from items that are specifically described as numeric items.

Regardless of the class of an item, it is usually possible to store a value in the item, at object time, which is "illegal". Thus, non-numeric ASCII characters can be placed into a field described as numeric class, and an alphabetic class field may be loaded with non-alphabetic characters.

To increase readability, the following sections occasionally omit the word "class" when describing an item; however, the reader should regard the descriptive word, numeric, or alphabetic, or alphanumeric, as referring to the class of an item unless it applies specifically to the value in the item.

This chapter discusses non-numeric class data and the non-arithmetic, non-input-output operations that manipulate this type of data.

3.2 DATA ORGANIZATION

Usually, the data areas in a COBOL program are organized into group items with subordinate elementary items. A group item is a data item that is followed by one or more data items (elementary items) with higher valued level numbers. An elementary item has no higher valued subordinate level number.

All of the data areas used by COBOL programs (except for certain registers and switches) must be described in the Data Division of the source program. The compiler allocates memory space for these fields and fixes them in size at compilation time.

The following sub-sections (3.2.1 and 3.2.2) discuss, on a general level, how the compiler handles group and elementary data items.

3.2.1 Group Items

The size of a group item is the total size of the data area occupied by its subordinate elementary items. The compiler considers group items to be alphanumeric DISPLAY items. Thus, the software manipulates group items as if they had been described as PIC X() items, and ignores the structure of the data contained within them.

3.2.2 Elementary Items

The size of an elementary item is determined by the number of allowable symbols it contains that represent character positions. For example, consider the following fields:

```
01 TRANREC.
   03 FIELD-1 PIC X(7).
   03 FIELD-2 PIC S9(5)V99.
```

Figure 3-1
Field Sizes

Both fields consume seven bytes of memory; however, FIELD-1 contains seven alphanumeric bytes while FIELD-2 contains decimal digits and an operational sign. Although certain verbs handle these two classes of data differently, the data, in either case, occupies seven bytes of PDP-11 memory. COBOL operations on such fields are independent of the mapping of the field into PDP-11 memory words (16-bit words which hold two 8-bit bytes). Thus, a field may begin in the left- or right-hand byte of a word with no effect on the function of any operations that may refer to that field.

In effect, the compiler sees memory as a continuous array of bytes, not words. This becomes particularly important when declaring a table with the OCCURS clause (see Section 5.1, Table Handling).

Records (a 01 level entry and all of its subordinate entries) and data items that have a level number of 77 and all literal values given in the Procedure Division automatically begin on even byte addresses.

I/O verbs require that records and literals be aligned on word boundaries because the PDP-11 COBOL file system reads and writes integral numbers of words.

Non-input-output verbs do not require alignment of the data. However, when two fields are aligned identically, the processing verb can sometimes increase its efficiency by processing them a word at a time rather that a byte at a time.

In all cases, automatic word alignment of literals, records, and/or 77 items increases the opportunity for more efficient processing.

3.3 SPECIAL CHARACTERS

COBOL allows the user to manipulate any of the 128 characters of the ASCII character set as alphanumeric data even though many of the characters are control characters, which usually control input/output devices. Generally, alphanumeric data manipulations are performed in a manner that attaches no "meaning" to an 8-bit byte. Thus, the user can move and compare these control characters in the same manner as alphabetic and numeric characters.

Although the object program can manipulate all ASCII characters, certain control characters cannot appear in non-numeric literals since the compiler uses them to delimit the source text. Further, the keyboards of the console and keypunch devices have no convenient input key for many of the special characters, thus making it difficult to place them into non-numeric literals.

Special characters may be placed into data fields of the object program by placing the binary value of the special character into a numeric COMP field and redefining that field as alphanumeric DISPLAY. Consider the following example of redefinition. (Keep in mind that the even byte of a word corresponds to the low-order bits of a binary word.)

```
01 LF-COMP PIC 999 COMP VALUE 10.
01 LF REDEFINES LF-COMP PIC X.
01 HT-COMP PIC 999 COMP VALUE 9.
01 TAB REDEFINES HT-COMP PIC X.
01 CR-COMP PIC 999 COMP VALUE 13.
01 CR REDEFINES CR-COMP PIC X.
```

Figure 3-2
Redefining Special Characters

The sample coding in Figure 3-2 introduces each character as a 1-word COMP item with a decimal value, then redefines it as a single byte. (The second byte of the redefinition need not be described at the 01 level, since redefinition at this level does not require identically sized fields.)

The following ASCII code chart may be used to determine the decimal value for any ASCII character. To use the chart, find the desired character; then add its row and column values together to determine the decimal integer to be supplied as a VALUE for the computational item.

Column Value / Row Value	000	008	016	024	032	040	048	056	064	072	080	088	096	104	112	120
0	NUL	BS	DLE	CAN	space	(	0	8	@	H	P	X	grave	h	p	x
1	SOH	HT	DC1	EM	!	)	1	9	A	I	Q	Y	a	i	q	y
2	STX	LF	DC2	SUB	"	*	2	:	B	J	R	Z	b	j	r	z
3	ETX	VT	DC3	ESC	#	+	3	;	C	K	S	[	c	k	s	{
4	EOT	FF	DC4	FS	$	,	4	<	D	L	T	\	d	l	t	|
5	ENQ	CR	NAK	GS	%	-	5	=	E	M	U	]	e	m	u	}
6	ACK	SO	SYN	RS	&	.	6	>	F	N	V	(↑)	f	n	v	(ESC)
7	BEL	SI	ETB	US	apos	/	7	?	G	O	W	(←)	g	o	w	DEL

Figure 3-3
ASCII Code Chart

3.4 TESTING NON-NUMERIC FIELDS

3.4.1 Relation Tests

An IF statement that contains a relation condition (greater-than, less-than, equal-to, etc.) can compare the value in a non-numeric data item with another value and use the result to alter the flow of control in the program.

An IF statement with a relation condition compares two operands, either of which may be an identifier or a literal, except that both cannot be literals. If the relation exists between the two operands, the relation condition has a truth value of True.

Figure 3-4 illustrates the general format of a relation condition. (The relational characters ">", "<", and "=", although required, are not underlined to avoid confusion with other symbols such as greater-than-or-equal-to.)

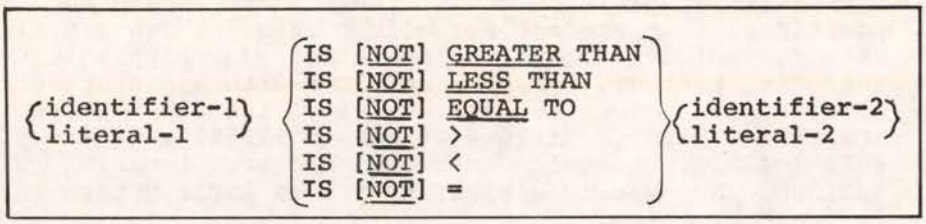


Figure 3-4
Relation Condition

When coding a relational operator, leave a space before and after each reserved word. When the reserved word, NOT, is present, the software considers it and the next key word or relational character to be one relational operator that defines the comparison. Figure 3-5 shows the meanings of the relational operators.

OPERATOR	MEANING
IS [NOT] GREATER THAN IS [NOT] >	The first operand is greater than (or not greater than) the second operand.
IS [NOT] LESS THAN IS [NOT] <	The first operand is less than (or not less than) the second operand.
IS [NOT] EQUAL TO IS [NOT] =	The first operand is equal to (or not equal to) the second operand.

Figure 3-5
The Meanings of the Relational Operators

3.4.1.1 Classes of Data - COBOL allows comparison of both numeric class operands and non-numeric class operands; however, it handles each class of data slightly differently. For example, it allows a comparison of two numeric operands regardless of the formats specified in their respective USAGE clauses, but requires that all other comparisons (including comparisons of any group items) be between operands with the same usage. It compares numeric class operands with respect to their algebraic values and non-numeric (or a numeric and a non-numeric) class operands with respect to a specified collating sequence.

If only one of the operands is numeric, it must be an integer data item or an integer literal and it must be DISPLAY usage; further, the manner in which the software handles numeric operands depends on the non-numeric operand. Consider the following two types of non-numeric operands:

1. If the non-numeric operand is an elementary item or a literal, the software treats the numeric operand as if it had been moved into an alphanumeric data item (which is the same size as the numeric operand) and then compared. This causes any operational sign, whether carried as a separate character

or as an overpunch, to be stripped from the numeric item; thus, it appears to be an unsigned quantity. In addition, if the picture-string of the numeric item contains trailing P characters indicating that there are assumed integer positions that are not actually present, these are filled in with zero digits during the operation of stripping any sign that is present. Thus, an item with a picture-string of S9999PPP is moved to a temporary location where it is described with a picture-string of 9999999. If its value is 432J (-4321), the value in the temporary location will be 4321000. The numeric digits, stored as ASCII bytes, take part in the comparison.

2. If the non-numeric operand is a group item, the software treats the numeric operand as if it had been moved into a group item (which is the same size as the numeric operand) and then compared. This is equivalent to a "group move". The software ignores the description of the numeric field (except for length) and, therefore, includes any operational sign, whether carried as a separate character or as an overpunch, in its length. (Overpunched characters are never ASCII numeric digits, but characters in the range of from A through R, { , or } .) Thus, the sign and the digits, stored as ASCII bytes, take part in the comparison, and zeros are not supplied for P characters in the picture-string.

The compiler will not accept a comparison between a non-integer numeric operand and a non-numeric operand, and any attempt to compare these two items will cause a diagnostic message at compile time.

3.4.1.2 The Comparison Operation - If the two operands are acceptable, the software compares them byte for byte starting at their left-hand end. It proceeds from left to right, comparing the characters in corresponding character positions until it either encounters a pair of unequal characters or reaches the right-hand end of the longer operand.

If the software encounters a pair of unequal characters, it considers their relative position in the collating sequence. The operand with the character that is positioned higher in the collating sequence is the greater operand.

If the operands have different lengths, the comparison proceeds as though the shorter operand were extended on the right by sufficient ASCII spaces (040) to make them both the same length.

If all of the pairs of characters compare equally, the operands are equal.

3.4.2 Class Tests

An IF statement that contains a class condition (NUMERIC or ALPHABETIC) can test the value in a non-numeric data item (USAGE DISPLAY only) to determine if it contains numeric or alphabetic data and use the result to alter the flow of control in the program.

Figure 3-6 illustrates the general format of a class condition. (If the data item consists entirely of the ASCII characters 0123456789 with or without the operational sign, the class condition would determine that it is NUMERIC. If the item consists entirely of the ASCII characters A through Z and space, the class condition would determine that it is ALPHABETIC.

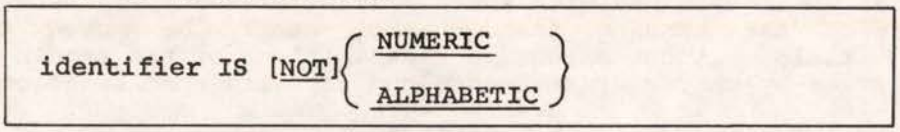


Figure 3-6
Class Condition, General Format

When the reserved word, NOT, is present, the software considers it and the next key word as one class condition that defines the class test to be executed; for example, NOT NUMERIC is a truth test for determining if an operand contains at least one non-numeric byte.

If the item being tested was described as a numeric data item, it may only be tested as NUMERIC or NOT NUMERIC. (For further information on using class conditions with numeric items, see Chapter 4.) The NUMERIC test cannot examine an item that was described either as alphabetic or as a group item containing elementary items whose data descriptions indicate the presence of operational signs.

3.5 DATA MOVEMENT

COBOL provides three statements (MOVE, STRING, and UNSTRING) that perform most of the data movement operations required by business oriented programs. The MOVE statement simply moves data from one field to another. The STRING statement concatenates a series of sending fields into a single receiving field. The UNSTRING statement disperses a single sending field into multiple receiving fields. Each has its uses and its limitations. This section discusses data movement situations which take advantage of the versatility of these statements.

The MOVE statement handles the majority of data movement operations on character strings. However, the MOVE statement has limitations in its ability to handle multiple fields; for example, it cannot, by itself, concatenate a series of sending fields into a single receiving field or disperse a single sending field into several receiving fields.

Two MOVE statements will, however, bring the contents of two fields together into a third (receiving) field if the receiving field has been "subdivided" with subordinate elementary items that match the two sending fields in size. If other fields are to be concatenated into the third field and they differ in size from the first two fields, then the receiving field will require additional subdivisions (through redefinition).

Another method of concatenation with the MOVE statement is to subdivide the receiving field into single character fields, creating a "table" of a single character field that occurs as many times as there are characters in the receiving field, and execute a data movement loop which moves each sending field, a character at a time, using a subscript that moves continuously across the receiving field.

Two MOVE statements can also be used to disperse the contents of one sending field to several receiving fields. The first MOVE statement can move the left-most end of the sending field to a receiving field; then the second MOVE statement can move the right-most end of the sending field to another receiving field. (The second receiving field must first be described with the JUSTIFIED clause.) Characters from the middle of the sending field cannot easily be moved to any receiving field without extensive redefinitions of the sending field or a character-by-character movement loop (as with concatenation).

The concatenation and dispersion limitations of the MOVE statement are handled quite easily by the STRING and UNSTRING statements. The following sections (3.6, 3.7, and 3.8) discuss these three statements in detail.

3.6 THE MOVE STATEMENT

The MOVE statement moves the contents of one field into another. The following illustration shows the format of the MOVE statement.

```
MOVE FIELD1 TO FIELD2.
```

FIELD1 is the name of the sending field and FIELD2 is the name of the receiving field. The statement causes the software to move the contents of FIELD1 into FIELD2. The two fields need not be the same size, class, or usage; and they may be either group or elementary items. If the two fields are not the same length, the software will align them on one end or the other -- and will truncate or pad (with spaces) the other end. The movement of group items and non-numeric elementary items is discussed below.

A point to remember when using the MOVE statement is that it will alter the contents of every character position in the receiving field.

3.6.1 Group Moves

If either the sending or receiving field is a group item, the software considers the move to be a group move. It treats both the sending and receiving fields in a group move as if they were alphanumeric class fields. If the sending field is a group item and the receiving field is an elementary item, the software ignores the receiving field description (except for the size description, in bytes, and any JUSTIFIED clause); therefore, the software conducts no conversion or editing on the receiving field.

3.6.2 Elementary Moves

If both fields of a MOVE statement are elementary items, their data description clauses control their data movement. (If the receiving field was described as numeric or numeric edited, the rules for numeric moves -- see Chapter 4, Numeric Character Handling -- control the data movement.)

The following table shows the legal (and illegal) non-numeric elementary moves.

Table 3-1
Legal Non-Numeric Elementary Moves

SENDING FIELD CATEGORY	RECEIVING FIELD CATEGORY	
	ALPHABETIC	ALPHANUMERIC ALPHANUMERIC EDITED
ALPHABETIC	Legal	Legal
ALPHANUMERIC	Legal	Legal
ALPHANUMERIC EDITED	Legal	Legal
NUMERIC INTEGER (DISPLAY ONLY)	Illegal	Legal
NUMERIC EDITED	Illegal	Legal

In all of the legal moves shown above, the software treats the sending field as though it had been described as PIC X(). If the sending field description contains a JUSTIFIED clause, the clause will have no effect on the move. If the sending field picture-string contains editing characters, the software uses them only to determine the field's size.

Numeric class data must be in DISPLAY (byte) format and it must be an integer.

If the description of the numeric data item indicates the presence of an operational sign (either as a character or an overpunch) or if there are P characters in the picture-string of the numeric data item, the software first moves the item to a temporary location. During this move, it removes the sign and fills out any P character positions with zero digits. It then uses the temporary value (which may be shorter than the original if a separate sign was removed, or longer if P character positions were filled in with zeros) as the sending field as if it had been described as PIC X(), that is, as if its category were alphanumeric.

If the sending item is an unsigned numeric class field with no P characters in its picture-string, the software does not move the item to a temporary location.

A numeric integer data item sending field has no effect on the justification of the receiving field. If the numeric sending field is shorter than the receiving field, the software space fills the receiving field.

In legal, non-numeric elementary moves, the receiving field actually controls the movement of data. All of the following items, in the receiving field, affect the move: (1) the size, (2) the presence of editing characters in its description, and (3) the presence of the JUSTIFIED RIGHT clause in its description. The JUSTIFIED clause and editing characters are mutually exclusive; therefore, the two classes are discussed separately below.

When a field that contains no editing characters or JUSTIFIED clause in its description is used as the receiving field of a non-numeric

elementary MOVE statement, the statement moves the characters by starting at the left-hand end of the fields and scanning across, character-by-character to the right. If the sending item is shorter than the receiving item, the software fills the remaining character positions with spaces.

3.6.2.1 Edited Moves - Alphabetic or alphanumeric fields may contain editing characters. Consider the following insertion editing characters. Alphabetic fields will accept only the B character; however, alphanumeric fields will accept all three characters.

B -- blank insertion position

0 -- zero insertion position

/ -- slash insertion position.

When a field that contains an insertion editing character in its picture-string is used as the receiving field of a non-numeric elementary MOVE statement, each receiving character position that corresponds to an editing character receives the insertion byte value. Figure 3-7 illustrates the use of such symbols with the statement, MOVE FIELD1 TO FIELD2. (Assume that FIELD1 was described as PIC X(7).)

	FIELD2	
FIELD1	PICTURE-STRING	CONTENTS AFTER MOVE
070476	XX/99/XX	07/04/76
04JUL76	99BAAAB99	04 JUL 76
2351212	XXXBXXXX/XX/	235 1212/ /
123456	0XB0XB0XB0X	01 02 03 04

Figure 3-7
Data Movement with Editing Symbols

Data movement always begins at the left end of the sending field, and moves only to the byte positions described as A, 9, or X in the receiving field picture-string. When the sending field is exhausted, the software supplies space characters to fill any remaining character positions (not insertion positions) in the receiving field. If the receiving field becomes exhausted before the last character is moved from the sending field, the software ignores the remaining sending field characters.

3.6.2.2 JUSTIFIED Moves - A JUSTIFIED RIGHT clause in the data description of the receiving field causes the software to reverse its usual data movement conventions. (It starts with the right-hand characters of both fields and proceeds from right to left.) If the sending field is shorter than the receiving field, the software fills the remaining left-hand character positions with spaces. Figure 3-8 illustrates various data description situations for the statement, MOVE FIELD1 TO FIELD2, with no editing.

FIELD1		FIELD2	
PICTURE-STRING	CONTENTS	PICTURE-STRING (AND JUST CLAUSE)	CONTENTS AFTER MOVE
XXX	ABC	XX	AB
		XXXXX	ABC Δ Δ
		XX JUST	BC
		XXXXX JUST	ΔΔABC

Figure 3-8
Data Movement with No Editing

3.6.3 Multiple Receiving Fields

If a MOVE statement is written with more than one receiving field, it moves the same sending field value to each of the receiving fields. It has essentially the same effect as a series of separate MOVE statements that all have the same sending field. (For information on subscripted fields, see Section 3.4).

The receiving fields need have no relationship to each other. The software checks the legality of each one independently, and performs an independent move operation on each one.

Multiple receiving fields on MOVE statements provide a convenient way to set many fields equal to the same value, such as during initialization code at the beginning of a section of processing. For example:

```
MOVE SPACES TO LIST-LINE, EXCEPTION-LINE, NAME-FLD.

MOVE ZEROS TO EOL-FLAG, EXCEPT-FLAG, NAME-FLAG.

MOVE 1 TO COUNT-1, CHAR-PTR, CURSOR.
```

3.6.4 Subscripted Moves

Any field of a MOVE statement may be subscripted and the referenced field may also be used to subscript another name in the same statement.

When more than one receiving field is named in the same MOVE statement, the order in which the software evaluates the subscripts affects the results of the move. Consider the following two situations:

```
Situation 1  MOVE FIELD1(FIELD2) TO FIELD2 FIELD3.

Situation 2  MOVE FIELD1 TO FIELD2, FIELD3(FIELD2).
```

Figure 3-9
Subscripted MOVE Statements

In situation 1, the software evaluates FIELD1(FIELD2) only once, before it moves any data to the receiving fields. In effect it is as if the statement were replaced with the following statements:

```
MOVE FIELD1(FIELD2) TO TEMP.

MOVE TEMP TO FIELD2.

MOVE TEMP TO FIELD3.
```

In situation 2, the software evaluates FIELD3(FIELD2) immediately before moving the data into it (but after moving the data from FIELD1 to FIELD2). Thus, it uses the newly stored value of FIELD2 as the subscript value. In effect, it is as if the statement were replaced with the following statements:

```
MOVE FIELD1 TO FIELD2.

MOVE FIELD1 TO FIELD3(FIELD2).
```

3.6.5 Common Errors, MOVE Statement

A most important thing to remember when writing MOVE statements is that the compiler considers any MOVE statement that contains a group item to be a group move. It is easy to forget this fact when moving a group item to an elementary item, and the elementary item contains editing characters, or a numeric integer. These attributes of the receiving field (which would determine the action of an elementary move) have no effect on the action of a group move.

3.7 THE STRING STATEMENT

The STRING statement concatenates the contents of two or more sending fields into a single field.

The statement has many forms; the simplest is equivalent, in function, to a non-numeric MOVE statement. Consider the following illustration; if the two fields are the same size, or if the sending field (FIELD1) is larger, the statement is equivalent to the statement, MOVE FIELD1 TO FIELD2.

```
STRING  FIELD1 DELIMITED BY SIZE INTO FIELD2.
```

Figure 3-10
Sample STRING Statement

If the sending field is shorter than the receiving field, an important difference between the STRING and MOVE statements emerges: the

software does not space fill the receiving field. Thus, the STRING statement may leave some portion of the receiving field unchanged.

Additionally, the receiving field must be an elementary alphanumeric field with no JUSTIFIED clause or editing characters in its description. Thus, the data movement of the STRING statement always fills the receiving field from left-to-right with no editing insertions.

3.7.1 Multiple Sending Fields

An important characteristic of the STRING statement is its ability to concatenate a series of sending fields into one receiving field. Consider the following example of the STRING statement:

```
STRING FIELD1A FIELD1B FIELD1C DELIMITED BY SIZE
                    INTO FIELD2.
```

Figure 3-11
Concatenation with the STRING Statement

In this sample STRING statement, FIELD1A, FIELD1B, and FIELD1C are all sending fields. The software moves them to the receiving field (FIELD2) in the order in which they appear in the statement, from left-to-right, resulting in the concatenation of their values.

If FIELD2 is not large enough to hold all three items, the operation stops when it is full. If this occurs while moving one of the sending fields, the software ignores the remaining characters of that field and any other sending fields not yet processed. For example, if FIELD2 became full while receiving FIELD1B, the software would ignore the rest of FIELD1B and all of FIELD1C.

If the sending fields do not fill the receiving field, the operation stops with the movement of the last character of the last sending item (FIELD1C in Figure 3-11). The software does not alter the contents or space-fill the remaining character positions of the receiving field.

The sending fields may be non-numeric literals and figurative constants (except for ALL literal). For example, the following statement sets up an address label with the literal period and space between the STATE and ZIP fields:

```
STRING CITY SPACE STATE ".△" ZIP

        DELIMITED BY SIZE INTO ADDRESS-LINE.
```

Figure 3-12
Literals as Sending Fields

Sending fields may also be subscripted. For example, the following statement uses indexes to concatenate the elements of a table (A-TABLE) into a single field (A-FOUR). (I, of course, must be a subscript or an index-name.)

```
STRING A-TABLE(I) A-TABLE(I+1) A-TABLE(I+2) A-TABLE(I+3)
        DELIMITED BY SIZE INTO A-FOUR.
```

Figure 3-13
Indexed Sending Fields

3.7.2 The POINTER Phrase

Although the STRING statement normally starts at the left-hand end of the receiving field, with the POINTER phrase it is possible to start it scanning at another point within the field. (The scanning, however, remains left-to-right.)

```
MOVE 5 TO P.
STRING FIELD1A FIELD1B DELIMITED BY SIZE
        INTO FIELD2 WITH POINTER P.
```

Figure 3-14
Sample POINTER Phrase

When the POINTER phrase is used, the value of P determines the starting character position in the receiving field. In Figure 3-14, the 5 in P causes the software to move the first character of FIELD1A into character position 5 of FIELD2 (the left-most character position of the receiving field is character position 1) and leave positions 1 through 4 unchanged.

When the STRING operation is complete, the software leaves P pointing to one character position beyond the last character replaced in the receiving field. If FIELD1A and FIELD1B in Figure 3-14 are both four characters long, P will contain a value of 13 (5+4+4) when the operation is complete (assuming that FIELD2 is at least 12 characters long).

3.7.3 The DELIMITED BY Phrase

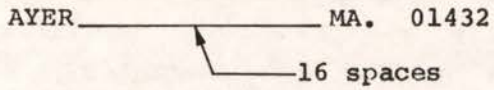
Although the sending fields of the STRING statement are fixed in size at compile time, they frequently contain variable-length items that are padded with spaces. For example, a 20-character city field may contain only the word MAYNARD and 13 spaces. A valuable feature of the STRING statement is that it may be used to move only the useful data from the left-hand end of the sending field. The DELIMITED BY phrase, written with a data-name or literal, instead of the word SIZE, performs this operation. (The delimiter may be a literal, a data item, a figurative constant, or the word SIZE. It may not be ALL literal since ALL literal has an indefinite length. When the phrase contains the word SIZE, the software moves each sending field, in total, until it either exhausts the sending field, or fills the receiving field.)

Consider the following example:

```
STRING CITY SPACE STATE ".Δ" ZIP
        DELIMITED BY SIZE INTO ADDRESS-LINE.
```

Figure 3-15
Delimiting with the Word SIZE

If CITY is a 20-character field, the result of the STRING operation shown in Figure 3-15 might look like the following:



A far more attractive printout can be produced by having the STRING operation produce the following:

AYER, MA. 01432

To accomplish this, use the figurative constant SPACE as a delimiter on the sending field; thus,

```
MOVE 1 TO P.
STRING CITY DELIMITED BY SPACE
         INTO ADDRESS-LINE WITH POINTER P.
STRING ",Δ" STATE ".Δ" ZIP
         DELIMITED BY SIZE
         INTO ADDRESS-LINE WITH POINTER P.
```

Figure 3-16
SPACE as a Delimiter

This sample coding uses the pointer's characteristic of pointing to one character position beyond the last character replaced in the receiving field to enable the second STRING statement to begin at a position one character past where the first STRING statement stopped. (The first STRING statement moves data characters until it encounters a space character -- a match of the delimiter SPACE. The second STRING statement adds the literal, the 2-character STATE field, another literal, and the 5-character ZIP field.)

The delimiter can be varied for each field within a single STRING statement by repeating the DELIMITED BY phrase after the sending field names to which it applies. Thus, the following shorter statement has the same effect as the preceding example. (Placing the operands on separate source lines, as shown in this example, has no effect on the operation of the statement, but improves program readability and simplifies debugging.)

```
STRING CITY DELIMITED BY SPACE
           ",Δ" STATE ".Δ"
           ZIP DELIMITED BY SIZE
               INTO ADDRESS-LINE.
```

Figure 3-17
Repeating the DELIMITED BY Phrase

The sample STRING statement in Figure 3-17 cannot handle 2-word city names, such as New York, since the software would consider the space between the two words as a match for the delimiter SPACE. A longer delimiter, such as two or three spaces (non-numeric literal), can solve this problem. Only when a sequence of characters matches the delimiter will the movement stop for that data item. With a 2-byte delimiter, the same statement can be rewritten in a simpler form:

```
STRING CITY ",Δ" STATE ".Δ" ZIP
            DELIMITED BY "ΔΔ" INTO ADDRESS-LINE.
```

Figure 3-18
Delimiting with More Than One Space Character

Since only the CITY field may contain two consecutive spaces (the entire STATE field is only two bytes long), the delimiter's search of the other fields will always be unsuccessful and the effect is the same as moving the full field (delimiting by SIZE).

Data movement under control of a data-name or literal is generally slower in execution speed than movement delimited by SIZE.

The example in Figure 3-18 illustrates a frequent source of error in the use of STRING statements to concatenate fields. The remainder of the receiving field is not space filled as with a MOVE statement. If ADDRESS-LINE is to be printed on a mailing label, for example, the STRING statement should be preceded by the statement, MOVE SPACES TO ADDRESS-LINE. This guarantees a space fill to the right of the concatenated result. Alternatively, the last field concatenated by the STRING statement can be a field previously set to SPACES. (This sending field must be moved under the control of a delimiter other than SPACE, of course.)

3.7.4 The OVERFLOW Phrase

When the SIZE option of the DELIMITED BY phrase controls the STRING operation and either the pointer value is known or the POINTER phrase is not used, the programmer can tell, by simple addition, if the receiving field is large enough to hold the sending fields. However, if the DELIMITED BY phrase contains a literal or an identifier, or if the pointer value is not predictable, it may be difficult to tell whether the size of the receiving field is adequate, and an overflow may occur.

Overflow occurs when the receiving field is full and the software is either about to move a character from a sending field or considering a new sending field. Overflow may also occur if, during the initialization of the statement, the pointer contains a value that is either less than 1 or greater than the length of the receiving field. In this case, the software moves no data to the receiving field and terminates the operation immediately.

The ON OVERFLOW phrase at the end of the STRING statement tests for an overflow condition:

```
STRING FIELD1A FIELD1B DELIMITED BY "C"
               INTO FIELD2 WITH POINTER PNTR
               ON OVERFLOW GO TO PN57.
```

Figure 3-19
The ON OVERFLOW Phrase

The ON OVERFLOW phrase cannot distinguish the overflow caused by a bad initial value in pointer PNTR from the overflow caused by a receiving field that is too short. Only a separate test, preceding the STRING statement, can distinguish between the two.

The following examples illustrate the overflow condition:

```
      DATA DIVISION.

           ...
           01 FIELD1A PIC XXX VALUE "ABC".
           01 FIELD2 PIC XXXX.

      PROCEDURE DIVISION.

           ...
   1.      STRING FIELD1A QUOTE DELIMITED BY SIZE INTO FIELD2.
   2.      STRING FIELD1A FIELD1A DELIMITED BY SIZE INTO FIELD2.
   3.      STRING FIELD1A FIELD1A DELIMITED BY "C" INTO FIELD2.
   4.      STRING FIELD1A FIELD1A FIELD1A FIELD1A
                DELIMITED BY "B" INTO FIELD2.
   5.      STRING FIELD1A FIELD1A "C" DELIMITED BY "C"
                INTO FIELD2.
   6.      MOVE 2 TO P.
           STRING FIELD1A "AC" DELIMITED BY "C"
                INTO FIELD2 WITH POINTER PNTR.
```

Figure 3-20
Various STRING Statements
Illustrating the Overflow Condition

The results of executing the numbered statements follow:

Table 3-1
Results of the
Preceding Sample Statements

Value of FIELD2 after the STRING operation	Overflow?
1. ABC"	NO
2. ABCA	YES
3. ABAB	NO
4. AAAA	NO
5. ABAB	YES
6. AABA	NO

3.7.5 Subscripted Fields in STRING Statements

All data-names used in the STRING statement may be subscripted, and the pointer value may be used as a subscript.

Since the pointer value might be used as a subscript on one or more of the fields in the statement, it is important to understand the order in which the software evaluates the subscripts and exactly when it updates the pointer. (The use of the pointer as a subscript is not specified by ANS-74 COBOL. Before using it, read the note at the end of this subsection.)

The software updates the pointer after it moves the last character out of each sending field. Consider the following sample coding:

```
MOVE 1 TO P.
STRING "ABC"
     SPACE
     "DEF" DELIMITED BY SIZE
     INTO R WITH POINTER P.
```

Figure 3-21
STRING Statement with Pointer

During the movement of "ABC" into the receiving field (R), the pointer value remains at 1. After the move, the software increases the pointer value by 3 (the size of the sending field literal "ABC") and the pointer takes on the value 4. The software then moves the figurative constant SPACE and increases the pointer value by 1 and it takes on the value 5. "DEF" is then moved and, on completion of the move, the software increases the pointer to its final value for this operation,8.

Now, consider the updating characteristics of the pointer when applied to subscripting:

```
MOVE 1 TO P.
STRING CHAR(P)
       CHAR(P)
       CHAR(P)
       CHAR(P) DELIMITED BY SIZE
       INTO R WITH POINTER P.
```

Figure 3-22
Subscripting with the Pointer

If CHAR is a one character field in a table, the pointer increases by one after each field has been moved and the software will move fields into R as if they had been subscripted as CHAR(1), CHAR(2), CHAR(3), and CHAR(4). If CHAR is a two character field, the pointer increases by two after each field has been moved and the fields will move into R as if they had been subscripted as CHAR(1), CHAR(3), CHAR(5), and CHAR(7).

Thus, the software evaluates the subscript of a sending item once, immediately before it considers the item as a sending item.

The software evaluates the subscript of a receiving item only once, at the start of the STRING operation. Therefore, if the pointer is used as a subscript on the receiving field, changes occurring to the

pointer during the execution of the STRING statement will not alter the choice of which receiving string is altered.

Even the delimiter field can be subscripted, and it too can be subscripted with the pointer. The software re-evaluates the delimiter subscript once for each sending field, immediately before it compares the delimiter to the field. Thus, by subscripting it with the pointer value, the delimiter can be changed for each sending field. This has the peculiar effect of choosing the next sending field's delimiter based on the position, in the receiving field, into which its first character will fall. For example, consider the following sample coding:

```
01   DTABLE.
     03 D PIC X OCCURS 7 TIMES.
      )              (
      MOVE 1 TO P.
      STRING "ABC"
             "ABC"
             "ABC" DELIMITED BY D (P)
             INTO R WITH POINTER P.
```

Figure 3-23
Subscripting the Delimiter

The following table shows the value that will arrive in the receiving field (R) from the three "ABC" literals if DTABLE contains the values shown in the left-hand column:

Table 3-2
Results of the
Preceding Sample Statements

DTABLE Value	R Value
ABCDEFG	(Unchanged)
BCDEFGH	AABABC
CDEFGHI	ABABCABC
CCCCCCCC	ABABAB

NOTE

The rules in this section, concerning subscripts in the STRING statement, are rules that are not specified by 1974 American National Standard COBOL. Dependence on these rules, particularly those involving the use of the pointer field as a subscript, may produce programs that will not perform the same way on other COBOL compilers.

If the pointer field is not used as a subscript on any of the fields in the statement, the point at which the software evaluates the subscripts is immaterial to the execution of the statement. Thus, by avoiding the use of the pointer as a subscript, uniform results can be expected from all COBOL compilers that adhere to 1974 ANS COBOL.

3.7.6 Common Errors, STRING Statement

The most common errors that programmers make when writing STRING statements are:

- using the word "TO" instead of "INTO"
- forgetting to write "DELIMITED BY SIZE";
- forgetting to initialize the pointer;
- initializing the pointer to 0 instead of 1;
- forgetting to provide for space fill of the receiving field when it is desirable.

3.8 THE UNSTRING STATEMENT

The UNSTRING statement disperses the contents of a single sending field into multiple receiving fields.

The statement has many forms; the simplest is equivalent, in function to a non-numeric MOVE statement. Consider the following illustration; the sample statement is equivalent to MOVE FIELD1 TO FIELD2, regardless of the relative sizes of the two fields.

```
UNSTRING FIELD1 INTO FIELD2.
```

Figure 3-24
Sample UNSTRING Statement

The sending field (FIELD1) may be either a group item, or an alphanumeric or alphanumeric edited elementary item. The

receiving field (FIELD2) may be alphabetic, alphanumeric, or numeric, but it cannot specify any type of editing.

If the receiving field is numeric, it must be DISPLAY usage. The picture-string of a numeric receiving field may contain any of the legal numeric description characters except for P and, of course, the editing characters. The UNSTRING statement moves the sending field to numeric receiving fields as if the sending field had been described as an unsigned integer; further, it automatically truncates or zero fills as required.

If the receiving field is not numeric, the software follows the rules for elementary non-numeric MOVE statements. It left-justifies the data in the receiving field, truncating or space-filling as required. (If the data-description of the receiving field contains a JUSTIFIED clause, the software right-justifies the data, truncating or space filling to the left as required.)

3.8.1 Multiple Receiving Fields

An important characteristic of the UNSTRING statement is its ability to disperse one sending field into several receiving fields. Consider the following example of the UNSTRING statement written with multiple receiving fields:

```
UNSTRING FIELD1 INTO
        FIELD2A FIELD2B FIELD2C.
```

Figure 3-25
Multiple Receiving Fields

In this sample statement, FIELD1 is the sending field. The software performs the UNSTRING operation by scanning across FIELD1 from left-to-right. When the number of characters scanned is equal to the number of characters in the receiving field, the software moves the scanned characters into the receiving field and begins scanning the next group of characters for the next receiving field.

Assume that each of the receiving fields in the preceding illustration (FIELD2A, FIELD2B, and FIELD2C) is five characters long, and that FIELD1 is 15 characters long. The size of FIELD2A determines the number of characters for the first move. The software scans across FIELD1 until the number of characters scanned equals the size of FIELD2A (5). It then moves those first five characters to FIELD2A, and sets the scanner to the next (sixth) character position in FIELD1. The size of FIELD2B determines the size of the next move. The software begins this move by scanning across FIELD1 from character position six, until the number of scanned characters equals the size of FIELD2B (5). It then moves the sixth through the tenth characters to FIELD2B, and sets the scanner to the next (eleventh) character position in FIELD1. FIELD2C determines the size of the last move (for this example) and causes characters 11 through 15 of FIELD1 to be moved into FIELD2C, thus terminating this UNSTRING operation.

Each data movement acts as an individual MOVE statement, the sending field of which is an alphanumeric field equal in size to the receiving field. If the receiving field is numeric, the move operation will convert the data to the numeric form. For example, consider what

would happen if the fields under discussion had the data descriptions and were manipulating the values shown in the following table:

Table 3-3
Values Moved Into the Receiving Fields
Based on the Value in the Sending Field

FIELD1 PIC X(15). VALUE IS:	FIELD2A PIC X(5)	FIELD2B PIC S9(5) LEADING SEPARATE	FIELD2C PIC S999V99
ABCDE1234512345	ABCDE	+12345	3450{
XXXXX0000100123	XXXXX	+00001	1230{

FIELD2A is an alphanumeric field and, since it is, the software simply conducts an elementary non-numeric move with the first five characters.

FIELD2B, however, has a leading separate sign that is not included in its size. Thus, the software moves only five numeric characters and generates a positive sign in the separate sign position.

FIELD2C has an implied decimal point with two character positions to the right of it, plus an overpunched sign on the low order digit. The sending field should supply five numeric digits; but, since the sending field is alphanumeric, the software treats it as an unsigned integer; it truncates the two high-order digits and supplies two zero digits for the decimal positions. Further, it supplies a positive overpunch sign, making the low-order digit a +0 (or the ASCII character, {). (There is no simple way to have UNSTRING recognize a sign character or a decimal point in the sending field.)

If the sending field is shorter than the sum of the sizes of the receiving fields, the software ignores the remaining receiving fields. If it reaches the end of the sending field before it reaches the end of one of the receiving fields, the software moves the scanned characters into that receiving field. It left justifies and space fills the remaining character positions for alphanumeric data, or decimal point aligns and zero fills the remaining character positions for numeric data. Consider the following examples of a sending field that is too short. (The statement is UNSTRING FIELD1 INTO FIELD2A FIELD2B. FIELD2A is a 3-character alphanumeric field, and receives the first three characters of FIELD1 (ABC) in every operation. FIELD2B, however, runs out of characters every time before filling. Since FIELD2A always contains the characters ABC, it is not shown.)

Table 3-4
Handling a Sending Field that is Too Short

FIELD1 PIC X(6) VALUE IS:	FIELD2B PICTURE IS:	FIELD2B Value after UNSTRING Operation
ABCDEF	XXXXX	DEF
ABC246	S99999	0024F
	S9V999	600{
	S9999 LEADING SEPARATE	+0246

3.8.2 The DELIMITED BY Phrase

The size of the data to be moved can be controlled by a delimiter, rather than by the size of the receiving field. The DELIMITED BY phrase supplies the delimiter characters.

UNSTRING delimiters are quite flexible; they can be literals, figurative constants (including ALL literal), or identifiers (identifiers may even be subscripted data-names). This subsection discusses the use of these three types of delimiters. Subsequent sections cover multiple delimiters, the COUNT phrase, and the DELIMITER phrase. Subscripting delimiters is discussed at the end of this section under Subscripted Fields in UNSTRING Statements.

Consider the following sample UNSTRING statement; it uses the figurative constant, SPACE, as a delimiter:

```
UNSTRING FIELD1 DELIMITED BY SPACE INTO FIELD2.
```

Figure 3-26
Delimiting with a Space Character

In this example, the software scans the sending field (FIELD1), searching for a space character. If it encounters a space, it moves all of the scanned (non-space) characters that precede that space to the receiving field (FIELD2). If it finds no space character, it moves the entire sending field. When it has determined the size of the sending field, the software moves the contents of that field following the rules for the MOVE Statement, truncating or zero filling as required.

The following table shows the results of an UNSTRING operation that delimits with a literal asterisk (UNSTRING FIELD1 DELIMITED BY "*" INTO FIELD2).

Table 3-5
Results of Delimiting with an Asterisk

FIELD1 PIC X(6) VALUE IS:	FIELD2 PICTURE IS:	FIELD2 VALUE AFTER UNSTRING
ABCDEF	XXX	ABC
	X(7)	ABCDEFΔ
	XXX JUSTIFIED	DEF
******	XXX	ΔΔΔ
*ABCDE	XXX	ΔΔΔ
A*****	XXX JUSTIFIED	ΔΔA
246***	S9999	024F
12345*	S9999 SEPARATE TRAILING	2345+
2468**	S999V9 SEPARATE LEADING	+4680
*246**	9999	0000

If the delimiter matches the first character in the sending field, the software considers the size of the sending field to be zero. The movement operation still takes place, however, and fills the receiving field with spaces or zeroes depending on its class.

A delimiter may also be applied to an UNSTRING statement that has multiple receiving fields:

```
UNSTRING FIELD1 DELIMITED BY SPACE
         INTO FIELD2A FIELD2B.
```

Figure 3-27
Delimiting with Multiple Receiving Fields

The sample instruction in Figure 3-27 causes the software to scan FIELD1 searching for a character that matches the delimiter. If it finds a match, it moves the scanned characters to FIELD2A and sets the scanner to the next character position to the right of the character that matched. It then resumes scanning FIELD1 for a character that matches the delimiter. If it finds a match, it moves all of the characters that lie between the character that first matched the delimiter and the character that matched on the second scan, and sets the scanner to the next character position to the right of the character that matched. (The DELIMITED BY phrase could handle additional receiving fields in the same manner as it handled FIELD2B.)

The following table shows the results of an UNSTRING operation that applies a delimiter to multiple receiving fields (UNSTRING FIELD1 DELIMITED BY "*" INTO FIELD2A FIELD2B).

Table 3-6
Results of Delimiting
Multiple Receiving Fields

FIELD1 PIC X(8) VALUE IS:	VALUES AFTER UNSTRING OPERATION	
	FIELD2A PIC X(3)	FIELD2B PIC X(3)
ABC*DEF*	ABC	DEF
ABCDE*FG	ABC	FGΔ
A*B*****	AΔΔ	BΔΔ
*AB*CD**	ΔΔΔ	ABΔ
**ABCDEF	ΔΔΔ	ΔΔΔ
A*BCDEFG	AΔΔ	BCD
ABC**DEF	ABC	ΔΔΔ
A******B	AΔΔ	ΔΔΔ

The last two examples illustrate the limitations of a single character delimiter. Accordingly, the delimiter may be longer than one character and it may be preceded by the word ALL.

The following table shows the results of an UNSTRING operation that uses a 2-character delimiter (UNSTRING FIELD1 DELIMITED BY "**" INTO FIELD2A FIELD2B):

Table 3-7

Results of Delimiting
with Two Asterisks

FIELD1 PIC X(8) VALUE IS:	VALUES AFTER UNSTRING OPERATION	
	FIELD2A PIC XXX	FIELD2B PIC XXX JUSTIFIED
ABC**DEF	ABC	DEF
A*B*C*D*	A*B	ΔΔΔ
AB***C*D	ABΔ	C*D
AB**C*D*	ABΔ	*D*
AB**CD**	ABΔ	ΔCD
AB***CD*	ABΔ	CD*
AB*****CD	ABΔ	ΔΔΔ

Unlike the STRING statement, the UNSTRING statement accepts the ALL literal as a delimiter. When the word ALL precedes the delimiter, the action of the UNSTRING statement remains essentially the same as with one delimiter until the scanning operation finds a match. At this point, the software scans farther, looking for additional consecutive strings of characters that also match the delimiter item. It considers the "ALL delimiter" to be one, two, three, or more adjacent repetitions of the delimiter item.

The following table illustrates the results of an UNSTRING operation that uses an ALL delimiter (UNSTRING FIELD1 DELIMITED BY ALL "*" INTO FIELD2A FIELD2B).

Table 3-8
Results of Delimiting
with ALL Asterisks

FIELD1 PIC X(8) VALUE IS:	VALUES AFTER UNSTRING OPERATION	
	FIELD2A PIC XXX	FIELD2B PIC XXX JUSTIFIED
ABC*DEF*	ABC	DEF
ABC**DEF	ABC	DEF
A******F	AΔΔ	ΔΔF
A*F*****	AΔΔ	ΔΔF
A*CDEFG*	AΔΔ	EFG

The next table illustrates the results of an UNSTRING operation that combines ALL with a 2-character delimiter (UNSTRING FIELD1 DELIMITED BY ALL "**" INTO FIELD2A FIELD2B).

Table 3-9

Results of Delimiting with
ALL Double Asterisks

FIELD1 PIC X(8) VALUE IS:	VALUES AFTER UNSTRING OPERATION	
	PIC XXX	PIC XXX JUSTIFIED
ABC**DEF	ABC	DEF
AB**DE**	ABΔ	ΔDE
A***D***	AΔΔ	Δ*D
A*******	AΔΔ	ΔΔ*

In addition to unchangeable delimiters, such as literals and figurative constants, delimiters may be designated by identifiers. Identifiers (which may even be subscripted data-names) permit variable delimiting. Consider the following sample statement:

```
UNSTRING FIELD1 DELIMITED BY DEL1
        INTO FIELD2A FIELD2B.
```

Figure 3-28
Delimiting with an Identifier

The data-name, DEL1, must be alphanumeric. It may be a group or elementary item, and it may be edited. (Since the delimiter is not a receiving field, any editing characters will not affect its use, other than contributing to the size of the item.)

If the delimiter contains a subscript, the subscript may be varied as a side effect of the UNSTRING operation. The evaluation of subscripts is discussed later in this section.

3.8.2.1 Multiple Delimiters - The UNSTRING statement has the ability to scan a sending field, searching for a match from a list of delimiters. This list may contain ALL delimiters and delimiters of various sizes. The only requirement of the list is that delimiters must be connected by the word OR.

The following sample statement separates a sending field into three receiving fields. The sending field consists of three strings separated by the following: (1) any number of spaces, or (2) a comma followed by a single space, or (3) a single comma, or (4) a tab character, or (5) a carriage return character. (The ",Δ" must precede the "," in the list if it is ever to be recognized.)

```
UNSTRING FIELD1 DELIMITED BY
        ALL SPACE OR
        ",Δ" OR
        "," OR
        TAB OR
        CR
        INTO FIELD2A FIELD2B FIELD2C.
```

Figure 3-29
Multiple Delimiters

The following table illustrates the potential of this statement. The tab (represented by the letter t) and carriage return (represented by the letter r) characters represent single character fields containing the ASCII horizontal tab and carriage return characters.

Table 3-10
Results of the Multiple Delimiters
Shown in Figure 3-29

FIELD1 PIC X(12)	FIELD2A PIC XXX	FIELD2B PIC 9999	FIELD2C PIC XXX
A,0,Cr	AΔΔ	0000	CΔΔ
At456,ΔE	AΔΔ	0456	EΔΔ
AΔΔΔ3ΔΔΔ9	AΔΔ	0003	9ΔΔ
AttBr	AΔΔ	0000	BΔΔ
A,,C	AΔΔ	0000	CΔΔ
ABCD,Δ4321,Z	ABC	4321	ZΔΔ

t--tab character, r--carriage return character

3.8.3 The COUNT Phrase

The COUNT phrase keeps track of the size of the sending string and stores the length in a user-supplied data area.

The length of a delimited sending field may vary widely (from zero to the full length of the field) and some programs may require knowledge of this length. For example, if it exceeds the size of the receiving field (which is fixed in size) same data may be truncated and the program's logic may require this information.

To use the phrase, simply follow the receiving field name with the words COUNT IN and an identifier. Consider the following sample statement:

```
UNSTRING FIELD1 DELIMITED BY ALL "*"
         INTO FIELD2A COUNT IN COUNT2A
         FIELD2B COUNT IN COUNT2B
         FIELD2C.
```

Figure 3-30
The COUNT Phrase

In this sample statement, the software will count the number of characters between the left-hand end of FIELD1 and the first asterisk in FIELD1 and place that value into COUNT2A; thus, COUNT2A contains the size of the first sending string. The software does not include the delimiter in the count (as it is not a part of the string).

The software then counts the number of characters in the second sending field and places that value into COUNT2B.

The phrase should be used only where needed; in this example the length of the string moved to FIELD2C is not needed, so no COUNT phrase follows it.

If the receiving field is shorter than the value placed in its count field, the software truncates the sending string. (If the number of integer positions in a numeric field is smaller than the value placed into its count field, high order numeric digits have been lost.)

If the software finds a delimiter match on the first character it examines, it places a zero in the count field.

The count field must be described as a numeric integer, either COMP or DISPLAY usage, with no editing symbols nor the character P in its picture-string. The software moves the count value into the count field according to the rules for an elementary numeric MOVE statement.

The COUNT phrase may be used only in conjunction with the DELIMITED BY phrase.

3.8.4 The DELIMITER Phrase

The DELIMITER phrase causes the actual character or characters that delimited the sending field to be stored in a user-supplied data area. This phrase is most useful when: (1) the statement contains a delimiter list, (2) any one of the items in the list might have delimited the field, and (3) program logic flow depends on which one found a match. In fact, the DELIMITER and COUNT phrases could be used together and program logic flow could depend on both the size of the sending string and the delimiter character that terminated it.

To use the DELIMITER phrase, simply follow the receiving field name with the words DELIMITER IN and an identifier. (The software places the delimiter character in the area named by the identifier.) Consider the following sample UNSTRING statement:

```
UNSTRING FIELD1 DELIMITED BY "," OR TAB OR
         ALL SPACE OR CR
         INTO FIELD2A DELIMITER IN DELIMA
         FIELD2B DELIMITER IN DELIMB
         FIELD2C.
```

Figure 3-31
The DELIMITER Phrase

After moving the first sending string to FIELD2A, the software takes the character (or characters) that delimited that string and places it in DELIMA. DELIMA then contains a comma, or a tab, or a carriage return, or any number of spaces. Since the delimiter string is moved under the rules of the elementary non-numeric MOVE statement, the software truncates or space fills with left or right justification (depending on its data description).

The software then moves the second sending string to FIELD2B and places its delimiting character into DELIMB.

When a sending string is delimited by the end of the sending field (rather than a match on a delimiter) the delimiter string is of zero length. This causes the DELIMITER item to be space filled. The phrase should be used only where needed; in this example, the character that delimits the last sending string is not needed, so no DELIMITER phrase follows FIELD2C.

The data item named in the DELIMITER phrase must be described as an alphanumeric item. It may contain editing characters and it may even be a group item.

When the DELIMITER and COUNT phrases are used together, they must appear in the correct order (DELIMITER phrase preceding the COUNT phrase). Both of the data items named in these phrases may be subscripted or indexed. If they are subscripted, the subscript may be varied as a side effect of the UNSTRING operation. (The evaluation of subscripts is discussed at the end of this section.)

3.8.5 The POINTER Phrase

Although the UNSTRING statement normally starts at the left-hand end of the sending field, the POINTER phrase permits the user to select a character position in the sending field for the software to begin scanning. (The scanning, however, remains left-to-right.)

When a sending field is to be dispersed into multiple receiving fields, it often happens that the choice of delimiters, the size of subsequent receiving fields, etc. depend on the value in the first sending string or the character that delimited that string. Thus, the program may need to move the first field, hold its place in the sending field, and examine the results of the operation to determine how to handle the sending items that follow. This is done by using an UNSTRING statement with a POINTER phrase that fills only the first receiving field. When the first string has been moved to a receiving item, the software updates the pointer data item with a new position (one character beyond the delimiter that caused the interruption) to begin the next scanning operation. The program may then examine the new position, the receiving field, the delimiter value, the sending string size, and resume the scanning operation by executing another UNSTRING statement with the same sending field and pointer data item. Thus, the UNSTRING statement can move one sending string at a time, with the form of each move being dependent on the context of the preceding string of data.

The POINTER phrase must follow the last receiving item in the statement. Consider the following two UNSTRING statements with their accompanying POINTER phrases and tests:

```
MOVE 1 TO P.
UNSTRING FIELD1 DELIMITED BY
         ":" OR TAB OR CR OR ALL SPACE
         INTO FIELD2A
         DELIMITER IN DELIMA
         COUNT IN LSIZEA
         WITH POINTER PNTR.
IF LSIZEA = 0 GO TO NO-LABEL-PROCESS.
IF DELIMA = ":"
         IF PNTR > 8 GO TO BIG-LABEL-PROCESS
         ELSE GO TO LABEL-PROCESS.
IF DELIMA = TAB GO TO BAD-LABEL-PROCESS.

     ...

UNSTRING FIELD1 DELIMITED BY ... WITH POINTER PNTR.
```

Figure 3-32
The POINTER Phrase

PNTR contains the current position of the scanner in the sending field. The second UNSTRING statement uses PNTR to begin scanning the additional sending strings in FIELD1.

Since the software considers the left-most character to be character position one, the value returned by PNTR may be used to examine the next character. To do this, simply use PNTR as a subscript on the sending field (providing that the sending field is also described as a table of characters). For example, consider the following sample coding:

```
01 FIELD1.
   02 FIELD1-CHAR OCCURS 40 TIMES.

     ...

   UNSTRING FIELD1
            ...
            WITH POINTER PNTR.
   IF FIELD1-CHAR(PNTR) = "X"...
```

Figure 3-33
Examining the Next Character
By Using the Pointer Data
Item as a Subscript

Another way to examine the next character of the sending field is to use the UNSTRING statement to move it to a 1-character receiving field. Consider the following sample coding:

```
UNSTRING FIELD1
         ...
         WITH POINTER PNTR.
UNSTRING FIELD1 INTO CHAR1 WITH POINTER PNTR.
SUBTRACT 1 FROM PNTR.
IF CHAR1 = "X" ...
```

Figure 3-34
Examining the Next Character
By Placing It Into a 1-Character Field

The program must decrement PNTR in order for this case to work like the one illustrated in Figure 3-33, since the second UNSTRING statement will increment the pointer value by 1.

The program must initialize the POINTER phrase data item before the UNSTRING statement uses it. The software will terminate the UNSTRING operation if the initial value of the pointer is less than one or greater than the length of the sending field. (A pointer value that is less than one or greater than the length of the sending field causes an overflow condition. Overflow conditions are discussed later in this section.)

The POINTER and TALLYING phrases may be used together in the same UNSTRING statement; but, when both are used, the POINTER phrase must precede the TALLYING phrase.

3.8.6 The TALLYING Phrase

The TALLYING phrase counts the number of receiving fields that received data from the sending field.

When an UNSTRING statement contains several receiving fields, the possibility exists that there may not always be as many sending strings as there are receiving fields. The TALLYING phrase provides a convenient method for keeping a count of how many fields were acted upon.

```
MOVE 0 TO RCOUNT.
UNSTRING FIELD1 DELIMITED BY "," OR ALL SPACE
         INTO FIELD2A
              FIELD2B
              FIELD2C
              FIELD2D
              FIELD2E
              TALLYING IN RCOUNT.
```

Figure 3-35
The TALLYING Phrase

If the software has moved only three sending strings when it reaches the end of FIELD1, it adds 3 to RCOUNT. The first three fields (FIELD2A, FIELD2B, and FIELD2C) contain data from the operation, and the last two (FIELD2D and FIELD2E) do not.

The TALLYING data item always contains the sum of its initial contents plus the number of sending strings acted upon by the UNSTRING command just executed. Thus, the programmer may want to initialize the tally count before each use.

When used in the same statement with a POINTER phrase, the TALLYING phrase must follow the POINTER phrase and both phrases must follow all of the field names, the DELIMITER and COUNT phrases. The data items for both phrases must contain numeric integers, that is, be without editing characters or the letter P in their picture-strings; both data items may be in either COMP or DISPLAY usage. They may be signed or unsigned and, if they are DISPLAY usage, they may contain any desired sign option.

The data items for both phrases may be subscripted or indexed, or they may be used as subscripts on other fields in the statement. (The evaluation of subscripts is discussed toward the end of this section.) A convenient use of the TALLYING phrase data item is as a subscript of a receiving field. Consider the following sample coding which causes program control to execute the UNSTRING statement repeatedly until it exhausts the sending field.

```
      MOVE 1 TO PNTR, TLY.
PAR1. UNSTRING FIELD1 DELIMITED BY "," OR CR
               INTO FIELD2(TLY)
               DELIMITER IN DEL2
               WITH POINTER PNTR
               TALLYING IN TLY.
      IF DEL2 = "," GO TO PAR1.
```

Figure 3-36
The POINTER and TALLYING Phrases
Used Together

This sample coding causes program control to loop through the UNSTRING statement, using the pointer, PNTR, to scan across FIELD1 with successive executions. Each comma isolates a sending string until control reaches either a carriage return character or the end of FIELD1. If it reaches the end of the field without encountering a carriage return character, the software places a space into the delimiter field, DEL2, and control falls through the IF statement and out of the loop.

Since the TALLYING data item, TLY, is increased by 1 after each data movement, it serves as a subscript on the receiving field. In effect this causes the software to unpack the value in FIELD1 into an array of fixed-size fields. Further, an array of COUNT data items can be supplied and loaded by the UNSTRING/TALLYING statement by adding the following phrase to the coding in Figure 3-36:

```
COUNT IN C(TLY)
```

Figure 3-37
Subscripting the COUNT Phrase
With the TALLYING Data Item

The TALLYING data item, in the above example, is one greater than the number of receiving fields acted upon by the UNSTRING operation. This is because the data item must be initialized to a value of one in order to be used as a subscript for the first receiving item.

3.8.7 The OVERFLOW Phrase

The OVERFLOW phrase detects the overflow condition and provides an imperative statement to be executed when it detects the condition. An overflow condition exists when either of the following two situations occurs:

1. The UNSTRING statement is about to be executed and its pointer data item contains a value of less than one or greater than the size of the sending field. When it detects this situation, the software executes the OVERFLOW phrase before it moves any data. Thus, the values of all of the receiving fields remain unchanged.

2. The UNSTRING statement has filled all of the receiving fields and data still remains in the sending field that has not been matched as a delimiter or included in a sending string. When it detects this situation, the software executes the OVERFLOW phrase after it has executed the UNSTRING statement. Thus, the values of all of the receiving fields are updated, but some data has not been moved.

If the UNSTRING operation causes the scanner to move off the end of the sending field (thus exhausting it), the software will not execute the OVERFLOW phrase.

Consider the following set of instructions which cause program control to execute the UNSTRING statement repeatedly until it exhausts the sending field. The TALLYING data item is a subscript indexing the receiving field. (Compare this loop with the one in Figure 3-36 which accomplishes the same thing.)

```
      MOVE 1 TO TLY PNTR.
PAR1. UNSTRING FIELD1 DELIMITED BY "," OR CR
               INTO FIELD2(TLY)
               WITH POINTER PNTR
               TALLYING IN TLY
               ON OVERFLOW GO TO PAR1.
```

Figure 3-38
Using the OVERFLOW Phrase

NOTE

The overflow condition also occurs if the value of a pointer data item lies outside the sending field at the start of execution of the UNSTRING statement. (The pointer value must not be less than 1, nor greater than the length of the sending field.) This type of overflow is not distinguishable from the overflow condition described at the start of this section, except that this condition causes the UNSTRING statement to terminate before any data movement takes place. Then, the values of all receiving fields remain unchanged.

3.8.8 Subscripted Fields in UNSTRING Statements

Since the flexibility of the UNSTRING statement is enhanced by subscripting and indexing and particularly by subscripting with other fields within the statement (such as subscripting the receiving field with the TALLYING data item as discussed above), it is important to understand how often and exactly when the software evaluates these subscripts and indexes. This subsection discusses the frequency and times of subscript evaluation.

The software evaluates subscripts and indexes on the following items only once, at the initiation of the UNSTRING statement; thus, any change in subscript values during the execution of the statement has no effect on these fields:

1. Sending field,
2. POINTER data item,
3. TALLYING data item.

The software evaluates subscripts and indexes on the following items immediately before it moves data into the item. It moves the data to these items in the order in which they are listed in the statement (which is the same order as below):

1. Receiving field,
2. DELIMITER data item,
3. COUNT data item.

The software evaluates any subscripts and indexes on the data-names in the DELIMITED BY phrase (delimiters) immediately before it scans each sending string looking for a delimiter match. Thus, it re-evaluates these data-names once for each receiving field in the statement.

If any of the following items are used as subscripts on any receiving fields, the programmer must be aware of the point at which these items are updated:

- POINTER data-item,
- TALLYING data-item,
- COUNT data-item,
- Another receiving field.

Figure 3-39 illustrates, with a flow chart, the sequence of evaluation operations:

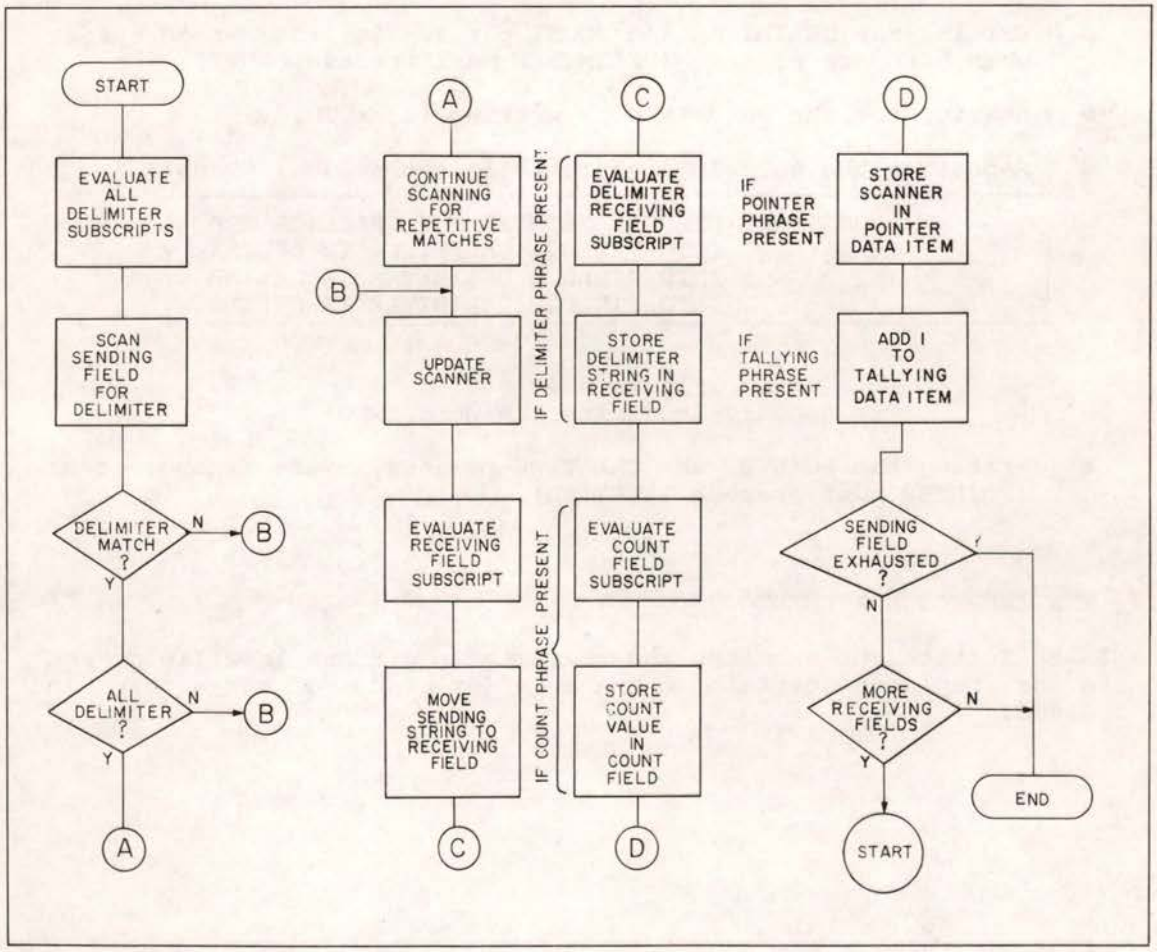


Figure 3-39
Sequence of Subscript Evaluation

NOTE

The rules in this section concerning the exact point at which the software evaluates the identifiers in the DELIMITED BY phrase and the point at which it updates the POINTER and TALLYING data items, are rules that are specified by 1974 American National Standard COBOL, as opposed to the STRING statement where these are not so specified.

3.8.9 Common Errors, UNSTRING Statement

The most common errors that programmers make when writing UNSTRING statements are:

- Leaving the OR connector out of a delimiter list;
- Misspelling or interchanging the words, DELIMITED and DELIMITER;
- Writing the DELIMITER and COUNT phrases in the wrong order when both are present (DELIMITER must precede COUNT);
- Leaving out the word INTO or writing it as TO;
- Repeating the word INTO where it is not needed; thus:

```
UNSTRING FIELD1 DELIMITED BY SPACE OR TAB
         INTO FIELD2A DELIMITER IN DELIMA
         INTO FIELD2B DELIMITER IN DELIMB
         INTO FIELD2C DELIMITER IN DELIMC.
```

Figure 3-40
Erroneously Repeating the Word INTO

- Writing the POINTER and TALLYING phrases in the wrong order (POINTER must precede TALLYING).

3.9 THE INSPECT STATEMENT

The INSPECT statement examines the character positions in a field and counts or replaces certain characters (or groups of characters) in that field.

Like the STRING and UNSTRING operations, INSPECT operations scan across the field from left to right; further, like those two statements, the INSPECT statement features a phrase which allows it to begin or terminate the scanning operation with a delimiter match. (Thus, the operation can begin within the field instead of at the left-hand end, or it may begin at the left-hand end and terminate within the field.)

The TALLYING operation (which counts certain characters in the field) and the REPLACING operation (which replaces certain characters in the field) are quite versatile and may be applied to all of the characters in the delimited area of the field being inspected, or they may be applied only to those characters that match a given character string under stated conditions. Consider the following sample statements, which both cause a scan of the complete field:

```
INSPECT FIELD1 TALLYING TLY FOR ALL "B".
```

Figure 3-41
Sample INSPECT...TALLYING Statement

This statement scans FIELD1 looking for the character B. Each time it finds a B, it increments TLY by 1.

```
INSPECT FIELD1 REPLACING ALL SPACE BY ZERO.
```

Figure 3-42
Sample INSPECT...REPLACING Statement

This statement scans FIELD1 looking for space characters. Wherever it finds a space character, it replaces it with zero.

One INSPECT statement can contain both a TALLYING phrase and a REPLACING phrase. However, when used together, the TALLYING phrase must precede the REPLACING phrase. An INSPECT statement with both phrases is equivalent to two separate INSPECT statements and, in fact, the software compiles such a statement into two distinct INSPECT statements. (To simplify debugging, therefore, it is best to initially write the two phrases in separate INSPECT statements.)

3.9.1 The BEFORE/AFTER Phrase

The BEFORE/AFTER phrase acts as a delimiter and (possibly) restricts the area of the field being inspected.

The following sample statement would count only the zeros that precede the percent sign (%) in FIELD1.

```
INSPECT FIELD1 TALLYING TLY
        FOR ALL ZEROS BEFORE "%".
```

Figure 3-43
Sample INSPECT...BEFORE Statement

The delimiter (the percent sign in the preceding sample statement) can be a single character, a string of characters, or any figurative constant. Further, it can be either an identifier or a literal.

- If the delimiter is an identifier, it must be an elementary data item of DISPLAY usage. It may be alphabetic, alphanumeric, or numeric, and, it may contain editing characters. The compiler always treats the item as if it had been described as an alphanumeric string. (It does this by implicit redefinition of the item, as described in Section 3.9.2.)

- If the delimiter is a literal, it must be non-numeric.

The software repeatedly compares the delimiter character against an equal number of characters in the field being inspected. If none of the characters match the delimiter, or if insufficient characters remain in the field for a full comparison (at the right-hand end), the software considers the comparison to be unequal.

The following examples of the INSPECT statement illustrate the way the delimiter character finds a match in the field being inspected. (The portion of the field which the statement ignores as a result of the BEFORE/AFTER phrase delimiters is crossed out with a slash, and the portion which it inspects is underlined.)

INSTRUCTION	FIELD1 VALUE
INSPECT FIELD1...BEFORE "E".	ABCDEFGHI
INSPECT FIELD1...AFTER "E".	ABCDEFGHI
INSPECT FIELD1...BEFORE "K".	ABCDEFGHI
INSPECT FIELD1...AFTER "K".	ABCDEFGHI
INSPECT FIELD1...BEFORE "AB".	ABCDEFGHI
INSPECT FIELD1...AFTER "AB".	ABCDEFGHI
INSPECT FIELD1...BEFORE "HI".	ABCDEFGHI
INSPECT FIELD1...AFTER "HI".	ABCDEFGHI
INSPECT FIELD1...BEFORE "I Δ".	ABCDEFGHI
INSPECT FIELD1...AFTER "I Δ".	ABCDEFGHI
The ellipsis represents the position of the TALLYING or REPLACING phrase.	

Figure 3-44
Matching the Delimiter Characters
To the Characters in a field

The software scans the field for a delimiter match before it scans for the inspection operation (TALLYING or REPLACING), thus establishing the limits of the operation before beginning the actual inspection. The importance of the separate scan is discussed further in Section 3.9.3.

3.9.2 Implicit Redefinition

The software requires that certain fields referred to by the INSPECT statement be alphanumeric fields. If one of these fields was described as another data class, the compiler redefines that field so the INSPECT statement can handle it as a simple alphanumeric string. This implicit redefinition is conducted as follows:

- If the field was described as alphabetic, alphanumeric edited, or unsigned numeric, the compiler simply redefines it as alphanumeric. This is a compile-time operation; no data movement occurs at object-time.

- If the field was described as signed numeric, the compiler first removes the sign and then redefines the field as alphanumeric. If the sign is a separate character, the compiler ignores that character, essentially shortening the field, and that character does not participate in the implicit redefinition. If the sign is an "overpunch" on the leading or trailing digit, the compiler actually removes the sign value and leaves the character with only the numeric value that was stored in it. The compiler alters the digit position containing the sign before beginning the INSPECT operation and restores it to its former value after the operation. If the sign's digit position does not contain a valid ASCII signed numeric digit, the action of the redefinition causes the value to change. Table 3-10 shows these original, altered, and restored values.

The compiler never moves an implicitly redefined item from its storage position. All redefinition occurs in place.

The position of an implied decimal point on numeric quantities does not affect implicit redefinition.

Table 3-11

Original, Altered, and Restored Values Resulting from Implicit Redefinition

ORIGINAL VALUE	ALTERED VALUE	RESTORED VALUE
} (173)	0 (60)	} (173)
A (101)	1 (61)	A (101)
B (102)	2 (62)	B (102)
C (103)	3 (63)	C (103)
D (104)	4 (64)	D (104)
E (105)	5 (65)	E (105)
F (106)	6 (66)	F (106)
G (107)	7 (67)	G (107)
H (110)	8 (70)	H (110)
I (111)	9 (71)	I (111)
{ (175)	0 (60)	{ (175)
J (112)	1 (61)	J (112)
K (113)	2 (62)	K (113)
L (114)	3 (63)	L (114)
M (115)	4 (64)	M (115)
N (116)	5 (65)	N (116)
O (117)	6 (66)	O (117)
P (120)	7 (67)	P (120)
Q (121)	8 (70)	Q (121)
R (122)	9 (71)	R (122)
0 (60)	0 (60)	} (173)
1 (61)	1 (61)	A (101)
2 (62)	2 (62)	B (102)
3 (63)	3 (63)	C (103)
4 (64)	4 (64)	D (104)
5 (65)	5 (65)	E (105)
6 (66)	6 (66)	F (106)
7 (67)	7 (67)	F (107)
8 (70)	8 (70)	H (110)
9 (71)	9 (71)	I (111)
All other values	0 (60)	} (173)

3.9.3 The INSPECT Operation

Regardless of the type of inspection (TALLYING or REPLACING), the INSPECT statement has only one method for inspecting the characters in the field. This section describes this method.

However, before discussing how the inspection operation is conducted, let's analyze the INSPECT statement itself:

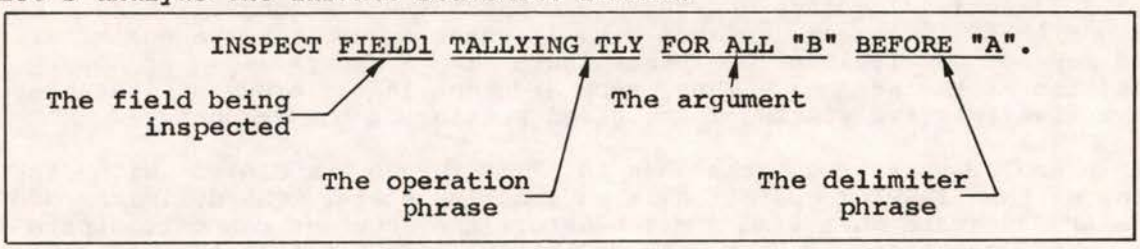


Figure 3-45
Sample INSPECT Statement

The format of the INSPECT statement requires that a field be named which is to be inspected (FIELD1 above); the field name must be followed by an operation phrase (TALLYING TLY above); and, that phrase must be followed by one or more identifiers or literals ("B" above). These identifiers or literals comprise the "arguments" (items to be compared to the field being inspected). More than one argument makes up the "argument list".

- TALLYING Arguments

  Each argument in an argument list can have other fields associated with it. Thus, each argument that is used in a TALLYING operation must have a tally counter (TLY above) associated with it. The software increments the tally counter each time it matches the argument with a character or group of characters in the field being inspected.

- REPLACING Arguments

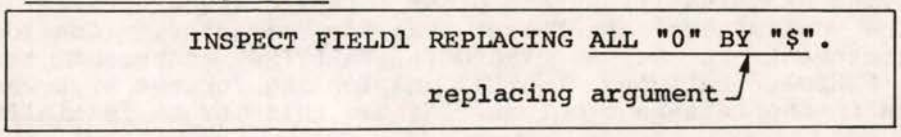


Figure 3-46
Sample REPLACING Argument

  Each argument in an argument list that is used in a REPLACING operation must have a replacement item ($ above) associated with it. The software uses the replacement item to replace each string of characters in the field that matches the argument.

Each argument in an argument list (that is used with either a TALLYING or REPLACING operation) may have a delimiter field (BEFORE/AFTER phrase) associated with it. If the delimiter field is not present, the software applies the argument to the entire field. If the delimiter field is present, the software applies the argument only to that portion of the field specified by the BEFORE/AFTER phrase.

3.9.3.1 Setting the Scanner - The INSPECT operation begins by setting the scanner to the left-most character position of the field being inspected. It remains on this character until an argument has been matched with a character (or characters) or until all arguments have failed to find a match at that position.

3.9.3.2 Active/Inactive Arguments - When an argument has a BEFORE/AFTER phrase associated with it, that argument has a delimiter and may not be eligible to participate in a comparison at every position of the scanner. Thus, each argument in the argument list has an active/inactive status at any given setting of the scanner.

For example, an argument that has an AFTER phrase associated with it starts the INSPECT operation in an inactive state. The delimiter of the AFTER phrase must find a match before the argument can participate in the comparison. When the delimiter finds a match, the software retains the character position beyond the matched character string; then, when the scanner reaches or passes this position, the argument becomes active.

```
INSPECT FIELD1 TALLYING TLY
        FOR ALL "B" AFTER "X".
```

Figure 3-47
Sample AFTER Delimiter Phrase

If FIELD1 in Figure 3-47 has a value of "ABABXZBA", the argument B remains inactive until the scanner finds a match for the delimiter X. Thus, argument B remains inactive while the software scans character positions 1 through 5. At character position 5, the delimiter X finds a match, and since the character position beyond the matched delimiter character is the point at which the argument becomes active, argument B is compared for the first time at character position 6. It finds a successful match at character position 7 and this causes TLY to be incremented by 1.

The following examples illustrate other situations where the arguments and/or the delimiters are longer than one character. (Consider the sample statement to be an INSPECT...TALLYING statement that is scanning FIELD1, tallying in TLY, and looking for the arguments and delimiters in the left-hand column. Assume that TLY is initialized to 0.)

ARGUMENT AND DELIMITER	FIELD1 VALUE	ARGUMENT ACTIVE AT POSITION	CONTENTS OF TLY AFTER SCAN
"B" AFTER "XX"	BXBXXXXBB	6	2
	XXXXXXXX	3	0
	BXBXBBBBXX	never	0
"X" AFTER "XX"	BXBXXBXXB	6	2
	XXXXXXXX	3	6
	BBBBBBXX	never	0
"B" AFTER "XB"	BXYBXBXX	7	0
	XBXBXBXB	3	3
	BBBBBBXB	never	0
"BX" AFTER "XB"	XXXXBXXXX	6	0
	XXXXBBXXX	6	1
	XXBXXXXBX	4	1

Figure 3-48
Where Arguments Become Active in a Field

When an argument has an associated BEFORE delimiter, the inactive/active states reverse roles: the argument is in an active state when the scanning begins, and becomes inactive at the character position that matches the delimiter. Additionally, regardless of the presence of the BEFORE delimiter, an argument becomes inactive when the scanner approaches the right-hand end of the field and the remaining characters are fewer in number than the characters in the argument. (In such a case, the argument cannot possibly find a match in the field so it becomes inactive.)

Since the BEFORE/AFTER delimiters are found on a separate scan of the field, the software recognizes and sets up the delimiter boundaries before it scans for an argument match; therefore, the same characters can be used as arguments and delimiters in the same phrase.

3.9.3.3 Finding an Argument Match - The software selects arguments from the argument list in the order in which they appear in the list. If the first one it selects is an active argument and the conditions stated in the INSPECT statement allow a comparison, the software compares it to the character at the position of the scanner. If the active argument does not find a match, the software takes the next active argument from the list and compares that to the same character. If none of the active arguments find a match, the scanner moves one position to the right and begins the inspection operation again with the first active argument in the list. The inspection operation terminates at the right-hand end of the field.

When an active argument does find a match, the software ignores any remaining arguments in the list and conducts the TALLYING or REPLACING operation on the character. The scanner moves to a new position and the next inspection operation begins with the first argument in the list. (The INSPECT statement may contain additional conditions, which are described later in this section; this discussion, however, assumes that the argument match is allowed to take place and that inspection is allowed to continue following the match.)

The software updates the scanner by adding the size of the matching argument to it. This moves the scanner to the next character beyond the string of characters that matched the argument. Thus, once an active argument matches a string of characters, the statement does not inspect those character positions again unless program control executes the entire statement again.

3.9.4 Subscripted Fields in INSPECT Statements

Any identifier named in an INSPECT statement may be subscripted or indexed.

The software evaluates all subscripts in an INSPECT statement once, before the inspection begins; therefore, if the action of the INSPECT statement alters one of the subscripts, the new subscript value has no effect on the selection of operands during that inspection operation. For example, consider the following illustration:

```
MOVE 1 TO TLY.
INSPECT FIELD1 TALLYING TLY
        FOR ALL X(TLY).
```

Figure 3-49
Sample Subscripted Argument

In this sample statement, the software evaluates the address of X(TLY) only once, before it begins inspecting the field; hence, it will evaluate X(TLY) as X(1). The alteration of TLY by the action of inspecting and tallying has no effect on the choice of the X operand. (X(1) will be used throughout the operation.)

NOTE

When subscripting an INSPECT statement that contains both a TALLYING and a REPLACING phrase, keep in mind that the statement will be compiled into two separate INSPECT statements. Therefore, any field that is altered by the action of the INSPECT...TALLYING statement will be in its altered state if used as a subscript by the INSPECT...REPLACING statement.

3.9.5 The TALLYING Phrase

An INSPECT statement that contains a TALLYING phrase counts the occurrence of various character strings under certain stated conditions. It keeps the count in a user-designated field called, here, a tally counter.

3.9.5.1 The Tally Counter - The identifier which follows the word TALLYING designates the tally counter. The identifier may be subscripted or indexed. The data item must be a numeric integer with no editing or P characters; it may be COMP or DISPLAY usage, and it may be signed (separate or overpunched).

Each time the tally argument matches the delimited string being inspected, the software adds 1 to the tally counter.

The programmer can initialize the tally counter to any numeric value. (The INSPECT statement does not initialize it.)

3.9.5.2 The Tally Argument - The tally argument specifies a character-string and a condition under which that string should be compared to the delimited string being inspected. The following figure shows the format of the tally argument:

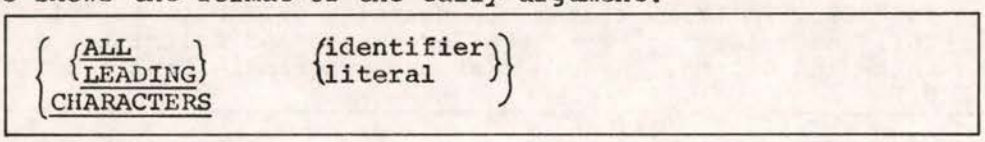


Figure 3-50
Format of the Tally Argument

The CHARACTERS form of the tally argument specifies that every character in the delimited string being inspected should be considered to match an imaginary character which serves as the tally argument. This increments the tally counter by a value that equals the size of the delimited string. For example, the statement in the following illustration causes TLY to be incremented to the number of characters that precede the first comma, regardless of what those characters might be.

```
INSPECT FIELD1 TALLYING TLY FOR
        CHARACTERS BEFORE ",".
```

Figure 3-51
CHARACTERS Form of the Tally Argument

The ALL and LEADING forms of the tally argument specify a particular character string, which may be represented by either a literal or an identifier. The tally argument character string may be any length; however, each character of the argument must match a character in the delimited string before the software considers the argument matched.

- A literal character string must be either non-numeric or a figurative constant (other than ALL literal). A figurative constant, such as SPACE, ZERO, etc., represents a single character and can be written as " ", or "0", etc., with the same effect.

- An identifier must be an elementary item of DISPLAY usage. It may be any data class. However, if it is other than alphanumeric, the software performs an implicit redefinition of the item. (This redefinition is identical to the BEFORE/AFTER delimiter redefinition discussed earlier in Section 3.8.2.)

The words ALL and LEADING supply conditions which further delimit the inspection operation.

- The word ALL specifies that every match that the search argument finds in the delimited character string be counted in the tally counter. When a literal follows the word ALL, it does not have the same meaning as the figurative constant, ALL literal. (The ALL literal meaning of ALL "," is a string of consecutive commas, as many as the context of the statement requires.) ALL "," used as a tally argument means, "count each comma without regard to adjacent characters."

- The word LEADING specifies that only adjacent matches of the TALLY argument at the left-hand end of the delimited character string be counted. At the first failure to match the tally argument, the software terminates counting and causes the argument to become inactive. Consider the following examples. (The sample statement is an INSPECT...TALLYING statement, scanning FIELD1, tallying in TLY, and looking for the arguments and delimiters in the left-hand column. Assume that the program initializes TLY to 0.)

ARGUMENT AND DELIMITER	FIELD1 VALUE	CONTENTS OF TLY AFTER SCAN
LEADING "*" AFTER "0".	F***0**F	2
	F**0F**	0
	F**F**0	0
	0***F**	3
LEADING "**" AFTER "0".	F**0**F***	1
	F**F0***FF**	1
	F**F0****F**	2
	F**F**0*	0

Figure 3-52
Results of Counting with the
LEADING Condition

3.9.5.3 The Tally Argument List - One INSPECT...TALLYING statement can contain more than one tally argument, and each argument can have a separate BEFORE/AFTER phrase and tally counter associated with it. These tally arguments with their associated tally counters and BEFORE/AFTER phrases form an argument list. The manner in which this list is processed affects the action of any given tally argument.

The following sample statements show INSPECT statements with argument lists. The text following each one tells how that list will be processed.

```
INSPECT FIELD1 TALLYING T FOR
        ALL ","
        ALL "."
        ALL ";".
```

Figure 3-53
Argument List Adding Into
One Tally Counter

These three tally arguments have the same tally counter, T, and are active over the entire field being inspected. Thus, this statement adds the total number of commas, periods, and semicolons in FIELD1 to the initial value of T.

```
INSPECT FIELD1 TALLYING
        T1 FOR ALL ","
        T2 FOR ALL "."
        T3 FOR ALL ";".
```

Figure 3-54
Argument List Adding Into
Separate Tally Counters

Each tally argument in this statement has its own tally counter, and is active over the entire field being inspected. Thus, the action of this statement is to add the total number of commas in FIELD1 to the initial value of T1, the total number of periods to the initial value of T2, and the number of semicolons to T3.

```
INSPECT FIELD1 TALLYING
        T1 FOR ALL "," AFTER "A"
        T2 FOR ALL "." BEFORE "B"
        T3 FOR ALL ";".
```

Figure 3-55
Argument List (with Delimiters) Adding
into Separate Tally Counters

Each tally argument in this statement has its own tally counter; the first two arguments have delimiter phrases, and the last one is active over the entire field being inspected. Thus, the first argument is initially inactive and becomes active only after the scanner encounters an A; the second argument begins the scan in the active state but becomes inactive after a B has been encountered; and the third argument is active during the entire scan of FIELD1.

The following Figure shows various values of FIELD1 and the contents of the three tally counters after the scan. Assume that the counters are initialized to 0 before the INSPECT statement.

FIELD1 VALUE	CONTENTS OF TALLY COUNTERS AFTER SCAN		
	T1	T2	T3
A.C;D.E,F	1	2	1
A.B.C.D	0	1	0
A,B,C,D	3	0	0
A;B;C;D	0	0	3
*,B,C,D	0	0	0

Figure 3-56
Results of the Scan in Figure 3-55

The BEFORE/AFTER phrase applies only to the argument that precedes it, and delimits the field for that argument only. Each BEFORE/AFTER phrase causes a separate scan of the field to determine the limits of the field for its corresponding argument.

3.9.5.4 Interference in Tally Argument Lists - When several tally arguments contain one or more identical characters that are active at the same time, they may interfere with each other (i.e., when one of the arguments finds a match, the scanner is stepped past the matching character(s) which prevents those character(s) from being considered for any other match).

The following example illustrates two identical tally arguments that do not interfere with each other since they are not active at the same time. (The first A in FIELD1 causes the first argument to become inactive and the second argument to become active.)

```
MOVE 0 TO T1 T2.
INSPECT FIELD1 TALLYING
        T1 FOR ALL "," BEFORE "A"
        T2 FOR ALL "," AFTER "A".
```

Figure 3-57
Two Tallying Arguments that
Do Not Interfere with Each Other

The two identical tally arguments in the following figure will interfere with each other since both are active at the same time. (For any given position of the scanner, the arguments are applied to FIELD1 in the order in which they appear in the statement. When one of them finds a match, the scanner moves to the next position and ignores the remaining arguments in the argument list.) Each comma in FIELD1 causes T1 to be incremented by 1 and the second argument to be ignored. Thus, T1 will always contain an accurate count of all of the commas in FIELD1, and T2 will always be unchanged.

```
INSPECT FIELD1 TALLYING
        T1 FOR ALL ","
        T2 FOR ALL "," AFTER "A".
```

Figure 3-58
Two Tallying Arguments that
Do Interfere with Each Other

The following statement achieves the same results as the statement in Figure 3-57. The first argument does not become active until the scanner encounters an A. The second argument tallies all commas that precede the A. After the A, the first argument counts all commas and causes the second argument to be ignored. Thus, T1 contains the number of commas that precede the first A and T2 contains the number of commas that follow the first A. This statement works well as written, but could be more confusing to debug than the one in Figure 3-57.

```
INSPECT FIELD1 TALLYING
        T2 FOR ALL "," AFTER "A"
        T1 FOR ALL ",".
```

Figure 3-59
Two Tallying Arguments that,
Because of their Positioning,
Do Not Interfere with Each Other

The preceding three examples show that one INSPECT statement cannot count any character more than once. Thus, when using the same character in more than one argument of an argument list, consider the nature of the interference and choose the order of the arguments very carefully. The solution to the problem may require two or more INSPECT statements. Consider the following problem:

```
INSPECT FIELD1 TALLYING
        T1 FOR ALL "AB"
        T2 FOR ALL "BC".
```

Figure 3-60
An Attempt to Tally the Character B
with Two Arguments

If FIELD1 contains "ABCABC", after the scan T1 will be incremented by a 2 and T2 will be unaltered. The successful matching of the argument includes each B in the field. Each match resets the scanner to the character position to the right of the B, and causes the second argument to never be successfully matched. Reversing the order of the arguments has no effect, the results remain the same. Only separate INSPECT statements can develop the desired counts.

Sometimes the programmer can use the interference characteristics of the INSPECT statement to good advantage. Consider the following sample argument list:

```
MOVE 0 TO T4 T3 T2 T1.
INSPECT FIELD1 TALLYING
        T4 FOR ALL "*****"
        T3 FOR ALL "****"
        T2 FOR ALL "***"
        T1 FOR ALL "*".
```

Figure 3-61
Tallying Asterisk Groupings

The argument list in Figure 3-61 counts all of the asterisks in FIELD1 but in four different tally counters. T4 counts the number of times that four asterisks occur together; T3 counts the number of times three asterisks appear together; T2 counts twos; and T1 counts singles.

If FIELD1 contains a string of more than four consecutive asterisks, the argument list breaks the string into groups of four, and counts them in T4. It then counts the less-than-four remainder in T3, T2, or T1.

Reversing the order of the arguments in this list causes T1 to count all of the asterisks and T2, T3, and T4 to remain unchanged.

When the LEADING condition is used with an argument in the argument list, that argument becomes inactive as soon as it fails to be matched in the field being inspected. Therefore, when two arguments in an argument list contain one or more identical characters and one of the arguments has a LEADING condition, the argument with the LEADING condition should appear first. Consider the following sample statement:

```
MOVE 0 TO T1 T2.
INSPECT FIELD1 TALLYING
        T1 FOR LEADING "*"
        T2 FOR ALL "*".
```

Figure 3-62
Placing the LEADING Condition
in the Argument List

The placement of the LEADING condition in this sample statement causes T1 to count only leading asterisks in FIELD1; the occurrence of any other character stops this counting and causes the first tally argument to become inactive. T2 keeps a count of any remaining asterisks in FIELD1.

Reversing the order of the arguments in this statement results in an argument list that can never increment T1.

```
INSPECT FIELD1 TALLYING
        T2 FOR ALL "*"
        T1 FOR LEADING "*".
```

Figure 3-63
Reversing the Argument
List in Figure 3-62

If the first character in FIELD1 is not an asterisk, neither argument can match it and the second argument becomes inactive. If the first character in FIELD1 is an asterisk, the first argument matches and causes the second argument to be ignored. The first non-asterisk character in FIELD1 will fail to match the first argument and the second argument will become inactive. (The second argument becomes inactive because it has not found a match in all of the preceding characters.)

An argument with both a LEADING condition and a BEFORE phrase can sometimes sucessfully "delimit" the field being inspected:

```
MOVE 0 TO T1 T2.
INSPECT FIELD1 TALLYING
     T1 FOR LEADING SPACES
     T2 FOR ALL "ΔΔΔ" BEFORE "."
     T2 FOR ALL "ΔΔ" BEFORE "."
     T2 FOR ALL "Δ" BEFORE ".".
IF T2 > 0 ADD 1 TO T2.
```

Figure 3-64
An Argument List that Counts
Words in a Statement

The statements in Figure 3-64 count the number of "words" in the English statement in FIELD1. (This assumes that no more than three spaces separate the words in the sentence and that the sentence ends with a period.) When FIELD1 has been scanned, T2 contains the number of gaps between the words. Since a count of the gaps renders a number that is one less than the number of words, the conditional statement adds one to the count.

The first argument removes any leading spaces, counting them in a different tally counter. This shortens FIELD1 by preventing the application of the second through the fourth arguments until the scanner finds a non-space character. The BEFORE phrase on each of the other arguments causes them to become inactive when the scanner reaches the period at the end of the sentence. Thus, the BEFORE phrases "shorten" FIELD1 by making the second through the fourth arguments inactive before the scanner reaches the right-hand end of FIELD1. If the sentence in FIELD1 is indented with tab characters instead of spaces, a second LEADING argument can count the tab characters. The following sample statement illustrates this technique:

```
INSPECT FIELD1 TALLYING
        T1 FOR LEADING SPACES
        T1 FOR LEADING TAB
        T2 FOR ALL "    " etc.
```

Figure 3-65
Counting Leading Tab or Space Characters

When an argument list contains a CHARACTERS argument, it should be the last argument in the list. Since the CHARACTERS argument always matches the field, it prevents the application of any of the following arguments in the list. However, as the last argument in an argument list, it can count the remaining characters in the field being inspected. Consider the following illustration.

```
MOVE 0 TO T1 T2 T3 T4 T5.
INSPECT FIELD1 TALLYING
        T1 FOR LEADING SPACES
        T2 FOR ALL "." BEFORE ","
        T3 FOR ALL "+" BEFORE ","
        T4 FOR ALL "-" BEFORE ","
        T5 FOR CHARACTERS BEFORE ",".
```

Figure 3-66
Counting the Remaining Characters
With the CHARACTERS Argument

If FIELD1 is known to contain a number in the form frequently used to input data, it may contain a plus or minus sign, and a decimal point; further, the number may possibly be preceded by spaces and terminated by a comma. If this statement were compiled and executed, it would deliver the following results:

T1 would contain the number of leading spaces,

T2 would contain the number of periods,

T3 would contain the number of plus signs,

T4 would contain the number of minus signs,

T5 would contain the number of remaining characters (assumed to be numeric), and

the sum of T1 through T5 (plus 1) gives the character position occupied by the terminating comma.

3.9.6 The REPLACING Phrase

When an INSPECT statement contains a REPLACING phrase, that statement selectively replaces characters or groups of characters in the designated field.

The REPLACING phrase names a search argument consisting of a character string of one or more characters and a condition under which the string may be applied to the field being inspected. Associated with the search argument is the replacement value, which must be the same length as the search argument. Each time the search argument finds a match in the field being inspected, under the condition stated, the replacement value replaces the matched characters.

A BEFORE/AFTER phrase may be used to delimit the area of the field being inspected. A search argument applies only to the delimited area of the field.

3.9.6.1 The Search Argument - The search argument of the REPLACING phrase names a character string and a condition under which the character string should be compared to the delimited string being inspected. The following Figure shows the format of the search argument:

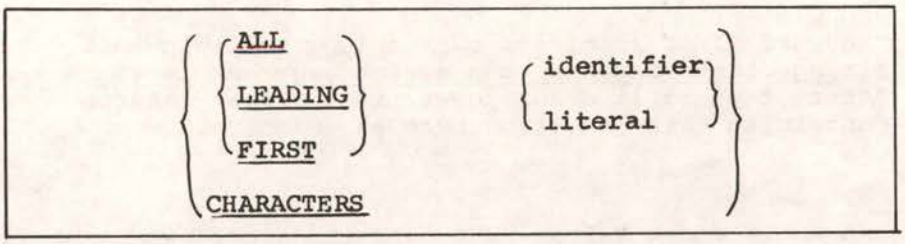


Figure 3-67
Format of the Search Argument

The CHARACTERS form of the search argument specifies that every character in the delimited string being inspected should be considered to match an imaginary character which serves as the search argument. Thus, the replacement value replaces each character in the delimited string. (The replacement value, in this case, must be one character long.)

The ALL, LEADING, and FIRST forms of the search argument specify a particular character string, which may be represented by a literal or an identifier. The search argument character string may be any length. However, each character of the argument must match a character in the delimited string before the software considers the argument matched.

- A literal character string must be either non-numeric or a figurative constant (other than ALL literal). A figurative constant, such as SPACE, ZERO, etc., represents a single character and can be written as " ", "0", etc. with the same effect. Since a figurative constant represents a single character, the replacement value must be one character long.

- An identifier must represent an elementary item of DISPLAY usage. It may be any class. However, if it is other than alphabetic, the software performs an implicit redefinition of the item. (This redefinition is identical to the BEFORE/AFTER delimiter redefinition discussed in Section 3.9.2.)

The words ALL, LEADING, and FIRST supply conditions which further delimit the inspection operation:

- The word ALL specifies that each match that the search argument finds in the delimited string is to be replaced by the replacement value. When a literal follows the word ALL, it does not have the same meaning as the figurative constant, ALL literal. (The figurative constant meaning of ALL "," is a string of consecutive commas, as many as the context of the statement requires.) ALL "," as a search argument of the REPLACING phrase means, "replace each comma without regard to adjacent characters."

- The word LEADING specifies that only adjacent matches of the search argument at the left-hand end of the delimited character string be replaced. At the first failure to match the search argument, the software terminates the replacement operation and causes the argument to become inactive.

- The word FIRST specifies that only the left-most character string that matches the search argument is to be replaced. After the replacement operation, the search argument containing this condition becomes inactive.

3.9.6.2 The Replacement Value - Whenever the search argument finds a match in the field being inspected, the matched characters are replaced by the replacement value. The word BY followed by an identifier or literal specifies the replacement value.

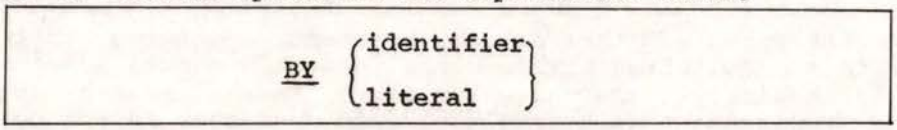


Figure 3-68
Format of the Replacement Value

The replacement value must always be the same size as its associated search argument.

If the replacement value is a literal character string, it must be either a non-numeric literal or a figurative constant (other than ALL literal). A figurative constant represents as many characters as the length that the search argument requires.

If the replacement value is an identifier, it must be an elementary item of DISPLAY usage. It may be any class. However, if it is other than alphanumeric, the software conducts an implicit redefinition of the item. (This redefinition is the same as the BEFORE/AFTER redefinition discussed in Section 3.9.2.)

3.9.6.3 The Replacement Argument - The replacement argument consists of the search argument (with its condition and character string), the replacement value, and an optional BEFORE/AFTER phrase.

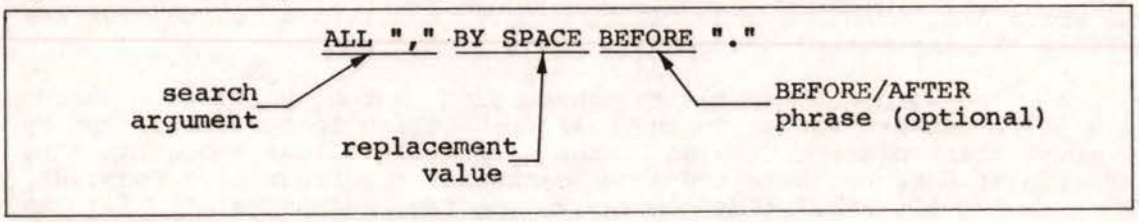


Figure 3-69
The Replacement Argument

3.9.6.4 The Replacement Argument List - One INSPECT...REPLACING statement can contain more than one replacement argument. Several replacement arguments form an argument list, and the manner in which the list is processed affects the action of any given replacement argument.

The following examples show INSPECT statements with replacement argument lists. The text following each one tells how that list will be processed.

```
INSPECT FIELD1 REPLACING
        ALL "," BY SPACE
        ALL "." BY SPACE
        ALL ";" BY SPACE.
```

Figure 3-70
Replacement Argument List that is
Active Over the Entire Field

These three replacement arguments all have the same replacement value, SPACE, and are active over the entire field being inspected.

Thus, this statement replaces all commas, periods, and semicolons with space characters; and leaves all other characters unchanged.

```
INSPECT FIELD1 REPLACING
        ALL "0" BY "1"
        ALL "1" BY "0".
```

Figure 3-71
Replacement Argument List that
"Swaps" Ones for Zeros and Zeros for Ones

Each of these two replacement arguments has its own replacement value, and is active over the entire field being inspected. This statement exchanges zeros for ones and ones for zeros, and leaves all other characters unchanged.

NOTE

When a search argument finds a match in the field being inspected, the software replaces that character string and scans to the next position beyond the replaced characters. It ignores the remaining arguments and applies the first argument in the list to the character string in the new position. Thus, it never inspects the new value that was supplied by the replacement operation. Because of this, the search arguments may have the same values as the replacement arguments with no chance of interference.

```
INSPECT FIELD1 REPLACING
        ALL "0" BY "1" BEFORE SPACE
        ALL "1" BY "0" BEFORE SPACE.
```

Figure 3-72
Replacement Argument List that
Becomes Inactive with the
Occurrence of a Space Character

This sample statement is identical to the statement in Figure 3-71, except that here the first occurrence of a space character in FIELD1 causes both arguments to become inactive.

```
INSPECT FIELD1 REPLACING
        ALL "0" BY "1" BEFORE SPACE
        ALL "1" BY "0" BEFORE SPACE
        CHARACTERS BY "*" BEFORE SPACE.
```

Figure 3-73
Argument List with Three Arguments
That Become Inactive with the
Occurrence of a Space

Just as in the argument list in Figure 3-72, the first space character causes all of these replacement arguments to become inactive. This argument list exchanges zeros for ones, ones for zeros, and asterisks for all other characters that are in the delimited area.

If the BEFORE phrase is removed from the third argument, that argument will remain active across all of FIELD1. Within the area delimited by the first space character, the third argument replaces all characters except ones and zeros with asterisks. Beyond this area, it replaces all characters (including the space that delimited FIELD1 for the first two arguments and any zeros and ones) with asterisks.

3.9.6.5 Interference in Replacement Argument Lists - When several search arguments that are active at the same time contain one or more identical characters, they may interfere with each other, and consequently have an effect on the replacement operation. This interference of one search argument with the matching of other search arguments is similar to the interference that occurs between tally arguments.

The action of a search argument is never affected by the BEFORE/AFTER delimiters of other arguments, since the software scans for delimiter matches before it scans for replacement operations.

The action of a search argument is never affected by the characters of any replacement value, since the scanner does not inspect the replaced characters again during execution of the INSPECT statement.

Interference between search arguments, therefore, depends on the order of the arguments, the values of the arguments, and the active-inactive status of the arguments. (The discussion in Section 3.9.5.4 Interference in Tally Argument Lists, applies, generally, to replacement arguments as well.)

The following rules will help minimize interference in replacement argument lists:

1. Place search arguments with LEADING or FIRST conditions at the start of the list;

2. Place several arguments with the CHARACTERS condition at the end of the list;

3. Consider, very carefully, the order of appearance of any search arguments that contain one or more identical characters.

3.9.7 Common Errors, INSPECT Statement

The most common errors that programmers make when writing INSPECT statements are:

- Leaving the FOR out of an INSPECT...TALLYING statement.
- Using the word "WITH" instead of "BY" in the REPLACING phrase.
- Failing to initialize the tally counter.
- Omitting the word "ALL" viz:

```
INSPECT FIELD1 TALLYING TLY FOR SPACES.
```

CHAPTER 4

NUMERIC CHARACTER HANDLING

4.1 INTRODUCTION

This chapter discusses numeric class data and the COBOL operations that may be performed on numeric class data.

The Introduction to Chapter 3 (Non-numeric Character Handling) covers COBOL data classes. This chapter assumes that the reader understands the concept of COBOL data classes.

4.2 USAGES, DISPLAY/COMP

The USAGE of a numeric class item specifies the form in which that item's data is held in memory. COBOL has two basic formats for data storage, DISPLAY and COMPUTATIONAL (DISPLAY, DISPLAY-6, and DISPLAY-7 are all equivalent):

- ANS-74 COBOL standards prescribe DISPLAY usage to be a string of characters or bytes in decimal radix, with an assumed decimal point location and various sign conventions.

- The same standards prescribe COMPUTATIONAL (COMP) usage to be a real number with the same range of values as a DISPLAY usage number. However, with COMP usage, the compiler implementor has the liberty of specifying the form in which that number is held in memory.

PDP-11 COBOL stores COMP usage fields as binary numbers in one, two, three or four words with an assumed decimal scaling position. Consider the following field description:

```
01 GEMINI PIC 99V99 COMP.
```

If this field contains the value 12.34, PDP-11 COBOL stores it as a 1-word binary number. The word contains the integer value 1234 and another location contains the scaling factor. In this example, the scaling factor records the fact that this integer has two decimal fractional positions associated with it. Thus, the COBOL OTS knows that the stored binary integer is 100 times larger than the programmer intends it to be.

If the compiler encounters the following statement:

```
ADD 1 TO GEMINI.
```

it generates instructions to add a 1 to the 1234 in GEMINI. The OTS, however, scales the literal 1 up by two decimal places and adds the resultant literal, 100, to the number in GEMINI. Thus, after the ADD operation, GEMINI contains the new value 1334 (which is actually 13.34 with the stored decimal scaling position).

Thus, the PDP-11 COBOL compiler and OTS manipulate the data in both DISPLAY and COMP usage items in much the same way. Both USAGES have exactly the same accuracy and precision, and can be freely mixed in a

program. If a DISPLAY usage number and a COMP usage number are both involved in the same arithmetic statement, the OTS converts them to a common radix, with no loss of information. It also converts the result (if necessary), before storing it, with no significant loss.

The only effect of specifying the COMP usage is that it reduces the space required for most numbers and speeds up the execution of arithmetic statements.

4.2.1 Sign Conventions

DISPLAY or COMP usage numeric items may be signed or unsigned. Unsigned numbers may contain values that range from zero to the largest positive value allowed by their declared precision. Negative values are not allowed. All PDP-11 COBOL arithmetic operations yield signed results. When the OTS must store such a result, whether positive or negative, in an unsigned data item, it stores only the absolute value of the result. Thus, unsigned items always contain zero or positive values.

This guide does not recommend unsigned numbers for general use. They are usually a source of programming errors, and are handled less efficiently than signed quantities by the OTS.

Signed quantities always contain a numeric value and an operational sign. The OTS stores the sign with the numeric value in a variety of ways depending on the USAGE of the item and the presence of the SIGN clause.

PDP-11 COBOL always stores signed COMP items in two's complement binary form. Thus, the high-order bit indicates the sign of the item.

PDP-11 COBOL always stores signed DISPLAY items as a sequence of byte positions containing numeric ASCII characters. It may include the sign in the high-order byte, the low order byte, or as a separate, extra, byte on either the high-order or low-order end of the item.

When the OTS stores the sign as part of a byte which also contains a numeric digit, the sign causes a value change in that byte and, hence, changes the value of the numeric digit. The following table shows the actual ASCII character that results when a numeric value and a sign share the same byte.

Table 4-1
The Resulting ASCII Character From a
Sign and Digit Sharing the Same Byte

		DIGIT VALUE									
		0	1	2	3	4	5	6	7	8	9
SIGN	+	{	A	B	C	D	E	F	G	H	I
	-	}	J	K	L	M	N	O	P	Q	R

A byte containing a +0 stores as an octal 173, which prints as either a { or a [depending on the printing device.

A byte containing a -0 stores as an octal 175, which prints as either a } or a] depending on the printing device.

When the OTS stores the sign as a separate distinct character, the actual ASCII character that it stores is the graphic plus sign (octal 053) or the graphic minus sign (octal 055).

4.2.2 Illegal Values in Numeric Fields

All PDP-11 COBOL arithmetic operations store legal values into their result fields. However, it is possible, by reading invalid data or through redefinition and group moves, to store data in numeric fields that do not obey the descriptions of those fields. (For example, it is possible to place signed values into unsigned fields, and to place non-numeric or improperly signed data into signed numeric DISPLAY fields.) PDP-11 COBOL handles this data in the following manner.

NOTE

The following four compiler techniques are not specified by ANS-74 COBOL standards. Dependence on them may yield programs that are not compiler independent.

1. When a quantity described as unsigned enters into an arithmetic operation, the OTS treats it as a signed quantity. If it contains no sign, the OTS either considers the sign to be positive, or ignores the sign if the value of the field is zero. If the field does contain a legally constructed sign, the OTS interprets the sign as if the item had been described as a signed quantity.

   Thus the OTS treats a negative value in an unsigned COMP item as a negative value. Likewise a negative sign, stored as J through R,or } in the right-most digit of an unsigned DISPLAY item, causes the OTS to treat that value as a negative value.

   When an arithmetic operation or legal elementary MOVE statement places its result in an unsigned item, that item receives the absolute value only (no sign information is encoded in the result).

   For example the following coding results in an unsigned value of 15 in field B:

```
      .
      .
      .
02 A PIC S99 VALUE IS 5.
Ø2 C PIC XX VALUE IS "2}".
Ø2 B REDEFINES C PIC 99.
      .
      .
      .
ADD A TO B.
```

   However, given the same original values in A and B, the following statement would result in a value of -15 in field A. (Field B remains at -20.)

```
ADD B TO A.
```

2. When a signed quantity enters into an arithmetic operation, the OTS interprets the sign as follows:

   - If the item is COMP usage, the OTS takes the value as a two's complement number;

   - If the item is DISPLAY usage and the sign is encoded within the leading or trailing digit position, the OTS takes the sign as positive if that position contains either a { (octal 173) or any ASCII byte that collates less than J. Thus, the OTS considers a space, 0 through 9, and A through I, to be positive. Further, it considers any ASCII character that collates equal-to or higher-than J (except { --octal 173) to be negative. (The OTS conducts this sign determination separately from the numeric value determination for the same byte.)

VALUE	SIGN DETERMINATION
ØØØA	+0001
ØØØJ	-0001

   - If the item is DISPLAY usage and the sign is encoded as a separate leading or trailing byte, the OTS takes the sign as negative only if that byte contains the ASCII character - (octal 055). The OTS considers that all other ASCII characters in the separate sign position indicate a positive field.

3. A COMP usage item may receive a value that is larger than the specified range. For example, the OTS stores a field described as PIC S9999 COMP as a 16-bit binary number. The declared range is four decimal digits, but the field has the capability of storing any value from -32,768 to +32,767 (decimal). The OTS stores the results of an arithmetic operation on such a field as a value modulo the declared decimal range. Thus, any value that exceeds 9,999 stores a modulo 10,000 value. (The binary value of 10,000 stores as a zero.)

   When a COMP usage field enters an arithmetic operation, however, the OTS uses the full binary number as the binary value of the field. Thus, a value stored into a COMP field by a group move may cause the field to contain a value that exceeds the declared range. Arithmetic operations will use that value as found.

4. A DISPLAY usage item may contain a value in which some or all of the numeric digit positions contain illegal values.

   When a DISPLAY usage numeric item enters an arithmetic operation, the OTS converts each character to a binary value either before or during the operation. This conversion maps certain ASCII characters into the numeric values 0 through 9, and all other ASCII characters into the numeric value 0. The following table shows these conversion values:

Table 4-2
Conversion Values

ASCII CHARACTERS	BINARY VALUES
A J 1	0001 (1)
B K 2	0010 (2)
C L 3	0011 (3)
D M 4	0100 (4)
E N 5	0101 (5)
F O 6	0110 (6)
G P 7	0111 (7)
H Q 8	1000 (8)
I R 9	1001 (9)
ALL OTHERS	0000 (0)

All arithmetic operations (including the numeric elementary MOVE deliver the ASCII characters 1 through 9 and 0 into all digit positions of a numeric DISPLAY field. A digit position that also contains a sign value receives a correctly coded sign value as shown in Table 4-1.

4.3 TESTING NUMERIC FIELDS

COBOL provides the following three kinds of tests for evaluating numeric fields:

1. Relation tests, that compare the field's contents to another numeric value;

2. Sign tests, that examine the field's sign to see if it is positive or negative; and,

3. Class tests, that inspect the field's digit positions for legal numeric values.

The following sub-sections explain these tests in detail.

4.3.1 Relation Tests

A relation test compares two numeric quantities and determines if the specified relation between them is True. For example, the following statement compares FIELD1 to FIELD2 and determines if the numeric value of FIELD1 is greater than the numeric value of FIELD2. If so, the relation condition is True and program control takes the True path of the statement.

```
IF FIELD1 > FIELD2 ...
```

Either field in a relation test may be a numeric literal or the figurative constant, ZERO. (The numeric literals 0, 00, 0.0, or ZERO are all equivalent, both in meaning and in execution speed.)

The size of the fields in a numeric relation test do not have to be the same (this includes the sizes of numeric literals). The comparison operation aligns both fields on their assumed decimal positions (through actual scaling operations in temporary locations or by accessing the individual digits) and supplies leading or trailing (as required) zeroes to either or both fields.

The comparison operation always compares the signs of non-zero fields and considers positive fields to be greater than negative fields. However, since it does not compare them, positive zeroes and negative zeroes are equal. (A negative zero could arrive in a field through redefinition of the field or a MOVE to a group item.) Further, the operation considers unsigned numeric fields to be positive.

The form of representation of the number (COMP or DISPLAY usage) and the various methods of storing DISPLAY usage signs have no effect on numeric relation tests.

For comparison purposes, the operation converts any illegal characters stored in DISPLAY usage fields to zeroes. It does not, however, alter the actual values in those fields.

4.3.2 Sign Tests

The sign test compares a numeric quantity to zero and determines if it is greater (positive), less (negative), or equal (zero). Both the relation test and the sign test can perform this function. For example, consider the following relation test:

```
IF FIELD1 > 0 ...
```

Now consider the following sign test:

```
IF FIELD1 POSITIVE ...
```

Both of these tests accomplish the same thing and would always arrive at the same result. The sign test, however, shortens the statement and shows, at a glance, that it is testing the sign.

Table 4-3 shows the sign tests and their equivalent relation tests as applied to FIELD1.

Table 4-3
The Sign Tests

SIGN TEST	EQUIVALUENT RELATION TEST
IF FIELD1 POSITIVE ...	IF FIELD1 > 0
IF FIELD1 NOT POSITIVE ...	IF FIELD1 NOT > 0
IF FIELD1 NEGATIVE ...	IF FIELD1 < 0 ...
IF FIELD1 NOT NEGATIVE ...	IF FIELD1 NOT < 0
IF FIELD1 ZERO ...	IF FIELD1 = 0 ...
IF FIELD1 NOT ZERO ...	IF FIELD1 NOT = 0 ...

Sign tests have no execution speed advantage over relation tests. The compiler actually substitutes the equivalent relation test for every correctly written sign test. (Sections 4.2.1 and 4.2.2 discuss the acceptable sign values and the treatment of illegal sign values.)

4.3.3 Class Tests

The class test interrogates a numeric field to determine if it contains numeric or alphabetic data, and uses the result to alter the flow of control in a program. For example, the following statement determines if FIELD1 contains numeric data. If so, the test condition is True and program control takes the True path of the statement.

```
IF FIELD1 IS NUMERIC ...
```

When reading in newly prepared data, it is often desirable to check certain fields for valid values. Relation tests and sign tests can only determine if the field's contents are within a certain range, and these tests both treat illegal characters in DISPLAY usage items as zeroes. Thus, some data preparation errors could pass both of these tests.

The NUMERIC class test checks numeric (or alphanumeric) DISPLAY usage fields for valid numeric digits.

If the field being tested contains a sign (whether carried as an overpunch or as a separate character), the test checks it for a valid sign value. If the character position carrying the sign contains an illegal sign value, the NUMERIC class test fails the item and program control takes the False path of the IF statement. If the character position contains a valid sign and all digit positions in the field contain valid numeric digits, the NUMERIC class test passes the item and program control takes the True path of the IF statement.

The ALPHABETIC class test checks alphabetic (or alphanumeric) fields for valid alphabetic characters and the space character. If all of the character positions of the field contain ASCII characters (A-Z or space), the item passes the ALPHABETIC class test and causes program control to take the True path of the IF statement. (For further information concerning the ALPHABETIC class test, see Chapter 3, Section 3.3.2.)

4.4 THE MOVE STATEMENT

The MOVE statement moves the contents of one field into another. The following sample MOVE statement moves the contents of FIELD1 into FIELD2.

```
MOVE FIELD1 TO FIELD2.
```

Section 3.5 discusses the basic MOVE statement. This section considers MOVE statements as applied to numeric fields. These MOVE statements can be grouped into the following three categories:

1. Group moves,

2. Elementary moves with numeric receiving fields, and

3. Elementary moves with numeric edited receiving fields.

The following three sub-sections (4.4.1, 4.4.2, and 4.4.3) discuss each of these categories separately.

4.4.1 Group Moves

The software considers a move to be a group move if either the sending field or the receiving field is a group item. It treats both fields in a group move as alphanumeric class fields and performs the move as an alphanumeric to alphanumeric elementary move.

If either field in a group move is a numeric elementary item, the OTS treats the storage area occupied by that item as a field of alphanumeric bytes; thus, it ignores the USAGE, sign, and decimal point location characteristics of the numeric item.

Only the item's allocated size, in bytes, affects the move operation. The OTS considers a separate sign character to be part of the item and moves it with the numeric digit positions.

4.4.2 Elementary Numeric Moves

If both fields of a MOVE statement are elementary items and the receiving field is numeric, the OTS considers the move to be an elementary numeric move. (The sending field may be either numeric or alphanumeric.) The numeric receiving field may be either DISPLAY or COMP usage. The elementary numeric move converts the data format of the sending field to the data format of the receiving field.

An alphanumeric sending field may be either an elementary data item or any alphanumeric literal other than the figurative constants SPACE, QUOTE, LOW-VALUE, HIGH-VALUE, or ALL "literal". The elementary numeric move accepts the figurative constant ZERO and considers it to be equivalent to the numeric literal 0. It treats alphanumeric sending fields as unsigned integers of DISPLAY usage.

If necessary, the numeric move operation converts the sending field to the data format of the receiving field and aligns the sending field's decimal point on that of the receiving field. It then moves the sending field digits to their corresponding receiving field digits.

If the sending field has more digit positions than the receiving field, the decimal point alignment operation truncates the sending field, with the resultant loss of digits. The end truncated (high-order or low-order) depends upon the number of sending field digit positions that find matches on each side of the receiving field's decimal point. If the receiving field has fewer digit positions on both sides of the decimal point, the operation truncates both ends of the sending field. Thus, if a field described as PIC 999V999 is moved to a field described as PIC 99V99 , it loses one digit from the left end and one from the right end. The following Figure illustrates this alignment operation (the caret (^) indicates the stored decimal scaling position):

```
01 GANDALF PIC 99V99.

   ...

   MOVE 123.321 TO GANDALF.
```

Before execution	00^00
After execution	23^32

Figure 4-1
Truncation Caused By Decimal Point Alignment

If the sending field has fewer digit positions than the receiving field, the move operation supplies zeroes for all unfilled digit positions. Figure 4-2 illustrates this alignment (the caret (^) indicates the stored decimal scaling position):

```
01 RIVENDELL PIC 999V99.

   ...

   MOVE 1 TO RIVENDELL.
```

Before execution	000^00
After execution	001^00

Figure 4-2
Zero Filling Caused By Decimal Point Alignment

The following statement produces the same results:

```
MOVE 001.00 TO RIVENDELL.
```

Consider the following two MOVE statements and their resultant truncating and zero filling effects:

STATEMENT	RIVENDELL AFTER EXECUTION
MOVE 00100 TO RIVENDELL	100^00
MOVE "00100" TO RIVENDELL	100^00

Literals with leading or trailing zeroes have no significant advantage in space or execution speed with PDP-11 COBOL, and the zeros are often lost by decimal point alignment.

The MOVE statement's receiving field dictates how the sign will be moved. A signed DISPLAY usage receiving field causes the sign to be moved as a separate quantity. An unsigned DISPLAY usage receiving field causes no sign movement. A COMP usage receiving field, whether signed or unsigned, causes the sign to be moved; however, if the receiving field is unsigned, the OTS sets its value to absolute.

4.4.3 Elementary Numeric Edited Moves

The PDP-11 COBOL object time system considers an elementary numeric move to a receiving field of the numeric edited category to be an elementary numeric edited move. The sending field of an elementary numeric edited move may be either numeric or alphanumeric and, if numeric, it can be either DISPLAY usage or COMP usage. The OTS treats alphanumeric sending fields in numeric edited moves as unsigned DISPLAY usage integers.

The OTS considers the receiving field to be numeric edited category if it contains a combination of the following symbols:

B -- Space insertion position;

P -- Decimal scaling position;

V -- Location of assumed decimal point;

Z -- Leading numeric character position to be replaced by a space if the position contains a zero:

0 -- Zero insertion position;

9 -- Position contains a numeric character;

/ -- Slash insertion position;

, -- Comma insertion position;

. -- Decimal point insertion position;

* -- Leading numeric character position to be replaced by an asterisk if the position contains a zero;

+ -- Positive editing sign control symbol;

- -- Negative editing sign control symbol;

CR -- Credit editing sign control symbol;

DB -- Debit editing sign control symbol;

cs -- Currency symbol ($) insertion position.

A numeric edited field may contain 9, V, and P, but combinations of those symbols without an editing character do not make the field numeric edited.

The numeric edited move operation first converts the sending field to DISPLAY usage and aligns both fields on their decimal point locations, truncating or padding (with zeroes) the sending field until it contains the same number of digit positions on both sides of the decimal point as the receiving field. It then moves the resulting digit values to the receiving field digit positions following the COBOL editing rules.

The COBOL editing rules allow the numeric edited move operation to perform any of the following editing functions:

Suppress leading zeroes with either spaces or asterisks;

- Float a currency sign and a plus or minus sign through suppressed zeroes, inserting the sign at either end of the field;
- Insert zeroes and spaces;
- Insert commas and a decimal point.

Figure 4-3 illustrates several of these functions with the statement, MOVE FRODO TO RIVENDELL. (Assume that FRODO is described as S9999V99.)

FRODO	RIVENDELL	
	PICTURE STRING	CONTENTS AFTER MOVE
0023^00	ZZZZ.99	Δ.Δ23.00
0085^90	++++.99	Δ -85.96
1234^00	Z,ZZZ.99	1,234.00
0012^34	$,$$$.99	Δ $12.34
0000^34	$,$$9.99	Δ Δ$0.34
1234^00	$$,$$$.99	$1,234.00
0012^34	$$9,999.99	$0,012.34
0012^34	$$$$,$$$.99	ΔΔΔΔ $12.34
0000^00	$$$,$$$.$$	ΔΔΔΔΔΔΔΔΔΔ
0012^3M	++++.99	-12.34
001234	$***,***.99	$**1,234.00
0012^34	$***,***.99	$*****12.34

Figure 4-3
Numeric Editing

The currency symbol ($) and the editing sign control symbols (+ -) are the only floating symbols. To float them, enter a string of two or more occurrences of the symbol.

4.4.4 Common Errors, Numeric MOVE Statements

The most common errors made when writing numeric MOVE statements are:

- Placing an incorrect number of replacement characters in a numeric edited item.
- Moving non-numeric data into numeric fields with group moves.
- Trying to float the $ or + insertion characters past the decimal point to force zero values to appear as .00 instead of spaces. (Use $$.99 or ++.99.)
- Forgetting that the $ or + insertion characters require an additional position on the left-most end that cannot be replaced by a digit (unlike the * insertion character which can be completely replaced).

4.5 THE ARITHMETIC STATEMENTS

The COBOL arithmetic statements, ADD, SUBTRACT, MULTIPLY, and DIVIDE, allow COBOL programs to perform simple arithmetic operations on numeric data.

This section covers the use of the COBOL arithmetic statements. The first five sub-sections (4.5.1 through 4.5.5) discuss the common features of the statements and the last four (4.5.6 through 4.5.9) discuss the individual arithmetic statements themselves.

4.5.1 Intermediate Results

Most forms of the arithmetic statements perform their operations in temporary work locations, then move the results to the receiving fields, aligning the decimal points and truncating or zero filling the resultant values.

This temporary work field, called the intermediate result field, has a maximum size of 18 numeric digits. The actual size of the intermediate result field varies for each statement, and is determined at compile time based on the sizes of the operands used by the statement.

When the compiler determines that the size of the intermediate result field exceeds 18 digits, it truncates the excess high order digits. Thus, a program that requests a multiplication operation between the following two fields,

PIC 9(18) and PIC V99.

(which would otherwise cause the compiler to set up a 20-digit intermediate result field -- 9(18)V99) actually causes the following intermediate result field

PIC 9(16)V99.

PDP-11 COBOL truncates high-order digits or low order digits to the right of the decimal point, based on the assumption that most large data declarations are larger than ever need be, so zeroes occupy most of their high-order digit positions. Numeric data may be declared as PIC 9(12) or PIC 9(15) but the values that are placed in these fields will probably not exceed nine digits of range (1-billion) in most applications.

When using large numbers (or numbers with many decimal places) that are close to 18 digits long, examine all of the arithmetic operations that manipulate those numbers to determine if truncation will occur.

If truncation is a possibility, reduce the size of the number by dividing it by a power of 10 prior to the arithmetic operation. (This scaling down operation causes the low-order end to lose digits, but these are probably less critical.) Then, after the arithmetic operation, multiply the result by the same power of 10.

To save the low-order digits in such an operation, move the field to a temporary location before the scaling DIVIDE, perform separate, identical arithmetic operations on both the original and the temporary fields, then, after the scaling MULTIPLY, combine their results.

4.5.2 The ROUNDED Phrase

Rounding-off is an important tool with most arithmetic operations. The ROUNDED phrase causes the OTS to round-off the results of COBOL arithmetic operations.

The phrase may be used on any COBOL arithmetic statement. Rounding-off takes place only when the ROUNDED phrase requests it, and then only if the intermediate result has more low-order digits than the result field.

PDP-11 COBOL rounds-off by adding a 5 to the leftmost truncated digit of the absolute value of the intermediate result before it stores that result.

Consider the following illustration and assume an intermediate result of 54321.2468:

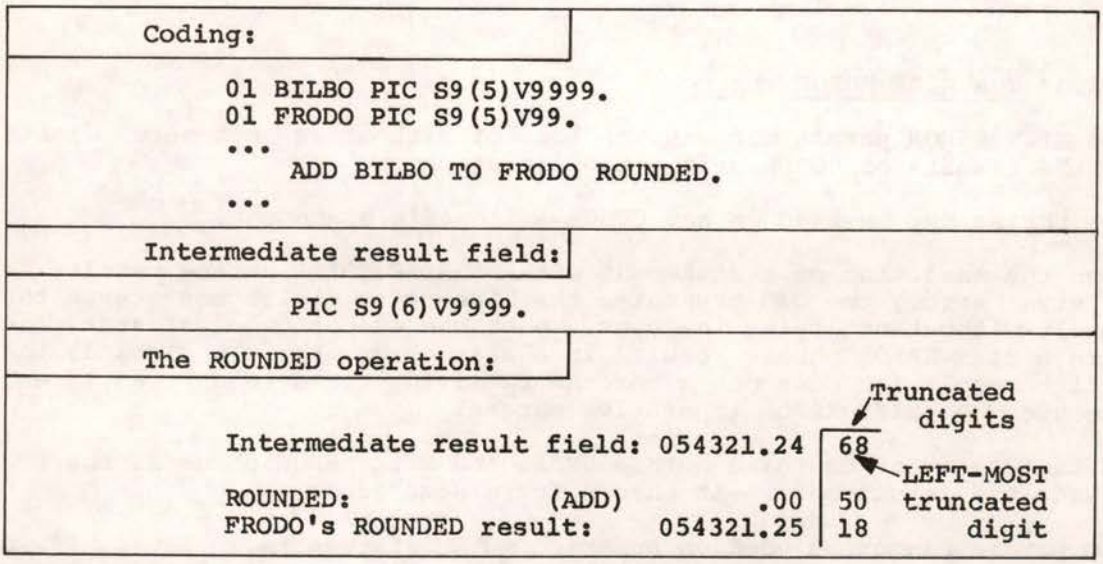


Figure 4-4
Rounding Truncated Decimal Point Positions

The following ROUNDING example rounds-off to the decimal scaling position (P). Assume an intermediate result of 24689. (Section 4.5.4 discusses the GIVING phrase in numeric operations.)

Coding:

```
01  GANDALF PIC 9999.
01  SARUMAN PIC 9999PP.

    ...
    MULTIPLY GANDALF BY 10
        GIVING SARUMAN ROUNDED.
    ...
```

Intermediate result field:

PIC 999999.

The ROUNDED operation:

			Truncated
Intermediate result field:	0246	89.	digits
ROUNDED (ADD):		50.	
SARUMAN ROUNDED result:	0247	39.	

Figure 4-5
Rounding Truncated Decimal Scaling Positions

4.5.3 The SIZE ERROR Phrase

The SIZE ERROR phrase detects the loss of high-order non-zero digits in the results of COBOL arithmetic operations.

The phrase may be used on any COBOL arithmetic statement.

When the execution of a statement with no SIZE ERROR phrase results in a size error, the OTS truncates the high-order digits and stores the result without notifying the user. When the execution of a statement with a SIZE ERROR phrase results in a size error, the OTS discards the entire result (it does not alter the receiving field in any way) and executes the SIZE-ERROR imperative phrase.

If the statement contains both ROUNDED and SIZE ERROR phrases, the OTS rounds the result before it checks for a size error.

The phrase cannot be used on numeric MOVE statements. Thus, if a program moves a numeric quantity to a smaller numeric field, it may inadvertently lose high-order digits. For example, consider the following MOVE of a field to a smaller field:

```
01 BIGFOOT PIC 9(8)V99.

01 LITTLEFOOT PIC 9(4)V99.

   ...

MOVE BIGFOOT TO LITTLEFOOT.
```

This MOVE operation always loses four of BIGFOOT's high-order digits. Either of the following two statements could determine whether these

digits are zero or non-zero, and could be tailored to any size field:

1. IF BIGFOOT NOT > 9999.99
   MOVE BIGFOOT TO LITTLEFOOT
   ELSE ...

2. ADD ZERO TO BIGFOOT GIVING LITTLEFOOT
   ON SIZE ERROR ...

Both of these alternatives allow the MOVE operation to occur only if BIGFOOT loses no significant digits. If the value in BIGFOOT is too large, both alternatives avoid altering LITTLEFOOT and take the alternative execution path.

4.5.4 The GIVING Phrase

The GIVING phrase moves the intermediate result field of an arithmetic operation to a receiving field. (The phrase acts exactly like a MOVE statement with the intermediate result serving as a sending field and the data item following the word GIVING (in the statement) serving as a receiving field.)

The phrase may be used on any COBOL arithmetic statement.

If the data item following the word GIVING is a numeric edited field, the OTS performs the editing the same way it does for MOVE statements.

4.5.5 Multiple Operands in ADD and SUBTRACT Statements

Both the ADD and SUBTRACT statements may contain a string of more than one operand preceding the word TO, FROM, or GIVING.

Multiple operands in either of these statements cause the OTS to add the string of operands together and use the intermediate result of that operation as a single operand to be added to or subtracted from, the receiving field.

The following three equivalent coding groups illustrate how the software executes the multiple operand statements:

1. Statement: ADD A B C D TO E.

   Equivalent coding:
   ```
   ADD A B, GIVING TEMP1.
   ADD TEMP1, C, GIVING TEMP2.
   ADD TEMP2, D, GIVING TEMP3.
   ADD TEMP3, E, GIVING E.
   ```

2. Statement: SUBTRACT A, B, C, FROM D.

   Equivalent coding:
   ```
   ADD A, B, GIVING TEMP1.
   ADD TEMP1, C GIVING TEMP2.
   SUBTRACT TEMP1 FROM D GIVING D.
   ```

3. Statement: ADD A B C D GIVING E.

Equivalent coding:
```
ADD A B GIVING TEMP1.
ADD TEMP1 C GIVING TEMP2.
ADD TEMP2 D GIVING E.
```

(Just as with all COBOL statements, the commas in these statements are optional.)

Only statement 3 may have a numeric edited receiving field, since it is the only statement containing a GIVING phrase.

4.5.6 The ADD Statement

The ADD statement adds two or more operands together and stores the result.

The statement may contain multiple operands and the ROUNDED and SIZE ERROR phrases. It may be written in either of the following two formats:

Format 1. ADD FIELD1 ... TO FIELD2.

Format 2. ADD FIELD1 FIELD2 ... GIVING FIELD3.

In Format 1, the receiving field (FIELD2) is one of the addends. It must not be numeric edited category.

In Format 2, the receiving field (FIELD3) is not one of the addends. It may either be numeric or numeric edited. When using this format, omit the word TO.

4.5.7 The SUBTRACT Statement

The SUBTRACT statement subtracts one, or the sum of two or more, operands from another operand and stores the result.

The statement may contain multiple operands and the ROUNDED and SIZE ERROR phrases. It may be written in either of the following two formats:

Format 1. SUBTRACT FIELD1 ... FROM FIELD2.

Format 2. SUBTRACT FIELD1 ... FROM FIELD2 GIVING FIELD3.

In Format 1, the receiving field (FIELD2) is both the subtrahend and the difference (the result). It must not be numeric edited category.

In Format 2, the receiving field (FIELD3) is used only to store the result. It may be either numeric or numeric edited.

4.5.8 The MULTIPLY Statement

The MULTIPLY statement multiplies one operand by another and stores the result.

The statement may contain the ROUNDED and SIZE ERROR phrases. It must not contain multiple operands. It may be written in either of the following two formats:

Format 1. MULTIPLY FIELD1 BY FIELD2.

Format 2. MULTIPLY FIELD1 BY FIELD2 GIVING FIELD3.

In Format 1, the receiving field (FIELD2) is also the multiplier. It must not be numeric edited category.

In Format 2, the receiving field (FIELD3) is neither multiplier nor multiplicand. It may be either numeric or numeric edited.

COBOL's "near English" format could cause a problem with the MULTIPLY statement, since it is common to speak of multiplying a number (multiplicand) by another number (multiplier) and to think of the result as a new value for the multiplicand; thus:

```
MULTIPLY EARNINGS BY 0.24.
                     ↖Multiplier
         ↖Multiplicand
```

This statement is incorrect since the OTS stores the result in the multiplier field, and this multiplier is a literal. The compiler could diagnose this error, but would not diagnose it if the multiplier were a data item. Consider this multiplier written as a data item:

```
MULTIPLY EARNINGS BY TAX-RATE.
```

The compiler would not diagnose this statement's error, and would store the result of the operation in TAX-RATE. A good practice when using MULTIPLY statements is to always write them in Format 2. This ensures that the result is properly stored. The following two statements safely capture their results:

```
MULTIPLY EARNINGS BY 0.24 GIVING EARNINGS.
```

or

```
MULTIPLY EARNINGS BY TAX-RATE GIVING TAXES.
```

4.5.9 The DIVIDE Statement

The DIVIDE statement divides one operand into another and stores the result.

The statement may contain the ROUNDED and SIZE ERROR phrases. It must not contain multiple operands. It may be written in any of the following three formats:

Format 1. DIVIDE FIELD1 INTO FIELD2.

Format 2. DIVIDE FIELD1 INTO FIELD2 GIVING FIELD3.

Format 3: DIVIDE FIELD2 BY FIELD1 GIVING FIELD3.

In Format 1, the receiving field (FIELD2) is also the dividend. It must not be numeric edited category.

In Formats 2 and 3, the receiving field (FIELD3) is neither dividend nor divisor. It may be either numeric or numeric edited.

4.5.10 Common Errors, Arithmetic Statements

The most common errors made when using arithmetic statements are:

- Using an alphanumeric class field in an arithmetic statement. The MOVE statement allows data movement between alphanumeric class fields and certain numeric class fields, but arithmetic statements require that all fields be numeric.

- Writing the ADD or SUBTRACT statements without the GIVING phrase, but attempting to put the result into a numeric edited field.

- Writing a Format 2 ADD statement with the word TO; thus

```
ADD A TO B GIVING C.
```

- Subtracting a 1 from a numeric counter that was described as an unsigned quantity, and testing for a value of less than zero.

- Forgetting that the MULTIPLY statement, without the GIVING phrase, stores the result back into the second operand (multiplier).

- Performing a series of calculations in such a way as to generate an intermediate result that is larger than 18 digits when the final result will be fewer digits. (The programmer should be careful to intersperse divisions with multiplications or to drop non-significant digits that result from multiplying large numbers (or numbers with many decimal places).

- Performing an operation on a Field that contains a value greater than the precision of the result. This can happen only if the field was messed up by a group move or redefinition.

CHAPTER 5

TABLE HANDLING

5.1 INTRODUCTION

With COBOL, as with any other language, any data item that the program refers to must be uniquely identified. This unique identification of data items is usually accomplished by assigning a unique name to each item. However, in many applications this is tedious and inconvenient; often programs require too many names for items that have different names but contain the same type of information. Tables provide a simple solution to this problem. PDP-11 COBOL includes level 1 table handling capabilities as outlined for standard COBOL in the 1974 ANS Standards.

A table is a repetition of one item (element) in memory. This repetition is accomplished by the use of the OCCURS phrase in the data description clause. The literal value in the OCCURS phrase causes the software to duplicate the data description entry as many times as indicated by that value, thus creating a matrix or table.

The elements may be initialized with the VALUE clause or with a procedural instruction. They may contain synchronized or unsynchronized data. They may be accessed only with subscripted procedural instructions. A subscript is a parenthesized integer or data name (with an integer value). The integer value represents the desired occurrence of the element.

This chapter discusses how to set up tables and access them accurately and efficiently. It attempts to cover any problems that may be encountered while handling tables. Read it through carefully before setting up tables with PDP-11 COBOL. Section 6 of the COBOL Reference Manual for the PDP-11 contains reference information on the individual table handling instructions (OCCURS, USAGE IS INDEX, and SET).

5.2 DEFINING TABLES

To define a table with PDP-11 COBOL, simply complete a standard data description for one element of the table and follow it with an OCCURS phrase. The OCCURS phrase contains an integer which dictates the number of times that element will be repeated in memory, thus creating a table.

```
01 A-TABLE
   03 A-GROUP PIC X(5) OCCURS 10 TIMES.
```

Figure 5-1
Defining a Table

The software will automatically map the elements into memory. When mapping a table into memory, the software follows the rules for mapping which depend on whether the element contains synchronized items or not. If they do not contain synchronized items, the software maps them into adjacent memory locations and the size of the table can be easily calculated by multiplying the size of the element times the number of occurrences (5X10 for the table illustrated in Figure 5-1, or 50 bytes of memory).

5.3 MAPPING TABLE ELEMENTS

As mentioned above, when the software detects an OCCURS phrase in an unsynchronized item, it maps the table elements into adjacent locations in memory. Consider the following data description of a simple table and the way it is mapped into memory:

Table Description:
```
01 A-TABLE
   03 A-GROUP PIC X(5) OCCURS 10 TIMES.
```

Memory Map:

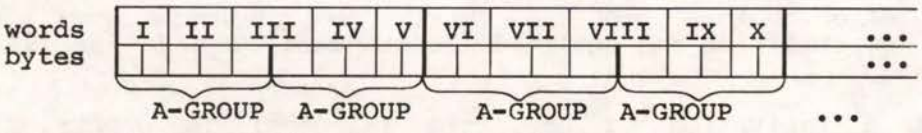


Figure 5-2
Mapping a Table into Memory

The data description in Figure 5-2 causes the software to set up ten items of five bytes each (elements) and place them in adjacent ascending memory locations for a total of 50 character positions, thus creating a table. Since the length of each A-GROUP element is odd (5), the memory addresses of each subsequent element will alternate between odd and even locations.

The SYNCHRONIZED clause causes the software to add a fill byte to group items that contain an odd number of bytes, thereby making the number of bytes in that group item even. This ensures that each subsequent occurrence of the element will not alternate between odd and even addresses, but will map the same (odd or even) as the first repetition of that element.

If the data description of A-GROUP contained a SYNCHRONIZED clause, the software would map it quite differently. If A-GROUP were synchronized, it would expand its length to three words. The item will, by default, be synchronized to the left occupying the first five characters of the three words. The software supplies a padding character to fill out the third word. This padding character is not a part of the A-GROUP element and table instructions referring to A-TABLE will not detect the presence or absence of the character.

The padding character does, however, affect the overall length of the group item and, hence, the table. Without the SYNCHRONIZED clause, A-TABLE required only 50 character positions; now, with the clause,

it requires 60 character positions. (This length includes the last padding character -- following the tenth element in the table.)

Although the SYNCHRONIZED clause has little value when used with alphanumeric fields, an understanding of the concept is essential before attempting to use COMP and INDEX data items in tables. The software automatically synchronizes all COMP and INDEX usage data items, and will most probably alter the size of any table (often drastically) that contains these data types. Consider the following illustration of a synchronized data item being mapped by the software:

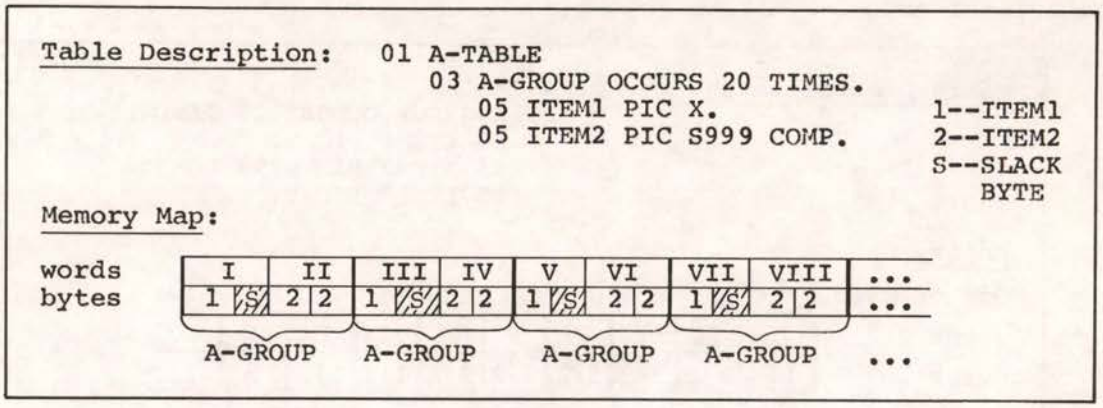


Figure 5-3
Synchronized COMP Item in a Table

Since the software synchronizes the ITEM2 fields (COMP), these fields each occupy a single word in memory; thus, a slack byte follows each occurrence of ITEM1. Each repetition of A-GROUP consumes four bytes of memory -- one byte for ITEM1, one byte for the slack byte, and two bytes for ITEM2. A-TABLE, then, requires 80 bytes of memory (20 elements of four bytes each).

Now, consider the effect of adding a 1-byte field to A-TABLE. If we place the field between ITEM1 and ITEM2, it will take the space formerly occupied by the slack byte. This has the effect of adding a data byte but leaving the size of the table unchanged. Consider the following illustration:

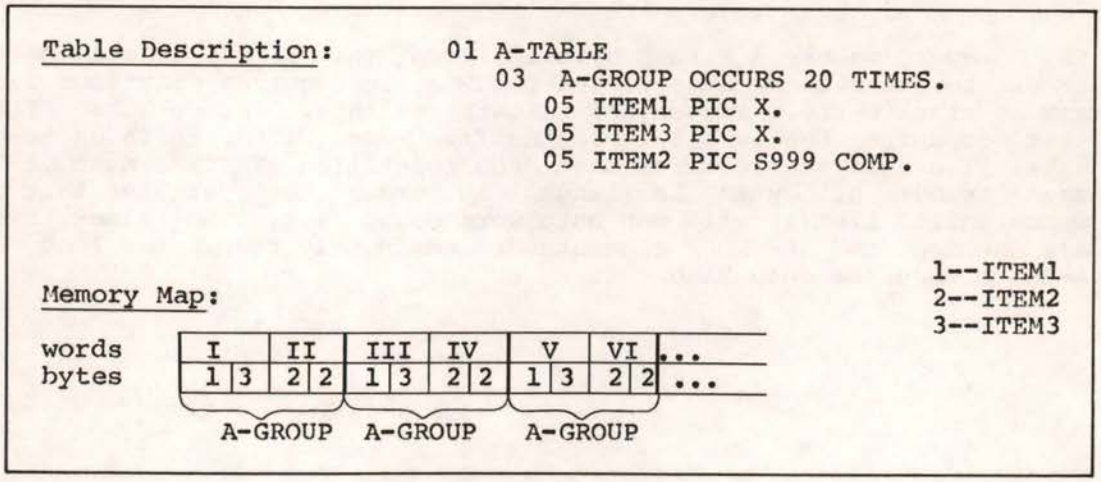


Figure 5-4
Adding a Field without Altering the Table Size

If, however, we place the 1-byte field after ITEM2, it has the effect of adding its own length plus another slack byte. Now, each element requires six full bytes and the complete table consumes 120 bytes of memory (6X20)! This is due to the fact that the first repetition of ITEM1 falls on an even byte and, in order to keep the mapping of each A-GROUP element the same, the software allocates each successive repetition of ITEM1 to an even byte address. Thus, it assigns ITEM3 to the even byte of the third word and adds a slack byte to guarantee that the next element begins on an even byte. Consider the following illustration:

Table Description:

```
01 A-TABLE
   03 A-GROUP OCCURS 20 TIMES.
      05 ITEM1 PIC X.
      05 ITEM2 PIC S999 COMP.
      05 ITEM3 PIC X.
```

Memory Map:



Figure 5-5
Adding one Byte which Adds Two Bytes
to the Element Length

NOTE

The illustrations in this section show each byte with an even address (E) as the leftmost byte and each byte with an odd address (O) as the rightmost byte. (The two bytes, odd and even, are reversed in actual memory.)

If, however, we use a FILLER byte to force the first allocation of ITEM1 to occur on an odd byte, A-GROUP again requires only four bytes and no slack bytes. Figure 5-6 illustrates this. Since the FILLER item occupies the even byte of the first word, ITEM1 falls on an odd byte. The software requires that each repetition of ITEM1 must be an even number of bytes in length in order to guarantee that the synchronized item(s) will map onto word boundaries. No slack bytes are needed and A-GROUP elements are again only four bytes long, and A-TABLE requires only 81 bytes.

Table Description:

```
01 A-TABLE.
   03 FILLER PIC X.
   03 A-GROUP OCCURS 20 TIMES.
      05 ITEM1 PIC X.
      05 ITEM2 PIC S999 COMP.
      05 ITEM3 PIC X.
```

Memory Map:

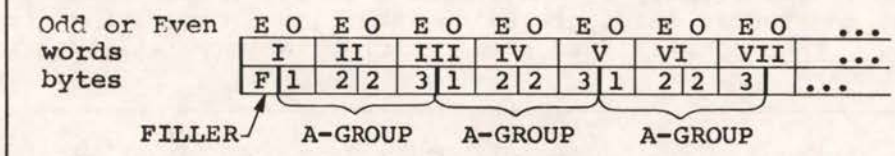


Figure 5-6
Forcing an Odd Address By Adding a 1-Byte FILLER Item to the Head of the Table

If we try to force ITEM1 onto an odd byte with a SYNCHRONIZED RIGHT clause, the software maps ITEM1 into the odd byte, but prohibits all repetitions of the element from using the even byte. Thus, the first repetition of A-GROUP has a slack byte at its beginning and, so that the next element can begin (with a slack byte) at an even address, another slack byte (odd) following ITEM3. This expands the element length to six bytes and the table length to 120 bytes.

Table Description:

```
01 A-TABLE.
   03 A-GROUP OCCURS 20 TIMES.
      05 ITEM1 PIC X SYNCHRONIZED RIGHT.
      05 ITEM2 PIC S999 COMP.
      05 ITEM3 PIC X.
```

Memory Map:

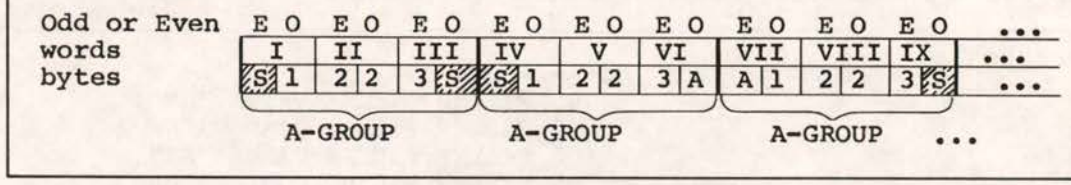


Figure 5-7
The Effect of a SYNCHRONIZED RIGHT Clause Instead of a FILLER Item as shown in Figure 5-6

To determine how the software will map a given table, apply the following two rules:

1. The software maps all items in the first repetition of a table element into memory words as with any item properly defined with a data description, obeying any implicit or explicit synchronization requirements.

2. If the first repetition contains any elementary items with implicit or explicit synchronization, the software maps each successive repetition of the element into memory words in the same way as the first repetition. It does this by adding one

slack byte, if necessary, to make the size of the element even.

5.3.1 Initializing Tables

If a table contains only DISPLAY items, it can be set to any desired initial value (initialized). To initialize a table, simply specify a VALUE phrase on the record level preceding the item containing the OCCURS clause. The sample data definitions, below, will set up initialized tables:

Table Description:

```
01 A-TABLE VALUE IS "JANFEBMARAPRMAY
                    JUNJULAUGSEPOCTNOVDEC".
   03 MONTH-GROUP PIC XXX USAGE DISPLAY
                 OCCURS 12 TIMES.
```

Memory Map:

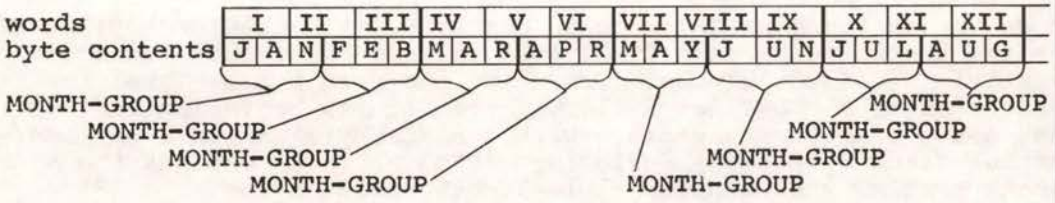


Figure 5-8
Initializing Tables

Often a table is too long to initialize with a single literal, or it contains items that cannot be initialized (numeric, alphanumeric, or COMP). These items can be individually initialized by redefining the group level preceding the level that contains the OCCURS clause. Consider the following sample table descriptions:

Table Description:

```
01 A-RECORD-ALT.
   05 FILLER PIC XX VALUE "AX".
   05 FILLER PIC 99 COMP VALUE 1.
   05 FILLER PIC XX VALUE "BX".
   05 FILLER PIC 99 COMP VALUE 2.

01 A-RECORD REDEFINES A-RECORD-ALT.
   03 A-GROUP OCCURS 26 TIMES.
      05 ITEM1 PIC X.
      05 ITEM2 PIC S99 COMP.
```

Memory Map:

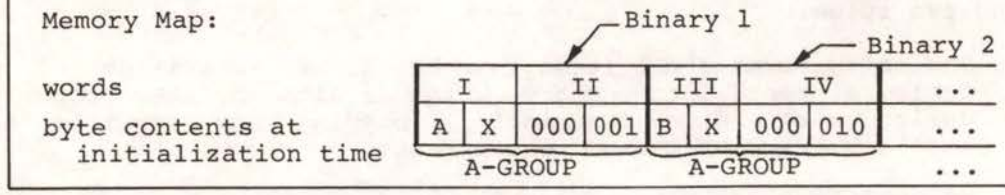


Figure 5-9
Initializing Mixed Usage Fields

In the preceding example, the slack bytes in the alphanumeric fields (ITEM1) are being initialized to X.

Table Description:

```
01 A-RECORD-ALT.
   03 FILLER PIC X(30) VALUE IS
      "AAAAAAAAAABBBBBBBBBBCCCCCCCCCC".
   03 FILLER PIC X(30) VALUE IS
      "DDDDDDDDDDEEEEEEEEEEFFFFFFFFFF".
(etc.)

01 A-RECORD REDEFINES A-RECORD-ALT.
   03 ITEM1 PIX X(10) OCCURS 26 TIMES.
```

Memory Map:

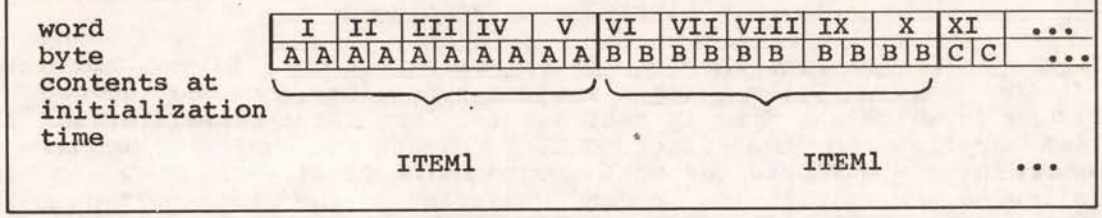


Figure 5-10
Initializing Alphanumeric Fields

In the preceding example, each FILLER item initializes three 10-byte table elements.

When redefining or initializing table elements, allow space for any slack bytes that may be added due to synchronization (implicit or explicit). The slack bytes do not have to be initialized; however, they may be and, if initialized to an uncommon value, they may even serve as a debugging aid for situations such as a statement referring to the record level above the OCCURS clause or another record redefining that level.

Sometimes the length and format of table items are such that they would best be initialized by statements in the Procedure Division. This initialization coding could be executed once and then overlaid (due to the automatic segmentation feature) if the entire Procedure Division is too large to be held in memory at one time.

Once the OCCURS clauses have established the necessary tables, the program must be able to access the elements of those tables individually. Subscripting and indexing are the two methods provided by COBOL for accessing individual elements.

5.4 SUBSCRIPTING AND INDEXING

To refer to a particular element within a table, simply follow the name of the desired element with a parenthesized subscript or index. A subscript is an integer or a data name that has an integer value; the integer value represents the desired occurrence of the element -- an integer value of 3, for example, refers to the third occurrence of the element. An index is a data name that has been named in an INDEXED BY phrase in the OCCURS clause.

5.4.1 Subscripting with Literals

A literal subscript is simply a parenthesized integer whose value represents the occurrence number of the desired element. In figure 5-11, the literal subscript in the MOVE instruction (2) causes the software to move the contents of the second element of the table, A-TABLE, to I-RECORD.

Table Description	01 A-TABLE. 03 A-GROUP PIC X(5) OCCURS 10 TIMES.
Procedural Instruction	MOVE A-GROUP(2) TO I-RECORD.

Figure 5-11
Literal Subscripting

If the table has more than one level (or dimension), follow the name of the desired item with a list of subscripts, one for each OCCURS clause to which the item is subordinate. The first subscript in the list applies to the first OCCURS clause to which the item is subordinate. (This is the most encompassing level -- A-GROUP in the following example.) The second subscript in the list applies to the next most encompassing level, and the last subscript applies to the lowest level OCCURS clause being accessed (or the desired occurrence number of the item named in the procedural instruction -- ITEM5 in the following example).

Consider Figure 5-12; the subscripts (2,11,3) in the MOVE instruction cause the software to move the third repetition of ITEM5 in the eleventh repetition of ITEM3 in the second repetition of A-GROUP to I-FIELD5. (For illustration simplicity, I-FIELD5 is not defined.) (ITEM5(1,1,1) would refer to the first occurrence of ITEM5 in the table and ITEM5(5,20,4) would refer to the last occurrence of ITEM5.)

Table Description	01 A-TABLE. 03 A-GROUP OCCURS 5 TIMES. 05 ITEM1 PIC X. 05 ITEM2 PIC 99 COMP OCCURS 20 TIMES. 05 ITEM3 OCCURS 20 TIMES. 07 ITEM4 PIC X. 07 ITEM5 PIC XX OCCURS 4 TIMES.
Procedural Instruction	MOVE ITEM5(2,11,3) TO I-FIELD5.

Figure 5-12
Subscripting a Multi-dimensional Table

NOTE

Since ITEM5 is not subordinate to ITEM2, an occurrence number for ITEM2 is not permitted in the subscript list.

Figure 5-13 summarizes the subscripting rules for each of the above items and shows the size of each field in bytes.

NAME OF FIELD	NUMBER OF SUBSCRIPTS REQUIRED TO REFER TO THE NAMED FIELD	SIZE OF FIELD
A-TABLE	NONE	1110
A-GROUP	ONE	222
ITEM1	ONE	1*
ITEM2	TWO	2
ITEM3	TWO	9
ITEM4	TWO	1
ITEM5	THREE	2
* Plus a slack byte		

Figure 5-13
Subscripting Rules for a
Multi-dimensional Table

5.4.1.1 Operations Performed by the Software - When a literal subscript is used to refer to an item in a table, the software performs the following steps to determine the exact address of the item:

1. The compiler converts the literal to a 1-word binary value.

2. The compiler range checks the subscript value (the value must not be less than 1 nor greater than the number of repetitions specified by the OCCURS clause) and prints a diagnostic message if the value is out of range.

3. The compiler decrements the value of the subscript by 1 and multiplies it by the size of the item that contains the OCCURS clause corresponding to this subscript, thus forming an index value; it then stores this value in the object program.

4. At object execution time, the run time system adds the index value (from 3 above) to a base address, thus determining the address of the desired item.

5.4.2 Subscripting with Data-names

As discussed earlier in this section, subscripts may also be specified using data-names instead of literals. To use a data-name as a subscript, simply define it as a numeric integer (COMP or DISPLAY). It may be signed, but the sign must be positive at the time it is used as a subscript.

The sample subscripts in figure 5-14 refer to the same element accessed in the previous sample, Figure 5-12, (2,11,3).

Data Descriptions of Subscript data names	01 KEY1 PIC 99 USAGE DISPLAY. 01 KEY2 PIC 99 USAGE COMP. 01 KEY3 PIC S99.
Procedural Instructions	MOVE 2 TO KEY1. MOVE 11 TO KEY2. MOVE 3 TO KEY3. GO TO TABLERTN. TABLERTN. MOVE ITEM5(KEY1 KEY2 KEY3) TO I-FIELD5.

Figure 5-14
Subscripting with Data-names

5.4.2.1 Operations Performed by the OTS - When a data-name subscript is used to refer to an item in a table, the OTS performs the following steps at object execution time:

1. If the data-name's data type is DISPLAY, the software converts it to a 1-word binary value.

2. The software range checks the subscript value (the value must not be less than 1 nor greater than the number of repetitions specified by the OCCURS clause) and terminates the object program (with a diagnostic message that refers to the line number of the verb currently being executed) if it is out of range.

3. The software decrements the value of the subscript by 1 and multiplies it by the size of the item that contains the OCCURS clause corresponding to this subscript, thus forming an index value.

4. The software adds the index value (from 3 above) to a base address, thus determining the address of the desired item.

5.4.3 Subscripting with Indexes

The same rules apply for the specification of indexes as apply to subscripts except that the index must be named in the INDEXED BY phrase of the OCCURS clause.

An index-name item (an item named in the INDEXED BY phrase of the OCCURS clause) has the ability to hold an index value. (The index value is the product formed in step C of the operations performed by the software for literal or data-name subscripts -- the relative location, within the table, of the desired item.)

The compiler allocates a 2-part data item for each name that follows an INDEXED BY phrase. These index-name items cannot be accessed as normal data items; they cannot be moved about, redefined, written to a file, etc. However, the SET verb can change their values and relation tests can examine their values. One part of the 2-part index-name item contains a subscript value and the other part contains an index value. Consider the following illustration:

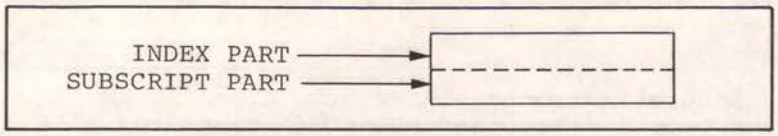


Figure 5-15
Index-name Item

Whenever a SET statement places a new value in the subscript part, the software performs an index value computation and stores the result in the index part. Only the subscript part of the item acts as a sending or receiving field. The index part is never altered by any other operation and is never moved to another item. It is used only when the index-name is used as an index referring to a table item. The sample MOVE statement in Figure 5-16 would move the contents of the third repetition of A-GROUP to I-FIELD. (For illustration simplicity once again, I-FIELD is not defined.)

Table Description	01 A-TABLE. 03 A-GROUP OCCURS 5 TIMES INDEXED BY IND-NAME.
Procedural Instructions	SET IND-NAME TO 3. MOVE A-GROUP (IND-NAME) TO I-FIELD.

Figure 5-16
Subscripting With Index-name Items

5.4.3.1 Operations Performed by the OTS - The OTS performs the following steps when it executes the SET statement:

1. The OTS converts the contents of the sending field of the SET statement to a 1-word binary value.

2. The OTS range checks the value (the value must not be less than 1 nor greater than the number of repetitions specified in the OCCURS clause) and terminates the object program with a diagnostic message if it is out of range.

3. The OTS decrements the value by 1 and multiplies it by the size of the item that contains the OCCURS clause, thus forming an index value.

Once the SET statement has been executed and the software has encountered the index-name item as an index, it only has to add the index value (from 3 above) to a base address to determine the address of the desired item. Since this is the only action performed, the execution speed of a procedural statement with an indexed data-name is equivalent to a reference with a literal subscript.

In addition, PDP-11 COBOL initializes the value of all index-name items to a subscript value of 1 (index value of 0), hence an attempt to use an index-name item as an index before it has been the receiving field of a SET verb will not result in an out-of-range termination.

NOTE

Initialization of index-name items is an extension to the ANS COBOL standards. Users concerned with writing COBOL programs that adhere to standard COBOL should not rely on this feature.

5.4.3.2 Relative Indexing - To perform relative indexing, when referring to a table item, simply follow the index-name with a plus or minus sign and an integer literal. The OTS computes the index value of the literal only and adds it to (+), or subtracts it from (-) the index part of the index-name item in a temporary location. This forms a new index value which the OTS adds to a base address to determine the address of the table item. A test is made on the index value thus computed to ensure that it is within range. The index portion of index-name is not altered.

The sample MOVE instruction in Figure 5-17 moves the fourth repetition of A-GROUP to I-FIELD if IND-NAME has not been altered with a SET verb.

```
MOVE A-GROUP(IND-NAME + 3) TO I-FIELD.
```

Figure 5-17
Relative Indexing

The actual operation of accessing a table element is shorter at run time since the compiler has calculated the index value of the literal at compile time and has stored it in the object program ready for use. Relative indexing, therefore, involves two additions and a range check during object execution. It leaves the index-name item unaltered.

5.4.4 Index Data Items

Often a program will require that the value of an index-name item be stored outside of that item. It is for this purpose that PDP-11 COBOL provides the index data item.

Index data items are 1-word binary integers with implicit synchronization. (The 1-word size corresponds to the subscript part of the index-name item.) They must be declared with a USAGE IS INDEX phrase and they may be modified (explicitly) only by the SET statement.

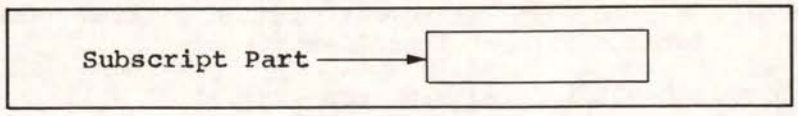


Figure 5-18
Index Data Item

Since index data items are considered to contain only the subscript part of an index-name item, when a SET statement "moves" an index-name item to an index data item, only the subscript part is moved. Likewise, when a SET statement "moves" an index data item to a index-name item, the index data item is moved to the subscript part of the index-name item and a new index value is computed by the software. This is done to guarantee that an index-name item will always contain a good index value.

The only advantage gained by using index data items over numeric COMP items is that the data description is shorter, easier to write, and more self-documenting. Further, the restrictions placed on access to index items may be useful in debugging the program.

5.4.5 The SET Statement

The SET statement alters the values of index-name items and copies their values into other items. When used without the UP BY/DOWN BY clause, it functions like a MOVE statement. Figure 5-19 illustrates the legal data movements that the SET statement can perform.

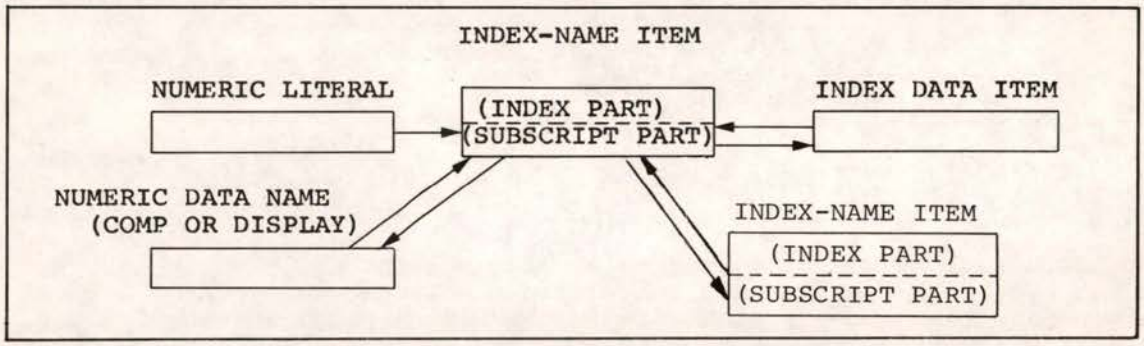


Figure 5-19
Legal Data Movement with the SET Statement

The SET statement may be used with the UP BY/DOWN BY phrase to alter the value of an index-name item arithmetically. The numeric literal is added to (UP BY) or subtracted from (DOWN BY) the subscript part, and the index part is recalculated by the software after the appropriate range check.

CHAPTER 6

FILE HANDLING

6.1 INTRODUCTION

PDP-11 COBOL provides two ways to arrange the records in its files (file organization): sequential and relative. The ORGANIZATION phrase specifies the file organization for COBOL files.

The format of a file is defined in the Data Division and is determined by the compiler. Reading and writing of the file is performed by the COBOL object time system (OTS).

PDP-11 COBOL provides three ways to read or write the records in its files (file access): sequential, random, and dynamic. The ACCESS MODE clause specifies the file access mode for every file used by COBOL programs.

The following chart shows the two file organization methods and the file access methods that apply to each of them:

FILE ORGANIZATION	FILE ACCESS
SEQUENTIAL	SEQUENTIAL
RELATIVE	SEQUENTIAL RANDOM DYNAMIC

Once a program creates a file, all other programs that access it must describe it with the same file organization. For example, it is not possible to create a sequential file in one program and read it as a

relative file with another program.

This chapter discusses PDP-11 COBOL file handling. It begins with a discussion of the two file organization methods (sequential and relative), and goes on to buffering, filenames, and the statements that access COBOL input/output files. It includes a section on the ACCEPT and DISPLAY statements; a section on the compatibilities between COBOL, FORTRAN, and MACRO files; and concludes with a section on I/O errors and the I/O error messages.

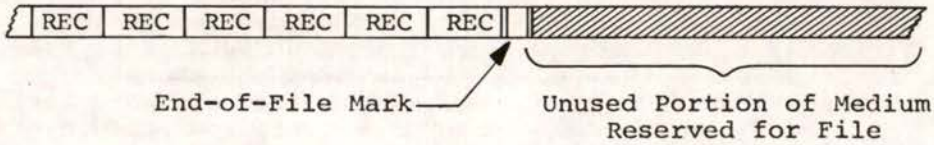
6.2 SEQUENTIAL FILE ORGANIZATION

6.2.1 Format

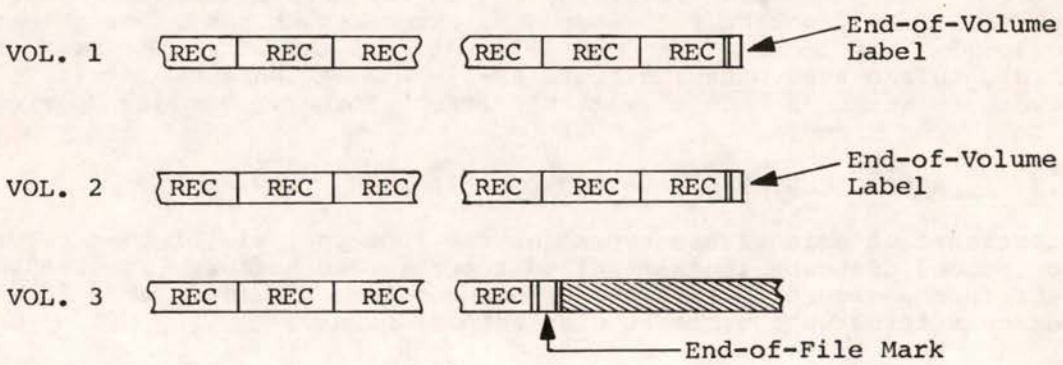
Sequential file organization arranges the records in a file serially; each record (except the first) has another record preceding it and each record (except the last) has another record following it. The records remain in the order in which they were written. Thus, COBOL statements cannot delete records from the file, insert new records between existing records, or alter the order of the existing records in any way. However, they can replace existing records and add new records onto the end of the file.

The opening operation for reading, writing, or updating sequential files must begin with the first record in the file and proceed by the prescribed order through the file. For example, to read a particular record in the file, say the 15th record, the program must open the file and successfully execute 14 READ statements before the 15th execution can read the desired record. The program can read all of the remaining records (from record 16 on), but it cannot read any record prior to record 16 without opening the file again and beginning with record 1.

Sequential files always contain an end-of-file mark that designates the end of the file. COBOL statements can write over the end-of-file mark and, thus, extend the length of a file. (The software inserts another end-of-file mark after the last record written.) Since the end-of-file mark indicates the end of useful data, PDP-11 COBOL provides no method for reading the area of the medium which is reserved for the file but is beyond the end-of-file mark.



Often a file with sequential organization is so large that it requires more than one volume (such as a multi-reel DECtape file). PDP-11 COBOL provides an end-of-volume label for multi-volume files that marks the end of recorded information on each volume, and signals the file system to switch to a new volume. On multi-volume files, the end-of-file mark appears once, at the end of the last record on the last volume.

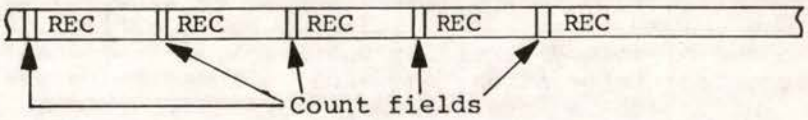


A Multi-Volume File

6.2.2 Record Size

If there is only one record description for a file or if there are more than one that describe the same length record, that file contains fixed-length records. When a program creates a sequential file with fixed-length records, the software places the record length in the file header when it creates that file (file headers for RSX-11D files are described in RSX-11D I/O Operation Reference Manual, DEC-11-OXFSA-A-D). When a COBOL statement requests a record from that file, the software releases a record as long as the number of characters specified by the file header. The medium contains no additional record size information; the records contain no record size "count fields" as do variable-length records.

If the data descriptions for a sequential file consist of more than one record description, describing several different-sized records, that file contains variable-length records. When a program creates a sequential file with variable-length records, the software places a count field in front of each record it writes into the file. This count field contains the number of character positions in the record. When a COBOL statement requests the record, the software releases a record as long as the number of characters specified by the count field. The OTS creates and uses the count field automatically. COBOL statements cannot access it during input operations, and the 01 level record description entries must not describe it.



6.2.2.1 RECORD CONTAINS Clause - The RECORD CONTAINS clause documents the size of the data records, and, if the data descriptions for a sequential file describe the records as fixed-length (only one record description, or more than one but all of them for the same size record), this clause with its "integer-1 TO ..." phrase forces the software to set up a file of variable-length format records. Consider the following example:

```
RECORD CONTAINS 200 TO 350 CHARACTERS
```

The presence of this clause overrides the fact that all of the record descriptions describe the same sized record. The software ignores the values in the clause; it is the presence of the clause in the format shown that forces the variable size record format.

If the clause does not contain the "integer-1 TO ..." phrase, the software ignores it completely.

If the data descriptions for a sequential file describe variable-length records, the software sets up variable sized records automatically and ignores this clause.

Even though the software ignores it, the clause may be used in any program to document record sizes.

6.2.2.2 Print-Controlled Records - If a sequential file is described in a LINAGE IS clause or an APPLY PRINT-CONTROL clause, and the file is not going directly to a printing device (is going to be spooled), the software designates the file as a print-controlled file. Print-controlled files contain form advancing information with each record.

When a program creates a sequential file that has been designated as a print-controlled file, the software places a forms control word (two bytes) before each record in the file and places forms control information in the control word of each record based on the ADVANCING clause of the WRITE statement for each record. It places the word and information on each and every record in the file (whether fixed or variable in size), thus increasing each record's size by two bytes.

When a program writes a print-controlled file to a device, such as magtape or disk, the OTS also places an entry in the file header so that when the file is later printed, the spooler will recognize the file as a print-controlled file. The spooler removes the forms control word from each record and interprets it as a carriage return and the appropriate number of line feeds.

When a program processes a print-controlled file that has been written to magtape or disk, the OTS detects the print-control entry in the file header and accepts the extra word in each record as part of the record size. Although the OTS reads the forms control word, it does not make the word available to the program.

If a REWRITE statement replaces the record in the file, it replaces the forms control bytes without change. Thus, a program can read (and update) a print-controlled file with no effect on the forms control information associated with each record.

6.2.3 Record Blocking

The manner in which the file system blocks the records of sequential files depends on the device to which the file is assigned and the presence and format of the BLOCK CONTAINS clause.

COBOL programs can assign sequential files to DECtape and disk, which both require fixed-length physical blocks, and to magnetic tape, which allows variable-length blocks.

If the file description for a sequential file does not have a BLOCK CONTAINS clause, the compiler assigns default blocking to the file. If the file description does have a BLOCK CONTAINS clause, but ends with the key word CHARACTERS, as shown below, the compiler ignores the clause and assigns default blocking.

```
BLOCK CONTAINS 512 CHARACTERS.
```

The compiler assigns default blocking as follows:

- If the program assigns the file to disk, the OTS packs the records together (end-to-end) until the file fills a block; then it writes the block to disk and places the remaining part of any partially-written record into the next block. Thus, the disk default mode allows records to span block boundaries. (Henceforth, this guide refers to this technique as record spanning.)

- If the program assigns the file to magnetic tape, the OTS does not block the records at all, but considers each record to be a physical block. (Henceforth, this guide refers to this technique as unblocked.)

If the file description has a BLOCK CONTAINS clause that ends with the key word RECORDS, as shown below, the compiler overrides default blocking.

```
BLOCK CONTAINS 300 RECORDS
```

The compiler overrides default blocking as follows:

- If the program assigns the file to disk, the OTS prohibits record spanning. It places a fixed or variable number of full records in each block (depending on whether the records are fixed or variable) and writes the block when it cannot place another full record in the block. The software ignores the number (of records) stated in the BLOCK CONTAINS clause, since the block size of disk is fixed.

- If the program assigns the file to magnetic tape, the compiler uses the number (of records) stated in the BLOCK CONTAINS ... RECORDS clause to determine the size of the buffer to reserve for that file. It calculates the buffer size by multiplying the number stated in the clause (integer-2) by the size of the largest record in the file. It determines the largest record size by rounding the size of the largest record up to an even number; adding two bytes if the file header has a print-control entry; and, adding two more bytes (for the count field) if the records have a variable-length record format. Thus, the compiler uses the BLOCK CONTAINS ... RECORDS clause to determine the size of a buffer, not to determine the number of records placed in each block. The OTS writes the block when it cannot place another full record into it. For fixed-length records, this has the same effect as if the software had counted the records rather than the bytes that they occupy. For variable-length records, this guarantees that even if all of the records are of maximum size, that each block contains at least the number of records specified by the BLOCK CONTAINS clause. However, if the records vary in size by a sufficient amount and many of the records are smaller, the block may contain more than the number of records specified by the clause.

When a program assigns a file to magnetic tape, in order to guarantee an accurate allocation of buffers, all programs that access the file must describe it the same way that the creating program described it.

Files that are assigned to disk carry the record spanning indicator, and print-control indicator. When these files are opened for INPUT, I-O (Input/Output), or EXTEND, the indicators that they carry override any entries in the new file description.

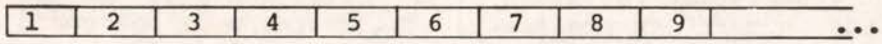
6.2.4 Device Restrictions

Sequential files may reside on any medium. Files stored on cards, magtape, paper tape, printing and display devices must be sequential. Files stored on disk may be either sequential or relative.

6.3 RELATIVE FILE ORGANIZATION

Relative file organization arranges the records of the file into numbered record positions. It assigns each record position a number

that identifies that position relative to the beginning of the file (the first record position in the file has record number 1, the second has record number 2, etc.).



Record Positions in a Relative File

When a program executes a READ or a WRITE operation on a relative file, the OTS places a value in the relative key. The relative key uses this value to select records from these numbered record positions in the same way that a subscript selects an item in a table.

Thus, while sequential and relative files both arrange their record positions in a serial order, COBOL statements can address the record positions of a relative file by their position numbers, and successive accesses do not have to proceed through the file in a prescribed, serial, order. In short, the statements can access the records randomly.

Another significant feature of relative file organization is that each record position does not have to contain a valid record. Although each position actually occupies one record space, a flag indicates whether or not that space contains a valid record. Thus, a file may have fewer records than it has record positions, and the indicated empty record positions may be anywhere in the file.

The numerical order of the record positions remains the same during all operations on a relative file; however, the accessing statements can move a record from one position to another, delete a record from a position, or insert new records into empty positions.

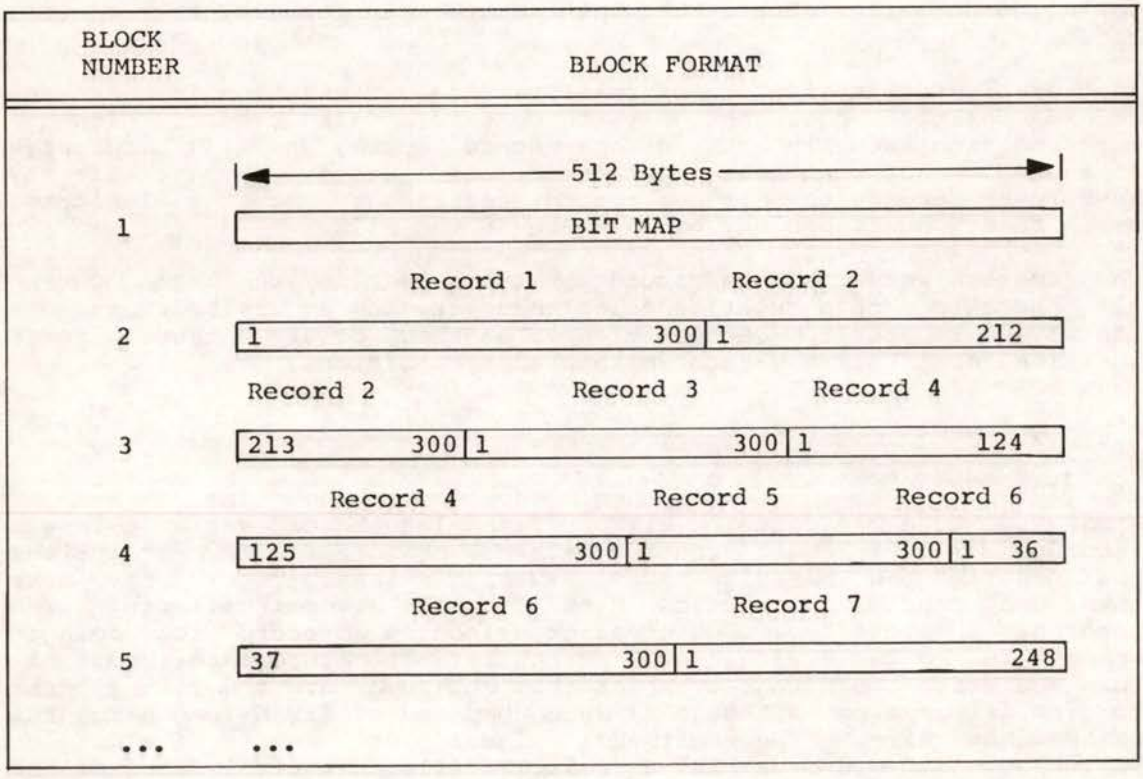
6.3.1 Record Size

A relative file may contain either fixed-length or variable-length records. (Fixed-length records have one or more record descriptions that describe the same size record. Variable-length records have more than one record description that describe several different sized records.) However, the COBOL compiler allocates a record area, both in memory and on the device, equal to the largest record described. All read and write operations transfer this entire area; the file appears to the file system as though it were composed of fixed-length records of maximum size. The software places no count fields on variable-length records in a relative file; it simply records the size of the largest record described in the file header and uses that size to release records when it reads the file. Thus, when the software opens the file for reading, the pre-recorded information in the file header supersedes the information supplied by the program (description of record size, method of blocking, etc.).

6.3.2 Record Blocking

Relative files may reside only on disk. A standard disk block is 256 words long. The first block of a relative file contains a bit map for the first 4096 record positions of the file. Each bit of the bit map corresponds to a record position (bit 1 corresponds to record position 1, bit 2 to record position 2, etc.) and indicates whether the position is occupied (1) or empty (0). If the length of the file exceeds 4096 records, the OTS writes another bit map block in the first available block following record position 4096. Thus, each group of 4096 records has its own bit map.

Relative file records will span physical blocks unless the BLOCK CONTAINS clause suppresses record spanning. Since all of the records in a relative file are the same size, the file system writes the records on the disk medium without regard to physical block boundaries when using record spanning. For example, a record 300 bytes long would cause the following mapping at the beginning of the file:



Record spanning causes the system to write files compactly, in the minimum amount of space. Further, record spanning can reduce access

time by reducing head movement. However, the system must perform two disk read operations to access a record that spans two blocks, and the two operations increase the time required to read that record. In the preceding example, a single read operation can read only records 1, 3, and 5. Records 2, 4, 6, and 7 span blocks and each, therefore, requires two read operations.

The presence of the BLOCK CONTAINS ... RECORDS clause in the file description of a relative file suppresses record spanning, resulting in an integral number of records per block. The system ignores the integer value supplied by the clause (since the hardware has fixed the disk block size at 512 bytes) and packs as many complete records into a block as the 512 bytes will take.

To determine blocking factors and file sizes of files that do not permit record spanning, perform the following calculations:

1. Round the record size to an even number of bytes. (Since each record occupies an integral number of words and since each word requires two bytes, all records occupy even numbers of bytes.)

2. Divide the rounded number of bytes into the number 512 to calculate the number of records in a block.

3. Divide the number of records in a block (from Step 2) into the number of records in the file to calculate the file size in blocks.

For example, to determine the number of 85 byte records that the system places in a block, round 85 to 86 and divide it into 512. The number 86 goes into 512 five times, so the blocking factor for this record is 5. (If the number of bytes were not rounded, this particular computation would yield an erroneous answer of 6.) If the file contains a thousand records, divide 5 into 1000 and find that the file requires 200 blocks.

The following table illustrates the blocking factors for files that do not permit record spanning. When more than one record size occurs on a line, those record sizes all result in the same number of records per block. The unused bytes for each of these is shown by position; thus, 58 results in 8 records per block and 48 unused bytes, 60 results in 8 and leaves 32, etc. The ellipses separating the larger record sizes (from 86 on) represent every even number between the two numbers on the line. All of the numbers on a line result in the same number of records per block. The unused bytes between record sizes 86 and 102 decrease by 10 for each 2-byte increase; thus, a record size of 88 leaves 72 unused bytes; 90 leaves 62, etc. They decrease by 12 for each two bytes between 130 and 170; by 4 between 172 and 256; and by 2 between 258 and 512.

Table 6-1
Blocking Factors

Record Size in Bytes (after rounding)	Number of Records per Block	Unused Bytes at End of Block
2	256	0
4	128	0
6	85	2
8	64	0
10	51	2
12	42	8
14	36	8
16	32	0
18	28	8
20	25	12
22	23	6
24	21	8
26	19	18
28	18	8
30	17	2
32	16	0
34	15	2
36	14	8
38	13	18
40 42	12	32 8
44 46	11	28 6
48 50	10	32 12
52 54 56	9	44 26 8
58 60 62 64	8	48 32 16 0
66 68 70 72	7	50 56 22 8
74 76 78 80 82 84	6	68 56 44 32 20 8
86 ... 102	5	82 72 ... 2
104 ... 128	4	96 88 ... 0
130 ... 170	3	122 110 ... 12
172 ... 256	2	68 64 ... 0
258 ... 512	1	254 252 ... 0

Table 6-1 illustrates that as record sizes increase so do the number of unused bytes at the ends of the blocks. Certain sizes (56, 64, 72, 84, 102, 128, 170, and 250) minimize this wasted space and can still be retrieved by a single disk read operation. If saving disk space and access time are important factors in your record design, consider these sizes.

6.3.3 Media Restrictions

Since the file system accesses relative files randomly, they must reside on a device that can quickly select any record position in the file. Thus, relative files are restricted to disk media.

6.4 CURRENT RECORD AREA

The file system reserves a record processing area in memory for each file. This area is the current record area.

The system fixes the location of the current record area when it opens the file. It reserves an area as large as the largest record described for the file. (If the length of the largest record is an odd number of bytes, the system adds an extra byte so the record area occupies an integral number of words.) Further, the system reserves a word in front of each current record area for possible print-control bytes.

6.4.1 SAME RECORD AREA Clause

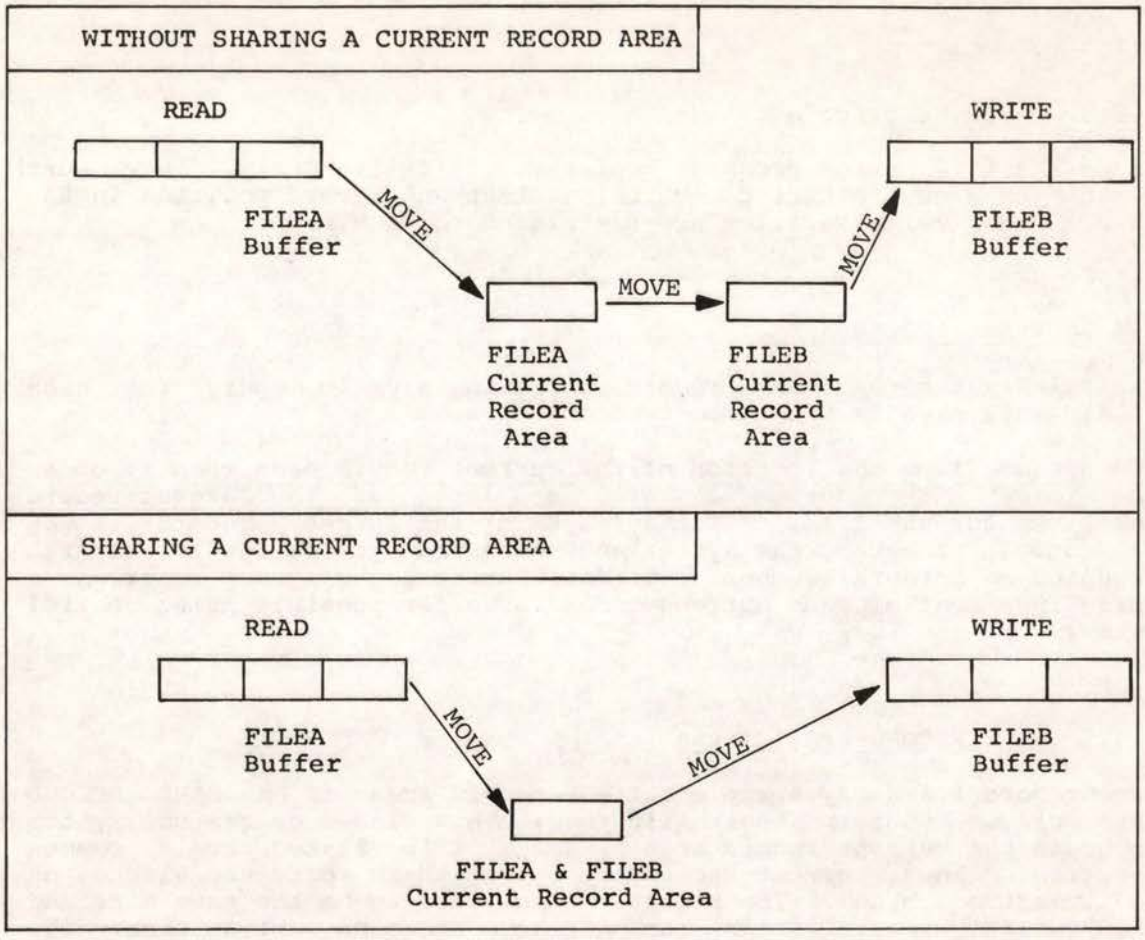
Two or more files may share a current record area if a SAME RECORD AREA clause contains their filenames. This clause causes the system to begin the current record area of each file listed at a common location. (Thus, current record areas that share space are aligned on their leftmost bytes.) The records do not have to be the same size and the current record areas need not have the same maximum size. The following sample statement would cause FILEA and FILEB to share the same current record area:

```
I-O-CONTROL.

    SAME RECORD AREA FOR FILEA FILEB

    ...
```

Since the system places a file's current record area in a separate location from its buffers, each READ, WRITE, and REWRITE operation causes a record to move between the buffers and the current record area. (The following section discusses this record movement in greater detail.) When a program reads a record from a file, modifies it, and writes it into another file, a SAME RECORD AREA clause, containing both filenames, can save an entire move of the record. The following illustration shows these record movements:



Record Movement Caused by
Reading, Processing, and Writing
Records in Two Files

6.5 BUFFERING

When the system performs an input operation, it reads a block from the medium into the buffer, and moves a record from the buffer to the current record area. Each subsequent read operation moves a record from the buffer to the current record area. When it has exhausted the buffer (has read an entire block), the system reads another block into the buffer.

When performing an output operation, each write operation moves a record from the file's current record area into the file's buffer. Each subsequent write operation moves a record from the current record area into the buffer. When it has filled the buffer (has written an entire block of records), the system writes the block to the medium.

The following subsections discuss the size of the buffers, the number of buffers, and the sharing of buffers.

6.5.1 Buffer Size

The size of the buffers assigned to disk and DECtape files is always 256 words (512 bytes). The size of the buffers for files on magnetic tape depends on the size of the largest record in the file and on the blocking factor.

6.5.2 Double Buffering

Unless instructed otherwise, the compiler assigns each file a single buffer. The RESERVE ... AREAS clause instructs the compiler to increase the number of buffers for the file to two. (The system ignores the value supplied by the clause; it is simply the presence of the clause that causes the system to set up an additional single alternate buffer area.)

When the RESERVE clause is specified, the file control system, by default, performs a read-ahead on buffers of files open for input (INPUT), and a write-behind on files open for output (OUTPUT) or INPUT/OUTPUT (I-O). The APPLY READ-AHEAD clause in the I-O-CONTROL paragraph can change the I-O default from write-behind to read-ahead.

When the RESERVE clause sets up two buffers for a particular file, the file control system uses them as follows:

- Read-ahead -- When a READ statement fetches a record from an input file, it also fetches enough records from adjacent higher numbered record positions to fill both buffers. Then, when the program exhausts a buffer, the file system fills the exhausted buffer (from the device) while the program reads records from the other buffer.

- Write-behind -- When a WRITE statement writes a record into an output or input/output file, the file system places the record into one of the two buffers. Then, when the program fills a buffer, the system writes that buffer to the device while the program places records into the other buffer.

6.5.3 Sharing Buffer Space Among Files

The SAME AREA clause provides a simple method of sharing buffer space among several files.

Two or more files share the same buffers if the SAME AREA clause contains their filenames and only one of them is open at any time during program execution. Further, since only one file is open at a

time, the files will also share the same current record area. The OTS sets the size of the current record area to the size of the largest record description specified in the group.

If only one of these files is open at a time, the following sample statement causes them all to share the same buffer and current record area.

```
I-O-CONTROL.

    SAME AREA FOR FILEA FILEB FILEC.

    ...
```

6.6 DEVICES, FILES, FILENAMES

The PDP-11 COBOL object time system, through the use of the operating system's file control services, supports any device that the operating system supports. This includes the following devices. (For further information concerning the devices themselves, see Chapter 7, Devices, in this guide.)

Table 6-2
Device Codes

DEVICE	DEVICE CODE
Card Reader	CR
Disk (RK03, RK05)	DK
Disk (RF11/RS11)	DF
Disk (RP11/RP03)	DP
DECtape	DT
Line Printer	LP
Magnetic Tape	MT

The OTS and the operating system use the device codes to communicate with the devices. Further, the COBOL OTS uses the operating system's file specification and interfaces for all file manipulation with file storage devices (disk, DECtape, magtape). The VALUE OF ID clause (discussed in the following subsection) in the FD entry describes the

file specification to the OTS. The format for the full file specification follows:

```
dev:[uic] filename.typ;version/switches
```

The meanings of these entries follow:

dev: - device code

[uic] - user's identification code or the code of the user for whom the file was created - the user directory ID. (The brackets [] are required.)

filename - an alphanumeric field containing up to nine characters that identify the file.

typ - an alphanumeric field containing up to three characters that qualify the filename.

version - a numeric field containing up to five octal digits that give the version number of the file. By specifying version numbers, the user can maintain several versions of the same file on a directory device.

switches - identifies certain actions for the operating system to perform for the file. (Subsection 6.6.1.1 discusses these switches.)

These entries default as follows:

dev: - the device code of the disk containing the operating system.

[uic] - the user identification code of the user currently using the system.

filename - null

typ - null

version - the version number defaults differently for input and output files.

- input files - the highest numbered version of the file (thus selecting the latest version);
- output files - one greater than that of the highest numbered version of the file (thus creating the latest version).

Switches - null.

For example, the following sample file specification causes the file system to process version 3 of a file on disk named ARIES. The user has an identification code of 140,222.

```
DK:[140,222]ARIES;3
```

6.6.1 Using Explicit File-names (VALUE OF ID Clause)

The VALUE OF ID clause, in the FD entry, describes the file specification to the COBOL OTS. The VALUE OF ID clause is optional; however, the system requires it whenever the program refers to an explicit file. The clause accepts either a literal entry or an identifier entry. Consider the following sample literal form of the clause:

```
VALUE OF ID IS "DK:[140,222]ARIES;3"
```

If the file specification in the VALUE OF ID clause includes a device code, that code supersedes any device code specified in the ASSIGN clause for the file. (Subsection 6.6.2 discusses the ASSIGN clause.)

When written in the literal form, the literal may be a complete file specification or a part of a file specification.

When written in the identifier form, the value of the identifier may be a complete or partial file specification.

The identifier form of this clause is especially useful when different runs of a program process different files. If a program must process different files in the same way on different runs, an ACCEPT statement in the Procedure Division can request a file specification from the user at the user's console or from a batch input stream.

The following example illustrates how a COBOL program could request a file specification from an interactive terminal:

```
DATA DIVISION
   .
   .
   .
FD FILEIN
   VALUE OF ID IS INFILE.
   .
   .
   .
WORKING-STORAGE SECTION.
   77 INFILE PIC X(20).
PROCEDURE DIVISION.
   .
   .
   .
   DISPLAY "TYPE IN INPUT FILE SPEC".
   ACCEPT INFILE.
   .
   .
   .
   OPEN INPUT FILEIN.
```

This sample coding causes the following interaction between the program and the user (the message printed by the program is underlined):

TYPE IN INPUT FILE SPEC
DK1:THOREAU

Following this interaction, the sample OPEN statement will open (for input) the file, THOREAU on DK1.

6.6.1.1 Switches - There are three optional switches which may qualify the file specification. These switches modify the processing performed by the COBOL OTS when it opens the file.

A list of the switches and their meanings to the OTS follows:

SWITCH	MEANING
/CL:n	Allocate disk space in "clusters" of n blocks whenever the file needs additional storage space during output operations. (n may be any number from 1 to 1000 (decimal).) If the number is followed by a decimal point, the software considers the number to be decimal; if it does not have a decimal point, it is considered to be octal). This switch would be used only when very large files are to be created and the output device can hold the entire file (i.e., an RP03 disk). The effect of this switch is to make file accessing faster when the file is being processed sequentially.
/CO:n	Allocate a contiguous file of n disk blocks to the file when it is opened. This switch ensures that n blocks are available for the file prior to actual processing. (When many users are sharing the same disk, you can use this switch to ensure that your entire file will fit on the disk.) It applies only to output files being created. (If n applies to an RK type disk, the maximum value of n is 4000 (decimal). If n applies to an RP type disk, the maximum is 26,000 (decimal). If a decimal point follows the number, the system considers it to be decimal; otherwise, octal.)
/SHR	This file is a shared file, available for processing by other tasks concurrently with the COBOL program.

6.6.2 Device Assignment by ASSIGN Clause

If the VALUE OF ID clause does not specify an explicit device, the ASSIGN clause in the File-Control paragraph can assign a default file specification to the file. The ASSIGN clause must be written as part of the SELECT statement as shown below:

```
SELECT THOREAU ASSIGN TO "DK1:"
```

This example assigns a default device code "DK1:" for the location of the file THOREAU. Another device code specification in the VALUE OF ID clause could override it later in the source program.

6.7 COBOL I/O STATEMENTS

The COBOL input/output statements allow COBOL programs to communicate with the system devices.

The COBOL I/O statements perform their actions in one way for relative files and in another way for sequential files. Therefore, this section separates the I/O statements by file organization; subsection 6.7.1 discusses all of the I/O statements as they apply to sequentially organized files, and subsection 6.7.2 discusses them as they apply to relatively organized files.

The following table shows which statements apply to the two file organization methods:

Table 6-3
I/O Statements

SEQUENTIAL I/O STATEMENTS	RELATIVE I/O STATEMENTS
CLOSE	CLOSE
OPEN	DELETE
READ	OPEN
REWRITE	READ
WRITE	REWRITE
	START
	WRITE

Before a COBOL program can access a file (whether sequential or relative), it must open the file; then, when the program is finished with the file, it must close the file.

6.7.1 Sequential I/O Statements

As Table 6-3 shows, the COBOL I/O statements, CLOSE, OPEN, READ, REWRITE, and WRITE, can refer to sequential files.

A COBOL program may open a sequential file in one of four modes, INPUT, OUTPUT, I-O (input/output), or EXTEND. The following table shows which statements apply to the four different OPEN modes of sequential files. (The table does not include the OPEN and CLOSE statements since they apply to all modes.)

Table 6-4
Sequential OPEN Modes

STATEMENT	OPEN MODE			
	INPUT	OUTPUT	INPUT-OUTPUT	EXTEND
READ	X		X	
REWRITE			X	
WRITE		X		X

6.7.1.1 Opening Sequential Files - The OPEN statement makes a file available for processing by a COBOL program.

A program must execute an OPEN statement for a file before it executes any other I/O statement for that file. Consider the following sample OPEN statement. It opens the file named THOREAU for input/output. The program containing this statement could, after executing it, READ, REWRITE, and CLOSE THOREAU.

```
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT THOREAU
    ASSIGN TO "DK1:".
.
.
.
DATA DIVISION.
FILE SECTION.
FD  THOREAU
.
.
.
PROCEDURE DIVISION.
.
.
.
    OPEN I-O THOREAU.
.
.
.
```

The OPEN statement must refer to the file by the filename appearing in both the SELECT clause in the Environment Division and the FD paragraph in the Data Division.

When the OTS executes an OPEN statement, it performs the following actions for the file named in the statement:

- If the file is already open, the OTS generates an error message and performs the USE procedure section (if specified). (Section 6.10 discusses USE procedures and Section 6.11 discusses error messages.)

- If the SELECT clause of the File-Control paragraph declares the file OPTIONAL, the OTS displays the following message:

  "FILE nnn ... OPTIONAL FILE MOUNTED? Y OR N?"

  (nn represents the filename.)

  If the file is available for processing, type a Y. If not, type an N. If the file is not available (N), the OTS disables all I/O processing on the file except READ and CLOSE; a later READ statement causes program control to take the AT END imperative path.

- If a SAME AREA clause contains the name of the file and none of the other files named in the clause is open, the OTS allocates buffer space for the file.

- When the file has passed all of the preceding checks and is ready for opening, the OTS instructs the file control system to open the file. If the file control system fails to open the file, the OTS reports an error condition and performs any applicable USE procedure (if present).

- If the program is creating the file (OPEN OUTPUT) and the file description specifies LINAGE or APPLY PRINT-CONTROL, the OTS initializes the LINAGE counters, and positions the print form to head of form.

  If the program writes the file to a device such as magnetic tape or disk, to be spooled to the printer later, the OTS makes an entry in the file header as discussed in Section 6.2.2.2.

- Finally, depending on which statements apply to the open mode, the OTS enables or disables all of the program's I/O

statements that refer to the file (see Table 6-4). For example, if the OPEN mode is INPUT, it enables all READ statements for that file and disables all REWRITE and WRITE statements for that file.

Since the EXTEND mode simply allows the WRITE statement to add records onto the end of the file, files opened in this mode must already exist on disk or tape. If the file does not exist, the OPEN statement fails and the OTS issues an error message.

6.7.1.2 Reading Sequential Files - The READ statement makes the next logical record of an open sequential file available to the program. If the preceding I/O operation was an OPEN, it makes the first record of the file available to the program.

Consider the following example. If the last I/O operation on the file named THOREAU was an OPEN, this statement would provide the program with the first record in the file THOREAU. Every time the statement is executed, it provides the program with the next sequential record in THOREAU. Program control transfers to the paragraph named LIBRARY when an end-of-file mark is encountered during the READ.

```
BEGIN.  OPEN THOREAU.
LOOP.   READ THOREAU AT END GO TO LIBRARY.
.
.
.
        GO TO LOOP.
```

If the file contains variable-length records, the program must determine the length of the record just read. The OTS supplies no such information.

If the file is open in the I-O mode, the successful execution of a READ statement enables any following REWRITE of the record just read. (For further information on the REWRITE statement, see the following subsection -- 6.7.1.3.)

If the file has more than one record description, the records automatically share the same current record area. The OTS does not clear this area before it executes the READ statement (no blank filling, etc.). Therefore, if the record read by the latest READ statement does not fill the entire current record area, the area not overlaid by the incoming record remains unchanged. For example, if the file's record area contains ten 3's, and a READ operation moves in a 6-character record containing all 1's, the current record area then contains six 1's followed by four 3's. Consider the following example:

Current Record Area with all 3's	3333333333
Next Record in the File	111111
Current Record Area after READ	1111113333

6.7.1.3 Rewriting Records into Sequential Files - The REWRITE statement places the record just read from an input-output file back into its file on disk or magnetic tape. (The WRITE statement cannot access I-O files.) The following sample statement writes the record, REC1, back into its file. (REC1, of course, must be a record in the file read by the preceding READ for that file.)

```
REWRITE REC1
```

Before the REWRITE statement can refer to a record, the program containing the statement must meet the following conditions:

- The file containing the record must be open in the I-O mode;
- The last I/O operation on the file containing the record must have been a successful READ;
- The record length of the record to be rewritten must be the same as the record last read from the file.

6.7.1.4 Writing Sequential Files - The WRITE statement releases a logical record to an output file, thereby creating an entirely new record in the file.

The following sample WRITE statement releases the record PRINT-LINE to the device assigned to that record's file then skips 3 lines. When it reaches the end of a page (as specified by the LINAGE clause), it causes program control to transfer to the subroutine, HEADER-RTN.

```
WRITE PRINT-LINE BEFORE ADVANCING 3 LINES
    AT EOP GO TO HEADER-RTN.
```

Note that this produces two blank lines following every line printed.

The WRITE statement releases records to files that are open in either the output or extend mode. The following text discusses the two modes separately.

- OUTPUT Mode - The WRITE statement can create the following two kinds of files in the OUTPUT mode:

  1. Print-files - A print-file produces a listing on a printing device. The LINAGE clause or the APPLY PRINT-CONTROL clause designates a file as a print-file. Any information contained in the optional ADVANCING phrase will reside in the forms control word. If the file is not described with either the LINAGE or APPLY PRINT-CONTROL clauses and the ADVANCING option is present, the software writes a storage file with an appropriate number of blank records.

present but it will not be enabled.)

The statement does not have to release print-files directly to a printing device, but may also release them to a storage medium such as disk for printing at a later time. When a program handles a file in this manner, the OTS designates the file as a print-controlled file and places a word containing carriage control information before each record in the file. (For further information concerning print-controlled files, see Section 6.2.2.2 in this chapter.)

2. Storage files - A storage file remains on disk or tape for future reference. All files that are not print-files are storage files. A sample storage file WRITE statement follows; this statement produces a storage file of the records named WALDEN:

```
WRITE WALDEN
```

- EXTEND Mode - A WRITE to a storage file opened in the EXTEND mode simply adds new records logically in sequence after the last record in the file. As the statement extends the file, the file control system automatically handles requests for additional storage space. (Print-files on disk cannot be opened for EXTEND.)

6.7.1.5 Closing Sequential Files - The CLOSE statement terminates processing on the file referred to in the statement. The following sample CLOSE statement terminates processing on the file named THOREAU:

```
CLOSE THOREAU
```

When the CLOSE statement closes a file, no other I/O operations can access that file until another OPEN statement opens the file.

If the statement specifies the LOCK option, the program cannot open the file again in this run. The CLOSE statement with the LOCK clause is shown below:

```
CLOSE THOREAU WITH LOCK
```

The lock option has no effect on the physical device containing the file.

If a SAME AREA clause contains the name of the file just closed, the program may open one of the other files named in the clause.

6.7.2 Relative I/O Statements

The COBOL I/O statements, CLOSE, DELETE, OPEN, READ, REWRITE, and START can refer to relative files.

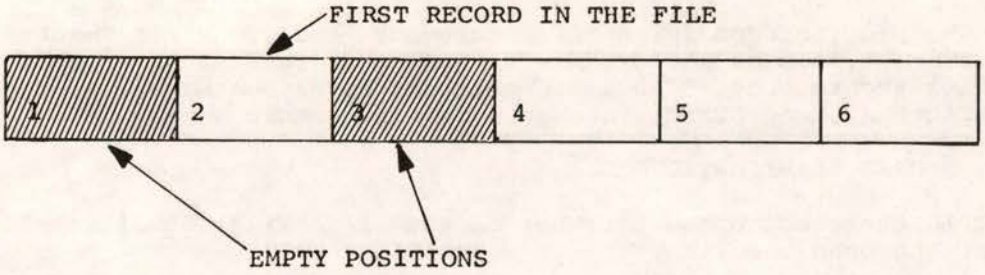
A COBOL program may open a relative file in one of three modes, INPUT, OUTPUT, or INPUT/OUTPUT, and access an open relative file in one of three ways, sequentially, randomly, or dynamically. The following table shows which statements and access methods apply to the three different OPEN modes of relative files.

Table 6-3
Relative OPEN Modes

FILE ACCESS MODE	STATEMENT	OPEN MODE		
		INPUT	OUTPUT	INPUT-OUTPUT
Sequential	DELETE			X
	READ	X		X
	REWRITE			X
	START	X		X
	WRITE		X	
Random	DELETE			X
	READ	X		X
	REWRITE			X
	START			
	WRITE		X	X
Dynamic	DELETE			X
	READ	X		X
	READ NEXT	X		X
	REWRITE			X
	START	X		X
	WRITE		X	X

6.7.2.1 Access Modes - As discussed earlier in this chapter, relative file organization arranges the records in a file into numbered record positions, and assigns each record position a number starting with 1 at the first position.

Each record position in a relative file does not have to contain a valid record. Thus, if record positions 1 and 3 do not contain valid records, in a file containing six record positions the first existing record in the file occupies record position 2 and the next record occupies position 4, and so on.



The ACCESS MODE clause in the File-Control paragraph dictates which of the three access modes may be used on that file.

When the ACCESS MODE clause specifies SEQUENTIAL, the I/O statements must refer to the records in the file sequentially, starting (after opening) with the first record and stepping through with each reference to the end of the file. The I/O statements ignore record positions that do not contain valid records.

When the ACCESS MODE clause specifies RANDOM, the I/O statements refer to the records in the file by record position. Thus, the statements may refer to record positions that do not contain valid records. The program must specify the desired record position number by placing a value in the file's relative key. If an I/O statement refers to a record position with the relative key, and that record position does not exist (either because the position does not contain a record or because it is beyond the end of the file), the INVALID KEY imperative statement may be executed depending on the particular I/O statement used. (The INVALID KEY imperative statement is explained with each of the relative I/O statements in this section.)

When the ACCESS MODE clause specifies DYNAMIC, the I/O statements may refer to the records in the file either sequentially or randomly. The OTS determines which access method to use from the OPEN mode (INPUT, OUTPUT, or I-O) and the form of the I/O statement. For example, if the statement READ ...INVALID KEY is to access an open input file dynamically, the OTS uses the relative key for random access; if the statement READ NEXT ... is to access an open input file dynamically, the OTS sequentially accesses the next existing record.

The following sections (6.7.2.2 through 6.7.2.8) on using the I/O statements themselves contain additional information about access modes.

6.7.2.2 Opening Relative Files - The OPEN statement for a relative file makes an INPUT, OUTPUT, or I-O mode file available so the COBOL program can access the records in the file sequentially, randomly, or dynamically.

For example, the following sample OPEN statement opens the file named ARTICHOKE for input, and sets ARTICHOKE's current record pointer to

zero. The program containing this statement could, after executing the statement, access ARTICHOKE with READ and START statements in the sequential access mode, READ statements in the random access mode, or READ, READ NEXT, and START statements in the dynamic access mode.

```
OPEN INPUT ARTICHOKE.
```

If an OPEN statement encounters any of the following conditions, it will fail to open the file:

- The input or input/output file specified does not exist;
- The input or input/output file specified is not a relative file;
- Contiguous space, required by the output file specified, is not available;
- The file is either already open or closed with lock.

The OPEN statement sets the current record pointer for the file to zero.

If the file is being accessed randomly or dynamically, the program must maintain a correct value in the relative key. If the file is being accessed sequentially, the OTS ignores the value of the relative key, but updates it to contain the position number of the record being accessed.

6.7.2.3 Reading Relative Files - When applied to a file being accessed sequentially, the READ statement makes the next logical record of an open file available to the program.

When applied to a file being accessed randomly, the READ statement selects a specified record from an open file and makes it available to the program. The value of the relative key for the file identifies the specific record.

When applied to a file being accessed dynamically, the READ statement has two formats so that it can either select the next logical record (sequentially) or select a specified record (randomly) and make it available to the program. The READ NEXT statement takes the number in the current record pointer and finds the next present record. The following sample READ statement reads the file named ARTICHOKE sequentially and, when it exhausts the file, causes program control to transfer to the subroutine named FILEOUT:

```
READ ARTICHOKE NEXT RECORD
    AT END GO TO FILEOUT.
```

For further information concerning the mechanics of the READ NEXT statement, see sub-section 6.7.2.7 Specifying the Next Record to be Read.

The READ with key takes the value in the relative key, moves it to the current record pointer, and reads the record being pointed to. The following READ (with key) statement reads the file named ARTICHOKE randomly, selecting records through the value in the file's relative key; if the relative key supplies a value that does not contain a valid record, the statement causes program control to transfer to the subroutine named NO-REC.

```
READ ARTICHOKE RECORD
     INVALID KEY GO TO NO-REC.
```

6.7.2.4 Rewriting Records into a Relative File - The REWRITE statement places a record back into its file on disk or magnetic tape. The following sample statement writes the record, BREAKERS, back into its file.

```
REWRITE BREAKERS.
```

If the file is open in the sequential access mode, the statement rewrites the record just successfully read. If the file is open in either the random or dynamic access mode, the statement rewrites the record to the record position specified by the relative key.

The number of bytes in the record specified by the REWRITE statement must equal the number of bytes in the record being replaced. If not, the system displays an error message.

6.7.2.5 Writing Records in a Relative File - The WRITE statement releases a logical record to an output or input-output file. The following sample WRITE statement releases the record BREAKERS to the device assigned to that record's file. If the record already exists, or the relative key points to an area not in the file, program control transfers to the subroutine, WRITE-ERR.

```
WRITE BREAKERS
     INVALID KEY GO TO WRITE-ERR.
```

The WRITE statement releases records to files that are open in either the OUTPUT or INPUT-OUTPUT mode. The following text discusses the two modes separately:

- OUTPUT Mode - The WRITE statement's only function with output files is to place entirely new records into the file. If more space is required for new record positions, the file control system automatically extends the file size, regardless of the access mode being employed.

- INPUT-OUTPUT Mode - The statement's function with input-output files is to place records in record positions that already exist and are empty. The file control system does not extend the file for new record positions; thus, the

statement must specify an existent empty record position. The length of the records must be an even number of bytes and must not exceed the maximum length record specified for the file when it was created.

The relative WRITE statement creates only storage files since print-files are sequential files. The following WRITE statement produces a storage file of the records named BREAKERS:

```
WRITE BREAKERS.
```

6.7.2.6 Deleting Records From a Relative File - The DELETE statement logically removes an existing record from a relative file. After a DELETE statement has successfully removed a record from a file, that record can no longer be accessed.

The DELETE statement applies only to relative files.

If the file is open in the sequential access mode, the statement removes the record just successfully read. For example, the following sample statement removes the record just read from the file named ARTICHOKE:

```
DELETE ARTICHOKE RECORD.
```

If the file is open in either the random or dynamic access mode, the statement removes the record from the record position specified by the relative key. For example, the following sample statement deletes the record specified by the relative key from the file named ARTICHOKE; if the relative key supplies a value that does not contain a valid record, the statement transfers control to the subroutine named NEW-KEY.

```
DELETE ARTICHOKE RECORD
    INVALID KEY GO TO NEW-KEY.
```

6.7.2.7 Specifying the Next Record to be Read - The START statement specifies which record in an input or input-output file will be the next one to be referenced sequentially.

The statement contains a conditional phrase that sets the current record pointer to one less than the number of the first record in the file whose position number satisfies the comparison. A READ NEXT statement should follow the START statement since the READ NEXT statement reads the next record from the one being pointed to by the current record pointer.

If the data area, SOMETHING, in the following example contains a 30 and position 33 in the file contains the next present record, the START statement sets the current record pointer to one less than 33 (32). The READ NEXT statement would then find the next present record,

which we know is 33.

```
WORKING-STORAGE SECTION.
77 SOMETHING PIC S99 VALUE 30.
77 ARTKEY PIC S99.
.
.
.
RD-SET.   MOVE SOMETHING TO ARTKEY.
          START ARTICHOKE
                KEY IS GREATER THAN ARTKEY
                INVALID KEY GO TO NEWKEY.

IN1.      READ ARTICHOKE NEXT RECORD
                AT END GO TO FILEOUT.
```

The value of the RELATIVE KEY data item specified in the statement (ARTKEY in the preceding example) together with the conditional phrase specified in the statement (IS GREATER THAN in the preceding example) determines which record in the file will be accessed by the READ NEXT statement.

The START statement uses the value in the RELATIVE KEY data item to set the current record pointer. If record positions 30 and 33 contain valid records, the START statement would set the current record pointer and RELATIVE KEY data item as follows:

1. If the conditional phrase specifies KEY IS GREATER THAN 30, the statement sets both the current record pointer and the RELATIVE KEY data item to 32.

2. If the conditional phrase specifies KEY IS EQUAL TO 30 or NOT LESS THAN 30, the statement sets the current record pointer and the RELATIVE KEY data item to 29.

The READ NEXT statement takes the number in the current record pointer and finds the next present record from that number. (If the pointer contains a 30 and the next present record is in position 33, it finds record number 33). The READ NEXT statement gets that record and places its record position number (33) into the current record pointer and the relative key.

A subsequent READ NEXT takes the number in the current record pointer, which is now 33 in our example, and finds the next present record. It fetches that record and places its record position number in the current record pointer and relative key.

6.7.2.8 Closing Relative Files - The CLOSE statement terminates processing on the file referred to in the statement. The following sample CLOSE statement terminates processing on the file named ARTICHOKE:

```
CLOSE ARTICHOKE.
```

When the statement closes a file, no other I/O operation can access that file until another OPEN statement opens the file.

If the statement specifies the LOCK option, the program cannot open the file again in that run.

If a SAME AREA clause contains the name of the file just closed, the program may open one of the other files named in the clause.

6.8 COMMUNICATING WITH THE PROGRAM

The ACCEPT and DISPLAY statements allow low-volume, terminal-oriented interaction between a COBOL program and the user of the program.

While these statements are primarily intended for use with keyboard devices, PDP-11 COBOL allows the ACCEPT statement to accept cards from a card reader, and the DISPLAY statement to display data on a line printer. The following two subsections (6.8.1 and 6.8.2) discuss these capabilities in greater detail.

6.8.1 Using the ACCEPT Statement

The ACCEPT statement makes small amounts of data available to the specified data item.

Consider the following example; it causes the OTS to transfer data from the device identified by the mnemonic-name, OPERATOR, to the area represented by the identifier, COMM-AREA.

```
ACCEPT COMM-AREA FROM OPERATOR.
```

OPERATOR must be a mnemonic-name specified for a device in the Special-Names paragraph (in this example, possibly the console). The area represented by the identifier COMM-AREA receives the data requested without any editing. The OTS transfers enough bytes to fill the identifier's area, and transfers the bytes as it finds them. The size of COMM-AREA in this example dictates the number of bytes being transferred.

If the device contains more data than there are bytes in COMM-AREA, the OTS truncates the data.

- If the length of the identifier exceeds 80 bytes, the OTS performs one or more additional transfers of data until it either fills the identifier or transfers less than 80 bytes.

- If the length of the identifier is less than or equal to 80 bytes and the length of the data is less than the identifier on a teletype or cards, the OTS pads the identifier with blank characters.

The ACCEPT statement has a second format that allows it to retrieve the current DAY, DATE, or TIME from the system and store it in the specified identifier. (DAY, DATE, and TIME are reserved words that the user does not define. The user must define identifiers into which to accept the values in DAY, DATE, or TIME.) The following sample statement places the current date in the identifier, GREENWICH:

```
ACCEPT GREENWICH FROM DATE.
```

The RSX-11D system treats DAY and DATE as equivalent. (The system currently has no facility for specifying day of the year -- 1 through 365.) It provides the date as follows (YY is the year; MM is the month; DD is the day):

```
DATE -- YYMMDD
```

If the date were July 4, 1976, the system would provide GREENWICH with the number 760704.

The system provides the time as follows (HH is the hour; MM is the minutes; SS is the seconds; CC is the hundredths of a second):

```
TIME -- HHMMSSCC
```

If the time were 20 seconds after 5:15 PM, the system (which has a 24-hour clock) would provide the numbers 17152000. (Since the PDP-11 clock has no hundredths of a second capability, the system places zeroes in the last two positions.)

The identifier receives the data according to the rules for the MOVE statement. Chapters 3 and 4 in this guide discuss the MOVE statement as applied to non-numeric fields (Chapter 3) and numeric fields (Chapter 4).

6.8.2 Using the DISPLAY Statement

The DISPLAY statement transfers small amounts of data from the specified data item or literal to the specified device.

Consider the following example; it causes the OTS to transfer data from the area represented by the identifier, COMM-AREA, to the device with the mnemonic-name, OPERATOR:

```
DISPLAY COMM-AREA UPON OPERATOR.
```

OPERATOR must be a mnemonic-name specified for a device in the Special-Names paragraph (possibly the console in this example). The area represented by the identifier, COMM-AREA, contains the data being transferred.

The DISPLAY statement causes the OTS to transfer a stream of bytes to the device specified in the UPON phrase if it is present (OPERATOR in the preceding example). If the UPON phrase is not present, the OTS transfers the data to the user's console.

The OTS first transfers a line feed, then transfers all of the bytes in all of the identifiers or literals in the DISPLAY statement. The size of COMM-AREA, in this example, dictates the number of bytes being transferred.

The system does not convert COMP items from binary to ASCII; it simply transfers them as they exist in storage.

If a single DISPLAY statement must transfer large amounts of data, that data must contain appropriate vertical and horizontal form control characters. If the data being transferred does not contain form control characters and the length of the data stream exceeds the device's single line capacity, the excess characters will all print in the last position (overprinting each other).

The following table contains several of the terminal form control characters:

Form Control Characters

OCTAL CODE	CONTROL CHARACTER	FUNCTION
007	BEL (CTRL G)	Bell ringer
011	HT (CTRL I)	Horizontal tab
012	LF (CTRL J)	Line feed or line space (new line)
013	VT (CTRL K)	Vertical tab
014	FF (CTRL L)	Form feed to head of form
015	CR (CTRL M)	Carriage return

When it has transferred all of the data from all of the items listed in the statement, the OTS automatically appends a carriage return character onto the data. The WITH NO ADVANCING phrase suppresses this appending operation.

NOTE

The WITH NO ADVANCING phrase is not ANS-74 standard.

6.9 FILE COMPATIBILITIES -- COBOL, FORTRAN, AND MACRO

6.9.1 Writing Files for FORTRAN or MACRO

PDP-11 COBOL writes files that can be read by FORTRAN or MACRO. However, when creating a file that is to be read by another language, adhere to the following restrictions:

- Ensure that the file has sequential file organization.

- Ensure that the file is not a COBOL print file (no LINAGE or APPLY PRINT-CONTROL clauses are applicable to the file). Printer control is handled differently by each of the three languages.

- Do not use the ADVANCING option in WRITE statements when creating the file.

The file may contain fixed-length or variable-length records, and the records may contain both ASCII and binary data.

The program that reads the COBOL-created file (either FORTRAN or MACRO) must contain descriptions of the file and its records; further, the reading program must be able to discern whether the file has fixed-length or variable-length records.

6.9.2 Reading Files Written by FORTRAN or MACRO

PDP-11 COBOL reads files that were written by FORTRAN or MACRO. However, before reading a file that was written by another language, be certain that the file meets the following restrictions:

- Ensure that the file is an ASCII file.

- Ensure that the file has sequential file organization.

- Ensure that the file does not have a carriage control attribute (the FORTRAN carriage control file attribute must not be set).

FORTRAN meets all of these restrictions when it writes ASCII data with unformatted WRITE statements, as shown below (formatted WRITE statements cause the carriage control attribute to be set for the file):

```
WRITE (u) list
     or
WRITE (u)
```

To process a file that meets all of the restrictions, simply describe the file as shown below:

- If the file contains fixed-length records of a known size, declare the file to be sequentially organized and describe the dimensions of the record in a standard COBOL record description clause.

- If the file contains variable-length records, or if it is not known what type of records the file contains, declare the file to be sequentially organized, with variable-length records, and provide the file with a maximum size record area.

6.10 PROCESSING I/O ERRORS

The USE statement provides COBOL programs with a way to process I/O errors. It allows the program to specify possible recovery steps following the I/O handling procedures performed by the software.

When a COBOL program contains a USE procedure and an I/O error occurs, the OTS and file control system execute their standard I/O error handling procedures and then transfer control to the procedure following the USE statement. (For further information concerning run-time I/O errors, see section 6.11 in this chapter.)

Consider the following sample coding. When either THOREAU or ARTICHOKE causes an I/O error, the OTS executes its standard I/O error procedures and then transfers control to the paragraph (or paragraphs) that follow the USE statement.

```
PROCEDURE DIVISION.
    DECLARATIVES.
    REPAIR SECTION.
        USE AFTER STANDARD ERROR PROCEDURE
                ON THOREAU ARTICHOKE.
        IF ...
```

The paragraphs following the USE statement may contain any valid COBOL statement, except for the following:

1. Those statements that refer to a procedure outside of the USE procedure being executed. (Any attempt to transfer control out of the USE procedure section causes the OTS to abort the program.)

2. Those statements that would cause the USE procedure being executed to be invoked again. (Recursive USE procedures cause the OTS to abort the program.)

   The OTS executes USE procedures in the same manner that it executes PERFORM ranges in Procedure Division coding. Therefore, paragraphs with the USE procedure section should follow all rules specified for paragraphs within PERFORM ranges. For further information on PERFORM ranges, see Use of the PERFORM Statement in Chapter 8 of this guide.)

   If a status key is declared for the file in error, the OTS makes all status information available for processing in the USE procedure.

6.11 RUNTIME FILE I/O ERROR PROCEDURES

When it meets an error condition during I/O operations, the OTS follows the procedure shown below:

1. If the file status key for the file is present, the OTS sets it to the appropriate code for the error condition. (See Table 6-6 for sequential file status keys, or Table 6-7 for relative file status keys.)

2. If an AT END or INVALID KEY imperative condition is specified for the I/O operation, the OTS takes the path indicated by the imperative statement. The file system performs no other processing in the file for the current statement.

3. If a USE procedure is declared for the file, the OTS performs the USE procedure section, then returns control to the program. The file system performs no further processing for this file.

   If no USE procedure is declared for the file, a fatal error condition exists; the OTS aborts the program and displays the following I/O error message:

```
"CBL -- W00037 FILE:  NN... NO USE PROCEDURE FOR
                I/O ERROR"
"CBL -- IO ERROR NUMBER - XX"
```

   NN represents the name of the file:

   XX represents the file control service error code. (See Table 6-8 for these error codes.)

The following tables show various error numbers and error codes that identify error conditions and messages. The error codes in Tables 6-6 and 6-7 are accessible to the user's program though the declaration and use of the FILE STATUS key in the program. The error codes in Table 6-8 are returned to the OTS (not the user's program) and represent error conditions detected by the File Control System.

The error message numbers in Section 6-12 are merely identifying numbers for the messages and appear at the user terminal in the following form:

```
"CBL --X000nn -, message ... "
```

X may be any one of the following:

```
I - Information message
W - Warning error
F - Fatal error
```

nn is the message number.

A table of status key codes follows. The left-hand digit of the status key code is status key 1, and the right-hand digit is status key 2.

Table 6-6
Sequential I/O File Status Key Values

Status Key Code	Meaning
00	No further information (successful)
10	End-Of-File indicator detected
30	Permanent error
34	Permanent error (boundary error)
93	REWRITE attempted without prior READ
95	Allocation failure (no file space on device)
96	No buffer space (program tried to open a file that is sharing buffer space (SAME AREA) with another file)
97	No such file (the file named in an OPEN statement was not found)

Table 6-7
Relative I/O File Status Key Values (ASCII)

Status Key Code	Meaning
00	No further information (successful)
10	End-Of-File indicator detected
22	Duplicate key error
23	No such record error
24	Permanent error (boundary error)
93	REWRITE or DELETE attempted without prior READ
95	Allocation failure (no file space on device)
96	No buffer space (program tried to open a file that is sharing buffer space (SAME AREA) with another file)
97	No such file (the file named in an OPEN statement was not found)

Any of the following I/O error conditions could occur during COBOL program execution. The codes appear in a COBOL message in the form shown below:

"CBL -- IO ERROR NUMBER - nn"

(nn represents the 2-digit file control service error code).

Table 6-8
File Control Service (FCS) Error Codes

Code	Meaning
11	End of volume detected
12	Write attempted to a locked unit
24	Device full (allocation failure)
26	No such file
27	File locked from write access
38	File not properly closed
39	No buffer space available for file
40	Record too long on READ
46	Record number too large
50	Bad directory file
53	File already open
54	Bad filename
55	Bad device name

6.12 RUN-TIME ERROR MESSAGES

NUMBER	MESSAGE	MEANING
1		(Not used)
6	FILE: NN... ATTEMPT TO OPEN 2 'SAME AREA' FILES SIMULTANEOUSLY	The program tried to open a file that uses the same buffer area of another file that is still open. (NN... represents the filename.)
7	FILE: NN... NOT OPEN	The program attempted to perform an I/O operation on a file that was not open. (NN... represents the filename.)
10	FILE: NN... ALREADY OPEN	The program attempted to open a file that was already open. (NN... represents the filename.)
11	SUBSCRIPT TOO BIG	A subscript value used in a subscripted data item reference has exceeded the upper bounds of the number of items in the table.
12	TOO MANY ACTIVE PERFORMS	The number of nested active PERFORM statements being executed by the program has exceeded 30 levels.
13	SUBJECT OF ALTER NOT "GO TO"	An ALTER statement attempted to alter the path of a statement that is not a "GO TO" statement.
14	STOP, CR TO CONTINUE	The program executed a STOP statement. The OTS waits indefinitely. To continue, type carriage return.

15	STOP RUN	The program executed a STOP RUN statement. The program stops all activity and closes all open files.
16	SUBSCRIPT TOO SMALL	The subscript value of a data item is less than or equal to zero.
17	UNDEFINED PROCEDURE REFERENCE	Some malfunction in the COBOL compiler or OTS has caused a fatal error.
20	FILE: NN... OPTIONAL FILE MOUNTED? Y OR N?	The OTS is asking the operator to specify whether the file NN... is available to the running program. (NN... represents the filename.) Type a Y for yes, or N (or some other character) for no.
21	FILE: NN... NOT ALLOCATED	The program requested that the OTS open file NN... with some number of contiguous blocks. The operating system cannot provide the number of blocks requested. This is a warning message; however, the file is not opened. (NN... represents the filename.)
22	INDEX VALUE TOO SMALL OR TOO LARGE AT SOURCE LINE NNNNN	A value for an index name is being used in a SET statement that is outside the bounds of the table. (NNNNN represents the source program's page-line number.)
23		(Not used)
24	WRITE ERROR IN DISPLAY	A DISPLAY statement encountered a bad device or a record length of more than 132 characters.
25		(Not used)
26		(Not used)

27	OPEN/CLOSE ERROR IN ACCEPT DISPLAY	The program's attempt to open a logical unit and communicate with the specified device has failed. The device is not in the system, or the device handler is not installed in the operating system. The OTS opens an input-only device for DISPLAY or an output-only device for ACCEPT.
30	ACCEPT-INPUT TOO LONG	A single ACCEPT statement has attempted to read more than 80 characters. The OTS currently imposes a limit of 80 characters on the ACCEPT statement.
31	FILE: NN...OPEN ERROR IO ERROR NUMBER - XX	The program attempted to open file NN... but the open failed. The file control services error code specifies the kind of error. (See Table 6-8 for the FCS error codes.) (NN.. represents the filename. XX represents the error code.)
32	FILE: NN...CLOSE ERROR IO ERROR NUMBER- XX	The program attempted to close file NN... but the close operation failed. The FCS error code specifies the kind of error. (See Table 6-8 for the FCS error codes.) (NN... represents the filename. XX represents the error code.)
33	FILE: NN... NOT OPEN	The program attempted to close file NN... but file NN... is not open. (NN... represents the filename.)
34	FILE: NN... INVALID LINAGE	The LINAGE clause specified a page body size that has been calculated to be zero. (NN... represents the filename.)
35		(Not used)
36	FILE: NN... REWRITE/ DELETE NOT LEGAL WITHOUT PRIOR READ	The program requested a REWRITE or a DELETE operation on a sequential file and the last I/O operation in the file was not a READ.
37	FILE: NN... NO USE PROCEDURE FOR I/O ERROR. IO ERROR NUMBER - XX	The OTS detected an I/O error for file NN... and no USE procedure is specified for the file (explicitly or implicitly). The FCS error code XX, specifies the kind of error.

		(See Table 6-8 for the FCS error codes.) This message results from a fatal error; the OTS executes a STOP RUN and closes all open files.
40	FILE: NN... LOCKED	The program previously closed the file with lock during this program execution. (NN... represents the filename.)
41	FILE: NN... INVALID OPERATION	The program attempted to issue one of the following I/O statements on a file open in an incompatible mode: • A READ on a file open for output; • A WRITE on a file open for input or I-O; • A REWRITE or DELETE on a file open for input or output.
42	FATAL ERROR ON SOURCE LINE NNNNNN	The OTS is executing an object program that has fatal compilation errors on the indicated source line. This message appears only during debugging of a PDP-11 COBOL program. (Fatal compilation errors usually suppress object program generation.) (NNNNNN represents the source program's page-line number.)

CHAPTER 7

DEVICES

7.1 INTRODUCTION

Any device that is supported by the RSX-11D operating system can process PDP-11 COBOL files. Thus, COBOL input and output files can use any of the following devices during both compilation and execution:

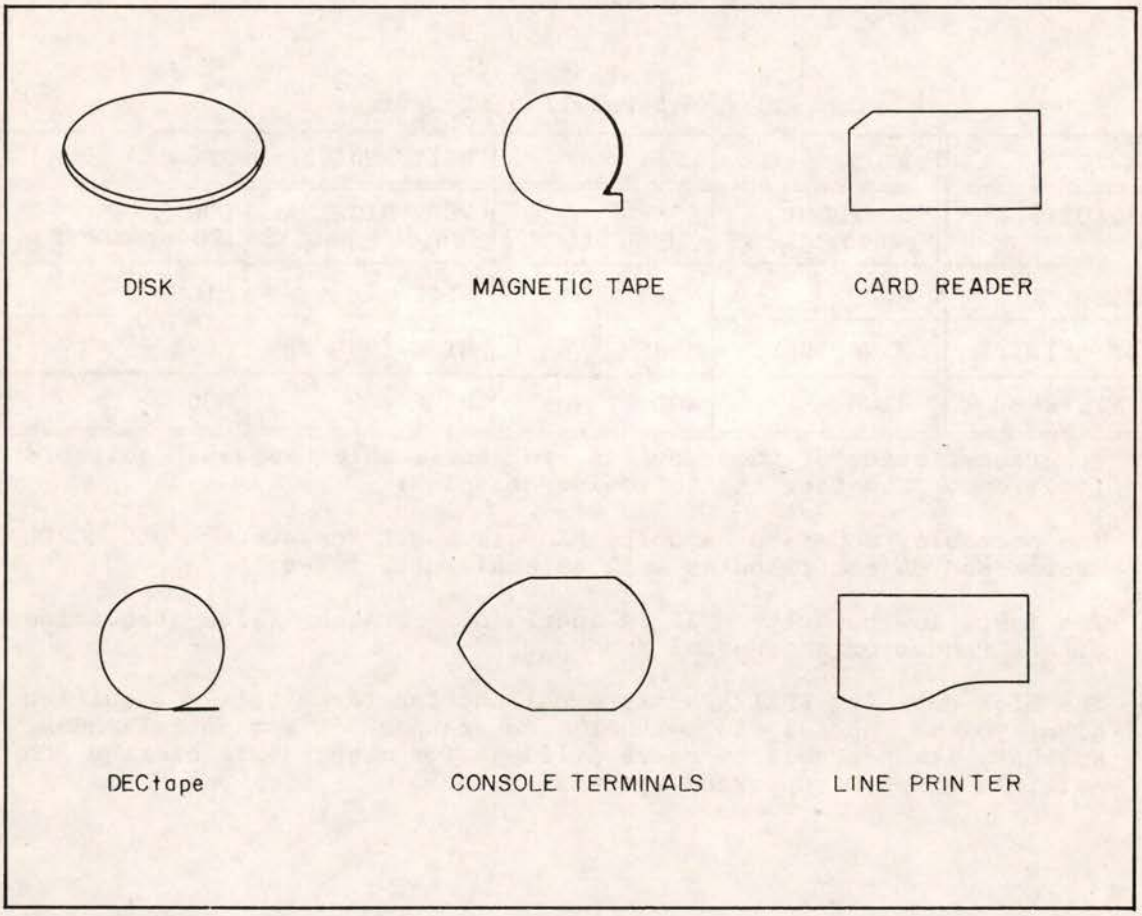


Figure 7-1
COBOL I/O Devices

Some devices, however, are better suited to certain uses than others. For example, since PDP-11 COBOL is a disk-oriented system, the disk provides COBOL files with the best performance and reliability. On the other hand, COBOL files on magnetic tape are limited to sequential organization, and the file system does not support multi-reel magnetic tape files.

This chapter discusses the devices that are available and how to use them to best advantage.

7.2 DISK

The primary means for storage and processing of PDP-11 COBOL files is a disk. RSX-11D supports several disk units, including RK05, RF11, RP11/RP03, and RS04. Each device has its own file-handling characteristics, and differs with respect to capacity, speed, and portability. The following table compares these characteristics. The values, low, mod (moderate), and high, describe the relative characteristics of the devices in the table. The efficiency characteristic combines the values of capacity, speed, and portability for that unit into a single value.

Table 7-1
Comparison of PDP-11 Disk Devices

DEVICE	RK05	RF11	RP11/RP03	RS04
CAPACITY	LOW/MOD (4800 BLOCKS)	LOW (1024 BLOCKS)	VERY HIGH (80,000 BLOCKS)	LOW (2048 BLOCKS)
SPEED	MOD	HIGH	HIGH	HIGH
PORTABILITY	HIGH (EASY)	NONE	HIGH (BULKY)	NONE
EFFICIENCY	HIGH	MOD	HIGH	MOD

The characteristics of the devices in the table suggest suitable applications. Consider the following examples:

- The portable, moderate capacity RK05 is ideal for storage of COBOL source and object files as well as small data files.
- The fast, low capacity RF11 is ideal for scratch files requiring either random or sequential access.
- The high-capacity RP11/RP03 is excellent for large files requiring high volume access in either the random or sequential modes. Further, its portability makes it ideal for master file storage on multiple RSX-11D supported systems.

7.3 DECTAPE

DECtape provides a convenient, inexpensive, and reliable medium for storing small files, such as COBOL source and object files.

DECtape's small size makes it convenient for storage and transportation, and its 512-block capacity makes it good for a number of small files. It is too slow, however, to be suitable for data files or storage of files that are frequently accessed.

7.4 MAGNETIC TAPE

The RSX-11D operating system supports magnetic tape files; however, they are not supported to the extent required for full COBOL usage. RSX-11D does not now support multi-volume files.

Therefore, it is not recommended that PDP-11 COBOL be used to process magnetic tape files until this feature is implemented. (Future versions of the RSX-11D operating system are expected to support multi-volume files.)

To process magnetic tape files, as they are now supported, adhere to the restrictions noted below:

- A program can WRITE a magnetic tape file if it is the only file to be written on the tape and no other files are already on the tape.
- A program can READ a magnetic tape file if the file is fully contained on the tape and, when the tape is a multi-file tape, each file has a unique name.

7.5 CARD READER AND LINE PRINTER

COBOL programs can use both the card reader and the line printer as I/O devices.

If these devices have been assigned logical names in the Special-Names paragraph of the Environment Division, the ACCEPT and DISPLAY statements can access them. For example, consider the following coding:

```
        ENVIRONMENT DIVISION.

          ...

        SPECIAL-NAMES
            CARD-READER IS CARDS
            LINE-PRINTER IS LOG.

          ...

        PROCEDURE DIVISION.

          ...

            ACCEPT INREC FROM CARDS.

          ...

            DISPLAY PRINTREC UPON LOG.
          ...
```

Figure 7-2
Assigning Logical Names to the
Card Reader and Line-Printer

If file-names have been assigned to these devices in the SELECT clause of the File-Control paragraph, the READ and WRITE statements can access them for I/O files. For example, consider the following coding:

```
...

FILE-CONTROL.
    SELECT INFILE ASSIGN TO "CR:".
    SELECT OUTFILE ASSIGN TO "LP:".

...
FD OUTFILE
    DATA RECORD IS OUTREC.

...

PROCEDURE DIVISION.

...

    READ INFILE.

...

    WRITE OUTREC.
```

Figure 7-3
Assigning the Card Reader and
Line-Printer to Files

CHAPTER 8

GOOD PROGRAMMING PRACTICES

8.1 FORMATTING THE SOURCE PROGRAM

Since most COBOL programs are unusually long, the programmer needs techniques that will help him to simplify and improve the readability of his COBOL programs. The guidelines in this chapter, if followed, will help produce source programs that are easy to read and maintain.

Before considering these guidelines, consider the reference formats that are available with PDP-11 COBOL:

1. the Conventional (ANS) format, and

2. the Terminal format.

Although the Conventional format produces ANS compatible programs, it also produces source printouts that are somewhat more cluttered than those produced by the Terminal format. These guidelines, therefore, recommend the use of that format and all of the following suggestions and examples assume the use of that format. Besides the obvious advantage of an uncluttered printout, the Terminal format has other programming advantages:

1. it requires less storage area;

2. it requires no line numbers;

3. its statements may be aligned with tab characters.

Further, whenever required, the REFORMAT utility program will convert Terminal format programs to the Conventional format. (The REFORMAT utility program is discussed in Chapter 11.)

The following suggestions should help to further simplify even the most complicated source programs.

1. Begin division, section, and paragraph names in column 1. Although these names may start anywhere in Area A, aligning them in column 1 produces a much more readable listing. When required, place the * and - in column 1. (Column 1 then becomes column 0.)

2. Insert a blank line, or one or more comment lines (describing the purpose of the file) before each SELECT statement in the FILE-CONTROL paragraph. Place the phrases of the SELECT statement on separate lines and begin each of them in column 5 (use the tab character to skip over Area A). Consider the following illustration of a typical SELECT statement:

```
AREA A          | AREA B
                |
1 . . .         | 5 . . . . . .
                | SELECT MASTER-FILE
                | ASSIGN TO "DK1:"
                | ORGANIZATION IS RELATIVE
                | ACCESS IS SEQUENTIAL.
```

3. Place the phrases of the file description statement on separate lines and begin each of them in column 5. (Use the tab to skip over Area A.) Consider the following illustration of a typical file description entry:

```
AREA A          | AREA B
                |
1 . . .         | 5 . . . . . .
FD              | MASTER-FILE
                | LABEL RECORDS ARE STANDARD
                | VALUE OF ID IS MASTER-FILE-NAME
                | DATA RECORD IS MASTER-RECORD.
```

4. In both the File and Working-Storage sections, begin all 01 level items in column 1.

   Indent, by four columns, all subordinate items with higher-valued level numbers. (For example, if the item that is subordinate to a 01-level record description is 05, begin the record description level number in column 1 and the 05 level number in column 5). Use the tab character for the first indentation, a tab and four spaces for the second, two tabs for the third, etc. When indented in this manner, the listing will show, clearly and neatly, the hierarchical relationships of all of the data-names in the program as well as their level number values.

   Increment level numbers by 5; then later, if it becomes necessary to insert additional group items, they may be inserted without having to change the level numbers of all items that are subordinate to that group.

   If desired, write the level numbers as single digits (such as 1 instead of 01.

Use level number 01 instead of 77 in the Working-Storage Section. (77, as a level number has the same meaning as 01, and 77 may eventually be omitted from the COBOL standard.)

Since all elementary items, except for index data items, require PICTURE clauses, these clauses fill a good part of the source program listing. However, the PICTURE clause itself may be simplified to enhance the listing's readability as follows:

a. use PIC as an abbreviation for PICTURE,

b. omit the noiseword IS, and

c. align the PIC clauses on successive lines. (Use the tab character to align the clauses.)

5. Put all paragraph name declarations in the Procedure Division on lines separate from the statements in the paragraph. This not only makes the program more readable, it also makes modification of the first statement in the paragraph easy.

6. Follow all imperative statements with a period, making them one-statement sentences. Place only one statement on a line. In addition to making the lines shorter and more readable, this will prove quite helpful when debugging the program. For example, if the program contains a coding error, it will be on one line and therefore easier to modify without affecting the other portions of the sentence; further, the diagnostic messages will refer to the correct line and their meanings will be clear.

   Since left-aligned statements in any program enhance the readability of that program, develop the habit of starting all COBOL sentences in column 5. (Use the tab character to skip over Area A.) Some statements, however, should be further indented, as explained in the following paragraphs.

7. If the True path of a conditional statement contains another conditional statement or more than one imperative statement, place all statements in the True path on lines immediately following the conditional statement and indent them to show their dependence upon that statement. Consider the following illustration of an IF statement and its True path:

```
IF COMPUTED-TAX > TAX-LIMIT
   SUBTRACT TAX-LIMIT FROM COMPUTED-TAX GIVING EXCESS-TAX
   MOVE TAX-LIMIT TO COMPUTED-TAX
   ADD EXCESS-TAX TO TOTAL-EXCESS-TAX.
```

   If the statement has an ELSE (or False) path, align the word ELSE under the preceding IF and indent all statements that are dependent on the ELSE statement. Thus:

```
IF condition
   true path statement
   true path statement
   . . .
ELSE
   false path statement
   false path statement.
```

Be sure to place the period after the last statement only!

Another good method for simplifying conditionals is to write only a single imperative statement in the True or False path. If the path requires more statements, place them in a separate paragraph and either PERFORM the paragraph from the path or GO to it. This technique avoids the possibility of inadvertently placing a period at the end of a statement within the path, thereby terminating it early.

When writing a GO TO ... DEPENDING statement, place each procedure name on a separate line and indent them all. Consider the readability of the following sample statement:

```
GO TO P35
      P40
      P45
      P60
      P65
      DEPENDING ON P-SWITCH.
```

8. When grouping statements into paragraphs and sections, use the following organizational ideas:

   Group together logical units of processing into a section. Select a section name that reflects the type of processing being conducted within that section (such as TAX-COMPUTATION SECTION, PRINT-LINE-FORMATTER SECTION, etc.). Follow the section name with sufficient comment lines to explain the processing that is carried out by the statements within that section.

   Make paragraph names as short and simple as possible. A numbered abbreviation of the section name often suffices. Thus the paragraph names in the TAX-COMPUTATION section might be TC10, TC20, TC30, etc. Use paragraph names sparingly, placing them only where the True and False paths of conditional statements require branch points for GO TO statements. If the temptation arises to give a paragraph a longer name in an attempt to reflect the type of processing in that paragraph, use comment lines instead. (Comment lines usually convey more information, more clearly.)

   When using simple numbered paragraph names, assign increasing numeric characters to sequential paragraphs. If the numeric portion of the names increases by 5 or 10, new ones may be inserted later without disturbing the sequence of the names.

Do not use the PERFORM verb in the form, PERFORM a THRU b. If the paragraphs a thru b must be performed, place them in a section by themselves and PERFORM the section, thus avoiding the use of the THRU option.

Place single paragraphs that are to be performed into sections and use the section name as the object of PERFORM verbs. Then, if future design changes introduce complicated conditional logic into the paragraph, requiring additional paragraph names, the PERFORM statements need not be altered.

The preceding guidelines divide the Procedure Division into modular blocks of coding. If these guidelines are used, the following additional techniques may be applied.

a. Restrict entry to all sections through the first statement of the section by use of a GO TO, a PERFORM, or a "fall through" from the preceding section;

b. Ensure that all GO TO statements refer to only section names or paragraph names that are internal to the section containing the GO TO statement.

8.2 USE OF PUNCTUATION

Avoid using the COBOL punctuation characters, comma and semicolon. They lend little to the readability of programs that have their statements neatly aligned, as discussed earlier in this chapter. Further, it is quite easy to misuse these characters, which can cause serious errors for many compilers. (Other compilers either ignore incorrect punctuation characters or flag them with warning messages.) At best, even when used correctly and in the proper places, they have no effect on the meaning of the program.

8.3 USE OF THE ALTER VERB

Avoid using the ALTER verb to change the flow of control in a program. It is impossible to test the setting of an alterable GO statement except by executing it. Also, unless explicit comments accompany an alterable GO statement, it is difficult to tell whether or not it is referenced by ALTER statements or what the possible destinations might be. All of this makes debugging programs that contain these statements quite difficult. There are two other techniques that may be used in their place:

1. If control branches one of two ways (i.e., a binary switch), write the switch as a conditional variable. Consider the following sample coding:

```
01  P-SWITCH PIC S9 COMP VALUE 0.
    88 NO-PRINT VALUE 1.
```

```
           MOVE 1 TO P-SWITCH
           .
           .
           .
           IF NO-PRINT GO TO P40.

       P40.
           MOVE 0  TO P-SWITCH.
```

2. If control branches more than two ways, use MOVE statements to place integers into a data item, and a GO TO ... DEPENDING ... statement to test the data item and branch accordingly. Consider the following sample coding:

```
       01  P-SWITCH PIC S9999 COMP VALUE 0.
           .
           .
           .
           MOVE 3 TO P-SWITCH.
           .
           .
           .
           GO TO
                PART-TIME
                PIECE-WORK
                HOURLY
                SALARIED-WEEKLY
                SALARIED-OTHER
                DEPENDING ON P-SWITCH.
       *   FALL THROUGH IS A BUG
           DISPLAY "?17".
           STOP RUN.
```

8.4 USE OF THE PERFORM VERB

The implementation of the PERFORM verb varies widely among COBOL compilers, with the differences being related to the mechanism for returning from the end of the PERFORM range.

1. Many compilers stack the return addresses on a true push-down stack, with the result that the program may perform a section recursively. Each time the program executes a PERFORM statement, it places a new return address on the stack, and each time the program exits from the perform range, it removes a return address from the stack. Since COBOL standards impose no restrictions on the use of GO statements, the program might not take the exits from the ends of the perform ranges in the exact reverse order in which it placed their return addresses on the stack; and if it does not, the flow of control will probably be incorrect.

2. Some compilers stack their return addresses on push-down stacks, but forbid multiple activation of any end of a perform range. Recursion is prohibited and the program must exit from each perform range before entering that range again. This system works fairly well, but the compilers cannot detect out-of-order returns, with the result that this system is very little improved over the system described in paragraph 1, above.

3. Other compilers associate an exit mechanism with the end of the perform range. During the establishment of this mechanism at object execution, the system examines the end of the perform range to see if the exit is still active from the execution of a previous perform. If so, the system halts execution, since this type of compiler does not allow recursion.

4. Still other compilers associate the exit mechanism with the end of the perform range, as in C, but fail to examine the exit when establishing it, to determine if it is still active. Thus, when the system executes another perform, that perform overwrites and loses the existing active exit. Although this type of compiler does not allow recursion, it makes no attempt to detect recursion.

The following example illustrates the problem:

```
Line No.

   1      A.  PERFORM B THRU D.
                .
                .
                .
          B.
   2          PERFORM E.
          C.
                .
                .
                .
   3      D.  EXIT.
                .
                .
                .
          E.
                .
                .
                .
   4          IF . . . GO TO C.
                .
                .
   5          (end of paragraph E)
```

The conditional GO TO statement on line 4 may cause control to return to paragraph C; thus, the program may never execute the exit mechanism at the end of paragraph E, that was established by the PERFORM

statement on line 2. All four types of compilers will compile this program correctly, but each will execute it differently when the condition tested on line 4 is True.

If the object program was compiled by the type of compiler discussed in paragraph 1 (above), it will pop the stack when it reaches line 3. However, since the PERFORM statement on line 2 was the last PERFORM to establish a return address on the stack, that address is at the top of the stack and not the one established by the PERFORM statement on line 1. (The program never removed the return address established by the PERFORM statement on line 2 because it never executed line 5.) This error may be difficult to detect.

If the object program was compiled by the type of compiler discussed in paragraph 2, it will also fail to return to line 1 from line 3; but, when it executes line 2 for the second time, it will detect the exit mechanism's active state for line 5 and abort execution. Program testing may not detect this error, especially if the test data did not cause the program to execute line 2 for a second time. This error will probably reveal itself later, during a production run.

If the object program was compiled by the type of compiler discussed in paragraph 3, it will always return correctly from any perform range to the point of the PERFORM that activated it. However, when it executes line 2 for the second time, it will detect the active exit mechanism for line 5 and abort execution, similar to the type of compiler discussed in paragraph 2.

If the object program was compiled by the type of compiler discussed in paragraph 4, it will always return correctly from any perform range, as above. It will allow subsequent executions of line 2 since it simply overwrites, with a new value, the active exit mechanism for line 5.

The PDP-11 COBOL compiler combines the mechanisms discussed in paragraphs 2 and 3. As the program executes each PERFORM statement, the OTS places the return address in a list. Each exit from a paragraph or section causes the OTS to interrogate the last entry in the list to determine if that exit marks the end of the most recently activated perform range. If so, and if the conditions discussed below are satisfied, it executes the return and removes that return address from the list. Thus, it cannot take exits from perform ranges out of order. (The end points of all other active perform ranges cause the flow of control to fall through to the next procedure.)

In addition, before it places the return address in the list, the OTS interrogates the existing perform range list to determine if either of the following two conditions exist: (It performs these checks in the order listed and, if either one exists, it displays an error message and terminates object program execution.)

1. Is this same PERFORM statement already active? (This check prevents recursion.)

2. Is the end point of this perform range already an active end point of another PERFORM statement? (This check prevents several active perform ranges from using the same end point.)

NOTE

The end point of a section is distinct from the end point of the last paragraph of a section. Thus, a PERFORM statement can perform a section and then, from within the section, perform the last paragraph of the section.

Since the ANS COBOL specification does not explicitly state this distinction in end points, the use of this feature should be avoided if compatibility with other COBOL compilers is of concern. To remove the problem and make the source program independent of varying interpretations of the standard, simply place a "null" paragraph after the "last" paragraph of the section.

8.5 USE OF LEVEL 88 CONDITION-NAMES

Condition-names provide a convenient method for testing a value or range of values in a field. The use of condition-names makes programs easier to maintain, because it ensures a uniform method of testing fields and helps to reduce recoding when the specifications of the program change.

The following example illustrates the use of condition-names and shows the advantages inherent in their use.

Suppose the records of a file each describe a student in an educational institution (or an employee in a corporation). Some of the records contain categories of information which are not present in other records. A "code" field, which contains a digit or letter, indicates the presence (or type) of some categories; while a special value in the information itself (such as a numeric value being zero, negative, or maximum) indicates the presence of other categories. The processing of such a record may vary considerably depending on these indicator fields. The fields may require interrogation at various points in the program, and the interrogation may require more than a simple relation test.

Consider a "code" field that holds one of seven values, coded as a mnemonic character. For example, S,1,2,3,4,G,P might be seven values that indicate student categories of Special, 1st year, 2nd year, 3rd year, 4th year, Graduate, and Postgraduate. The field is described as follows:

```
05 STUDENT-CATEGORY PIC X.
```

Program logic requires certain processing for enrolled undergraduates, different processing for special students, and still different processing for all students except enrolled undergraduates. Without the aid of condition-names, statements might be written as follows to resolve this problem:

```
IF STUDENT-CATEGORY = "S" ...

IF STUDENT-CATEGORY NOT LESS THAN "1"
     IF STUDENT-CATEGORY NOT GREATER THAN "4" ...

IF STUDENT-CATEGORY EQUAL TO "G" NEXT SENTENCE
     ELSE IF STUDENT-CATEGORY EQUAL TO "P"
          NEXT SENTENCE ELSE GO TO ...
```

However, if various level 88 entries followed the STUDENT-CATEGORY description, as shown below, condition-names can simplify this coding.

```
05 STUDENT-CATEGORY PIC X.
   88 UNDERGRADUATE VALUE "1" THRU "4".
   88 SPECIAL-STUDENT VALUE "S".
   88 GRAD-STUDENT VALUE "G" "P".
   88 SENIOR VALUE "4".
   88 NON-DEGREE-STUDENT VALUE "S" "P".
```

Now, the following procedural statements can solve the problem:

```
IF SPECIAL-STUDENT ...
IF UNDERGRADUATE ...
IF GRAD-STUDENT ...
```

Procedural statements with condition-names are much easier to read and debug than those containing the complete test. For example, the procedural statements, IF UNDERGRADUATE ..., and IF STUDENT-CATEGORY NOT LESS THAN "1" IF STUDENT-CATEGORY NOT GREATER THAN "4" both accomplish the same thing, but the first statement is simpler and less confusing.

In addition, the statement, IF NOT UNDERGRADUATE ... can test the category of not being an undergraduate, which is equivalent to any one of the following statements:

```
IF NOT (STUDENT-CATEGORY NOT < "1" AND
     STUDENT-CATEGORY NOT > "4") ...
```

or

```
IF STUDENT-CATEGORY < "1" OR
   STUDENT-CATEGORY > "4" ...
```

or (in statements allowed in PDP-11 COBOL)

```
IF STUDENT-CATEGORY < "1" NEXT SENTENCE
   ELSE IF STUDENT-CATEGORY > "4" NEXT SENTENCE
      ELSE GO TO ...
```

Statements such as these are tedious to write and a frequent source of coding errors. Further, if a change creates a new student category, the recoding takes more time and is even more error prone. A careful and controlled use of condition-names forces a high degree of programming control and checkout. If the program logic does require the modification of the STUDENT-CATEGORY field, it can even be named FILLER thus removing the opportunity to shortcut the use of condition-names.

To apply condition-names, follow the description of the item to be tested with a level 88 entry. The item being tested, known as the conditional variable (STUDENT-CATEGORY in the preceding illustrations), may be either DISPLAY or COMPUTATIONAL usage, but not INDEX usage; it may also be a group item.

The compiler stores all of the values supplied by the level 88 entries in the object program exactly as written. (They are pooled with all of the literals from the Procedure Division.) A value supplied by a level 88 entry for a conditional variable of COMPUTATIONAL usage is stored in binary format to save conversion at object time. The compiler stores all other values as byte strings with the proper attributes. It does not make the level 88 entries equal to their conditional-variables in size. This means that it neither truncates nor pads (with spaces) non-numeric literals. Further, it neither truncates nor pads (with zeros) numeric literals, but stores them as written or, if converted to binary, in the minimum size COMP item that will hold the converted value. It stores signs as trailing overpunches on numeric DISPLAY literals, and removes and remembers decimal points.

Do not enter level 88 items under group items that have subordinate entries containing any of the following clauses: SYNCHRONIZED, JUSTIFIED, COMPUTATIONAL, INDEX.

CHAPTER 9

PDP-11 COBOL UTILITY PROGRAMS

9.1 COBRG (COBOL REPORT GENERATOR)

9.1.1 Introduction

The COBRG program provides a simple mechanism for producing printed reports from data files. And, although it may be tempting to design and write personally tailored report generating algorithms, it is usually easier and faster to use this utility program.

Most personally tailored programs that produce printed reports adhere to the following pattern:

1. the program reads an input file that has been sorted on any number of keys;

2. it prints the contents of certain fields from each input record on a printed report (usually the fields are in designated columns with a heading on each page to label the columns);

3. it keeps running totals in accumulators to be printed on the report (and zeroed for new summations) whenever designated fields change values, thus developing and printing subtotals at the end of each section or subsection of the report;

4. finally, at the end of the input file, it prints cumulative totals for the entire report.

An analysis of typical report generating programs would show that they contain several of the following elements:

- a description of the page headings;

- a description of the input and output files;

- a set of rules for moving information from the input records to the detail print lines;

- a set of rules for adding values into accumulators;

- a set of rules for monitoring the sort keys for value changes;
- a set of rules for constructing and printing the accumulator values.

The COBRG utility program features input specification lines which provide all of the preceding elements. COBRG uses these specification lines to produce a tailored COBOL source program. This program, when compiled and executed, generates the actual report. The following list contains a brief description of the input specification lines along with the information each provides:

- NAME Specification -- provides the program name and the description of the input and output files;
- INPUT Specification -- provides a description of the records in the input file;
- OUTPUT Specification -- provides a description of the records in the output file (detail lines and total messages);
- HEADER Specification -- provides a description of the page headers;
- BREAK Specification -- provides monitoring information for all control break fields;
- ACCUMULATOR Specification -- provides information on all fields to be added and sets up accumulators to contain the sums;
- TOTAL Specification -- provides information on accumulators that are to be moved directly to control footing print lines;
- EMIT Specification -- provides textual information for explaining the totals on control footing lines;
- LIST Specification -- provides detail line listing (output) information.

Subsequent sections of this chapter describe, in detail, these specification lines. For illustration purposes, sample entries for preparing a sample report accompany the rules for forming the lines. A description of the sample problem follows.

9.1.2 The Sample Problem

The input file contains sales information. Each record within this file contains the customer's name and location, along with the sales to that customer for a given period. A previous sort run has sorted the file on three location keys; a CITY key, a STATE key, and a REGIONAL key.

Management requires a report which lists all of their customers, shows the sales to those customers, and provides subtotals for each city, for each state, and for each region.

The input file contains 80-character records organized as follows:

```
01  INPUT-RECORD.
    05 CUSTOMER-NAME          PIC X(20).
    05 CUSTOMER-ADDRESS       PIC X(20).
    05 CITY                   PIC X (10).
    05 STATE                  PIC XX.
    05 ZIP                    PIC X(5).
    05 REGION                 PIC X.
    05 SALES                  PIC S9(10)V99.
    05 OTHER-DATA             PIC X(10).
```

The preceding sort run sorted the file on REGION as the major sort key, STATE as the secondary sort key, and CITY as the minor sort key. Thus, for example, if Region 1 is New England, Connecticut will probably be first (CT), and the cities of Bridgeport, Hartford, and Waterbury, if present, will appear in that order.(Section 9.1.6.4 shows the first 50 records of the file.)

9.1.3 The Specification Lines

Before COBRG can produce a report generating COBOL source program, it requires a file containing one or more sets of the specification lines discussed earlier. These specification lines describe the report or reports to be produced by the compiled source program. It is the responsibility of the user to ensure that they are filled out completely and accurately, as indicated in this and following sections.

Use a comma to separate each parameter in a specification line from the next and a carriage return to terminate each complete specification line.

Although the COBRG program requires most of the parameters in a specification line, some of them may be omitted. (This chapter denotes those that may be omitted and explains the circumstances for omission.) However, to omit a parameter from the middle of the line, enter a comma in its place. This comma, following the separator comma of the preceding parameter, notifies the software that a parameter is missing. If the omission occurs at the end of the line, just enter the usual carriage return (the software requires no additional missing-parameter indicator at the end of a line).

The first parameter of every line identifies the type of specification line and should be placed in column 1. The comma following this parameter is optional.

Since COBRG generates COBOL source language programs containing COBRG-generated data and procedure names, avoid using names that conflict with the generated names. Consider them as an extension of the COBOL reserved word list.

The following generated names, in addition to the standard PDP-11 COBOL reserved words, cannot be used as data names in the COBRG specification of the input and output files.

ACCUMULATOR-x-n	LEVEL-x-BREAK	PRINT-LINE
BEGIN	LEVEL-n-SW	PRINT-CH-n
FINISH	L-PRINT	PRINT-DETAIL
HEADER	OUTPUT-LINE	PRINT-n
HEADER-PAGE	PAGE-COUNT	READ-IN
IN-FILE	PL-EXIT	RESET-LEVEL-x
LEVEL-x-ACS	PL-HDR	SAVE-nn

NOTE

n may be any number from 0-9; x may be any number from 0-9 or F

9.1.3.1 NAME Specification - This specification line must be the first specification in the input sequence. It provides the name of the COBOL source program and describes the input file and the output file (the report). In fact, the NAME specification supplies parameters for setting up the IDENTIFICATION, ENVIRONMENT, and DATA Divisions of the COBOL source program. It must contain the following entries:

Parameter 1 -- Specification identification. Enter the letter N to identify this line as the NAME specification.

Parameter 2 -- Program name. Enter a unique name, up to nine characters long, to identify the COBOL source program.

Parameter 3 -- Input blocking factor (optional). Enter the blocking factor for the input data file. If this parameter is omitted or zero, the software assumes that the file is not blocked.

Parameter 4 -- Input file specification. Enter the complete RSX-11D file specification for the input file, with delimiting quotes.

Parameter 5 -- Output file specification. Enter the complete RSX-11D file specification for the output file, with delimiting quotes.

Parameter 6 -- Author (optional). Enter your name (up to 20 characters long). This parameter may be omitted.

The following entries would satisfactorily complete the NAME specification line for the sample program:

```
                N,COBRGTEST, ,"SALES","LP:",R.A.FUSCI
Parameter    }  ↑    ↑     ↑    ↑      ↑      ↑
Numbers      }  1    2     3    4      5      6
```

9.1.3.2 INPUT Specifications - The NAME specification line has already described the input file; now all of the records in the file must be described. The INPUT specification lines provide the software with the names and descriptions of the records and fields of the input file.

These specifications are simply individual lines of a complete COBOL record description; however, each line begins with an I in character position one.

Parameter 1 -- Specification identification. Enter the letter I to identify this line as an INPUT specification.

Parameter 2 -- Data description. Enter one line of a COBOL data description, including the level-number, data-name, and picture clause. If the record description already exists in a library file, enter the word COPY, followed by the name of the library file.

Since the software requires several INPUT specification lines to complete one record description, the record description (01-level) must be presented to the software first, with the field descriptions following in the same order as they appear in the record; however, the software requires only the names of those fields that are to be used in the report, all others may be described as FILLER items.

The following INPUT specifications would satisfactorily complete the record description for the input file of the sample program:

```
I01   INPUT-RECORD.
I     05   COMPANY     PIC  X(20).
I     05   FILLER      PIC  X(20).
I     05   CITY        PIC  X(10).
I     05   STATE       PIC  XX.
I     05   FILLER      PIC  X(5).
I     05   REGION      PIC  X.
I     05   SALES       PIC  S9(10)V99.
I     05   FILLER      PIC  X(10).
```

1 2

If the input file for the sample program already existed in a library file (named SALESR.LIB) which contained the actual complete record description, the following INPUT specification would satisfactorily complete the record description:

```
ICOPY SALESR.
```

1 2

COBRG removes the I and places each line in the File Section of the COBOL source program under the FD entry for the input file. Hence all rules for the formation of data descriptions in the File Section apply. (In particular, VALUE clauses are not allowed.)

COBRG does not check the contents of the INPUT specifications for correct COBOL syntax, and since it doesn't, any coding errors will not be discovered until the COBOL compiler compiles the source program.

If horizontal tab characters are included in I specifications, a warning diagnostic (probably harmless) will be issued upon subsequent compilation of the generated program.

9.1.3.3 OUTPUT Specification - The NAME specification line has already described the output file; now the detail lines of the report must be described. The OUTPUT specification lines provide the software with descriptions of the records and fields of the output file (the detail lines), and the control footing messages.

Like the INPUT specifications, the OUTPUT specifications are simply individual lines of a complete COBOL record-description, describing a detail line of the output file; except that each line begins with an O in character position one and the record-description appears in the WORKING-STORAGE Section instead of in the FILE Section. Further, the COBRG program will not accept a 01-level record-description entry. (COBRG supplies a standard 01-level and name -- 01 OUTPUT-LINE.)

Parameter 1 -- Specification identification. Enter the letter O to identify this line as an OUTPUT specification.

Parameter 2 -- Data description. Enter one line of a COBOL data description, including the level-number, data-name, and PICTURE clause. Do not enter a 01-level record-name for the first OUTPUT specification (COBRG provides a standard one, as discussed above).

If the record area is to be redefined, enter any level-number higher than 01 (ensuring, of course, that it is lower than the level-number of the fields in the record). Then, use the same level-number for the redefined record. Consider the entries below for the sample program, level 03 defines the records and 05 defines the fields. COBRG will subordinate both record areas to "01 OUTPUT-LINE."

If the record-description already exists in a library file, enter the word COPY, followed by the name of the library file.

Since the software requires several OUTPUT specification lines to complete one detail line, any record-name entry must be presented to COBRG first, with the field descriptions following in the same order as they are to appear on the report. Use FILLER items to separate the various fields. (All FILLER items will be space filled on the report.) The following OUTPUT specification entries would satisfactorily complete the description of one detail line of the report for the sample program, as well as a message line for the control footing totals. The message line, which must be described on an EMIT specification, redefines the detail record area. (EMIT specifications are discussed later in this chapter.)

```
003  DETAIL-LINE.
O    05  PRINT-COMPANY          PIC X(20).
O    05  FILLER                 PIC XX.
O    05  PRINT-CITY             PIC X(10).
O    05  FILLER                 PIC XX.
O    05  PRINT-STATE            PIC XX.
O    05  FILLER                 PIC XX.
O    05  PRINT-SALES            PIC Z,ZZZ,ZZZ,ZZZ.ZZ-.
003  CFLINE REDEFINES DETAIL-LINE.
```

```
O    05  MESS1                PIC X(20).
O    05  ACCOUNT              PIC ZZZZZ.
O    05  MESS2                PIC X(13).
O    05  FILLER               PIC X(22).
1         2
```

If a library file (named, for example, PRTLIN.LIB) already existed which contained these entries, the following OUTPUT specification would satisfactorily complete the record description:

```
OCOPY PRTLIN.
1     2
```

COBRG strips off the O and places each line in the WORKING-STORAGE Section of the COBOL source program under the 01-level entry for the output record. Hence, all rules for the formation of data descriptions in the WORKING-STORAGE Section apply.

COBRG does not check the contents of the OUTPUT specifications for correct COBOL syntax, and since it doesn't, any coding errors will not be discovered until the COBOL compiler compiles the source program.

If OUTPUT specifications contain horizontal tab characters, a warning diagnostic (probably harmless) will be issued upon subsequent compilation of the generated program.

9.1.3.4 HEADER Specifications - The HEADER specification lines provide COBRG with descriptions of the page headings.

COBRG assumes that the report is to be printed on standard printer forms, and allows 66 lines per page. It prints on only 60 of the lines and leaves three blank lines at the top and three blank lines at the bottom of each page. The first printing line (line 4) may be used as a header line and a number of blank lines may be designated to follow the header line. The remainder of the printing area is called the body of the page and is devoted to printing detail lines and lines that contain totals.

The first HEADER specification supplies header data for columns 1 through 60, and the second HEADER specification (if required) supplies header data for columns 61 through 132.

Parameter 1 -- Specification identification. Enter the letter H to identify this line as the HEADER specification.

Parameter 2 -- Page number field size. Enter the size of the field that will contain the page number in the heading line.

Parameter 3 -- Page number location. Enter the number of the starting column for the page number in the heading line.

Parameter 4 -- Blank lines. Enter the number of lines to be left blank after printing the heading line; an entry of 0 indicates no blank lines.

Parameter 5 -- Skip channel (not yet implemented). Enter the number (0-8) of the printer channel to which a skip is to be

made after printing the heading line; an entry of 0 indicates no skipping. At present, COBRG treats a non-zero entry as a 1 (top-of-page-skip).

Parameter 6 -- Header data. Enter the data to be placed in columns 1 through 60 at the top of each page of the report. Do not include commas within the header. COBRG will not accept imbedded commas since it considers them to be parameter separators.

Parameter 7 -- Page number reset (optional). Enter the break priority number at which the page number is to be reset to 1 (break levels are discussed under BREAK Specification in the next section). The page number is always 1 on the first page. If the page number is not to be reset, this parameter can be omitted.

If the information to be printed on the header line exceeds 60 columns, a second HEADER specification may be completed to supply up to 72 columns of additional header information. Since the information supplied by parameters 2, 3, 4, 5, and 7 has already been presented by the first HEADER specification, the second one requires only the identification and header data parameters:

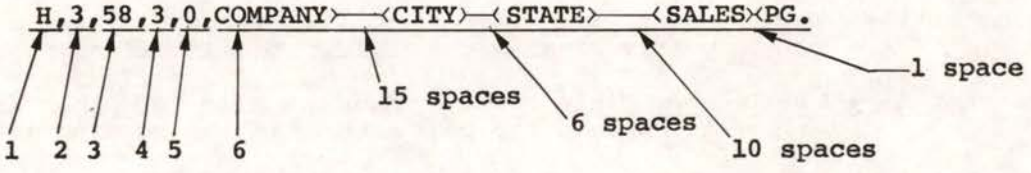
Parameter 1 -- Specification identification. Enter the letter H to identify this line as a HEADER specification.

Parameter 2 -- Header data. Enter the data to be placed in columns 61 through 132 of the header line. Do not include commas within the header.

Consider the report to be produced by the sample program. Each detail line requires 60 columns and contains the customer-name, a line number, and the sales to that customer. The first detail line after any subtotal line also contains the city and state for the detail lines that follow.

Section 9.1.6.5 provides an example of the desired output format. The HEADER specification, which is the second line of input to COBRG in the sample program, contains a description of the header line and some of the spacing information.

The following HEADER specification entries would satisfactorily complete the description of the report header for the sample program.



NOTE

The spacing between the words, in parameter 6, aligns the header with the detail lines, thus forming columns.

9.1.3.5 BREAK Specification - This specification line provides the information that COBRG requires to monitor the control break fields. Its parameters identify the fields and the levels of those fields

within the control break hierarchy; further, its parameters provide the information for any required printer manipulation for control footing.

COBRG considers the entire report as a hierarchy of levels. The lowest level is the control field that is expected to change the most often. (The lowest level control field usually corresponds to the minor sort key.) In the sample program, the lowest level control field is the CITY field.

As input records are read and detail lines are printed, the COBOL program supplied by COBRG monitors the control field associated with each level in the report's hierarchy of control fields. When it detects a change in value in a control field (a control break has occurred), it prints a control footing. Control footings contain totals and literal values with spacing requirements (to make them stand out from the detail lines) that may differ from those of the detail lines. Control footings, therefore, usually require separate descriptions which are supplied by Parameters 3 through 6 of this specification.

COBRG accepts up to 25 BREAK specifications completed as discussed below:

Parameter 1 -- Specification identification. Enter the letter B to identify this line as the BREAK specification.

Parameter 2 -- Break level. Enter a digit from 0 to 9 or F (for final). The number 0 has the lowest priority and the number 9 has the highest priority. More than one field can have the same priority number, in which case a break of the same level occurs whenever any of those fields change value.

Parameter 3 -- Blank lines, before printing. Enter a number from 0 to 9 that represents blank lines to space before printing the control footing; an entry of 0 indicates no spacing.

Parameter 4 -- Skip channel, before printing (not yet implemented). Enter a number from 0 to 8 that represents the printer control channel to which a skip is to be made before printing the control footing; an entry of 0 indicates no skipping. At present, COBRG treats a non-zero entry as a 1 (top-of-page skip).

Parameter 5 -- Blank lines, after printing. Enter a number from 0 to 8 that represents the number of blank lines to space after printing the control footing; an entry of 0 indicates no blank lines.

Parameter 6 -- Skip channel, after printing (not yet implemented). Enter a number from 0 to 8 that represents the printer control channel to which a skip is to be made after printing the control footing; an entry of 0 indicates no skipping. At present, COBRG treats a non-zero entry as a 1 (top-of-page skip).

Parameter 7 -- Field size. Enter the number of characters or digits in the break field.

Parameter 8 -- Field name. Enter the name of the break field. The field must be named in (and spelled the same as) the input record description.

The following example contains the BREAK specification for the sample program:

```
1 2 3 4 5 6 7   8
B,0,1,0,0,0,10,CITY
B,1,1,,,,2,STATE
B,2,1,,,,1,REGION
B,F,1
```

The COBOL program supplied by COBRG looks for a break on every level from high to low in every input record and when it detects a break, it "forces" breaks to occur at all lower levels, regardless of whether or not it detected breaks at those levels. Therefore, in the sample program (which has the data file arranged so that the last input record for FALMOUTH, MA is followed immediately by the record for FALMOUTH, ME), it would detect a break at the state level (level 1) between the last FALMOUTH, MA record and the first FALMOUTH, ME record and force a break to occur at the city level (level 0).

9.1.3.6 ACCUMULATOR Specification - While the object program produced by COBRG is reading input records and printing detail lines, it can also add values from the input record into accumulators and print the totals on control footing lines when control breaks occur. This specification line provides the information that COBRG requires to set up these accumulators. In addition to describing the accumulator (name and size), its parameters also specify the name of the field to be added into the accumulator and, optionally, the name of the field in the detail line print record into which the accumulator is to be moved for detail listing.

COBRG accepts up to 10 ACCUMULATOR specification lines completed as discussed below:

Parameter 1 -- Specification identification. Enter the letter A to identify this line as the ACCUMULATOR specification.

Parameter 2 -- Accumulator number. Enter a unique digit that specifies the accumulator and can be used to refer to it. Any number from 0-9 may be entered.

Parameter 3 -- Accumulator size. Enter a number that specifies the size of the accumulator in digits.

Parameter 4 -- Decimal positions. Enter a number that specifies the number of decimal places in the accumulator.

Parameter 5 -- Input field. Enter the name of the field in the input record that is to be accumulated; the name may be up to 24 characters long.

Parameter 6 -- Output field (optional). Enter the name of the field in the printer record into which the accumulator is to be moved for listing on detail lines. If the accumulator is not to be listed on detail lines, omit this parameter.

When determining the accumulator size, ensure that enough integer and decimal positions are reserved. COBRG always provides a signed field for accumulators.

The "input" field that is to be added to the accumulator (Parameter 5), although usually a field in the input record, does not have to be in the input record at all, but may, in fact, be a literal (without commas) or even a field in the output record. If this field is to appear on a detail line, it must be named in a LIST specification.

The following ACCUMULATOR specifications will set up the accumulators required by the sample program (2 are required):

```
A,0,14,2,SALES

A,1,5,0,1,LINENO
1 2 3 4 5 6
```

In the sample problem entries above, accumulator 0 contains a summation of the SALES field from the input records and accumulator 1 contains a count of the detail lines within the lowest level (the program uses this accumulator to number the detail lines); the literal 1, in Parameter 5, is the quantity added to this accumulator for each detail line in the field named LINENO. (Accumulator 0 is not printed on the detail lines.)

Accumulators are numeric fields in the COBOL source program produced by COBRG. By moving them to fields described on the OUTPUT specifications, they can be printed on the report. Further, by using the OUTPUT specification to describe a numeric edit picture, they may be edited with any legal COBOL numeric editing mask.

Each accumulator number requested on this specification line actually causes the software to set up a family of accumulators, one for every defined break level.

At the beginning of execution of the object program, the software initializes all accumulators to zero values and begins adding the requested fields or literals into the lowest level. Whenever a break occurs on any level, the software adds the contents of the accumulator at the lowest level in the family to the next level accumulator and zeros the lower; this accumulator is then added to the next higher level accumulator and zeroed, and so on up to the level of the break. At the level of each break, the software moves the contents of the accumulator at that level to its associated control footing line, prints the control footing line, adds the contents of that accumulator to the next higher level (or F-level if the break occurred on the highest requested level), and zeros the accumulator. This summing and zeroing of levels upward through the family is called "rolling forward" and is the technique employed by COBRG to acquire cumulative totals.

The TOTAL specification line (discussed in the next sub-section) coordinates the accumulators and break levels, and specifies which accumulators to roll forward and print on which break levels. Of course, all break levels and accumulators coordinated by the TOTAL specification line must have been defined on the BREAK and ACCUMULATOR specification lines respectively.

9.1.3.7 TOTAL Specification - This specification line provides the information that COBRG requires to move the accumulated totals to the control footing lines and print them. It supplies COBRG with the number of the accumulator to be printed, the name of the output field into which that accumulator is to be moved, and a list of the control break levels which will cause that accumulator to be rolled forward, moved, and printed.

COBRG accepts up to 50 TOTAL specification lines completed as discussed below:

Parameter 1 -- Specification identification. Enter the letter T to identify this line as the TOTAL specification.

Parameter 2 -- Accumulator number. Enter the number of the accumulator that is to be moved into an output field and printed.

Parameter 3 -- Output field. Enter the name of the output field into which the accumulator is to be moved. The name may contain up to 24 characters.

Parameter 4 -- Break numbers. Enter the break numbers (0-9,F) which will cause the accumulator (identified in parameter 2 above) to be printed; do not separate these numbers with commas. An entry of F signifies the final break at the end of input data.

The sample program requires control footings at the CITY, STATE, and REGION levels; as well as a final (national) level at the end of the report. In the control footing caused by the CITY (lowest) level, the program must print the total sales for each city. In the control footing caused by the STATE (second) level, the program must print the total sales for each state, which is actually the sum of that state's CITY totals. Likewise, it must print REGIONAL (the sum of the states within the region), and final (the sum of all the regions) totals.

The following TOTAL specification entries will complete the description of the totals required by the sample program.

```
T,0,PRINT-SALES,012F

T,1,ACCOUNT,2F
 1 2 3       4
```

(Accumulator 0, in the sample problem, prints in all control footings and will appear beneath the SALES column.)

Consider what happens when the first break occurs at the STATE level in the sample program. Accumulator 0 contains a summation of the SALES field from all of the detail records that have been accessed up to the record that caused the break. The break at the STATE level forces a break to occur on the CITY level (all lower levels). Thus the program produces a control footing for the CITY level followed by a control footing for the STATE level. Step-by-step, the software takes the following actions when it detects the STATE level break.

1. It prepares and prints the control footing for control level 0 (CITY).

2. It rolls forward the level 0 accumulator by adding the contents of that accumulator to the level 1 accumulator and zeroing the level 0 accumulator.

3. It prepares and prints the control footing for control level 1 (STATE) which is the level of the break.

4. It rolls forward the level 1 accumulator by adding the contents of that accumulator to the level 2 (REGION) accumulator and zeroing the level 1 accumulator.

5. It then returns to printing detail lines.

Thus, the program has printed the total of SALES (accumulator 0) for the CITY level, added them to the STATE level, printed them for the STATE level, and added them to the REGION level. The accumulators for the CITY and STATE levels contain zeroes and the REGION level contains the total sales thus far. When the REGION level breaks, the software will add the sum for the last state in the region to the accumulator for the region level, print that sum in a control footing for the region, add the sales for that region to the final level accumulator (F), and zero the region level accumulator.

The control footing lines for all of these totals will probably require additional textual information to explain the totals being printed. The following specification line provides this information.

9.1.3.8 EMIT Specification - This specification line provides any textual information required by the control footing totals. It contains the break level and the message to be displayed, as well as the name of the field into which the message is to be moved.

COBRG accepts up to 50 EMIT specification lines completed as discussed below:

Parameter 1 -- Specification identification. Enter the letter E to identify this line as the EMIT specification.

Parameter 2 -- Break number. Enter the break priority number which will cause this message to print with its total line.

Parameter 3 -- Message. Enter the message to be placed on the total line. If the information is an alphanumeric literal, enclose it within quotation marks. If the information is either a numeric literal or a data-name, do not enclose it within quotation marks. COBRG will accept a message length of up to 24 characters including the quotation marks; it will not, however, accept imbedded commas since it considers commas to be parameter separators.

Parameter 4 -- Output field. Enter the name of the output field into which this message is to be moved. (This field must have been defined on an OUTPUT specification.)

The following EMIT specification lines will satisfactorily meet the requirements of the total lines in the sample problem:

```
E,0,"CITY SALES",MESS2
E,1,"STATE SALES",MESS2
E,2,"TOTAL SALES =",MESS2
E,F,"TOTAL SALES =",MESS2
1 2 3                 4
```

9.1.3.9 LIST Specification - This specification determines what will be listed on each detail line. It may be thought of as a MOVE statement, and is, in fact, turned into a MOVE statement by the COBRG program. The LIST specification refers to a sending field and a receiving field, which both appear in the source program produced by COBRG as operands in the MOVE statement.

The sending field may be a literal (numeric or alphanumeric) or a field in the input record, and the receiving field must be a field that has been described in an OUTPUT specification (it may be a group item).

The LIST specification contains a control break level field (parameter 2), and if this field contains a break level number from 0 to 9, the field described in parameter 3 will be moved to the field described in parameter 4 and printed only on the first detail line printed after a break on that level. This feature suppresses the printing of columns that would normally contain the same, repetitive, information.

COBRG accepts up to 50 LIST specification lines completed as discussed below:

Parameter 1 -- Specification identification. Enter the letter L to identify this line as the LIST specification.

Parameter 2 -- Control break level. Enter a control break level number or the letter A. If this parameter contains a break level number (0-9), it will cause the item named in parameter 3 (below) to print only after the total lines caused by a break at this level. If this parameter contains an A, the program will print this item on every detail line.

Parameter 3 -- Sending field. Enter the name of the input item to be listed. This item may be a literal.

Parameter 4 -- Receiving field. Enter the name of the output field into which the item in parameter 3 (above) is to be moved for listing.

In the sample program, the software is to print the CITY and STATE fields only after either of them breaks, but it is to detail list the COMPANY and SALES fields. The following LIST specifications will accomplish the requirements of the sample problem:

```
L,A,COMPANY,PRINT-COMPANY
L,A,SALES,PRINT-SALES
L,0,CITY,PRINT-CITY
L,0,STATE, PRINT-STATE
1 2 3      4
```

9.1.4 Output From COBRG

The output from COBRG consists of one or more COBOL source programs and a listing file. The name of each COBOL source program is the program name entered in parameter 2 of the NAME specification line; however, COBRG adds an extension of .CBL to each of these names. The listing file contains a list of input specifications for all of the programs that were generated, and any errors that were detected. (COBRG's error messages are discussed later in this chapter.)

9.1.5 COBRG Command String

To operate COBRG, simply run the program and, in response to the prompting message, enter a command string of the following form (listfile.ext is the name and extension of the listing file, and infile.ext is the name and extension of the input file containing the specifications for one or more programs):

```
CRG>listfil.ext=infile.ext
```

9.1.5.1 Default Assumptions - If the name of the listing file is omitted, COBRG assigns the name of the input file to the listing file. If the extension of the listing file is omitted, COBRG assigns .LST as the extension.

The name of the input file must be entered; COBRG has no default assumptions for the input file name. If the extension of the input file is omitted, COBRG assigns .CRG as the extension.

All version numbers are default. Version numbers may not appear in the command string. Device specifications may not appear in the command string. To assign devices, re-assign the appropriate LUN. (The Appendices of this guide contain a list of current LUN assignments.)

For example, to produce a COBOL source program and a listing file from an input file named COBTST.CRG, enter either of the following two command strings (they are semantically equivalent).

CBL>COBRG/RUN	CBL>COBRG/RUN
CRG>COBTST.LST=COBTST.CRG	CRG>=COBTST

Do not enter the names of the COBOL source programs being produced by COBRG. They are not a part of the command string; COBRG places them on the device associated with the appropriate LUN with user-assigned names and extensions of .CBL.

9.1.6 The Sample Program

The following example of a sample program produces a COBOL source program that lists all of the customers for a marine hardware supply house. It shows the sales to those customers and provides subtotals for each city, for each state, and for each region. (The problem is discussed in detail earlier in this chapter.)

9.1.6.1 Specification Lines - The following input specification lines will produce the desired source program:

```
N,COBRGTEST,,"SALES","LP:",R. A. FUSCI
H,3,58,3,0,COMPANY                CITY       STATE      SALES PG.
I01  INPUT-RECORD.
I    02  COMPANY          PIC X(20).
I    02  FILLER           PIC X(20).
I    02  CITY             PIC X(10).
I    02  STATE            PIC XX.
I    02  FILLER           PIC X(5).
I    02  REGION           PIC X.
I    02  SALES            PIC S9(10)V99.
I    02  FILLER           PIC X(10).
O    03  DETAIL-LINE.
O        05  PRINT-COMPANY        PIC X(20).
O        05  FILLER               PIC XX.
O        05  PRINT-CITY           PIC X(10).
O        05  FILLER               PIC XX.
O        05  PRINT-STATE          PIC XX.
O        05  FILLER               PIC XX.
O        05  PRINT-SALES          PIC Z,ZZZ,ZZZ,ZZZ.ZZ-.
O        05  LINENO               PIC Z(5).
O    03  CFLINE REDEFINES DETAIL-LINE.
O        05  MESS1                PIC X(20).
O        05  ACCOUNT              PIC Z(5).
O        05  MESS2                PIC X(13).
O        05  FILLER               PIC X(22).
B,0,1,0,0,0,10,CITY
B,1,1,0,0,0,2,STATE
B,2,1,0,0,0,1,REGION
B,F,1,0,0,0
A,0,14,2,SALES
A,1,5,0,1,LINENO
T,0,PRINT-SALES,012F
T,1,ACCOUNT,2F
E,0," CITY SALES =",MESS2
E,1,"STATE SALES =",MESS2
E,2,"REGIONAL ACCOUNTS =",MESS1
E,2,";     SALES =",MESS2
E,F,"NATIONAL ACCOUNTS =",MESS1
E,F,";     SALES =",MESS2
L,A,COMPANY,PRINT-COMPANY
L,A,SALES,PRINT-SALES
L,0,CITY,PRINT-CITY
L,0,STATE,PRINT-STATE
```

9.1.6.2 Listing File - When the preceding specification lines have been entered, COBRG produces a listing file containing the following and names it COBTST.LST.

```
  NAME     BLOCKING   INPUT FILE SPEC     OUTPUT FILE SPEC  DATE   AUTHOR
N COBRGTEST  Ø      "SALES"              "LP:"          Ø8/Ø6/74  R. A. FUSCI

   PAGE   HEADER
 SIZE LOC CONTROL RESET PAGE   HEADING

H  3  058   30               COMPANY      CITY    STATE     SALES PG.

          PRINTER-CONTROL     BREAK
  LEVEL   BEFORE  AFTER   SIZE   NAME

B   0       10      00      010    CITY
B   1       10      00      002    STATE
B   2       10      00      001    REGION
B   F       10      00

    ACCUMULATOR
   #  SIZE  POINT     INPUT NAME                LISTING NAME

A  0   14    02       SALES
A  1   05    00       1                         LINENO

   AC NUMBER      WHEN        OUTPUT NAME

T      0          012F        PRINT-SALES
T      1          2F          ACCOUNT

  BREAK   INPUT NAME                    OUTPUT NAME

E   0     " CITY SALES ="               MESS2
E   1     "STATE SALES ="               MESS2
E   2     "REGIONAL ACCOUNTS ="         MESS1
E   2     ";     SALES ="               MESS2
E   F     "NATIONAL ACCOUNTS ="         MESS1
E   F     ";     SALES ="               MESS2


   WHEN   INPUT NAME                    OUTPUT NAME

L   A     COMPANY                      PRINT-COMPANY
L   A     SALES                        PRINT-SALES
L   0     CITY                         PRINT-CITY
L   0     STATE                        PRINT-STATE
```

9.1.6.3 Source Program - In addition to the listing file, COBRG also produces the following source program and names it COBRGTEST.CBL.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. COBRGTEST.
AUTHOR. R.A.FUSCI,
DATE-WRITTEN. 08/06/74.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT IN-FILE          ASSIGN TO " ".
    SELECT L-PRINT          ASSIGN TO " ".

DATA DIVISION.
FILE SECTION.

FD  IN-FILE
    LABEL RECORDS ARE STANDARD
    VALUE OF ID IS "SALES".

01  INPUT-RECORD.
    02  COMPANY          PIC X(20).
    02  FILLER           PIC X(20).
    02  CITY             PIC X(10).
    02  STATE            PIC XX.
    02  FILLER           PIC X(5).
    02  REGION           PIC X.
    02  SALES            PIC S9(10)V99.
    02  FILLER           PIC X(10).

FD  L-PRINT
    LINAGE IS 60 LINES
    LINES AT TOP 3
    LINES AT BOTTOM 3
    LABEL RECORDS ARE STANDARD
    VALUE OF ID IS "LP:".

01  PRINT-LINE DISPLAY-7.
    02  FILLER               PIC X(57).
    02  HEADER-PAGE          PIC ZZ9.
    02  FILLER               PIC X(72).

/
WORKING-STORAGE SECTION.

77  PAGE-COUNT           PIC S9(3)      COMP.

77  SAVE-01              PIC X(10).
77  SAVE-02              PIC XX.
77  SAVE-03              PIC X.

77  LEVEL-0-SW           PIC 9.
77  LEVEL-1-SW           PIC 9.
77  LEVEL-2-SW           PIC 9.
```

```
01  OUTPUT-LINE DISPLAY-7.
    03  DETAIL-LINE.
        05  PRINT-COMPANY      PIC X(20).
        05  FILLER             PIC XX.
        05  PRINT-CITY         PIC X(10).
        05  FILLER             PIC XX.
        05  PRINT-STATE        PIC XX.
        05  FILLER             PIC XX.
        05  PRINT-SALES        PIC Z,ZZZ,ZZZ,ZZZ.ZZ-.
        05  LINENO             PIC Z(5).
    03  CFLINE REDEFINES DETAIL-LINE.
        05  MESS1              PIC X(20).
        05  ACCOUNT            PIC Z(5).
        05  MESS2              PIC X(13).
        05  FILLER             PIC X(22).
01  LEVEL-0-ACS COMP.
    02  ACCUMULATOR-0-0   PIC S9(12)V9(2).
    02  ACCUMULATOR-0-1   PIC S9(5).

01  LEVEL-1-ACS COMP.
    02  ACCUMULATOR-1-0   PIC S9(12)V9(2).
    02  ACCUMULATOR-1-1   PIC S9(5).

01  LEVEL-2-ACS COMP.
    02  ACCUMULATOR-2-0   PIC S9(12)V9(2).
    02  ACCUMULATOR-2-1   PIC S9(5).

01  LEVEL-F-ACS COMP.
    02  ACCUMULATOR-F-0   PIC S9(12)V9(2).
    02  ACCUMULATOR-F-1   PIC S9(5).

01  HEADER.
    02  FILLER             PIC X(30)
            VALUE "COMPANY              CITY     ".
    02  FILLER             PIC X(30)
            VALUE "  STATE       SALES PG.      ".
    02  FILLER             PIC X(36)
            VALUE "                                    ".
    02  FILLER             PIC X(36)
            VALUE "                                    ".


/
PROCEDURE DIVISION.

BEGIN.
    OPEN INPUT IN-FILE.
    OPEN OUTPUT L-PRINT.
    MOVE SPACES TO OUTPUT-LINE.

    MOVE ZERO TO PAGE-COUNT.
    PERFORM PL-HDR THRU PL-EXIT.

    READ IN-FILE AT END
        DISPLAY "? EMPTY INPUT FILE" STOP RUN.
    GO TO RESET-LEVEL-F.
```

```
READ-IN.
    READ IN-FILE AT END GO TO LEVEL-0-BREAK.


    IF SAVE-03 NOT = REGION
        MOVE 1 TO LEVEL-2-SW
        GO TO LEVEL-0-BREAK.

    IF SAVE-02 NOT = STATE
        MOVE 1 TO LEVEL-0-SW
        GO TO LEVEL-0-BREAK.

    IF SAVE-01 NOT = CITY
        MOVE 1 TO LEVEL-0-SW
        GO TO LEVEL-0-BREAK.


PRINT-DETAIL.
    ADD SALES                          TO ACCUMULATOR-0-0.
    ADD 1                              TO ACCUMULATOR-0-1.

    MOVE COMPANY                       TO PRINT-COMPANY.
    MOVE SALES                         TO PRINT-SALES.

    MOVE ACCUMULATOR-0-1 TO LINENO.

    MOVE OUTPUT-LINE TO PRINT-LINE.
    PERFORM PRINT-1 THRU PL-EXIT.
    MOVE SPACES TO OUTPUT-LINE.

    GO TO READ-IN.

/
LEVEL-0-BREAK.
    MOVE SPACES TO PRINT-LINE.
    PERFORM PRINT-1 THRU PL-EXIT.

    MOVE ACCUMULATOR-0-0 TO PRINT-SALES.
    MOVE " CITY SALES ="               TO MESS2.
    MOVE OUTPUT-LINE TO PRINT-LINE.
    PERFORM PRINT-1 THRU PL-EXIT.
    MOVE SPACES TO OUTPUT-LINE.

    ADD ACCUMULATOR-0-0 TO ACCUMULATOR-1-0.
    ADD ACCUMULATOR-0-1 TO ACCUMULATOR-1-1.

    IF LEVEL-0-SW = 1 GO TO RESET-LEVEL-0.

LEVEL-1-BREAK.
    MOVE SPACES TO PRINT-LINE.
    PERFORM PRINT-1 THRU PL-EXIT.

    MOVE ACCUMULATOR-1-0 TO PRINT-SALES.
    MOVE "STATE SALES ="               TO MESS2.
    MOVE OUTPUT-LINE TO PRINT-LINE.
    PERFORM PRINT-1 THUR PL-EXIT.
    MOVE SPACES TO OUTPUT-LINE.

    ADD ACCUMULATOR-1-0 TO ACCUMULATOR-2-0.
    ADD ACCUMULATOR-1-1 TO ACCUMULATOR-2-1.

    IF LEVEL-1-SW = 1 GO TO RESET-LEVEL-1.
```

```
LEVEL-2-BREAK.
    MOVE SPACES TO PRINT-LINE.
    PERFORM PRINT-1 THRU PL-EXIT.

    MOVE ACCUMULATOR-2-0 TO PRINT-SALES.
    MOVE ACCUMULATOR-2-1 TO ACCOUNT.
    MOVE "REGIONAL ACCOUNTS ="    TO MESS1.
    MOVE ";      SALES ="         TO MESS2.
    MOVE OUTPUT-LINE TO PRINT-LINE.
    PERFORM PRINT-1 THRU PL-EXIT.
    MOVE SPACES TO OUTPUT-LINE.

    ADD ACCUMULATOR-2-0 TO ACCUMULATOR-F-0.
    ADD ACCUMULATOR-2-1 TO ACCUMULATOR-F-1.

    IF LEVEL-2-SW = 1 GO TO RESET-LEVEL-2.

LEVEL-F-BREAK.
    MOVE SPACES TO PRINT-LINE.
    PERFORM PRINT-1 THUR PL-EXIT.

    MOVE ACCUMULATOR-F-0 TO PRINT-SALES.
    MOVE ACCUMULATOR-F-1 TO ACCOUNT.
    MOVE "NATIONAL ACCOUNTS ="    TO MESS1.
    MOVE ";      SALES ="         TO MESS2.
    MOVE OUTPUT-LINE TO PRINT-LINE.
    PERFORM PRINT-1 THRU PL-EXIT.
    MOVE SPACES TO OUTPUT-LINE.

    GO TO FINISH.

/
RESET-LEVEL-F.
    MOVE LOW-VALUES TO LEVEL-F-ACS.

RESET-LEVEL-2.
    MOVE LOW-VALUES TO LEVEL-2-ACS.
    MOVE REGION                   TO SAVE-03.
    MOVE ZERO TO LEVEL-2-SW.

RESET-LEVEL-1.
    MOVE LOW-VALUES TO LEVEL-1-ACS.
    MOVE STATE                    TO SAVE-02.
    MOVE ZERO TO LEVEL-1-SW.

RESET-LEVEL-0.
    MOVE LOW-VALUES TO LEVEL-0-ACS.
    MOVE CITY                     TO SAVE-01.
    MOVE CITY                     TO PRINT-CITY.
    MOVE STATE                    TO PRINT-STATE.
    MOVE ZERO TO LEVEL-0-SW.
    GO TO PRINT-DETAIL.

/
PRINT-1.
    WRITE PRINT-LINE BEFORE 1 LINES
            AT EOP GO TO PL-HDR.
    GO TO PL-EXIT.

PRINT-3.
    WRITE PRINT-LINE BEFORE 3 LINES
            AT EOP GO TO PL-HDR.
    GO TO PL-EXIT.
```

```
PL-HDR.
    MOVE SPACES TO PRINT-LINE.

PRINT-CH-1.
    WRITE PRINT-LINE BEFORE PAGE.
    MOVE HEADER TO PRINT-LINE.
    ADD 1 TO PAGE-COUNT.
    MOVE PAGE-COUNT TO HEADER-PAGE.
    WRITE PRINT-LINE BEFORE 3 LINES.

PL-EXIT. EXIT.

FINISH.
    CLOSE IN-FILE.
    CLOSE L-PRINT.

    STOP RUN.
```

9.1.6.4 Input File - COBRG's job is finished when it has produced the listing file and source program. The source program must now be compiled and executed with the following input file:

```
WIDGET BOATS          13217 ATLANTIC AVE  BRIDGEPORTCT066041000002201942
TURNBUCKLE MARINE     477 MAIN ST         BRIDGEPORTCT066041000001196700
ANCHOR BOATS          991 SEAWARD RD      BRIDGEPORTCT066041000000812559
CHAIN REPAIR          94 WATER ST         BRIDGEPORTCT066041000000487300
BLOCK MARINA          9473 ATLANTIC AVE   BRIDGEPORTCT066041000000023173
WINDY BOAT SALES      741 FRONT ST        NEW HAVEN CT065131000001278811
WINDWARD SAILS        21 OCEANSIDE        NEW LONDONCT063201000000734612
HARBOR MARINA         137 FRONT ST        NEW LONDONCT063201000004669400
SNUG HARBOR MARINE    14 HIDDEN COVE RD   NEW LONDONCT063201000000445227
BLUE WATER SAILS      BOX 973             NEW LONDONCT063201000001350000
CABIN OUTFITTERS      4973 ATLANTIC AV    BOSTON     MA021011000003151991
SAFE HARBOR MARINA    788 ATLANTIC AV     BOSTON     MA021031000001246774
HIDDEN COVE MARINA    HIDDEN COVE RD      BOSTON     MA021031000000962588
SPINNAKER SPECIALTY   24 SEAWARD LANE     BOSTON     MA021021000001837300
CUDDY BOAT SALES      244 HARBOR RD       BOSTON     MA021041000002173100
HARBOR MARINE         T WHARF             BOSTON     MA021011000001228800
WIDGET SALES & SERV.  17MAIN ST           FALMOUTH   MA025401000004184651
HARBOR RIGGING        49 FRONT ST         FALMOUTH   MA025401000000929451
HARBOR BOAT SALES     49 FRONT ST         FALMOUTH   ME041051000000895200
OCEAN SAILS           51 FRONT ST         FALMOUTH   ME041051000000300054
OCEAN HARDWARE        LAND'S END          FALMOUTH   ME041051000001981800
TURNBUCKLE HARDWARE   HARBOR LANE         PORTLAND   ME041011000002726629
ANCHOR SALES          95 ATLANTIC ST      PORTLAND   ME041011000002726629
SALT WATER HARDWARE   14 BAY RD           PORTLAND   ME041011000004617257
SAFE ANCHORAGE        LEEWARD LANE        PORTLAND   ME041011000000173000
HARBOR RIGGING        73 HARBOR RD        PORTLAND   ME041011000000138917
OUT-BOUND MARINE      721 ATLANTIC AVE    PORTLAND   ME041011000001464773
HARBOR TACKLE         24 EAST MAIN ST     PORTLAND   ME041011000002529524
OCEAN BOATS           196 BAY RD          PORTLAND   ME041011000004255300
DOCKSIDE REPAIR       MAIN WHARF          PORTLAND   ME041011000001310171
MARINE HARDWARE       94 FRONT ST         PORTLAND   ME041011000000981922
HARBOR ROPE WORKS     BACKWATER LANE      PORTLAND   ME041011000000076700
WIDGET HARDWARE       14 MAIN ST          PORTSMOUTHNH038011000000822599
SNUG COVE TACKLE      SNUG COVE RD        PORTSMOUTHNH038011000002137313
OCEAN SAILMAKERS      NEWELL LANE         PORTSMOUTHNH038011000004713129
WIDGET BOAT HAUL      317 RIVER ST        PORTSMOUTHNH038011000001248837
HARBOR BOAT SALES     245 RIVER ST        PORTSMOUTHNH038011000007654644
GALLEY HARDWARE       724 RIVER ST EXTEN. PORTSMOUTHNH038011000000369475
RACING RIGGING        BOX 427             PORTSMOUTHNH038011000003595200
BOAT BUILDERS         425 TIDE FLAT RD    PORTSMOUTHNH038011000000400022
INLAND BOAT SALES     137 LAKE ST         MONTPELIERVT056021000000101999
SNUG COVE BOATS       21 BAY ST           MONTPELIERVT056021000000106748
EASTERN LAKE MARINA   91 HARBOR RD        BUFFALO    NY142222000000402575
GREAT LAKE BOATS      1321 HARBOR RD      BUFFALO    NY142252000000347300
HARBOR SLS & SERV     FRONT ST            BUFFALO    NY142252000000233194
```

9.1.6.5 Printed Report - As it reads the preceding input file, CO9RGTEST.CBL produces the following:

```
COMPANY                 CITY          STATE        SALES PG.     1


WIDGET BOATS            BRIDGEPORT    CT            22,019.42     1
TURNBUCKLE MARINE                                   11,867.00     2
ANCHOR BOATS                                         8,125.54     3
CHAIN REPAIR                                         4,873.00     4
BLOCK MARINE                                           231.73     5

                              CITY SALES =          47,216.74
WINDY BOAT SALES        NEW HAVEN     CT            12,788.11     1

                              CITY SALES =          12,788.11
WINDWARD SAILS          NEW LONDON    CT             7,346.12     1
HARBOR MARINA                                       86,694.00     2
SNUG HARBOR MARINE                                   4,452.27     3
BLUE WATER SAILS                                    13,500.00     4

                              CITY SALES =          71,992.39

                             STATE SALES =         131,987.24
CABIN OUTFITTERS        BOSTON        MA            31,519.91     1
SAFE HARBOR MARINA                                  12,467.74     2
HIDDEN COVE MARINA                                   9,625.88     3
SPINNAKER SPECIALTY                                 18,373.00     4
CUDDY BOAT SALES                                    21,731.00     5
HARBOR MARINE                                       12,788.00     6

                              CITY SALES =         106,005.53
WIDGET SALES & SERV.    FALMOUTH      MA            41,846.51     1
HARBOR RIGGING                                       9,294.51     2

                              CITY SALES =          51,141.02

                             STATE SALES =         157,146.55
HARBOR BOAT SALES       FALMOUTH      ME             8,952.00     1
OCEAN SAILS                                          3,000.54     2
OCEAN HARDWARE                                      19,818.00     3

                              CITY SALES =          31,770.54
TURNBUCKLE HARDWARE     PORTLAND      ME            27,266.29     1
ANCHOR SALES                                        27,266.29     2
SALT WATER HARDWARE                                 46,172.57     3
SAFE ANCHORAGE                                       1,730.00     4
HARBOR RIGGING                                       1,389.17     5
OUT-BOUND MARINE                                    14,647.73     6
HARBOR TACKLE                                       25,295.24     7
OCEAN BOATS                                         42,553.00     8
DOCKSIDE REPAIR                                     13,101.71     9
MARINE HARDWARE                                      9,819.22    10
HARBOR ROPE WORKS                                      767.00    11

                              CITY SALES =         210,008.22

                             STATE SALES =         241,778.76
WIDGET HARDWARE         PORTSMOUTH    NH             8,225.99     1
SNUG COVE TACKLE                                    21,373.13     2
OCEAN SAILMAKERS                                    47,131.29     3
WIDGET BOAT HAUL                                    12,488.37     4
```

```
COMPANY                  CITY        STATE        SALES PG.    2

HARBOR BOAT SALES                                 76,546.44    5
GALLEY HARDWARE                                    3,694.75    6
RACING RIGGING                                    35,952.00    7
BOAT BUILDERS                                      4,000.22    8

                         CITY SALES =            209,412.19

                        STATE SALES =            209,412.19
INLAND BOAT SALES        MONTPELIER  VT            1,019.99    1
SNUG COVE BOATS                                    1,067.48    2

                         CITY SALES =              2,087.47

                        STATE SALES =              2,087.47

REGIONAL ACCOUNTS =   42;      SALES =            742,422.21
EASTERN LAKE MARINA      BUFFALO     NY            4,025.75    1
GREAT LAKE BOATS                                   3,473.00    2
HARBOR SLS & SERV                                  2,331.94    3

                         CITY SALES =              9,830.69

                        STATE SALES =              9,830.69

REGIONAL ACCOUNTS =    3;      SALES =              9,830.69

NATIONAL ACCOUNTS =   45;      SALES =            752,252.90
```

9.1.7 Error Messages

If any of the specification lines contain errors, or if the number of one type of specification lines exceeds COBRG's limit for that type, the listing file will contain the following error messages and COBRG will abort execution.

Before issuing an error message, COBRG assumes that any non-numeric character appearing in a numeric parameter is a zero. It ignores imbedded blanks; however, if a numeric parameter contains all blanks, COBRG assumes the blanks to be zeroes.

SPECIFICATION LINE	MESSAGE	EXPLANATION
(ALL)	IMPROPER INPUT CARD	Parameter 1 (specification identification) of a specification line does not contain one of the following characters: N,I,O,H,B,A,T,E,L. (COBRG accepts 1 in place of I and 0 in place of O.)
ACCUMULATOR	DUPLICATE AC	Parameter 2 (accumulator number) contains a number that has already been specified.
	IMPROPER AC SIZE	Parameter 3 (accumulator size) does not contain a number from 1 to 18.

SPECIFICATION LINE	MESSAGE	EXPLANATION
	IMPROPER POINT LOCATION	Parameter 4 (decimal positions) contains a number that places the decimal point outside of the accumulator.
	NO INPUT NAME	Parameter 5 (input field) does not contain an entry, leaving the accumulator with no input.
BREAK	BOTH SPACE AND SKIP	Both parameter 3 (blank lines, before printing) and parameter 4 (skip channel, before printing) contain entries, or both parameter 5 (blank lines after printing) and parameter 6 (skip channel, after printing) contain entries. (COBRG will not accept both a space and a skip before printing, or both a skip and a space after printing.)
	IMPROPER FIELD SIZE	Parameter 7 (field size) contains a zero.
	IMPROPER LEVEL	Parameter 2 (break level) contains an entry other than 0-9, or F.
	IMPROPER SPACE OR SKIP	Either parameter 3 (blank lines, before printing) does not contain a number from 0-9; or parameter 4 (skip channel, before printing), parameter 5 (blank lines, after printing), or parameter 6 (skip channel, after printing) does not contain a number from 0-8. COBRG looks for the first non-blank character in the parameter and assumes non-numeric characters to be zeroes.)
	TOO MANY BREAKS	COBRG found more than 25 BREAK specification cards.
EMIT	IMPROPER BREAK NUMBER	Parameter 2 (break number) contains a number other than 0-9 or F.
	NO INPUT NAME	Parameter 3 (message) does not contain an entry.
	NO OUTPUT NAME	Parameter 4 (output field) does not contain an entry.
	TOO MANY EMIT CARDS	COBRG found more than 50 EMIT specification lines.
	UNDEFINED BREAK NUMBER	Parameter 2 (break number) contains a break level number that has not been defined on a BREAK specification line.

SPECIFICATION LINE	MESSAGE	EXPLANATION
HEADER	IMPROPER BREAK FOR RESET	Parameter 7 (page number reset) contains an F. The page counter cannot be reset on an F (final) break level (the report is completed with that break).
	IMPROPER HEADER SKIPPING	Parameter 5 (skip channel) contains a 9.
	NO PAGE SIZE	Parameter 2 (page number field size) contains no entry; however, parameter 3 (page number location) does contain an entry (indicating that a page number field is required by the program).
	NON-EXISTENT BREAK FOR RESET	Parameter 7 (page number reset) contains a break level number that has not been defined on a BREAK specification line.
	PAGE LOC TOO FAR RIGHT	Parameter 3 (page number location) contains a column number that places it so far to the right of the page that it runs off the right side (past column 132).
	SAME BREAK NOTED TWICE	Parameter 7 (page number reset) contains more than one entry of the same break level.
	TOO MANY HEADER CARDS	COBRG found more than two HEADER cards.
INPUT	MORE THAN 1 BATCH OF I CARDS	COBRG found some other specification line separating the INPUT specification lines. (All INPUT specification lines must be consecutive, as must all OUTPUT specification lines.)
	TOO MANY RECORD CARDS	The sum of all of the INPUT and OUTPUT specification lines exceeds 997.
LIST	TOO MANY LIST CARDS	COBRG found more than 50 LIST specification lines.
	IMPROPER BREAK NUMBER	Parameter 2 (control break level) contains an entry other than 0-9 or A.
	NO INPUT NAME	Parameter 3 (sending field) does not contain an entry.
	NO OUTPUT NAME	Parameter 4 (receiving field) does not contain an entry.

SPECIFICATION LINE	MESSAGE	EXPLANATION
	UNDEFINED BREAK LEVEL	Parameter 2 (control break level) contains a break level number that has not been defined on a BREAK specification line.
NAME	NO PROG-ID	Parameter 2 (program name) is missing.
	NO INPUT FILE SPEC	Parameter 4 (input file specification) is missing.
	NO OUTPUT FILE SPEC	Parameter 5 (output file specification) is missing.
OUTPUT	MORE THAN 1 BATCH OF 0 CARDS	COBRG found some other specification line separating the OUTPUT specification lines. (All OUTPUT specification lines must be consecutive, as must all INPUT specification lines.)
	TOO MANY RECORD CARDS	The sum of all of the OUTPUT and INPUT specification lines exceeds 997.
TOTAL	DUPLICATE BREAKS	Parameter 4 (break numbers) contains the same break level number twice.
	IMPROPER BREAK	Parameter 4 (break numbers) contains an entry other than 0-9 or F.
	NON-EXISTENT AC	Parameter 2 (accumulator number) contains an accumulator number that has not been set up with an ACCUMULATOR specification line.
	NO BREAKS SPECIFIED	Parameter 4 (break numbers) does not contain an entry.
	NO OUTPUT NAME	Parameter 3 (output field) does not contain an entry.

9.2 REFORMAT

9.2.1 Introduction

COBOL, as implemented on the PDP-11, accepts source programs that were coded using either the Conventional 80-column card reference Format or the shorter, terminal-oriented PDP-11 format. The REFORMAT utility program reads source programs that were coded in the Terminal format and converts them to 80-column source programs.

Consider the two formats (Section 2.1, Developing a Program, discusses both formats in greater detail):

- The Terminal format is designed for ease of use with context editors controlled from an on-line console keyboard and is, therefore, more compatible for use with the PDP-11 system. It eliminates the line-number and identification fields. It allows horizontal tab characters and short lines.

- The conventional format produces source programs that are compatible with the reference format of other COBOL compilers throughout the industry.

REFORMAT lets you write source programs in the Terminal format; then, if compatibility is ever required for any of those programs, it provides a simple method for conversion to the Conventional format.

REFORMAT uses the following steps to expand each line of Terminal format coding to the 80-character Conventional format coding:

- It generates a 6-character line number of 000010, places that number in the first six character positions of the line, and increases it by 000010 for each subsequent line;

- It places any continuation or comment symbols (-,*, or /) into character position 7;

- It places the coding from the Terminal format line into character positions 8-72, thereby creating a line of Conventional format coding;

- It replaces any horizontal tabs with the appropriate number of space characters to simulate tab stops at character positions 5,13,21,29,37,45,53,61, and 66 of the Terminal format line;

- It moves spaces into any character positions left between the last character of coding and character position 73;

- It places either identification characters (if they were supplied at program initialization) or spaces into character positions 73-80;

- It right justifies (against margin R) the first line of a continued non-numeric literal; thus guaranteeing that the literal will remain the same length as it was in the default format;

- It right justifies (against margin R) the first part of any COBOL word that is split over two lines;

- It creates a line containing a slash (/) in position 7 and space characters in positions 8 through 72 for every form-feed character that it encounters.

9.2.2 REFORMAT Command String

Since REFORMAT is written in COBOL, it runs as a COBOL object program. It has no logical switches. To run it, simply type in the following sequence of responses (prompting messages, typed by the system or REFORMAT are underlined):

```
MCR>CBL
CBL>REFORMAT/RUN
```

These responses cause REFORMAT to begin execution. It immediately requests the file specifications for the two files (input and output) to be processed. In response to its prompting messages, type in the file specifications for your two files.

```
RFM-INPUT FILE SPEC:
RFM-OUTPUT FILE SPEC:
```

When the system has successfully opened both files, REFORMAT types the following request for an identification entry in columns 73 through 80. If you desire an identification entry, type in from 1 to 8 characters. REFORMAT places these characters, left justified, in columns 73 through 80 of each output line. If no entry is required, type a Carriage Return.

```
RFM-COLS 73 TO 80:
```

Following this response, REFORMAT reads the input file and writes it as 80-character records, in Conventional reference format.

When it has processed the last record in the file, REFORMAT displays the following messages. The first indicating the number (nnnnn) of output records produced and the second requesting another input file.

```
RFM-nnnnn LINES PROCESSED.
RFM-INPUT FILE SPEC:
```

If there is another file to be reformatted, follow the same sequence with the specifications for the next file. If not, type Control Z to terminate execution.

9.2.3 Error Messages

If any of the responses to the prompting messages contain detectable errors, REFORMAT displays the following messages indicating the problem.

```
RFM-ERROR IN OPENING INPUT FILE
RFM-TRY AGAIN
RFM-INPUT FILE SPEC:
```

The system could not open the input file. Either the file is not present on the device specified (the default device is SY:) or the file name is typed incorrectly. The usual I/O error messages precede this message.

To continue processing that file, examine the input file spec and type in a corrected version. To process another file, type in a new input file spec. To terminate execution, type Control Z.

```
RFM-ERROR IN OPENING OUTPUT FILE
RFM-TRY AGAIN
RFM-OUTPUT FILE SPEC:
```

The system could not open the output file. An incorrectly typed file specification usually causes this error. (The default device is SY:.) The usual I/O error messages precede this message.

To continue, examine the output file spec and type in a corrected version. To terminate execution, type Control Z.

```
RFM-INPUT FILE IS EMPTY
RFM-INPUT FILE SPEC:
```

The system successfully opened the input file, but the first READ statement encountered the AT END condition.

To continue, type in a new input file specification for another file. To terminate execution, type Control Z

```
RFM-ERROR IN READING INPUT FILE
RFM-INPUT FILE SPEC:
```

The first attempt to read the input file was unsuccessful. This error is usually caused by an input record length exceeding 86 characters. (Although terminal format records should not exceed 66 characters in length, REFORMAT provides a record area of 86 characters and ignores the right-most 20 characters.)

To continue, type in a new input file specification for another file. To terminate execution, type Control Z.

```
RFM-ERROR IN READING INPUT FILE
RFM-REFORMATTING ABORTED
RFM-nnnnn LINES PROCESSED
RFM-INPUT FILE SPEC:
```

While reading input records (other than the first record), REFORMAT was unsuccessful in an attempt to read a record. It terminates execution and closes both files.

To process another file, type in a new input file spec and continue with the prompting message sequence. To terminate execution, type Control Z.

```
RFM-ERROR IN WRITING OUTPUT FILE
RFM-REFORMATTING ABORTED
RFM-nnnnn LINES PROCESSED
RFM-INPUT FILE SPEC:
```

REFORMAT was unsuccessful in an attempt to write an output record. It terminates execution and closes both files.

To process another file, type in a new input file spec and continue with the prompting message sequence. To terminate execution, type Control Z.

CHAPTER 10

ERROR MESSAGES

10.1 COMPILER SYSTEM ERRORS

The PDP-11 COBOL compiler is a complex system program consisting of many program overlays that manipulate numerous data structures. Throughout the compiler, consistency checks are performed on program flow and the contents of data fields. If the compiler detects an inconsistency, it types a message on the console and terminates the compilation.

Since these messages are very infrequent and require the attention of DEC software support personnel with additional compiler documentation, the message contains only a number. For example, if the compiler detects system error 1, it displays "SYSTEM ERROR 1" on the console before it terminates the compilation. Some consistency checks can occur only when certain language elements are used in the source program.

In the event of a PDP-11 COBOL compiler system error, contact your DEC Software Support Specialist immediately.

10.2 DIAGNOSTIC ERROR MESSAGES

This chapter contains a numerical listing of the diagnostic messages generated by the PDP-11 COBOL compiler. The compiler generates these messages whenever it detects an error in the source program. In general, a source error detected by the compiler results in the associated diagnostic message being embedded within the source program listing. That is, when an error is detected in the source program, the compiler prints the diagnostic message either before or after the erroneous source program line. There are two exceptions to the general concept of "embedded diagnostics":

1. There may be diagnostic messages listed after the last entry in the Data Division and before the PROCEDURE DIVISION header. These diagnostic messages indicate the detection of duplicate data-name declarations and erroneous data-names referenced in the RELATIVE KEY, FILE STATUS, LINAGE, and VALUE OF ID clauses.

2. There may be diagnostic messages listed after the last line of the Procedure Division. These diagnostic messages

indicate the detection of duplicate procedure names and references to undefined procedure names.

In addition to the error-message number and message text, the display contains a source line number, which identifies the error line, and an alphabetic code (discussed below) which informs the user of the seriousness of the error. The information within a diagnostic message line is displayed (from left to right) in the following order:

1. the alphabetic code,

2. the source line number,

3. the numerical error number,

4. the text of the diagnostic message.

For convenience, the alphabetic code is left-justified in the listing so the user merely scans the listing to identify any diagnostic message issued during compilation. Again, for the user's convenience, a summary of the number of errors detected during the compilation is given at the end of the source listings. If no errors are detected during the compilation, the compiler prints "NO ERRORS" at the end of the source listing.

The following illustration shows a typical diagnostic message and the manner in which it appears on the source listing:

```
COBOL 01.00 SRC:XMV903.CBL;1***   14-AUG-74   18:49:10  PAGE 003

  00096          MOVE 72.5 TO N2
  00097          IF N2 NOT = T2 DISPLAY "? #10".
  00098      *
  00099          MOVE 3250 TO N3.

I 00099 372 POSSIBLE LOW ORDER RECEIVING FIELD TRUNCATION.

  00100          IF N3 NOT = T3 DISPLAY "? #11".
  00101      *
  00102          MOVE -432 TO N4.
  00103          IF N4 NOT = T4 DISPLAY "? #12".
  00104      *
```

In the example, the diagnostic message is immediately identified by the appearance of the left-justified alphabetic code "I". The alphabetic code indicates that the message is an I-type (informational) diagnostic; the diagnostic is issued for source line number 99; the error number is 372; and the text of the message is "POSSIBLE LOW ORDER RECEIVING FIELD TRUNCATION." Note that the diagnostic message line, in this example, appears after the source line for which it was issued.

The error messages, used in conjunction with this chapter, provide the user with an important debugging tool. This chapter contains information necessary for interpreting the messages. It explains what caused the error and how the compiler handled the error.

Since different errors cause varying degrees of problems for the compiler (some do not affect the compilation at all, while others may be so critical that they cause an abort of the compilation), the PDP-11 COBOL compiler provides four general types (or severity

levels) of diagnostic messages. Alphabetic codes(I,W,F, and A) identify these error levels. When it detects an error in the source program, the compiler attempts to recover from the error and continue to compile the program. This recovery action may force the compiler to make an assumption about the source program. The four levels of diagnostic messages are categorized according to the likelihood that the result of the compiler's assumption will be an object program that runs as originally intended by the programmer.

The following list explains the purpose of and the compiler's action for each of the four message levels:

I (Informational) Informative diagnostic. The purpose of such a diagnostic is to convey information to the user in an observational or advisory capacity. The compiler's error recovery (if any is required) is almost certain to be that desired by the user.

W (Warning) Warning diagnostic. The purpose of this type of message is to warn the user that something is wrong with the associated source statement, but that the compiler can take corrective action on the source element in error. The compiler's recovery action may not be that desired by the user, but the statement, as corrected by the compiler, will be executable.

F (Fatal) Fatal diagnostic. The purpose of such a diagnostic is to indicate to the user that something is fatally wrong with the indicated source statement. By fatal, the compiler means it cannot generate the object code required for the functionality the programmer coded in the erroneous source statement. The compiler's error recovery action will probably leave out a portion of the source program. In general, the compiler will not produce an object program for COBOL source programs which have F-type errors in them. However, the user can force the compiler to generate an object program by specifying the /ACC:2 switch in the command string input to the compiler prior to compilation (See Chapter 2 for a detailed explanation of the /ACC:n switch.) The /ACC:2 switch instructs the compiler to generate an object program, even if the source program contains F-type errors. In this case, when an F-type error is detected in the Procedure Division, the compiler generates special error trap object code in place of the incorrect source statement. When the object program is executed and the special error trap code is encountered, the software displays the following message on the console and aborts the program execution:

FATAL ERROR ON SOURCE LINE XXXXX

where XXXXX is the source line number for which an F-type diagnostic was issued during compilation. For F-type diagnostics issued in the Identification, Environment, and Data Divisions, no special error trap coding is generated since, in general, executable code is not generated for these divisions. However, the fact that F-type diagnostics are issued for these divisions can have a definite effect on the behavior of the execution of the object program.

WARNING: when the user specifies the /ACC:2 switch, the user is formally acknowledging to the software a willingness to let the program go into execution even though it may have fatal errors in it. Because the source program has very severe errors in it, the behavior of the associated object program is, in general, unpredictable. In certain cases, such as a COBOL program with files OPENed in I-O mode, letting the program with F-type errors go into execution could be disastrous. Thus, the /ACC:2 switch should be used with caution. The facility is provided as an extra debugging option. It can be useful in shortening the compile-debug cycle, particularly if applied to large COBOL programs which take considerable compilation time. The point is that the user should use the /ACC:2 facility wisely and discretely.

A (Abortive) Abortive diagnostic. The purpose of this type of diagnostic is to inform the user that the compiler must abort compilation. The compiler's error recovery is not possible: it can make no valid assumptions and has no choice but to abort the compilation.

The following pages of chapter 10 contain the PDP-11 COBOL compiler diagnostic error messages arranged in numerical order for easy reference. The general format of presentation is to give the error message number and the text of the diagnostic message to the left. On the right, a detailed explanation of the diagnostic is given indicating the reason(s) for which the diagnostic message is issued and the recovery action taken by the compiler.

NOTE

In many explanations, the word "Fatal." appears as the very last sentence of the explanation. This means that this is a fatal diagnostic issued in the Procedure Division. If the /ACC:2 switch is specified in the command string input to the compiler, the associated diagnostic message will cause the generation of the special error trap coding discussed previously.

```
001  CONTINUE PUNCH WITH BLANK          ;A blank line has a continue
     STATEMENT. IGNORED.                ;punch. The continue punch is
                                        ;ignored.

002  QUOTE OR CONTINUE PUNCH MISSING.   ;A non-numeric literal has no
     QUOTE ASSUMED.                     ;quote and the following line
                                        ;has no continue punch. A
                                        ;terminal quote is assumed at
                                        ;the end of the line.

003  VIOLATION OF AREA A.               ;The first non-blank character
     ASSUMED CORRECT.                   ;on a continued line occurs in
                                        ;Area A. The error is
                                        ;ignored.
```

```
004  LINE LENGTH EXCEEDS INPUT
     BUFFER.  TRUNCATED.
                                  ;Continuation lines cause a
                                  ;COBOL word to exceed the
                                  ;capacity of the input buffer.
                                  ;The word is truncated on the
                                  ;right;  the  number  of
                                  ;characters retained depends
                                  ;on the type of word being
                                  ;processed.

005  .IO CONTROL.  WITHOUT .FILE
     CONTROL.  IGNORED.
                                  ;An  I-O-CONTROL  paragraph
                                  ;appears when no FILE-CONTROL
                                  ;paragraph was present.  The
                                  ;I-O-CONTROL  paragraph  is
                                  ;ignored.

006  .STRING.  DATA ITEM MUST HAVE
     DISPLAY USAGE.
                                  ;A data  item  in  a  STRING
                                  ;statement has been given a
                                  ;COMP or INDEX usage.  Fatal.

007  NAME EXCEEDS 30 CHARACTERS.
     TRUNCATED TO 30.
                                  ;A character  string  which
                                  ;appears  to be a name exceeds
                                  ;30 characters in length.  The
                                  ;string is  truncated  on  the
                                  ;right to 30 characters.

010  NUMERIC LITERAL OVER 18
     DIGITS.  TRUNCATED TO 18.
                                  ;A numeric literal exceeds 18
                                  ;digits    in    length.  The
                                  ;literal is truncated on the
                                  ;right,  with  any  necessary
                                  ;adjustment to scaling.  The
                                  ;sign is retained.

011  NUMERIC LITERAL HAS MULTIPLE
     DECIMAL POINTS.
                                  ;A numeric  literal  has  more
                                  ;than one decimal point.

012  PICTURE CLAUSE ILLEGAL ON
     GROUP LEVEL.  IGNORED.
                                  ;A group  level  item  has  a
                                  ;PICTURE clause.  The clause
                                  ;is ignored.

013  .SELECT. NOT FOUND.  SENTENCE
     IGNORED.
                                  ;A  FILE-CONTROL  statement
                                  ;should begin with the word
                                  ;SELECT, but does not.  All
                                  ;words  up  to the next period
                                  ;are ignored.

014  JUST.SYNC.BLANK CLAUSES
     WRONG AT GROUP.  IGNORED.
                                  ;A group  level  item  may not
                                  ;contain  JUSTIFIED,  SYNCHRO-
                                  ;NIZED,  or  BLANK  WHEN ZERO
                                  ;clauses.    The    clause    is
                                  ;ignored.

015  FILENAME MISSING OR
     INVALID.  SELECT IGNORED.
                                  ;A SELECT  statement  either
                                  ;contains no user  name or the
                                  ;the user name is invalid.  The
                                  ;SELECT statement is ignored.

016  USAGE CONFLICTS WITH GROUP
     USAGE.  USES  GROUP.
                                  ;The usage  specified for this
                                  ;item  differs  from the usage
                                  ;stated  at  a  higher  group
                                  ;level.  The group level usage
                                  ;is used.
```

```
017  ILLEGAL NUMERIC DATANAME
     IN .STRING.                    ;A numeric data item in a
                                    ;STRING statement has an
                                    ;illegal description. Fatal.

020  .ALL. ILLEGAL IN CONTEXT OF
     .STRING. STATEMENT.            ;An ALL literal has been used
                                    ;in a STRING statement. Fatal.

021  SYNTAX ERROR OR NO
     TERMINATOR. CLAUSES SKIPPED.   ;A SELECT statement is missing
                                    ;its terminating period; or an
                                    ;error causes the statement to
                                    ;be processed before all
                                    ;clauses were found. The
                                    ;SELECT statement is ignored.

022  NUMERIC LITERAL ILLEGAL
     IN THIS STATEMENT.             ;A STRING, UNSTRING, or
                                    ;INSPECT statement contains a
                                    ;numeric literal. Fatal.

023  SENDING LIST OMITTED IN
     .STRING. STATEMENT.            ;A STRING statement contains
                                    ;no sending fields before a
                                    ;DELIMITED BY phrase. Fatal.

024  MORE THAN ONE FILENAME
     IN .ASSIGN.                    ;The non-numeric literal of an
                                    ;ASSIGN clause contains more
                                    ;than one file specification.
                                    ;Only the first specification
                                    ;is used.

025  ILLEGAL DATANAME FOLLOWS
     .INTO. IN .STRING.             ;The receiving field of a
                                    ;STRING statement is invalid.
                                    ;Fatal.

026  SUBSCRIPTING DEPTH EXCEEDS
     3.  OVER 3 IGNORED.            ;This OCCURS clause is nested
                                    ;more than 3 deep. The OCCURS
                                    ;clause is ignored.

027  VALUE ILLEGAL IN OCCURS
     ITEM.  IGNORED.                ;A VALUE clause appears in an
                                    ;item with an OCCURS clause or
                                    ;in an item subordinate to an
                                    ;OCCURS clause. The VALUE
                                    ;clause is ignored.

030  VALUE ILLEGAL IN
     REDEFINES ITEM.  IGNORED.      ;A VALUE clause appears in an
                                    ;item which either contains a
                                    ;REDEFINES clause, or is
                                    ;subordinate to an item with a
                                    ;REDEFINES clause.

031  NO TERMINATOR FOR .IO
     CONTROL.  PARAGRAPH.           ;The I-O-CONTROL paragraph is
                                    ;not terminated by a period.
                                    ;The terminator is assumed
                                    ;present.

032  .MAP. NOT APPLICABLE
     TO SEQ. FILE.  IGNORED.        ;An APPLY clause with the MAP
                                    ;option was given for a file
                                    ;that has SEQUENTIAL
                                    ;organization. The APPLY
                                    ;clause is ignored.
```

```
033  AN IO CONTROL CLAUSE
     WITHOUT FILES.                    ;A file-name is missing in a
                                       ;clause of the I-O-CONTROL
                                       ;paragraph. The clause is
                                       ;ignored.

034  SYNTAX ERROR IN .APPLY..          ;An APPLY clause has illegal
                                       ;syntax. The clause is
                                       ;ignored.

035  INVALID ACCESS MODE.
     TREAT AS SEQUENTIAL.              ;The SELECT statement contains
                                       ;an invalid ACCESS mode.
                                       ;SEQUENTIAL access mode is
                                       ;assumed.

036  INVALID FILE ORGANIZATION.
     TREAT AS SEQUENTIAL.              ;THE SELECT statement contains
                                       ;an invalid ORGANIZATION
                                       ;specification. SEQUENTIAL
                                       ;organization is assumed.

037  NO SELECT STATEMENTS.             ;A FILE-CONTROL paragraph
                                       ;either contains no SELECT
                                       ;statements or none of those
                                       ;present are valid. The
                                       ;FILE-CONTROL paragraph is
                                       ;ignored.

040  .ASSIGN. OMITTED FROM
     SELECT. SELECT IGNORED.           ;A SELECT statement contains
                                       ;no ASSIGN clause. The SELECT
                                       ;statement is ignored.

041  DECIMAL PLACES TRUNCATED.         ;Decimal places have been
                                       ;truncated from a numeric
                                       ;literal during conversion for
                                       ;use as an integer. The
                                       ;integer positions are used.

042  INTEGER EXPECTED, ZERO
     ASSUMED.                          ;An integer literal was
                                       ;expected but fractional
                                       ;positions were found. The
                                       ;literal is ignored and a
                                       ;value of zero is assumed.

043  INTEGER VALUE TOO BIG.
     LARGEST VALUE USED.               ;A numeric literal is too big
                                       ;for conversion as an integer
                                       ;in the given context. A
                                       ;value of 32,767 is used.

044  ERROR IN DATA RECORDS
     CLAUSE. CLAUSE SKIPPED.           ;The word DATA is not followed
                                       ;by RECORD or RECORDS in the
                                       ;DATA RECORDS clause. The
                                       ;DATA RECORDS clause is
                                       ;ignored.

045  ERROR IN LABEL RECORDS
     CLAUSE. CLAUSE SKIPPED.           ;The word LABEL is not
                                       ;followed by RECORD or RECORDS
                                       ;in the LABEL RECORDS clause.
                                       ;The LABEL RECORDS clause is
                                       ;ignored.
```

```
046  NO INTEGER IN BLOCK
     CLAUSE.  CLAUSE SKIPPED.

047  BAD VALUE IN BLOCK
     CLAUSE.  CLAUSE SKIPPED.

050  NO INTEGER IN RECORD
     CLAUSE.  CLAUSE SKIPPED.

051  INVALID VALUE IN RECORD
     CLAUSE.  CLAUSE SKIPPED.

052  INVALID FILENAME.
     FD SKIPPED.

053  FD TERMINATOR MISSING.
     ASSUMED PRESENT.

054  KEY WORD EXPECTED.
     REMAINING CLAUSES SKIPPED.

055  NO  LABEL CLAUSE IN FD.
     .STANDARD. ASSUMED.

056  NO SELECT. FILE
     DELETED.

057  ALLOCATED SPACE EXCEEDS
     LARGEST RECORD.
```

```
;The BLOCK clause does not
;contain a numeric literal.
;The BLOCK clause is ignored.

;The numeric literal in the
;BLOCK clause is not greater
;than zero. The BLOCK clause
;is ignored.

;The RECORD CONTAINS clause
;does not contain a numeric
;literal. The RECORD CONTAINS
;clause is ignored.

;The numeric literal in the
;RECORD CONTAINS clause is not
;greater than zero. The
;RECORD CONTAINS clause is
;ignored.

;The word following FD is not
;valid as a filename. The FD
;entry is ignored.

;The file description entry
;contains no period
;terminator. The error is
;ignored.

;A keyword, which begins a
;clause, such as BLOCK, LABEL,
;DATA, etc. is missing. The
;remainder of the FD entry is
;ignored.

;The FD entry contains no
;LABEL RECORD clause.
;LABEL RECORD IS STANDARD
;is assumed.

;The FD entry's filename has
;no corresponding SELECT
;statement. The FD entry is
;ignored. All references to
;the filename will be
;diagnosed as undefined.

The maximum record size
;specified by the RECORD
;CONTAINS clause exceeds the
;space required for any 01
;entry under the same file.
;The value specified by the
;RECORD CONTAINS clause is
;used.
```

```
060  RECORD AREA EXTENDED TO
     CONTAIN LARGEST RECORD.
                                  ;The space required by the
                                  ;largest 01 record under a
                                  ;file description exceeds the
                                  ;space required by the RECORD
                                  ;CONTAINS clause in the FD
                                  ;entry. The value derived
                                  ;from the 01 record
                                  ;description is used.

061  NO RECORD AREA. FILE
     DELETED.
                                  ;No record area is allocated
                                  ;for a file description. The
                                  ;file description is ignored.
                                  ;All references to the file
                                  ;will be diagnosed as
                                  ;undefined.

062  ILLEGAL DATANAME FOLLOWS
     .WITH POINTER. PHRASE.
                                  ;The data item used as a
                                  ;pointer in a STRING or
                                  ;UNSTRING statement is
                                  ;illegal. Fatal.

063  ILLEGAL SYNTAX IN .STRING.
     STATEMENT.
                                  ;A STRING statement contains
                                  ;illegal syntax. Fatal.

064  77 ILLEGAL IN FILESECTION.
     CHANGED TO 01.
                                  ;A 77 level item description
                                  ;has been found in the FILE
                                  ;SECTION. The 77 level is
                                  ;treated as an 01 level.

065  ILLEGAL WORD FOLLOWS
     .DELIMITED BY. PHRASE.
                                  ;A data-name or literal is
                                  ;expected following a
                                  ;DELIMITED BY phrase in a
                                  ;STRING or UNSTRING statement.
                                  ;Fatal.

066  ILLEGAL USE OF .ALL..
     IGNORED.
                                  ;In the VALUE clause, an ALL
                                  ;numeric literal is detected.
                                  ;This is illegal. ALL is
                                  ;ignored by the compiler.

067  CONDITION NAME MISSING OR
     INVALID.  88 IGNORED.
                                  ;The condition-name in an 88
                                  ;level entry is either missing
                                  ;or invalid. The entire entry
                                  ;is ignored.

070  .AREAS. NOT PRESENT IN
     .RESERVE.. ASSUMED.
                                  ;The RESERVE ... AREAS
                                  ;clause has incorrect syntax.
                                  ;The error is ignored.

071  .REDEFINES. ON 01 LEVEL
     IN FILE SECTION INVALID.
                                  ;The REDEFINES clause is
                                  ;present on the 01 level
                                  ;in the FILE SECTION, where
                                  ;redefinition is implicit.
                                  ;REDEFINES clause is ignored.

072  PICTURE IGNORED
     FOR INDEX ITEM.
                                  ;An item defined as USAGE
                                  ;INDEX has a PICTURE clause.
                                  ;The PICTURE clause
                                  ;is ignored.
```

```
073  NONNUMERIC PIC ON COMP
     ITEM.  TREATED AS DISPLAY.      ;An item defined as USAGE COMP
                                     ;has a picture-string with
                                     ;non-numeric characters. The
                                     ;stated usage is ignored. The
                                     ;item is treated as DISPLAY
                                     ;usage.

074  SUBSCRIPT OUT OF RANGE.
     ASSUME 1.                       ;A literal subscript is either
                                     ;less than 1 or greater than
                                     ;the maximum allowable value.
                                     ;A value of 1 is used.

075  .STATUS.  OMITTED FROM
     .FILE STATUS..  ASSUMED.        ;The FILE STATUS clause has
                                     ;incorrect syntax. The error
                                     ;is ignored.

076  SOME FILES WITHOUT POSIT.
     NO. IN MUL.  FILE TAPE.         ;A MULTIPLE FILE TAPE clause
                                     ;contains file-names with
                                     ;POSITION CLAUSES. Not all
                                     ;the file-names contain
                                     ;POSITION clauses. The error
                                     ;is ignored. File searching
                                     ;during OPEN will find the
                                     ;file.

077  .MULTIPLE FILE TAPE.  SYNTAX
     ERROR.                          ;A MULTIPLE FILE TAPE clause
                                     ;contains a syntax error. The
                                     ;clause is ignored.

100  OPERAND CLASSES IN CONFLICT.    ;One or more operands in a
                                     ;statement have invalid class.
                                     ;Fatal.

101  POSSIBLE RECEIVING
     FIELD TRUNCATION.               ;A MOVE statement results
                                     ;in right hand truncation of
                                     ;the receiving field value.
                                     ;This is not an error and is
                                     ;ignored.

102  TOO FEW SOURCE FIELDS
     FOR ADD .GIVING..               ;At least two valid source
                                     ;operands must appear in an
                                     ;ADD...GIVING statement.
                                     ;Fatal.

103  .EXIT. WAS NOT THE ONLY
     VERB IN PARAGRAPH.              ;An EXIT statement is not the
                                     ;only statement in a
                                     ;paragraph. The EXIT
                                     ;statement is ignored.

104  SENDING ITEM INVALID
     OR OMITTED.                     ;A MOVE statement contains
                                     ;an invalid or missing
                                     ;sending operand. Fatal.

105  SENDING ITEM NOT FOLLOWED
     BY .TO..                        ;A MOVE statement does not
                                     ;have a TO following the
                                     ;sending operand. Fatal.

106  RECEIVING ITEM INVALID  OR
     OMITTED.                        ;A MOVE statement has no valid
                                     ;receiving operand. Fatal.
```

```
107  INVALID CLASS FOR
     DESTINATION FIELD.
                                  ;The receiving operand of an
                                  ;ADD or SUBTRACT statement is
                                  ;not numeric or numeric
                                  ;edited. Fatal.

110  RELATIVE KEY OR STATUS
     NAME INVALID. IGNORED.
                                  ;The name referenced in a
                                  ;RELATIVE KEY or file status
                                  ;clause is invalid. The
                                  ;clause is ignored.

111  .STOP. SYNTAX ERROR.
                                  ;The STOP statement is not
                                  ;followed by a literal or
                                  ;the word RUN. Fatal.

112  .SIZE ERROR. STATEMENT
     INCORRECT.
                                  ;The word ERROR is not found
                                  ;in ON SIZE clause. Fatal.

113  .PROCEDURE DIVISION. OMITTED.
                                  ;The source program does not
                                  ;contain a Procedure Division.
                                  ;Fatal.

114  INTERMEDIATE RESULT TOO LARGE.
     HIGH ORDER TRUNC.
                                  ;An arithmetic statement calls
                                  ;for an intermediate result in
                                  ;excess of 18 digits.
                                  ;The intermediate result is
                                  ;truncated on the left to 18
                                  ;digits with a possible loss
                                  ;of high order non-zero digits
                                  ;at execution time.

116  .DIVISION. OMITTED AFTER
     .PROCEDURE..
                                  ;The word DIVISION is missing
                                  ;in the Procedure Division
                                  ;header. The error is
                                  ;ignored.

117  TERMINATOR MISSING AFTER
     DIVISION HEADER.
                                  ;The period terminator is
                                  ;missing from a Division
                                  ;header. The error is
                                  ;ignored.

120  LITERAL INCOMPATIBLE WITH
     ATTEMPTED USAGE.
                                  ;Conversion of a literal
                                  ;from one form to another
                                  ;has failed. Fatal.

121  DATANAME MUST FOLLOW .INTO.
     IN THIS STATEMENT.
                                  ;A valid data-name is not
                                  ;present following INTO in
                                  ;a STRING or UNSTRING
                                  ;statement. Fatal.

122  NUMERIC OPERAND MUST
     BE INTEGER.
                                  ;A non-numeric operand is
                                  ;illegal in the context of
                                  ;this IF statement. Fatal.

123  OPERANDS CONFLICT IN .SET...
     TO. STATEMENT.
                                  ;A SET...TO statement
                                  ;references invalid operands.
                                  ;Fatal.
```

```
124  OPERANDS CONFLICT IN .SET ...
     BY. STATEMENT.
                                    ;A      SET...BY      statement
                                    ;references invalid operands.
                                    ;Fatal.

125  ILLEGAL FILENAME LITERAL
     OR FILENAME DATANAME.
                                    ;An  ASSIGN  statement or a
                                    ;VALUE    OF    ID   statement
                                    ;contains  an  invalid file
                                    ;specification or data-name.
                                    ;The statement is ignored.

126  INVALID SUBJECT OF
     SIGN CONDITION.
                                    ;The  subject  of  a  sign
                                    ;condition is not a numeric
                                    ;data-name.  Fatal.

127  ITEM IN TABLE MAY NOT
     BE USED AS A SUBSCRIPT.
                                    ;A data  item  used  as  a
                                    ;subscript is itself a table
                                    ;element.  Fatal.

130  .POINTER. MUST FOLLOW .WITH.
     IN THIS STATEMENT.
                                    ;A   STRING   or   UNSTRING
                                    ;statement  has  an  invalid
                                    ;WITH POINTER phrase.  Fatal.

131  RELATIVE KEY INVALID FOR
     SEQ. FILE.  IGNORED.
                                    ;A RELATIVE  KEY  clause has
                                    ;been applied to a file with
                                    ;SEQUENTIAL organization.  The
                                    ;RELATIVE   KEY   clause   is
                                    ;ignored.

132  INVALID KEY WORD OF
     CONDITION CLAUSE.
                                    ;An IF  statement  contains
                                    ;an invalid condition.  Fatal.

133  UNIDENTIFIABLE WORD FOUND IN
     SUBSCRIPT.
                                    ;A subscript  list  contains
                                    ;a  word  which  is  neither a
                                    ;data-name or numeric literal.
                                    ;The     remainder     of     the
                                    ;list or  sentence is ignored.
                                    ;Fatal.

134  INVALID OBJECT  OF
     CONDITION.
                                    ;The object  of  a relation
                                    ;condition   is   an   invalid
                                    ;operand.  Fatal.

135  SUBSCRIPTS OMITTED.  ASSUME
     VALUE OF 1.
                                    ;A reference  to  a table item
                                    ;contains  no  subscript list.
                                    ;Literal subscripts  of 1 are
                                    ;supplied as defaults.

136  RELATIVE INDEX LITERAL OUT
     OF   RANGE.  INDEX USED.
                                    ;The  literal  value  of  a
                                    ;relative  index  causes an
                                    ;out  of  range  reference
                                    ;to  the  table.  The literal
                                    ;value  is  ignored,   and
                                    ;the  index-name only is used.

137  SUBSCRIPTS GIVEN WHERE NOT
     REQUIRED.  IGNORED.
                                    ;A  reference  is  made  to
                                    ;a  non-table  item, and  a
                                    ;subscript  list  follows  the
                                    ;reference.  The   subscript
                                    ;list is ignored.
```

```
140  TOO FEW SUBSCRIPTS GIVEN.
     ASSUME 1 FOR REST.
                                  ;A reference to a table item
                                  ;contains a subscript list
                                  ;with too few subscripts.
                                  ;Default literal subscripts of
                                  ;1 are supplied for missing
                                  ;subscripts.

141  TOO MANY SUBSCRIPTS
     GIVEN.  IGNORE EXCESS.
                                  ;A reference to a table item
                                  ;contains too many subscripts
                                  ;in subscript list. Extra
                                  ;subscripts are ignored.

142  SUBJECT AND OBJECT USAGE
     MUST MATCH.
                                  ;A relation condition between
                                  ;non-numeric operands requires
                                  ;the same usage for both
                                  ;operands. Fatal.

147  ABSOLUTE VALUE STORED.
                                  ;A negative value has been
                                  ;supplied for an unsigned
                                  ;numeric item. The absolute
                                  ;value of the numeric literal
                                  ;is stored in the item.

151  VERB FOUND IN AREA A.
     ALLOWED.
                                  ;A statement begins in
                                  ;Area A. The error is ignored.

152  EXPECTED. RELATIVE KEY.
     DATANAME NOT DEFINED.
                                  ;The data-name given in a
                                  ;RELATIVE KEY clause has not
                                  ;been defined in the Data
                                  ;Division. Fatal.

153  .LINAGE.  CLAUSE DATAITEM
     IS TOO LONG.
                                  ;A data item named in a LINAGE
                                  ;clause is declared in the
                                  ;DATA DIVISION with more than
                                  ;four decimal integer
                                  ;positions of precision.
                                  ;Fatal.

154  PROCEDURE NAME DUPLICATES
     DATA NAME.  ALLOWED.
                                  ;A procedure name is identical
                                  ;to a data-name. The error is
                                  ;ignored, since there can be
                                  ;no ambiguity in legal
                                  ;references.

155  STATEMENTS FOLLOWING .GO.
     CAN NEVER BE EXECUTED.
                                  ;A statement follows an
                                  ;unconditional GO statement.
                                  ;The statements following the
                                  ;GO are compiled, but can
                                  ;not be executed.

156  NONSEQUENTIAL FILE MAY
     NOT BE OPTIONAL.
                                  ;The SELECT statement may
                                  ;specify OPTIONAL only
                                  ;on files with sequential
                                  ;organization. The word
                                  ;OPTIONAL is ignored.

157  FILE HAS IO CONTROL
     CLAUSE CONFLICTS.
                                  ;A file is given conflicting
                                  ;clause specifications in the
                                  ;I-O CONTROL paragraph of the
                                  ;INPUT-OUTPUT SECTION.
```

```
160  FILE REQUIRES REL. KEY.
     TREATED AS SEQ. ACCESS.
                                ;A file with relative
                                ;organization and random
                                ;or dynamic access has no
                                ;RELATIVE KEY clause. The
                                ;access mode is changed to
                                ;sequential.

161  INVALID SUBJECT OF CONDITION.
                                ;The word following IF is
                                ;invalid as the subject of
                                ;a condition. Fatal.

162  UNKNOWN WORD. SCAN TO
     NEXT CLAUSE.
                                ;An unknown word is
                                ;encountered when a clause
                                ;keyword is expected. All
                                ;words are ignored up to the
                                ;next valid clause.

163  CLAUSE DUPLICATED. SECOND
     OCCURRENCE USED.
                                ;A SELECT statement contains
                                ;two occurrences of the same
                                ;clause. The second
                                ;occurrence is used.

164  NO FD FOR THIS SELECT.
                                ;The filename supplied in a
                                ;SELECT statement is not
                                ;further described in an FD
                                ;in the Data Division. The
                                ;SELECT statement is ignored,
                                ;causing the filename to
                                ;become undefined.

165  DIFFERENT SAME REC. AREAS
     FOR SAME AREA.
                                ;The compiler has detected a
                                ;conflict between the SAME
                                ;RECORD AREA clause and the
                                ;SAME AREA clause. The
                                ;compiler rectifies the error
                                ;in the best possible manner.

166  .READ. WITHOUT .INVALID KEY.
     .AT END. OR .USE.
                                ;A READ statement contains
                                ;no conditional clauses and
                                ;the file being read has no
                                ;USE procedure applied to it.
                                ;Fatal.

167  IO CONTROL CLAUSE HAS FILE
     WITH NO .SELECT.
                                ;An I-O-CONTROL clause
                                ;references a file-name which
                                ;was not named in a SELECT
                                ;statement. The filename is
                                ;ignored in the I-O-CONTROL
                                ;statement.

170  INTEGER OMITTED IN
     .RESERVE.. ONE ASSUMED.
                                ;A RESERVE clause fails
                                ;to specify the number of
                                ;buffer areas to reserve. The
                                ;clause is ignored, and a
                                ;default of 1 area is
                                ;supplied.
```

```
171  INVALID SUBJECT OF CLASS
     CONDITION.                      ;The subject of a class
                                     ;condition is not a data
                                     ;item with acceptable class.
                                     ;Fatal.

172  VALUE EXCEEDS FIELD CAPACITY.
     TRUNCATED.                      ;A numeric literal supplied
                                     ;by a VALUE clause exceeds the
                                     ;length of the field. The
                                     ;value is right truncated and
                                     ;stored in the field.

173  NO DATA DIVISION STATEMENTS
     PROCESSED.                      ;The Data Division contains
                                     ;no valid entries. This
                                     ;is an observation only.

174  INVALID GRP LEV NUM.
     REST OF RECORD IGNORED.         ;A level number is encountered
                                     ;which terminates a previous
                                     ;group item, but does not
                                     ;match any previous group
                                     ;item's level number. All
                                     ;data entries are skipped
                                     ;until the next 01 level,
                                     ;level indicator or header.

175  RESERVED WORD AS
     PARAGRAPH NAME. IGNORED.        ;A COBOL reserved word is used
                                     ;as a paragraph or section
                                     ;name. The name is ignored.
                                     ;Fatal.

176  MISSING QUOTE ON CONTINUE
     LINE. QUOTE ASSUMED.            ;A non-numeric literal is
                                     ;continued, but the first
                                     ;non-space character is not a
                                     ;quote. The error is ignored
                                     ;by assuming a quote in front
                                     ;of the first non-space
                                     ;character.

177  COMPARISON OF LITERALS IS
     NOT PERMITTED.                  ;A relation condition has
                                     ;a literal as both subject and
                                     ;object. Fatal.

200  COPY IGNORED WITHIN
     LIBRARY TEXT.                   ;A COPY statement is
                                     ;encountered within library
                                     ;text. The COPY statement
                                     ;is ignored.

201  INVALID FILENAME ON COPY.
     COPY IGNORED.                   ;A COPY statement supplies
                                     ;a file specification which
                                     ;is invalid. The COPY
                                     ;statement is ignored.

202  COPY FILENAME NOT FOUND.        ;A COPY statement supplies
                                     ;a valid file specification,
                                     ;but the file cannot be found
                                     ;on the specified device. The
                                     ;COPY statement is ignored.

203  PERIOD OMITTED AFTER
     .DECLARATIVES..                 ;The word DECLARATIVES is not
                                     ;followed by a period. The
                                     ;error is ignored.
```

```
204  .DECLARATIVES. OMITTED FROM
     .END. STATEMENT.
                                    ;The word END is not followed
                                    ;by DECLARATIVES. END
                                    ;DECLARATIVES is assumed.

205  PERIOD OMITTED AFTER
     .END DECLARATIVES..
                                    ;The words END DECLARATIVES
                                    ;are not followed by a period.
                                    ;The error is ignored.

206  SOURCE PROGRAM ENDS IN
     DECLARATIVES.
                                    ;The end of the source program
                                    ;occurs in the Declaratives
                                    ;area. Fatal.

207  DATANAME MUST FOLLOW
     .WITH POINTER. PHRASE.
                                    ;A STRING or UNSTRING
                                    ;statement contains an invalid
                                    ;WITH POINTER phrase. Fatal.

210  .OVERFLOW. MUST FOLLOW .ON.
     IN THIS STATEMENT.
                                    ;A STRING or UNSTRING
                                    ;statement contains an invalid
                                    ;ON OVERFLOW phrase. Fatal.

211  ILLEGAL SENDING FIELD
     DATANAME IN .UNSTRING.
                                    ;The sending field of an
                                    ;UNSTRING statement has
                                    ;invalid class. Fatal.

212  ILLEGAL SYNTAX IN
     .UNSTRING. STATEMENT.
                                    ;An UNSTRING statement
                                    ;has invalid syntax. Fatal.

213  MULTIPLE SIGN CLAUSES
     ON THIS ITEM.
                                    ;More than one SIGN clause
                                    ;appears in a data
                                    ;description. (SEPARATE must
                                    follow LEADING or TRAILING.)
                                    ;The second clause is used.

215  SIGN CLAUSE ON
     NONNUMERIC ITEM.
                                    ;A SIGN clause appears in
                                    ;a non-numeric data
                                    ;description. The SIGN clause
                                    ;is ignored.

216  SIGN CLAUSE APPLIED
     TO NONDISPLAY ITEM.
                                    ;A SIGN clause appears in
                                    ;a numeric data description
                                    ;with usage other than
                                    ;DISPLAY. The SIGN clause is
                                    ;ignored.

217  SIGN CLAUSE APPLIED
     TO UNSIGNED DATAITEM.
                                    ;A SIGN clause appears in a
                                    ;numeric data description
                                    ;which has no "S" in its
                                    ;PICTURE string. The SIGN
                                    ;clause is ignored.

220  ILLEGAL DELIMITING DATA
     ITEM IN .UNSTRING.
                                    ;An UNSTRING statement
                                    ;references an invalid
                                    ;delimiter. Fatal.

221  .ALL. FIGURATIVE CONSTANT
     ILLEGAL IN .UNSTRING.
                                    ;An UNSTRING statement
                                    ;contains an ALL literal
                                    ;reference. Fatal.
```

```
222  ILLEGAL RECEIVING DATANAME
     IN .UNSTRING.
                                    ;An UNSTRING statement
                                    ;references a receiving data
                                    ;item which is invalid.
                                    ;Fatal.

223  .DELIMITED. CLAUSE REQUIRED
     IN THIS .UNSTRING.
                                    ;An UNSTRING statement
                                    ;contains no DELIMITED BY
                                    ;clause. Fatal.

224  DATANAME MUST FOLLOW
     .DELIMITER IN. PHRASE.
                                    ;An UNSTRING statement
                                    ;contains a DELIMITER
                                    ;IN phrase with an illegal
                                    ;reference. Fatal.

225  ILLEGAL DATANAME FOLLOWS
     .DELIMITER IN. PHRASE.
                                    ;An UNSTRING statement
                                    ;contains a DELIMITER IN
                                    ;phrase referencing a data
                                    ;item which is invalid.
                                    ;Fatal.

226  DATANAME MUST FOLLOW
     .COUNT IN. PHRASE.
                                    ;An UNSTRING statement
                                    ;contains a COUNT IN phrase
                                    ;with an illegal reference.
                                    ;Fatal.

227  ILLEGAL DATANAME FOLLOWS
     .COUNT IN. PHRASE.
                                    ;An UNSTRING statement
                                    ;contains a COUNT IN phrase
                                    ;which references a data item
                                    ;which is invalid. Fatal.

230  DATANAME MUST FOLLOW
     .TALLYING IN. PHRASE.
                                    ;An UNSTRING statement
                                    ;contains a TALLYING phrase
                                    ;with an illegal reference.
                                    ;Fatal.

231  ILLEGAL DATANAME FOLLOWS
     .TALLYING IN. PHRASE.
                                    ;An UNSTRING statement
                                    ;contains a TALLYING
                                    ;phrase referencing a data
                                    ;item which is invalid.
                                    ;Fatal.

232  DATANAME MUST FOLLOW
     .INSPECT. VERB.
                                    ;An INSPECT statement
                                    ;references a data item
                                    ;which is invalid. Fatal.

233  ILLEGAL DATANAME FOLLOWS
     .INSPECT. VERB.
                                    ;An INSPECT statement
                                    ;references a data item
                                    ;which is invalid. Fatal.

234  ILLEGAL DATANAME PRECEDES
     .FOR. IN .INSPECT.
                                    ;An INSPECT...TALLYING
                                    ;statement references a tally
                                    ;data item which is invalid.
                                    ;Fatal.

235  .FOR. OMITTED IN
     .INSPECT. STATEMENT
                                    ;An INSPECT...TALLYING
                                    ;statement has invalid syntax.
                                    ;Fatal.

236  DATANAME MUST FOLLOW
     .TALLYING. PHRASE.
                                    ;An INSPECT...TALLYING
                                    ;statement does not reference
                                    ;a tally data-name. Fatal.
```

```
237  ILLEGAL WORD FOLLOWS
     .FOR. IN .INSPECT.
                                  ;An INSPECT...TALLYING
                                  ;statement does not state
                                  ;a valid search condition.
                                  ;Fatal.

240  DATAITEM OMITTED AFTER
     .ALL. .LEADING. OR .FIRST.
                                  ;An INSPECT statement
                                  ;does not reference a
                                  ;valid search condition.
                                  ;Fatal.

241  .ALL. FIGURATIVE CONSTANT
     ILLEGAL IN .INSPECT.
                                  ;An ALL literal appears in an
                                  ;INSPECT statement. Fatal.

242  ILLEGAL DATANAME FOLLOWS
     .ALL. OR .LEADING.
                                  ;An INSPECT statement
                                  ;does not reference a
                                  ;valid search argument.
                                  ;Fatal.

243  ILLEGAL DATANAME FOLLOWS
     .BEFORE. OR .AFTER.
                                  ;An INSPECT statement does
                                  ;not reference a valid
                                  ;delimiter in the BEFORE/
                                  ;AFTER phrase. Fatal.

244  ILLEGAL DATANAME FOLLOWS
     .BY.
                                  ;An INSPECT statement
                                  ;does not reference a valid
                                  ;replacement argument. Fatal.

245  ILLEGAL DATANAME PRECEDES
     .BY.
                                  ;An INSPECT statement does not
                                  ;reference a legal dataname
                                  ;or literal preceding the
                                  ;BY phrase. Fatal.

246  DATAITEM OMITTED IN
     .BEFORE. OR .AFTER. PHRASE.
                                  ;An INSPECT statement does
                                  ;not reference a legal data
                                  ;name or literal after the
                                  ;BEFORE or AFTER phrase.
                                  ;Fatal.

247  ILLEGAL SYNTAX IN
     .INSPECT. STATEMENT.
                                  ;Both the TALLYING and
                                  ;REPLACING keywords are
                                  ;missing in the INSPECT
                                  ;statement. Fatal.

250  .BY. MUST FOLLOW .CHARACTERS.
     IN REPLACING LIST.
                                  ;The INSPECT...REPLACING
                                  ;statement must have
                                  ;CHARACTERS BY phrase
                                  ;completely specified. Fatal.

251  DATA ITEM OMITTED AFTER
     .BY. IN .INSPECT.
                                  ;The INSPECT...REPLACING
                                  ;statement does not reference
                                  ;a legal data-name or literal
                                  ;after BY. Fatal.
```

```
252  DATAITEM FOLLOWING .BY.
     EXCEEDS 1 CHARACTER.
                                  ;In an INSPECT...REPLACING
                                  ;statement, either when the
                                  ;CHARACTERS BY phrase is
                                  ;specified or when a
                                  ;figurative constant preceding
                                  ;the BY keyword of the ALL,
                                  ;LEADING, or FIRST phrase is
                                  ;specified, the data-name or
                                  ;literal after the BY keyword
                                  ;must be defined as one
                                  ;character in length. Fatal.

253  DATAITEMS BEFORE AND AFTER
     .BY. UNEQUAL IN SIZE.
                                  ;In an INSPECT...REPLACING
                                  ;statement, the data items
                                  ;before and after the BY
                                  ;keyword of the ALL, LEADING,
                                  ;or FIRST phrase must be equal
                                  ;in length. Fatal.

254  .BEFORE. OR .AFTER. OPERAND
     EXCEEDS 1 CHARACTER.
                                  ;In an INSPECT...REPLACING
                                  ;CHARACTERS BY statement, the
                                  ;data-name or literal follow-
                                  ;ing the BEFORE or AFTER
                                  ;keyword must be one character
                                  ;in length. Fatal.

255  ILLEGAL WORD FOLLOWS
     .REPLACING. IN INSPECT..
                                  ;A legal keyword was not
                                  ;recognized following
                                  ;REPLACING in the INSPECT
                                  ;statement. Fatal.

256  .BY. OMITTED AFTER REPLACING
     COMPARISON OPERAND.
                                  ;The keyword BY is omitted in
                                  ;the ALL, LEADING, or FIRST
                                  ;phrase where it separates
                                  ;operands to be compared.
                                  ;Fatal.

275  INDEX DATA ITEM ILLEGAL
     AS INDEX ON TABLE.
                                  ;An index data item is used as
                                  ;an index on a table. The
                                  ;index data item reference is
                                  ;ignored. A literal subscript
                                  ;of 1 replaces the index
                                  ;data item reference.

276  INDEX NAME NOT DEFINED
     FOR THIS TABLE.
                                  ;An index-name used in a sub-
                                  ;script list either is not
                                  ;defined for this table or
                                  ;appears in the wrong logical
                                  ;position of the subscript
                                  ;list for this table. The
                                  ;index-name is ignored and a
                                  ;default value of 1 is assumed
                                  ;as the subscript.

277  RELATIVE INDEX IS INVALID.
                                  ;The literal component of a
                                  ;relative index is zero or
                                  ;less in value or is an
                                  ;invalid word. Relative
                                  ;indexing is ignored and the
                                  ;index-name only is used.
```

```
300  THIS ELEMENTARY ITEM
     CANNOT BE A 01 RECORD.
                                  ;In the FILE SECTION, an
                                  ;elementary item cannot be a
                                  ;01 record if it is an edited
                                  ;item, or computational or
                                  ;numeric with sign processing.

301  LINAGE 0 OR LESS
     THAN FOOTING.
                                  ;The LINAGE clause must
                                  ;specify a page body of at
                                  ;least one line and that page
                                  ;body size must be equal to or
                                  ;greater than the footing size
                                  ;specified in the FOOTING
                                  ;phrase.

302  FILE CLOSED BUT NOT
     OPENED.
                                  ;A CLOSE statement was seen
                                  ;for a file that was not
                                  ;OPENed in this program.
                                  ;Fatal.

303  PRINT CONTROL ON RELATIVE
     FILE.  IGNORED.
                                  ;An APPLY PRINT-CONTROL clause
                                  ;references a file with
                                  ;RELATIVE organization. The
                                  ;filename is ignored in the
                                  ;APPLY clause.

304  SECTION NAME TOO LONG
     FOR .USE. HANDLER.
                                  ;There are too many characters
                                  ;in the section name in the
                                  ;USE statement procedure.
                                  ;Fatal.

305  SECTION OR PARAGRAPH
     NAME MISSING.
                                  ;The Procedure Division does
                                  ;not start with a section or
                                  ;paragraph name or a section
                                  ;header is not followed by a
                                  ;paragraph name.  Fatal.

306  .PROCEDURE. MISSING IN .USE.
     STATEMENT. ASSUMED.
                                  ;The keyword PROCEDURE is
                                  ;missing in the USE
                                  ;statement.  It is assumed and
                                  ;processing is continued.

307  .START. WITHOUT .INVALID
     KEY. OR .USE.
                                  ;The INVALID KEY option is
                                  ;missing from the START
                                  ;statement and no USE proce-
                                  ;dure is declared for the
                                  ;referenced file.  Fatal.

310  .WRITE. WITHOUT .INVALID
     KEY. OR .USE.
                                  ;The INVALID KEY option
                                  ;is missing from the WRITE
                                  ;statement and no USE
                                  ;procedure is declared for
                                  ;the referenced file.  Fatal.

311  DATA DIVISION MUCH TOO
     LARGE.
                                  ;Too much buffer space is
                                  ;being used for the
                                  ;files in this program.  Too
                                  ;many files are declared to be
                                  ;OPEN simultaneously.
```

```
320  FILENAME MUST FOLLOW .CLOSE.
     VERB.                          ;The data item following the
                                    ;CLOSE verb was not a
                                    ;filename. Fatal.

321  .NO. MUST FOLLOW .WITH.
     IT IS ASSUMED.                 ;The keyword NO is missing in
                                    ;the WITH NO REWIND phrase of
                                    ;the CLOSE statement. NO is
                                    ;assumed present.

322  .REWIND. MUST FOLLOW .NO.
     IT IS ASSUMED.                 ;The WITH NO REWIND phrase of
                                    ;the CLOSE statement must be
                                    ;completely specified. It is
                                    ;assumed present.

323  .REMOVAL. MUST FOLLOW .FOR.
     IT IS ASSUMED.                 ;The FOR REMOVAL phrase of the
                                    ;CLOSE statement must be
                                    ;completely specified. It is
                                    ;assumed present.

324  .LOCK. OMITTED AFTER .WITH.
     IT IS ASSUMED.                 ;The keyword WITH in a CLOSE
                                    ;statement is recognized but
                                    ;is not followed by one of the
                                    ;keywords NO or LOCK. The
                                    ;WITH LOCK phrase is assumed
                                    ;present.

325  DATANAME SPECIFIED WHERE
     FILENAME EXPECTED.             ;The name used in an I/O verb
                                    ;to reference a file was not a
                                    ;filename but was some other
                                    ;data-name. Fatal.

326  FILENAME MUST FOLLOW
     MODE SPEC. IN .OPEN..          ;The OPEN statement does not
                                    ;reference a valid filename
                                    ;where a filename reference is
                                    ;expected. Fatal.

327  ILLEGAL MODE SPECIFIED
     AFTER .OPEN. VERB.             ;One of the OPEN mode keywords
                                    ;INPUT, OUTPUT, I-O, or EXTEND
                                    ;is required immediately after
                                    ;the OPEN verb. None of these
                                    ;four keywords were
                                    ;recognized. Fatal.

330  .END. MUST FOLLOW .AT..
     IT IS ASSUMED.                 ;The keyword END was omitted
                                    ;in the AT END phrase of the
                                    ;READ statement. The AT END
                                    ;phrase is assumed present.

331  FILENAME MUST FOLLOW
     .READ. VERB.                   ;Either the filename was
                                    ;omitted following the READ
                                    ;verb or the data item
                                    ;following the READ verb is
                                    ;not a valid filename
                                    ;reference. Fatal.

332  DATANAME OMITTED AFTER .INTO.
     IN .READ.                      ;The data-name reference
                                    ;following the INTO keyword
                                    ;of the READ statement was
                                    ;omitted. Fatal.
```

```
333  RECORDNAME MUST FOLLOW
     .WRITE. OR .REWRITE.

334  STATEMENT IGNORED DUE
     TO ILLEGAL RECORDNAME.

335  .ADVANCING. OPTION OMITTED
     IN .WRITE.  1 ASSUMED.

336  .EOP. MUST FOLLOW .AT..
     IT IS ASSUMED.

337  DATANAME OMITTED AFTER
     .FROM.

340  .ADVANCING.  INTEGER TOO BIG.
     TRUNCATED TO 63.

341  .NO REWIND.  ILLEGAL WITH
     .IO. OR .EXTEND. MODE.

342  ILLEGAL .ADVANCING. DATANAME.
     1 IS ASSUMED.

343  FILENAME MUST FOLLOW
     .DELETE. VERB.
```

```
;The 01 record-name reference
;immediately following the
;WRITE or REWRITE verb was
;omitted.  Fatal.

;The data-name immediately
;following the WRITE or
;REWRITE verb is not a valid
;01 record-name reference.
;Fatal.

;A data-name reference, numer-
;ic integer literal reference,
;or the keyword PAGE was not
;recognized in the BEFORE/
;AFTER ADVANCING phrase of the
;WRITE statement.  A numeric
;integer literal value of 1 is
;assumed.

;The keyword EOP was omitted
;in the AT EOP phrase of the
;WRITE statement.  The AT EOP
;phrase is assumed present.

;The data-name reference
;following the FROM keyword of
;the WRITE or REWRITE
;statement was omitted.
;Fatal.

;The numeric integer in the
;BEFORE/AFTER ADVANCING phrase
;of the WRITE statement is
;greater than 63 in value.  63
;is assumed present.

;An OPEN statement with the
;I-O or EXTEND mode specified
;cannot have the NO REWIND
;phrase also specified.
;Fatal.

;The data-name in the BEFORE/
;AFTER ADVANCING phrase of the
;WRITE statement is not an
;elementary numeric integer
;data-name reference.  A
;numeric integer literal value
;of 1 is assumed.

;Either the file-name was
;omitted following the DELETE
;verb or the data item
;following the DELETE verb is
;not a valid file-name
;reference.  Fatal.
```

```
344  FILENAME MUST FOLLOW
     .START. VERB.
                                  ;Either the filename was
                                  ;omitted following the START
                                  ;verb or the data item
                                  ;following the START verb is
                                  ;not a valid filename
                                  ;reference. Fatal.

345  .LESS. OMITTED AFTER .NOT.
     IN .START. ASSUMED.
                                  ;The keyword LESS is omitted
                                  ;after NOT in the relational
                                  ;condition of the START
                                  ;statement. LESS is assumed
                                  ;present.

346  DATANAME OMITTED IN .KEY
     IS. PHRASE. ASSUMED.
                                  ;The RELATIVE KEY data-name
                                  ;for the referenced file was
                                  ;omitted in the KEY IS phrase
                                  ;of the START statement. The
                                  ;RELATIVE KEY data-name is
                                  ;assumed present.

347  RELATIONAL WORD OMITTED
     AFTER .KEY IS. PHRASE.
                                  ;None of the relational
                                  ;keywords EQUAL, GREATER, or
                                  ;NOT was recognized following
                                  ;the KEY IS phrase of the
                                  ;START statement. Fatal.

350  TERMINATOR IGNORED IN
     .IO CONTROL. PARAGRAPH.
                                  ;A clause is terminated by a
                                  ;period, but a header does not
                                  ;follow in Area A. The period
                                  ;is ignored; the compiler
                                  ;assumes it is still in the
                                  ;I-O-CONTROL paragraph.

351  TERMINATOR IGNORED IN
     .SPECIAL NAMES. PARAGRAPH.
                                  ;A clause is terminated by a
                                  ;period, but is not followed
                                  ;by a header in Area A. The
                                  ;period is ignored, and the
                                  ;compiler continues processing
                                  ;the SPECIAL-NAMES paragraph.

352  .NATIVE. MISSING IN
     SPECIALNAMES CLAUSE.
                                  ;The alphabet-name clause does
                                  ;not contain NATIVE or
                                  ;STANDARD-1. The alphabet-
                                  ;name clause is ignored.

353  SYNTAX ERROR IN .OBJECT
     COMPUTER. PARAGRAPH.
                                  ;The OBJECT-COMPUTER paragraph
                                  ;contains an unrecognizable
                                  ;word. Recovery is made by
                                  ;scanning over all words until
                                  ;a word is found in area A.

354  TERMINATOR OMITTED IN
     .OBJECT COMPUTER. PARA.
                                  ;The OBJECT-COMPUTER paragraph
                                  ;is not terminated by a
                                  ;period. Recovery is made by
                                  ;scanning over all words until
                                  ;a word is found in area A.

356  INVALID USAGE ON
     CONDITIONAL VARIABLE.
                                  ;The level 88 condition
                                  ;variable does not have
                                  ;DISPLAY or COMPUTATIONAL
                                  ;USAGE.
```

```
357  ILLEGAL SEPARATOR IN
     COBOL STATEMENT.  IGNORED.

                                  ;An illegal character was
                                  ;detected between two
                                  ;consecutive words of a COBOL
                                  ;statement.  The illegal
                                  ;character is ignored.

360  ILLEGAL CHARACTER FOUND
     WITHIN A COBOL WORD.

                                  ;Illegal characters were found
                                  ;in an alphanumeric COBOL
                                  ;word, not within an
                                  ;alphameric literal.  The
                                  ;illegal characters are
                                  ;replaced by dollar signs in
                                  ;the internal representation
                                  ;of the COBOL word.

361  UNRECOGNIZABLE TEXT FOUND
     IN COBOL STATEMENT.

                                  ;In scanning the source text,
                                  ;the compiler was unable to
                                  ;recognize an alphameric COBOL
                                  ;word (i.e., a keyword or
                                  ;user-defined word), an
                                  ;alphameric literal, or a
                                  ;numeric literal.  The error
                                  ;is not internally corrected
                                  ;and usually will propagate
                                  ;further error messages.

362  COBOL WORD BEGINS WITH
     OR ENDS IN HYPHEN.

                                  ;In attempting to recognize a
                                  ;keyword or user-defined word,
                                  ;the compiler has detected
                                  ;that the COBOL word begins
                                  ;or ends with a hyphen
                                  ;character.

363  NONNUMERIC LITERAL TOO LONG.
     TRUNCATED TO MAX.

                                  ;An alphameric literal greater
                                  ;than 132 characters in length
                                  ;is detected.  The literal is
                                  ;truncated on right, retaining
                                  ;the first 132 characters as
                                  ;the literal.

364  COBOL SOURCE LINE TOO LONG.
     TRUNCATED TO MAX.

                                  ;The indicated COBOL source
                                  ;line contains more than 65
                                  ;characters in terminal
                                  ;format.  The excess
                                  ;characters are ignored and
                                  ;only those characters in the
                                  ;printed COBOL source line are
                                  ;retained.

365  .BY. OMITTED IN REPLACING
     OPTION.  COPY IGNORED.

                                  ;The keyword BY was not found
                                  ;in this COPY...REPLACING
                                  ;statement.  The statement
                                  ;will be ignored.

366  TERMINATOR OMITTED IN
     .COPY.  IT IS ASSUMED.

                                  ;The required period
                                  ;terminating the COPY
                                  ;statement is omitted.  It is
                                  ;assumed present.
```

```
367  .LINAGE. CLAUSE DATANAME
     MUST BE AN INTEGER.
                                    ;A data-name referenced in the
                                    ;LINAGE clause of the FILE
                                    ;SECTION is defined in the
                                    ;WORKING-STORAGE SECTION with
                                    ;decimal places.

370  .LINAGE. CLAUSE DATANAME
     MUST BE UNSIGNED.
                                    ;A numeric data-name refer-
                                    ;enced in the LINAGE clause of
                                    ;the FILE SECTION is defined
                                    ;in the WORKING-STORAGE SEC-
                                    ;TION as a signed data item.

371  POSSIBLE HIGH ORDER
     RECEIVING FIELD TRUNCATION.
                                    ;Truncation of high order
                                    ;information during a MOVE
                                    ;or an arithmetic operation
                                    ;upon a receiving field is
                                    ;possible. This is an
                                    ;observation only.

372  POSSIBLE LOW ORDER
     RECEIVING FIELD TRUNCATION.
                                    ;Truncation of low order
                                    ;information during a MOVE
                                    ;or an arithmetic operation
                                    ;upon a receiving field is
                                    ;possible. This is an
                                    ;observation only.

373  PD HEADER NOT FOLLOWED
     BY AN AREA A WORD.
                                    ;The word following the
                                    ;PROCEDURE DIVISION header
                                    ;does not begin in Area A. A
                                    ;scan is made over all words
                                    ;until a word is found in Area
                                    ;A.

374  OPEN OPTIONAL FILES ONLY
     IN .INPUT. MODE.
                                    ;An OPTIONAL file can be
                                    ;OPENed in INPUT mode only.
                                    ;The compiler assumes that the
                                    ;OPTIONAL file is OPENed in
                                    ;INPUT mode.

375  EXPECTED .FILE STATUS.
     DATANAME NOT DEFINED.
                                    ;A data-name referenced in a
                                    ;FILE STATUS phrase of a
                                    ;SELECT clause in the FILE-
                                    ;CONTROL paragraph is not
                                    ;defined in the WORKING-
                                    ;STORAGE SECTION of the DATA
                                    ;DIVISION.

376  EXPECTED .VALUE OF ID.
     DATANAME NOT DEFINED.
                                    ;The data-name referenced in a
                                    ;VALUE OF ID clause of an FD
                                    ;is not defined in the
                                    ;WORKING-STORAGE SECTION of
                                    ;the DATA DIVISION.

377  EXPECTED .LINAGE. CLAUSE
     DATANAME NOT DEFINED.
                                    ;A data-name referenced in the
                                    ;LINAGE clause of the FILE
                                    ;SECTION is not defined in the
                                    ;WORKING-STORAGE SECTION of
                                    ;the DATA DIVISION.
```

```
400 .RELATIVE KEY. DATANAME
    HAS INVALID CLASS.
                                ;A data-name referenced in a
                                ;RELATIVE KEY phrase of a
                                ;SELECT clause in the FILE-
                                ;CONTROL paragraph is defined
                                ;in the WORKING-STORAGE
                                ;SECTION with non-numeric
                                ;class.

401 .RELATIVE KEY. DATANAME
    HAS INVALID USAGE.
                                ;A data-name referenced in a
                                ;RELATIVE KEY phrase of a
                                ;SELECT clause must be defined
                                ;with COMPUTATIONAL or
                                ;DISPLAY usage in the WORKING-
                                ;STORAGE section.

402 .RELATIVE KEY. DATAITEM
    IS TOO LONG.
                                ;A numeric integer data-name
                                ;referenced in a RELATIVE KEY
                                ;phrase is defined with more
                                ;than eight digits of
                                ;precision in the WORKING-
                                ;STORAGE SECTION.

403 .RELATIVE KEY. DATANAME
    MUST BE AN INTEGER.
                                ;A numeric data-name referenced
                                ;in a RELATIVE KEY phrase is
                                ;defined in the WORKING-
                                ;STORAGE SECTION with decimal
                                ;places.

404 .FILE STATUS. DATANAME
    HAS INVALID CLASS.
                                ;A data-name referenced in
                                ;the FILE STATUS phrase of a
                                ;SELECT clause is defined in
                                ;the WORKING-STORAGE SECTION
                                ;with non-alphanumeric class.

405 .FILE STATUS. DATANAME
    HAS INVALID USAGE.
                                ;A data-name referenced in a
                                ;FILE STATUS phrase of a
                                ;SELECT clause must be defined
                                ;with DISPLAY usage in the
                                ;WORKING-STORAGE SECTION

406 LENGTH OF .FILE STATUS.
    DATAITEM IS ILLEGAL.
                                ;An alphanumeric data-name
                                ;referenced in a FILE STATUS
                                ;phrase of a SELECT clause
                                ;must be defined as an
                                ;alphanumeric variable
                                ;consisting of two characters
                                ;in the WORKING-STORAGE
                                ;SECTION.

407 .VALUE OF ID. DATANAME
    HAS INVALID CLASS.
                                ;A data-name referenced in the
                                ;VALUE OF ID clause of an FD
                                ;is defined in the WORKING-
                                ;STORAGE SECTION with non-
                                ;alphanumeric class.
```

```
410  .VALUE OF ID.  DATANAME
     HAS INVALID USAGE.

411  LENGTH OF .VALUE OF ID.
     DATAITEM IS ILLEGAL.

412  .LINAGE. CLAUSE DATANAME
     HAS INVALID CLASS.

413  .LINAGE. CLAUSE DATANAME
     HAS INVALID USAGE.

414  INVALID RECEIVING OPERAND
     IN .SET..  IGNORED.

415  NO RECEIVING OPERAND
     SPECIFIED IN .SET..

416  OMITTED OR ILLEGAL OPERAND
     AFTER .TO. IN .SET..

417  ILLEGAL SYNTAX IN
     .SET. STATEMENT.

420  .BY. MUST FOLLOW .UP.
     OR .DOWN..  ASSUMED.

421  OMITTED OR ILLEGAL OPERAND
     AFTER .BY. IN .SET..

422  NO OPERANDS SPECIFIED
     IN .DISPLAY.
```

```
;A data-name referenced in a
;VALUE OF ID clause of an FD
;must be defined with DISPLAY
;USAGE in the WORKING-STORAGE
;SECTION.

;An alphameric data-name
;referenced in a VALUE OF ID
;clause of an FD must be
;defined in the WORKING-
;SECTION as an alphameric
;variable whose length L falls
;in the range 9<=L<=40
;characters.

;A data-name referenced in the
;LINAGE clause of the FILE
;SECTION is defined in the
;WORKING-STORAGE SECTION with
;non-numeric class.

;A data-name referenced in the
;LINAGE clause of the FILE
;SECTION must be defined with
;COMPUTATIONAL usage in the
;WORKING-STORAGE SECTION.

;A receiving operand of a
;SET statement is invalid.
;Fatal.

;No receiving operands are
;specified in a SET statement.
;Fatal.

;A SET statement has no valid
;sending operand.  Fatal.

;The words TO, UP or DOWN do
;not follow the receiving
;operands of a SET statement.
;Fatal.

;The keyword BY does not
;follow the word UP or DOWN in
;a SET statement.  BY is
;assumed present.

;The operand following the UP
;BY or DOWN BY phrase in a SET
;statement is invalid or
;omitted.  Fatal.

;No operands to be displayed
;were recognized by the
;compiler in this DISPLAY
;statement.  Fatal.
```

```
423  SETTING INDEX NAME OUT
     OF RANGE.  .SET. IGNORED.

424  .IF. TRUE PATH OMITTED.
     ASSUME .NEXT SENTENCE.

425  CONFLICTING SIGN SYMBOLS
     IN PICTURE STRING.

426  ZERO SUPPRESSION CONFLICTS
     IN PICTURE STRING.

427  ILLEGAL CHARACTER IN
     THE PICTURE STRING.

430  .BLANK WHEN ZERO.  CONFLICTS
     WITH ZERO SUPPRESS.

431  PARENTHESIZED SPECIFIER
     EXCEEDS FOUR DIGITS
```

```
;A SET statement is attempting
;to set an index name using a
;literal that is too large.
;Fatal.

;The true path code is omitted
;from the IF statement. NEXT
;SENTENCE is assumed as the
;true path of the IF
;statement.

;The compiler has recognized
;both the + and - sign symbols
;in this PICTURE string. The
;compiler ignores the user-
;supplied PICTURE and declares
;the data-name alphanumeric
;with a "PICTURE X"
;declaration.

;The compiler has recognized
;both the Z and * zero
;suppression symbols in this
;PICTURE string. The compiler
;ignores the user-supplied
;PICTURE and declares the
;data-name alphanumeric with a
;"PICTURE X" declaration.

;A character which is not in
;the PICTURE string character
;set is recognized in this
;PICTURE by the compiler. The
;compiler ignores the user-
;supplied PICTURE and declares
;the data-name alphanumeric
;with a "PICTURE X"
;declaration.

;A BLANK WHEN ZERO clause has
;been recognized with a zero
;suppression field specified
;in the PICTURE string. The
;compiler ignores the BLANK
;WHEN ZERO clause and
;continues with its
;processing.

;The specification contained
;inside parentheses of a
;PICTURE string exceeds four
;digits in length. The
;compiler ignores the extra
;digits.
```

```
432  SPECIFIER MISSING INSIDE
     PARENTHESES.
                                  ;The specification contained
                                  ;inside parentheses of a
                                  ;PICTURE string is missing.
                                  ;The compiler ignores the
                                  ;user-supplied PICTURE and
                                  ;declares the dataname
                                  ;alphanumeric with a "PICTURE
                                  ;X" declaration.

433  ILLEGAL SYMBOL PRECEDES
     LEFT PAREN. IN PICTURE.
                                  ;The compiler has recognized
                                  ;an S, V, C, R, D, or "."
                                  ;character preceding a left
                                  ;parenthesis in a PICTURE
                                  ;string. The error is ignored
                                  ;and processing continues.

502  INTEGER 1 BEYOND  AREA A
     TREATED AS LEVEL  NUMBER.
                                  ;An 01 level item was detected
                                  ;beyond Area A and accepted as
                                  ;if in Area A.

503  MULTIPLE PICTURES FOR
     SAME ITEM.  LAST USED.
                                  ;A data item has more than 1
                                  ;PICTURE clause. The compiler
                                  ;used the last PICTURE clause
                                  ;specified.

504  CLOSING PARENTHESIS MISSING
     IN PICTURE.
                                  ;The right parenthesis is
                                  ;missing in the PICTURE
                                  ;string. The compiler uses
                                  ;the last four digits of the
                                  ;PICTURE string.

506  PICTURE EXCEEDS 30 CHARACTERS.
     PIC X ASSUMED.
                                  ;The PICTURE string exceeds 30
                                  ;characters after expansion.
                                  ;The compiler ignores the
                                  ;user-supplied PICTURE and
                                  ;declares the data-name
                                  ;alphanumeric with a "PICTURE
                                  ;X" declaration.

507  SPECIFIER OMITTED BEFORE
     LEFT PAREN. IN PIC.
                                  ;The first character of a
                                  ;PICTURE string is a left
                                  ;parenthesis. The compiler
                                  ;ignores the user-supplied
                                  ;PICTURE and declares the
                                  ;data-name alphanumeric with a
                                  ;"PICTURE X" declaration.

510  SECTION NO. GREATER THAN
     49 TREATED AS 49.
                                  ;A segment number greater than
                                  ;49 follows the word SECTION.
                                  ;The segment is treated as if
                                  ;it were 49.

511  INVALID ITEM LENGTH IN
     PARENTHESES OF PICTURE.
                                  ;The parenthesized length
                                  ;specifier in a PICTURE
                                  ;contains non-numeric
                                  ;characters. The compiler
                                  ;ignores the user-supplied
                                  ;PICTURE and declares the
                                  ;data-name alphanumeric with a
                                  ;"PICTURE X" declaration.
```

```
514  MULTIPLE FLOATING FIELDS
     IN NUMERIC EDIT ITEM.
                                ;The PICTURE string contains
                                ;multiple floating fields.
                                ;The compiler ignores the
                                ;user-supplied PICTURE and
                                ;declares the data-name
                                ;alphanumeric with a "PICTURE
                                ;X" declaration.

515  MULTIPLE ZERO SUPPRESS
     FIELDS IN PICTURE STRING.
                                ;Multiple zero suppression
                                ;fields are detected in
                                ;PICTURE string. The compiler
                                ;ignores the user-supplied
                                ;PICTURE and declares the
                                ;data-name alphanumeric with a
                                ;"PICTURE X" declaration.

516  ZERO SUPPRESSION ILLEGAL
     WITH FLOATING FIELD.
                                ;The PICTURE string contains
                                ;both floating and zero
                                ;suppression fields. The
                                ;compiler ignores the user-
                                ;supplied PICTURE and declares
                                ;the data-name alphanumeric
                                ;with a "PICTURE X"
                                ;declaration.

517  ILLEGAL SYNTAX IN
     PICTURE STRING.
                                ;The PICTURE string is not
                                ;specified correctly according
                                ;to the rules of PICTURE
                                ;string syntax. The compiler
                                ;ignores the user-supplied
                                ;PICTURE and declares the
                                ;data-name alphanumeric with a
                                ;"PICTURE X" declaration.

520  MULTIPLE DECIMAL POINTS
     IN PICTURE.
                                ;The PICTURE string contains
                                ;multiple decimal point
                                ;specifications (V's, P's, or
                                ;periods). The compiler
                                ;ignores the user-supplied
                                ;PICTURE and declares the
                                ;data-name alphanumeric with a
                                ;"PICTURE X" declaration.

522  INVALID USAGE. IGNORED.
                                ;The USAGE clause contains an
                                ;invalid word. The compiler
                                ;ignores the entire USAGE
                                ;clause.

523  MULTIPLE USAGE CLAUSES.
     LAST USED.
                                ;The defined dataname has
                                ;multiple USAGE clauses
                                ;specified. The last USAGE
                                ;clause specified is used by
                                ;the compiler.

524  MULTIPLE OCCURS CLAUSES.
     LAST USED.
                                ;The defined dataname has
                                ;multiple OCCURS clauses
                                ;specified. The compiler uses
                                ;the last OCCURS clause
                                ;specified.
```

```
525  OCCURS SPECIFICATION ERROR.
     1 ASSUMED.
                                    ;The integer entry of the
                                    ;OCCURS clause is either
                                    ;non-numeric or non-integer or
                                    ;does not lie in the range 1
                                    ;to 4095. The compiler
                                    ;assumes an integer value
                                    ;of 1.

526  ILLEGAL WORD AS DATANAME.
     ASSUME FILLER.
                                    ;A reserved word other than
                                    ;FILLER was seen after a level
                                    ;number in a data description.
                                    ;The compiler assumes the word
                                    ;to be FILLER.

527  INVALID INDEX NAME.
     IGNORED.
                                    ;The compiler did not
                                    ;recognize a valid index name
                                    ;in the INDEXED BY phrase.
                                    ;The compiler ignores the
                                    ;INDEXED BY phrase.

530  USAGE OPTION NOT YET
     IMPLEMENTED. IGNORED.
                                    ;The compiler detected COMP-1
                                    ;in the USAGE clause. This
                                    ;option is not implemented and
                                    ;is ignored. The default
                                    ;USAGE of DISPLAY is used by
                                    ;the compiler.

531  TERMINATOR OMITTED AFTER
     DATAITEM DESCRIPTION.
                                    ;A data item description entry
                                    ;in the DATA DIVISION is not
                                    ;terminated by a period. The
                                    ;compiler assumes the period
                                    ;is present and continues
                                    ;processing.

532  INVALID SIGN IN NUMERIC
     PICTURE.
                                    ;The sign character S is
                                    ;detected in a position other
                                    ;than the leading character
                                    ;position of a numeric PICTURE
                                    ;string. The compiler ignores
                                    ;the user-supplied PICTURE and
                                    ;declares the dataname
                                    ;alphanumeric with a "PICTURE
                                    ;X" declaration.

533  PICTURE CLAUSE OMITTED ON
     ELEMENTARY ITEM.
                                    ;An elementary item is
                                    ;recognized with its PICTURE
                                    ;clause omitted in the
                                    ;description. The compiler
                                    ;declares the dataname either
                                    ;alphanumeric or numeric.

534  NUMERIC ITEM EXCEEDS 18
     DIGIT MAX. TRUNCATED.
                                    ;A numeric field is defined in
                                    ;this PICTURE with more than
                                    ;18 digits of precision. The
                                    ;numeric field is truncated to
                                    ;18 digits.
```

```
535  COMP ITEM EXCEEDS 18
     DIGITS.  ASSIGN 4 WORDS.
                                  ;A COMPUTATIONAL data item
                                  ;exceeds 18 digits in its
                                  ;specification. The compiler
                                  ;truncates it and allocates
                                  ;four words for its runtime
                                  ;storage.

536  INDEX ITEM HAS
     ILLEGAL CLAUSE.
                                  ;The compiler recognized a
                                  ;JUSTIFIED, SYNCHRONIZED,
                                  ;VALUE, PICTURE, or SIGN
                                  ;clause on a data item
                                  ;description which has INDEX
                                  ;USAGE. This is illegal. The
                                  ;compiler ignores the
                                  ;offensive clause.

537  NUMERIC VALUE FOR
     DISPLAY ITEM.  IGNORED.
                                  ;The VALUE clause specifies
                                  ;numeric value initialization
                                  ;for a non-numeric data item
                                  ;which is defined with DISPLAY
                                  ;USAGE. This is illegal. The
                                  ;VALUE clause is ignored.

540  VALUE TOO LONG.
     TRUNCATED.
                                  ;The length of the non-numeric
                                  ;literal in the VALUE clause
                                  ;is longer than the associated
                                  ;data item. The literal is
                                  ;truncated on the right to fit
                                  ;in the storage allocated to
                                  ;the dataitem.

541  CLAUSE DUPLICATION.  IGNORED.
                                  ;This clause has been
                                  ;previously recognized for
                                  ;this item. The duplicate
                                  ;clause is ignored.

542  INVALID WORD IN .BLANK
     WHEN ZERO..  IGNORED.
                                  ;The keyword ZERO was not
                                  ;recognized in the BLANK WHEN
                                  ;ZERO clause. The entire
                                  ;clause is ignored.

543  LEVEL NUMS UNEQUAL IN
     REDEFINES.  CLAUSE IGNORED.
                                  ;A REDEFINES clause attempts
                                  ;to redefine two items of
                                  ;different level numbers. The
                                  ;REDEFINES clause is ignored.

545  LEVEL ILLEGAL AFTER 77.
     TREATED AS 01.
                                  ;An invalid level number
                                  ;(02-49) follows a 77 level
                                  ;item. The 77 level item is
                                  ;treated as an 01 level item.
                                  ;This action may propagate
                                  ;further diags if it is not a
                                  ;valid group item.

550  REDEFINING LENGTH SHOULD
     MATCH ORIGINAL LENGTH.
                                  ;The length of a non-01 level
                                  ;redefines item is not the
                                  ;same as the length of the
                                  ;item it REDEFINES. The new
                                  ;length is used.
```

```
551  REDEFINITION OF .OCCURS.
     ITEM.  IGNORED.

552  PROCESSING RESUMES AFTER
     BAD FD.

553  INVALID CLAUSE KEYWORD.
     OTHER CLAUSES SKIPPED.

554  INVALID WORD FOLLOWING
     .VALUE..  IGNORED.

555  VALUE CONFLICT.
     GROUP VALUE USED.

556  LEVEL NUMBER OMITTED.
     ITEM IGNORED.

557  NO VALUE AFTER CONDITION
     NAME.  88 IGNORED.

561  .NO. MISSING IN
     ADVANCING PHRASE.  ASSUMED.
```

```
;Data items with the OCCURS
;clause cannot be REDEFINED.
;The REDEFINES clause is
;ignored.

;Prior to issuing this
;message, the compiler had
;discovered bad syntax in the
;FD of the FILE SECTION.  The
;compiler at that time issued
;an error message identifying
;the syntax error.  Then the
;compiler went into recovery
;mode attempting to recognize
;another FD, the WORKING-
;STORAGE SECTION header or the
;PROCEDURE DIVISION.  Upon
;recognizing one of these
;three language elements,
;the compiler issues this
;diagnostic indicating that
;normal processing resumes.

;A reserved clause keyword was
;expected at this point in a
;data item description entry of
;the DATA DIVISION, but was
;not recognized by the
;compiler.  The compiler skips
;to the next level number
;data item description.

;The VALUE clause contains an
;invalid word for this data
;description.  The entire
;VALUE clause is ignored.

;This VALUE clause assigns
;a value to an item
;subordinate to a group item
;that also has a VALUE clause.
;The subordinate VALUE clause
;is ignored.

;The level number has been
;omitted in a data item
;description.  All the source
;text is ignored up to and
;including the next period.

;An 88 level condition-name
;has no VALUE clause
;specified.  The entire 88
;level data item is ignored.

;The keyword NO is missing in
;the ADVANCING phrase of the
;DISPLAY statement.  NO is
;assumed present.
```

```
562  .ADVANCING. MISSING AFTER          ;The keyword ADVANCING is
     .NO.. ASSUMED.                     ;missing in the ADVANCING
                                        ;phrase of the DISPLAY
                                        ;statement. ADVANCING is
                                        ;assumed present.

563  DUPLICATE DATANAME.                ;In the DATA DIVISION, the
     FIRST USED.                        ;same dataname is defined more
                                        ;than once. Qualification is
                                        ;not yet implemented. The
                                        ;first definition of the
                                        ;dataname is used.

564  ILLEGAL PARAGRAPH HEADER           ;An illegal paragraph header
     ID DIV. PAR IGNORED.               ;appears in the IDENTIFICATION
                                        ;DIVISION. The paragraph is
                                        ;ignored.

565  ILLEGAL PARAGRAPH HEADER           ;An illegal paragraph header
     ENV DIV. PAR IGNORED.              ;appears in the ENVIRONMENT
                                        ;DIVISION. The paragraph is
                                        ;ignored.

566  NUMERIC LITERAL ILLEGAL            ;A numéric literal is illegal
     ON GROUP ITEM.  IGNORED.           ;in the VALUE clause of a
                                        ;group item. The VALUE clause
                                        ;is ignored.

567  .ENVIRONMENT. NOT FOLLOWED         ;The word ENVIRONMENT is
     BY .DIVISION..                     ;not followed by the word
                                        ;DIVISION.  DIVISION is
                                        ;assumed present.

570  TERMINATOR MISSING AFTER           ;The DATA DIVISION header is
     .DATA DIVISION. HEADER.            ;not followed by a period.
                                        ;The period is assumed present
                                        ;and processing continues.

571  TERMINATOR MISSING AFTER           ;A paragraph header in the
     PARAGRAPH HEADER.                  ;IDENTIFICATION or ENVIRONMENT
                                        ;DIVISION is not terminated by
                                        ;a period. The period is
                                        ;assumed present and
                                        ;processing continues.

572  LEVEL 66 NOT IMPLEMENTED.          ;A level 66 (RENAMES) data
     IGNORED.                           ;item has been recognized by
                                        ;the compiler. Level 66 items
                                        ;are not implemented by the
                                        ;compiler. The entire data
                                        ;description entry is ignored.

573  .SECTION. OMITTED FROM             ;An ENVIRONMENT DIVISION
     SECTION HEADER.                    ;section name is not followed
                                        ;by the word SECTION. The
                                        ;error is ignored.
```

```
574  TERMINATOR MISSING AFTER
     SECTION HEADER.
                                  ;An ENVIRONMENT DIVISION
                                  ;section header is not
                                  ;terminated by a period. The
                                  ;error is ignored.

575  IDENTIFICATION DIVISION
     WAS OMITTED.
                                  ;The program contains no
                                  ;IDENTIFICATION DIVISION. The
                                  ;compiler assigns a default
                                  ;PROGRAM-ID of COBL00 and
                                  ;continues processing at the
                                  ;DATA DIVISION header.

576  NO IDENTIFICATION OR
     ENVIRONMENT DIVISION FOUND.
                                  ;The program contains no
                                  ;IDENTIFICATION or ENVIRONMENT
                                  ;DIVISIONs. The compiler
                                  ;assigns a default PROGRAM-ID
                                  ;of COBL00 and continues
                                  ;processing at the DATA
                                  ;DIVISION header.

577  IDENTIFICATION DIVISION HEADER
     OMITTED.
                                  ;The program contains no
                                  ;IDENTIFICATION DIVISION
                                  ;header. The compiler resumes
                                  ;processing at the next
                                  ;paragraph header.

600  ILLEGAL LEVEL NUMBER.
     TREAT AS 01.
                                  ;This level number is not an
                                  ;01-49, 66, 77, or 88 level
                                  ;number. The level number is
                                  ;assumed to be 01.

601  TERMINATOR MISSING AFTER
     ENV  DIV HEADER.
                                  ;The ENVIRONMENT DIVISION
                                  ;header is not terminated by a
                                  ;period. The period is assumed
                                  ;present and processing
                                  ;continues.

602  .DATA. NOT FOLLOWED BY
     .DIVISION.
                                  ;The word DATA is not
                                  ;followed by the word
                                  ;DIVISION. DIVISION is
                                  ;assumed present.

603  ENVIRONMENT DIVISION HEADER
     OMITTED.
                                  ;The program contains no
                                  ;ENVIRONMENT DIVISION header.
                                  ;The compiler resumes
                                  ;processing at the next
                                  ;paragraph header.

604  NO VALID HEADERS FOUND.
                                  ;Over 50 words have been been
                                  ;scanned without finding the
                                  ;word IDENTIFICATION. The
                                  ;compiler assumes that its
                                  ;input is not a COBOL source
                                  ;program. Compilation is
                                  ;aborted.

605  .IDENTIFICATION. NOT FOLLOWED
     BY .DIVISION..
                                  ;The word IDENTIFICATION is
                                  ;not followed by the word
                                  ;DIVISION. DIVISION is
                                  ;assumed present.
```

```
606  TERMINATOR OMITTED AFTER
     .ID DIVISION. HEADER.          ;The IDENTIFICATION DIVISION
                                    ;header is not terminated by a
                                    ;period.  The period is
                                    ;assumed present and
                                    ;processing continues.

607  .PROGRAMID. EXPECTED AFTER
     DIVISION HEADER.               ;The IDENTIFICATION DIVISION
                                    ;header is not followed by
                                    ;the PROGRAM-ID paragraph.
                                    ;The error is ignored and
                                    ;processing continues.

610  TERMINATOR OMITTED AFTER
     .PROGID. PARA HEADER.          ;The PROGRAM-ID paragraph-name
                                    ;is not terminated by a
                                    ;period.  The period is
                                    ;assumed present and
                                    ;processing continues.

611  INVALID PROGRAM NAME IN
     .PROGRAM ID. PARAGRAPH.        ;The program name of the
                                    ;PROGRAM-ID paragraph contains
                                    ;an invalid character or
                                    ;exceeds nine characters in
                                    ;length.  The error is ignored
                                    ;and processing continues.

612  TOO MANY FILES FOR LUNS
     OR TEMPORARY SPACE.            ;The compiler has discovered
                                    ;either that more than 30
                                    ;files are declared in the
                                    ;program or that more than 30
                                    ;SAME RECORD AREA clauses are
                                    ;specified in the program.
                                    ;The compiler imposes a limit
                                    ;of 30 in both cases, because
                                    ;the associated compiler and/
                                    ;or object time table space is
                                    ;exhausted.

613  INVALID WORD SUSPENDS
     PROCESSING.  SCAN FORWARD.     ;An unidentifiable word is
                                    ;found where a verb is
                                    ;expected.  A scan is made to
                                    ;a verb, or period, or word in
                                    ;Area A.

614  PROCESSING RESTARTS ON
     VERB.                          ;Due to a previous syntax
                                    ;error, the compiler went into
                                    ;recovery mode looking for the
                                    ;next verb, period, or Area A
                                    ;word upon which to resume
                                    ;compilation.  The compiler
                                    ;has recognized a verb and
                                    ;resumes normal compilation at
                                    ;this point.  This message is
                                    ;an observation only.
```

```
615  PROCESSING RESTARTS ON
     PROCEDURE NAME.
                                  ;Due to a previous syntax
                                  ;error, the compiler went into
                                  ;recovery mode looking for the
                                  ;next verb, period, or Area A
                                  ;word upon which to resume
                                  ;compilation. The compiler
                                  ;has recognized an Area A word
                                  ;and resumes compilation at
                                  ;this point. This message is
                                  ;an observation only.

616  PROCESSING RESTARTS AFTER
     TERMINATOR
                                  ;Due to a previous syntax
                                  ;error, the compiler went into
                                  ;recovery mode looking for the
                                  ;next verb, period, or Area A
                                  ;word upon which to resume
                                  ;compilation. The compiler
                                  ;has recognized a period and
                                  ;resumes normal compilation on
                                  ;the word following the
                                  ;period. This is an
                                  ;observation only.

620  PARAGRAPH TERMINATOR
     ASSUMED OMITTED.
                                  ;A paragraph was terminated
                                  ;without a period. The period
                                  ;is assumed and processing
                                  ;continues.

621  .LINAGE. FOR RELATIVE
     FILE. CLAUSE IGNORED.
                                  ;The LINAGE clause must not be
                                  ;specified for a file which
                                  ;has RELATIVE organization.
                                  ;The LINAGE clause is ignored.

622  TERMINATOR MISSING AFTER
     PROCEDURE NAME.
                                  ;A section or paragraph
                                  ;name is not terminated by a
                                  ;period. The period is
                                  ;assumed present and
                                  ;and processing continues.

623  .ELSE DOES NOT HAVE
     ASSOCIATED .IF.. IGNORED.
                                  ;The word ELSE has no
                                  ;associated IF statement. The
                                  ;ELSE is ignored.

624  VERB EXPECTED TO FOLLOW
     ELSE.. .ELSE. IGNORED.
                                  ;A sentence ends with the
                                  ;word ELSE. The ELSE is
                                  ;ignored.

625  .JUSTIFY. WITH NUMERIC OR
     EDITED ITEM. IGNORED.
                                  ;The JUSTIFIED clause must not
                                  ;be specified for a numeric or
                                  ;numeric-edited dataitem. The
                                  ;JUSTIFIED clause is ignored.

626  .BLANK WHEN ZERO. ILLEGALLY
     SPECIFIED. IGNORED.
                                  ;The BLANK WHEN ZERO clause
                                  ;must be specified only for a
                                  ;numeric or numeric-edited
                                  ;data item. The clause is
                                  ;ignored.
```

```
627  REDEFINED ITEM NOT FOUND.
     .REDEFINES. IGNORED.
                                  ;The second data-name in a
                                  ;REDEFINES clause is not a
                                  ;data-name previously defined.
                                  ;The REDEFINES clause is
                                  ;ignored.

630  .REDEFINES. MUST FOLLOW
     DATA NAME.  IGNORED.
                                  ;The REDEFINES keyword appears
                                  ;in the wrong position of a
                                  ;data description entry.  The
                                  ;REDEFINES clause is ignored.

631  DEPTH OF NESTED .IF.
     EXCEEDS LIMIT.
                                  ;A nested IF statement has
                                  ;exceeded the maximum depth of
                                  ;30 levels.  The compiler
                                  ;ignores nesting beyond this
                                  ;depth of nesting.

632  PROCEDURE NAME DUPLICATED.
                                  ;In the PROCEDURE DIVISION,
                                  ;the same procedure-name is
                                  ;defined more than once.  The
                                  ;compiler uses the first
                                  ;occurrence of the name and
                                  ;ignores the duplicate entry.
                                  ;All references to the
                                  ;procedure-name will refer to
                                  ;the first definition of the
                                  ;procedure-name.

633  UNDEFINED PROCEDURE NAME.
                                  ;In the PROCEDURE DIVISION, a
                                  ;reference is made to an
                                  ;undefined paragraph or
                                  ;section name.  Fatal.

634  FILENAME LITERAL TOO LONG.
     TRUNCATED.
                                  ;A file specification in the
                                  ;ASSIGN clause exceeds 40
                                  ;characters in length.  It is
                                  ;truncated to 40 characters.

635  BAD PROCEDURE NAME AFTER
     .GO TO.
                                  ;The word after GO TO is an
                                  ;invalid procedure-name.
                                  ;Fatal.

636  INVALID INTEGER OR
     DATANAME.
                                  ;In the LINAGE clause, the
                                  ;compiler failed to recognize
                                  ;a non-negative integer
                                  ;literal or a numeric integer
                                  ;data-name.  This phrase of the
                                  ;LINAGE clause is ignored.

637  .GO TO. HAS MULTIPLE
     PROCEDURE NAMES.
                                  ;A simple GO TO statement
                                  ;(i.e., without the DEPENDING
                                  ;ON phrase) has more than one
                                  ;procedure-name.  The GO TO
                                  ;statement is ignored.
```

```
640  INVALID WORD FOLLOWS
     .DATA DIVISION.
                                  ;The word following the DATA
                                  ;division header either does
                                  ;not start in Area A or is not
                                  ;one of the reserved words
                                  ;FILE, WORKING-STORAGE, or
                                  ;PROCEDURE. The compiler goes
                                  ;into recovery mode skipping
                                  ;all source text until one of
                                  ;the keywords FILE, WORKING-
                                  ;STORAGE or PROCEDURE
                                  ;is recognized.

641  INVALID WORD IN FILE
     SECTION. SCAN FORWARD.
                                  ;An invalid word was detected
                                  ;in the FILE SECTION where
                                  ;the keyword FD is expected.
                                  ;The compiler goes into
                                  ;recovery mode skipping all
                                  ;source text until one of the
                                  ;keywords FD, WORKING-STORAGE,
                                  ;or PROCEDURE is recognized.

642  .VALUE OF ID. WITH
     .OMITTED LABELS. IGNORED.
                                  ;The VALUE OF ID clause is
                                  ;specified with the LABEL
                                  ;RECORDS ARE OMITTED clause.
                                  ;This is illegal. The VALUE
                                  ;OF ID clause is ignored.

643  .SECTION. EXPECTED AFTER
     HEADER WORD.
                                  ;The keyword SECTION is
                                  ;omitted after the word FILE
                                  ;or WORKING-STORAGE. SECTION
                                  ;is assumed present and
                                  ;processing continues.

644  TERMINATOR EXPECTED AFTER
     SECTION HEADER.
                                  ;The FILE SECTION or WORKING-
                                  ;STORAGE SECTION header is not
                                  ;terminated by a period. The
                                  ;period is assumed and
                                  ;processing continues.

645  OTS BUFFER SPACE OVERFLOW.
                                  ;The total buffer space
                                  ;required by the files in
                                  ;the program has exceeded
                                  ;32K characters. This cannot
                                  ;be accommodated by the
                                  ;object-time system.

646  .OF. OR .ID. MISSING
     IN .VALUE OF ID..
                                  ;One or both of the keywords
                                  ;OF or ID is omitted in the
                                  ;VALUE OF ID clause. Their
                                  ;presence is assumed and
                                  ;processing continues.
```

```
647  ILLEGAL WORD IN AREA A.
     SCAN FORWARD.
                                  ;In the WORKING-STORAGE
                                  ;SECTION, an 01 or 77
                                  ;level number or the
                                  ;PROCEDURE keyword was
                                  ;expected in Area A, but was
                                  ;not recognized. The compiler
                                  ;goes into recovery mode
                                  ;skipping source text until
                                  ;one of the three language
                                  ;elements aforementioned is
                                  ;recognized in Area A.

650  GROUP LEVEL .VALUE.
     DISALLOWED.
                                  ;The VALUE clause on this
                                  ;group item is not permitted
                                  ;because a subordinate
                                  ;elementary item has a non-
                                  ;DISPLAY usage specified or
                                  ;has a SYNCHRONIZED clause
                                  ;specified. The group VALUE
                                  ;clause is ignored.

652  RELATIVE FILE IN .MULTIPLE.
     FILE TAPE. CLAUSE.
                                  ;In the I-O CONTROL paragraph,
                                  ;the MULTIPLE FILE TAPE clause
                                  ;is specified for a file whose
                                  ;organization is RELATIVE.
                                  ;This is illegal. The
                                  ;MULTIPLE FILE TAPE clause is
                                  ;ignored for this file.

653  .VALUE. CLAUSE ILLEGAL IN
     FILE SECTION.
                                  ;A VALUE clause is specified
                                  ;for a data description entry
                                  ;given in the FILE SECTION.
                                  ;This is illegal. The VALUE
                                  ;clause is ignored.

654  SYNTAX ERROR IN CURRENCY
     CLAUSE.
                                  ;The alphanumeric literal
                                  ;expected in the CURRENCY SIGN
                                  ;clause of the SPECIAL-NAMES
                                  ;paragraph is omitted. The
                                  ;clause is ignored and the
                                  ;currency sign defaults to the
                                  ;dollar sign.

655  ILLEGAL CURRENCY SIGN.
                                  ;The alphanumeric literal in
                                  ;the CURRENCY SIGN clause is
                                  ;not allowed as the currency
                                  ;sign either because the
                                  ;literal is longer than one
                                  ;character or because it is an
                                  ;invalid COBOL currency sign.
                                  ;The CURRENCY SIGN clause is
                                  ;ignored and the currency sign
                                  ;defaults to the dollar sign.
```

```
656  SPECIALNAMES CLAUSE INVALID.          ;An unrecognizable word
                                           ;appears in a position where a
                                           ;SPECIAL-NAMES paragraph
                                           ;clause keyword is expected.
                                           ;All source text is skipped up
                                           ;to the next recognizable
                                           ;keyword.

657  SYNTAX ERROR IN                       ;The keyword COMMA is omitted
     DECIMALPOINT CLAUSE.                  ;in the DECIMAL-POINT IS COMMA
                                           ;clause of the SPECIAL-NAMES
                                           ;paragraph. The clause is
                                           ;ignored.

660  .AFTER. MISSING IN                    ;The keyword AFTER is omitted
     .USE. STATEMENT.  ASSUMED.            ;in the USE statement. AFTER
                                           ;is assumed present and
                                           ;processing continues.

661  NO .ERROR. OR .EXCEPTION.             ;One of the keywords ERROR or
     IN .USE. ASSUMED.                     ;EXCEPTION is omitted in the
                                           ;USE statement. The missing
                                           ;keyword is assumed present
                                           ;and processing continues.

662  NO KNOWN CLAUSES IN                   ;The SPECIAL-NAMES PARAGRAPH
     SPECIALNAMES.                         ;contains no valid clauses.
                                           ;This is an observation only.

663  REDUNDANT .USE. COVERAGE.             ;Multiple USE statements have
     PREV. .USE. IGNORED.                  ;referenced the same file.
                                           ;The last USE statement
                                           ;specified is then applied to
                                           ;the referenced file.

664  UNKNOWN OPEN MODE IN                  ;An unrecognizable OPEN mode
     .USE. STATEMENT.                      ;option was specified in the
                                           ;USE statement. Fatal.

665  GROUP ITEM HAS BEEN CALLED            ;A FILLER item cannot have any
     FILLER.                               ;elementary items subordinate
                                           ;to it.

666  MISSING ENVIRONMENT                   ;The program does not contain
     DIVISION.                             ;an ENVIRONMENT DIVISION. The
                                           ;compiler skips to the DATA
                                           ;DIVISION and continues
                                           ;processing.

667  DIVISION BY ZERO.                     ;The divisor of a DIVIDE
                                           ;statement is a literal of
                                           ;zero value. The error is
                                           ;ignored.

670  VALUE NOT PERMITTED WITH              ;A VALUE clause is recognized
     THIS ITEM.                            ;in a data description entry
                                           ;which contains a REDEFINES or
                                           ;an OCCURS clause. This is
                                           ;illegal. The VALUE clause is
                                           ;ignored.
```

```
671  INVALID CONSTANT OR
     LITERAL FOLLOWING .ALL..

                                    ;The reserved word ALL is not
                                    ;followed by a non-numeric
                                    ;literal or a figurative
                                    ;constant. Thus, this is not
                                    ;a valid ALL literal. ALL is
                                    ;ignored and processing
                                    ;continues.

672  BAD FILENAME IN .USE.
     STATEMENT.

                                    ;An unrecognizable word
                                    ;appears where a filename is
                                    ;expected in the USE
                                    ;statement. Fatal.

673  FILE NOT CLOSED.

                                    ;The referenced file was
                                    ;OPENed but there was no CLOSE
                                    ;statement detected for this
                                    ;file in the program.

675  FILE COVERED BY CONFLICTING
     USE PROCEDURE.

                                    ;There was more than one
                                    ;conflicting USE procedure
                                    ;specified for the referenced
                                    ;file. Fatal.

677  SUPPLIED VALUE INVALID FOR
     NUM ITEM. IGNORED.

                                    ;The VALUE clause specifies
                                    ;invalid value initialization
                                    ;for a numeric data item. The
                                    ;compiler ignores the VALUE
                                    ;clause.

700  FILE ACCESSED BY VERB
     REQUIRING RELATIVE ORG.

                                    ;A file whose organization is
                                    ;SEQUENTIAL is referenced by
                                    ;the START or DELETE verbs or
                                    ;by an I/O verb which has the
                                    ;INVALID KEY clause specified.
                                    ;This is illegal. In all
                                    ;these cases, the referenced
                                    ;file must have RELATIVE
                                    ;organization. Fatal.

701  FILE ACCESSED BY VERB REQ.
     SEQUENTIAL ORG.

                                    ;A file whose organization is
                                    ;RELATIVE is referenced by an
                                    ;I/O verb which has the AT EOP
                                    ;or ADVANCING clauses
                                    ;specified. This is illegal.
                                    ;The referenced file must have
                                    ;SEQUENTIAL organization.
                                    ;Fatal.

702  VERB NOT IMPLEMENTED.

                                    ;An ANS 1974 COBOL verb
                                    ;appears that is not
                                    ;implemented in this release
                                    ;of the compiler. The
                                    ;compiler scans to another
                                    ;verb, period, or word in Area
                                    ;A.

704  OCCURS ILLEGAL FOR 01
     OR 77 ITEM. IGNORE.

                                    ;An OCCURS clause is specified
                                    ;for an 01 or 77 level
                                    ;data-name. The compiler
                                    ;ignores the OCCURS clause.
```

```
705  .ACCEPT FROM. OBJECT NOT
     IN SPECIALNAMES.
                                  ;The mnemonic name used in the
                                  ;ACCEPT statement was not
                                  ;defined in the SPECIAL-NAMES
                                  ;paragraph. Fatal.

706  ACCEPT IDENTIFIER INVALID.
                                  ;The word following the ACCEPT
                                  ;verb is not a data-name or is
                                  ;a data-name which has non-
                                  ;DISPLAY usage or invalid
                                  ;class. Fatal.

707  VERB OR COND. CLAUSE
     CONFLICTS WITH FILE ACCESS.
                                  ;There is a conflict between
                                  ;the ACCESS MODE of the
                                  ;referenced file and the I/O
                                  ;verbs and/or condition
                                  ;clauses which reference this
                                  ;file. Fatal.

710  DATANAME AFTER .GO
     DEPENDING. INVALID.
                                  ;The word following the
                                  ;DEPENDING ON phrase of the GO
                                  ;TO statement is not a
                                  ;dataname or is a data-name
                                  ;which has INDEX usage. This
                                  ;is illegal. Fatal.

711  INVALID CLASS OF DATANAME
     AFTER .GO DEPENDING.
                                  ;The data-name following the
                                  ;DEPENDING ON phrase of the GO
                                  ;TO statement is not a numeric
                                  ;data-name or is a numeric,
                                  ;non-integer data-name. This
                                  ;is illegal. Fatal.

712  .DISPLAY UPON. OBJECT
     NOT IN SPECIALNAMES.
                                  ;The mnemonic name used in the
                                  ;DISPLAY statement was not
                                  ;defined in the SPECIAL-NAMES
                                  ;paragraph. Fatal.

713  .DISPLAY. OPERAND IS INVALID.
                                  ;A data item in the DISPLAY
                                  ;statement has invalid class
                                  ;or USAGE.

714  MISSING OR INVALID OPERAND.
     OF .MULTIPLY..
                                  ;One of the operands of the
                                  ;MULTIPLY statement either is
                                  ;missing or is invalid.
                                  ;Fatal.

714  ILLEGAL .MULTIPLY. DUE
     TO MISSING .BY..
                                  ;The keyword BY is omitted in
                                  ;the MULTIPLY statement.
                                  ;Fatal.

716  MISSING OR INVALID OPERAND
     OF .DIVIDE..
                                  ;One of the operands of the
                                  ;DIVIDE statement either is
                                  ;missing or is invalid.
                                  ;Fatal.

717  ILLEGAL .DIVIDE. DUE TO
     MISSING .BY. OR .INTO..
                                  ;One of the keywords BY or
                                  ;INTO is omitted in the DIVIDE
                                  ;statement. Fatal.
```

```
720  .GIVING. OPTION OF .DIVIDE.
     MISSING.                          ;The GIVING option must be
                                       ;specified in a DIVIDE
                                       ;statement when one of the
                                       ;following syntactic elements
                                       ;is present in the DIVIDE
                                       ;statement: (1) a numeric
                                       ;literal follows the keyword
                                       ;INTO or (2) the keyword BY
                                       ;is specified. In this DIVIDE
                                       ;statement, the GIVING option
                                       ;was omitted while one of the
                                       ;two aforementioned syntactic
                                       ;elements were present.
                                       ;Fatal.

721  MISSING OR INVALID OPERAND
     OF .ADD..                         ;One of the operands of
                                       ;the ADD statement either is
                                       ;missing or is invalid.
                                       ;Fatal.

722  .TO. OR .GIVING. MISSING
     FROM .ADD..                       ;One of the keywords TO or
                                       ;GIVING is omitted in the ADD
                                       ;statement. Fatal.

723  MISSING OR INVALID
     OPERAND OF SUBTRACT.              ;One of the operands in the
                                       ;SUBTRACT statement either is
                                       ;missing or is invalid.
                                       ;Fatal.

724  FILE NEEDS DYNAMIC ACCESS
     FOR .READ NEXT..                  ;In a READ NEXT statement, the
                                       ;referenced file must have
                                       ;ACCESS MODE IS DYNAMIC
                                       ;specified in the FILE-CONTROL
                                       ;paragraph. Fatal.

725  BAD PROCEDURE NAME IN
     .PERFORM..                        ;A missing or invalid
                                       ;procedure-name is recognized
                                       ;in the PERFORM statement.
                                       ;Fatal.

726  ILLEGAL OPERAND OF .TIMES.
     OPTION OF .PERFORM..              ;The TIMES operand of the
                                       ;PERFORM statement is not
                                       ;a numeric integer data-name or
                                       ;numeric integer literal. The
                                       ;compiler assumes a value of 1
                                       ;for the TIMES operand.

727  .TIMES. MISSING FROM
     .PERFORM.. ASSUMED.               ;The PERFORM statement does
                                       ;not contain the keyword TIMES
                                       ;but does contain the
                                       ;iteration value required to
                                       ;execute the PERFORM
                                       ;correctly. The keyword TIMES
                                       ;is assumed present.

730  PROCEDURE NAME OMITTED
     IN .ALTER..                       ;A valid procedure-name was
                                       ;not recognized in the ALTER
                                       ;statement. Fatal.
```

```
731  ILLEGAL .ALTER.  DUE
     TO MISSING .TO..          ;The keyword TO was not
                               ;recognized in the ALTER
                               ;statement.  Fatal.

732  FILE HAS VAR. SIZE RECS.
     .READ INTO. ILLEGAL.      ;It is illegal for the READ
                               ;INTO statement to reference a
                               ;file which has multiple
                               ;record descriptions of
                               ;different lengths.  Fatal.

733  FILE ACCESSED BY VERB
     REQUIRING .LINAGE.        ;A file is accessed by an I/O
                               ;verb which did not have a
                               ;LINAGE clause in its
                               ;specification.  Fatal.

734  .DELETE. OR .REWRITE.
     WITHOUT INV. KEY OR USE.  ;A DELETE or REWRITE statement
                               ;references a file for which
                               ;there was no USE procedure
                               ;specified and for which the
                               ;INVALID KEY option was not
                               ;specified in that DELETE or
                               ;REWRITE statement.  Fatal.

735  OPEN MODE OR NO READ
     PROHIBITS REWRITE OR DELETE.
                               ;A DELETE or REWRITE statement
                               ;references a file which was
                               ;not OPENed in the proper mode
                               ;or which has no READ
                               ;statement referencing it in
                               ;the program.  Fatal.

736  .START. CONFLICTS WITH OPEN
     MODE.                     ;A START statement references
                               ;a file which was not opened
                               ;in the proper mode.  Fatal.

737  .WRITE. CONFLICTS WITH OPEN
     MODE.                     ;A WRITE statement references
                               ;a file which was not opened
                               ;in the proper mode.  Fatal.

740  .READ. CONFLICTS WITH OPEN
     MODE.                     ;A READ statement references
                               ;a file which is only opened
                               ;in OUTPUT or EXTEND
                               ;mode.  Fatal.

741  USE NOT IN DECLAR. OR NOT
     FOLLOWING SECTION NAME.   ;The USE statement is not in
                               ;the DECLARATIVES section of
                               ;the PROCEDURE DIVISION or is
                               ;not immediately following a
                               ;section name inside the
                               ;DECLARATIVES.  Fatal.

743  INTEGER IN SWITCH CLAUSE
     INVALID OR OMITTED.       ;A SWITCH clause of the
                               ;SPECIAL-NAMES paragraph
                               ;either contains an invalid
                               ;numeric integer or has
                               ;omitted the integer in its
                               ;specification.  A SWITCH
                               ;clause integer, say n, must
                               ;fall in the decimal range
                               ;1<=n<=16.  The SWITCH clause
                               ;is ignored.
```

744 .IS. OMITTED IN SPECIALNAMES. ASSUMED PRESENT.

;The required keyword is ;omitted in a clause of the ;SPECIAL-NAMES paragraph. IS ;is assumed present and ;processing continues.

745 DEVICE MNEMONIC OMITTED IN SPECIALNAMES.

;A valid device mnemonic-name ;is not recognized in one of ;the CONSOLE, LINE-PRINTER, ;CARD-READER, PAPER-TAPE- ;READER, or PAPER-TAPE-PUNCH ;clauses of the SPECIAL-NAMES ;paragraph. All source text is ;skipped up to the next ;recognizable keyword.

746 TERMINATOR OMITTED IN SPECIALNAMES.

;The SPECIAL-NAMES paragraph ;is not terminated by a ;period. The period is ;assumed present and ;processing continues.

750 KEYWORD OMITTED IN .SWITCH. CLAUSE.

;One of the keywords OFF or ON ;is omitted in the SWITCH ;clause of the SPECIAL-NAMES ;paragraph. The SWITCH clause ;is ignored.

751 CONDITION NAME MISSING IN .SWITCH. CLAUSE.

;A valid condition-name is not ;recognized in the SWITCH ;clause of the SPECIAL-NAMES ;paragraph. The SWITCH clause ;is ignored.

752 .CR. OR .DB. NOT AT RIGHT END OF PICTURE.

;The PICTURE symbol CR or DB ;does not appear at the ;right end of the PICTURE ;string. The compiler ignores ;the user-supplied PICTURE and ;declares the data-name ;alphanumeric with a "PICTURE ;X" declaration.

753 .CR. OR .DB. USED WITH SIGNED ITEM.

;Both the PICTURE symbols, CR ;or DB, and a sign, + or -, ;appear in the same PICTURE. ;The compiler ignores the ;user-supplied PICTURE and ;declares the data-name ;alphanumeric with a "PICTURE ;X" declaration.

754 MULTIPLE DEFINITION OF SWITCH. FIRST USED.

;Multiple definitions of a ;COBOL switch are detected in ;the SPECIAL-NAMES paragraph. ;All but the first definition ;of the SWITCH are ignored.

```
755  .SENTENCE. ASSUMED AFTER          ;The keyword NEXT is not
     .NEXT..                           ;followed by the keyword
                                       ;SENTENCE. SENTENCE is
                                       ;assumed present and
                                       ;processing continues.

756  SUBSCRIPT NOT NUMERIC             ;A data-name used as a
     INTEGER.                          ;subscript is not numeric
                                       ;in class. A default value of
                                       ;1 is assumed as the
                                       ;subscript.
```

APPENDIX A

THE COBOL FORMATS

The following standard notation is used in these formats:

- Underlined upper-case words (key words) - required words;
- Upper-case words (not underlined) - optional words;
- Lower-case words - generic terms, must be supplied by the user;
- Brackets [] - enclosed portion is optional; if several enclosed words are vertically stacked, only one of them may be used;
- Braces { } - a selection must be made from the vertical stack of enclosed words;
- Ellipsis... - the position at which repetition may occur;
- Comma and semicolon - optional punctuation;
- Period - required where shown in the formats.

NOTE

Shaded entries indicate PDP-11 COBOL extensions to the ANS-74 list of formats.

```
IDENTIFICATION DIVISION.

PROGRAM-ID.  program-name.

[AUTHOR.  [comment-entry] ...]

[INSTALLATION.  [comment-entry] ...]

[DATE-WRITTEN.  [comment-entry] ...]

[DATE-COMPILED.  [comment-entry] ...]

[SECURITY.  [comment-entry] ...]


ENVIRONMENT DIVISION.

CONFIGURATION SECTION.

SOURCE-COMPUTER.  PDP-11.

OBJECT-COMPUTER.  PDP-11          [ MEMORY SIZE integer { WORDS      } ]
                                                        { CHARACTERS }
                                                        { MODULES    }

          [ PROGRAM COLLATING SEQUENCE IS alphabet-name].

[SPECIAL-NAMES.

  [ { { CARD-READER       }                                    } ]
    { { CONSOLE           }     IS mnemonic name               }
    { { LINE-PRINTER      }                                    }  ...
    { { PAPER-TAPE-PUNCH  }                                    }
    { { PAPER-TAPE-READER }                                    }

    [SWITCH integer-1 { ON  STATUS IS condition-name-1                    } ]
                      { OFF STATUS IS condition-name-2]                   }  ...
                      { OFF STATUS IS condition-name-2                    }
                      { ON  STATUS IS condition-name-1]                   }

   [ Alphabet-name IS  { NATIVE     } ]
                       { STANDARD-1 }
    [CURRENCY SIGN IS literal-1]
    [DECIMAL-POINT IS COMMA].]
```

```
[INPUT-OUTPUT SECTION.

FILE-CONTROL.

    {SELECT [OPTIONAL] file-name

     ASSIGN TO literal-1

     [RESERVE integer-1 [AREA ]]
                        [AREAS]

     [ORGANIZATION IS {RELATIVE  }]
                      {SEQUENTIAL}

     [ACCESS MODE IS   {SEQUENTIAL   [RELATIVE KEY IS data-name-1]}]
                       {RANDOM     }  RELATIVE KEY IS data-name-1
                       {DYNAMIC    }

     [FILE STATUS IS data-name-2].} ...

[I-O-CONTROL.

     [SAME  [RECORD]  AREA FOR file-name-1  {file-name-2}...] ...

     [MULTIPLE FILE TAPE CONTAINS file-name-3  [POSITION integer-1]

          [file-name-4  [POSITION integer-2]]... ] ...

     {APPLY  {MAP4          }
             {MAP5          }
             {MAP6          }
             {MAP7          }  ON file-name-5 [ file-name-6] ...]...  .]]
             {MAP8          }
             {READ-AHEAD    }
             {WRITE-BEHIND  }
             {PRINT-CONTROL }
```

```
DATA DIVISION.

[FILE SECTION.

[FD file-name

     [BLOCK CONTAINS  [integer-1 TO] integer-2   {RECORDS   }]
                                                 {CHARACTERS}

     [RECORD CONTAINS [integer-3 TO] integer-4    CHARACTERS]

     LABEL      {RECORD IS  }  {STANDARD}
                {RECORDS ARE}  {OMITTED }

     [VALUE OF ID IS     {data-name-1}]
                         {literal-1  }

     [DATA      {RECORD IS  }data-name-3    [data-name-4]  ...]
                {RECORDS ARE}

     [LINAGE IS  {data-name-5} LINES   [WITH FOOTING AT   {data-name-6}]
                 {integer-5  }                            {integer-6  }

          [LINES AT TOP {data-name-7}]   [LINES AT BOTTOM  {data-name-8}]]
                        {integer-7  }                      {integer-8  }

[CODE-SET IS alphabet-name].

[record-description-entry]...]...]

[WORKING-STORAGE SECTION.

     [77-level-description-entry  ]
     [record-description-entry    ]        . . .]

Data description entry:

Format 1:

     level-number  {data-name-1}
                   {FILLER     }

     [REDEFINES      data-name-2]

     [{PICTURE}      IS character-string]
     [{PIC    }

     [   [USAGE IS] {COMPUTATIONAL}                    ]
     [              {COMP         }                    ]
     [              {DISPLAY      }                    ]
     [              {DISPLAY-6    }                    ]
     [              {DISPLAY-7    }                    ]
     [              {INDEX        }                    ]

     [   [SIGN IS]  {LEADING }  [SEPARATE CHARACTER]   ]
     [              {TRAILING}                         ]
                                                  (Continued)
```

```
[{SYNCHRONIZED}[LEFT ]]
 {SYNC       } [RIGHT]]
[{JUSTIFIED}      ]
 {JUST     }  RIGHT ]
     [BLANK WHEN ZERO]
     [VALUE IS literal]
[OCCURS integer-1 TIMES
                [INDEXED BY index-name-1 [ index-name-2] ... ]].
```

Format 2:

```
88 condition-name {VALUE IS   } literal-1 [{THROUGH}    literal-2 ]
                  {VALUES ARE }           {THRU   }
 [literal-3       [{THROUGH}    literal-4]... .
                  {THRU   }
```

PROCEDURE DIVISION.

Format 1:

```
     [DECLARATIVES.
     {section-name SECTION  [segment-number].  declarative-sentence
     [paragraph-name. [sentence] ...] ...} ...
     END DECLARATIVES.]
     {section-name SECTION [segment-number].
     [paragraph-name. [sentence] ...] ...} ...
```

Format 2:

```
     {paragraph-name.  [sentence] ...} ...
```

STATEMENTS:

```
     ACCEPT identifier [FROM mnemonic-name]

     ACCEPT identifier FROM {DATE}
                            {DAY }
                            {TIME}

     ADD {identifier-1} [identifier-2] ...TO identifier-3 [ROUNDED]
         {literal-1   } [literal-2   ]
                    [  ON SIZE ERROR imperative-statement]
```

```
ADD {identifier-1} {identifier-2}  [identifier-3]...
    {literal-1   } {literal-2   }  [literal-3   ]

    GIVING identifier-4 [ROUNDED]

        [ON SIZE ERROR imperative-statement]
```

```
ALTER procedure-name-1 TO [PROCEED TO] procedure-name-2
```

```
CLOSE file-name-1 [ {REEL  } [WITH NO REWIND ] ]
                  [ {UNIT  } [FOR REMOVAL    ] ]
                  [                            ]
                  [ WITH    {NO REWIND}        ]
                  [         {LOCK    }        ]

                  [ file-name-2 [ {REEL  } [WITH NO REWIND ] ] ]  ...
                  [             [ {UNIT  } [FOR REMOVAL    ] ] ]
                  [             [                            ] ]
                  [             [ WITH    {NO REWIND}        ] ]
                  [             [         {LOCK    }        ] ]
```

```
DELETE file-name RECORD [INVALID KEY imperative-statement]
```

```
DISPLAY {identifier-1} [identifier-2]
        {literal-1   } [literal-2   ]
    [UPON mnemonic-name] [WITH NO ADVANCING]
```

```
DIVIDE

    {identifier-1} INTO identifier-2 [ROUNDED]
    {literal-1   }

        [ON SIZE ERROR imperative-statement]
```

```
DIVIDE

    {identifier-1} INTO {identifier-2} GIVING identifier-3 [ROUNDED]
    {literal-1   }      {literal-2   }

        [ON SIZE ERROR imperative-statement]
```

```
DIVIDE

    {identifier-1} BY {identifier-2} GIVING identifier-3 [ROUNDED]
    {literal-1   }    {literal-2   }

        [ON SIZE ERROR imperative-statement]
```

```
EXIT.

GO TO procedure-name-1
GO TO procedure-name-1 [procedure-name-2]... procedure-name-n

    DEPENDING ON identifier

IF condition  {statement-1    }   {ELSE statement-2    }
              {NEXT SENTENCE}   {ELSE NEXT SENTENCE}

INSPECT identifier-1 TALLYING

    {identifier-2 FOR{  {{ALL    }{identifier-3}}  [{BEFORE}  INITIAL  {identifier-4}]}...}...
                     {  {{LEADING}{literal-1   }}  [{AFTER }         {literal-2   }]}
                     {  {CHARACTERS              }

INSPECT identifier-1 REPLACING

    {CHARACTERS BY {identifier-6}  {BEFORE}  INITIAL {identifier-7}]
                   {literal-4   }  {AFTER }         {literal-5   }
    {{{ALL    }  {identifier-5}  BY {identifier-6}  [{BEFORE}  INITIAL {identifier-7}]}...}...
    {{{LEADING}  {literal-3   }                     [{AFTER }         {literal-5   }]}
    {{{FIRST  }

INSPECT identifier-1 TALLYING

    {identifier-2 FOR  {{{ALL    }{identifier-3}}  [{BEFORE}  INITIAL  {identifier-4}]}...}...
                       {{{LEADING}{literal -1  }}  [{AFTER }         {literal-2   }]}
                       {  {CHARACTERS            }

REPLACING

{CHARACTERS BY {identifier-6}  [{BEFORE}  INITIAL {identifier-7}]
               {literal-4   }  [{AFTER }         {literal-5   }]
{  {{ALL    }  {identifier-5}  BY {identifier-6}  [{BEFORE}  INITIAL {identifier-7}]}...}...
{  {{LEADING}  {literal-3   }                     [{AFTER }         {literal-5   }]}
{  {{FIRST  }

MOVE {identifier-1} TO identifier-2  [identifier-3]...
     {literal     }

MULTIPLY {identifier-1} BY identifier-2 [ROUNDED]
         {literal-1   }
    [ON SIZE ERROR imperative-statement]

MULTIPLY {identifier-1} BY {identifier-2} GIVING identifier-3 [ROUNDED]
         {literal-1   }    {literal-2   }
    [ON SIZE ERROR imperative-statement]

     {INPUT file-name-1[WITH NO REWIND]  [file-name-2[WITH NO REWIND]] ...}
OPEN {OUTPUT file-name-3[WITH NO REWIND]  [file-name-4[WITH NO REWIND]] ...}...
     {
     {I-O file-name-5    [file-name-6]  ...
     {EXTEND file-name-7 [file-name-8]  ...

PERFORM procedure-name-1 [{THROUGH} procedure-name-2]
                         {THRU   }

PERFORM procedure-name-1 [{THROUGH} procedure-name-2]  {identifier-1} TIMES
                         {THRU   }                     {integer-1   }
```

```
READ file-name [NEXT] RECORD [INTO identifier] [AT END imperative-statement]

READ file-name RECORD [INTO identifier] [INVALID KEY imperative-statement]

REWRITE record-name[FROM identifier] [INVALID KEY imperative-statement]

SET {identifier-1  [identifier-2] ...}  TO {identifier-3}
    {index-name-1  [index-name-2] ...}     {index-name-3}
                                           {integer-1   }

SET index-name-4 [index-name-5] ...{UP BY  } {identifier-4}
                                   {DOWN BY} {integer-2   }

START file-name [ KEY {IS EQUAL TO       } data-name ]
                      {IS =              }
                      {IS GREATER THAN   }
                      {IS >              }
                      {IS NOT LESS THAN  }
                      {IS NOT <          }

     [INVALID KEY imperative-statement]

STOP {RUN    }
     {literal}

STRING {identifier-1} [identifier-2] ...DELIMITED BY {identifier-3}
       {literal-1   }  {literal-2   }                {literal-3   }
                                                     {SIZE        }

     [{identifier-4}          [identifier-5]                        {identifier-6}  ] ...
      {literal-4   }  ...      {literal-5   }  ...DELIMITED BY      {literal-6   }
                                                                    {SIZE        }

INTO identifier-7 [WITH POINTER identifier-8]
     [ON OVERFLOW imperative-statement]

SUBTRACT {identifier-1} [identifier-2] ...FROM identifier-3 [ROUNDED]
         {literal-1   }  {literal-2   }

     [ON SIZE ERROR imperative-statement]

SUBTRACT {identifier-1} [identifier-2] ...FROM {identifier-3}
         {literal-1   }  {literal-2   }       {literal-3   }
     GIVING identifier-4 [ROUNDED]
     [ON SIZE ERROR imperative-statement]

UNSTRING identifier-1
     [DELIMITED BY [ALL] {identifier-2} [ OR [ALL] {identifier-3} ] ... ]
                         {literal-1   }            {literal-2   }
       INTO identifier-4 [DELIMITER IN identifier-5] [COUNT IN identifier-6]
       [identifier-7 [DELIMITER IN identifier-8] [COUNT IN identifier-9]]...
       [WITH POINTER identifier-10]  [TALLYING IN identifier-11]
       [ON OVERFLOW imperative-statement]
```

```
USE AFTER STANDARD {EXCEPTION} PROCEDURE ON { file-name-1 [file-name-2] ... }
                   {ERROR    }             { INPUT                          }
                                           { OUTPUT                         }
                                           { I-O                            }
                                           { EXTEND                         }

    WRITE record-name  [FROM identifier-1]

        [ {BEFORE} ADVANCING { identifier-2 } [ LINE ] ]
        [ {AFTER }           { integer      } [ LINES] ]
                             { [PAGE]       }

          [AT {END-OF-PAGE} imperative-statement]
              {EOP        }

    WRITE record-name [FROM identifier] [INVALID KEY imperative-statement]
```

```
COPY {text-name}
     {literal-1}

    [ REPLACING { literal-2 } BY { literal-3 } ... ] .
                { word-1    }   { word-2    }
```

NOTE: A COPY statement may appear anywhere that a word appears in the COBOL source program.

APPENDIX B

LOGICAL UNIT NUMBER (LUN) ASSIGNMENTS

LUN	ASSIGNMENT
1	Console, input
2	Console, output
3	Work file
4	Listing file
5	Source file
6	Unused
7	Intermediate object file and load map listing
10	Object file
11	COPY library file
12	ACCEPT device LUN
13	DISPLAY device LUN
14-30	Run-time COBOL files

APPENDIX C

PDP-11 COBOL FIELD RELEASE VERSION 1.00 RESTRICTIONS

INTRODUCTION

The following is a list of restrictions for the Field Release Version 1.00 of the PDP-11 COBOL compiler. These restrictions represent areas of the COBOL language which we know that the compiler supports incorrectly or does not support at all at the time of the release of the Field Version 1.00 of the PDP-11 COBOL compiler. Therefore, the user is advised not to use these "restricted" language constructs as their results will be unpredictable. In all cases, the user can "program around" the restriction. As these restrictions are removed, the user will be notified via PATCH updates in the Software Bulletin.

RESTRICTIONS

1. The user may not use the DAY option of the ACCEPT statement.

   E.G.: ACCEPT DAY-STRING FROM DAY.

   There is no Julian date facility available in the operating system.

2. The user may not specify the ROUNDED phrase for the "rounding" of a numeric-edited GIVING operand in all arithmetic statements.

   E.G.: 1) MULTIPLY A BY B GIVING NUM-ED ROUNDED.

   2) DIVIDE A BY B GIVING NUM-ED ROUNDED.

3. All procedure-names must contain an alphabetic character. Procedure names containing only numeric characters or numeric characters and hyphens are rejected. These conditions are diagnosed by the compiler.

4. 01 entries in the File Section of the Data Division may be described as elementary only if they are not

   .alphanumeric edited class
   .numeric edited class
   .COMPUTATIONAL usage
   .contain the SIGN clause.

   These items may be described in the File Section with level numbers other than 01. Thus

```
01  NUM-EDIT  PIC  Z(10).
```

   may be described as

```
01  RECORD-LEVEL.
    02  NUM-EDIT  PIC  Z(10).
```

   This restriction is diagnosed by the compiler.

5. MOVE of a subscripted sending item to multiple receiving items where one of the receiving items other than the last is also one of the subscripts of the sending item. This delivers the wrong sending item to some receiving fields. Example:

```
MOVE  A(I)  TO  B, I, C.
```

C will receive a different A(I) than B. The above statement will execute correctly if rewritten as

```
MOVE  A(I)  TO  B, C,I.
```

6. No more than one file with a LINAGE clause may appear in the same program, if reference is to be made to the LINAGE-COUNTER, since duplicate LINAGE-COUNTERs receive warning diagnostics and qualification of the LINAGE-COUNTER reference is not operational.

7. The SET statement delivers the wrong sending field value to multiple receiving fields under the following conditions: The SET of multiple receiving items TO an indexed sending field where one of the receiving items (other than the last) is also one of the indexes of the sending field. Example:

```
SET  A, INDEX-NAME, B TO C(INDEX-NAME).
```

B will receive a different C(INDEX-NAME) than A.

The above statement will execute correctly if rewritten as

```
SET A,B,INDEX-NAME TO C(INDEX-NAME)
```

8. The user may not specify 88 level alphanumeric literals as this may produce unpredictable results at object-time.

9. The compiler fails to diagnose the illegal MOVE from a numeric field to an alphabetic field whose PICTURE character string contains the editing character "B".

Example:

```
.
.
.
01  A PIC  9(4).
01  B  PIC  AABBAA.

.
.
.
    MOV A TO B.

.
.
.
```

10. The compiler fails to issue a warning diagnostic in the case where the user mixes index-names and data-names in a subscript list used to reference a table item. Although the compiler does not warn the user of the violation, there is no difficulty at run time in correctly referencing the desired table element. The compiler simply fails to enforce a somewhat arbitrary rule of the ANS-COBOL standard.

Example:

```
.
.
.

01  I  PIC  9(4).
01  TABLE.
    02  TBL-1 OCCURS 5 TIMES.
        03  TABLE-ITEM PIC 9(8) OCCURS 5 TIMES INDEXED BY J.

.
.
.

    MOVE 1 TO TABLE-ITEM (I,J).
```

INDEX

Abortive diagnostics (level A), 10-4
ACCESS MODE clause, 6-25
Access modes, 6-24
ACCEPT statement, using the, 6-30
Access time, saving, 6-10
Accumulator specification, COBRG, 9-10
Active/Inactive arguments, INSPECT operation, 3-42
ADD statement, 4-16
ADVANCING clause, 6-4
ADVANCING option of the WRITE statement, 6-22
ALL search argument, 3-53
Alphabetic field, 3-1
Alphanumeric class field, 3-1
ALTER statement, using the, 8-5
ANS-74 COBOL, 1-3
APPLY PRINT-CONTROL clause, 6-4, 6-22
Arithmetic statements, 4-11
Area A, 2-5, 2-8
Area B, 2-5, 2-8
Arithmetic statements, common errors, 4-18
ASCII code chart, 3-4
ASSIGN clause, device assignment by, 6-17
Asterisk, 2-4, 2-7
AT END imperative condition, 6-35

BEFORE/AFTER phrase, INSPECT statement, 3-37
Bit map, 6-8
BLOCK CONTAINS clause, 6-5, 6-6
BLOCK CONTAINS ... RECORDS clause, 6-9
Blocking, default, 6-5
Blocking factors, determining, 6-9
Blocking factors (table), 6-10
Blocks, fixed-length, 6-5
Block size, disk, 6-9
Block, standard disk, 6-8
Blocks, variable-length, 6-5
Break specification, COBRG, 9-8
Buffer, 6-12
Buffering, 6-12
Buffer size, determining, 6-6
Buffer size, 6-13
Bytes, 6-11

Calling the COBOL compiler, 2-18
Card reader, 7-3
Character handling, non-numeric, 3-1
Character position 0, 2-6
CHARACTERS, search argument, 3-53
Classes of data, 3-5
Class tests, 3-6, 4-7
/CL:n switch, 6-17
CLOSE statement, 6-23
Closing relative files, 6-29
Closing sequential files, 6-23
CLOSE statement, 6-29
COBOL command string, 2-19
COBOL command string, common entry errors, 2-24
COBOL command string, examples of the, 2-23
COBOL compiler, calling the, 2-18
COBOL, file compatibilities, 6-32
COBOL I/O devices, 7-1
COBOL I/O statements, 6-18
COBOL library file, creating a, 2-9
COBOL object time system (OTS), 6-1
COBOL reference formats, A-1
COBOL utility programs, 1-8
COBRG, accumulator specification, 9-10
COBRG, break specification, 9-8
COBRG (COBOL Report generator), 9-1
COBRG command string, 9-15
COBRG, default assumptions, 9-15
COBRG, emit specification, 9-13
COBRG, error messages, 9-25
COBRG, header specifications, 9-7
COBRG, input specification lines, 9-2
COBRG, input specifications, 9-5
COBRG, list specification, 9-14
COBRG, name specification, 9-4
COBRG, output from, 9-15
COBRG, output specification, 9-6
COBRG (sample), input file, 9-23
COBRG (sample), listing file, 9-17
COBRG (sample), printed report, 9-24
COBRG (sample), source program, 9-18
COBRG (sample), specification lines, 9-16
COBRG, summary, 1-8
COBRG, the sample program, 9-15
COBRG, total specification, 9-12
Comment line, 2-7
Comment lines, 2-4
Common entry errors, COBOL command string, 2-24
Common errors, arithmetic statements, 4-18
Common errors, INSPECT statement, 3-54
Common errors, library facility, 2-18
Common errors, MOVE statement, 3-12

Common errors, numeric MOVE statements, 4-11
Common errors, object code execution, 2-25
Common errors, STRING statement, 3-20
Common errors, UNSTRING statement, 3-36
Communicating with the program, 6-30
Comparison of disk devices, 7-2
Compile-and-go, 2-24
Compile-and-go concept, 1-4
Compiler, installing the, 2-19
Compiler switches, 2-22
Compiler system errors, 10-1
Compiler, using the, 2-18
COMP items, signed, 4-2
COMPUTATIONAL (COMP) usage, 4-1
Concatenation, 3-7
Configuration section, 1-6
Console, creating source programs at the, 2-2
/CO:n switch, 6-17
Continuation/comment indicator, 2-3
Control characters, 3-3
Control word, forms, 6-4
Conventional (ANS) format, 8-1
Conventional format, 2-2, 2-3
Copy, library facility, 2-9
Count field, 6-3
Count phrase, UNSTRING statement, 2-38
Creating a COBOL library file,
Creating source programs at the console, 2-2
Current date, 6-31
Current record area, 6-11, 6-12
Current record pointer, 6-25, 6-26, 6-28, 6-29

Data categories, five, 3-1
Data classes, three, 3-1
Data Division, 1-7
Data item, RELATIVE KEY, 6-29
Data movement, 3-7
Data-names, subscripting with, 5-9
Data organization, 3-2
DATE, 6-31
Date, current, 6-31
DAY, 6-31
DECtape, 7-2
Default assumptions, COBRG, 9-15
Default blocking, 6-5
Default blocking, overriding, 6-6
Defaults, file specification, 2-10, 6-15
Defining tables, 5-1
DELETE statement, 6-28
Deleting records from a relative file, 6-28
DELIMITED BY phrase, 3-14
DELIMITED BY phrase, UNSTRING statement, 3-23
DELIMITER phrase, UNSTRING statement, 3-29
Determining blocking factors, 6-9
Determining buffer size, 6-6
Determining file sizes, 6-9
Developing a program, 2-1
Device assignment by ASSIGN clause, 6-17
Device codes, 6-14
Device restrictions, sequential files, 6-6
Devices, 7-1
Devices, COBOL I/O, 7-1
Diagnostic error messages, 10-1
Diagnostic message, typical, 10-2
Disk, 7-2
Disk block size, 6-9
Disk devices comparison of, 7-2
Disk space, saving, 6-10
DISPLAY/COMP, usages, 4-1
DISPLAY items, signed, 4-2
DISPLAY statement, 6-30, 6-31
DISPLAY usage, 4-1
DIVIDE statement, 4-17
Double buffering, 6-13

Edited moves, 3-10
Elementary items, 3-2
Elementary moves, 3-8
Elementary numeric edited moves, 4-10
Elementary numeric moves, 4-8
EMIT specification, COBRG, 9-13
Empty record positions, relative files, 6-7
End-of-file mark, 6-2, 6-21
End-of-volume, label, 6-3
Environment division, 1-6
Error code, file control service, 6-35
Error message numbers, 6-35
Error messages, 10-1
Error messages, COBRG, 9-25
Error message severity levels, 10-3
Error messages, REFORMAT, 9-30
Error messages, run-time, 6-38
Errors, compiler system, 10-1
Even byte addresses, 3-3
Examples of the COBOL command string, 2-23
Execute only, 2-24
Executing an OPEN statement, 6-20
Executing the object code, 2-24
Explicit file-name, 6-16
EXTEND mode, 6-21, 6-23

Fatal diagnostics (level F), 10-3
Field release, PDP-11 COBOL Version 1.00 restrictions, C-1

File access methods, 6-1
File compatibilities--COBOL, FORTRAN, and MACRO, 6-32
File control service error code, 6-35
File control service (FCS) error codes (table), 6-37
File handling, 6-1
File-name, explicit, 6-16
File organization methods, 6-1
File organization, relative, 6-6
File, print-controlled, 6-4, 6-23
File Section, 1-7
Files, multi-volume, 6-3
File specification, 6-15
File specification defaults, 2-10, 6-15
File specification library files, 2-10
File specification, listing, 2-19
File specification, object program, 2-19
File specification, source program, 2-21
Files sizes, determining, 6-9
FILE STATUS key, 6-35
File status key values (table), relative I/O, 6-36
File status key values (table), sequential I/O, 6-36
Finding an argument match, INSPECT operation, 3-43
FIRST search argument, 3-53
Five data categories, 3-1
Fixed-length blocks, 6-5
Fixed-length format records, 6-4
Fixed-length records, 6-3, 6-7
Formats, COBOL reference, A-1
Formatting the source program, 8-1
Form control characters, 6-32
Forms, control word, 6-4
FORTRAN carriage control, 6-33
FORTRAN, file compatibilities, 6-32
FORTRAN, reading files written by, 6-33
FORTRAN, writing files, 6-32
Four major divisions, 1-6

GIVING phrase, 4-15
Good programming practices, 8-1
Group items, 3-2
Group moves, 3-8

Header specifications, COBRG, 9-7
Head of form, 6-20

Identification characters, 2-5
Identification division, 1-6
Illegal values in numeric fields, 4-3
Imperative condition, AT END, 6-35
Imperative condition, INVALID KEY, 6-35
Implicit redefinition, INSPECT statement, 3-39
Incrementing level numbers, 8-2
Indenting subordinate items, 8-2
Index data items, 5-12
INDEXED BY phrase, 5-7
Indexing, relative, 5-12
Indexing, subscripting and, 5-7
Indexes, subscripting with, 5-10
Index-name item, 5-11
Index-name items, subscripting with, 5-11
Indicator, print-control, 6-6
Indicator, record spanning, 6-6
Informative diagnostics (level I), 10-3
Initializing mixed usage fields, 5-6
Initializing tables, 5-6
Input file, COBRG (sample), 9-23
Input-Output section, 1-7
Input specification lines, COBRG, 9-2
Input specifications, COBRG, 9-5
Inserting blank lines or comment lines, 8-2
INSPECT operation, Active/Inactive arguments, 3-42
INSPECT operation, finding an argument match, 3-43
INSPECT operation, setting the scanner, 3-42
INSPECT statement, 3-36
INSPECT statement, BEFORE/AFTER phrase, 3-37
INSPECT statement, common errors, 3-57
INSPECT statement, implicit redefinition, 3-39
INSPECT statement, interference in tally argument lists, 3-48
INSPECT statement, REPLACING arguments, 3-41
INSPECT statement, REPLACING phrase, 3-52
INSPECT statements, subscripted fields in, 3-44
INSPECT statement, tally argument, 3-45
INSPECT statement, tally argument list, 3-46
INSPECT statement, tally counter, 3-45
INSPECT statement, TALLYING arguments, 3-41
INSPECT statement, TALLYING phrase, 3-44
Installing the compiler, 2-19
Interference in tally argument lists, INSPECT statement, 3-48

Intermediate result field, 4-12
INVALID KEY imperative condition, 6-35
I/O error procedures, runtime file, 6-34
I/O errors, processing, 6-34
I-O mode, 6-21
I/O statements, relative, 6-24
I/O statements, sequential, 6-19

JUSTIFIED moves, 3-10

Label, end-of-volume, 6-3
LEADING search argument, 3-53
Legal data movement, with SET statement, 5-13
Legal non-numeric elementary moves, 3-9
Level 88 condition-names, use of, 8-9
Library facility, common errors, 2-18
Library facility copy, 2-9
Library files, file specification, 2-10
Library text, listing, 2-16
LINAGE clause, 6-22
LINAGE counters, 6-20
Line printer, 7-3
Listing file, COBRG (sample), 9-17
Listing file specification, 2-21
Listing library text, 2-16
List specification, COBRG, 9-14
Literals with leading or trailing zeroes, 4-9
LOCK option, 6-23, 6-30
Logical Unit Number (LUN) assignments, B-1

MACRO file compatibilities, 6-32
MACRO, reading files written by, 6-33
MACRO, writing files, 6-32
Magnetic tape, 7-3
Mapping table elements, 6-2
Mark, end-of-file, 6-21
Media restrictions, relative files, 6-11
Merging a library file, 2-12
Merging a library file into Area B, 2-13
Mnemonic-name, 6-30
Moves, edited, 3-10
Moves, elementary numeric, 4-8
Moves, elementary numeric edited, 4-10
Moves, justified, 3-10
Moves, subscripted, 3-11
MOVE statement, 3-7, 3-8
MOVE statement, common errors, 3-12
MOVE statement, numeric, 4-7
Multi-dimensional table, subscripting a, 5-8
Multiple delimiters, UNSTRING statement, 3-27
Multiple operands in ADD and SUBTRACT statements, 4-15
Multiple receiving fields, MOVE statement, 3-11
Multiple sending fields, STRING statement, 3-13
Multiple receiving fields, UNSTRING statement, 3-21
MULTIPLY statement, 4-16
Multi-volume files, 6-3, 7-3

Name specification, COBRG, 9-4
Non-numeric character handling, 3-1
Non-numeric elementary moves, legal, 3-9
Numbered record positions, 6-6
Numbers, error message, 6-35
Numeric character handling, 4-1
Numeric editing symbols, 4-10
Numeric fields, illegal values in, 4-3
Numeric, MOVE statement, 4-7
Numeric MOVE statements, common errors, 4-11

Object code, executing the, 2-24
Object code execution, common errors, 2-25
Object program file specification, 2-19
Object time system (OTS), COBOL, 6-1
OCCURS phrase, 5-1
Opening relative files, 6-25
Opening sequential files, 6-19
OPEN modes, sequential, 6-19
OPEN statement, 6-19, 6-25
OPEN statement, executing an, 6-20
Organizational ideas, 8-4
Output from COBRG, 9-15
Output specification, COBRG, 9-6
OVERFLOW phrase, 3-16
OVERFLOW phrase, UNSTRING statement, 3-33
Overriding default blocking, 6-6

Padding character, 5-2
Paragraphs, 1-7
PDP-11 COBOL field release version 1.00 restrictions, C-1
PDP-11 COBOL utility programs, 9-1
POINTER phrase, 3-14
POINTER phrase, UNSTRING statement, 3-30
Print-control, 6-11
Print-control indicator, 6-6

Print-controlled file, 6-4, 6-23
Printed report COBRG (sample), 9-24
Print-files, 6-22
Procedure Division, 1-7
Processing I/O errors, 6-34
Punched cards, 2-1

Random access, 6-7
Read-ahead, 6-13
Reading files written by FORTRAN or MACRO, 6-33
Reading relative files, 6-25
Reading sequential files, 6-21
READ NEXT statement, 6-26, 6-28
READ statement, 6-21, 6-26
READ with key, 6-27
Record blocking, 6-5, 6-8
RECORD CONTAINS clause, 6-4
Record positions, numbered, 6-6
Records (definition), 3-3
Records, fixed length, 6-3, 6-7
Record size, relative files, 6-7
Record size sequential files, 6-3
Record spanning, 6-5, 6-8
Record spanning indicator, 6-6
Records, variable-length, 6-3, 6-7, 6-21
Recursive USE procedures, 6-34
Reference, COBOL formats, A-1
Reference formats, 2-2, 8-1
REFORMAT command string, 9-29
REFORMAT, error messages, 9-30
REFORMAT, summary, 1-9
REFORMAT utility program, 8-1, 9-28
Relational operator, 3-5
Relation condition, 3-4
Relation tests, 3-4, 4-5
Relative file, deleting records from a, 6-28
Relative file organization, 6-6
Relative file, rewriting records into a, 6-27
Relative files, closing, 6-29
Relative files, empty record positions, 6-7
Relative files, media restrictions, 6-11
Relative files, opening, 6-25
Relative files, reading, 6-25
Relative files, record size, 6-7
Relative indexing, 5-12
Relative I/O file status key values (table), 6-36
Relative I/O statements, 6-24
Relative key, 6-7, 6-26
RELATIVE KEY data item, 6-29
Replacement argument, 3-64
Replacement argument list, 3-54
Replacement value, 3-54
REPLACING arguments, INSPECT statement, 3-41
REPLACING phrase, INSPECT statement, 3-52
REPLACING phrase, search argument, 3-52
RESERVE ... ALTERNATE AREAS clause, 6-13
Restrictions, PDP-11 COBOL field release version 1.00, C-1
REWRITE statement, 6-27
REWRITE statements, 6-22
Rewriting records into a relative file, 6-27
Rewriting records into sequential files, 6-22
Rounded phrase, 4-13
Run-time error messages, 6-38
RUNTIME file I/O error procedures, 6-34

SAME AREA clause, 6-13, 6-23, 6-30
SAME RECORD AREA clause, 6-11
Saving access time, 6-10
Saving disk space, 6-10
Search argument, 3-53
Search argument, ALL, 3-53
Search argument, CHARACTERS, 3-53
Search argument, FIRST, 3-53
Search argument, LEADING, 3-53
Search argument, REPLACING phrase, 3-52
Sections, 1-7
Sequence numbers, 2-3
Sequential file organization, 6-2
Sequential files, closing, 6-23
Sequential files, device restrictions, 6-6
Sequential files, opening, 6-19
Sequential files, record size, 6-3
Sequential I/O statements, 6-19
Sequential I/O file status key values (table), 6-36
Sequential OPEN modes, 6-19
SET statement, 5-13
SET statement, legal data movement with the, 5-13
Setting the scanner, INSPECT operation, 3-42
Shared file, 6-17
Sharing buffer space among files, 6-13
/SHR switch, 6-17
Sign conventions, 4-2
Sign determination, 4-4
Signed COMP items, 4-2
Signed DISPLAY items, 4-2
Signed quantities, 4-2
Sign tests, 4-6
Simplifying conditionals, 8-3
SIZE ERROR phrase, 4-14
Slack bytes, 5-7
Slash (/), 2-4, 2-7
SORT summary, 1-9

Source program, COBRG (sample), 9-18
Source program file specification, 2-21
Special characters, 3-3
Specification lines, COBRG, 9-3
Specification lines, COBRG (sample), 9-16
Specifying the next record to be read, 6-28
Splitting a word or numeric literal, 2-7
Splitting non-numeric literals, 2-4, 2-7
Standard disk block, 6-8
START statement, 6-28
Status key, 6-34
Storage files, 6-23
STRING statement, 3-12
STRING statement, common errors, 3-20
STRING statement, multiple sending fields, 3-31
STRING statement, subscripting the delimiter, 3-19
STRING statements, subscripted fields in, 3-18
Subscripted fields in INSPECT statements, 3-44
Subscripted fields in STRING statements, 3-18
Subscripted fields in UNSTRING statements, 3-34
Subscripted moves, 3-11
Subscripting a multi-dimensional table, 5-8
Subscripting and indexing, 5-7
Subscripting the delimiter, STRING statement, 3-19
Subscripting with data-names, 5-9
Subscripting with indexes, 5-10
Subscripting with index-name items, 5-11
Subscripting with literals, 5-8
SUBTRACT statement, 4-16
Switch, /CL:n, 6-17
Switch, /CO:n, 6-17
Switches, 6-17
Switches, compiler, 2-22
Switch, /SHR, 6-17
SYNCHRONIZED clause, 5-2
System errors, compiler, 10-1

Table handling, 5-1
Table, subscripting a multi-dimensional, 5-8
Tab stop, 2-5
Tab stops, 2-8
Tally argument, INSPECT statement, 3-45
Tally argument list, INSPECT statement, 3-46
Tally counter, INSPECT statement, 3-45
TALLYING arguments, INSPECT statement, 3-41
TALLYING phrase, INSPECT statement, 3-44
TALLYING phrase, UNSTRING statement, 3-32
Terminal format, 2-5, 8-1
Testing non-numeric fields, 3-4
Testing numeric fields, 4-5
The ACCESS MODE clause, 6-1
The COBOL source program, 1-6
The comparison operation, 3-6
The COPY REPLACING statement, 2-14
The COPY statement, 2-10
The history of COBOL, 1-1
The INSPECT operation, 3-41
The PDP-11 COBOL compiler, 1-4
The PERFORM statement, use of, 8-6
The sample program, COBRG, 9-15
Three data classes, 3-1
TIME, 6-31
Total specification, COBRG, 9-12
Truncation, 4-12
Two buffers, 6-13
Typical diagnostic message, 10-2

Unblocked, 6-5
UNSTRING statement, 3-20
UNSTRING statement, common errors, 3-36
UNSTRING statement, COUNT phrase, 3-28
UNSTRING statement, DELIMITED BY phrase, 3-23
UNSTRING statement, DELIMITER phrase, 3-29
UNSTRING statement, multiple delimiters, 3-27
UNSTRING statement, multiple receiving fields, 3-21
UNSTRING statement, OVERFLOW phrase, 3-33
UNSTRING statement, POINTER phrase, 3-30
UNSTRING statements, SUBSCRIPTED fields in, 3-34
UNSTRING statement, TALLYING phrase, 3-32
UPON phrase, 6-31
Usages, DISPLAY/COMP, 4-1
USE statement, 6-34
Use of punctuation, 8-5
USE procedures, recursive, 6-34
Using the ALTER statement, 8-5
Use of level 88 condition-names, 8-9

Use of the PERFORM statement, 8-6
Using the ACCEPT statement, 6-30
Using the compiler, 2-18
Utility programs PDP-11 COBOL, 9-1

Warning diagnostics (level W), 10-3
WITH NO ADVANCING phrase, 6-32
Word boundaries, 3-3
Working-Storage Section, 1-7
Write-behind, 6-13
WRITE statement, 6-22, 6-27
WRITE statement, ADVANCING option, 6-22
Writing files for FORTRAN or MACRO, 6-32
Writing records in a relative file, 6-27
Writing sequential files, 6-22

Zeroes, literal with leading or trailing, 4-9

Valid records, 6-24
VALUE OF ID clause, 6-14, 6-16
Variable-length blocks, 6-5
Variable-length format records, 6-4
Variable-length records, 6-3, 6-7, 6-21
Version 1.00 PDP-11 COBOL field release restrictions, C-1

HOW TO OBTAIN SOFTWARE INFORMATION

SOFTWARE NEWSLETTERS, MAILING LIST

The Software Communications Group, located at corporate headquarters in Maynard, publishes newsletters and Software Performance Summaries (SPS) for the various Digital products. Newsletters are published monthly, and contain announcements of new and revised software, programming notes, software problems and solutions, and documentation corrections. Software Performance Summaries are a collection of existing problems and solutions for a given software system, and are published periodically. For information on the distribution of these documents and how to get on the software newsletter mailing list, write to:

Software Communications
P. O. Box F
Maynard, Massachusetts 01754

SOFTWARE PROBLEMS

Questions or problems relating to Digital's software should be reported to a Software Support Specialist. A specialist is located in each Digital Sales Office in the United States. In Europe, software problem reporting centers are in the following cities.

Reading, England	Milan, Italy
Paris, France	Solna, Sweden
The Hague, Holland	Geneva, Switzerland
Tel Aviv, Israel	Munich, West Germany

Software Problem Report (SPR) forms are available from the specialists or from the Software Distribution Centers cited below.

PROGRAMS AND MANUALS

Software and manuals should be ordered by title and order number. In the United States, send orders to the nearest distribution center.

Digital Equipment Corporation	Digital Equipment Corporation
Software Distribution Center	Software Distribution Center
146 Main Street	1400 Terra Bella
Maynard, Massachusetts 01754	Mountain View, California 94043

Outside of the United States, orders should be directed to the nearest Digital Field Sales Office or representative.

USERS SOCIETY

DECUS, Digital Equipment Computer Users Society, maintains a user exchange center for user-written programs and technical application information. A catalog of existing programs is available. The society publishes a periodical, DECUSCOPE, and holds technical seminars in the United States, Canada, Europe, and Australia. For information on the society and membership application forms, write to:

DECUS	DECUS
Digital Equipment Corporation	Digital Equipment, S.A.
146 Main Street	81 Route de l'Aire
Maynard, Massachusetts 01754	1211 Geneva 26
	Switzerland

READER'S COMMENTS

NOTE: This form is for document comments only. Problems with software should be reported on a Software Problem Report (SPR) form (see the HOW TO OBTAIN SOFTWARE INFORMATION page).

Did you find errors in this manual? If so, specify by page.

_____

_____

_____

_____

_____

_____

Did you find this manual understandable, usable, and well-organized? Please make suggestions for improvement.

_____

_____

_____

_____

_____

_____

Is there sufficient documentation on associated system programs required for use of the software described in this manual? If not, what material is missing and where should it be placed?

_____

_____

_____

_____

_____

_____

Please indicate the type of user/reader that you most nearly represent.

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Non-programmer interested in computer concepts and capabilities

Name _____ Date _____

Organization _____

Street _____

City _____ State _____ Zip Code or Country _____

If you do not require a written reply, please check here. ☐

---------- Fold Here ----------

---------- Do Not Tear - Fold Here and Staple ----------

FIRST CLASS
PERMIT NO. 33
MAYNARD, MASS.

BUSINESS REPLY MAIL
NO POSTAGE STAMP NECESSARY IF MAILED IN THE UNITED STATES

Postage will be paid by:

digital

Software Communications
P. O. Box F
Maynard, Massachusetts 01754

DIGITAL EQUIPMENT CORPORATION, Maynard, Massachusetts, Telephone: (617) 897-5111 • ARIZONA, Phoenix • CALIFORNIA, Sunnyvale, Santa Ana, Los Angeles, San Diego and San Francisco (Mountain View) • COLORADO, Engelwood • CONNECTICUT, Meriden • DISTRICT OF COLUMBIA, Washington (Riverdale, Md.) • FLORIDA, Orlando • GEORGIA, Atlanta • ILLINOIS, Northbrook • INDIANA, Indianapolis • LOUISIANA, Metairie • MARYLAND, Riverdale • MASSACHUSETTS, Cambridge and Waltham • MICHIGAN, Ann Arbor and Detroit (Southfield) • MINNESOTA, Minneapolis • MISSOURI, Kansas City and Maryland Heights • NEW JERSEY, Fairfield, Metuchen and Princeton • NEW MEXICO, Albuquerque • NEW YORK, Huntington Station, Manhattan, New York, Syracuse and Rochester • NORTH CAROLINA, Durham/Chapel Hill • OHIO, Cleveland, Dayton and Euclid • OKLAHOMA, Tulsa • OREGON, Portland • PENNSYLVANIA, Bluebell, Paoli and Pittsburgh • TENNESSEE, Knoxville • TEXAS, Dallas and Houston • UTAH, Salt Lake City • WASHINGTON, Bellevue • WISCONSIN, Milwaukee • ARGENTINA, Buenos Aires • AUSTRALIA, Adelaide, Brisbane, Crows Nest, Melbourne, Norwood, Perth and Sydney • AUSTRIA, Vienna • BELGIUM, Brussels • BRAZIL, Rio de Janeiro, Sao Paulo and Porto Alegre • CANADA, Alberta, Vancouver, British Columbia; Hamilton, Mississauga and Ottawa, Ontario; and Quebec • CHILE, Santiago • DENMARK, Copenhagen and Hellerup • FINLAND, Helsinki • FRANCE, Grenoble and Rungis • GERMANY, Cologne, Hannover, Frankfurt, Munich and Stuttgart • INDIA, Bombay • ISRAEL, Tel Aviv • ITALY, Milano • JAPAN, Osaka and Tokyo • MEXICO, Mexico City • NETHERLANDS, The Hague • NEW ZEALAND, Auckland • NORWAY, Oslo • PHILIPPINES, Manila • PUERTO RICO, Miramar and Santurce • REPUBLIC OF CHINA, Taiwan • SCOTLAND, West Lothian • SPAIN, Barcelona and Madrid • SWEDEN, Solna and Stockholm • SWITZERLAND, Geneva and Zurich • UNITED KINGDOM, Birmingham, Bristol, Edinburgh, London, Manchester, Reading and Warwickshire • VENEZUELA, Caracas

PRINTED IN USA