

COBOL/3000 COMPILER

Reference Manual



HEWLETT-PACKARD COMPANY
5303 STEVENS CREEK BLVD., SANTA CLARA, CALIFORNIA 95050

PART NO. 32213-90001
PRODUCT NO. 32213B

Printed in U.S.A. 6/76
Update No. 1 Incorporated 3/77

NOTICE

The information contained in this document is subject to change without notice.

HEWLETT-PACKARD MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Hewlett-Packard shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance or use of this material.

Hewlett-Packard assumes no responsibility for the use or reliability of its software on equipment that is not furnished by Hewlett-Packard.

This document contains proprietary information which is protected by copyright. All rights are reserved. No part of this document may be photocopied, reproduced or translated to another program language without the prior written consent of Hewlett-Packard Company.

HP Computer Museum
www.hpmuseum.net

For research and education purposes only.

LIST OF EFFECTIVE PAGES

The List of Effective Pages gives the date of the current edition and of any pages changed in updates to that edition.

Second Edition Jul 1975

Changed Pages	Effective Date	Changed Pages	Effective Date
Title	Jun 1976	7-22	Jun 1976
ii to v	Jun 1977	8-6	Jun 1977
vii to viii	Jun 1977	8-9	Jun 1977
x to xi	Jun 1977	8-11 to 8-12	Jun 1977
1-4	Jun 1977	8-25	Jun 1977
1-14	Jun 1977	8-27 to 8-29	Jun 1977
2-3	Jun 1977	9-1	Jun 1977
2-6	Jun 1977	10-5	Jun 1977
4-1 to 4-2	Jun 1977	10-21 to 10-22	Jun 1977
4-6 to 4-7	Jun 1977	10-28	Jun 1976
4-9	Jun 1977	11-1 to 11-3	Jun 1977
5-2	Jun 1977	A-2	Jun 1976
5-4	Jun 1977	A-4 to A-5	Jun 1976
5-6	Jun 1977	A-7 to A-7d	Jun 1977
6-3 to 6-6	Jun 1977	B-8 to B-9	Jun 1977
6-8	Jun 1977	B-16	Jun 1977
6-11	Jun 1977	C-4	Jun 1976
6-14	Jun 1977	C-6	Jun 1977
6-17	Jun 1977	C-35 to C-36	Jun 1977
6-28	Jun 1977	C-42 to C-43	Jun 1977
6-37	Jun 1977	C-44	Jun 1976
6-39	Jun 1977	C-45 to C-47	Jun 1977
7-2 to 7-4	Jun 1977	F-10	Jun 1977
7-8	Jun 1977	G-1	Jun 1976
7-20	Jun 1977	Index 1 to Index 5	Jun 1977

PRINTING HISTORY

New editions incorporate all update material since the previous edition. Update packages, which are issued between editions, contain additional and replacement pages to be merged into the manual by the customer. The date on the title page and back cover changes only when a new edition is published. If minor corrections and updates are incorporated, the manual is reprinted but neither the date on the title page and back cover nor the edition change.

First Edition	Oct 1974
Second Edition	Jul 1975
Update Package #1	Jun 1976
Update Incorporated	Mar 1977
Update Package #2	Jun 1977

Preface

This publication is the reference manual for the HP 3000 COBOL programming language (COBOL/3000). COBOL/3000 is an implementation of ANS COBOL for the HP 3000 Computer System. The standard for this language is American National Standard COBOL X3.23-1968 as approved by the American National Standards Institute.

American National Standard COBOL incorporates eight processing modules:

- Nucleus
- Table Handling
- Sequential Access
- Random Access
- Sort
- Segmentation
- Library
- Report Writer

COBOL/3000 is an implementation of all of these modules except for the Report Writer module.

The reader of this manual should have a general knowledge of the HP 3000 Computer System and the MPE/3000 Operating System. Additional information can be found in the following publications:

- *MPE Commands Reference Manual* (30000-90009)
- *MPE Intrinsic Reference Manual* (30000-90010)
- *System Reference Manual* (30000-90020)
- *Using the HP 3000* (03000-90121)
- *Systems Programming Language Reference Manual* (30000-90024)

For 3000 systems which are not Series II, the differences in manuals should be noted:

- Whenever the *MPE Commands Reference Manual* or the *MPE Intrinsic Reference Manual* is referenced in this manual, use the *MPE/3000 Operating System Reference Manual* (32000-90002).
- The *HP 3000 Computer System Reference Manual* (03000-90019) is used in place of the *System Reference Manual*.

This manual is a reference manual rather than a tutorial text for programmers using COBOL for the first time. Programmers desiring a tutorial text are referred to: *A Guide for COBOL Programming* by Daniel D. McCracker and Umberto Garbassi (2nd Edition, New York, Wiley Interscience 1970).

Acknowledgment

Any organization interested in using the COBOL specifications as the basis for an instruction manual or for any other purpose is free to do so. However, all such organizations are requested to reproduce this section as part of the introduction to the document. Those using a short passage, as in a book review, are requested to mention 'COBOL' in acknowledgment of the source, but need not quote this entire section.

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 COBOL 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.

Procedures have been established for the maintenance of COBOL. Inquiries concerning the procedures for proposing changes should be directed to the Executive Committee of the Conference on Data Systems Languages.

The authors and copyright holders of the copyrighted material used herein

FLOW-MATIC (Trademark of Sperry Rand Corporation), Programming for the UNIVAC[®] 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.

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.

This complete American National Standard edition of COBOL may not be reproduced without permission of the American National Standards Institute.

Contents

PART 1 INTRODUCTION

SECTION I Introduction to COBOL/3000

DIVISIONS OF A COBOL PROGRAM	1-2
COBOL FORMATS	1-3
Uppercase Words	1-3
Lowercase Words	1-4
Brackets	1-4
Braces	1-5
Ellipsis	1-5
Special Characters	1-5
Punctuation	1-6
COBOL CHARACTER SET	1-6
Collating Sequence	1-7
Complete ASCII Character Set	1-8
Conversion Considerations	1-9
CONSTANTS	1-9
FIGURATIVE CONSTANTS	1-10
SPECIAL REGISTERS	1-11
METHODS OF DATA REFERENCE	1-12
Qualification	1-12
Indexing	1-14
Subscripting	1-15

SECTION II Coding Source Programs

CODING FORM	2-1
Sequence Numbers	2-2
Area A and Area B	2-2
Continued Lines	2-3
Comment Lines	2-3
Identification Field	2-3
CODING CONVENTIONS	2-4
USING THE EDITOR	2-6

PART 2. PROGRAM DIVISIONS

SECTION III Identification Division

GENERAL FORMAT	3-1
GENERAL RULES	3-1
RULES FOR COMMENT-ENTRY PARAGRAPHS	3-2

SECTION IV Environment Division

GENERAL FORMAT	4-1
CONFIGURATION SECTION	4-2
SPECIAL-NAMES Paragraph	4-2
Decimal Point Clause	4-4
INPUT-OUTPUT SECTION	4-4
SELECT (Sequential Files)	4-5
SELECT (Sort-Files)	4-7
SELECT (Random Files)	4-9
I-O-CONTROL Paragraph	4-10

SECTION V Data Description Conventions

FILE SECTION	5-1
WORKING-STORAGE SECTION	5-2
LINKAGE SECTION	5-2
CONCEPT OF LEVELS WITHIN DATA DEFINITIONS	5-2
General Structure of Data Description Entries	5-3
Rules for Coding Levels	5-4
STANDARD DATA POSITIONING	5-5
Classes of Data	5-5
Standard Positioning of Alphabetic and Alphanumeric Data	5-5
Standard Positioning of Numeric Data	5-6

SECTION VI Data Description Clauses

FILE DESCRIPTION ENTRIES	6-1
BLOCK CONTAINS CLAUSE	6-3
DATA RECORDS CLAUSE	6-3
LABEL RECORDS CLAUSE	6-4
RECORD CONTAINS CLAUSE	6-5
RECORDING MODE CLAUSE	6-5
VALUE OF CLAUSE	6-6
DATA DESCRIPTION	6-6
DATA NAME OR FILLER CLAUSE	6-8
BLANK WHEN ZERO CLAUSE	6-9
JUSTIFIED CLAUSE	6-11
OCCURS CLAUSE	6-12
Rules for OCCURS Clause	6-13
DEPENDING ON Option	6-14
KEY Option	6-15
INDEXED BY Clause	6-16
PICTURE CLAUSE	6-17
PICTURE Clause Character-String	6-19
Character-String Symbols	6-20
Editing Conventions	6-23

Sequence of Character-String Symbols	6-29
Precedence of Symbols Used in the PICTURE Clause	6-30
REDEFINES CLAUSE	6-31
RENAMES CLAUSE	6-32
SYNCHRONIZED CLAUSE	6-34
USAGE CLAUSE	6-36
DISPLAY Option	6-37
COMPUTATIONAL Options	6-38
Storing Numeric Data in Magnetic Files	6-40
INDEX Option	6-40
VALUE CLAUSE	6-41
SECTION VII Procedure Division	
CONVENTIONS	7-2
GENERAL STRUCTURE AND FORMAT	7-3
CONDITIONAL STATEMENTS	7-5
ARITHMETIC EXPRESSIONS	7-6
CONDITIONS	7-9
COMPARISON OF OPERANDS	7-11
PERMISSIBLE RELATIONAL COMPARISONS	7-13
RELATION CONDITION	7-15
CLASS CONDITION	7-17
CONDITION-NAME CONDITION	7-18
SIGN CONDITION	7-19
NOT CONDITION	7-20
INTERPROGRAM COMMUNICATION	7-21
COMPILE-TIME OPTIONS	7-22
COMMON OPTIONS IN PROCEDURAL STATEMENTS	7-22
ROUNDED Option	7-23
SIZE ERROR Option	7-24
CORRESPONDING Option	7-25
SECTION VIII Procedure Division Statements	
ADD	8-2
ALTER	8-4
CALL	8-5
COMPUTE	8-9
COPY	8-10
DIVIDE	8-13
ENTER	8-16
ENTRY	8-17
EXAMINE	8-18
EXIT	8-20
EXIT PROGRAM	8-21
GOBACK	8-22
GO TO	8-23
IF	8-25
MOVE	8-27
MULTIPLY	8-32
NOTE	8-34
PERFORM	8-35

SEARCH	8-47
SET	8-52
STOP	8-55
SUBTRACT	8-57
PART 3. INPUT-OUTPUT	
SECTION IX I-O Conventions	
INTERACTION WITH MPE	9-1
SECTION X Input-Output Statements	
ACCEPT	10-2
CLOSE	10-5
DISPLAY	10-7
OPEN	10-9
READ	10-11
RELEASE	10-15
RETURN	10-16
SEEK	10-17
SORT	10-18
USE	10-24
WRITE	10-28
SECTION XI Locking and Unlocking Files	
PART 4. APPENDICES	
APPENDIX A Using the COBOL/3000 Compiler	
REFERENCING FILES	A-3
COMPILING/PREPARING/EXECUTING PROGRAMS	A-7
:COBOL	A-8
:COBOLPREP	A-10
:COBOLGO	A-12
:PREP	A-13
:PREPRUN	A-15
:RUN	A-18
APPENDIX B Compiler Subsystem Commands	
SYNTAX AND FORMAT	B-1
COMMAND SUMMARY	B-6
LISTING AND COMPILATION OPTIONS (\$CONTROL)	B-6
CONDITIONAL COMPILATION (\$IF)	B-11
SOFTWARE SWITCHES FOR CONDITIONAL COMPILATION (\$SET)	B-12
PAGE TITLE IN STANDARD LISTING (\$TITLE)	B-14
PAGE TITLE AND EJECTION (\$PAGE)	B-14
SOURCE TEXT MERGING AND EDITING (\$EDIT)	B-15
APPENDIX C COBOL Diagnostic Messages	
COBOL COMPILER DIAGNOSTIC MESSAGES	C-1
OBJECT PROGRAM DIAGNOSTIC MESSAGES	C-4
APPENDIX D COBOL/3000 Reserved Words	
	D-1

APPENDIX E	COBOL Language Formats	E-1
APPENDIX F	Sample COBOL/3000 Program	F-1
APPENDIX G	Software Arithmetic Trap for Interprogram Communication	G-1
INDEX		

FIGURES

Figure F-1. Source Program Listing	F-7
Figure F-2. Symbol Table Map	F-8
Figure F-3. P-Map	F-10
Figure F-4. Object Program Output	F-11

TABLES

Table 8-1. Permissible Moves	8-29
Table B-1. Compiler Subsystem Command Summary	B-6
Table C-1. COBOL Compiler Diagnostic Messages	C-5
Table C-2. Object Program Diagnostic Messages	C-43
Table C-3. COBOL Status Parameter Values Returned by KSAM	C-47

PART 1
Introduction

SECTION I

Introduction to COBOL/3000

COBOL is one of a number of high-level computer languages. Such languages, because they are problem oriented and relatively free of hardware constraints, allow the programmer to concentrate his efforts on problem solving.

Because COBOL source statements use common English words and conventional arithmetic symbols, the language is to some extent self-documenting. The following is a typical COBOL sentence:

MULTIPLY HOURS-WORKED BY PAY-RATE GIVING REGULAR-PAY.

Before the sentence above can perform useful work within the computer, it must be translated into machine instructions. Such language translation is the job of a special program, the *compiler*. The COBOL compiler accepts source statements as data. From this data the compiler produces a listing of the source statements with any errors flagged by diagnostic messages. Also, the compiler translates the COBOL statements into the various machine instructions required to accomplish the desired job. The translated statements form the *object program*.

After the entire source program has been translated, the object program can be loaded into memory and run (or used), or it can be stored in a library for future use. If the programmer wishes to change the object program, he must recompile the program. After the object program produces the desired results, the source program is no longer required. At this point, what the programmer thinks of as the "program" is the set of machine instructions generated by the compiler; the source program is required only for backup purposes and so that the program may be changed in the future.

DIVISIONS OF A COBOL PROGRAM

Every COBOL program consists of four *divisions*:

- The *Identification Division* assigns a name to the program and allows the programmer to enter other documentary information, such as the programmer's name, the date the program was written, and so on.
- The *Environment Division* defines the computer used to compile and then run this program. Also, the Environment Division provides a portion of the program where each computer manufacturer may require specific information concerning input-output devices to be used when running the program.
- The *Data Division* provides the compiler with a detailed description of the characteristics of every data item used within the program. The major characteristics of a data item include its length, its name, its decimal position location (if any), its storage format (allowable contents), and (if the item is part of an external record) its relationship to other data items in the record.
- The *Procedure Division* contains the COBOL statements required to solve a particular data processing problem. The statement is the basic unit of the Procedure Division. Each statement must be syntactically valid combination of words and symbols starting with a COBOL verb. There are three types of statements: an *imperative statement* causes some action to take place (ADD, for example); *conditional statements* cause a test for some condition; and *compiler-directing statements* such as NOTE, which tells the compiler not to generate object code for the associated sentence.

A *sentence* is made up of one or more statements. A sentence is terminated by a period followed by a space.

A sentence or group of sentences may be assigned a name to form a *paragraph*. If the programmer wishes to alter the logical flow of the program, he must tell the compiler the name of the paragraph to which control is to be transferred. A paragraph begins with the paragraph-name (the paragraph-name must begin in columns eight through eleven on the coding sheet, and must be terminated with a period followed by a space). The paragraph ends immediately before the next paragraph-name. A paragraph may also be terminated by a section-name, the end of the Procedure Division, or — in the Declarative portion — by the words END DECLARATIVES (the Declarative portion is explained below).

One or more paragraphs form a *section*. If the programmer does not specify any sections, then the entire Procedure Division is a single section. As an option, the programmer may divide the Procedure Division into more than one section. Each section must begin with a section heading (a section-name assigned by the programmer followed by the word SECTION, and, optionally, a priority number, and terminated with a period followed by a space). When a program is divided into sections, the programmer should attempt to group functions that are logically related (housekeeping procedures, for example) in the same section. This helps the MPE Operating System to utilize its virtual memory more efficiently, since the less frequently used sections need not be held in real memory. In addition, a program which is logically sectioned is usually easier to debug and maintain. A section ends immediately before the next section-name. A section may also be terminated with the physical end of the program.

The general term *procedure-name* refers to either a paragraph-name or a section-name.

As an option, the Procedure Division may contain a special-purpose portion, the *Declarative* portion. Declarative procedures are user-written procedures that are in addition to the normal system software routines for label processing or error handling. Such procedures are performed only under the conditions specified by a USE statement.

COBOL FORMATS

Throughout this manual, general formats are used to indicate the sequence of clauses within a COBOL program. The elements of these formats are uppercase words, underlined uppercase words, lowercase words, brackets, braces, punctuation marks, ellipses, and certain special characters.

The meaning of these symbols within the formats is explained in the following examples.

Uppercase Words



An uppercase word in a format represents a *reserved word*. Reserved words have particular meaning to the compiler and must be spelled exactly as shown in order to be interpreted correctly by the compiler.

Uppercase words in a format may be underlined. If so, the word is a *key word*. A key word is always required if the clause or statement of which it is a part is to be included in the program.

Uppercase words that are not underlined are optional words that the programmer may include or omit at will. These words, which have no effect on the object program, help to make the program more readable and easy to understand.

EXAMPLE

In the following example, the words DATA and RECORDS (or RECORD) are key words. The word ARE (or IS) is not a key word and may be included if desired for readability.

$$\underline{\text{DATA}} \left\{ \begin{array}{l} \underline{\text{RECORDS}} \text{ ARE} \\ \underline{\text{RECORD}} \text{ IS} \end{array} \right\}$$

Lowercase Words

Each lowercase word in a format represents a word that is to be supplied by the programmer, for example, program-names, data-names, procedure-names, and section-names. In general, these words may be up to thirty characters in length and may contain only letters of the alphabet, the digits 0 through 9, and the hyphen (-) character. The word must not begin or end with the hyphen character, and cannot contain an embedded space.

In specific formats, the rules for programmer-supplied words are more restrictive than the general rules for words stated above. A program-name, for example, should contain no more than fifteen characters. The rules for these words are fully explained along with the particular statement or clause.

A lowercase word in a format may also represent a literal to be supplied by the programmer. Literals are explained later in this section under "Constants."

EXAMPLE

In the following format, the programmer must supply a procedure-name, which may be either a paragraph-name or section-name in the Procedure Division. The example shows both the format, and a completed statement in which the programmer has supplied the procedure-name UPDATE-ROUTINE.

```
GO TO procedure-name.      (this is the format)
GO TO UPDATE-ROUTINE.     (this is a coding example)
```

Brackets

When a portion of a format is enclosed in brackets, it is optional to statement or clause. If the bracketed portion of the format is not required for the program, it is omitted; if applicable, then the bracketed portion must be included in the program.

EXAMPLE

Brackets are frequently used to indicate the possibility of multiple operands as shown in the following ADD statement format (this is not a complete format):

```
ADD  identifier-1 [, identifier-2] [, identifier-3] . . . TO
      identifier-m [, identifier-n] . . .
```

Braces

Items enclosed in braces indicate that one of the items must be included in the program. The alternatives are stacked vertically.

Similarly, multiple items enclosed in brackets also indicate a choice of items. However, all items enclosed in brackets are optional.

EXAMPLE

The following is the complete format for the ADD statement partially illustrated in the previous example:

$$\begin{array}{l} \underline{\text{ADD}} \quad \left\{ \begin{array}{l} \text{identifier-1} \\ \text{literal-1} \end{array} \right\} \left[\begin{array}{l} , \text{identifier-2} \\ , \text{literal-2} \end{array} \right] \dots \underline{\text{TO}} \text{identifier-m} \left[\underline{\text{ROUNDED}} \right] \\ \quad \left[, \text{identifier-n} \left[\underline{\text{ROUNDED}} \right] \right] \dots \\ \quad \left[; \text{ON } \underline{\text{SIZE ERROR}} \text{ imperative-statement} \right] . \end{array}$$

According to this format, the program must add at least one item (identifier-1 or literal-1) to at least one other item (identifier-m). As an option, the statement may be used to add a series of items (identifier-1, identifier-2, etc.) to a series of other items (identifier-m, identifier-n, etc.).

Ellipsis

The ellipsis (. . .) indicates that the item immediately preceding the ellipsis may be repeated at the user's option. The preceding item is always enclosed in brackets to help remove any ambiguity as to which item may be repeated. For example, the construction] . . .] . . . indicates that the item immediately preceding the first ellipsis may be repeated; the entire statement or clause associated with the second ellipsis may also be repeated.

Special Characters

The symbols <, >, and = are used in conditional statements to represent the key words LESS THAN, GREATER THAN, and EQUAL, respectively. Although these represent key words, they are not underlined in the formats.

Punctuation

The punctuation symbols semicolon, comma, and period are shown in the formats. The semicolon is always optional and may be used to improve readability. The comma is also optional in every case but one: when a series of indexes or subscripts are enclosed within a single pair of parentheses, the indexes or subscripts must be separated by a comma followed by a space. The programmer must always insert periods where they are required by the format.

Note: In this reference manual, two special characters (Δ and \wedge) are used for illustrative purposes in examples and are not a part of the COBOL language. In many examples, it is useful to have a character which explicitly denotes the space character; the Δ character is used to represent a space in this manual. Also, the period is not used to represent the decimal point in memory since the computer does not carry actual decimal points in memory; instead the caret (\wedge) is used to represent the implied decimal point.

COBOL CHARACTER SET

Although the HP 3000 Computer System can recognize 128 characters, the COBOL language character set is restricted to the 52 characters most common for business purposes. This enhances the universality of the language as nearly all computers include these 52 characters. However, this compiler will accept any valid HP 3000 character when an item is defined as alphanumeric.

Note: Lowercase letters are converted to uppercase letters by the compiler when they appear outside literals.

	CLASS	CHARACTER	MEANING	
alpha- numeric	numeric	0, 1, . . . , 9	digit	
	alphabetic	A, B, . . . , Z and space	letter	
	special characters		+	plus sign
			-	minus sign (hyphen)
			*	asterisk
			/	stroke (virgule, slash)
			=	equal sign
			\$	currency sign
			,	comma
			;	semicolon
			.	period (decimal point)
			”	quotation mark (or the substitute character apostrophe)
			(left parenthesis
)	right parenthesis
			>	greater-than symbol
	<	less-than symbol		

Collating Sequence

Each character in the HP 3000 Computer has a unique octal value which establishes the collating sequence of the character set. This sequence conforms to the American Standard Code for Information Interchange (ASCII). The entire set of characters is illustrated on the following page.

Characters in the chart are arranged top to bottom, left to right, according to ascending value. The numbers at the top and left side of the chart indicate the octal values of the characters.

The collating sequence, which is illustrated in the following chart, is based on the relative value of the characters.

Space “ \$ ’ () * + , - . / 0 thru 9 ; < = > A thru Z
 (Lowest) (Highest)

Complete ASCII Character Set

		CONTROL CHARACTERS		GRAPHIC CHARACTERS					
	COL.	0	1	2	3	4	5	6	7
ROW	BITS	000	001	010	011	100	101	110	111
0	0000	NUL	DLE	SP	0	@	P		p
1	0001	SOH	DC1	!	1	A	Q	a	q
2	0010	STX	DC2	"	2	B	R	b	r
3	0011	ETX	DC3	#	3	C	S	c	s
4	0100	EOT	DC4	\$	4	D	T	d	t
5	0101	ENQ	NAK	%	5	E	U	e	u
6	0110	ACK	SYN	&	6	F	V	f	v
7	0111	BEL	ETB	'	7	G	W	g	w
8	1000	BS	CAN	(8	H	X	h	x
9	1001	HT	EM)	9	I	Y	i	y
10	1010	LF	SUB	*	:	J	Z	j	z
11	1011	VT	ESC	+	;	K	[k	{
12	1100	FF	FS	,	<	L	\	l	
13	1101	CR	GS	-	=	M]	m	}
14	1110	SO	RS	.	>	N	^	n	-
15	1111	SI	US	/	?	O	_	o	DEL

Conversion Considerations

When programs originally written for other systems are run on the HP 3000 Computer System, variations in the collating sequence may cause variations in output. For example, the Binary Coded Decimal (BCD) and the related Extended Binary Coded Decimal Information Code (EBCDIC) both collate the letters of the alphabet before (or lower than) the digits 0 through 9. This is substantially different from the ASCII collating sequence (where digits are collated before letters), and may require program modifications. Also, a number of special characters collate differently in the various codes.

To simplify conversion, the COBOL/3000 language allows the substitution of the quotation mark (") by the apostrophe (') in printed output and in source statements. The standard quotation (") is preferred in COBOL/3000. A compiler subsystem command is provided for this purpose. Check the punch card codes for these characters as the codes vary between keypunches. The compiler accepts a 7-8 punch for the quotation mark, 5-8 for the apostrophe.

Substitution of the apostrophe alters the collating sequence slightly; the apostrophe collates just after the dollar sign; the quotation mark just before the dollar sign.

CONSTANTS

A *constant* is a data item whose value is not changed during execution of the object program. A heading which is printed on each page of a report is a typical constant. Rules for coding constants are explained under "Coding Form" in Section II.

Literals

In effect, a literal is a constant that is not defined in the Data Division. Instead, the programmer simply codes the desired value directly in the COBOL procedural statement.

Literals may be either numeric or non-numeric.

NUMERIC LITERALS. The programmer codes a numeric literal simply by including the desired value in the COBOL statement (the COBOL formats indicate when literals may be used):

ADD 1 TO PAGE-NUMBER

A numeric literal may contain the digits 0 through 9, the operational signs + and -, and a decimal point. When an operational sign is used, it must be the leftmost character in the literal. Unsigned literals are assumed to be positive. Numeric literals may contain a maximum of 18 digits.

A decimal point is specified by entering the period character anywhere within the literal (except as the rightmost character, where it is interpreted as the period in a sentence).

If a literal conforms to the rules above, but it is enclosed in quotation marks, then it is a non-numeric literal and cannot be used in arithmetic operations.

NON-NUMERIC LITERALS. The programmer codes a non-numeric literal by enclosing the desired character string in quotation marks and then including it in the COBOL statement:

```
MOVE "OVERTIME REPORT" TO HEADING-1.
```

Each non-numeric literal may be from 1 through 120 characters in length and may contain any HP 3000 character except the quotation mark.

The value of a non-numeric literal is the value of the character-string itself, excluding the quotation marks.

FIGURATIVE CONSTANTS

Certain constants occur so often that they have been assigned fixed data-names within the COBOL language. These constants do not require definition in the program, and may be referred to by their assigned data-names:

SPACE SPACES	Represents one or more spaces.
ZERO ZEROS ZEROES	Represents one or more occurrences of the digit 0.
LOW-VALUE LOW-VALUES	Represents one or more characters with the lowest possible value in the HP 3000 collating sequence (all eight bits are OFF).
HIGH-VALUE HIGH-VALUES	Represents one or more characters with the highest possible value in the HP 3000 collating sequence (all eight bits are ON).
QUOTE QUOTES	Represents one or more quotation marks. The figurative constant QUOTE provides a means for coding the quotation mark as a literal in statements such as MOVE QUOTES . . . , etc. Notice that there is no means for coding a quotation mark within a literal. One technique for handling this situation in the data division is:

```
01 QUOTH-THE-RAVEN.  
05 FILLER PIC X VALUE QUOTE.  
05 FILLER PIC X (10) VALUE "NEVERMORE."  
05 FILLER PIC X VALUEQUOTE.
```

A subsequent reference to QUOTH-THE-RAVEN in the procedure division will make available "NEVERMORE."

ALL literal Represents one or more sets of the character or character-string comprising the literal. The literal must be either a non-numeric literal or a figurative constant. When a figurative constant is specified, ALL is redundant but may be used for readability.

The length assigned to a figurative constant is determined according to the following rules:

- When the figurative constant is associated with another data item, as in a VALUE clause or when the constant is moved to or compared with another item, then the figurative constant assumes the same length as the associated item.
- When a figurative constant is not associated with some other item, as when the figurative constant appears in a DISPLAY, EXAMINE, or STOP statement, then the length assigned to the figurative constant is one character. The figurative constant ALL may not be used in these cases.
- In the case of the ALL literal, the specified literal is repeated until the associated item is filled. For example, if FIELD-A is a ten-character item, the statement MOVE ALL "123" TO FIELD-A has the following results:

1231231231

SPECIAL REGISTERS

The COBOL language defines a special register called TALLY, the primary function of which is to store information produced by the EXAMINE statement. In addition to TALLY, this compiler includes two other special registers: CURRENT-DATE and TIME-OF-DAY.

- TALLY is the name of a five-digit unsigned field whose usage is COMPUTATIONAL. Although the primary use of the TALLY register is to store information produced by the EXAMINE statement, it may be used as a data-name whenever an unsigned numeric value with no decimal positions is desired. For example, the TALLY register may be used as a subscript.
- CURRENT-DATE is an eight-character alphanumeric item which is valid only as the sending field in a MOVE or DISPLAY statement. CURRENT-DATE is in MM/DD/YY format (month/day/year). The slashes are included in the data and should not be inserted by the programmer.
- TIME-OF-DAY is a six-character numeric DISPLAY item which is valid only as the sending field in a MOVE or DISPLAY statement. TIME-OF-DAY is in HHMMSS format (hours, minutes, seconds). The data is unedited, but DISPLAY TIME-OF-DAY results in the edited format HH: MM: SS.

The clock time required for execution of the program can be determined by printing out the TIME-OF-DAY register at the beginning and end of the job. However, the clock time required for a job in a multiprogramming environment is not necessarily related to the computer time required for the job and will vary according to the mix of jobs that are active when this program is executed.

METHODS OF DATA REFERENCE

Every identifier used in a COBOL source program must be unique. An identifier is unique if no other name in the program has the identical spelling or if the name is made unique through the use of qualification, indexing, or subscripting.

In the formats for COBOL statements, the general term *identifier* is used to represent a data-name which may be qualified, indexed, or subscripted.

Qualification

Qualification allows the programmer to duplicate data-names and procedure-names within a program. The programmer *must* use identical names for data fields if he intends to make use of the CORRESPONDING option in the Procedure Division.

Format (data-names)

$$\left\{ \begin{array}{l} \text{data-name-1} \\ \text{condition-name-1} \end{array} \right\} \left[\left(\begin{array}{c} \underline{\text{IN}} \\ \underline{\text{OF}} \end{array} \right) \text{data-name-2} \right] \dots$$

Format (procedure-names)

$$\text{paragraph-name} \left[\left(\begin{array}{c} \underline{\text{IN}} \\ \underline{\text{OF}} \end{array} \right) \text{section-name} \right]$$

Qualification is specified by following a data-name or procedure-name with one or more qualifiers, each preceded by IN or OF. Either word may be used since both have the same meaning to the compiler.

Qualification may be used at any time, whether or not it is required to make a reference unique. However, when identical names are used in the program, enough qualification to make the name unique must be specified each time the name is referenced.

Note: For a data-name, there may be more than one qualifier that makes the name unique. Any or all of the qualifiers that make the name unique may be mentioned when the name is referenced; the programmer is required only to make the reference unique.

A data-name cannot be subscripted when it is used as a qualifier.

The only qualifier possible for a paragraph-name is the name of the section of the program which contains the paragraph. Therefore, paragraph-names may be duplicated within the Procedure Division, but not within a section. When a paragraph-name is qualified, the reserved word SECTION is not used with the qualifying section-name. Although a paragraph-name need not be qualified when referred to from within the same section, using the qualifier provides better program documentation.

The qualifier for a data-name is another name within the save record description. The qualifier must precede that data name in the record hierarchy and must have a more significant level number or level indicator. Although identical data names may appear in a single record, a data-name must never appear to qualify itself.

Although identical data-names can appear in a single record, they must not appear unless they can be made unique through qualification.

EXAMPLE

The coding on the left is illegal because the level 05 A cannot be made unique through qualification. (A IN RECORD-1 could be interpreted as a reference to either the level 05 A or the level 10 A.)

The coding on the right is legal because the level 03 E allows the level 05 A to be uniquely qualified (A IN E).

01	RECORD-1.	01	RECORD-2.
05	A.	03	E.
10	B...	05	A.
10	C...	10	B...
05	D.	10	C...
10	A...	03	F.
		05	D.
		10	A...

INDEXING AND SUBSCRIPTING

A table is essentially a list of similar items all with the same data-name. In effect, indexing and subscripting are specialized forms of qualification used to inform the system which item in the list is to be referenced.

Additional information about tables may be found under the OCCURS and INDEXED BY clauses and under the SEARCH, SET, and PERFORM statements.

Indexing

When a data item within a table is accessed by indexing, its data description (or the description of a data item to which it is subordinate) must include the INDEXED BY clause.

There are two types of indexing: direct and relative.

DIRECT INDEXING. For one dimensional tables, the programmer uses the following format to refer to a table item through direct indexing:

data-name (index-name)

Data-name must either be an entry whose data description contains an OCCURS clause or must be subordinate to such an item.

Data-name is always terminated by a single space followed by the left parenthesis.

The name of the desired table-index must be enclosed within the parentheses. No spaces may appear within the parentheses.

For multiple-dimensional tables, each index-name other than the last must be terminated by a comma followed by a single space. The last index-name is terminated by the right parenthesis. No other spaces are permitted within the parentheses.

Note: Index names in multiple-dimensional tables must be specified in order of decreasing inclusiveness.

EXAMPLES

If AMOUNT is a table item indexed by AMOUNT-INDEX, then the following data reference is used to access AMOUNT:

AMOUNT (AMOUNT-INDEX)

If AMOUNT is an item in a three-dimensional table associated with the indexes YEAR-INDEX, MONTH-INDEX, and AMOUNT-INDEX, then the following data reference is used to access AMOUNT:

AMOUNT (YEAR-INDEX,ΔMONTH-INDEX,ΔAMOUNT-INDEX)

RELATIVE INDEXING. The programmer uses the following format to refer to an item in a single dimensional table by relative indexing:

$$\text{data-name} \Delta \left(\text{index-name} \Delta \left\{ \begin{array}{c} + \\ - \end{array} \right\} \Delta \text{integer} \right)$$

The rules for direct indexing also apply to relative indexing. However, within the parentheses must be the index-name, a single space, the arithmetic operator plus or minus, another single space, and an unsigned numeric literal. For a single dimensional table, no other spaces may appear within the parentheses; the right parenthesis terminates the index.

For multiple-dimensional tables, each index other than the last must be terminated by a comma followed by a space. Direct and relative indexes may be mixed within the parentheses. The last index is terminated by the right parenthesis.

When relative indexing is specified, the system accesses the table element which corresponds to the current setting of the index plus or minus the number of occurrences specified by integer. For example, if TABLE-INDEX is set to reference the tenth element within the table, then (TABLE-INDEX Δ - Δ 1) references the ninth element.

Subscripting

Subscripting provides an alternative method for accessing an element within a table. In general, subscripting is less efficient than indexing.

The programmer uses the following format to refer to a table by subscripting:

$$\text{data-name} \Delta \left(\left(\begin{array}{c} \text{literal} \\ \text{subscript-name} \end{array} \right) \right)$$

Data-name must either be an entry whose data description contains an OCCURS clause or must be subordinate to such an item.

Data-name is always terminated by at least one space. The left parenthesis of the subscript must immediately follow this space.

If a literal subscript is specified, the only entry permitted within the parentheses is an unsigned numeric literal. This literal must correspond to an occurrence within the table. Therefore, literal must be greater than zero, but not greater than the number specified in the OCCURS clause associated with the table.

If subscript-name is specified, the subscript must be an elementary numeric item without any positions to the right of the implied decimal point. This item must be defined in either the File Section or the Working-Storage Section of the Data Division. The item must contain a positive value greater than zero but not greater than the number specified in the OCCURS clause associated with the table. For maximum efficiency, the item should be defined as a COMPUTATIONAL SYNCHRONIZED item.

The subscript-name may be qualified, but may not be indexed or subscripted.

When a multi-dimensional table is accessed, the parentheses may contain up to three indexes or subscripts. However, indexes and subscripts may not be mixed within the same set of parentheses.

When multiple subscripts are specified, each subscript except the last must be terminated with a comma followed by a single space. The last subscript is terminated by the right parenthesis.

The Special Register TALLY may be used as a subscript.

Sequence Numbers

Columns 1 through 6 of the coding form are reserved for a sequence number. Sequence numbers must not contain any alphabetic or special characters, and may range from 000000 through 999999.

As compile-time options, the programmer may request the compiler to sequence check or to renumber the source statements. The use of sequence numbers is recommended when punched cards are used for input to the compiler, as it is easy for a card deck to be dropped.

Sequence numbers are especially useful if the program must be recompiled. As a compile-time option, the programmer may request that a copy of the source program be written to a file. Then, if the program needs to be recompiled, the programmer need submit only the statements that are to be changed or added. These statements are merged into the existing program according to sequence number. If the programmer intends to use this feature, he should increment the sequence number by 10 or 100 to allow for the possible addition of new source statements.

The options mentioned above are functions of the compiler subsystem (described in Appendix B).

Area A and Area B

Columns 8 through 72 are reserved for the text of the program. (Column 7 is explained following this discussion of areas A and B.)

Area A, columns 8 through 11, is reserved for the beginning of division headings, section-names, paragraph-names, level indicators, and certain level numbers.

Area B, columns 12 through 72, is reserved for all other COBOL text.

DIVISION HEADINGS. Each division of a COBOL program must begin with a division heading. The division heading must begin in Area A and then continue into Area B. With the exception of the Procedure Division heading, all division headings must be terminated with a period followed by a space, and must be the only entry on the line. The Procedure Division heading may be followed by a space and then a USING clause if this program is to be called by another program.

SECTION HEADING. A section-name must begin in Area A. The name is followed by a space, the word SECTION, and a period. If the program requests segmentation, the word SECTION may be followed by a space, a priority number, and a period. No other text may appear on the same line except a COPY statement, or, in the Declarative portion only, a USE statement.

PARAGRAPH-NAMES. Paragraph-names should begin in Area A. The first sentence in the paragraph must begin in Area B of either this or the next line. All procedural statements must be coded entirely within Area B.

DATA DESCRIPTION ENTRIES. The level indicators FD and SD and the level numbers 01 and 77 must all begin in Area A of the coding form; their associated data description entries — including names — must be coded entirely within Area B.

All other level numbers may begin in either Area A or Area B. Again, their associated data description entries must be coded entirely within Area B.

Continued Lines

Any sentence or entry that requires more than one line must be continued in Area B of the next line.

When a word or a numeric literal is broken from one line to the next, the programmer must place a hyphen (—) in column 7 of the continuation line to indicate that the first nonspace character in Area B is a part of the word or literal broken on the previous line.

When a nonnumeric literal is broken from one line to the next, the programmer must place a hyphen in column 7. Also, a quotation mark must precede the continuation of the literal. The literal may begin anywhere within Area B of the continuation line.

Comment Lines

Explanatory comments may be entered anywhere in the program by placing an asterisk (*) in column 7 of a source line. When column 7 contains an asterisk, the compiler will simply print that line on the program listing. This has no effect on the object program. Comment lines must not appear between a continuation line and the continued line.

Identification Field

Columns 73 through 80 on the coding form are for an identification code. The identification code, if used, appears to the left of the source line in the program listing. This feature can provide useful documentation of program changes. For example, assume that a program is assigned the identification code UPDATE-1 when it is first compiled. If the program must be recompiled at a later date to incorporate changes, the new or changed statements can be assigned a different identification code such as UPDATE-2.

The identification field is also the library-name for source statements that are to be placed in a copy-library. Therefore, all such statements must have an identification field. (For additional information about the copy feature, please refer to the COPY statement in Section VIII.)

CODING CONVENTIONS

The coding conventions discussed below are not requirements of the COBOL language nor of COBOL/3000. For the most part, the following suggestions simplify coding problems, improve the readability of the program, and help in debugging the program.

Using the Coding Form

The neater and more legibly a program is written, the less likely it is to require corrections because of keypunching errors.

Skipping lines on the coding form helps make the program more legible. Also, this provides room to enter a forgotten line.

Use Indentation

In the Data Division indentation improves the readability of the program and provides a graphic representation of data structures. The higher (numerically) the level number of an item, the more it should be indented, as in the following example:

SYNOPSIS		COBOL STATEMENT												
6	7	8	12	16	20	24	28	32	36	40	44	48		
	01													MAILING-LABEL.
			03											ADDRESS.
				05										STREET.
					07									NUMBER PIC 99999.
					07									STREET-NAME PIC X(20).
				05										CITY PIC X(20).
				05										STATE PIC X(3).
				05										ZIP PIC 9(5).
			03											NAME PIC X(20).

Code One Statement Per Line

The programmer should code only one statement per line. This technique makes it much easier to insert a statement if needed. Also, when more than one statement is coded on a line, statements that begin in the middle of the line tend to get lost in the text. This makes debugging the program much more difficult.

When a sentence requires more than one line of coding, the continuation line should be indented. If this convention is followed, the COBOL verb or condition which begins each procedural sentence stands out at the left of the coding to provide a convenient summary of the routine. This technique is especially useful for debugging.



EXAMPLES

The following are examples of coding technique. Both examples are valid COBOL procedures and perform the same function. Notice that the first example is difficult to read and understand (and equally difficult to debug).

		DATE	PROGRAM
7	A	COBOL STATEMENT	
7	B	12	16
7	B	20	24
7	B	28	32
7	B	36	40
7	B	44	48
7	B	52	56
7	B	60	64
		PAGE-BREAK-RTN.	
		IF LINE-CNT IS LESS THAN 50 GO TO UPDATE-RTN.	
		MOVE ZEROS TO LINE-CNT.	
		ADD 1 TO PAGE-CNT.	
		WRITE PRINT-LINE FROM HEAD-1 AFTER ADVANCING PAGE-TOP.	
		WRITE PRINT-LINE FROM HEAD-2 AFTER ADVANCING 2.	
		ADD 2 TO LINE-CNT.	
		MOVE SPACES TO PRINT-LINE.	
		GO TO UPDATE-RTN.	

1ER			DATE	PROGRAM															
E	C	A	COBOL STATEMENT																
R	J	B	6	7	8	12	16	20	24	28	32	36	40	44	48	52	56	60	64
*			PAGE-BREAK-RTN.																
			ADD 1 TO LINE-CNT.																
			IF LINE-CNT < 50																
			GO TO UPDATE-RTN.																
			ADD 1 TO PAGE-CNT.																
			WRITE PRINT-LINE FROM HEAD-1 AFTER PAGE-TOP.																
			WRITE PRINT-LINE FROM HEAD-2 AFTER 2.																
			MOVE 2 TO LINE-CNT.																
			MOVE SPACE TO PRINT-LINE.																
			GO TO UPDATE-RTN.																
*			NEXT-PARA.																

USING THE EDITOR

The EDIT/3000 program may be used to prepare a COBOL Source program. The Editor includes a special formatting option for COBOL. Refer to *Using the HP 3000: A Guide for the Terminal User* for instructions on using the Editor to prepare source programs.

PART 2
Program Divisions

SECTION III

Identification Division

The Identification Division, which must always be the first division of a COBOL program, supplies the program-name used to identify both the source program and the resultant object program. Also, this division provides optional paragraphs which allow the programmer to enter such information as the date when the program is written, security requirements for the program, and so on.

Only the PROGRAM-ID paragraph is required in the Identification Division; the remaining paragraphs are treated as comments. (Comments are printed on the program listing, but have no other effect on the compilation.)

GENERAL FORMAT

IDENTIFICATION DIVISION.
PROGRAM-ID. program-name.
AUTHOR. [comment-entry] . . .]
INSTALLATION. [comment-entry] . . .]
DATE-WRITTEN. [comment-entry] . . .]
DATE-COMPILED. [comment-entry] . . .]
SECURITY. [comment-entry] . . .]
REMARKS. [comment-entry] . . .]

GENERAL RULES

The IDENTIFICATION DIVISION heading must always be present. Division headings must start in field A and must be terminated with a period followed by a space.

The PROGRAM-ID paragraph must always be present as the first paragraph in the Division; this paragraph assigns a name to the program. Program-names must start with an alphabetic character and must conform to the rules for the formation of paragraph-names. However, this compiler deletes all hyphens from the program-name and then accepts only the first fifteen remaining characters. Thus the program-name END-OF-WEEK-ACTIVITY-RECAP is converted to ENDOFWEEKACTIVI. These fifteen characters must provide a unique name for the program.

The program-name may be a non-numeric literal; the name enclosed by the quotation marks must conform to the rules stated above.

RULES FOR COMMENT-ENTRY PARAGRAPHS

All paragraphs other than the PROGRAM-ID paragraph are optional and may be included in this division at the user's choice. If used, these optional paragraphs should be presented to the compiler in the sequence shown in the general format. All paragraph-names must begin in the A Area.

Comment-entries may contain any printable characters and must conform to the rules for sentence and paragraph structure. Comment-entries must be coded within the B Area of the coding form.

If the optional DATE-COMPILED paragraph is used, the compiler automatically prints the current date on the line following the paragraph heading. This compiler-supplied date is assumed to terminate the comment-entry for the DATE-COMPILED paragraph. If the programmer wishes to make his own comment-entry in this paragraph, the entry must be coded on the same line as the paragraph header; subsequent lines in this paragraph will not be listed when the date is inserted.

SECTION IV

Environment Division

The Environment Division, the second division of every COBOL program, consists of two sections: the Configuration Section and the Input-Output Section.

The Configuration Section describes the source computer and the object computer (the computer on which the program is to be compiled, and the computer on which the program is to be run, respectively). This compiler assumes that both the source and object computers are the HP 3000 and treats the associated paragraphs as comments. The Configuration Section also allows the programmer to assign his own names to certain system functions which are described later in this section.

The optional Input-Output Section names the files to be used in the program and defines the characteristics of the files.

The clauses of the Environment Division are individually explained on the following pages.

GENERAL FORMAT

```
ENVIRONMENT DIVISION.  
[ CONFIGURATION SECTION.  
  SOURCE-COMPUTER.  source-computer-entry.  
  OBJECT-COMPUTER. object-computer-entry.  
  [ SPECIAL-NAMES.  [mnemonic-name-entry] . . .  
                    [CURRENCY SIGN IS literal]  
                    [DECIMAL-POINT IS COMMA] . ]  
  [ INPUT-OUTPUT SECTION.  
    [FILE-CONTROL.  file-control-entry] . . .  
    [I-O-CONTROL.  input-output-control-entry] ] ]
```

CONFIGURATION SECTION

This compiler assumes that all programs are to be compiled and run on the HP 3000 Computer. Therefore, the source-computer-entry and the object-computer-entry are treated as comments. The Object Computer paragraph frequently includes the MEMORY SIZE and SEGMENT LIMIT clauses. Because the HP 3000 features virtual storage, neither clause is meaningful to the compiler. If present, these clauses are treated as comments.

Note: Because any entry in the Source or Object computer paragraphs is treated as a comment, a COPY statement coded in either of these paragraphs will have no effect on the compilation.

SPECIAL-NAMES Paragraph

The SPECIAL-NAMES paragraph is optional; when used, it must immediately follow the OBJECT-COMPUTER paragraph. This paragraph allows the programmer to assign mnemonic-names to certain system functions. In addition, the programmer may also specify a CURRENCY SIGN other than \$ and may exchange the functions of periods and commas in editing currency amounts.

SYSTEM-FUNCTIONS. The Special-Names entry format for system functions is as follows:

system-function *IS* mnemonic-name

The compiler recognizes five system functions. The Special-Names paragraph allows the programmer to assign a name of his choice (a mnemonic-name) to these functions. Each of the system-functions is explained in the following list:

- TOP — This function is associated with the ADVANCING clause of the WRITE statement. When included in the ADVANCING clause, the mnemonic-name assigned by the programmer to TOP causes the printer to skip to channel 1, which is normally associated with the top of a printer form.
- NO SPACE CONTROL — This function sets the Control parameter of the FWRITE (an MPE intrinsic generated by the COBOL compiler for a WRITE statement) to 0 rather than 1, thus altering the effect of WRITE statements for printer files. For additional information, refer to the WRITE statement in this manual and to the *MPE Intrinsic Reference Manual*.
- SYSIN — SYSIN is a COBOL word which refers to the MPE \$STDIN device (the user's terminal for a session, the card reader or operator's console for batch mode).
- SYSOUT — SYSOUT is a COBOL word which refers to the MPE \$STDLIST device (the user's terminal for a session, the line printer for batch mode).
- CONSOLE — CONSOLE is a COBOL word which refers to the operator's console.

If any of these functions are to be used in the program, they may be assigned mnemonic-names as shown in the following example:

SPECIAL-NAMES.

**TOP IS EJECT.
SYSIN IS CARD-READER.
NO SPACE CONTROL IS NOCCTL.**

In the example, EJECT and CARD-READER are mnemonic-names. EJECT and CARD-READER are then used in the appropriate procedural statements (e.g., WRITE PRINT-REC AFTER EJECT, ACCEPT CURRENT-PERCENTAGE FROM CARD-READER). The programmer may assign any mnemonic-name he wishes to these functions, with the exceptions of any data- or procedure-names used elsewhere in the program and reserved COBOL words. (Within the Special-Names paragraph, the system-function names themselves are reserved words. Thus the statement TOP IS TOP cannot be specified.)

CURRENCY SIGN CLAUSE. The literal in the CURRENCY SIGN clause is used in place of the \$ character in both PICTURE clauses and in printed reports. The literal must be a single character enclosed in quotation marks and must not be any character that can appear in a PICTURE clause character-string. Therefore, the following characters may not be specified:

A
B
C
D
P
R
S
V
X
Z
0 through 9
*
+
-
,
.
;
(
)
" or '
space

If the program is to print the character “M” to represent the Mark, then the M replaces the \$ as an editing symbol in all PICTURE clause character-strings, and the following clause must appear in the SPECIAL-NAMES paragraph:

CURRENCY SIGN IS “M”

Decimal-Point Clause

The DECIMAL-POINT IS COMMA clause causes the compiler to interchange the functions of commas and periods in both PICTURE clauses and in numeric literals. This convention is frequently observed in other countries, especially when dealing with monetary amounts. An item with the PICTURE S9,999.99 must be coded as S9.999,99 if the DECIMAL-POINT IS COMMA clause is specified in the Special-Names paragraph. Notice that this clause remains in effect throughout the entire program.

INPUT-OUTPUT SECTION

This section may be omitted only when the program does not read or write any file data. This can be done when only a limited amount of data is input or output through the use of the ACCEPT or DISPLAY statements, or when the program is used as a subprogram and derives all its data from another program.

The FILE-CONTROL paragraph heading must always be the first entry within the INPUT-OUTPUT SECTION as shown in the General Format for the Environment Division.

The file-name of every file used in the program must appear in its own SELECT statement. The file-name used in a SELECT statement is the same as that used in the FD or SD level entry for the file. Every SELECT statement must appear within the FILE-CONTROL paragraph of the INPUT-OUTPUT SECTION.

Because the SELECT statement is somewhat different for sequential and random files, each is discussed individually on the following pages. Sort-file considerations are discussed with sequential files.

SELECT (Sequential Files)

Every SELECT statement must appear within the FILE-CONTROL paragraph.

FORMAT

```
FILE-CONTROL. { SELECT [OPTIONAL] file-name
  ASSIGN TO [integer-1] "system-file-name-1" [, "system-file-name-2"] ...
  [ FOR MULTIPLE { REEL } ] [ RESERVE { integer-2 } ALTERNATE [ AREA ] ]
  [ , { FILE-LIMIT IS } { data-name-1 } THRU { data-name-2 }
    { FILE-LIMITS ARE } { literal-1 } ]
  [ , { data-name-3 } THRU { data-name-4 } ] ... ]
  [ , ACCESS MODE IS SEQUENTIAL ]
  [ , PROCESSING MODE IS SEQUENTIAL ] . } ...
```

Although this compiler accepts SELECT statements in the format shown above, this compiler neither requires nor supports certain features of the SELECT statement. When the clauses associated with these features appear in the SELECT statement, this compiler treats the clauses as comments. Clauses treated as comments are listed below:

- Integer-1, used to indicate the number of input-output units assigned to a particular file, is treated as a comment.
- All system-file-names except system-file-name-1 are treated as comments. (The compiler generates a warning message for this condition.) System-file-names must be enclosed in quotation marks.
- The MULTIPLE REEL/UNIT clause is treated as a comment.
- The FILE-LIMIT clause documents the beginning and end of logical segments of a mass storage file. Since this is a normal function of the MPE operating system, all of the FILE-LIMITS clause is treated as a comment.

SELECT (Sequential files)

FILE-NAME. Each file described in the Data Division must be named in one and only one SELECT statement. Conversely, each selected file must have an FD or SD level entry in the Data Division.

OPTIONAL CLAUSE. The OPTIONAL clause is permitted only for sequentially accessed input files which may not be present every time the object program is executed. A monthly or quarterly summary file is an example of such a file.

When a program containing an OPTIONAL file is run and the file is not present, the MPE :FILE command must be used to equate the file-name to \$NULL. This establishes the linkage required for MPE to logically "open" the file without causing the program to abort.

ASSIGN CLAUSE. The ASSIGN clause associates the file-name with a particular type of input-output device. If file-name represents a sort-file, this is the only clause which has meaning to the compiler.

Integer-1 and all system-file-names except system-file-name-1 are treated as comments by this compiler.

System-file-name must have the following format:

```
name[, [class] [, [recording mode] [, [device [(CCTL)]] [, [filesize] [, [formsmesssage] [, L]]]]]]
```

for example, "OUTFILE,UT,,TAPE(CCTL)" or "INVOICE,,,DISC,500,,L"

- *Name* must be an MPE formal file designator. These designators may contain up to eight alphabetic or numeric characters. The file designator must begin with an alphabetic character and must not contain any embedded spaces. The MPE designators \$STDIN and \$STDLIST may be used for standard input and standard list devices, respectively.
- *Class* must consist of the letters DA for a mass storage device, UT for a utility device such as a tape drive, or UR for a unit record device such as a card reader. UT and UR devices are logically equivalent since both are sequential devices. The compiler assumes DA when no entry is present.
- *Recording mode*; as an option, the programmer may specify whether *recording mode* is to be ASCII or binary. Enter the letter A to represent ASCII, the letter B to represent binary. The compiler assumes A when no entry is present.
- *Device* is an MPE specification that allows the programmer to assign the file to a particular type of file device. Since the device codes are subject to change, please refer to the MPE Commands Reference Manual for the specific codes. If *device* is followed by a left-parenthesis, CCTL is assumed regardless of which letters follow the left parenthesis. If only *device* is specified, the default setting is NOCCTL. (Refer to the *MPE Commands Reference Manual* for the meaning of CCTL and NOCCTL.)
- The optional *filesize* may be a 9-digit number which represents the number of records in a file. The compiler assumes a filesize of 10,000 records if no filesize is specified.
- *Formsmesssage* is a request for the computer operator to provide appropriate forms, such as blank checks or inventory report forms, on an output device. This parameter can contain up to 56 characters terminated by a period.
- "L" enables dynamic locking for a disc file. If anything other than L is specified, a warning message is displayed and locking is not enabled unless the \$CONTROL LOCKING command is specified.

Note: Only the name entry is required in the system-file-name. The other entries may be used if desired. When the system-file-name provides the system with adequate information, no control cards are required when the object program is executed. If a control card is entered for a particular file at program execution time, the control card overrides all entries made in the source program except for the file-name.

MULTIPLE REEL/UNIT CLAUSE. The compiler treats this clause as comments.

RESERVE CLAUSE. The compiler automatically reserves one buffer for each file. The optional RESERVE integer ALTERNATE AREAS clause allows the programmer to assign additional buffer areas to the file. Additional buffers usually improve input-output performance at the cost of more memory. Additional buffers should be assigned to the slowest devices first such as card readers or punches.

Notice that the buffers assigned in this clause are in addition to the one buffer automatically allocated by the compiler. Thus, if the clause RESERVE 2 ALTERNATE AREAS is specified, then three buffers are allocated to the file.

FILE-LIMITS CLAUSE. The compiler treats this clause as comments.

ACCESS MODE CLAUSE. For devices such as magnetic tape access is always sequential. For mass storage devices such as the disc, access may be either sequential or random. The ACCESS MODE IS SEQUENTIAL clause specifies that mass storage records are to be read or written sequentially. Thus the file is logically equivalent to a magnetic tape file.

PROCESSING MODE CLAUSE. This clause indicates that mass storage records are to be processed in the order in which they are accessed. The compiler treats this clause as comments.

SELECT Statement (Sort-Files)

Although this compiler will accept sort-file SELECT statements in the format shown below, only the ASSIGN clause has meaning to the compiler.

Format 1

```

FILE-CONTROL. { SELECT file-name
                ASSIGN TO [integer-1] system-file-name-1
                [, system-file-name-2] . . . . } . . .

```


SELECT (Sequential Files)

Format 2

FILE-CONTROL. { SELECT file-name ASSIGN TO system-file-name-3
[, system-file-name-4] . . . OR system-file-name-5
[, system-file-name-6] . . .
[FOR MULTIPLE { REEL }] . } . . .
[UNIT] . } . . .

Integer-1, all system-file-names other than system-file-name-1, the OR option, and the MULTIPLE REEL/UNIT option are all treated as comments.

File-name must be the subject of an SD level entry in the Data Division.

SELECT (Random Files)

Although this compiler will accept SELECT statements for randomly accessed files in the format shown below, only the ASSIGN, ACCESS MODE, and ACTUAL KEY clauses have meaning to the compiler.

```
FILE-CONTROL. { SELECT file-name  
  ASSIGN TO [integer-1] "system-file-name-1" [, "system-file-name-2"] ...  
  [ , { FILE-LIMIT IS } { data-name-1 } THRU { data-name-2 }  
    { FILE-LIMITS ARE } { literal-1 } { literal-2 }  
  [ , { data-name-3 } THRU { data-name-4 } ] ... ]  
  [ , ACCESS MODE IS RANDOM]  
  [ , PROCESSING MODE IS SEQUENTIAL]  
  [ , ACTUAL KEY IS data-name-5 ] . } ...
```

Integer-1, all system-file-names other than system-file-name-1, the FILE-LIMITS clause, and the PROCESSING MODE clauses are all treated as comments.

For an explanation of the ASSIGN clause, refer to the description of the SELECT statement for sequential files.

The ACCESS MODE IS RANDOM clause indicates that this file is stored on a randomly accessed mass storage device.

The ACTUAL KEY clause must name a field in either the File Section or the Working-Storage Section of the Data Division. The ACTUAL KEY must be a binary integer which corresponds to a relative record number. Therefore, the USAGE of the ACTUAL KEY must be COMPUTATIONAL SYNCHRONIZED. The ACTUAL KEY may contain from five to nine digits. Thus the PICTURE for ACTUAL KEY must be as follows:

S9(5) through S9(9) USAGE COMP SYNCHRONIZED

An ACTUAL KEY may also be named for a sequential output file which is being written to a mass storage file. In this case the software places the relative record address of a record in the ACTUAL KEY when the record is written. The programmer may capture this information and build it into a table. The data in this table may then be used to access the record randomly in a subsequent program.

For additional information about the ACTUAL KEY, refer to the READ and/or WRITE statements for random files.

I-O-CONTROL Paragraph

The I-O-CONTROL paragraph is optional. It may not be used when the FILE-CONTROL paragraph is not used. This paragraph allows the user to specify the sharing of a storage area by more than one file.

Although this compiler accepts I-O-CONTROL paragraphs in the following format, the RERUN clause and the MULTIPLE FILE clauses are treated as comments.

FORMAT

$$\begin{array}{l} \underline{\text{I-O-CONTROL.}} \left[\underline{\text{RERUN}} \left[\underline{\text{ON}} \left\{ \begin{array}{l} \text{file-name-1} \\ \text{system-file-name} \end{array} \right\} \right] \right. \\ \left. \left. \left. \left. \left. \left. \begin{array}{l} \text{EVERY} \left\{ \begin{array}{l} \left[\underline{\text{END OF}} \right] \left\{ \begin{array}{l} \underline{\text{REEL}} \\ \underline{\text{UNIT}} \end{array} \right\} \\ \text{integer-1} \underline{\text{RECORDS}} \\ \text{integer-2} \underline{\text{CLOCK-UNITS}} \\ \text{condition-name} \end{array} \right\} \right\} \text{OF file-name-2} \right\} \right\} \right\} \dots \\ \left[; \underline{\text{SAME}} \left\{ \begin{array}{l} \underline{\text{SORT}} \\ \underline{\text{RECORD}} \end{array} \right\} \right] \text{ AREA FOR file-name-3 } \{ , \text{file-name-4} \} \dots] \dots \\ \left[; \underline{\text{MULTIPLE FILE TAPE CONTAINS}} \text{ file-name-5 } \left[\underline{\text{POSITION}} \text{ integer-3} \right] \right. \\ \left. \left[, \text{file-name-6} \left[\underline{\text{POSITION}} \text{ integer-4} \right] \right] \dots \right] \dots \end{array}$$

This paragraph must always begin with the heading I-O-CONTROL.

RERUN Clause

The RERUN clause, which is treated as a comment by this compiler, specifies a file upon which rerun information is to be recorded and also specifies when this information is to be recorded.

SAME AREA Clause

The SAME AREA clause allows the user to conserve data storage by reassigning a data area to more than one file or more than one record.

As can be seen in the format, this clause may be coded as SAME AREA, SAME SORT AREA, or SAME RECORD AREA.

The SAME AREA and SAME SORT AREA clauses are logically equivalent; however, when the SAME SORT AREA format is used, at least one of the files named in the clause must be a sort-file.

When the compiler finds that multiple files are to share the same buffer area, it allocates memory space to the file that requires the greatest amount of storage. The buffers allocated to all the remaining files named in the SAME AREA clause are mapped within the same memory area. Because they share the same buffers, only one of the files may be open at a time.

A file-name that represents a sort-file must not appear in more than one SAME SORT AREA clause. However, sort-files and non-sort-files may share an area of memory if *all* the files that are to share the area are named in the SAME SORT AREA clause, and only the non-sort-files are named in a SAME AREA clause. In this case, the user must be certain that the SORT statement is not executed when any of the other files are open.

The SAME RECORD AREA clause specifies that two or more files are to use the same memory area for processing of the current logical record. All of the files may be open at the same time. A logical record in the SAME RECORD AREA is considered to be a logical record to each opened output file whose file-name appears in the SAME RECORD area clause and to the most recently read input file whose file-name appears in the SAME RECORD AREA clause. (Notice, however, that when the record is written, it is output only to the file whose FD level entry qualifies the record-name specified in the WRITE statement.)

If one or more file-names appearing in a SAME AREA clause appear in a SAME RECORD AREA clause, all the file-names in the SAME AREA clause must appear in that SAME RECORD AREA clause. However, additional file-names may appear in the SAME RECORD AREA clause. The rule that only one of the file-names mentioned in the SAME AREA clause may be open takes precedence over the rule that all files mentioned in the SAME RECORD AREA clause may be open at the same time.

EXAMPLES

In the following example, the two sort-files SORTFILE and WORKFILE share the same memory area at different times.

```
I-O-CONTROL.
    SAME SORT AREA FOR SORTFILE, WORKFILE.
```

In the following example, the sort-files SORTFILE and WORKFILE share the same memory area with the non-sort-files CARDIN, and CARDOUT.

```
I-O-CONTROL.
    SAME SORT AREA FOR SORTFILE, WORKFILE, CARDIN, CARDOUT
    SAME AREA FOR CARDIN, CARDOUT.
```

MULTIPLE FILE Clause

The compiler treats this clause as a comment.

SECTION V

Data Description Conventions

In the Data Division, the programmer describes for the compiler all the data that is to be manipulated in the program. The Data Division consists of three major sections as shown in the following skeletal format:

DATA DIVISION.

```
[FILE SECTION.                                ]  
[file description entry                          ]  
[record description entry] ... ] ...  
  
[WORKING-STORAGE SECTION.                    ]  
[data item description entry] ...                ]  
[record description entry] ...                    ]  
  
[LINKAGE SECTION.                            ]  
[data item description entry] ...                ]  
[record description entry] ...                    ]
```



Every COBOL program must include the DATA DIVISION heading. Each of the sections of the Data Division is optional and may be left out when the section is not needed. Whenever a given section is required, its heading must appear in the sequence indicated in the skeletal format. All division and section headings must be coded beginning in the A field of the coding sheet and must be terminated with a period followed by a space.

FILE SECTION

Although the File Section is technically optional, it will appear in nearly all COBOL programs since the File Section must be used to describe any data read from or written to a file device. (It is possible for a program to manipulate a small amount of data using only the ACCEPT and DISPLAY statements. In such a program it would be possible to omit the File Section.)

As indicated in the skeletal format, the File Section contains both file description entries and record description entries. Each file description entry provides information about the physical structure of the file, including such information as block size and record size. Each file description entry requires one or more record description entries. Record description entries allocate memory space for the record and associate data-names with individual fields of the record.

WORKING-STORAGE SECTION

The Working-Storage Section may contain data descriptions for individual data items such as counters and accumulators or for descriptions of records that are not directly a part of an external data file. Unrelated (noncontiguous) data items are assigned the special level number 77. Syntax rules for the COBOL language require that all 77-level entries be coded before any Working-Storage record description entries.

Each data description entry within Working-Storage allocates memory space for the data item and associates a data-name with the item. In addition, the programmer may assign values to data items in Working-Storage. Thus, Working-Storage is useful for defining report headings, tables with pre-defined values, counters with initial values, etc.

LINKAGE SECTION

The Linkage Section describes data made available from another program. Data item description entries and record description entries provide data-names and descriptions, but do not allocate any memory space since the data already exists in another program. Any data description clauses may be used in the Linkage Section, with a single exception: Because memory space is not allocated for data items in the Linkage Section, the VALUE clause may not be used except for level-88 condition-names.

CONCEPT OF LEVELS WITHIN DATA DEFINITIONS

In most business processing applications, data is organized into a hierarchical structure consisting of files, records, and fields. A file is a collection of related records. There are two types of records: physical records and logical records. In a card file, for example, the 80-column card is a physical record containing one logical record. For a magnetic tape file, a physical record is a block consisting of one or more logical records. A logical record consists of a number of related data items or fields. The operating system presents only logical records to the COBOL programmer.

General Structure of Data Description Entries

In the Data Division of a COBOL program, the hierarchy of data structures is represented by using a system of level indicators and level numbers. Each data description entry begins with either a level indicator or a level number. This is followed by the name of the item being described, which is followed by a sequence of clauses that describe the attributes of the item. The last data description clause is always terminated with a period followed by a space.

LEVEL INDICATOR. The level indicator FD (for File Description) specifies the beginning of a file description entry. If the file is a sort-file the level indicator SD must be used. Level indicators must be coded beginning in Area A. The clauses associated with file descriptions are fully explained in this section under "File Description Entries."

LEVEL NUMBERS. The numbers 01 (or simply 1) through 49 are used to indicate the structural hierarchy of records and fields. Since records are the most inclusive data structures, the level number 01 is used to indicate a record. Less inclusive data items are assigned numerically higher level numbers 02 through 49. These numbers need not be successive. Once a subdivision has been specified through the use of these level numbers, it may be further subdivided to provide more detailed data reference.

The most basic subdivisions of a record — those that have no further subdivision — are called *elementary items*; items that do have subdivisions are called *group items*. The following example (which is not a complete data description entry) illustrates the concept of group and elementary items:

```
FD PAYROLL-FILE
  LABEL RECORD IS OMITTED.
01 PERSONNEL-RECORD.
  03 EMPLOYEE-ID.
    05 EMPLOYEE-NUMBER PIC 9(5).
    05 SOCIAL-SECURITY-NUMBER PIC 9(9).
  03 ADDRESS.
    05 STREET PIC X(20).
    05 LOCATION.
      07 CITY PIC X(20).
      07 STATE PIC X(20).
    05 ZIP PIC 9(5).
```

Note: All elementary items have picture clauses.

In the example, all data items with a level number of 05 are elementary items with the exception of LOCATION. LOCATION is a group item containing the elementary items CITY and STATE. The group item ADDRESS contains the elementary items STREET and ZIP and the group item LOCATION. Notice that the level numbers used in the example are not successive. The compiler's interpretation of this example would not be altered if the level numbers 02, 03, and 04 had been used in place of 03, 05, and 07, respectively.

SPECIAL LEVEL NUMBERS. Three special level numbers are used to indicate entries that follow no true concept of levels:

- Level 66 entries specify elementary or group items introduced by a RENAME clause under "Data Description Clauses."
- Level 77 entries specify elementary noncontiguous Working-Storage items. These are items such as internal counters and accumulators that are not subdivisions of any other item and that are not themselves subdivided. If level 77 entries are required, they should be the first entries in the Working-Storage Section.
- Level 88 entries specify condition-names associated with particular values of a conditional variable. This level number is always associated with the VALUE clause and is fully discussed with the VALUE clause under "Data Description Clauses."

Rules for Coding Levels

The level indicators FD and SD and the level numbers 01 and 77 must be coded within columns 8 through 11 of the coding form. Subsequent data description entries may be coded in the same columns as the level indicators, or they may be progressively indented for each numerically higher level number as in the example for level numbers. Indentation is useful for documentation purposes and has no effect on the compiler.

A group includes all group and elementary items following it until a level number less than or equal to the level number of that group is encountered. All items which are immediately subordinate to a group item must have the same level number. The following examples illustrate this rule by showing a portion of the example of level numbers coded improperly:

IMPROPER CODING OF LEVEL NUMBERS	
EXAMPLE A	EXAMPLE B
05 LOCATION.	05 LOCATION.
06 CITY...	07 CITY...
07 STATE...	06 STATE...
05 ZIP...	05 ZIP...

In example A, the 07 level number for STATE tells the compiler that CITY is a group item, the first and only element of which is STATE. This coding is illogical (but not illegal if CITY has no picture) since LOCATION, CITY, and STATE all refer to the same item. In example B, the 06 level number for STATE tells the compiler that STATE is a group item. However, there are no elementary items associated with the group item STATE. This example violates the rules for level numbers and must not be used.

STANDARD DATA POSITIONING

Classes of Data

Within the COBOL language, all data is grouped into three classes: alphabetic, numeric, and alphanumeric. These classes are independent of either internal or external storage formats.

Alphabetic data may consist of the twenty-six letters of the alphabet and the space character. *Numeric* data may consist of the digits 0 (zero) through 9 and may also include an operational sign. Only numeric data is valid for use in arithmetic operations. *Alphanumeric* data may consist of any combination of HP 3000 data set characters. However, to maintain full compatibility with the COBOL language, only the characters of the COBOL data set should be used. Alphanumeric data falls into three categories:

- Alphanumeric (without editing)
- Alphanumeric edited
- Numeric edited

Every elementary item belongs to one of these classes or categories. Every group item is treated as an alphanumeric item regardless of the classes of the elementary items subordinate to the group.

Standard Positioning of Alphabetic and Alphanumeric Data

Alphabetic and alphanumeric data is aligned within the receiving field beginning at the leftmost character of the receiving field. If the receiving field is larger than the data being stored, unused positions to the right are filled with spaces. If the receiving field is smaller than the data being stored, data is transferred from left to right only until the receiving field has been filled; any remaining data in the sending field is truncated.

EXAMPLE

In the following example, the character Δ represents a space. The examples show the results of moving various length items into an eleven-character field.

DATA TO BE STORED	RECEIVING FIELD BEFORE TRANSFER	RECEIVING FIELD AFTER TRANSFER
ABC	PQRSTUVWXYZ	ABC $\Delta\Delta\Delta\Delta\Delta\Delta\Delta$
ABCDEFGHIJK	PQRSTUVWXYZ	ABCDEFGHIJK
ABCDEFGHIJKLMNO	PQRSTUVWXYZ	ABCEDFGHIJK

Standard Positioning of Numeric Data

Numeric data is always aligned by decimal point and is placed in the receiving field with zero fill or truncation on either end as required. However, the actual decimal point character is never present in any numeric field; any item containing a combination of digits and editing characters such as the decimal point, commas, etc. are classified as alphanumeric and are not valid for arithmetic operations.

If a decimal point is not specified for a numeric item, the compiler assumes that the item has a decimal-point immediately following its least significant digit and aligns the item as in the paragraph above.

If the receiving field is a numeric edited item, the data moved to the edited field is aligned by decimal position with zero fill or truncation at either end as required, except where editing conventions cause replacement of any leading zeros.

If the receiving field is alphanumeric (other than a numeric edited item), the sending data is aligned within the receiving field as though it were alphabetic with space fill or truncation at the right end of the field as required.

EXAMPLE

In the following examples, the position of the implied decimal point is represented by the caret (^). The character Δ represents a space. Notice that the compiler assumes a decimal point at the right-hand end of the field to be stored.

DATA TO BE STORED	RECEIVING FIELD BEFORE TRANSFER	RECEIVING FIELD AFTER TRANSFER	
		Format	Content
12 [^]	987654 [^]	9(3)V9(3)	001200 [^]
123456 [^]	987654 [^]	9(3)V9(3)	123456 [^]
1234567890 [^]	987654 [^]	9(3)V9(3)	345678 [^]
12 [^]	987654 [^]	9(3)V9(3)	012000 [^]
12 [^]	PQRSTU (alphanumeric item)	X6	12 $\Delta\Delta\Delta\Delta$
"1234567890"	PQRSTU (alphanumeric item)	X6	123456
1234567890	987654 [^]	9(3)V9(3)	890000 [^]

SECTION VI

Data Description Clauses

Data description clauses fall into two general groups. The first group pertains to file description and is always associated with FD or SD level indicators. The second group pertains to individual records or data items. In this section of the manual, clauses relating to file description are grouped together, followed by clauses relating to data description. Clauses are listed alphabetically within each group.

FILE DESCRIPTION ENTRIES

The file description entry consists of the level indicator FD (or SD for sort-file descriptions), followed by the file-name, followed by a series of clauses that describe the file. Because these clauses describe an external file structure, the particular clauses required will vary according to the file structure. The clauses may be coded in any order. The entry must be terminated with a period followed by a space.

Each file description entry must be followed by one or more 01 level record description entries. When more than one 01 record description is present following an FD statement, each subsequent record description implies a redefinition of the same area of storage; the compiler allocates only enough storage to accommodate the longest record description. Because multiple record descriptions following a single level indicator implicitly redefine the same area, the REDEFINES clause is not required and must not be used.

All punctuation shown in the following general format is optional except the period at the end of the file description entry.

GENERAL FORMAT

Format 1

DATA DIVISION.

FILE SECTION.

FD file-name

$$\left[; \underline{\text{BLOCK CONTAINS}} \text{ [integer-1 TO] integer-2 } \left\{ \begin{array}{l} \underline{\text{RECORDS}} \\ \underline{\text{CHARACTERS}} \end{array} \right\} \right]$$
$$\left[; \underline{\text{DATA}} \left\{ \begin{array}{l} \underline{\text{RECORD IS}} \\ \underline{\text{RECORDS ARE}} \end{array} \right\} \text{data-name-1 [, data-name-2] . . .} \right]$$
$$\left[; \underline{\text{RECORDING MODE IS}} \left\{ \begin{array}{l} \text{F} \\ \text{V} \\ \text{U} \end{array} \right\} \right]$$
$$; \underline{\text{LABEL}} \left\{ \begin{array}{l} \underline{\text{RECORD IS}} \\ \underline{\text{RECORDS ARE}} \end{array} \right\} \left\{ \begin{array}{l} \underline{\text{STANDARD}} \\ \underline{\text{OMITTED}} \end{array} \right\} \text{data-name-3 [, data-name-4] . . .} \left. \right\}$$
$$[; \underline{\text{RECORD CONTAINS}} \text{ [integer-3 TO] integer-4 CHARACTERS}]$$
$$\left[; \underline{\text{VALUE OF}} \text{ data-name-5 IS } \left\{ \begin{array}{l} \text{data-name-6} \\ \text{literal-1} \end{array} \right\} \right]$$
$$\left[; \text{data-name-7 IS } \left\{ \begin{array}{l} \text{data-name-8} \\ \text{literal-2} \end{array} \right\} \right] \dots \left. \right]$$

Format 2

SD sort-file-name

$$\left[; \underline{\text{DATA}} \left\{ \begin{array}{l} \underline{\text{RECORD IS}} \\ \underline{\text{RECORDS ARE}} \end{array} \right\} \text{data-name-1 [, data-name-2] . . .} \right]$$
$$[; \underline{\text{RECORD CONTAINS}} \text{ [integer-1 TO] integer-2 CHARACTERS}].$$

Format 1 is the complete skeletal entry for a typical file. Format 2 is the complete skeletal entry for a sort-file. For a sort-file, label procedures and the rules for blocking and internal storage are peculiar to the SORT statement. The remaining clauses in the sort-file description follow the same rules as the clauses for an FD statement.

BLOCK CONTAINS CLAUSE

The **BLOCK CONTAINS** clause specifies the size of a physical record as stored on some external file device. This clause may not be used with the SD level indicator.

FORMAT

$$\underline{\text{BLOCK CONTAINS}} \text{ [integer-1 TO] integer-2 } \left\{ \begin{array}{l} \underline{\text{RECORDS}} \\ \underline{\text{CHARACTERS}} \end{array} \right\}$$

Integer-2 (and integer-1, if specified) must be a positive integer. If only integer-2 is given, it represents the exact size of the physical record. If both integer-1 and integer-2 are given, they refer to the minimum and maximum size of the physical record, respectively. This second option indicates a sequential file containing variable length records.

This clause is always required except when one physical record contains exactly one complete logical record. An example of this situation is an 80-character logical record read from a card file.

For a magnetic tape file, the block size may be stated in terms of records. However, for a disc file, the block size must be stated in terms of characters unless the block is the exact size (256 characters) of a disc sector or some multiple thereof. If the clause **BLOCK CONTAINS 0 (zero)** is used for a disc file, the compiler assumes the MPE default option.

When the **CHARACTERS** option is specified, the user must calculate the exact number of bytes — including any slack bytes (slack bytes are explained under the **USAGE** clause) — then multiply by the number of records per block. Thus, if the user specifies a block containing five records of 100 bytes each, the clause would be coded **BLOCK CONTAINS 500 CHARACTERS**

If logical records of differing size are grouped into a block, the compiler assigns the maximum record size to each logical record. Thus, a block containing one 100-character record and four 60-character records is equivalent to a block containing five 100-character records.

DATA RECORDS CLAUSE

The **DATA RECORDS** clause documents the names of the data records contained in the file associated with this FD or SD statement. This clause serves only as documentation and has no effect on the compiler.

FORMAT

$$\underline{\text{DATA}} \left\{ \begin{array}{l} \underline{\text{RECORD IS}} \\ \underline{\text{RECORDS ARE}} \end{array} \right\} \text{data-name-1 [,data-name-2] . . .}$$

Data-name-1, data-name-2, and so on must be the name of data records contained in this file and must have 01 level numbers.

The presence of more than one data name indicates that this file contains more than one type of record. These records need not have the same description. The order in which the records are listed is not significant to the compiler.

If the record descriptions have different lengths, the compiler allocates memory space for the longest possible record. All records within this file share the same area.

LABEL RECORDS CLAUSE

The LABEL RECORDS clause specifies whether disc file labels are present and identifies the label, if any. This clause is required in every file description entry.

FORMAT

$$\text{LABEL} \left\{ \begin{array}{l} \text{RECORD IS} \\ \text{RECORDS ARE} \end{array} \right\} \left\{ \begin{array}{l} \text{STANDARD} \\ \text{OMITTED} \\ \text{data-name-1 [, data-name-2] . . .} \end{array} \right\}$$

The OMITTED option specifies that no explicit labels exist for the file or for the device to which the file is assigned. This option must be used for all non-disc devices.

The STANDARD option specifies that labels exist for a disc file and that the labels conform to MPE specifications.

The LABEL RECORDS ARE data-name-1, data-name-2, etc., option specifies that the file has user labels as well as a standard label. Data-name-1, data-name-2, and so on must be the data-names of 01 level record descriptions associated with this FD statement. Data-names listed in the LABEL RECORDS clause must not appear in the DATA RECORDS clause. Also, any data-name that appears in the LABEL RECORDS clause must be referred to only in a USE procedure within the Procedure Division. Label records for a given file share the same area of memory.

Up to eight user labels may be specified for each disc file; no user label can exceed 80 characters.

Note: The LABEL RECORDS clause does not apply to magnetic tape labels. Tape file FD statements must always contain LABEL RECORD OMITTED.

RECORD CONTAINS CLAUSE

The RECORD CONTAINS clause documents the size of the data records associated with this FD or SD statement.

FORMAT

RECORD CONTAINS [integer-1 TO] integer-2 CHARACTERS

Integer-2 (and integer-1, if specified) must be a positive integer. The size of each data record is fully defined within the record description entry; therefore, the RECORD CONTAINS clause is never required, and is only used as information for the compiler in the case of variable-length records. When the clause is specified, the following rules apply:

- If both integer-1 and integer-2 are shown, they refer to the number of characters in the smallest data record and the largest data record, respectively.
- Integer-2 should not appear by itself unless all data records in the file are the same size. In this case, integer-2 represents the exact number of characters in the data record.
- The size of the data record must be specified in standard data format (in terms of the number of bytes occupied internally by its characters, regardless of the number of characters used to represent the items within the record). The number of bytes occupied internally by a data item is determined as noted in the discussion of the USAGE clause later in this section. The size of a record is determined according to the rules for determining the size of a group item.

Whether the RECORD CONTAINS clause is specified or omitted, the compiler determines the record lengths from the record descriptions furnished. When a data item description entry in a record contains an OCCURS DEPENDING ON clause, the compiler uses the maximum value of the variable to calculate the record length.

(Record description entries and data item description entries are discussed under DATA DESCRIPTION - - GENERAL INFORMATION, later in this section.)

RECORDING MODE CLAUSE

The RECORDING MODE CLAUSE specifies the format of the logical records in the file.

FORMAT

RECORDING MODE IS $\left\{ \begin{array}{c} F \\ V \\ U \end{array} \right\}$

F (fixed-length) format is specified when all logical records in the file are the same length. (This implies that no OCCURS DEPENDING ON clause is associated with an entry in any record description for the file.) If more than one record description entry is given following the FD entry, all record lengths calculated from the record descriptions must be equal.

V (variable-length) format may be specified for any combination of record descriptions (when the file contains records of different lengths). A variable-length logical record is always preceded by a control field containing the length of the logical record. Blocks of variable-length records include a block-length control field. The block is terminated by an end of block indicator (-1).

U (undefined) format may also be used to denote any combination of record descriptions. It is comparable to the V format, except that U-format records are not blocked and include no preceding control field.

When present, the RECORDING MODE clause is treated as a comment by the compiler, and serves only as documentation.

VALUE OF CLAUSE

The VALUE OF clause particularizes the description of an item in the label records associated with this file description entry.

FORMAT

$$\underline{\text{VALUE OF}} \text{ data-name-1 IS } \left\{ \begin{array}{l} \text{data-name-2} \\ \text{literal-1} \end{array} \right\} \left[, \text{data-name-3 IS } \left\{ \begin{array}{l} \text{data-name-4} \\ \text{literal-2} \end{array} \right\} \right] \dots$$

Data-name-1, data-name-3, etc., must be label records named in the LABEL RECORDS clause. Data-name-2, data-name-4, etc., must be in Working-Storage.

The VALUE OF clause is treated as a comment by this compiler, and serves only as documentation.

DATA DESCRIPTION — GENERAL INFORMATION

Each data description entry consists of a level number, followed by a programmer-assigned data-name or the key word FILLER, followed by a series of clauses which describe the attributes of the data. All punctuation within the data description entry is optional; however, the entry must be terminated with a period followed by a space. The clauses within the entry may be written in any order with one exception: the REDEFINES clause, if used, must immediately follow the data-name.

There are two types of data description entries: the data *item* description, and the *record* description entry. The data item description, which always has the level number 77, defines an independent, noncontiguous item that has no structural relationship to any other item. Data item descriptions are valid in the Working-Storage and Linkage Sections of the Data Division.. Record description entries describe any data which is part of a larger group or which is itself subdivided. Each record description either has or is subordinate to an 01 level entry. Because the record description entry describes any data that is subdivided, it is not necessarily related to a logical record. Record description entries are valid in the File, Working-Storage, and Linkage Sections of the Data Division.

In general, data description entries do not themselves generate object code. Therefore, any extra coding -- especially redefinition -- in the Data Division that makes it possible for the programmer to delete statements from the Procedure Division usually helps make the object program more efficient.

GENERAL FORMATS

Format 1

level number { data-name }
 { FILLER }

[REDEFINES Clause]

[BLANK WHEN ZERO Clause]

[JUSTIFIED Clause]

[OCCURS Clause]

[PICTURE Clause]

[SYNCHRONIZED Clause]

[USAGE Clause]

[VALUE Clause]



Format 2

66 data-name-1 RENAMES Clause.

Format 3

88 condition-name VALUE Clause.

Syntax Rules

The clauses in a data description entry may be separated by either commas or semicolons. All internal punctuation is optional; however, the entry must be terminated with a period followed by a space.

Level-numbers in Format 1 may be any number from 01 through 49 for a record description entry. The level-number must be 77 for a data item description.

The PICTURE clause must be specified for every elementary item except an index data item, for which this clause is prohibited.

The clauses BLANK WHEN ZERO, JUSTIFIED, PICTURE, and SYNCHRONIZED must appear only at the elementary item level. Furthermore, certain clauses such as BLANK WHEN ZERO and VALUE are illogical when used to describe data read from input files. In the case of the VALUE clause, for example, any value coded for an input field is immediately replaced when the first record is read.

Format 2 is fully explained under the RENAMES clause.

Format 3 indicates a condition-name. Each condition-name requires a separate entry with level-number 88. Format 3 is fully explained under the VALUE clause.

DATA-NAME OR FILLER CLAUSE

A data-name specifies the name of the data being described. The key word **FILLER** specifies an elementary item that is never directly referred to in the Procedure Division and therefore needs no explicit name.

FORMAT

$$\text{level-number } \left\{ \begin{array}{l} \text{data-name} \\ \underline{\text{FILLER}} \end{array} \right\}$$

Level-number may be any number from 01 through 49 for record description entries, or 77 for data item entries. Special level numbers 66 and 88 must be used as described under the **RENAMES** clause or the **VALUE** clause, respectively. Level numbers 01 and 77 must be coded in the A field of the coding sheet. All other level-numbers may be indented as desired.

A data-name or the key word **FILLER** must be the first word following the level number in every data description entry.

Under no circumstances can a **FILLER** item be referred to directly. If the **FILLER** item falls under an **OCCURS** clause, the item may be referenced through the use of indexing or subscripting. In **MOVE**, **ADD**, and **SUBTRACT** statements with the **CORRESPONDING** option, **FILLER** items are ignored.

Level 77 and level-01 entries in the Working-Storage and Linkage Sections must be given unique data-names as they cannot be qualified. Subordinate data-names need not be unique if they can be adequately qualified.

This compiler automatically assigns 01 and 77-level items starting on a word boundary. Word boundaries are fully explained under the Synchronized Clause in this section.

BLANK WHEN ZERO CLAUSE

The **BLANK WHEN ZERO** clause permits the blanking of an item when its value is zero.

FORMAT

BLANK WHEN ZERO

The **BLANK WHEN ZERO** clause can be specified only for elementary items whose **PICTURE** is specified as numeric or numeric edited. However, the compiler assumes any numeric item defined with the **BLANK WHEN ZERO** clause to be a numeric edited item. Such an item is not valid in an arithmetic statement (except as the object of the **GIVING** option or a **COMPUTE** statement) since it is not possible to add blanks to any numeric field.

The **BLANK WHEN ZERO** clause cannot be used with variable-length items.

The **BLANK WHEN ZERO** clause cannot be used for level-66 or level-88 items.

The **BLANK WHEN ZERO** clause causes generation of object code within the Procedure Division. Before data is placed in a field defined with this clause (by a **MOVE** statement, **COMPUTE** statement, or as the object of a **GIVING** option in an arithmetic statement), a comparison is made to test the data for zeros. If the data contains any non-zero character, the data is stored according to the conventions specified for the receiving field; if the field contains all zeros, they are replaced with blanks with one exception: when asterisks are specified as the replacement character for leading zeros (as is commonly done when printing checks), the editing specified for this field overrides the **BLANK WHEN ZERO** clause.

EXAMPLE

The following examples show the effect of the BLANK WHEN ZERO clause on a field called PRINT-AMOUNT. The caret (^) shown in the field VALUE indicates the position of the implied decimal point.

VALUE	DESCRIPTION OF PRINT-AMOUNT	RESULT
001234 ^	ZZZZ.99	12.34
001234 ^	ZZZZ.99 BLANK WHEN ZERO	12.34
000000 ^	ZZZZ.99	.00
000000 ^	ZZZZ.99 BLANK WHEN ZERO	△△△△△△
001234 ^	\$\$\$\$.99	\$12.34
001234 ^	\$\$\$\$.99 BLANK WHEN ZERO	\$12.34
000000 ^	\$\$\$\$.99	\$.00
000000 ^	\$\$\$\$.99 BLANK WHEN ZERO	△△△△△△
001234 ^	****.99	**12.34
001234 ^	****.99 BLANK WHEN ZERO	**12.34
000000 ^	****.99	****.00
000000 ^	****.99 BLANK WHEN ZERO	****.**

Notice that in the last example the asterisk editing characters override the BLANK WHEN ZERO clause.

JUSTIFIED CLAUSE

The JUSTIFIED clause specifies non-standard positioning of alphabetic or alphanumeric data within a receiving data item.

FORMAT

$$\left\{ \begin{array}{l} \underline{\text{JUSTIFIED}} \\ \underline{\text{JUST}} \end{array} \right\} \text{ RIGHT}$$

The JUSTIFIED clause can be specified only at the elementary item level.

JUST is an abbreviation for JUSTIFIED.

The JUSTIFIED clause cannot be specified for numeric or numeric edited items.

The JUSTIFIED clause must not be used for level-66 or level-88 items.

The rules for standard data positioning are explained under "Standard Data Positioning" in Section V.

EXAMPLE

In the following example, assume that PRINT-MONTH is an eight-character alphabetic field used in printing a date such as MAY 17, 1973. The following chart shows the result of the COBOL statement IF MONTH EQUALS 5 MOVE "MAY" TO PRINT-MONTH when PRINT-MONTH is defined with and without the JUSTIFIED clause:

USING JUSTIFIED	WITHOUT JUSTIFIED
△△△△MAY 17, 1973	MAY△△△△17, 1973

The following example, which uses the eight-character field PRINT-MONTH as in the preceding example, illustrates the effects of truncation with standard data positioning and with the JUSTIFIED clause. The COBOL statement IF MONTH EQUALS 9 MOVE 'SEPTEMBER' TO PRINT-MONTH has the following effects:

USING JUSTIFIED	WITHOUT JUSTIFIED
EPTEMBER 17, 1973	SEPTEMBE 17, 1973

OCCURS CLAUSE

The OCCURS clause eliminates the need for separate entries for lists or tables of repeated items by indicating the number of times a series of items with identical formats is repeated. The OCCURS clause also supplies the compiler with the information required for the use of indexes or subscripts.

Each of the options associated with the OCCURS clause (INDEXED, KEY, etc.), is individually discussed following the OCCURS clause itself.

FORMATS

Format 1

OCCURS integer-2 TIMES

$$\left[\left\{ \begin{array}{l} \underline{\text{ASCENDING}} \\ \underline{\text{DESCENDING}} \end{array} \right\} \text{ KEY IS data-name-2 } [, \text{ data-name-3}] \dots \right] \dots$$

$$[\underline{\text{INDEXED}} \text{ BY index-name-1 } [, \text{ index-name-2}] \dots]$$

Format 2

OCCURS integer-1 TO integer-2 TIMES [DEPENDING ON data-name-1]

$$\left[\left\{ \begin{array}{l} \underline{\text{ASCENDING}} \\ \underline{\text{DESCENDING}} \end{array} \right\} \text{ KEY IS data-name-2 } [, \text{ data-name-3}] \dots \right] \dots$$

$$[\underline{\text{INDEXED}} \text{ BY index-name-1 } [, \text{ index-name-2}] \dots]$$

Rules for OCCURS Clause

The OCCURS clause must not be used in the data description of an item having level-number 77 or 01.

Any other data description clauses associated with an entry that includes an OCCURS clause apply to each occurrence of the item. However, the VALUE clause (except for condition-name entries) may not be stated in an entry which includes an OCCURS clause or which is subordinate to an OCCURS clause.

Format-1 indicates a table with a fixed number of occurrences within it. Format-2 indicates a table which has a variable number of occurrences within it.

The table handling feature of the COBOL language allows the use of three levels of indexing or subscripting. Thus three nested levels of OCCURS clauses are permitted. (That is, an item containing an OCCURS clause may include a subordinate item containing an OCCURS clause, which in turn may also include an OCCURS clause.) However, the OCCURS clause may not be specified in a data description entry that describes an item whose size is variable. Because Format-2 indicates a table with a variable number of occurrences, an OCCURS clause with Format-2 can never be subordinate to any other OCCURS clause.

When OCCURS clauses are nested, the table is said to have more than one dimension. A two-dimensional table is defined when an item described with the OCCURS clause has a subordinate item also described with the OCCURS clause. A three dimensional table has yet another subordinate item described with the OCCURS clause.

Integer-1 and integer-2 must always be positive integers. Integer-1, if used, indicates the minimum number of occurrences in the table and may be equal to zero. Integer-2, which indicates either the maximum number or the exact number of occurrences in the table, must always be greater than zero and greater than integer-1. Notice that when both integer-1 and integer-2 are used in the OCCURS clause, the *number* of occurrences within the table may vary. The *size* of each occurrence is not affected.

The subject of an OCCURS clause (the data-item whose description includes the OCCURS clause) must always be indexed or subscripted in all procedural statements other than SEARCH. If the subject of the OCCURS clause is subordinate to other OCCURS clauses, one index or subscript must be mentioned for each OCCURS clause in the heirarchy. A maximum of three indexes or subscripts is allowed.

The maximum size for any table is 65,536 bytes.

DEPENDING ON Option

The DEPENDING ON option is required only when the last occurrence of the subject cannot otherwise be determined. Thus the DEPENDING ON option applies only to a table containing a variable number of occurrences as indicated by Format-2. The item described by the OCCURS . . . DEPENDING may not be subordinate to another OCCURS clause.

FORMAT

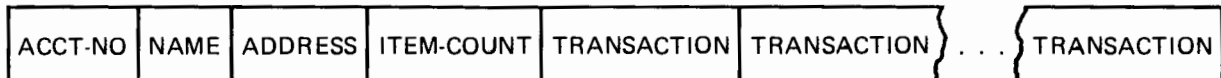
OCCURS integer-1 TO integer-2 TIMES [DEPENDING ON data-name-1] . . .

Data-name-1 must be a positive numeric field which contains a number not less than integer-1 and not greater than integer-2.

If this OCCURS clause and data-name-1 both appear in the data description for a logical record, data-name-1 must precede the OCCURS entry in both the physical record and in the data description. Only one item within a record may be described with the OCCURS . . . DEPENDING, and this must be the last item in the record.

EXAMPLE

The DEPENDING option is required only when the last occurrence of a table cannot otherwise be determined. Assume, for example, that each logical record in a file contains a table which may contain from zero to ten occurrences. A billing application provides a typical example of such a record: the record contains a fixed portion with the customer's name and address. This fixed portion of the record is followed by a table where each occurrence represents a transaction for that account. Data-name-1 in such a record must physically precede the table portion of the record. The following is a graphic illustration of such a record:



The data description entry for this type of table would be as follows:

TRANSACTION OCCURS 0 TO 10 TIMES DEPENDING ON ITEM-COUNT.

KEY Option

The KEY option identifies the data-names of the major and minor keys in a table and indicates whether the keys are arranged in ascending or descending order.

FORMAT

$$\left[\left\{ \begin{array}{l} \underline{\text{ASCENDING}} \\ \underline{\text{DESCENDING}} \end{array} \right\} \text{ KEY IS data-name-2 } [, \text{ data-name-3}] \dots \right]$$

The KEY option is required whenever the SEARCH ALL statement is used to access the table associated with this OCCURS clause. The option may be used as documentation whether or not the SEARCH ALL statement is used to access the table.

A maximum of twelve keys may be named for a given table.

Data-name-2 may be the name of the item that contains this OCCURS clause. If so, each occurrence within the table is a group item used as a key.

If data-name-2 is not the subject of this OCCURS clause, all of the items mentioned as keys must be within the group item which is the subject of this OCCURS clause. However, the key must not be associated with any other OCCURS clause, even though the second OCCURS clause might be subordinate to the subject of this entry. For example, an item in the second level of a two-dimensional table may not be named as a key for the first level of the table.

EXAMPLES

The following data description coding is from the description of a table of branch office code numbers. Since the table contains only code numbers, the subject of the OCCURS clause and the key for the table are the same data-name.

```
05  BRANCH-CODE OCCURS 50 TIMES ASCENDING KEY IS  
    BRANCH-CODE INDEXED BY BRANCH-INDEX PICTURE XXXX.
```

The following coding is from an inventory table description. Although each table occurrence contains the items PART-NUMBER, COST, BIN-NUMBER, QUANTITY-ON-HAND, and LOW-LIMIT, only the item PART-NUMBER is used as a key.

```
01 INVENTORY-TABLE.  
    03 STOCK-ITEMS OCCURS 150 TIMES ASCENDING KEY IS  
        PART-NUMBER INDEXED BY STOCK-INDEX.  
            05 BIN-NUMBER PICTURE 999.  
            05 PART-NUMBER PICTURE 9999.  
            05 QUANTITY-ON-HAND PICTURE 9999.  
            05 LOW-LIMIT PICTURE 9999.  
            05 COST PICTURE 999.99.
```

In the example above, notice that the key item PART-NUMBER is not the first item in the table occurrence. Keys may be located anywhere within the table element.

INDEXED BY Clause

The INDEXED BY option is required if any item within this table is to be accessed by indexing. Tables may be accessed via indexing or subscripting. In general, indexing provides greater efficiency than subscripting.

FORMAT

INDEXED BY index-name

Each index-name must follow the COBOL rules for the formation of names.

Each INDEXED BY clause must be associated with an OCCURS clause. For each INDEXED BY clause, the compiler builds a table index. The index is a unique entity that exists only within the system. The programmer never defines the index (except for naming it in the INDEXED BY clause); also, the index does not exist as a data item and cannot be associated with any data hierarchy.

For each index-name, this compiler generates a one-word (two-byte) field containing a positive binary value. This value is the displacement (expressed in actual number of characters) of a table occurrence from the beginning of the table. Displacement is calculated by multiplying the occurrence number minus one by the length (in bytes) of the table occurrence.

An index-data-item is defined by the USAGE IS INDEX clause. Although index-data-items may be used to store the contents of an index-name, they may not be used as an index.

The programmer may assign more than one index-name within the OCCURS clause. However, if the SEARCH ALL statement is used with the table, the compiler automatically uses the first assigned index-name for the search. In the SEARCH statement, the programmer may name the desired index in the VARYING clause.

PICTURE CLAUSE

The PICTURE clause specifies the size, general characteristics, and editing requirements of an elementary item. Notice that the PICTURE clause describes an item, but does not specify the contents of that item.

FORMAT

$$\left. \begin{array}{l} \text{PICTURE} \\ \text{PIC} \end{array} \right\} \text{ IS character-string}$$

General Rules

The PICTURE clause must be used in the description of every elementary data item except an index-data-item. The PICTURE clause is not allowed for any group item or for any index-data-item.

The character-string in a PICTURE clause may contain up to 30 symbols. With this compiler, it is theoretically possible (but of limited practical value) to represent an elementary field of up to 65,535 characters in the character string. The symbols allowed in the character string are fully explained on the following pages.

The PICTURE clause determines a data-item's category. (Categories of data are explained under "Standard Data Positioning," in Section V.)

- **Alphabetic Data**

The character-string for an alphabetic data item can only specify the symbol "A". The contents of an alphabetic data item are restricted to the characters of the alphabet and the space character, for example:

$$\left. \begin{array}{l} \text{PICTURE IS AAAAAAAAA.} \\ \text{PICTURE IS A(8).} \end{array} \right\} \text{ (these are equivalent)}$$

- **Numeric Data**

The character-string for a numeric data item can only specify the symbols 9, P, S, and V. The contents of a numeric data item are restricted to the numerals 0 through 9 and an operational sign, if any.

The number of 9's in the PICTURE clause specifies the number of digits in the field rather than the number of characters of storage required for the field. The storage format assigned to a numeric field by the USAGE clause may require a different number of storage positions than are indicated by the PICTURE clause. The maximum size permitted for a numeric item is 18 digits, for example:

PICTURE IS S999V99

PICTURE IS SPP99

- **Alphanumeric Data**

The character-string for an alphanumeric data item can specify only the symbols A, X, and 9, but must contain at least one X. The contents of an alphanumeric field may be any combination of HP 3000 characters.

The A's and 9's in the character-string for an alphanumeric item merely indicate the types of characters that would normally be expected to occupy the corresponding positions in the item. The object program performs no checks on the types of data in an alphanumeric item. Thus, the object code generated for the following two examples would be identical, for example:

PICTURE IS AA99XX

PICTURE IS XXXXXX

- **Alphanumeric Edited Data**

The character-string for an alphanumeric edited item can only specify the following symbols:

A X 9 B 0

Since the B (blank), and 0 (zero) are editing characters, an alphanumeric field is an edited item only if its PICTURE clause contains at least one of these editing characters and at least one A or X.

The contents of an alphanumeric edited item may be any combination of HP 3000 characters. The A's and 9's in the character-string merely indicate the types of characters that would normally be expected to occupy the corresponding positions in the item, for example:

PICTURE IS XXX99BBAAA0

- **Numeric Edited Data**

The character-string for an edited numeric item can only specify the following symbols:

B P V Z 0 9 , . * + - CR DB \$

Certain of these symbols are mutually exclusive, and certain symbols must appear in a particular order. Each of the symbols is individually explained on the following pages.

The maximum number of digits allowed in the PICTURE clause for an edited numeric item is 18. However, the entire edited field may be longer than 18 characters because of the insertion of editing characters.

PICTURE Clause Character-String

The PICTURE clause usually requires the repeated and consecutive use of certain symbols. For example, to describe a four-character alphanumeric field, the programmer uses the clause PICTURE IS XXXX. However, COBOL permits the use of integers enclosed in parentheses to indicate the number of repetitive occurrences of these symbols. Thus, the clause PICTURE IS X (4) has the same meaning to the compiler as PICTURE IS XXXX. Notice that the integer used to indicate repetition must be enclosed in parentheses and must immediately follow the repeated character.

The symbols S, V, . (period or decimal point), CR, and DB may each occur only one time in a PICTURE clause. Furthermore, V and . (period) are mutually exclusive in the character-string. V indicates an implied decimal point, and . (period) indicates an actual decimal point; it is not possible for a numeric field to have two significant decimal points. (Each of the character-string symbols is fully explained on the following pages.) The following symbols may be repeated in the character-string:

A X 9 P Z B 0 * , + - \$

EXAMPLE

PICTURE IS 999999	equals	PICTURE IS 9(6)
PICTURE IS AAAAA99999	equals	PICTURE IS A(5)9(5)

Character-String Symbols

A

Each A in the character-string represents a character position which can contain only a letter of the alphabet or the space character.

B

Each B in the character-string represents a character position into which the space character will be inserted. For example, if the six-digit date 011072 is moved into a field with the PICTURE 99B99B99, the result will be 01△10△72.

P

The symbol P indicates an assumed decimal scaling position and is used to specify the location of an assumed decimal point when the point is not within the number that appears in the data item.

The P in a character-string does not occupy a position in memory and does not add to the size of the item. However, the P has the effect of the digit zero whenever the item is accessed. When the item is used for arithmetic operations, the P must be counted as a digit to determine whether the field exceeds the 18-digit maximum size for a numeric field.

The symbol P can only be placed at either the left-hand end or the right-hand end of the character-string. Since the symbol P itself implies a decimal position, the use of the V symbol and the P symbol in the same character-string is redundant.

If a field in memory contains the digits 25, and the PICTURE for the field is PPP99, the field has an implied value of ^00025. If the same field has the PICTURE 99PPP, the field has an implied value of 25000^. In both cases, however, only the digits 25 are actually stored in memory.

S

The symbol S in a character-string indicates that the value of the data item may be positive or negative. Therefore, the symbol S is used only with numeric or numeric edited data. If used, the symbol S must be the left-most character in the character-string.

The symbol S is not counted in the length of the elementary field. Although the S indicates the presence of a sign in the data, it implies nothing about the format or the location of the sign in storage.

V

The symbol V is used in the character-string to indicate the location of the assumed decimal point and may appear only one time in the character-string. The V does not represent a character position and therefore is not counted in the size of the elementary item. When the assumed decimal point is to the right of the right-most symbol in the string, the V is redundant. If the character-string includes the symbol . (period), the V cannot be used; to do so would imply that the data has two decimal points.

X

Each X in the character-string represents a character position in the data item that may contain any HP 3000 character.

Z

Each Z in the character-string represents a leading numeric character position. When that position contains a non-significant zero, the zero is replaced by a space character. Each Z is counted in the size of the item.

9

Each 9 in the character-string represents a character position which contains a numeral. Each 9 is counted in the size of the item; however, the size of the item and the amount of storage required for the item may differ depending on the USAGE assigned to the item.

0

Each 0 (zero) in the character-string represents a character position into which the numeral zero will be inserted. The 0 is counted in the size of the item.

, (comma)

Each , (comma) in the character-string represents a character position into which a comma will be inserted. This character position is counted in the size of the item. The insertion character comma must not be the last symbol in the character-string.

. (period)

The symbol . (period) may appear only once in the character-string. This symbol represents the decimal point for alignment purposes. Also, the period is an insertion editing character and will be inserted into the item. The period is counted in the size of the item. The period must not be the last character in the character-string.

Note: Within a given program, the functions of the period and the comma may be exchanged through the use of the DECIMAL-POINT IS COMMA in the SPECIAL-NAMES paragraph of the Environment Division. In this exchange, the rules for the period apply to the comma, and the rules for the comma apply to the period whenever they appear in a character-string.

+, -, CR, DB

The editing sign control symbols + (plus), - (minus), CR (credit), and DB (debit) are mutually exclusive; if, for example, a character-string includes the CR sign control symbol, no other sign control symbol may appear in the same character-string. The editing sign control symbols occupy character positions and must be included in the length of the item. However, no other data can ever be moved into these character positions.

The editing sign control symbol specified in the PICTURE clause may appear in altered form in the data item. For example, if a picture contains the CR symbol, and a positive value is moved to the data item described by this picture, the CR symbol is replaced by two spaces. (Editing rules are fully explained on the following pages.)

* (asterisk)

The editing symbol * (asterisk) in a character-string represents a leading numeric character position. When that position contains a non-significant zero, the zero is replaced by an asterisk. Each asterisk is counted in the size of the item.

The asterisk editing symbol is often referred to as a check-protection character. The character is used to replace leading zeros in dollar amounts printed on checks to help prevent tampering.

Currency Symbol

A currency symbol in the character-string represents a character position into which the currency symbol is to be placed. This compiler normally assigns the character \$ as the currency symbol. The programmer may assign some other character as the currency symbol through the use of the CURRENCY SIGN clause in the SPECIAL-NAMES paragraph in the Environment Division. The currency symbol occupies a character position and is counted in the size of the item.

The currency symbol may not be any of the digits 0-9 nor any symbol that might normally appear in the character-string of a PICTURE clause. Therefore, none of the following may be assigned as a currency symbol:

A B C D P R S V X Z 0 through 9 , . + - *

Editing Conventions

The PICTURE clause provides two basic kinds of editing: character insertion, and character suppression/replacement. Editing takes place only when data is moved into an elementary data item whose PICTURE clause specifies editing. Edited data is intended primarily for output to some external media such as a printed report. Because data that contains editing characters cannot be used in computations, editing is normally one of the final processing steps.

SIMPLE INSERTION EDITING. Simple insertion editing is performed using the editing symbols B (blank), , (comma), and 0 (zero) in the character-string of a PICTURE clause. These editing symbols are then automatically inserted into the corresponding character positions of the data item as blanks, commas, and zeros, respectively. The inserted characters must be counted in the size of the item.

EXAMPLES

The following examples illustrate the effect of simple insertion editing. Notice that the rules for standard data positioning and truncation also apply to edited items. In the examples, the character Δ represents a space.

VALUE TO BE STORED	PICTURE OF RESULT	RESULT
011673	99B99B99	01Δ16Δ73
987654321	99,999,999	87,654,321 †
57	99,000	57,000
AKAY	AB0AAA	AΔ0KAY
PHOTOGRAPH	ABABABABA	PΔHΔOΔTΔO †

† Item is truncated according to rules for standard data positioning.

SPECIAL INSERTION EDITING. The editing symbol . (period) in a character-string represents a decimal point for item alignment. Thus, the symbol is treated as both an insertion character and as an indication of the decimal point location. In the clause "PICTURE 9.9." the period between the 9's is an editing symbol. The right-hand period terminates the COBOL sentence and is not part of the character-string.

When data is moved to an item defined with the special insertion character period, the compiler automatically provides truncation and/or zero fill to both the left and right of the decimal point. (Other editing conventions may override zero fill. These conventions are explained on the following pages.)

EXAMPLES

The following examples illustrate the effect of the special insertion character . (period) as well as the simple insertion character , (comma). In the examples, the caret (^) indicates the position of the implied decimal point.

VALUE TO BE STORED	PICTURE OF RESULT	RESULT
654321 ^	9,999.99	6,543.21
01 ^	9,999.99	0,000.01
87654321 ^	9,999.99	6,543.21 †
123321 ^	9.9	3.3 †

† Item is truncated according to the rules for standard data positioning.

FIXED INSERTION EDITING. Fixed insertion editing is accomplished through the use of the editing sign control symbols + (plus), - (minus), CR (credit), and DB (debit) and/or the currency symbol \$. Only one sign control symbol and one currency sign symbol may appear in the character-string of the PICTURE clause. (However, the programmer may specify multiple sign control symbols and currency sign symbols if floating insertion editing is desired. Floating insertion editing is explained below.)

The character positions occupied by the currency symbol and sign control characters must be counted in the total length of the item. For example, an item with the PICTURE \$9,999.99CR has a length of eleven characters. However, only six of the eleven character positions are available for digits. If required, the compiler will truncate data to both the left and the right of the decimal point in order to make it fit the receiving field; neither the currency sign nor the sign control positions will be made available for data.

The sign control symbols CR and DB each occupy two character positions in the item. These symbols must be counted in the size of the item. CR and DB, if used, must be the right-most characters in the character-string. These sign control symbols appear only when the item contains a negative value. If the item contains a positive value, the sign control positions are filled with spaces.

The sign control characters + and - each occupy one character position and must be counted in the length of the item. If used, + or - must be either the left-most or right-most symbol in the character-string. The + is the only sign control symbol that causes the insertion of a sign character for either a positive or a negative value. A + is inserted when the item contains a positive value; a - is inserted when the item contains a negative value. The - (minus) sign control symbol causes a - to be inserted only when the item contains a negative value; the sign control position is filled with a space if the item contains a positive value.

The currency symbol (\$) unless changed in the SPECIAL NAMES paragraph) occupies one character position and must be counted in the length of the item. If used, the currency symbol must be the left-most symbol in the character-string except when a + or a - is the left-most symbol in the character-string. In this case, the currency symbol may appear immediately to the right of the sign control symbol.

The following chart summarizes the effects of the sign control symbols.



EFFECT OF EDITING SIGN CONTROL SYMBOLS		
EDITING SYMBOL IN CHARACTER-STRING	RESULT	
	DATA ITEM POSITIVE	DATA ITEM NEGATIVE
+	+	-
-	space	-
CR	2 spaces	CR
DB	2 spaces	DB

EXAMPLES

The following examples illustrate the effects of the fixed insertion characters \$, +, -, CR, and DB. In addition, these examples also include the comma and period characters since these are usually found in combination with fixed insertion editing. In the examples, the sign of the value to be stored is shown as the right-hand character of the value.

VALUE TO BE STORED	PICTURE OF RESULT	RESULT
654321 [^] +	+\$9,999.99	+\$6,543.21
654321 [^] -	+\$9,999.99	-\$6,543.21
654321 [^] -	\$9,999.99+	\$6,543.21-
654321 [^] +	\$9,999.99-	\$6,543.21
654321 [^] -	\$9,999.99-	\$6,543.21-
12345 [^] -	\$999.99CR	\$123.45CR
12345 [^] +	\$999.99CR	\$123.45
12345 [^] +	\$999.99DB	\$123.45
12345 [^] -	\$999.99DB	\$123.45DB
0000 [^]	+999.99	+000.00 †

† This compiler treats zero as a unique value which is always assumed to be positive.

ZERO SUPPRESSION EDITING. Zero suppression editing in a numeric item permits the replacement of leading zeros by spaces or asterisks. The suppression and replacement of leading zeros is indicated by one or more contiguous Z's or *'s (asterisks) in the character-string of a PICTURE clause. These symbols are mutually exclusive in the character-string. The editing symbol Z specifies replacement of leading zeros with the space character; the symbol * specifies replacement of leading zeros with the asterisk character.

Embedded within the zero suppression editing symbols may be the special insertion editing symbol . (period) and the simple insertion editing symbols , (comma), B (blank), and 0 (zero). Also, the character-string may contain a fixed insertion currency sign symbol and sign control symbol. If a leading zero to the right of a simple insertion symbol is suppressed, then the simple insertion character also is suppressed. Thus, if the value 000000100 is moved into an item with the PICTURE Z,ZZZ,ZZZ.99, the result appears as 1.00 rather than $\Delta,\Delta\Delta\Delta,\Delta\Delta 1.00$.

The programmer has two options when representing zero suppression editing in the PICTURE clause: He may use zero suppression editing symbols to represent some or all of the character positions to the left of the decimal point, or he may use zero suppression editing symbols to represent the entire item. Thus the PICTURE's ZZ.99 and ZZ.ZZ are acceptable; the PICTURE ZZ.Z9 is incorrect and must not be used. The compiler will apply zero suppression either to all or none of the character positions to the right of the decimal point.

If the zero suppression editing symbols are only to the left of the decimal point, zero suppression and replacement takes place up to the first non-zero character or up to the decimal point, whichever is encountered first in scanning the item from left to right.

If the zero suppression editing symbols represent the entire item, the editing results depend on the value moved into the item:

- If the value is other than zero, the result is the same as if zero suppression characters were located only to the left of the decimal point.
- If the value is zero and the editing symbols are Z's, then the entire item is replaced by spaces.
- If the value is zero and the editing symbols are *'s, then the item contains all asterisks except for the decimal point which is not suppressed within the item.

When the asterisk is used as the zero suppression symbol and the BLANK WHEN ZERO clause appears in the same entry, the zero suppression editing overrides the function of the BLANK WHEN ZERO clause.

EXAMPLES

VALUE TO BE STORED	PICTURE OF RESULT	RESULT
000 [^] 321	Z,ZZZ.99	3.21
000 [^] 321	*,***.99	****3.21
0000 [^] 00	Z,ZZZ.ZZ	$\Delta\Delta\Delta\Delta\Delta\Delta\Delta$
0000 [^] 00	*,***.**	*****.**
0000 [^] 01	Z,ZZZ.ZZ	.01
0000 [^] 01	*,***.**	*****.01
1123 [^] 12	*,***.**	1,123.12

FLOATING INSERTION EDITING. Floating insertion editing is similar to zero suppression editing in that it can replace leading zeros with space characters. However, floating insertion editing also permits the insertion of the currency symbol, +, or - immediately to the left of the first non-zero digit in the field. For example, if the value 0000345 is moved to an item with the PICTURE \$\$\$\$\$.99, the result will appear as $\Delta\Delta\Delta\$3.45$.

Floating insertion editing requires a character-string containing at least two of the permissible insertion symbols in the character-string. These symbols represent those left-most character positions into which a floating insertion character can be placed. Notice that the character-string must specify at least one extra floating insertion symbol beyond the left-most character of the value; this is to ensure that the desired character will be inserted even if there is no leading zero in the value.

In floating insertion editing, the insertion symbols +, -, and currency symbol are mutually exclusive. The character-string may contain the following insertion characters:

. , B 0 CR DB

When the character-string contains simple insertion symbols, it is possible that the floating insertion character will replace one of the insertion characters. For example, if the value 0056456 is moved into an item with the PICTURE \$\$\$,\$\$.99, the result appears as $\Delta\Delta\Delta\$564.56$. In this example, the \$ replaces the comma insertion character.

The programmer has two options when representing floating insertion editing in the character-string of a PICTURE clause: He may use floating insertion symbols to represent some or all of the character positions to the left of the decimal point, or he may use floating insertion symbols to represent the entire item. Thus the PICTURE's \$.99 and \$\$. are acceptable; the PICTURE \$\$.\$9 is incorrect and must not be used.

If the floating insertion symbols are all to the left of the decimal point, a single floating insertion character is placed in the character position to the immediate left of the first non-zero digit or the decimal point, whichever is encountered first in scanning the item from left to right.

If the floating insertion symbols represent the entire item, the result depends on the value moved into the item. If the value is zero, the entire item is filled with spaces. If the value is anything other than zero, the result is the same as though the floating insertion symbols were located only to the left of the decimal point.

EXAMPLES

VALUE TO BE STORED	PICTURE OF RESULT	RESULT
43576 [^] +	\$\$\$\$.99	\$435.76
00033 [^] -	++++.99	-.33
00033 [^] +	++++.99	+.33
00033 [^] +	----.99	.33
00033 [^] -	----.99	-.33
00000 [^] +	\$\$\$\$.99	\$.00
00001 [^] +	\$\$\$\$.\$\$	\$.01
00000 [^]	\$\$\$\$.\$\$	△△△△△△△

Sequence of Character-String Symbols

The following table defines the precedence of symbols which can be used in the character-string of a PICTURE clause.

Those symbols (First Symbol) which can precede (be to the left of) of the given symbol in the left most column (Second Symbol) are indicated by an x. A blank indicates that the symbol in the row cannot precede the symbol in the column.

At least *one* of these symbols: A, X, Z, 9, or *, or at least *two* of these symbols: +, -, or CS must be present in the character-string of a PICTURE clause.

CS is an abbreviation for the currency symbol.

These symbols: + and -, (non-floating insertion) and Z, *, +, -, and CS (floating insertion) occur twice. Their first appearance in a row and column represents their use to the left of the decimal point position; their second appearance represents their use to the right of the decimal point position.

Precedence of Symbols Used in the PICTURE Clause

FIRST SYMBOL \ SECOND SYMBOL		Non-floating Insertion Symbols							Floating Insertion Symbols						Other Symbols						
		B	0	,	.	+ -	+ -	CR DB	CS	Z *	Z *	+ -	+ -	CS	CS	9	A X	S	V	P	P
Non-floating Insertion Symbols	B	X	X	X	X	X			X	X	X	X	X	X	X	X		X		X	
	0	X	X	X	X	X			X	X	X	X	X	X	X	X		X		X	
	,	X	X	X	X	X			X	X	X	X	X	X	X			X		X	
	.	X	X	X		X			X	X		X		X		X					
	+ or -																				
	+ or -	X	X	X	X				X	X	X			X	X	X			X	X	X
	CR or DB	X	X	X	X				X	X	X			X	X	X			X	X	X
CS					X																
Floating Insertion Symbols	Z or *	X	X	X		X			X	X											
	Z or *	X	X	X	X	X			X	X	X							X		X	
	+ or -	X	X	X					X			X									
	+ or -	X	X	X	X				X			X	X						X		X
	CS	X	X	X		X								X							
	CS	X	X	X	X	X								X	X				X		X
Other Symbols	9	X	X	X	X	X			X	X		X		X		X	X	X		X	
	A X	X	X												X	X					
	S																				
	V	X	X	X		X			X	X		X		X		X		X		X	
	P	X	X	X		X			X	X		X		X		X		X		X	
	P					X			X									X	X		X

REDEFINES CLAUSE

The REDEFINES clause allows the same storage area to contain different data items or provides an alternative grouping or description of the same data. Notice that the REDEFINES clause specifies the redefinition of a storage area, not of the data items that occupy the area.

Redefinition of a data item permits the programmer to assign different data structures to the same memory area. For example, if the programmer wishes to test a field with both the IF NUMERIC and IF ALPHABETIC statements, the field must be defined as X-type data or else the compiler will issue a warning message. Then, if the same field is to be used in an arithmetic operation, it must also be defined as 9-type data. These multiple definitions are accomplished through the use of the REDEFINES clause.

Redefinition of an area also makes it possible to use the same memory area for entirely unrelated data items during different phases of the program. For example, more than one printed report can be built in the same area. This sharing of the same data area by different items reduces the memory requirements for the program.

FORMAT

level-number data-name-1 REDEFINES data-name-2

The level numbers of the REDEFINES entry and data-name-2 must be identical, but cannot be 66 or 88. Data-name-2 is the name of the item being redefined. Data-name-1 is an alternate name for the same area. When written, the REDEFINES clause must immediately follow data-name-1.

The redefinition of a memory area must immediately follow the item being redefined.

The REDEFINES entry may have subordinate entries. If subordinate data entries are present, the redefinition begins with the REDEFINES entry and continues until an entry is encountered with a level number equal to or lower than the number of the REDEFINES entry. The redefinition must specify a number of characters equal to or less than the item being redefined.

The following coding example allows the program to deal with transaction codes in a variety of formats.

```
03 TRANSACTION-TYPE PIC X(9).
03 TRAN-TYPE-1 REDEFINES TRANSACTION-TYPE.
    05 LOCATION-NO PIC 999.
    05 TRAN-CODE PIC AAAAAA.
03 TRAN-TYPE-2 REDEFINES TRANSACTION-TYPE.
    05 TYPE-NO PIC 999.
    05 LOC-NO PIC 999A.
```

Notice that to redefine the same memory area more than once, the programmer must use a series of REDEFINES entries.

The REDEFINES clause must not be used at the 01-level in the File Section. Redefinition is implied when more than one 01-level entry follows a file description entry. Such 01-level redefinitions may be of differing lengths.

The data-description entry for data-name-2 must not contain an OCCURS clause, nor can data-name-2 be subordinate to any entry containing an OCCURS clause. An item subordinate to data-name-2 may contain an OCCURS clause without the DEPENDING ON option. Data-name-1 or any items subordinate to data-name-1 may contain an OCCURS clause without the DEPENDING ON option.

If any item subordinate to the REDEFINES clause includes the SYNCHRONIZED clause, the programmer must be certain that the redefined item has the proper word boundary alignment.

Items subordinate to a REDEFINES entry may not contain any VALUE clauses except for 88-level condition-names.

EXAMPLE

The REDEFINES clause allows the programmer to assign different characteristics to a given area of memory. The following example illustrates the Data Division coding for an application which uses a number of different discount rates for various customers. The discount rate may range from 1.250 to 33.33. Notice that this requires the discount field to be defined as having either two or three decimal positions.

```
05 DISCOUNT PICTURE 9V999.  
05 PREFERRED-CUSTOMER-DISCOUNT REDEFINES DISCOUNT  
    PICTURE 99V99.
```

RENAMES CLAUSE

The RENAMES clause permits alternate, possibly overlapping, groupings of elementary items. Also, the RENAMES clause allows the user to assign alternate names to elementary or group items in order to avoid the need for qualifying frequently used data-names in the Procedure Division.

FORMAT

```
66 data-name-1 RENAMES data-name-2 [THRU data-name-3]
```

All RENAMES entries associated with a given logical record must immediately follow its last data description entry.

One or more RENAMEs entries can be written for a logical record.

A 66-level entry cannot rename another 66-level entry, nor can it rename a 77, 88, or 01-level entry.

Data-name-1 cannot be used as a qualifier, and can be qualified only by the names of the level 01 or RD entries.

Data-name-2 and data-name-3 must be names of elementary or group items in the associated record, and must not be the same name. Data-name-2 must precede data-name-3 in the record description, and, if the record description includes any redefinition, the beginning point of data-name-3 must physically follow the beginning point of data-name-2. Data-name-3 cannot be the name of any item subordinate to data-name-2.

Neither data-name-2 nor data-name-3 may have an OCCURS clause in its description.

Data-name-2 and data-name-3 may be qualified.

If the THRU option is not specified, data-name-1 and data-name-2 refer to the same item, whether it is a group item or an elementary item. The programmer can avoid the use of qualification in the Procedure Division through the use of this technique.

If the THRU option is specified, data-name-1 is a group item which begins with the first elementary item within data-name-2 (or with data-name-2 itself if that is an elementary item) and ends with the last elementary item in data-name-3 (or with data-name-3 itself if that is an elementary item).

EXAMPLES

In the following set of examples, examples 1 and 2 are equivalent; example 3 is incorrect because data-name-3 (STATE) is subordinate to data-name-2 (EMPLOYEE-ADDRESS).

EXAMPLE 1	EXAMPLE 2	EXAMPLE 3 (INCORRECT)
03 EMPLOYEE-ADDRESS	03 EMPLOYEE-ADDRESS	03 EMPLOYEE-ADDRESS
05 STREET	05 STREET	05 STREET
05 CITY	05 CITY	05 CITY
05 STATE	05 STATE	05 STATE
05 ZIP	05 ZIP	05 ZIP
66 ADDRESS RENAMES	66 ADDRESS RENAMES	66 ADDRESS RENAMES
EMPLOYEE-ADDRESS.	STREET THRU ZIP.	EMPLOYEE-ADDRESS THRU STATE.

SYNCHRONIZED CLAUSE

The SYNCHRONIZED clause specifies the alignment of an elementary item on a natural boundary of the computer memory. For the HP 3000, the 16-bit (or two byte) *word* constitutes a natural memory boundary.

FORMAT

$$\left\{ \begin{array}{l} \underline{\text{SYNCHRONIZED}} \\ \underline{\text{SYNC}} \end{array} \right\} \quad \left[\begin{array}{l} \underline{\text{LEFT}} \\ \underline{\text{RIGHT}} \end{array} \right]$$

When the SYNCHRONIZED clause is used, this compiler aligns certain items on word boundaries to facilitate arithmetic operations. Only those items defined with the USAGE IS COMPUTATIONAL or the USAGE IS INDEX clauses are affected by the SYNCHRONIZED clause. All other items are aligned on byte (or character) boundaries. Since the character is the smallest directly-addressable unit within the COBOL language, the SYNCHRONIZED clause has no meaning when applied to an item with any usage other than COMPUTATIONAL or INDEX.

This compiler automatically assigns all level-01 and level-77 items starting on a word boundary.

If the SYNCHRONIZED item does not fall naturally on a word boundary, the compiler assigns the item the next higher byte address, which will always be a word boundary. This has the effect of inserting an extra FILLER byte — called a slack byte — before the SYNCHRONIZED item. The slack byte does not affect the length of the SYNCHRONIZED item. However, slack bytes are included in the length of the group item to which the SYNCHRONIZED item or items belong.

If the SYNCHRONIZED clause is not specified, no space is reserved for slack bytes. When computation is performed on unaligned COMPUTATIONAL data, the compiler first moves the data to a work area which has the alignment required for computation.

Because of the word structure used by the HP 3000, the LEFT and RIGHT options are irrelevant; this compiler treats the LEFT and RIGHT options as comments.

When the SYNCHRONIZED clause is specified for an item that also contains a REDEFINES clause, the programmer must ensure that the redefined item has the proper boundary alignment for the item that redefines it. Thus, in the following coding example, the programmer must be certain that ITEM-1 begins on a word boundary:

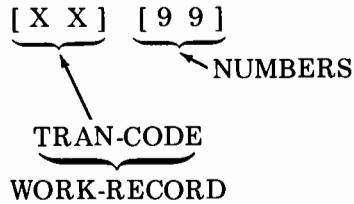
```
02  ITEM-1 PICTURE X(4).  
02  ITEM-2 REDEFINES ITEM-1 PIC S9(4) USAGE COMP SYNC.
```

When the SYNCHRONIZED clause is specified for an item within the range of an OCCURS clause, each occurrence of the item is synchronized.

Slack Bytes

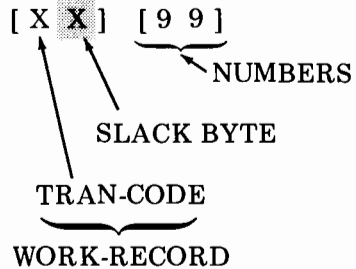
The following examples illustrate the effect of the SYNCHRONIZED clause. Each example includes both coding samples and a graphic illustration of the effect of the coding in memory. In the illustrations of memory, each box represents one word or two bytes. A slack byte is represented by shading.

```
01 WORK-RECORD.  
   02 TRAN-CODE PICTURE XX.  
   02 NUMBERS PICTURE S99 COMP SYNCHRONIZED.
```



In the example above, the SYNCHRONIZED field NUMBERS happen to fall on a word boundary. No slack byte is added, and, in this case, the SYNCHRONIZED clause serves only as documentation.

```
01 WORK-RECORD.  
   02 TRAN-CODE PICTURE X.  
   02 NUMBERS PICTURE S99 COMP SYNCHRONIZED.
```



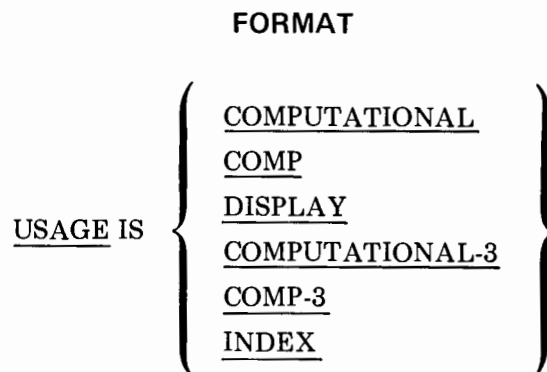
In this example, the SYNCHRONIZED field NUMBERS does not fall naturally on a word boundary. Therefore, the compiler added a slack byte following the field TRAN-CODE. The effect is the same as though the programmer had coded one byte of FILLER between TRAN-CODE and NUMBERS. The slack byte is similar to FILLER in that it cannot be addressed in the program.

From the above examples, it can be seen that the SYNCHRONIZED clause generates a slack byte only when the SYNCHRONIZED item is preceded in the record by an odd number of bytes.

Slack bytes are carried in file media. The programmer must be certain to account for any slack bytes in a file record. Otherwise, the record description will become misaligned following the slack byte.

USAGE CLAUSE

The USAGE clause specifies the storage format of a data item within the computer storage.



The USAGE clause can be written at any level. If the USAGE clause is written at the group level, it applies to each elementary item in the group. Neither the USAGE clause nor the PICTURE clause of any of the elementary items may contradict the USAGE clause of the group.

If the usage clause is not specified for an elementary item or for any group to which the item belongs, the USAGE is assumed to be DISPLAY. Thus the programmer is not required to code the USAGE IS DISPLAY clause except as documentation.

COMP is an abbreviation for COMPUTATIONAL; COMP-3 is an abbreviation for COMPUTATIONAL-3.

This clause specifies the manner in which a data item is represented in memory. It does not affect the use of the data item, although the specifications for some statements in the Procedure Division may require certain operands to have a particular type of USAGE clause.

DISPLAY Option

The DISPLAY option specifies that the data item is stored as ASCII characters, one character per eight-bit byte. This is the form in which information is represented for punched cards and paper tape and for printed information.

The compiler treats unsigned numeric data entered in DISPLAY format as though it were positive.

Signed decimal fields entered through punched cards are known as zone-signed fields. To represent a positive value, an overpunch is placed in the 12-zone above the right-most digit of the field; to represent a negative value, an overpunch is placed in the 11-zone above the right-most digit of the field. Zone signs cause the signed digit to have the same punch configuration as some other data character. Therefore, the programmer must be certain to include the sign symbol (S) in the picture clause for each zone-signed field. The following tables show the data character equivalents for zone-signed digits:

POSITIVE VALUES									
1	2	3	4	5	6	7	8	9	0
A	B	C	D	E	F	G	H	I	{

NEGATIVE VALUES									
1	2	3	4	5	6	7	8	9	0
J	K	L	M	N	O	P	Q	R	}

When DISPLAY items are used for computations, the compiler operates the necessary data conversion routines.

Zoned overpunches in positions other than the rightmost digit of the field are ignored. If a data item is used as a numeric item and it contains zoned overpunches in positions other than the rightmost digit, it will be converted to standard numeric DISPLAY format. For example, 1B3D will be converted to 123D. This conversion takes place even if the data item is being used in a comparison. The data item continues to have this value for all references to it that follow the conversion.

COMPUTATIONAL Options

A COMPUTATIONAL or COMPUTATIONAL-3 item represents a numeric value. The PICTURE of a COMPUTATIONAL or COMPUTATIONAL-3 item can contain only 9's, the operational sign character S, the implied decimal point character V, and one or more P's.

If a group item is described as COMPUTATIONAL or COMPUTATIONAL-3, each elementary item in the group is COMPUTATIONAL or COMPUTATIONAL-3, respectively, and may be used in arithmetic operations. However, the group itself is considered to be alphanumeric and cannot be used in arithmetic operations.

USAGE IS COMPUTATIONAL: For the HP 3000, this option specifies binary data items.

Each COMPUTATIONAL item is assigned storage according to its PICTURE clause as indicated below:

PICTURE	NUMBER OF BYTES ASSIGNED
S9 to S9(4)	2
S9(5) to S9(9)	4
S9(10) to S9(18)	8

Although COMPUTATIONAL items are assigned storage in fixed groups of bytes, these items still follow the normal rules for truncation. For example, if the item FIELD-A has the picture S9(5), the statement MOVE +123456789 TO FIELD-A places the binary equivalent of +56789 in FIELD-A. Computational items may not contain more than 18 decimal digits.

The left-most bit in a COMPUTATIONAL item is the sign bit. This bit is off for a positive value, on for a negative value. Negative values are represented in two's complement format.

USAGE IS COMPUTATIONAL-3: Each COMPUTATIONAL-3 item is stored in signed packed decimal format. In this format, there are two digits per byte, with a sign in the low order four bits of the right-most byte.

Each COMPUTATIONAL-3 item may contain up to 18 digits plus a sign. If the picture for the item does not contain a sign, the sign position in the data field is occupied by a bit configuration that is interpreted as positive. The following chart illustrates the bit configurations used to represent signs in packed decimal fields. Notice that the bit configuration 1100 specifies a positive value; however, any four-bit configuration in the sign position other than that reserved for a negative value (1101) is also considered positive.

Note: The chart also shows the hexadecimal equivalents for these bit configurations, although the HP 3000 architecture is not based on this numbering system. Signed packed decimal fields are effectively in a hexadecimal format.

SIGN	BIT CONFIGURATION	HEXADECIMAL VALUE
+	1100	C
-	1101	D

The following is a graphic illustration of packed decimal fields as they might appear in memory or in a magnetic file device. Notice that these items follow the normal rules for truncation, even though the field may include an unused half-byte position. This compiler fills the unused half-byte with a zero; however, the contents of this half-byte are unpredictable when data is interchanged with other computer systems. In the illustration, byte-boundaries are indicated by a box.

VALUE TO BE STORED	PICTURE OF RESULT	RESULT																												
+1234 [^]	S9999	<table border="1"> <tr><td></td><td>01</td><td>23</td><td>4C</td></tr> <tr><td></td><td>12</td><td>34</td><td>5C</td></tr> <tr><td></td><td>12</td><td>34</td><td>5F</td></tr> <tr><td>00</td><td>01</td><td>20</td><td>0D</td></tr> <tr><td>00</td><td>00</td><td>50</td><td>0D</td></tr> <tr><td>00</td><td>00</td><td>12</td><td>2C</td></tr> <tr><td></td><td>12</td><td>34</td><td>5F</td></tr> </table>		01	23	4C		12	34	5C		12	34	5F	00	01	20	0D	00	00	50	0D	00	00	12	2C		12	34	5F
	01	23	4C																											
	12	34	5C																											
	12	34	5F																											
00	01	20	0D																											
00	00	50	0D																											
00	00	12	2C																											
	12	34	5F																											
+12345 [^]	S99999																													
12345 [^]	99999																													
-12 [^]	S999V999																													
-5 [^]	S999V999																													
+122172 [^]	S999V999																													
-12345	99999																													

Storing Numeric Data in Magnetic Files

When the programmer has the option of selecting the format of numeric data to be recorded on a magnetic file device, two factors must be considered: program efficiency and the amount of storage space required.

DISPLAY items require the greatest amount of storage since they require one byte for each digit. Storage requirements for COMPUTATIONAL and COMPUTATIONAL-3 items vary according to the number of digits to be stored. For example, an item with the picture S9(10) requires six bytes of storage if its usage is COMPUTATIONAL-3 and eight bytes if its usage is COMPUTATIONAL. However, an item with the picture S9 (9) requires five bytes of storage if its usage is COMPUTATIONAL-3 and only four bytes if its usage is COMPUTATIONAL.

Note: The above examples do not account for the possibility of a slack byte within a COMPUTATIONAL item. Slack bytes are explained under the SYNCHRONIZED Clause.

INDEX Option

The INDEX option specifies an index data item in which the programmer may save the contents of a table index when processing data tables. Any value stored in an index data item must correspond to an occurrence number of a table-element. In the Procedure Division, an index data item can be referred to only by the SEARCH or SET statements or in a relation condition.

If a group item is defined with the USAGE IS INDEX clause, each item within the group is an index data item. However, the group itself is not an index data item and cannot be used in SEARCH or SET statements or in a relation condition. An index data item may be part of a group which is referred to in a MOVE or input-output statement.

When an item is described using the USAGE IS INDEX clause, the compiler automatically assigns all the proper attributes to the item. (This compiler stores index data items as two-byte binary items.) No other data description clauses such as BLANK WHEN ZERO, PICTURE, VALUE, or JUSTIFIED may be used with an index data item. However, this compiler allows the use of the SYNCHRONIZED clause for an index data item, thus providing maximum efficiency.

EXAMPLE

The following index data item descriptions are equivalent:

- 77 SAVE-MONTH-POINTER USAGE IS INDEX.
- 77 SAVE-STOCK-POINTER USAGE IS INDEX.
- 01 SAVE-INDEX USAGE IS INDEX.
- 02 SAVE-MONTH-POINTER.
- 02 SAVE-STOCK-POINTER.

VALUE CLAUSE

The VALUE clause specifies the initial value of Working-Storage items, or the values associated with an 88-level condition-name. For initial values, Working-Storage items are set to the values indicated in their respective VALUE clauses when the object program is loaded into memory. If the programmer omits the VALUE clause, the content of the items are unpredictable when the program is loaded.

FORMAT-1

VALUE IS literal

FORMAT-2

VALUE IS literal-1 [THRU literal-2]

[, literal-3 [THRU literal-4]] . . .



Initial Values

If the PICTURE clause defines an item as numeric, the literal in the VALUE clause must also be numeric. All numeric literals are automatically aligned to the implied decimal point in the PICTURE clause. If the numeric literal does not take up all character positions in the item, the compiler fills the remaining characters with zeros. However, the literal must not have a value that would require truncation of any non-zero characters.

If the PICTURE clause defines an item as alphabetic, the literal in the VALUE clause must also be alphabetic, and must be enclosed in quotes. The value is placed left-justified in the field. If the literal does not take up all character positions in the item, the compiler fills any remaining characters at the right-hand end of the field with spaces. However, the JUSTIFIED clause may specify non-standard positioning for the literal.

If the PICTURE clause defines an item as alphanumeric, the literal in the VALUE clause may contain any combination of HP 3000 characters, and must be enclosed in quotes. However, to maintain full compatibility with the COBOL language, the literal should be confined to the characters of the COBOL language.

The VALUE clause is the only clause that provides initialization. Other clauses such as BLANK WHEN ZERO, etc., have no effect on initial values.

Editing characters are included in the size of an item used for initial values.

A figurative constant such as ZEROS or SPACES may be substituted in both Format-1 or Format-2.

All numeric literals in a VALUE clause must have a value which is within the range of values indicated in the PICTURE clause. For example, the range of values permitted for an item with the PICTURE PPP99 are .00000 through .00099. However, the literal 55 is interpreted as 55.00000, and is not allowed. Thus, the literal must be compatible with any scaling position (indicated by the character P) included in the PICTURE.

Except for condition-name entries, the VALUE clause may not be used in the File Section or in the Linkage Section. Establishing an initial value for an area that will be filled with other data at the first I-O statement is illogical.

Except for condition-name entries, the VALUE clause must not be stated in a data description entry that contains an OCCURS clause or that is subordinate to an OCCURS clause. If the programmer wishes to establish initial values for a table, he must first enter the values, and then REDEFINE the area using the OCCURS clause.

If the VALUE clause is written at the group level, the literal must be a figurative constant or a non-numeric literal. In such a case, the group is initialized without regard for any items subordinate to the group. Also, the elementary items within the group may not contain the VALUE clause.

The VALUE clause must not be written for group items containing items with descriptions that include JUSTIFIED, SYNCHRONIZED, or any USAGE other than DISPLAY.

Condition-Names

The VALUE clause is required in any 88-level entry. The VALUE clause and the condition-name itself are the only clauses permitted in the entry.

Format-2 can be used only for condition-names. When the THRU phrase is used, literal-1 must be less than literal-2, literal-3 must be less than literal-4, and so on.

EXAMPLES

The following coding illustrates the initial use of the VALUE clause at the group level. Each of the individual counters in the example might have been coded as 77-level items. The programmer chose to code the items as a group in order to avoid coding needless repetitions of the VALUE IS ZERO clause.

```
01  COUNTERS VALUE ZERO.  
    02  PAGE-COUNTER PICTURE 9999.  
    02  LINE-COUNTER PICTURE 9999.  
    02  ITEM-COUNTER PICTURE 9999.  
    02  ERROR-COUNTER PICTURE 9999.
```

The following coding establishes a set of initial values for a table. Notice that the table definition, which includes an OCCURS clause, does not contain any VALUE clause.

```
01  MONTH-LIST.  
    02  FILLER PIC X(9) VALUE 'JANUARY'.  
    02  FILLER PIC X(9) VALUE 'FEBRUARY'.  
        .  
        .  
        .  
    02  FILLER PIC X(9) VALUE 'DECEMBER'.  
  
01  MONTH-TABLE REDEFINES MONTH-LIST.  
    02  MONTH PIC X(9) OCCURS 12 TIMES INDEXED BY MONTH-INDEX.
```

The following code establishes condition-names for the variable TRAN-TYPE.

```
03  TRAN-TYPE PICTURE 9.  
    88  PURCHASE VALUE 1.  
    88  PAYMENT VALUE 2.  
    88  LAYAWAY VALUE 3.  
    88  CREDIT VALUE 4.
```


SECTION VII

Procedure Division

The Procedure Division is the fourth division of a COBOL program. The power and convenience of the COBOL language are most apparent in this division. The programmer uses simple, English-like statements to specify the operations required to produce the desired result. Because these statements are relatively easy to understand, the program is to some extent self-documenting. Also, the programmer can generate an efficient program with a minimum of knowledge about the object computer or its software operating system.

The Procedure Division defines the operations to be performed by the COBOL program. In general, statements are executed sequentially in the order in which they are entered to the compiler. The programmer alters the flow of the program through the use of branching statements such as GO TO and PERFORM or the conditional IF statement. In addition, the Procedure Division can specify declarative procedures to be executed only under special circumstances such as when an I-O error occurs. Declarative procedures are not part of the sequential program structure.

Conventions

GENERAL FORMAT

PROCEDURE DIVISION [USING identifier-1 [, identifier-2] ...].

```
[ DECLARATIVES.
  { section-name SECTION. declarative-sentence.
    { paragraph-name. { sentence } ... } ... } ...
END DECLARATIVES. ]
```

{[section-name SECTION [priority-number].]

{paragraph-name. {sentence.} ...} ...} ...

Note: If a declaratives section is included, the remainder of the procedure division must be defined as one or more sections. Each section must contain at least one paragraph.

Every program must have a Procedure Division. This Division is identified by and must begin with the header PROCEDURE DIVISION followed by a period and a space. If present, the optional USING clause must appear before the period in the Procedure Division header. The USING clause is necessary if this object program is called by another object program containing a CALL statement with a USING clause. (See the CALL statement for a discussion of the USING clause.)

Declaratives

Declarative sections are optional. If included, they must appear first in the Procedure Division, preceded by the keyword DECLARATIVES and followed by the keywords END DECLARATIVES. A declarative procedure consists of an introductory sentence containing a USE statement followed by the procedure to be executed. USE statements are not executed; they define the conditions calling for the execution of the USE procedures. A separate section must be coded for each USE statement entered. Declarative procedures must not reference any non-declarative procedure. It is permissible for a PERFORM statement in the non-declarative portion of the program to refer to procedures associated with a USE declarative.

General Structure and Format

PROCEDURES

Procedures to be executed within the object program are defined either by sections or paragraphs. A procedure-name is a programmer-assigned word used to refer to a paragraph or section. If the programmer wishes to alter the flow of the program, he names a procedure in one of the branching statements (ALTER, GO TO or PERFORM).

Execution of the object program begins with the first statement in the non-declarative portion of the Procedure Division. Statements are executed in the order in which they appear in the source program unless the rules specify otherwise. The end of the Procedure Division signifies the physical end of the COBOL program.

Sections

A program may optionally be divided into sections. This technique of dividing the program into logical modules simplifies debugging and updating of the program. If sections are used, then:

- each section heading must be followed by a paragraph-name
- a section may contain one or more paragraphs
- all paragraphs in the program must be part of a section.

If no section heading is specified, the entire program constitutes one section. Section-names must always have a unique spelling.

The programmer may assign each section a priority number in the section header. The priority-number must be an integer in the range 0-99. If the priority-number is omitted from the section header, the priority is assumed to be 0. All consecutive sections with the same priority-number constitute a program *segment* with that priority.

GO TO statements in sections with priority numbers greater than or equal to 50 must not be altered by ALTER statements in sections that have a different priority number. Alterable GO TO's in sections with priority-numbers greater than or equal to 50 are always brought into memory in their initial state when control is transferred to that section.

Paragraphs

Paragraph-names must have a unique spelling within a section (or a program if no sections are defined). However, the same paragraph-name may be used in different sections. If a paragraph-name that appears in more than one section is referenced, it is usually qualified by its section-name, for example, PERFORM EDIT-ROUTINE IN UPDATE. If such a paragraph-name is referenced by a branching statement within the same section (i.e., a local reference is made), qualification is optional but may be useful for better documentation and to avoid confusion. However, if it is referenced by a statement in a different section, it must be qualified. These rules about paragraph-names apply regardless of how the program is segmented.

General Structure and Format

Main Programs

A compiled COBOL program is divided into segments. A main program produces two or more code segments depending on the number of sections in the Procedure Division. The first segment contains a program unit called the *outer block*. This block initializes the run-time data area for the main program. The outer block must be the first program unit, which uses global storage, to be presented in the User Sub Program Library (USL) when the compiled program is prepared using the MPE Segmenter. (Refer to Appendix A for more information.) The base address for the main program data area must be DB+0. The outer block cannot reside in a Relocatable Library (RL) or a system, account, or group Segmented Library (SL).

The remaining segments of a main program consist of program units formed by one or more sections with the same priority number. If priority numbers are not specified, the Procedure Division is compiled as one segment.

The program shown below produces three segments, one for initialization and two for the Procedure Division code.

```
001000 IDENTIFICATION DIVISION.  
001100 PROGRAM-ID. MAIN.  
001200 ENVIRONMENT DIVISION.  
001300 DATA DIVISION.  
001400 PROCEDURE DIVISION.  
001450 FIRST-SEC SECTION.  
001500 START  
001600     DISPLAY "START OF MAIN PROGRAM".  
001700     CALL "SUB".  
001720 SECOND-SEC SECTION [02.] ← Priority  
001730     START-2.           Number  
001750     CALL "SUBA".  
001800     DISPLAY "END OF MAIN PROGRAM".  
001900     STOP RUN.
```

Refer to Appendix F, Internal Name Assignment, for a description of how segment names and the names of program units contained in the segments are created.

Subprograms

Each COBOL subprogram produces two segments, one for initialization and one for the Procedure Division code. Subprograms cannot be segmented.

Subprograms are identified by using one of the \$CONTROL command parameters, DYNAMIC or SUBPROGRAM. (Refer to Appendix B for a description of the \$CONTROL command.) A DYNAMIC subprogram uses Q-relative data allocation. Variables are initialized each time a call is made to the subprogram. This type of subprogram can reside in a system, account, or group Segmented Library (SL). When exit is made from the subprogram, the data area is released. Recursive calls to a dynamic subprogram can build up the stack quite rapidly and may cause a stack overflow condition. Use the MAXDATA parameter of the :PREP or :RUN command to increase your data area.

If a subprogram is compiled with \$CONTROL SUBPROGRAM in effect, it is called non-dynamic and uses DB-relative data allocation. The variables retain their values from one call to the subprogram to the next. Files which are opened remain open after exit is made from the subprogram. The data area is allocated during the entire execution time. This type of subprogram cannot reside in a system, account, or group Segmented Library (SL). Refer to the *MPE Commands Reference Manual*, Appendix B, for more information about DB-relative and Q-relative addressing.

General Structure and Format

STATEMENTS AND SENTENCES

The basic unit of the Procedure Division is the statement. A series of statements can be combined to form a sentence. A paragraph is made up of one or more sentences. The COBOL language contains three types of procedural statements:

Type	Function
imperative statements	instruct the object program to perform some action
conditional statements	test for a particular condition to determine which of a number of alternate operations will be performed
compiler-directing statements	direct the <i>compiler</i> to perform some action during compilation of the source program

COBOL/3000 Imperative Statements

ACCEPT	DIVIDE*	MULTIPLY*	STOP
ADD*	EXAMINE	OPEN	SUBTRACT*
ALTER	EXIT	PERFORM	
CALL	GO TO	RELEASE	WRITE**
CLOSE	GOBACK	SEEK	
COMPUTE*		SET	
DISPLAY	MOVE	SORT	

*Without the SIZE ERROR OPTION

**Without the INVALID KEY option

COBOL/3000 Conditional Statements

IF

READ

RETURN

SEARCH

WRITE (with an INVALID KEY option)


ADD

COMPUTE

DIVIDE

MULTIPLY

SUBTRACT


 (with the SIZE ERROR option)
COBOL/3000 Compiler-Directing Statements

COPY

ENTER

NOTE

Notice that certain statements appear as both imperative statements and conditional statements. For example, the arithmetic statements (ADD, SUBTRACT, etc.) are usually considered to be imperative. However, these statements may include the SIZE ERROR option which gives the user the option of branching to a procedure when an overflow condition (the computation generates a result too large to be stored in the specified result field) has been detected. Certain input-output statements may also branch to alternate procedures under specific conditions. The READ statement, for example, names a procedure to be used when the end of the file is detected.

Arithmetic Expressions

Arithmetic statements are a subset of imperative statements and are used to perform arithmetic operations. They are introduced by the verbs:

ADD

COMPUTE

DIVIDE

MULTIPLY

SUBTRACT

All arithmetic operations are performed on synchronized data whose usage is computational. If the operands are not already in this format, the compiler automatically generates any necessary data conversion routines.

COMPUTE statements can include arithmetic expressions as operands. An arithmetic expression is defined as one of the following:

- an identifier of a numeric elementary item
- a numeric literal
- a combination of such identifiers and literals separated by arithmetic operators
- two arithmetic expressions separated by an arithmetic operator
- an arithmetic expression enclosed in parentheses

The five binary arithmetic operators (that is, those with two operands) that can be used in arithmetic expressions are:

+ (addition)

- (subtraction)

* (multiplication)

/ (division)

** (exponentiation)

Binary arithmetic operators must be preceded and followed by a space.

An arithmetic expression can be preceded by either of the two unary operators:

+ indicates multiplication by +1

- indicates multiplication by -1

Spaces between the unary operator and the expression are not allowed.

Identifiers and literals appearing in an arithmetic expression must represent either numeric elementary items or numeric literals on which arithmetic can be performed. The chart below illustrates the permissible combinations of identifiers, literals, and arithmetic operators.

Second Symbol First Symbol	Identifier or Literal	* / ** + -	Unary ±	()
Identifier or Literal	NO	YES	NO	NO	YES
* / ** + -	YES	NO	YES	YES	NO
Unary ±	YES	NO	NO	YES	NO
(YES	NO	YES	YES	NO
)	NO	YES	NO	NO	YES

YES indicates a permissible pair
NO indicates an invalid pair

Arithmetic Expressions are evaluated as follows:

- Expressions enclosed in parentheses are evaluated first. When there are nested parentheses, evaluation starts with the least inclusive expression, proceeding to the most inclusive.
- When there are no parentheses, or when parenthetical expressions are mutually exclusive, operators are evaluated in the order listed below.

If multiple operators of the same type appear in the expression (exponentiation appears more than once, for example), all operators of the same type are evaluated from left to right.

1. Unary + and unary -
2. ** (exponentiation)
3. * and / (multiplication and division)
4. + and - (addition and subtraction)

Arithmetic Expressions

Arithmetic expressions may begin only with a left parenthesis, a + or - symbol, or with an identifier or literal. They may end only with a right parenthesis, an identifier, or a literal. If parentheses are used, they must occur in pairs. Examples of legal arithmetic expressions:

-598.95

$X * Y - (M/2 + (L - Y))$

$(-(X/Y) + M**2) * 100$

An intermediate or final result of an expression cannot have an absolute value smaller than $\pm 10^{-28}$ or larger than $\pm 10^{28}$.

Conditions

A conditional statement causes the object program to choose between alternate paths of control depending on the results of a test or comparison. Assume, for example, that in a credit billing system any account with a balance of \$1,000 or more is given special handling. The following simple conditional statement alters the logical flow of the program for such accounts:

```
IF BALANCE IS GREATER THAN 999.99 GO TO LARGE-BALANCE-ROUTINE;  
ELSE NEXT SENTENCE.
```

In the example above, the first operand BALANCE is called the *condition subject*; the second operand, the literal 999.99, is called the *condition object*. The expression IS GREATER THAN is called a *relational operator*. Notice that the conditional statement contains the imperative statement GO TO LARGE-BALANCE-ROUTINE.

When the conditional statement above is executed, the system compares the contents of BALANCE to 999.99. If BALANCE is greater than 999.99, control transfers to the procedure named LARGE-BALANCE-ROUTINE. If BALANCE is less than or equal to 999.99, control passes to the next sentence.

Simple conditions are those in which truth or falsity depends on one condition only. Simple conditions fall into the following classifications (each of these classifications is explained on the following pages):

- relation condition
- class condition
- condition-name condition
- sign condition
- NOT condition

Simple conditions can be combined by the logical operators AND and OR to form compound conditions.

OPERATOR	MEANING
OR	Logical inclusive OR (the compound condition is true if either or both simple conditions are true)
AND	Logical conjunction AND (the compound condition is true only when both simple conditions are true)

Conditions

The following is an example of two simple conditions joined by the logical operator AND:

```
IF BALANCE IS GREATER THAN 999.99 AND LESS THAN 1500.00 GO TO  
LARGE-BALANCE-ROUTINE-1; ELSE NEXT SENTENCE.
```

In addition, a relational operator may contain an optional NOT, indicating logical negation of the condition. For example, the statement above may also be written as follows:

```
IF BALANCE IS GREATER THAN 999.99 AND NOT GREATER THAN 1499.99  
GO TO LARGE-BALANCE-ROUTINE-1; ELSE NEXT SENTENCE.
```

COMPARISON OF OPERANDS

The comparison of operands varies depending on whether the operands are defined as numeric or nonnumeric.

Numeric Operands

Comparison is made with respect to the algebraic value of the operands. The length of the operands and the USAGE assigned to the operands is not significant. When unequal length operands are compared, the effect is the same as though the shorter operand were extended with enough leading zeros to make it equal in length to the longer operand.

Signed and unsigned fields may be compared. In this case, the unsigned field is assumed to be positive.

Zero is considered a unique value regardless of the sign.

EXAMPLES:

+0 IS EQUAL TO -0
1 IS EQUAL TO 00001.0000
+1 IS GREATER THAN -1
1 IS GREATER THAN .9

Nonnumeric Operands

Comparison between nonnumeric operands or between numeric and nonnumeric operands is based on the HP 3000 collating sequence.

Numeric and nonnumeric operands may be compared only when their USAGE is the same, either implicitly or explicitly.

Conditions (Comparison of Operands)

Pairs of characters in corresponding positions of the two operands are compared from the high-order end through the low-order end until a pair of unequal characters is encountered. If one operand is shorter than the other, the effect is the same as though the shorter operand were extended to the right by a number of spaces sufficient to make it equal in length to the longer operand.

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

The first pair of unequal characters encountered are compared to determine their relative positions in the collating sequence. The operand that contains the character higher in the collating sequence has the greater relative value.

EXAMPLES:

In the examples, the Δ character represents a space.

TICK	IS LESS THAN	TOCK
A1 $\Delta \Delta$	IS LESS THAN	B1
A1	IS GREATER THAN	91 $\Delta \Delta \Delta \Delta \Delta$
WILLIAM	IS LESS THAN	WILLIAMS
BEAD	IS LESS THAN	BEAM
TOM	IS EQUAL TO	TOM $\Delta \Delta \Delta \Delta \Delta \Delta \Delta$

COMPARISON OF INDEX NAMES OR INDEX DATA ITEMS. Operands that are index-names or index data items may be compared with each other or with data items or literals with the following stipulations:

1. Comparison of two index-names has the effect of a comparison of their occurrence numbers.
2. The occurrence number of an index-name is used in a comparison of an index-name and a data item or literal.
3. Two index data items or an index data item and an index-name are compared using their actual values.

PERMISSIBLE RELATIONAL COMPARISONS

Second Operand (Object) / First Operand (Subject)	Group	Alphabetic	Alphanumeric	Alphanumeric Edited	Numeric Edited	Figurative Constant	Non-numeric Literal	Zero	Numeric Literal	External Decimal	Binary	Internal Decimal	Index-Name	Index-Data-Item
		}				}		}						
Group	X		X			X		X		X	X	X	Δ	Δ
Alphabetic Alphanumeric Alphanumeric Edited Numeric Edited	X		X			X		X		X	X ¹	X ¹	Δ	Δ
Figurative Constant Non-numeric Literal	X		X			Δ		Δ		X	X ¹	X ¹	Δ	Δ
Zero Numeric Literal	X		X			Δ		Δ		9	9	9	9 ¹	Δ
External Decimal	X		X			X		9		9	9	9	9 ¹	Δ
Binary	X		X ¹			X ¹		9		9	9	9	9 ¹	Δ
Internal Decimal	X		X ¹			X ¹		9		9	9	9	9 ¹	Δ
Index-Name	Δ		Δ			Δ		9 ¹		9 ¹	9 ¹	9 ¹	9 ²	9 ³
Index-Data-Item	Δ		Δ			Δ		Δ		Δ	Δ	Δ	9 ³	9 ³

Δ = prohibited.

X = a non-numeric comparison.

X¹ = If either operand is an arithmetic expression, then comparison is illegal; otherwise a warning diagnostic message is issued, but a non-numeric comparison code is generated (be careful with right-justified data).

9 = a numeric comparison.

9¹ = comparison with index-name valid only if numeric item is integer.

9² = comparison of two index-names.

9³ = comparison with index-data-item.

Conditions (Evaluation of Conditions)

Evaluation of Conditions

Conditions may be enclosed within parentheses. Conditions within parentheses are evaluated first beginning with the least inclusive set and proceeding to the most inclusive set. Within parentheses, or when parentheses are not used or are at the same level of inclusiveness, the following order of evaluation is used:

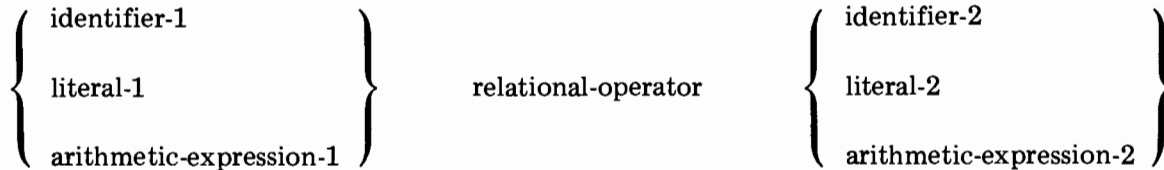
1. arithmetic expressions
2. all relational operators
3. NOT
4. AND
5. OR

The following table illustrates how conditions and logical operators can be combined. YES indicates that a combination is permissible; NO indicates it is not. The asterisk indicates that the symbol pair is permissible only if the condition is not preceded by NOT.

Second Symbol First Symbol	Condition	OR	AND	NOT	()
Condition	NO	YES	YES	NO	NO	YES
OR	YES	NO	NO	YES	YES	NO
AND	YES	NO	NO	YES	YES	NO
NOT	YES*	NO	NO	NO	YES	NO
(YES	NO	NO	YES	YES	NO
)	NO	YES	YES	NO	NO	YES

Two operands are compared in a relation condition. The operands can be an identifier, a literal, or an arithmetic expression.

FORMAT



The relational operator specifies the type of comparison to be made, and must be preceded and followed by a space. Relational operators are:

Operator	Meaning
IS [NOT] <u>GREATER THAN</u>	Greater than or not greater than
IS [NOT] >	
IS [NOT] <u>LESS THAN</u>	Less than or not less than
IS [NOT] <	
IS [NOT] <u>EQUAL TO</u>	Equal to or not equal to
IS [NOT] =	

EXAMPLE:

IF LAST-NAME IS EQUAL TO EMPLOYEE GO TO PAYROUTINE-3.

The condition in this statement compares the identifier LAST-NAME with the identifier EMPLOYEE. If the condition is true (the identifiers have the same value), the next statement GO TO PAYROUTINE-3 is executed. If the condition is false (the values of the identifiers are not equal), GO TO PAYROUTINE-3 is not executed and control passes to the statement following this conditional sentence.

Conditions (Relation Condition)

ABBREVIATING RELATION CONDITIONS. When relation conditions are written in a consecutive sequence, any of the conditions except the first may be abbreviated in the following manner:

- the subject of the relation condition may be omitted
- the subject and the relational operator of the relation condition may be omitted.

The effect of such abbreviation is as though the omitted subject were replaced by the last preceding stated subject, *or* the omitted relational operator were replaced by the last preceding stated relational operator.

When any portion of an abbreviated combined relation condition is enclosed in parentheses, the first relation condition within parentheses must not be abbreviated.

EXAMPLES:

The following example illustrates abbreviation of a condition by omitting the subject.

IF BALANCE IS GREATER THAN 999.99 AND BALANCE IS LESS THAN 1500.00
GO TO LARGE-BALANCE-ROUTINE.

IF BALANCE IS GREATER THAN 999.99 AND LESS THAN 1500.00 GO TO
LARGE-BALANCE-ROUTINE.

The following example, which illustrates abbreviation of a condition by omitting both the subject and relational operator, is from a program which calculates interest on accounts. Interest is not paid on accounts of less than a dollar nor on those with a negative balance.

IF BALANCE IS GREATER THAN ZERO AND BALANCE IS GREATER THAN 1.00
GO TO INTEREST-ROUTINE.

IF BALANCE IS GREATER THAN 1.00 GO TO INTEREST-ROUTINE.

Conditions (Class Conditions)

A class condition determines whether an operand is numeric or alphabetic.

FORMAT

$$\text{identifier IS [NOT] } \left\{ \begin{array}{l} \underline{\text{NUMERIC}} \\ \underline{\text{ALPHABETIC}} \end{array} \right\}$$

The USAGE of the operand being tested must be DISPLAY (see USAGE clause in Section 4).

The NUMERIC test should not be used with an item that has been described as alphabetic. (If the program contains such a test, appropriate object code is generated, but a diagnostic message is also generated.) If the PICTURE in the record description of the item being tested does not contain an operational sign, the item is considered numeric only if the contents are numeric and an operational sign is not present.

The ALPHABETIC test should not be used with an item that has been described as numeric. (If the program contains such a test, appropriate object code is generated, but a diagnostic message is also generated.) The item being tested is considered alphabetic only if the contents consist of any combination of the alphabetic characters A through Z and the space.

EXAMPLE

IF SUM-1 IS NOT NUMERIC GO TO ERR-5 ELSE ADD SUM-1 TO SUM-2.

If SUM-1 is found to be numeric, it is added to SUM-2. If it is not numeric, the program branches to a procedure called ERR-5.

Conditions (Condition-Name Condition)

Condition-Name Condition

A condition-name condition tests a conditional variable to determine whether its value is equal to one of the values associated with a condition-name.

FORMAT

condition-name

EXAMPLE:

```
02 PAYCODE          PIC 9.  
88 PAYMENT          VALUE 1.  
88 CREDIT           VALUE 2.  
88 LAYAWAY          VALUE 3.
```

PAYCODE is the conditional variable. PAYMENT, CREDIT, and LAYAWAY are the condition-names. IF PAYMENT is equivalent to IF PAYCODE = 1; IF CREDIT is equivalent to IF PAYCODE = 2; IF LAYAWAY is equivalent to IF PAYCODE = 3. Thus, the statement

```
IF CREDIT; GO TO INTEREST-ROUTINE.
```

would result in the program branching to a procedure called INTEREST-ROUTINE if PAYCODE contains the value 2.

Sign Condition

A sign condition determines whether the algebraic value of a numeric operand is less than, greater than, or equal to zero.

FORMAT

$$\left\{ \begin{array}{l} \text{identifier} \\ \text{arithmetic expression} \end{array} \right\} \text{ IS [NOT] } \left\{ \begin{array}{l} \underline{\text{POSITIVE}} \\ \underline{\text{NEGATIVE}} \\ \underline{\text{ZERO}} \end{array} \right\}$$

The operand is positive if its value is greater than zero, negative if its value is less than zero, and zero if its value is equal to zero.

EXAMPLE:

IF ACCOUNT-BALANCE IS NEGATIVE GO TO CREDIT-ROUTINE.

If the field ACCOUNT-BALANCE is negative — indicating that the account has been overpaid — control is transferred to CREDIT-ROUTINE.

Conditions (NOT Condition)

NOT Condition

The NOT condition is the Boolean NOT. It reverses all logical and relational operators in the condition it precedes. When it precedes a condition, the NOT may be interpreted as “not true that.”

FORMAT

NOT condition

When a compound condition enclosed in parentheses is preceded by the NOT, the expression is treated as a single condition and the entire condition is affected by the NOT.

When a compound condition or a compound abbreviated condition not enclosed in parentheses is preceded by a NOT, the NOT is distributed across the conditions.

The Boolean NOT must not be used when the relational operator contains a NOT; i.e., the word NOT may appear only once in a given condition. Thus the expression IF NOT A IS NOT GREATER THAN B is illegal and must not be used.

EXAMPLES:

IF NOT A = B . . .

is the same as IF A IS NOT EQUAL TO B

IF NOT (X = 5 AND Y = 1) . . .

is the same as IF X IS NOT EQUAL TO 5 OR Y IS NOT EQUAL TO 1

IF NOT X = 5 AND Y = 1 . . .

is the same as IF X IS NOT EQUAL TO 5 AND Y IS EQUAL TO 1

INTERPROGRAM COMMUNICATION

This compiler provides special facilities in both the Data Division and the Procedure Division to permit interprogram communication. This facility allows one program (the main or calling program) to call another (the subprogram) and also allows the two programs to share common data.

In the Data Division, the Linkage Section allows the programmer to define those data items which are to be picked up from some other program. Data definitions in this section do not reserve any storage area since the data is already stored in some other program.

In the Procedure Division, the special statements **CALL**, **ENTRY**, **GOBACK**, and **EXIT PROGRAM** permit communication between object programs. For additional information about these statements, see the individual statement descriptions in this section.

COMPILE-TIME OPTIONS

The options discussed below require knowledge of the MPE Operating System. For additional information, please refer to *MPE Commands Reference Manual*.

As a compile-time option of the \$CONTROL command, the programmer may designate a subprogram as either non-DYNAMIC or DYNAMIC. Non-DYNAMIC subprograms have their data storage allocated statically using DB-relative addressing; DYNAMIC subprograms have their data storage allocated dynamically using Q-relative addressing.

Each time a DYNAMIC subprogram is called, it is loaded into memory in its initial state. By contrast, a non-DYNAMIC subprogram is loaded in its initial state only when called for the first time. Each subsequent time the subprogram is called, it is loaded as it was following the last exit from the subprogram. Thus the programmer may have to initialize data items, PERFORM statements, the TALLY register, and GO TO statements that have been altered.

COMMON OPTIONS IN PROCEDURAL STATEMENTS

Three options are common to several of the statements that appear in the Procedure Division. These are the ROUNDED option, the SIZE ERROR option, and the CORRESPONDING option. To avoid repetition, these options are explained only once, below. When one of these options appears in a format in the subsequent pages, refer to the following discussion.

ROUNDED Option

The ROUNDED option is used to prevent truncation from occurring in the result field for an arithmetic operation. Truncation will occur if the ROUNDED option has not been declared, and the number of digits to the right of the decimal point exceeds the number of positions provided for in the result item. Also, truncation and/or rounding may occur to the left of the implied decimal point if the scaling symbol P represents the low order digits of a result field. In such a case, rounding or truncation occurs relative to the rightmost integer for which storage is allocated.

FORMAT

[ROUNDED]

When ROUNDED has been declared, the absolute value of the least significant digit of the storable result is increased by one whenever the value of the most significant truncated digit is five or greater.

EXAMPLE:

In the following examples, the caret (^) represents the implied decimal point.

Actual Result	Picture for Result Field	ROUNDED Value as Stored
3^141592654	9V999	3^142
7650^	99PP	77
3^141592654	9V99	3^14

SIZE ERROR Option

SIZE ERROR Option

If the value of the result of an arithmetic expression exceeds the largest value that can be contained in the result item, a size error condition exists. Division by zero always results in a size error condition. A size error condition occurs only in the final results of an arithmetic operation and does not apply to intermediate results.

FORMAT

[ON SIZE ERROR imperative statement]

If the **SIZE ERROR** option is not specified and a size error condition occurs, the value of the result item is unpredictable. Values of result items for which no size error condition occurs are unaffected by size errors that occur for other result items during execution of the operation.

If the **SIZE ERROR** option is specified, and a size error condition occurs, the values of result items affected by the size errors are not altered. After the arithmetic operation is completed, the specified imperative-statement is executed.

When the **ROUNDED** option is specified, rounding takes place before checking for a size error condition.

When the **CORRESPONDING** option is specified for **ADD** and **SUBTRACT** statements along with the **SIZE ERROR** option, the imperative-statement is not executed until the whole set of operations has been completed.

EXAMPLE:

```
DIVIDE SAMPLE BY SCORE GIVING AVERAGE; ON SIZE  
ERROR PERFORM ERR-RTN-6.
```

If **SCORE** contains zeros when the above statement is executed, control passes to the procedure **ERR-RTN-6**. The contents of **AVERAGE** remain unchanged.

CORRESPONDING Option

The CORRESPONDING option can be specified in an ADD, SUBTRACT, or MOVE statement. When the CORRESPONDING option is specified for one of these statements, the effect is the same as though the statement were written once for each corresponding pair of items, as shown in the example on the following page.

FORMAT

$$\left\{ \begin{array}{l} \underline{\text{CORRESPONDING}} \\ \underline{\text{CORR}} \end{array} \right\}$$

A pair of data items, one from each group, correspond if:

- They have the same name and the same qualifiers up to but not including the name of the group item.
- In the case of a MOVE statement with the CORRESPONDING option, at least one of the data items must be an elementary item. In the case of an ADD or SUBTRACT statement with the CORRESPONDING option, all pairs of data items must be elementary items.
- Neither of the data items can have been described in a 66-, 77-, or 88- level entry.
- A data item cannot contain a REDEFINES, OCCURS, or USAGE IS INDEX clause, nor can any subordinate item contain these clauses. However, the group items named as operands may contain or be subordinate to items containing REDEFINES or OCCURS clauses.
- A data item subordinate to a group item containing a RENAMES clause is ignored.

CORRESPONDING can be abbreviated CORR.

CORRESPONDING Option

EXAMPLE:

For the purposes of this example, assume that two records have the following descriptions:

01 NEW-HIRE-INFO.	01 PERSONNEL-RECORD.
02 NAME ...	03 EMPLOYEE-NUMBER ...
02 PERSONAL-DATA.	03 NAME ...
03 AGE ...	03 OTHER-DATA.
03 SEX ...	05 SEX ...
03 WEIGHT ...	05 AGE ...
03 HEIGHT ...	05 NUMBER-DEPENDENTS ...
03 COLOR-EYES ...	05 HEIGHT ...
	05 WEIGHT ...

The statement `MOVE CORRESPONDING PERSONAL-DATA TO OTHER-DATA` is equivalent to four discreet `MOVE` statements, one each for the items `AGE`, `SEX`, `HEIGHT`, and `WEIGHT`.

The statement `MOVE CORRESPONDING NEW-HIRE-INFO TO PERSONNEL-RECORD` moves only the item `NAME` to `PERSONNEL-RECORD`. The items `AGE`, `SEX`, etc. are no longer corresponding because they no longer have the same qualification in both records. (`AGE IN PERSONAL-DATA` is not equivalent to `AGE IN OTHER-DATA`, even though the two items have the same hierarchical position in their respective records.)

SECTION VIII

Procedure Division Statements

This section discusses all procedural statements except for the Input-Output statements which are discussed in Section X. Statements within this section appear alphabetically.

ADD Statement

The ADD statement causes two or more numeric operands to be summed and the result to be stored.

Format 1

$$\underline{\text{ADD}} \left\{ \begin{array}{l} \text{identifier-1} \\ \text{literal-1} \end{array} \right\} \left[\begin{array}{l} \text{,identifier-2} \\ \text{,literal-2} \end{array} \right] \dots \underline{\text{TO}} \text{identifier-m} \left[\underline{\text{ROUNDED}} \right]$$
$$\left[\text{,identifier-n} \left[\underline{\text{ROUNDED}} \right] \right] \dots \left[\text{;ON SIZE ERROR imperative-statement} \right]$$

Format 2

$$\underline{\text{ADD}} \left\{ \begin{array}{l} \text{identifier-1} \\ \text{literal-1} \end{array} \right\}, \left\{ \begin{array}{l} \text{identifier-2} \\ \text{literal-2} \end{array} \right\} \left[\begin{array}{l} \text{,identifier-3} \\ \text{,literal-3} \end{array} \right] \dots$$
$$\underline{\text{GIVING}} \text{ identifier-m} \left[\underline{\text{ROUNDED}} \right]$$
$$\left[\text{;ON SIZE ERROR imperative statement} \right]$$

Format 3

$$\underline{\text{ADD}} \left\{ \begin{array}{l} \underline{\text{CORRESPONDING}} \\ \underline{\text{CORR}} \end{array} \right\} \text{identifier-1} \underline{\text{TO}} \text{identifier-2}$$
$$\left[\underline{\text{ROUNDED}} \right] \left[\text{;ON SIZE ERROR imperative statement} \right]$$

Literals included in an ADD statement must be numeric. In Formats 1 and 2, identifiers must refer to elementary numeric items with one exception: the identifier appearing to the right of the word GIVING in Format 2 can refer to a data item that contains editing symbols. However, if the data item does contain editing symbols, it cannot be used as a numeric operand in subsequent arithmetic operations.

The maximum size of each operand is eighteen decimal digits. The maximum size of the resulting sum, after decimal point alignment, is eighteen decimal digits, not including any editing symbols in the data item following the word GIVING.

The COBOL/3000 compiler ensures that enough places are carried so that no significant digits are lost during execution. Thus it is possible to calculate a number too large to be stored in eighteen digits.

The CORRESPONDING, ROUNDED, and SIZE ERROR options are explained at the end of Section VII.

When Format 1 of the ADD statement is used, all operands preceding the word TO are added together. The total is then added to and stored in each operand following the word TO.

EXAMPLE

ADD BONUS TO WAGES-50, WAGES-75, WAGES-79.

	Before Execution	After Execution
BONUS	100^	100^
WAGES-50	750^	850^
WAGES-75	1200^	1300^
WAGES-79	1200^	1300^

When Format 2 is used, all operands preceding the word GIVING are added together. The total is stored in the operand following the word GIVING.

EXAMPLE

ADD SUM-A, SUM-B, 1000 GIVING TOTAL.

	Before Execution	After Execution
SUM-A	4951^	4951^
SUM-B	794^	794^
TOTAL	97855^	6745^

When Format 3 is used, selected elementary data items in the group item defined by identifier-1 are added to and stored in the elementary items that have matching names in the group item defined by identifier-2 (see CORRESPONDING option).

EXAMPLE

ADD CORR REC-IN TO REC-MONTH; ON SIZE ERROR GO TO ROUT-6.

The corresponding elementary data items in the group item referred to by REC-IN are added to and stored in those corresponding data items in the group item referred to by REC-MONTH.

ALTER Statement

The ALTER statement is used to modify the destination of a GO TO statement.

FORMAT

```
ALTER procedure-name-1 TO [PROCEED TO] procedure-name-2  
      [,procedure-name-3 TO [PROCEED TO] procedure-name-4] . . .
```

Each procedure-name-1, procedure-name-3, etc., is the name of a paragraph that contains only one sentence consisting of a GO TO statement. A GO TO statement modified by an ALTER statement must not include a DEPENDING option.

Each procedure-name-2, procedure-name-4, etc., is the name of a paragraph or section in the Procedure Division. Upon execution of the ALTER statement, such paragraphs or sections replace the object procedure-name in the GO TO statement.

A GO TO statement in a section with a priority number equal to or greater than 50 must not be referred to by an ALTER statement in a section with a different priority number. (With this compiler, the ALTER statement is executed normally, but its effect is nullified since sections with a priority number equal to or greater than 50 are always brought into memory in their initial state.) All other uses of the ALTER statement are valid and are executed normally.

EXAMPLE

In the following example, a paragraph is to be executed only once, the first time through the program. The use of the ALTER statement provides a means for skipping over the routine in subsequent passes through the program.

```
FIRST-TIME-SWITCH.  
  GO TO FIRST-TIME-ROUTINE.  
FIRST-TIME-ROUTINE.  
  ALTER FIRST-TIME-SWITCH TO PROCEED TO UPDATE-ROUTINE.  
  .  
  (procedural statements)  
  .  
UPDATE-ROUTINE.  
  .  
  (procedural statements)  
  .
```

The first time this coding is executed, control passes sequentially from FIRST-TIME-SWITCH to FIRST-TIME-ROUTINE to UPDATE-ROUTINE. Subsequent executions of this code branch from FIRST-TIME-SWITCH to UPDATE-ROUTINE, skipping over the FIRST-TIME-ROUTINE.

CALL Statement

The CALL statement permits communication between a COBOL object program and one or more COBOL subprograms or other language subprograms.

FORMAT

CALL literal-1 [USING identifier-1 [identifier-2] . . .]

The program which contains the CALL statement is the calling program; the subprogram invoked by the CALL is the called subprogram. A called subprogram may itself contain a CALL statement. However, a called subprogram must never attempt to call the program that originally called it. Thus, if program A calls subprogram B, then subprogram B cannot call program A. Nor can a called subprogram indirectly attempt to call the calling program. Thus, if A calls B, and B calls C, then C cannot call either A or B. To state this rule more simply, the CALL statement must never request two versions of the same program to be in memory at the same time.

Literal-1 must be a nonnumeric literal and must be either the name of the subprogram that is being called or the name of an entry point in the called subprogram. Literal-1 must conform to the rules for the formation of a program-name as explained in Section III "Identification Division." The first 15 characters of literal-1 are used to make the correspondence between the called and the calling program.

When the called subprogram is to be entered at the beginning of its Procedure Division, literal-1 must specify the program-name (from the PROGRAM-ID paragraph) of the called subprogram.

When the called subprogram is to be entered at entry points other than the beginning of the Procedure Division, these alternate entry points are identified by an ENTRY statement. Notice that the ENTRY statement appears in the called *subprogram*, not the calling program. Also, the ENTRY statement must include a USING option that corresponds to the USING option within the CALL statement. (The USING option is explained below.) When the subprogram is entered via an entry point, literal-1 in the CALL statement names the entry point rather than the subprogram. However, for the purposes of the CALL statement, entry point names must conform to the rules for program-names. Notice also that literal-1 is a nonnumeric literal and must therefore be enclosed in quotation marks.

CALL Statement

USING OPTION

The USING option makes up to 60 data items (or FD file names) defined in the calling program available to a called subprogram.

In the calling program, the USING option appears only within the CALL statement. Within the called subprogram, the USING option may appear in either of two places: When literal-1 in the CALL statement specifies the program-name of the called subprogram, the USING option appears in the Procedure Division header of the subprogram; When literal-1 in the CALL statement specifies an entry point, the USING option appears in an ENTRY statement within the subprogram. Thus there are three formats for the USING option:

FORMATS

Within a Calling Program

CALL literal-1 [USING identifier-1 [identifier-2] . . .]

Within a Called Subprogram

Option 1

ENTRY literal-1 [USING identifier-1 [identifier-2] . . .]

Option 2

PROCEDURE DIVISION [USING identifier-1 [identifier-2] . . .]

When the USING option is specified in the CALL statement, it must also appear on either the Procedure Division header of the called subprogram or in an ENTRY statement within the called subprogram.

Within the CALL statement, each identifier following the keyword USING must be defined as a data item in the File Section, Working-Storage Section, or Linkage Section. If the called subprogram is written in a language other than COBOL, the operands of the USING option may include file-names as well as data items. If the subprogram is an SPL procedure, the procedure parameter corresponding to the file name must be declared as type INTEGER or LOGICAL. It is passed by value. However, when a file-name is an operand of a USING option, the user must open the file in the calling program prior to execution of the CALL statement.

Within the subprogram, each identifier following the keyword USING must be defined as a data item in the subprogram's Linkage Section and must have a level number of 01 or 77. Data definitions within the Linkage Section assign data-names and data descriptions, but do not reserve any storage for the data. The compiler assumes that storage for items defined in the Linkage Section has already been assigned in another program.

The compiler assumes that identifier-1 in the USING option of the CALL statement corresponds to identifier-1 in the USING option associated with the called subprogram, and that identifier-2 corresponds to identifier-2, and so on. The compiler establishes this correspondence simply by the position of the pairs of identifiers; the names assigned to the identifiers are ignored. Therefore, the number of identifiers listed for the CALL statement must be identical to the number of identifiers given in the USING option of the called subprogram. The order of these operands is critical.

The compiler always assigns level 01 and 77 entries on a word boundary. Identifiers named in the USING option of the CALL statement do not have to be level 01 or 77 entries; they must, however, be aligned on a word boundary. This is especially important when the identifier is subscripted or indexed, as the compiler will not be able to test in these cases.

OPERATION OF THE CALL STATEMENT

When the CALL statement is executed, the system locates and loads the requested subprogram, establishes addressability for the data areas listed in the USING option, and then transfers control to the subprogram at the point specified by literal-1 in the CALL statement. The called subprogram continues to execute until a GOBACK, EXIT PROGRAM, or STOP RUN statement is encountered. When a GOBACK or EXIT PROGRAM statement is encountered during the execution of a subprogram, control is returned to the statement immediately following the CALL statement in the calling program. (This is similar to the return of control following an exit from a PERFORM procedure.)

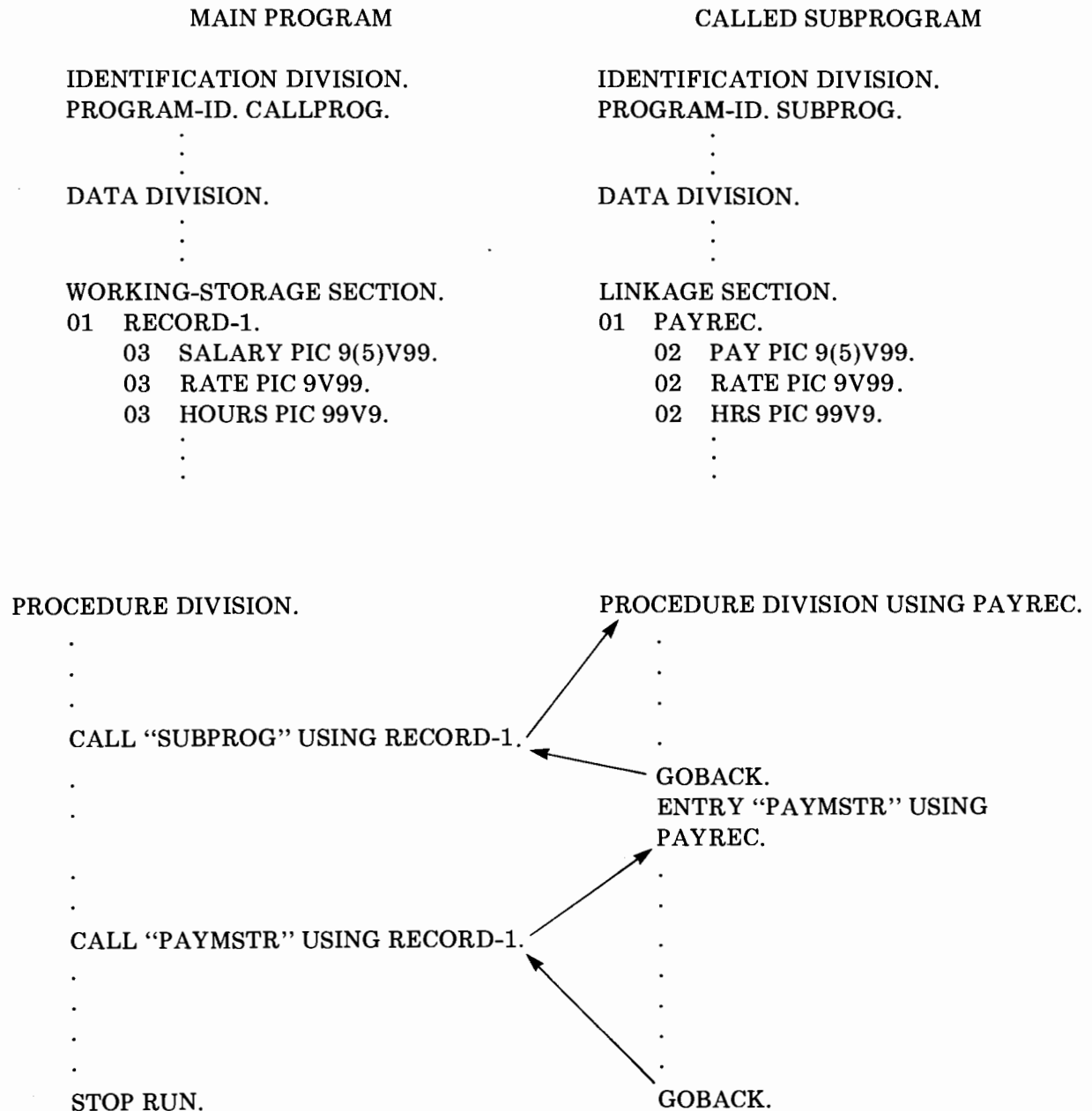
If a STOP RUN statement is encountered during the execution of a subprogram, the STOP RUN applies to the called subprogram and to its calling program or programs. Control returns to the MPE Operating System.

CALL Statement

EXAMPLE

In the following programs—one a calling program, and the other a called subprogram—notice that RECORD-1 and PAYREC have the same descriptions. RECORD-1 is the only item mentioned in the USING option of the CALL statement; similarly, PAYREC is the only item mentioned in the USING option of the Procedure Division header and the ENTRY statement in the called subprogram, and PAYREC is in the Linkage Section of the called subprogram. Therefore, the compiler can equate the items for interprogram communication.

Also notice that the subprogram is called twice, each time with a different entry point. The first entry point is at the beginning of the Procedure Division and is indicated by the program-name. The second entry point (PAYMSTR) is indicated by the ENTRY statement.



COMPUTE Statement

The purpose of the COMPUTE statement is to assign to a data item the value of a numeric data item, a numeric literal, or an arithmetic expression. The COMPUTE statement allows for the combination of arithmetic operations without the restrictions imposed by the ADD, SUBTRACT, MULTIPLY, and DIVIDE statements.

FORMAT

$$\underline{\text{COMPUTE}} \text{ identifier-1 } [\underline{\text{ROUNDED}}] = \left\{ \begin{array}{l} \text{identifier-2} \\ \text{literal-1} \\ \text{arithmetic} \end{array} \right\}$$


[;ON SIZE ERROR imperative statement]

Identifier-1 may contain editing symbols, but identifier-2 must refer to an elementary numeric item. Literal-1 must be numeric. The arithmetic expression can consist of any valid combination of identifiers, numeric literals, and arithmetic operators, parenthesized as required.

The maximum size of each operand is eighteen decimal digits.

The ROUNDED and SIZE ERROR options are discussed in the Procedure Division, Section VII.

EXAMPLE

COMPUTE TOTAL ROUNDED = (SUB * TAX) + SUB.

	Before Execution	After Execution
SUB	89^95	89^95
TAX	^05	^05
TOTAL	76^53	94^45

Note: COBOL uses packed arithmetic to compute the value of arithmetic expressions specified in a COMPUTE statement. The precision of intermediate results is determined by the operand with the greatest number of digits to the right of the decimal point. If rounding is specified, one additional digit is carried.

COPY Statement

The COPY statement provides the facility for copying prewritten text from a copy library into the source program. Also, this statement has the capability to substitute specified words as the text is being copied.

GENERAL FORMAT

COPY library-name

$$\left[\text{REPLACING word-1 BY } \left\{ \begin{array}{l} \text{word-2} \\ \text{identifier-1} \\ \text{literal-1} \end{array} \right\} \right]$$
$$\left[, \text{word-3 BY } \left\{ \begin{array}{l} \text{word-4} \\ \text{identifier-2} \\ \text{literal-2} \end{array} \right\} \right]$$

The format shown above is a general format for the COPY statement itself; however, the COPY statement is always used in conjunction with other COBOL elements such as a level number or a paragraph-name. See the specific COPY formats in this description of the COPY statement.

A word in this format must conform to the rules for the formation of a word, and must be one of the following:

- data-name
- procedure-name
- condition-name
- mnemonic-name
- file-name

THE LIBRARY

The copy library must be a file containing COBOL source statements. This file is maintained by the user via the MPE EDITOR. The user is responsible for both the creation and maintenance of this file.

The copy file must contain text that is syntactically correct when copied into a program. In the Procedure Division, for example, the COPY statement must follow either a paragraph-name or a section heading; therefore, the text in the copy file must not begin with that paragraph-name or section heading. To do so would cause duplicate names when the text is copied. Syntactically, a paragraph terminates when the next paragraph-name is found, and a section terminates when the next section heading is found; therefore, if the COPY statement is associated with a paragraph-name, the copy-library text should contain only the sentences for that one paragraph. A section, however, may contain any number of paragraphs.

Note: Syntactical correctness requires special attention concerning the use of the period character, as explained under "Specific Formats" on the following pages.

Text within a copy-library must not contain any COPY statements.

Each line of source coding entered into the copy-library must contain an identification field in columns 73 through 80. This identification field represents the library-name to which the source record belongs. (This is the library-name entered in the COPY statement.) Therefore, the library-name can be no more than eight characters in length. Library-names less than eight characters must be left-justified in the identification field.

The copy file may be a user library, or it may be the system's COPYLIB. When a user library is used, its name must be passed to the COBOL compiler via an MPE :FILE command; for example, :FILE COPYLIB=USERLIB. When no library name is given to the compiler, the system assumes that COPYLIB.group.account. is to be used.

Function of the COPY Statement

When the first COPY statement is encountered, the system automatically opens the copy-file. The copy-file is automatically closed when the compilation process is completed.

Each time a COPY statement is encountered, the system searches the appropriate copy-library for text whose identification field (entered as columns 73 through 80 of the source statements) matches the library-name specified in the COPY statement. If the text is found, all statements with the specified library-name are copied into the source program. These statements replace the COPY statement. The library-name of the copied statements is printed to the left of the statement on the source listing to identify the copied text. The compiler makes no distinction between copied and original text; therefore, the effect of the compilation is as though the copied text were entered as a part of the original source program.

If the requested library-name is not located in the copy-library, a diagnostic message is issued, and the compilation continues despite the unsuccessful copy attempt.

If the REPLACING phrase is used, each occurrence of word-1, word-3, etc., in the text being copied is replaced by the word, identifier, or literal specified in the corresponding REPLACING phrase. Use of the REPLACING phrase does not alter the text as it appears in the copy-file. The REPLACING edit function is performed internally and, although it affects the code that is generated, the compiled listing only displays the original text from the source and copy file.

SPECIFIC FORMATS

As can be seen from the following formats, the COPY statement is used in conjunction with other coding, such as a paragraph-name, FD level indicator, etc. The text within the copy-library should conform to the rules for each format. In the Data Division, for example, an 01-level record description may not contain another 01-level entry. An FD level entry may contain a number of 01-level entries; however, an FD level may not contain another FD statement.

COPY Statement

Because the COPY statement is used in conjunction with other coding, the use of the period as a sentence terminator requires special attention. In certain cases, replacement of the COPY statement by the copied text may delete a period required by the syntax. The programmer replaces the period by making the first record in the copy-file a source line which contains a period anywhere within Area B of the source line. This period cannot be used if the specific format for the COPY statement indicates a period immediately before the COPY verb (for example, *FILE-CONTROL*.copy-statement.). If the period follows the COPY verb (for example, *FD* file-name copy-statement.), the programmer must supply a period as described above.

Identification Division

The COPY statement is not allowed within the Identification Division.

Environment Division

The SOURCE COMPUTER and OBJECT COMPUTER paragraphs are treated as comments by this compiler; any COPY statements in these paragraphs will be printed on the compiler listing, but will have no other effect.

<u>SOURCE COMPUTER.</u>	copy statement.
<u>OBJECT COMPUTER.</u>	copy-statement.
<u>SPECIAL-NAMES.</u>	copy-statement.
<u>FILE-CONTROL.</u>	copy-statement.
<u>I-O-CONTROL.</u>	copy-statement.

Data Division—File Section

<u>FD</u>	file-name copy-statement.
<u>SD</u>	sort-file-name copy-statement.
01	data-name copy-statement.

Data Division—Working-Storage and Linkage Sections

01	data-name copy-statement.
----	---------------------------

Procedure Division

{	paragraph-name.	}	copy-statement.
{	section-name <u>SECTION</u> [priority-number].	}	

DIVIDE Statement

The **DIVIDE** statement divides one numeric data item into another and sets the value of a data item equal to the result.

Format 1

DIVIDE { identifier-1 }
 { literal-1 } INTO identifier-2 [ROUNDED]

 [;ON SIZE ERROR imperative-statement]

Format 2

DIVIDE { identifier-1 }
 { literal-1 } INTO { identifier-2 }
 { literal-2 }

 GIVING identifier-3 [ROUNDED]

 [;ON SIZE ERROR imperative-statement]

Format 3

DIVIDE { identifier-1 }
 { literal-1 } BY { identifier-2 }
 { literal-2 }

 GIVING identifier-3 [ROUNDED]

 [; ON SIZE ERROR imperative-statement]

Format 4

DIVIDE { identifier-1 }
 { literal-1 } INTO { identifier-2 }
 { literal-2 }

 GIVING identifier-3 [ROUNDED] REMAINDER identifier-4

 [; ON SIZE ERROR imperative-statement]

DIVIDE Statement

Format 5

$$\underline{\text{DIVIDE}} \left\{ \begin{array}{l} \text{identifier-1} \\ \text{literal-1} \end{array} \right\} \underline{\text{BY}} \left\{ \begin{array}{l} \text{identifier-2} \\ \text{literal-2} \end{array} \right\}$$

GIVING identifier-3 [ROUNDED] REMAINDER identifier-4

[; ON SIZE ERROR imperative-statement]

Each identifier, except the one associated with the word GIVING, must refer to a numeric elementary item. The identifier associated with the word GIVING may contain editing symbols. All literals must be numeric. The maximum size of each operand is eighteen digits.

The ROUNDED and SIZE ERROR options are discussed in Section VII.

When Format 1 is used, the value of identifier-1 or literal-1 is divided into the value of identifier-2. The value of identifier-2 (the dividend) is replaced by the quotient.

EXAMPLE

DIVIDE NUMBER INTO AVERAGE ROUNDED.

	Before Execution	After Execution
NUMBER	22	22
AVERAGE	9567.28	434.88

When Format 2 is used, the value of identifier-1 or literal-1 is divided into identifier-2 or literal-2. The result is stored in identifier-3.

EXAMPLE

DIVIDE NUMBER-OF-STUDENTS INTO TOTAL-TEST-SCORE GIVING AVERAGE.

	Before Execution	After Execution
NUMBER-OF-STUDENTS	50	50
TOTAL-TEST-SCORE	3710	3710
AVERAGE	65.8	74.2

When Format 3 is used, the value of identifier-1 or literal-1 is divided by the value of identifier-2 or literal-2. The result is stored in identifier-3.

EXAMPLE

DIVIDE YEARLY-INTR BY 12 GIVING RATE.

	Before Execution	After Execution
YEARLY-INTR	3600	3600
RATE	265	300

When Formats 4 and 5 are used, a remainder is produced in addition to the quotient. The remainder is the result of subtracting the product of the quotient and the divisor from the dividend.

EXAMPLE

DIVIDE FIELD-1 BY 33.3 GIVING FIELD-2 REMAINDER FIELD-3

If the value of FIELD-1 is 952.96, the value 28.6 is stored in FIELD-2, and .58 is stored in FIELD-3.

ENTER Statement

ENTER is a compiler-directing statement that permits the use of a language other than COBOL in the program.

FORMAT

ENTER language-name [routine-name]

This compiler treats the ENTER statement as a comment.

ENTRY Statement

The ENTRY statement establishes an entry point in a COBOL subprogram.

FORMAT

ENTRY literal-1 [USING identifier-1 [identifier-2] . . .]

Control is transferred to the entry point by a CALL statement in the invoking program.

Literal-1 is a nonnumeric literal and must therefore be enclosed in quotes. Literal-1 must not be the name of the called subprogram, but it must be formed according to the rules for program-names as explained for the PROGRAM-ID paragraph in the Identification Division, Section III.

Literal-1 must not be the name of any other entry point or program name in the run unit.

Once invoked, a called program is entered at that ENTRY statement whose operand literal-1 is the same as the literal-1 specified in the CALL statement.

USING Option

The USING option is fully explained under the CALL statement in this section.

EXAMINE Statement

The EXAMINE statement counts the occurrences of a given character within a data item. It is also used to replace those character occurrences with another character.

FORMAT

$$\begin{array}{c} \text{EXAMINE identifier} \\ \left\{ \begin{array}{l} \text{TALLYING} \left\{ \begin{array}{l} \text{UNTIL FIRST} \\ \text{ALL} \\ \text{LEADING} \end{array} \right\} \text{literal-1} \quad [\text{REPLACING BY literal-2}] \\ \\ \text{REPLACING} \left\{ \begin{array}{l} \text{ALL} \\ \text{LEADING} \\ [\text{UNTIL}] \text{FIRST} \end{array} \right\} \text{literal-3} \quad \text{BY literal-4} \end{array} \right\} \end{array}$$

The usage for the *identifier* must be displayed (explicitly or implicitly). Each literal must consist of a single character whose class is consistent with that of the identifier. Any of the literals can be any figurative constant except ALL.

An EXAMINE statement operates as follows:

1. Examination of a nonnumeric data item specified by *identifier* begins with the leftmost character, proceeding to the right. Each character in the data item is examined in turn.
2. A numeric data item referred to in the EXAMINE statement must consist of numeric characters. It may have an operational sign. Examination begins with the leftmost character (excluding the sign, if any) and proceeds to the right. Each character but the sign is examined in turn. The sign is ignored by the EXAMINE statement, regardless of its physical location.
3. When the TALLYING option is used, a count is placed in the TALLY register, destroying its previous contents. (The TALLY register is discussed under Language Elements in Section I.) The count is an integer and represents:

Option	Count
ALL	Total occurrences of literal-1
LEADING	Total occurrences of literal-1 prior to encountering a character other than literal-1
UNTIL FIRST	Total occurrences of characters other than literal-1 encountered before first occurrence of literal-1

4. When either of the REPLACING options are used, the following substitutions take place. References to literal-1 and literal-2 apply equally to literal-3 and literal-4, respectively.

Option	Result
ALL	Literal-2 replaces each occurrence of literal-1
LEADING	Each occurrence of literal-1 is replaced by literal-2 until the first occurrence of a character other than literal-1 or the rightmost margin of the data item is encountered
UNTIL FIRST	Literal-2 replaces all characters prior to the first occurrence of literal-1 or until the rightmost item of the data item is encountered
FIRST	Only the first occurrence of literal-1 is replaced by literal-2.

EXAMPLES

- EXAMINE RESPONSE TALLYING ALL 1.

If RESPONSE contains the value 41311, the value in the TALLY register is 3 after execution of this statement.

- EXAMINE RESPONSE TALLYING ALL 1 REPLACING BY 2.

The value of TALLY is 3; the value of RESPONSE is now 42322.

- EXAMINE HEADER REPLACING LEADING 0 BY SPACES.

If the value of HEADER is 000390460, it will be $\Delta\Delta\Delta 390460$ after execution of this statement.

Note: In this example HEADER must not be numeric as SPACES is not consistent with a numeric item.

EXIT Statement

The EXIT statement provides an end point common to all branches in a procedure or series of procedures.

FORMAT

EXIT.

The word EXIT must appear by itself in a sentence. The EXIT sentence must be preceded by a paragraph-name and can be the only sentence in that paragraph.

During execution of the object program, control is normally transferred from one paragraph or section to the next paragraph or section. Occasionally this does not have the required effect. For example, the point to which control is to be transferred may be at the end of a range of procedures governed by a PERFORM statement or it could be at the end of a DECLARATIVE section. In such cases, the EXIT statement enables a procedure-name to refer to the return point. When control is transferred to an EXIT paragraph, and no associated PERFORM or USE statement is active, control passes through the EXIT point to the first sentence of the next paragraph.

EXAMPLE

```
PARA-12. PERFORM DEDUCTIONS THRU PARA-16.  
      .  
      .  
DEDUCTIONS. IF DEPENDENT IS ZERO GO TO PARA-16.  
            IF DEPENDENT = 1 SUBTRACT 800 FROM TOTAL  
            GO TO PARA-16.  
            IF DEPENDENT = 2 SUBTRACT 1600 FROM TOTAL  
            GO TO PARA-16.  
            IF DEPENDENT = 3 . . .  
            .  
            .  
PARA-16. EXIT.  
      .  
      .
```

If the value of DEPENDENT is zero, then the remainder of the procedure named DEDUCTIONS is skipped over, and a return is made to the statement immediately following the PERFORM statement. Otherwise, execution proceeds through DEDUCTIONS until PARA-16 is encountered, whereupon, control returns to the statement following the PERFORM statement.

EXIT PROGRAM Statement

This form of the EXIT marks the logical end of a called subprogram.

FORMAT

paragraph-name. EXIT PROGRAM.

The EXIT PROGRAM statement must be preceded by a paragraph-name, and must be the only statement in the paragraph.

If control reaches an EXIT PROGRAM statement in a called subprogram, control returns to the point in the calling program immediately following the CALL statement.

If control reaches an EXIT PROGRAM statement in a main program, the statement is treated simply as an EXIT statement.

Note: The following chart, which lists the effects of program termination statements, is also given for the STOP RUN and GOBACK statements:

Termination Statement	Main Program	Subprogram
EXIT PROGRAM	Non-operational—treated as EXIT	Return to calling program
STOP RUN	Logical end of run—return control to MPE	Logical end of run for both subprogram and the calling program(s)—return control to MPE
GO BACK	Logical end of run—return control to MPE	Return to calling program

GOBACK Statement

The GOBACK statement marks the logical end of either a main program or a subprogram.

FORMAT

GOBACK.

A GOBACK statement must appear as the only statement in a sentence or as the last statement in a series of imperative statements in a sentence.

If control reaches a GOBACK statement while operating under the control of a CALL statement, control returns to the point in the calling program immediately following the CALL statement. In a subprogram, the GOBACK statement acts the same as an EXIT PROGRAM.

If control reaches a GOBACK statement in a main program, it will act the same as a STOP RUN statement.

Note: The following chart, which lists the effects of program termination statements, is also given for the EXIT PROGRAM and STOP RUN statements:

Termination Statement	Main Program	Subprogram
EXIT PROGRAM	Non-operational—treated as EXIT	Return to calling program
STOP RUN	Logical end of run—return control to MPE	Logical end of run for both subprogram and the calling program(s)—return control to MPE
GOBACK	Logical end of run—return control to MPE	Return to calling program

GO TO Statement

The GO TO statement transfers control to another part of the Procedure Division.

Format 1

GO TO [procedure-name].

Format 2

GO TO procedure-name-1 [,procedure-name-2] . . . ,
procedure-name-n DEPENDING ON identifier.

Each procedure-name must be a paragraph or section-name in the Procedure Division of the program. The identifier must have been described in the Data Division as an integer and a numeric elementary item.

When a GO TO statement using Format 1 is executed, control transfers to procedure-name-1, unless the GO TO statement was altered by an ALTER statement. If altered, control passes to the procedure-name named in the ALTER statement. If the GO TO statement omits the procedure-name option, it is required that an ALTER statement referring to this GO TO statement be executed prior to execution of the GO TO statement. The following rules apply to a GO TO statement that is referred to by ALTER statements:

1. It must have a paragraph-name
2. It must be the only statement in the paragraph
3. If the priority number of the section in which the GO TO statement appears is greater than 49, the ALTER statements referring to it must be in sections with the same priority number.

A GO TO statement in Format 1 can appear only as the last or only statement in a sequence of imperative statements.

EXAMPLE

IF SAMPLE < ZERO GO TO ERR-RTN-4.

If the value of SAMPLE is less than zero, the program branches to a procedure called ERR-RTN-4. If SAMPLE is equal to or greater than zero, the GO TO statement has no effect, and control passes to the next sentence.

A GO TO statement in Format 2 passes control to one of the named procedures, depending on the value of identifier. The value of identifier must be a positive, unsigned integer in the range 1, 2, . . . , n; otherwise, the GO TO statement has no effect. If the identifier is 1, control is transferred to the first procedure named; if the identifier is five, control is transferred to the fifth procedure named, and so on.

GO TO Statement

EXAMPLE

GO TO ERR-RTN-A, ERR-RTN-B, ERR-RTN-C DEPENDING ON CRDE.

The program branches to ERR-RTN-A if the value of CRDE is 1, to ERR-RTN-B if the value of CRDE is 2, or to ERR-RTN-C if the value of CRDE is 3. Control passes to the next sentence if the value in CRDE is anything else. The above GO TO statement is equivalent to the following coding:

IF CRDE = 1, GO TO ERR-RTN-A.

IF CRDE = 2, GO TO ERR-RTN-B.

IF CRDE = 3, GO TO ERR-RTN-C.

IF Statement

The IF statement causes a condition to be tested. The results of the test determine the subsequent action of the object program.

FORMAT


$$\text{IF condition; THEN } \left\{ \begin{array}{l} \text{statement-1} \\ \text{NEXT SENTENCE} \end{array} \right\} \left[\begin{array}{l} \text{;ELSE statement-2} \\ \text{; ELSE NEXT SENTENCE} \end{array} \right]$$

Note: The word THEN as shown in the format above is not currently defined as part of American National Standard COBOL, but is included in COBOL/3000 to simplify conversion of programs written for compilers that use this non-standard language extension. Users who wish to conform to ANS COBOL should avoid using THEN in IF statements.

Conditions are fully explained in Section VII.

Statement-1 and statement-2 can be either conditional or imperative statements. ELSE NEXT SENTENCE can be omitted except in compound conditions.

The IF operation is performed as follows:

- If the condition is true; statement-1 is executed, and control passes implicitly to the next sentence, unless explicitly directed elsewhere by statement-1.
- If the condition is false, either the statements following ELSE are executed or, if the ELSE clause is omitted, the next sentence is executed.
- When the NEXT SENTENCE option is present, control passes explicitly to the next sentence depending on the truth value of the condition and the placement of the NEXT SENTENCE clause in the statement.
- When control is transferred to the next sentence, either implicitly or explicitly, control passes to the next sentence as written or to a return mechanism of a PERFORM or USE statement.

EXAMPLES

IF ERROR-CODE = 1 THEN PERFORM ERR-RTN ELSE NEXT SENTENCE.

If the field ERROR-CODE contains a value of 1, the condition is true and control passes to ERR-RTN. If ERROR-CODE contains any value other than 1, the condition is false and control passes to the next sentence. In this example, the ELSE NEXT SENTENCE clause is not required since it immediately precedes the terminal period.

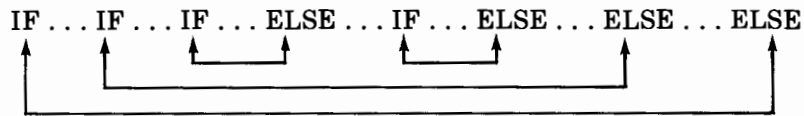
IF Statement

IF PAY < 6000 NEXT SENTENCE; ELSE GO TO TAX-ROUTINE.

If the field PAY contains a value less than 6000, the condition is true and control passes to the next sentence. If PAY contains a value equal to or greater than 6000, the condition is false and control is transferred to TAX-ROUTINE. In this example, the ELSE NEXT SENTENCE clause cannot be omitted. Note that there is no ELSE NEXT SENTENCE clause in this example.

Nested IF Statements

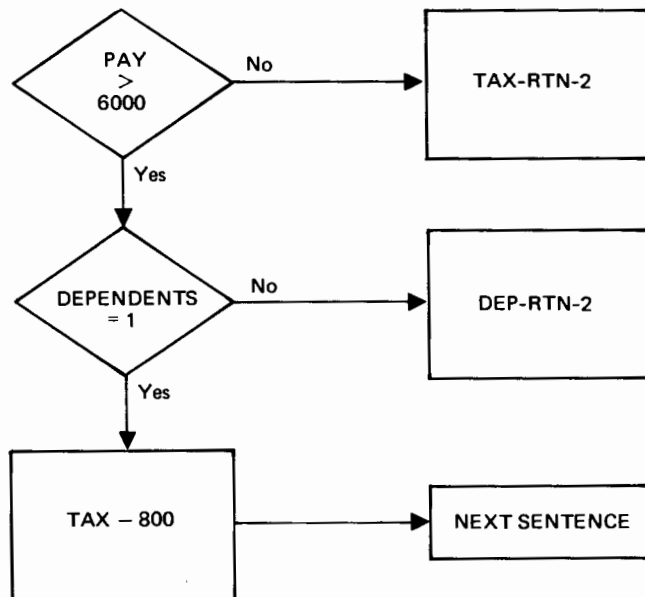
An IF statement is nested when statement-1 or statement-2 contain IF statements. Nested IF statements are considered paired IF and ELSE combinations, proceeding from left to right. Any ELSE encountered applies to the immediately preceding IF that has not yet been paired with an ELSE. The following diagram illustrates this concept:



EXAMPLE

IF PAY > 6000 THEN IF DEPENDENTS = 1 SUBTRACT
800 FROM TAX; ELSE GO TO DEP-RTN-2;
ELSE GO TO TAX-RTN-2.

The following flowchart illustrates the execution of this statement



MOVE Statement

The MOVE statement transfers data to one or more data areas and performs necessary data conversions and/or editing functions.

Format 1

$$\underline{\text{MOVE}} \left\{ \begin{array}{l} \text{identifier-1} \\ \text{literal} \end{array} \right\} \underline{\text{TO}} \text{identifier-2} [, \text{identifier-3}] \dots$$

Format 2

$$\underline{\text{MOVE}} \left\{ \begin{array}{l} \underline{\text{CORRESPONDING}} \\ \underline{\text{CORR}} \end{array} \right\} \text{identifier-1} \underline{\text{TO}} \text{identifier-2}$$

Identifier-1 and literal represent the sending area, while identifier-2, identifier-3, etc., represent the receiving area of a MOVE statement. An index data item cannot appear as an operand of a MOVE statement. The SET command is used for this purpose.

A MOVE statement in Format 1 causes the value in the sending area to be copied into each of the receiving areas, first to identifier-2, then to identifier-3, etc. Any indexing or subscripting associated with identifier-2, identifier-3, etc, is evaluated immediately before the data is moved. Data in the sending area is unchanged after the MOVE; however, the moved data overlays any data previously stored in the receiving areas.

When Format 2 is used, data items in identifier-1 are moved to corresponding data items in identifier-2 according to the rules of the CORRESPONDING option as described in Section VII. Identifier-1 and identifier-2 must be group items, and they may be qualified data-names. They cannot be data items with level numbers 66, 77, or 88. Only one identifier can appear to the right of the word TO. The results of a MOVE CORRESPONDING statement are the same as if each pair of corresponding identifiers had been referred to in separate MOVE statements.

The results of a MOVE operation vary according to the contents of the sending and receiving areas. Both the sending and receiving areas can be elementary or group items. The following four types of MOVE operations are possible:

Sending Item	Receiving Item
Elementary	Elementary
Elementary	Group
Group	Elementary
Group	Group

The MOVE operation can move up to 32767 bytes of data.

MOVE Statement

Elementary Move

An elementary move is one in which an elementary item is moved to an elementary item. The rules for such a move depend upon the class of the receiving item. Every elementary item belongs to one of the following classes:

- numeric
- alphabetic
- alphanumeric
- numeric edited
- alphanumeric edited

These data classes are described under Standard Data Positioning in Section V. Numeric literals are in the numeric class; non-numeric literals are in the alphanumeric class. The figurative constant ZERO is in the numeric class. The figurative constant SPACE is in the alphabetic class. All other figurative constants are alphanumeric.

Any necessary conversion of data from one form of internal representation to another takes place during the legal elementary moves, along with any editing specified for the receiving data item.

The following rules apply to an elementary move

- A numeric edited, alphanumeric edited, the figurative constant SPACE, or alphabetic data item must not be moved to a numeric or numeric edited data item.
- A numeric literal, the figurative constant ZERO, a numeric data item or a numeric edited data item must not be moved to an alphabetic item.
- A numeric literal, or a numeric data item whose implicit decimal point is not immediately to the right of the least significant digit, must not be moved to an alphanumeric or alphanumeric edited data item.
- When an alphanumeric edited, alphanumeric, or alphabetic item is a receiving item, justification and any necessary space-filling takes place as defined under the JUSTIFIED clause. If the size of the sending item is greater than the size of the receiving item, the excess characters are truncated after the receiving item is filled.
- When a numeric or numeric edited item is a receiving item, alignment by decimal point and any necessary zero-filling takes place, except where zeroes are replaced because of editing requirements. If the receiving item has no operational sign, the absolute value of the sending item is used. If the sending item has more digits to the left or right of the decimal point than the receiving item can contain, the excess digits are truncated. When a data item described as alphanumeric is the sending item, it is moved as though it was described as an unsigned numeric integer item (such moves are illegal if the receiving item's USAGE is COMP or COMP-3). If the sending item contains any nonnumeric characters, the results are undefined.
- When a receiving field is described as alphabetic and the sending data item contains any nonalphabetic characters, the results are undefined.
- When a packed decimal field is moved to an alphanumeric field, the sending field will be unpacked before being moved.

Non-elementary Move

Any move that is not an elementary move has exactly the same results as if it were an alphanumeric to alphanumeric elementary move, except that no editing or conversion of data from one form of internal representation to another takes place. Table 8-1 illustrates the valid MOVE operations.

Table 8-1. Permissible Moves

Receiving Field \ Source Field	Group	Alphabetic	Alphanumeric	External Decimal	Binary	Numeric Edited	Alphanumeric Edited	Internal Decimal
Group	Y	Y	Y	Y ¹	Y ¹	Y ¹	Y ¹	Y ¹
Alphabetic	Y	Y	Y	Δ	Δ	Δ	Y	Δ
Alphanumeric	Y	Y	Y	Y ⁴	Y ⁴	Y ⁴	Y	Y ⁴
External Decimal	Y ¹	Δ	Y ²	Y	Y	Y	Y ²	Y
Binary	Y ¹	Δ	Y ²	Y	Y	Y	Y ²	Y
Numeric Edited	Y	Δ	Y	Δ	Δ	Δ	Y	Δ
Alphanumeric Edited	Y	Y	Y	Δ	Δ	Δ	Y	Δ
Zeros (numeric or alphanumeric)	Y	Δ	Y	Y ³	Y ³	Y ³	Y	Y ³
Spaces	Y	Y	Y	Δ	Δ	Δ	Y	Δ
High-Value, Low-Value, Quotes	Y	Δ	Y	Δ	Δ	Δ	Y	Δ
All Literal	Y	Y	Y	Y ⁵	Y ⁵	Y ⁵	Y	Y ⁵
Numeric Literal	Y ¹	Δ	Y ²	Y	Y	Y	Y ²	Y
Nonnumeric Literal	Y	Y	Y	Y ⁵	Y ⁵	Y ⁵	Y	Y ⁵
Internal Decimal	Y ¹	Δ	Y ²	Y	Y	Y	Y ²	Y

Y = permissible; Δ = prohibited
 Y¹ = move without conversion
 Y² = permissible only if the decimal point is to the right of the least significant digit
 Y³ = a numeric move
 Y⁴ = the move is treated as an External Decimal (integer) field.
 Y⁵ = the literal must consist only of numeric characters and is treated as an External Decimal (integer) field.

MOVE Statement

EXAMPLE

MOVE ZEROS TO FIELD-A

	Before Execution	After Execution
FIELD-A	75608	00000

MOVE NAME TO EMPLOYEE

	Before Execution	After Execution
NAME	JOHNSON	JOHNSON
EMPLOYEE	HAYMAKER	JOHNSONΔ

MOVE MONTH TO FIELD-1, FIELD-2, FIELD-3

	Before Execution	After Execution
MONTH	MARCH	MARCH
FIELD-1	FEBRUARY	MARCHΔΔΔ
FIELD-2	JANUARY	MARCHΔΔ
FIELD-3	DECEMBER	MARCHΔΔΔ

EXAMPLE OF MOVE CORRESPONDING

For this example, assume the following data record structures:

01	UPDATE.	01	MAST-FILE
05	ACCT-ID.	02	ACCT-ID
07	ACCT-TYPE. . .	04	ACCT-TYPE
07	NAME.	04	NAME
09	LAST-NAME. . .	06	LAST-NAME
09	FIRST-NAME. . .	06	FIRST-NAME
07	ACCT-NUMBER. . .	04	ACCT-NUMBER
05	ACCT-HISTORY.	02	ACCT-HISTORY
07	YR-BEGAN. . .	04	YR-BEGAN
07	BALANCE. . .	04	BALANCE
05	PERS-HISTORY.		
07	AGE. . .		
07	SEX. . .		
07	SALARY. . .		

The statement **MOVE CORRESPONDING UPDATE TO MAST-FILE** moves the elementary fields **ACCT-TYPE**, **LAST-NAME**, **FIRST-NAME**, **ACCT-NUMBER**, **YR-BEGAN**, and **BALANCE** in the record **UPDATE** to the record **MAST-FILE**. **AGE**, **SEX**, and **SALARY** are not moved because they have no corresponding fields in **MAST-FILE**. Although the data to be moved occupies the same relative positions in the two records, this is not a requirement for the **MOVE CORRESPONDING** statement. In fact, its ability to reformat records is one of the most convenient features of the **MOVE CORRESPONDING** statement. However, this requires the user to be certain that the desired fields actually correspond. Fields correspond when they have the same name and the same qualification up to, but not including, the name of the group being moved. In the example **FIRST-NAME IN NAME IN ACCT-ID IN UPDATE** corresponds to **FIRST-NAME IN NAME IN ACCT-ID IN MAST-FILE**. So long as the qualification remains the same, this correspondence is not affected by the relative positions of the fields within their respective records.

In the example, the statement **MOVE CORRESPONDING UPDATE TO MASTE-FILE** generates the equivalent of the following individual moves:

```
MOVE ACCT-TYPE IN UPDATE TO ACCT-TYPE IN MAST-FILE.
MOVE LAST-NAME IN UPDATE TO LAST-NAME IN MAST-FILE.
MOVE FIRST-NAME IN UPDATE TO FIRST-NAME IN MAST-FILE.
MOVE ACCT-NUMBER IN UPDATE TO ACCT-NUMBER IN MAST-FILE.
MOVE YR-BEGAN IN UPDATE TO YR-BEGAN IN MAST-FILE.
MOVE BALANCE IN UPDATE TO BALANCE IN MAST-FILE.
```

MULTIPLY Statement

The MULTIPLY statement performs the algebraic multiplication of two numeric data items and stores the product.

Format 1

$$\underline{\text{MULTIPLY}} \left\{ \begin{array}{l} \text{identifier-1} \\ \text{literal-1} \end{array} \right\} \underline{\text{BY}} \text{identifier-2} \quad [\underline{\text{ROUNDED}}]$$

[; ON SIZE ERROR imperative statement]

Format 2

$$\underline{\text{MULTIPLY}} \left\{ \begin{array}{l} \text{identifier-1} \\ \text{literal-1} \end{array} \right\} \underline{\text{BY}} \left\{ \begin{array}{l} \text{identifier-2} \\ \text{literal-2} \end{array} \right\}$$

GIVING identifier-3 [ROUNDED]

[; ON SIZE ERROR imperative statement]

Identifier-1 and identifier-2 must refer to numeric elementary items. Identifier-3, which follows the word GIVING in Format 2, may refer to a data item that contains editing symbols. Literals must be numeric. The maximum size of an operand is eighteen decimal digits.

The ROUNDED and SIZE ERROR options are discussed in Section VII.

A MULTIPLY statement in Format 1 causes the value of identifier-1 or literal-1 to be multiplied by the value of identifier-2. The product replaces the value of identifier-2.

EXAMPLE**MULTIPLY RATE BY BASE**

	Before Execution	After Execution
BASE	1560 [^] 25	10141 [^] 625
RATE	6 [^] 5	6 [^] 5

A MULTIPLY statement in Format 2 multiplies the value of identifier-1 or literal-1 by the value of identifier-2 or literal-2, storing the result in identifier-3.

EXAMPLE**MULTIPLY BASE BY .083 GIVING PROFIT-SHARE**

	Before Execution	After Execution
BASE	13000 [^]	13000 [^]
PROFIT-SHARE	1204 [^] 5	1079 [^] 0

NOTE Statement

The purpose of the NOTE statement is to allow the programmer to insert comments into the Procedure Division of a program. While comment lines can appear anywhere in the program (see Section I), a NOTE statement is restricted to the Procedure Division.

FORMAT

NOTE character-string

The character string consists of any combination of characters in the HP 3000 character set.

The NOTE statement can be the only statement in a sentence. If a NOTE statement is the first statement of a paragraph, the character-string consists of the entire paragraph. If the NOTE statement is any other than the first statement of a paragraph, the character-string ends when the first period followed by a space is encountered.

EXAMPLE

PARAGRAPH-1. NOTE THIS SECTION OF THE PROCEDURE DIVISION
PERFORMS ALL ARITHMETIC OPERATIONS. THEN IT
TRANSFERS CONTROL TO THE BALANCE ROUTINE.

PERFORM Statement

The PERFORM statement interrupts the sequential execution of a program by transferring control to one or more procedures, which are executed a specified number of times or until a particular condition is satisfied. Sequential execution is then resumed. The PERFORM VARYING option (Format 4) causes the contents of one or more identifiers or index-names to be systematically altered.

Format 1

PERFORM procedure-name-1 [THRU procedure-name-2]

Format 2

PERFORM procedure-name-1 [THRU procedure-name-2]

{ identifier-1 }
{ integer-1 } TIMES

Format 3

PERFORM procedure-name-1 [THRU procedure-name-2]

UNTIL condition-1

Format 4

PERFORM procedure-name-1 [THRU procedure-name-2]

PERFORM Statement

$$\underline{\text{VARYING}} \left\{ \begin{array}{l} \text{index-name-1} \\ \text{identifier-1} \end{array} \right\} \underline{\text{FROM}} \left\{ \begin{array}{l} \text{index-name-2} \\ \text{literal-2} \\ \text{identifier-2} \end{array} \right\}$$
$$\underline{\text{BY}} \left\{ \begin{array}{l} \text{literal-3} \\ \text{identifier-3} \end{array} \right\} \underline{\text{UNTIL}} \text{ condition-1}$$
$$\left[\underline{\text{AFTER}} \left\{ \begin{array}{l} \text{index-name-4} \\ \text{identifier-4} \end{array} \right\} \underline{\text{FROM}} \left\{ \begin{array}{l} \text{index-name-5} \\ \text{literal-5} \\ \text{identifier-5} \end{array} \right\} \right.$$
$$\underline{\text{BY}} \left\{ \begin{array}{l} \text{literal-6} \\ \text{identifier-6} \end{array} \right\} \underline{\text{UNTIL}} \text{ condition-2}$$
$$\left[\underline{\text{AFTER}} \left\{ \begin{array}{l} \text{index-name-7} \\ \text{identifier-7} \end{array} \right\} \underline{\text{FROM}} \left\{ \begin{array}{l} \text{index-name-8} \\ \text{literal-8} \\ \text{identifier-8} \end{array} \right\} \right.$$
$$\underline{\text{BY}} \left\{ \begin{array}{l} \text{literal-9} \\ \text{identifier-9} \end{array} \right\} \underline{\text{UNTIL}} \text{ condition-3} \left. \right]$$

Each procedure-name must be the name of a section or paragraph in the Procedure Division. Each identifier must represent a numeric elementary item described in the Data Division. Identifier-1 in Format 2 must specify a numeric item with no positions to the right of the assumed decimal point. Literals must be numeric.

When a PERFORM statement is executed, program control transfers to the first statement of procedure-name-1. If only one procedure-name is specified, that paragraph or section is executed in sequence. If two procedure-names are specified, the sequence begins with the first statement in procedure-name-1, continuing through the last statement of procedure-name-2. Upon execution of all statements within the range of the PERFORM statement, an automatic return is made to the statement following the PERFORM statement. This return is subject to the options specified in the statement. Specifically; the rules are as follows:

- If procedure-name-1 is a paragraph-name and procedure-name-2 is not specified, then the return is made following execution of the last statement of procedure-name-1.

- If procedure-name-1 is a section-name and procedure-name-2 is not specified, the return is made following the last statement of the last paragraph in procedure-name-1.
- If procedure-name-2 is specified, and it is a paragraph-name, the return is made following execution of the last statement of the paragraph.
- If procedure-name-2 is specified, and it is a section-name, the return is made following the last sentence of the last paragraph in that section.

The last statement in a procedure referenced by a PERFORM statement cannot be a GO TO statement since this would not ensure the expected return path. When two or more direct paths to the return point of a PERFORM statement exist, procedure-name-2 may be the name of a paragraph consisting only of an EXIT statement. All alternate paths then must end at this paragraph.

GO TO and other PERFORM statements may occur between the start of procedure-name-1 and the end of procedure-name-2, as long as all the paths eventually end at the last sentence of procedure-name-2.

Segmentation Consideration

The COBOL/3000 compiler allows the PERFORM statement to have within its range sections with any priority number. When a PERFORM statement refers to a procedure-name that is in a segment with a different priority number, and that number is greater than 49, the segment referred to is made available in its initial state for each execution of the PERFORM statement, except for the PERFORM-TIMES statement, which executes the segment in its initial state only the first time.

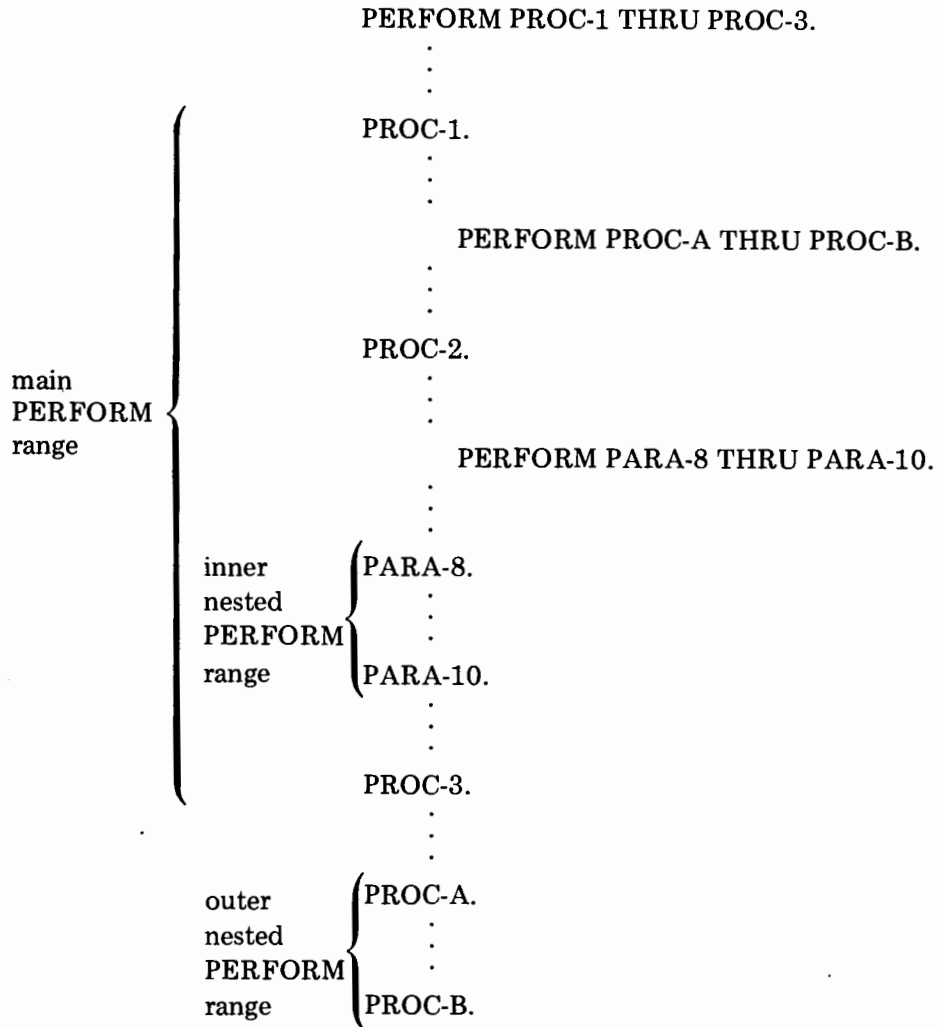
Nested PERFORM Statement

When a sequence of statements referred to by a PERFORM statement contains a nested PERFORM statement, the sequence of procedures referred to by the nested PERFORM statement must be either entirely within the sequence referred to by the initial PERFORM statement or entirely outside those limits.

When two or more PERFORM statements have the same procedure-name-2 (or implied procedure-name-2): following execution of procedure-name-2, a return to the statement following the last PERFORM statement will be executed. *No other returns will result.* In order to force returns for each PERFORM, separate procedure-name-2 procedure names should be used -- for example, procedure names containing only an EXIT statement.

PERFORM Statement

EXAMPLE



Simple PERFORM Statement

A procedure or procedures referenced by a PERFORM statement in Format 1 are executed once. Control then returns to the statement following the PERFORM statement.

EXAMPLE

```
PERFORM FICA.
```

The procedure named FICA is executed once.

TIMES Option

When Format 2 is used, the designated procedures are executed *n* times, *n* being the number specified by the value of identifier-1 or integer-1. The value of identifier-1 must be positive during execution of the statement. If it is negative or zero, control passes immediately to the statement following the PERFORM statement. If procedure-name-1 has a priority-number greater than 49 and is different from the current segment, procedure-name-1 is executed in initial state the first time only.

EXAMPLE

```
PERFORM WRITE-ROUTINE 50 TIMES.
```

Upon execution of this statement, control passes to the procedure named WRITE-ROUTINE, which is then executed 50 times, whereupon control returns to the statement following the PERFORM.

UNTIL Option

When Format 3 is used, the UNTIL option specifies that the procedure or procedures be executed until condition-1 is satisfied. If it happens that the condition is true when the PERFORM statement is encountered, the specified procedure is not executed, and control falls through to the next statement.

```
PERFORM DEDUCTIONS UNTIL DEPENDENT = 0.
```

This statement causes the procedure called DEDUCTIONS to be repeatedly executed until the value of DEPENDENT is equal to zero.

PERFORM VARYING Option

Format 4 is the PERFORM VARYING option, which is used to increase or decrease one or more identifiers or index-names during execution of the statement. The PERFORM VARYING statement is executed repetitively until a condition is satisfied. A maximum of three identifiers or index-names can be varied, each according to a specified condition. The identifier or literal following the word BY specifies the amount the identifier or index-name to be varied will be augmented. A negative value acts as a decrement; a positive value acts as an increment. A value of zero is invalid for this statement.

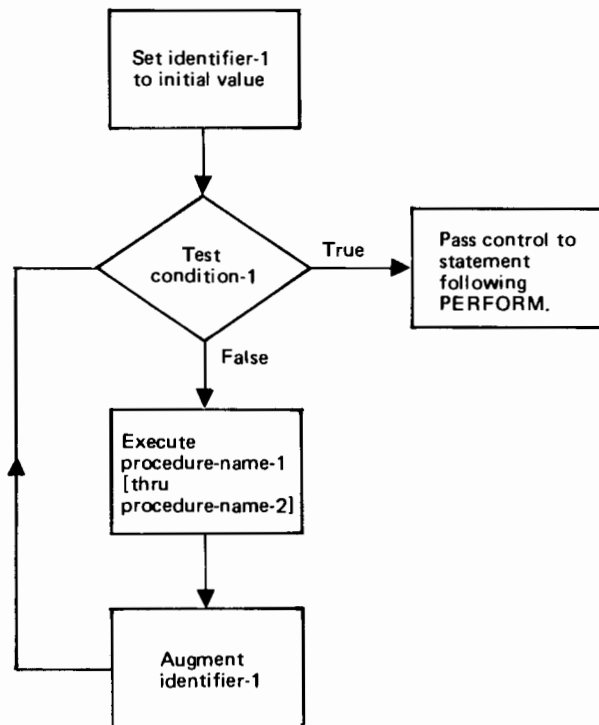
PERFORM Statement

Varying One Identifier

When one identifier (or index-name) is varied, the following sequence occurs:

1. Identifier-1 is set equal to its starting value, which is the value of index-name-2, identifier-2, or literal-2.
2. If condition-1 is true, control passes through to the next statement.
3. If the condition is false, the specified sequence of procedures is executed once, and the value of identifier-1 is augmented by the increment or decrement value specified by literal-3 or identifier-3.
4. Steps 2 and 3 are repeated until condition-1 is true.

The following flowchart illustrates this cycle.

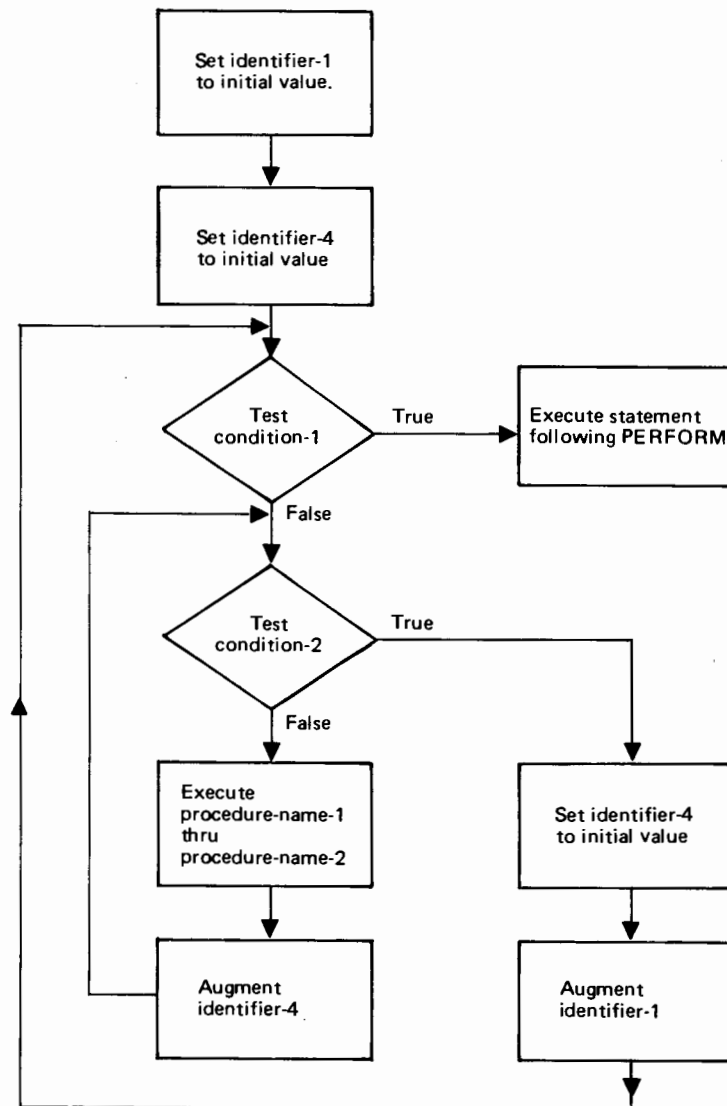


Varying Two Identifiers

When two identifiers (or index-names) are varied, the following sequence takes place.

1. Identifier-1 and identifier-4 are set to their initial values, which are identifier-2 and identifier-5, respectively.
2. Condition-1 is evaluated. If true, control passes to the statement following the PERFORM statement.
3. If condition-1 is false, condition-2 is evaluated.
4. If condition-2 is false, procedure-name-1 through procedure-name-2 is executed. Identifier-4 is augmented by identifier or literal-6, after which condition-2 is evaluated again. If the result is false, this cycle is repeated until condition-2 is true. As soon as condition-2 is true, identifier-4 is set to its initial value, identifier-1 is augmented by identifier-3, and steps 2 through 4 are repeated until condition-1 is true.

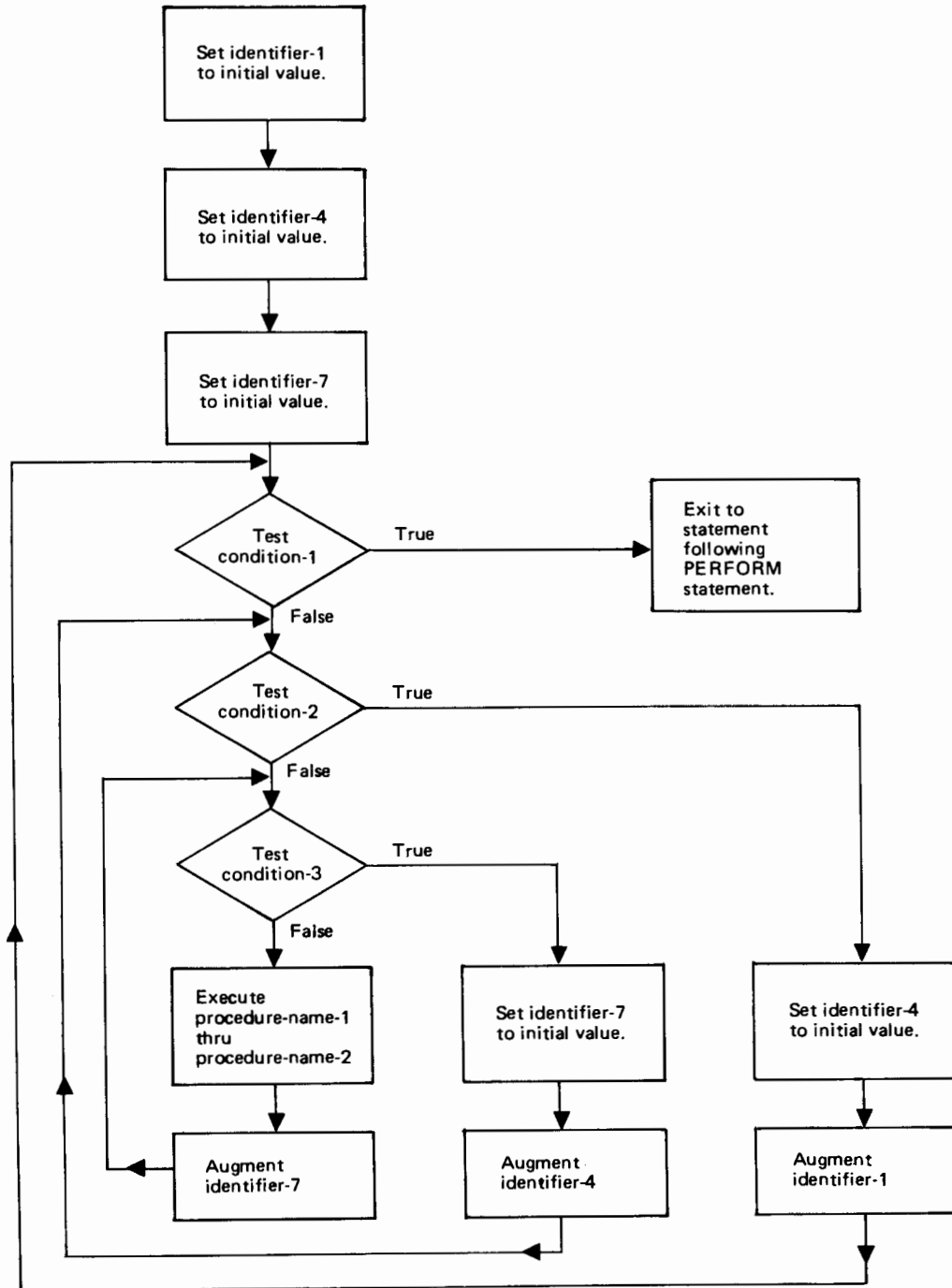
The following flowchart illustrates the above sequence.



Varying Three Identifiers

The procedure for varying three identifiers is essentially the same as for two identifiers. The third level of the PERFORM statement goes through a complete cycle each time the second level is augmented; and the second level goes through a complete cycle each time the first level is augmented. When all three conditions are satisfied, the PERFORM statement terminates. At this point, identifier-4 and identifier-7 contain their initial values, while the value of identifier-1 exceeds its last used setting by one increment or decrement, unless condition-1 was true when the PERFORM statement was first entered. In this case, identifier-1 also will contain its initial value.

The following flowchart illustrates the sequence that occurs when three identifiers are varied.



EXAMPLES: PERFORM VARYING

Although PERFORM with the VARYING option may be used for a variety of functions, it is an especially powerful tool for table handling. The following examples of the PERFORM VARYING statement are all based on a simple table containing projected and actual budget figures. This table is graphically illustrated below:

	Projected	Actual
<i>salaries</i>	9999999V99	9999999V99
<i>benefits</i>	9999999V99	9999999V99
<i>office equipment</i>	9999999V99	9999999V99
<i>rent</i>	9999999V99	9999999V99
<i>supplies</i>	9999999V99	9999999V99
<i>telephone</i>	9999999V99	9999999V99
<i>miscellaneous</i>	9999999V99	9999999V99
<i>total</i>	9999999V99	9999999V99

In the illustration, the titles "salaries," "benefits," etc., are added simply for the sake of clarity. These are neither data-names nor elements of the table. The names PROJECTED and ACTUAL are data-names used to access the data in the table. These data names are subordinate to an OCCURS clause and must always be indexed or subscripted when referenced in the Procedure Division. For the purposes of this example, the index-name ITEM-INDEX will be used. The Data Division coding for this table is as follows:

```

01 BUDGET-TABLE
   02 BUDGET OCCURS 8 TIMES INDEXED BY ITEM-INDEX.
      04 PROJECTED PIC 9(7)V99.
      04 ACTUAL PIC 9(7)V99.

```

When the sample coding is executed, CALC-RTN is executed a total of 84 times, seven times for each of the twelve months. Throughout these iterations of CALC-RTN, the PERFORM VARYING statement alters ITEM-INDEX and MONTH-INDEX so that items one through seven for January are manipulated, and then items one through seven for February, and so on. Notice that the indexes MONTH-POINTER and ITEM-POINTER are used as constants and are unchanged throughout the execution of CALC-RTN.

SEARCH Statement

The SEARCH statement is used to search a table for a table-element that satisfies the specified condition and to adjust the associated index-name to indicate that table-element.

Format 1

SEARCH identifier-1 [VARYING { index-name-1 }]

[; AT END imperative-statement-1]

; WHEN condition-1 { imperative-statement-2 }
NEXT SENTENCE }

[; WHEN condition-2 { imperative-statement-3 }] ...



Format 2

SEARCH ALL identifier-1 [; AT END imperative-statement-1]

; WHEN condition-1 { imperative-statement-2 }
NEXT SENTENCE }

In both formats 1 and 2, identifier-1 must not be subscripted or indexed, but its description must contain an OCCURS clause and an INDEXED BY clause. The description of identifier-1 for format 2 must also contain the KEY IS option in its OCCURS clause.

If the AT END option is not specified for either format, control passes to the next sentence when the end of the table is reached.

OPERATION OF FORMAT 1

Format 1 specifies a serial search starting at the current setting of the table-index associated with identifier-1. If the programmer wishes to search the entire table, he must first SET the table-index to 1. (See the SET statement in this section.)

The SEARCH statement evaluates the specified conditions only if the table-index indicates an occurrence within the table. If the table-index initially indicates an occurrence outside the table boundaries or if the table boundaries are exceeded during the SEARCH, control immediately passes to imperative-statement-1 or to the next sentence if the AT END option is not specified.

Condition-1, condition-2, etc., may be any condition. Conditions are fully explained in Section VII.

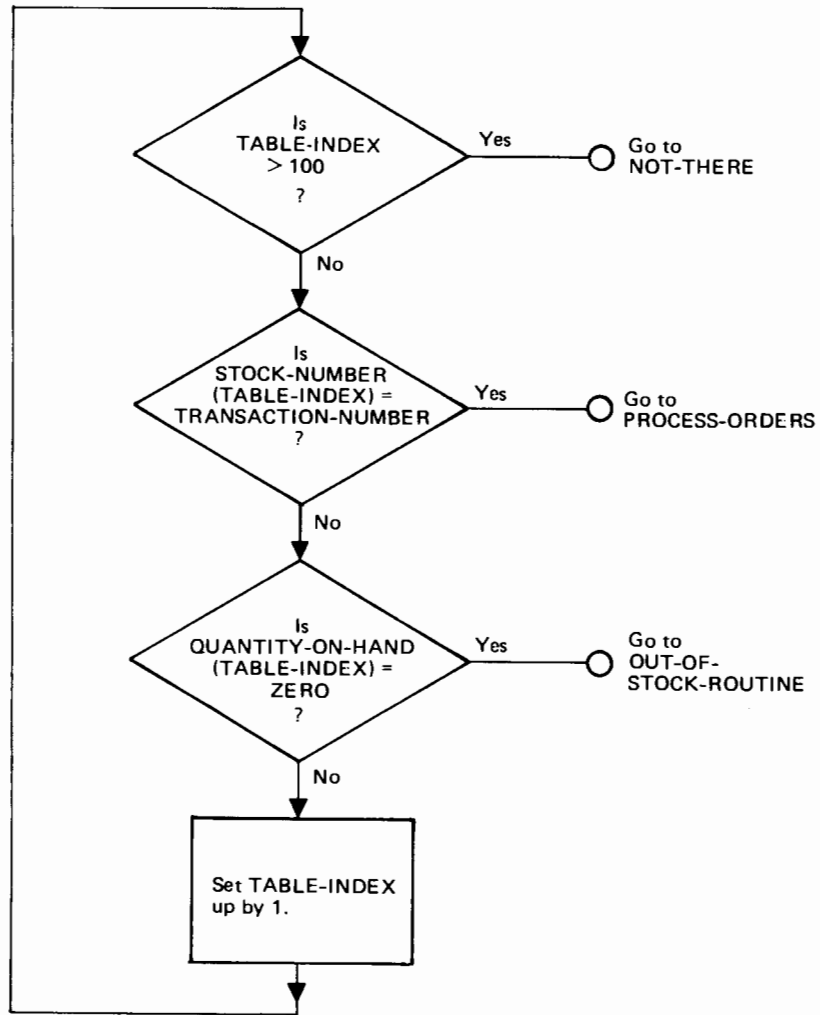
The SEARCH statement tests the conditions in the order in which they are written, making use of the table-index setting to determine the occurrence of the tested table element. If none of the conditions is satisfied, the table-index is incremented to indicate the next occurrence. This process continues until an element is found to satisfy any one of the specified conditions or until the end of the table is reached. (At this time, the table index exceeds the number of occurrences by one.) When an element is found to satisfy one of the conditions, control passes to the imperative-statement associated with that condition. The table-index indicates the element that satisfied the condition.

The effect of multiple WHEN clauses is similar to that of the logical OR operator in conditional statements. The SEARCH will be terminated when condition-1 OR condition-2 OR condition-3, etc., is satisfied. However, the use of multiple WHEN clauses allows the use of a separate imperative-statement for each WHEN clause. Effectively, this gives the user multiple exits from the SEARCH statement. For example, although the following SEARCH statements both terminate indicating the same table element, the second statements offers greater flexibility in processing:

```
SEARCH STOCK-TABLE AT END GO TO NOT-THERE;  
  WHEN STOCK-NUMBER (TABLE-INDEX) EQUALS TRANSACTION-NUMBER  
  OR QUANTITY-ON-HAND (TABLE-INDEX) EQUALS ZERO  
  GO TO PROCESS-ORDERS.
```

```
SEARCH STOCK-TABLE AT END GO TO NOT-THERE;  
  WHEN STOCK-NUMBER (TABLE-INDEX) EQUALS TRANSACTION-NUMBER  
  GO TO PROCESS-ORDERS; WHEN QUANTITY-ON-HAND (TABLE-INDEX) EQUALS  
  ZERO  
  GO TO OUT-OF-STOCK-RTN.
```

The second SEARCH statement includes two WHEN clauses. Assuming that the table has a maximum of 100 elements, the logic of this statement can be charted as shown below:



OPERATION OF FORMAT 2

The KEY clause is required in the description of any table which is accessed using Format 2 of the SEARCH statement.

Condition-1 in this format must be a condition-name condition, an EQUAL TO condition, or a compound condition with AND as the only connective. Condition-name conditions may be used only when the VALUE clause for the condition-name contains only a single value.

Each simple condition in condition-1 must be an EQUAL TO condition with a data-name that appears in the KEY clause of identifier-1 as subject or object (but not both). All preceding data-names in the KEY clause must be also included within condition-1.

The programmer is not required to initialize the table-index for the SEARCH ALL statement. This statement specifies a binary search, which requires that the table be organized with keys in either ascending or descending sequence. When the search is executed, the search logic—using the first-named index associated with identifier-1—locates the mid-point of the table and determines which half of the table contains the desired item. The search then locates the mid-point of the remaining portion of the table and then determines which half of the remaining portion contains the desired item. This process of dividing the table into ever-smaller sections continues until the desired item is found or until it can be determined that the item is not in the table. (For example, if the desired item has a key of 61, and the search finds that table-element 60 is immediately followed by element 65, then the desired item cannot be in the table.)

When condition-1 cannot be satisfied, the imperative statement specified in the AT END clause is executed, or—if the AT END clause is not specified—control passes to the next sentence.

The SEARCH ALL statement provides maximum efficiency when manipulating large tables. For small tables, with twenty-five elements or less, the SEARCH statement generally provides maximum efficiency.

VARYING OPTION

The effect of the VARYING option depends on the description of index-name-1 or identifier-2. Index-name-1 must be either a table-index associated with the table being searched or another table in the program. Identifier-2 may be either an index-data-item or an elementary numeric field with no positions to the right of the implied decimal point. In the latter case, the field may be used as a subscript. The effect of each of these options is listed below:

Index-Name in Same Table. The INDEXED BY clause associated with this table may list several indexes. The programmer uses the VARYING option to specify which one index from the list is to be used for this search. If the VARYING option is not used, or if one of the following options is specified in the VARYING option, then the index named first in the INDEXED BY clause is used for the search.

Table-indexes represent the displacement of a table element from the beginning of the table (rather than the occurrence number of an element). For example, the value of the table-index pointing to the third 50-character entry in a table would be 100.

Index-name in Different Table. Index-name-1 in the VARYING option may be associated with some other table in the program. In this case, the search of identifier-1 is accomplished using the first-named table-index for the table. As the search is executed, the index named in the VARYING option is varied by the same amount as the index used for the search. Thus, if the search of identifier-1 is satisfied by the eighth occurrence in the table, index-name-1 indicates the eighth occurrence in its associated table. Thus, parallel access of two different tables may be accomplished following a single SEARCH statement.

Identifier-2 (Index-Data-Item). An index-data-item stores the contents of a table-index without conversion to an occurrence number. Upon termination of the SEARCH statement, identifier-2 is set equal to the table-index used for the search, and may be used in subsequent SET statements or conditional statement. If the end of the table is reached in the SEARCH statement, both the table-index and identifier-2 contain a value indicating one occurrence past the end of the table.

Identifier-2 (Subscript). When identifier-2 is an elementary numeric item, its contents correspond to an occurrence number and may be used in SET statements or as a subscript for accessing a table. If the end of the table is reached in the SEARCH statement, identifier-2 contains an integer equal to the number of occurrences in the table plus one.

MULTI-DIMENSIONAL TABLES

Remember that the SEARCH statement specifies a serial search starting at the current setting of the index associated with identifier-1. Therefore, to search all levels of a multi-dimensional table requires more than one execution of the SEARCH statement (unless, of course, the search condition is satisfied). This concept is best illustrated through the use of an example. In the following example, assume that ITEM-INDEX is the index associated with ITEM-TABLE:

```
SEARCH ITEM-TABLE WHEN TRANSACTION-NUMBER EQUALS
      ITEM-NUMBER (DISTRICT-INDEX, WAREHOUSE-INDEX, ITEM-INDEX)
GO TO UPDATE-RECORD.
```

The search begins using the current setting of ITEM-INDEX, and only ITEM-INDEX is varied by the search. If the condition is not satisfied before ITEM-INDEX reaches its maximum occurrence number, then the item is not found and the search terminates. If the programmer wishes to continue searching this table, he must alter the setting of one or both of the other indexes associated with the table (SET WAREHOUSE-INDEX UP BY 1, for example).

In most cases it is more convenient to manipulate multi-dimensional tables with the PERFORM VARYING statement.

When multiple indexes are coded within parentheses, the following rules apply:

- The index-name of the most inclusive table is listed first; the index-name of the least inclusive table is listed last.
- A comma followed by a space must separate the index-names.
- No other spaces may appear within the parentheses except for those required to indicate relative indexing. (Relative indexing is fully explained under "Indexing and Subscripting" in Section I.)

SET Statement

The SET statement establishes reference-points for table handling operations by adjusting an index, index-data-item, or identifier.

Format 1

$$\underline{\text{SET}} \left\{ \begin{array}{l} \text{index-name-1} \quad [, \text{index-name-2}] \dots \\ \text{identifier-1} \quad [, \text{identifier-2}] \dots \\ \text{index-data-item-1} [, \text{index-data-item-2}] \dots \end{array} \right\} \underline{\text{TO}} \left\{ \begin{array}{l} \text{index-name-3} \\ \text{identifier-3} \\ \text{index-data-item-3} \\ \text{literal-1} \end{array} \right\} .$$

Format 2

$$\underline{\text{SET}} \text{ index-name-4} [, \text{index-name-5}] \dots \left\{ \begin{array}{l} \underline{\text{UP BY}} \\ \underline{\text{DOWN BY}} \end{array} \right\} \left\{ \begin{array}{l} \text{identifier-4} \\ \text{literal-2} \end{array} \right\} .$$

In both formats, a literal must be a positive integer.

Except for identifier-4, all identifiers may be either an index-data-item or an elementary numeric item with no positions to the right of the implied decimal point. Identifier-4 may not be an index-data-item.

The SET statement functions without regard for the limits of a table. Thus it is possible for the programmer to erroneously set a table index to refer to a nonexistent element. The programmer should provide logic to avoid such a situation.

Each index-name is associated with a particular table by the INDEXED BY clause for that table. Multiple index-names may be assigned to a particular table, but each index is associated with only one table. Index-data-items are simply storage locations where the contents of a table-index may be stored without any conversion from a displacement to an occurrence number. Index-data-items are not associated with any particular table.

Identifiers are not associated with a particular table. Therefore, the indexes for several different tables may all be set to the occurrence specified by one identifier.

In the following, all references to index-name-1, identifier-1, index-data-item-1, and index-name-4 apply equally to index-name-2, identifier-2, index-data-item-2, and index-name-5, respectively.

OPERATION OF FORMAT 1

An index-data-item can be SET only by an index-name or another index-data-item. Conversely, an index-data-item may SET only an index-name or another index-data-item. During execution of the SET statement, the contents of index-name-3 or index-data-item-3 are simply transferred to index-name-1 or index-data-item-1. The contents of index-name-3, index-data-item-3, etc. are unaffected by the SET statement.

Identifier-1 must be an elementary numeric field containing a positive value with no positions to the right of the implied decimal point. Only an index-name may be used to set identifier-1. During execution of the SET statement, the contents of the index-name are translated from a displacement to an occurrence number which corresponds to the current setting of the index-name. This value is then stored in identifier-1. Any indexing or subscripting associated with the identifier is evaluated immediately before the value is stored.

Index-name-1 may be set by any one of the options shown in the format.

The SET procedures are repeated for index-name-2, identifier-2, etc., if specified.

OPERATION OF FORMAT 2

During execution of the SET statement, index-name-4 is incremented (UP BY) or decremented (DOWN BY) a displacement value representing the number of occurrences represented by literal-2 or identifier-4. The process is repeated for index-name-5, etc., if specified. The value contained in identifier-4 is unaffected by the SET statement.

SET Statement

EXAMPLE

The following coding is a simple routine used to build a table from data read in on cards. Both formats of the SET statement are used to build the table.

```
INITIALIZE-TABLE.  
  SET TABLE-INDEX TO 1.  
BUILD-TABLE.  
  READ CARD-FILE AT END GO TO START-PROCESSING.  
  MOVE CARD-DATA TO TABLE-DATA (TABLE-INDEX).  
  IF TABLE-INDEX < 50 SET TABLE-INDEX UP BY 1, ELSE GO TO ERR-1.  
  GO TO BUILD-TABLE.  
START-PROCESSING.  
  .  
  .
```

Notice that this example contains an IF statement to control the number of items entered into the table. Because the SET statement operates without regard for the number of occurrences specified for the table, the user must ensure that the SET statement is not allowed to overlay other data areas.

Remember that index-names and index-data-items contain character displacements (not occurrence numbers). When index-name-1 is set to index-name-3 the displacement is converted to represent the table associated with index-name-1. An index-data-item, however, is not associated with a table and no occurrence number can be calculated. Therefore, when an index-name is set to an index-data-item, no conversion takes place.

STOP Statement

The STOP statement halts execution of the object program.

FORMAT

$$\underline{\text{STOP}} \left\{ \begin{array}{l} \text{literal} \\ \underline{\text{RUN}} \end{array} \right\} .$$

Literal Option

The literal can be either numeric or nonnumeric. It may also be any figurative constant other than ALL.

The STOP statement with the literal option is executed as follows:

1. The literal is communicated to the computer operator via a console print-out.
2. A system-generated message code is automatically displayed at the console, followed by the message, TYPE GO TO RESUME.
3. Object program execution is suspended.
4. When the computer operator enters the response, REPLY pin, GO, program execution resumes at the statement immediately following the STOP statement. Thus, program execution can be resumed only by operator intervention. (The term pin indicates Process Identification Number, as defined in the MPE or MPET reference manuals.)

RUN Option

The STOP RUN statement marks the logical end of the run. This statement returns control to MPE. If the STOP RUN statement is encountered during the execution of a called subprogram, it terminates the calling program and all related subprograms.

The STOP RUN statement must appear as the last statement in a series of imperative statements or as the only statement in a paragraph.

STOP Statement

Note: The following chart, which lists the effects of program termination statements, is also given for the EXIT PROGRAM and GOBACK statements. For additional information concerning interprogram communication, see the CALL statement.

Termination Statement	Main Program	Subprogram
EXIT PROGRAM	Non-operational—treated as EXIT	Return to calling program
STOP RUN	Logical end of run—return control to MPE	Logical end of run for both subprogram and the calling program(s)—return control to MPE
GOBACK	Logical end of run—return control to MPE	Return to calling program

SUBTRACT Statement

The purpose of the SUBTRACT statement is to subtract one or the sum of two or more numeric data items from one or more items. The result is stored in one or more of the items.

Format 1

$$\underline{\text{SUBTRACT}} \left\{ \begin{array}{l} \text{literal-1} \\ \text{identifier-1} \end{array} \right\} \left[\begin{array}{l} \text{,literal-2} \\ \text{,identifier-2} \end{array} \right] \dots$$

FROM identifier-m [ROUNDED]

[,identifier-n [ROUNDED]] ...

[; ON SIZE ERROR imperative-statement]

Format 2

$$\underline{\text{SUBTRACT}} \left\{ \begin{array}{l} \text{literal-1} \\ \text{identifier-1} \end{array} \right\} \left[\begin{array}{l} \text{,literal-2} \\ \text{,identifier-2} \end{array} \right] \dots$$

FROM $\left\{ \begin{array}{l} \text{literal-m} \\ \text{identifier-m} \end{array} \right\}$ GIVING identifier-n [ROUNDED]

[; ON SIZE ERROR imperative-statement]

Format 3

$$\underline{\text{SUBTRACT}} \left\{ \begin{array}{l} \underline{\text{CORRESPONDING}} \\ \underline{\text{CORR}} \end{array} \right\} \text{identifier-1}$$

FROM identifier-2 [ROUNDED]

[; ON SIZE ERROR imperative-statement]

SUBTRACT Statement

Identifiers in formats 1 and 2 must refer to numeric elementary items except for the identifier to the right of the word GIVING. This identifier may refer to a data item containing editing symbols. The maximum size of each operand is eighteen decimal digits. The maximum size of the result, after decimal point alignment, is eighteen decimal digits, excluding any editing symbols in the data item following the word GIVING.

The SIZE ERROR and ROUNDED options are explained in Section VII.

A SUBTRACT statement in Format 1 causes all literals or identifiers preceding the word FROM to be added together. This total is then subtracted from each identifier following the word FROM. The results are stored as the new values of the respective identifiers following the word FROM.

A SUBTRACT statement in Format 2 causes all literals or identifiers preceding the word FROM to be added together. This sum is then subtracted from literal-m or literal-n, and the result becomes the new value of identifier-n.

EXAMPLE

SUBTRACT CHECKS, CHARGES FROM BALANCE (MAR) GIVING BALANCE (APR).

	Before Execution	After Execution
CHECKS	576.95	576.95
CHARGES	7.00	7.00
BALANCE (MAR)	794.32	794.32
BALANCE (APR)	650.24	210.37

When Format 3 of SUBTRACT is used, data items in identifier-1 are subtracted from and stored into corresponding data items in identifier-2. See the discussion of the CORRESPONDING option at the start of this section.

PART 3
Input-Output

SECTION IX

I-O Conventions

Defining a file in a COBOL program requires entries in both the Environment Division and the Data Division.

In the Environment Division, the statement `SELECT file-name ASSIGN TO system-file-name` establishes the link between the program and external file devices. The name is the name by which the program references the file. System-file-name is the name of the file as it is known to MPE, and also specifies the class of device assigned to the file. If the user were to refer to a file by means of the MPE `:FILE` command, he would use the system-file-name (without the device class code). Within the COBOL program, all `OPEN`, `CLOSE`, and `READ` statements refer to the file-name. Other clauses in the `SELECT` statement specify other physical requirements of the file such as the number of buffers required for the file.

In the Data Division, the `FD` statement describes the characteristics of the file. Equally important, the Data Division coding establishes the relationship of individual records (01-level entries) and files (`FD` entries). In effect the compiler establishes a hierarchy in which a record is subordinate to a particular file-name which in turn is subordinate to a system-file-name. This is similar to the hierarchies established for data structures, except that the system-file-name is external to the program.

INTERACTION WITH MPE

The COBOL compiler builds a list of parameters from the information in the `SELECT` statement and the `FD` statement. This list becomes a part of the object program. When the program is executed, all the required parameters are passed to the MPE operating system which actually performs all input/output operations for the COBOL program. Because the program passes the parameters to MPE, there is no need to supply MPE commands each time the program is executed.

Note: A file defined as `OPTIONAL` may cause an exception to this rule. If the optional file is not present, the MPE command `:FILE` must be used to equate the file to `$NULL`. For additional information, see the `OPEN` statement in Section X.

If the user wishes to override the file description compiled into the COBOL program, he may do so by supplying MPE commands when the program is executed. Because the override does not change the file description compiled into the program, it is effective only for the current execution of the program.

SECTION X

Input-Output Statements

The COBOL/3000 input-output statements are presented alphabetically in this section.

ACCEPT Statement

The ACCEPT statement enables the program to accept low volume input data without the need to define a file structure for that data.

FORMAT

$$\underline{\text{ACCEPT}} \text{ identifier } \left[\underline{\text{FROM}} \left\{ \begin{array}{l} \underline{\text{SYSIN}} \\ \underline{\text{CONSOLE}} \\ \text{mnemonic-name} \end{array} \right\} \right].$$

If used, mnemonic-name must be specified in the Special-Names Paragraph of the Environment Division and must assume the meaning of SYSIN or CONSOLE.

The SYSIN device for a job run in the batch mode is the card reader; the SYSIN device for a session (the job is run from a terminal in the time-sharing mode) is the user's terminal. When the FROM option is not specified, the compiler assumes the SYSIN device.

Note: Special care must be taken when the SYSIN device is used for the ACCEPT statement, and the program is running in the batch mode. In this case, the SYSIN device is the card reader. An ACCEPT statement simply reads the next card in the reader. Without careful planning, this card could be a data card, or a control card for some other program.

The CONSOLE device is the computer operator's console.

Programming Considerations

The ACCEPT statement does not signal that it is waiting for a response. Therefore, a DISPLAY statement should usually precede an ACCEPT statement. This DISPLAY statement serves the dual purposes of warning the user that a response is required to continue the program, and to indicate what the expected response might be.

Certain hardware constraints apply to the ACCEPT statement. For example, when the SYSIN device is a card reader, the maximum number of characters that can be transferred is 80. The maximum number of characters that can be transferred from a terminal depends on the width of the device's carriage; responses from these devices are terminated by the user.

The ACCEPT statement will issue multiple requests for data until sufficient data is read. If the identifier specifies 60 characters and the SYSIN device is a card reader, the last 20 characters on the card(s) is ignored.

When an ACCEPT statement specifying the FROM mnemonic-name for CONSOLE option or FROM CONSOLE is executed, the field specified by identifier must not exceed 31 characters and the following actions result:

1. A system-generated message code is automatically displayed at the console, followed by the message, Awaiting Reply.
2. Object program execution is suspended.
3. When the computer operator enters the input data requested, this data is moved to the field specified by identifier. In this field, the input data is left-justified regardless of the associated PICTURE clause.

Note: If the field containing the input data is not completely filled by this operation, the low-order positions may contain invalid data.

When numeric data is to be input through the ACCEPT statement, the programmer must resolve the problems of decimal point alignment as well as leading and trailing zero fill. The following conventions should be observed:

- Identifier must be defined as X-type data or as a group item.
- The number of characters input should always be equal to the length defined for identifier.
- The possibility of error is greatly reduced if the program issues explicit instructions concerning the data to be entered to the person who must enter that data. The DISPLAY statement is invaluable for this purpose.
- If the period character is entered as a decimal point along with the significant data, the program must strip out the period before the numeric data can be used in arithmetic operations. This technique simplifies the task of data entry (and is therefore less error prone) at the cost of programming overhead.

EXAMPLES

The following coding is a typical example of an ACCEPT statement. Notice the use of the DISPLAY statement before the ACCEPT statement.

```
DISPLAY "IS THIS END-OF-MONTH? REPLY YES OR NO".
ACCEPT E-O-M-FLAG.
```

The following example presents one technique for removing a period entered as a part of a numeric field that must be used for subsequent arithmetic operations. The following data description coding appears in the Working-Storage Section:

```
01 INPUT-AMOUNT.          01 HOLD-AMOUNT.
   02 FIELD-1   PIC XX.    02 FIELD-A   PIC 99.
   02 FILLER    PIC X.     02 FIELD-B   PIC 99.
   02 FIELD-2   PIC XX.    01 CALC-AMT REDEFINES HOLD-AMOUNT.
                               02 CALC-AMOUNT PIC 99V99.
```

ACCEPT Statement

The following coding appears in the Procedure Division:

```
      GO TO GET-AMOUNT.  
ERROR-CHECK.  
      MOVE FIELD-1 TO FIELD-A.  
      MOVE FIELD-2 TO FIELD-B.  
      IF CALC-AMOUNT IS NOT NUMERIC GO TO BAD-AMOUNT.  
      ADD CALC-AMOUNT TO FACTOR-X.  
      .  
      .  
      .  
BAD-AMOUNT.  
      DISPLAY "AMOUNT ENTERED INCORRECTLY. TRY AGAIN."  
GET-AMOUNT.  
      DISPLAY "ENTER AMOUNT. FORMAT EQUALS 99.99."  
      DISPLAY "SUPPLY LEADING ZERO IF REQUIRED."  
      ACCEPT INPUT-AMOUNT.  
ERROR-EXIT.  
      GO TO ERROR-CHECK.
```

CLOSE Statement

SEQUENTIAL ACCESS FILES

The CLOSE statement suspends or terminates the processing of files.

FORMAT

$$\begin{array}{l} \underline{\text{CLOSE}} \text{ file-name-1} \left[\begin{array}{l} \underline{\text{REEL}} \\ \underline{\text{UNIT}} \end{array} \right] \left[\text{WITH} \left\{ \begin{array}{l} \underline{\text{NO REWIND}} \\ \underline{\text{LOCK}} \end{array} \right\} \right] \\ \left[\text{file-name-2} \left[\begin{array}{l} \underline{\text{REEL}} \\ \underline{\text{UNIT}} \end{array} \right] \left[\text{WITH} \left\{ \begin{array}{l} \underline{\text{NO REWIND}} \\ \underline{\text{LOCK}} \end{array} \right\} \right] \right] \dots \end{array}$$

An OPEN statement must be performed for a file prior to the execution of a close statement for the same file. A CLOSE statement must be performed for every open file prior to execution of the STOP RUN statement.

Note: MPE/3000 closes any files left open at end-of-program.

Within the restrictions mentioned above, a file may be closed at any point within the program.

After a file is closed it is no longer available for processing unless it is opened again.

REEL/UNIT Option

The REEL/UNIT option documents the type of device assigned to the file. REEL specifies a tape device, specifically a magnetic tape. UNIT specifies any input or output devices. This compiler treats the REEL/UNIT option as comments.

LOCK Option

This compiler treats the LOCK option as a comment.

This parameter has no relationship to the COBOLLOCK and COBOLUNLOCK procedures. (See Section XI.)

CLOSE Statement

NO REWIND OPTION

The NO REWIND option applies only to magnetic tape files.

Normally, the CLOSE statement causes a magnetic tape file to be rewound. As an option, the user may specify that the tape remain in its current position when the CLOSE statement is executed. (NOTE: Currently this option has little practical application as the OPEN statement automatically positions all tapes at the load point before label checking. Therefore, use of this option may increase run time or even force the operator to unload the tape manually.)

RANDOM ACCESS FILES

The CLOSE statement suspends or terminates the processing of files.

FORMAT

CLOSE file-name-1 [WITH LOCK] [,file-name-2 [WITH LOCK]] . . .

All the rules for sequential access files apply for random access files. Both sequential and random files may be closed by the same CLOSE statement.

EXAMPLE

```
CLOSE TAPEIN, CARDIN, LISTING.
```

DISPLAY Statement

The DISPLAY statement allows the programmer to output low volume data without the need to define a file structure for that data. Typically, such output will consist of short messages to the operator.

FORMAT


$$\underline{\text{DISPLAY}} \left\{ \begin{array}{l} \text{literal-1} \\ \text{identifier-1} \end{array} \right\} \left[\begin{array}{l} \text{, literal-2} \\ \text{, identifier-2} \end{array} \right] \dots \left[\underline{\text{UPON}} \left\{ \begin{array}{l} \underline{\text{SYSOUT}} \\ \underline{\text{CONSOLE}} \\ \text{mnemonic-name} \end{array} \right\} \right]$$

If used, mnemonic-name must be specified in the Special-Names paragraph in the Environment Division and must assume the meaning of either SYSOUT or CONSOLE.

The CONSOLE is the operator's console. The SYSOUT device for a job run in the batch mode is the line printer; the SYSOUT device for a session (the job is run from a terminal in the time-sharing mode) is the user's terminal. When the UPON option is not specified, the compiler assumes the SYSOUT device.

In the format, each literal may be any figurative constant other than ALL. Only a single occurrence of the figurative constant is displayed.

Identifier may not be any Special Register other than TALLY, TIME-OF-DAY, or CURRENT-DATE.

Note: DISPLAY TIME-OF-DAY displays the edited format of TIME-OF-DAY, HH: MM: SS.

The DISPLAY statement converts the contents of identifiers defined as USAGE COMPUTATIONAL or USAGE COMPUTATIONAL-3 into a signed DISPLAY format similar to that used for punched cards. For signed numeric data, the statement generates the equivalent of an overpunched sign above the low-order digit. Thus the value -987 appears as 98P when output by the DISPLAY statement.

The amount of data transferred by the DISPLAY statement is controlled by the number of characters contained in identifier or literal. When the DISPLAY statement specifies more than one operand, the number of characters output is the sum of the characters contained in all the identifiers and literals named in the DISPLAY statement. When multiple operands are present, they are output in the sequence in which they appear in the DISPLAY statement.

The positioning of data output by the DISPLAY statement follows the rules for standard data positioning for alphanumeric data. However, the number of characters transferred depends on the sending field(s) rather than the physical limitations of the output device. The first character to be output is placed in the left-most character position of the output device. Data is transferred from left-to-right until all the sending data has been exhausted. Space padding is provided to the right of the significant data if needed. When the number of characters to be transferred exceeds the physical limitations of the output device, the effect is the same as using multiple DISPLAY statements. For example, if a single DISPLAY statement outputs 150 characters to the line printer, the effect is the same as if one DISPLAY statement output the first 132 characters, and a second DISPLAY statement output the remaining 18 characters. However, except for formatting considerations, the programmer need not be concerned about the physical aspects of output via the DISPLAY statement.

DISPLAY Statement

Special care must be taken when the SYSOUT device is used for the DISPLAY statement, and the program is running in the batch mode. If the program includes a printer file, that file and the DISPLAY statement share the line printer. In such a case, execution of a DISPLAY statement could interfere with the line count of a formatted report. Also, the DISPLAY statement and the WRITE . . . AFTER ADVANCING statement both cause the printer to advance the form and then print. However, a WRITE . . . BEFORE ADVANCING causes the printer to advance the form after printing. Therefore, it is possible to cause inadvertant overprinting by including the DISPLAY statement in a program that uses some form of the WRITE . . . BEFORE ADVANCING statement.

EXAMPLE

Assume that a program run in batch mode is to output a message indicating the number of error items rejected during the run. The program includes the following DISPLAY statements in its end-of-run housekeeping procedures. The data item ERR-COUNT contains the number of rejected items.

```
DISPLAY ERR-COUNT,  " ITEMS REJECTED THIS RUN" UPON CONSOLE.  
DISPLAY ERR-COUNT,  " ITEMS REJECTED THIS RUN".
```

The first DISPLAY statement outputs the message to the operator's console. The second statement outputs the message to the line printer. (When the UPON option is not specified, the SYSOUT device is used.)

OPEN Statement

The OPEN statement initiates the processing of files. This statement performs checking and/or writing of labels and other input/output operations. In addition, this statement provides exits to declarative user label routines.

FORMAT

$$\text{OPEN} \left\{ \begin{array}{l} \text{INPUT file-name-1} \left[\begin{array}{l} \text{REVERSED} \\ \text{WITH NO REWIND} \end{array} \right] \left[, \text{file-name-2} \left[\begin{array}{l} \text{REVERSED} \\ \text{WITH NO REWIND} \end{array} \right] \right] \dots \\ \text{OUTPUT file-name-3} \left[\text{WITH NO REWIND} \right] \left[, \text{file-name-4} \left[\text{WITH NO REWIND} \right] . \right] \dots \\ \text{I-O file-name-5} \quad \quad \quad \left[, \text{file-name-6} \right] \dots \end{array} \right\} \dots$$

Each of the choices (INPUT, OUTPUT, or I-O) can be specified only once in each OPEN statement. The I-O option applies only to mass storage files.

File-name must be the file-name assigned the SELECT and FD statements for the file, and not the system-file-name from the ASSIGN clause.

The REVERSED option requires hardware that is not supported by the HP 3000. This compiler treats the REVERSED option as a comment and issues a diagnostic message during the compilation.

If the external medium permits rewinding, execution of the OPEN statement positions the file at its beginning. Therefore, this compiler treats the WITH NO REWIND option as a comment and issues a diagnostic message during the compilation.

With the exception of files under control of a SORT statement, an OPEN statement must be executed for each file prior to the first READ, WRITE, or CLOSE statement for that file.

The programmer may change the usage of a magnetic file during program execution by closing the file and reopening it with a different usage. For example, the programmer may create a work file in one phase of his program, close the file, and then reopen it as an input file for additional processing.

The OPEN statement neither accesses nor releases the first data record; only READ or WRITE statements may perform these functions.

OPEN Statement

When the FILE-CONTROL paragraph of the Environment Division designates a file as optional and that file is not present for a given execution of the program, then the programmer must use a :FILE command to equate the file-name to \$NULL. \$NULL is a non-existent "ghost" file recognized by MPE. When the first READ statement for the file is encountered, the program receives an end-of-file indication, and the imperative statement specified in the AT END clause of the READ statement is executed.

Note: The :FILE command may also be used to equate an output file to \$NULL. This allows the user to suppress unneeded output from a file without modifying the program. WRITE statements associated with the file are accepted by MPE, but no physical output is performed. Notice, however, that this is a feature of the HP 3000 operating system and not the COBOL language.

The I-O option permits the opening of a mass storage file for both input and output operations. When an I-O file is opened, the software first checks the existing label, then exits to the user's declarative label procedure (if any), and then writes a new label over the previous label.

EXAMPLE

```
OPEN INPUT CARDFILE, TAPEFILE,  
      I-O DISCFILE,  
      OUTPUT PRINTFILE.
```

READ Statement

SEQUENTIAL ACCESS FILES

The READ statement for sequential files makes available the next logical record from an input file. Also, the statement allows performance of an imperative statement when the end of the file is detected.

FORMAT

READ file-name RECORD [INTO identifier]; AT END imperative-statement

The file-name used in a READ statement must also appear in the following portions of the program:

- A SELECT statement in the Environment Division.
- An FD statement in the File Section of the Data Division.
- An OPEN statement executed prior to the first READ statement for this file.

File-name in the format must not represent a sort-file.

Notice that the COBOL formats specify READ file-name and WRITE record-name. This is because a file may contain more than one record format. The READ statement cannot predict what may exist in external storage; it can only access the next logical record and present it to the programmer for processing. By contrast, the WRITE statement must know which record is to be output to the file.

When a file contains more than one type of logical record, these records automatically share the same storage area in memory. This is equivalent to an implicit redefinition of the area. Only the information that is present in the current record is available to the program.

When a file described with the OPTIONAL clause (in the Environment Division) is not present, then a :FILE command must be used to equate the file to \$NULL. The first READ statement immediately transfers control to the imperative-statement of the AT END clause.

The previous record is no longer available for processing after another READ statement for the same file is executed.

INTO Option

The INTO option may be used only when all records in the input file are of equal length.

READ Statement

A READ statement with the INTO option accesses a record from an input file and immediately moves the record into the workarea specified by the programmer. Any subscripting or indexing associated with the INTO identifier is evaluated after the record has been read and immediately before the move. The move follows the rules for a MOVE statement without the CORRESPONDING option. The new record is available in both the input area and in the workarea. The new record overlays whatever data was previously contained in the workarea.

AT END Clause

The AT END clause specifies an imperative-statement to be executed when the software detects the end of the file.

If more than one READ statement is used to access the same file, each READ must include the AT END clause; however, the imperative-statements coded for the AT END clauses need not be the same.

The AT END imperative-statement is executed when another READ is initiated for the file after the last logical record has already been accessed. The previous record is no longer available to the program when the AT END procedures are executed. After the AT END procedures are executed, another READ statement for the file may not be issued unless the file is first closed and then reopened.

RANDOM ACCESS FILES

The READ statement for random access files makes available for processing the specific record whose relative record number is contained in the ACTUAL KEY at the time the READ statement is executed. The statement also specifies an imperative-statement to be performed if the ACTUAL KEY is found to be invalid.

FORMAT

READ file-name RECORD [INTO identifier] ; INVALID KEY imperative-statement

This format of the READ statement is valid only when the SELECT statement for the file specifies a mass storage device such as a disc and contains the ACCESS MODE IS RANDOM clause and the ACTUAL KEY clause.

The file-name used in a READ statement must also appear in the following portions of the program:

- A SELECT statement in the Environment Division.
- An FD statement in the Data Division.
- An OPEN statement executed prior to the first READ statement for this file.

File-name must not represent a sort-file.

Prior to each execution of the READ . . . INVALID KEY statement, the programmer must move location data for the desired record into the ACTUAL KEY.

The READ . . . INVALID KEY statement implicitly performs the function of a SEEK statement unless a SEEK statement is executed for the desired record prior to the READ. The use of a SEEK statement prior to the READ may provide timing advantages if the mass storage device contains multiple-record blocks. When the SEEK is used, the relative record number of the desired record must be moved to the ACTUAL KEY prior to the execution of the SEEK. If this ACTUAL KEY is invalid, the SEEK is inoperative. If a subsequent READ . . . INVALID KEY statement requests the same record, then the imperative-statement in the INVALID KEY clause is executed.

INTO OPTION

The rules for the use of the INTO option are the same as for sequential access as explained above.

ACTUAL KEY FORMAT

The ACTUAL KEY must be a two-word binary integer which corresponds to a relative record number. Therefore, the ACTUAL KEY must be COMPUTATIONAL SYNCHRONIZED. The ACTUAL KEY may contain from five through nine decimal digits. Thus the PICTURE for the ACTUAL KEY must be as follows:

S9(5) through S9(9) COMP SYNC.

The ACTUAL KEY must be a data item within the File Section or the Working-Storage Section of the Data Division.

The value moved to the ACTUAL KEY must correspond to a relative record number. The method used to generate the relative record number is the responsibility of the programmer.

READ Statement

An ACTUAL KEY is invalid under the following conditions:

- It contains a negative value.
- It is greater than the highest possible relative record number in the file.
- The value moved to the ACTUAL KEY contains more digits than can be stored in the ACTUAL KEY.

When a READ . . . INVALID KEY statement is executed and the ACTUAL KEY is found to be invalid, the contents of the input buffer are unpredictable. However, the previous record is no longer available for processing.

RELEASE Statement

The **RELEASE** statement transfers records to the initial phase of a **SORT** operation.

FORMAT

RELEASE sort-record-name [**FROM** identifier]

A **RELEASE** statement may be used only within the range of an input procedure associated with a **SORT** statement. Sort-record-name must be the name of a logical record defined under the **SD** level entry associated with this sort-file.

When the **INPUT PROCEDURE** option is specified, at least one **RELEASE** statement must be included within the range of the input procedures.

After the **RELEASE** statement is executed, the logical record is no longer available. When control passes from the input procedures, sort-file contains all of those records transferred to it by execution of the **RELEASE** statement(s).

When the **FROM** option is used, the contents of identifier are moved to sort-record-name, and then the contents of sort-record-name are released to sort-file. This move follows the rules of the **MOVE** statement without the **CORRESPONDING** option. Following execution of the **RELEASE** statement, the data is still available in identifier, but not in sort-record-name. Sort-record-name and identifier must not refer to the same storage area.

EXAMPLE

For an example of the statements associated with the sort feature, see the **SORT** statement in this section.

RETURN Statement

The RETURN statement retrieves individual records in sorted order from the final phase of a sort operation.

FORMAT

RETURN sort-file-name RECORD [INTO identifier]
AT END imperative-statement

Sort-file-name must be described by an SD level entry in the Data Division.

When the OUTPUT PROCEDURE option is specified, at least one RETURN statement must be included within the range of the output procedures. The RETURN statement may be used only within the range of the output procedures associated with a SORT statement.

After a record is retrieved by the RETURN statement, the programmer must reference the record using data-names defined in the record descriptions associated with the SD level entry for this sort-file, unless the INTO option is specified. (Optionally, the user may MOVE the record into a work area of his choice and achieve the same result.)

When the INTO option is specified, the contents of the sort-record are moved into identifier, and the record is available in either the sort-record or the work-area named by identifier. The sort-record is moved according to the rules of the MOVE statement without the CORRESPONDING option. Identifier must be the name of a Working-Storage area or an output record area.

With this compiler, the INTO option may be used only when the sort-file contains just one type of record. Any indexing or subscripting associated with the identifier is evaluated after the record is retrieved, but before it is moved.

Imperative-statement in the AT END clause specifies the action to be taken after all records have been retrieved from sort-file. After execution of the imperative statement, no more additional RETURN statements may be executed within the current output procedure.

EXAMPLE

For an example of the statements associated with the sort feature, see the SORT statement in this section.

SEEK Statement

The SEEK statement initiates access to a mass storage device for subsequent reading.

FORMAT

SEEK file-name RECORD

The SEEK statement is valid only for input files whose SELECT statement includes the ACCESS MODE IS RANDOM and ACTUAL KEY clauses. The SEEK statement is treated as comments for output files.

The file named in the SEEK statement must be opened prior to the first SEEK statement for that file.

Prior to each execution of the SEEK statement, the relative record number of the desired record must be moved to the ACTUAL KEY associated with this file. If the ACTUAL KEY is found to be invalid, the SEEK statement is ignored. If a subsequent READ . . . INVALID KEY statement requests a record with the same key, then the imperative-statement in the INVALID KEY clause is executed.

The SEEK statement causes physical transfer of the block containing the desired record from the storage device into memory for access by a subsequent READ statement. The use of a SEEK statement prior to the READ may provide timing advantages. However, because the SEEK function is implicit in the READ statement, the SEEK statement is not mandatory.

Two SEEK statements for the same mass storage file may logically follow each other. However, it is poor programming practice to expend input-output time for records that are not accessed.

EXAMPLE

Following execution of the following statements, the block containing the record identified by REC-ID is placed in memory for subsequent access by a READ statement.

```
MOVE REC-ID TO ACT-KEY.  
SEEK DISCIN.
```


SORT Statement

The SORT statement creates a sort-file by executing input procedures or by transferring records from another file, sorts the records in the sort-file on a set of keys specified by the programmer, and in the final phase of the sort operation, makes available in sorted sequence each record from the sort-file to some output procedures or to an output file.

FORMAT

$$\begin{array}{l} \text{SORT file-name-1 ON } \left\{ \begin{array}{l} \underline{\text{DESCENDING}} \\ \underline{\text{ASCENDING}} \end{array} \right\} \text{ KEY data-name-1 [, data-name-2] . . .} \\ \\ \left[\text{ ; ON } \left\{ \begin{array}{l} \underline{\text{DESCENDING}} \\ \underline{\text{ASCENDING}} \end{array} \right\} \text{ KEY data-name-3 [, data-name-4] . . .} \right] \dots \\ \\ \left\{ \begin{array}{l} \underline{\text{INPUT PROCEDURE IS}} \text{ section-name-1 [THRU section-name-2]} \\ \underline{\text{USING}} \text{ file-name-2} \end{array} \right\} \\ \\ \left\{ \begin{array}{l} \underline{\text{OUTPUT PROCEDURE IS}} \text{ section-name-3 [THRU section-name-4]} \\ \underline{\text{GIVING}} \text{ file-name-3} \end{array} \right\} \end{array}$$

Every SORT statement must obtain records from either an input procedure or a USING file and must output records to either an output procedure or to a GIVING file. The sort-file itself is a temporary file which is deleted when the program that created it reaches end-of-job.

File-name-1 in the format must be described in an SD entry in the Data Division. Each of the data-names specified as keys for the SORT statement must represent data items described in records associated with file-name-1.

File-name-2 and file-name-3 must each be described in a File Description (FD) rather than a Sort-File Description entry (SD). The actual size of the logical record(s) described for file-names-1, -2, and -3 must be equal. If the data descriptions for the elementary items that make up these records are not identical, the programmer must ensure that all the records require the same amount of storage. Special care must be taken to be certain that different record descriptions do not cause different record lengths because of the insertion or deletion of slack bytes.

Neither file-name-2 nor file-name-3 may be open when the SORT statement is executed. The SORT statement itself performs all opening and closing operations required for any file named in the SORT statement. If any file named in the SORT statement is open when the statement is encountered, an object-time error will occur when the SORT attempts to open the already opened file.

The Procedure Division may contain more than one SORT statement. SORT statements may appear anywhere within the Procedure Division except in Declarative procedures or in the input and output procedures associated with any SORT statement.

ASCENDING/DESCENDING Options

The **ASCENDING** and **DESCENDING** options name the keys to be used for the sort and specify whether the records are to be sorted into ascending or descending order, respectively. Both options may be stated in the same **SORT** statement.

At least one **ASCENDING** or **DESCENDING** clause must appear in every **SORT** statement.

Each data-name represents a **KEY** and must be described in the records associated with file-name-1. These **KEY** items are subject to the following rules:

- Keys may not be variable length items.
- Keys may not contain an **OCCURS** clause, nor can they be subordinate to an entry described with the **OCCURS** clause.
- Where more than one record description is associated with the **SD** entry, the key items need only be described in one of the record descriptions. (They may be described as **FILLER** in subsequent record descriptions.) When the key items are described in more than one record, the data descriptions must be equivalent, and the key items must occupy the same positions in each record.

Keys are always listed from left to right in order of decreasing importance, regardless of whether they are ascending or descending.

Records are sorted into ascending or descending order on data-name-1, and then, within data-name-1, they are sorted into ascending or descending order on data-name-2, etc. Thus, in order to sort a file of sales records into sequence by ascending district number with the salesmen within each district ordered by decreasing sales, the following **SORT** statement might be used: **SORT SALES-FILE ON ASCENDING DISTRICT-NUMBER; DESCENDING TOTAL-SALES . . .** (Note: This statement requires the addition of an input procedure or **USING** file and an output procedure or **GIVING** file to be complete.)

The direction of the sort depends on the use of the **ASCENDING** or **DESCENDING** clauses as follows:

- When an **ASCENDING** clause is used, the sorted sequence ranges from the lowest value of the key to the highest value according to the rules for comparison of operands in a Relation Condition.
- When a **DESCENDING** clause is used, the sorted sequence ranges from the highest value of the key to the lowest value.

SORT Statement

Input/Output Procedures

The SORT statement must always obtain records from either a USING file or from an input procedure and must always release records to either a GIVING file or an output procedure.

INPUT PROCEDURES. The input procedure, if present, must consist of one or more sections that appear contiguously in the source program and must not form any part of an output procedure. The input procedures are identified by section-name-1 and, optionally, section-name-2 in the format.

The input procedures must contain at least one RELEASE statement in order to transfer records to the sort-file.

Input procedures (and output procedures) should be considered “slave” procedures designed exclusively for the use of the SORT statement. If the input procedures are not executed under the control of a SORT statement, the RELEASE statement will cause the job to abort. In addition, the input procedures may not contain any explicit transfers of control to points outside the input procedures; ALTER, PERFORM, and GO TO statements within the input procedures can refer only to procedure-names within the input procedures. The compiler does not check for such violations. Transfers that occur under the control of Declarative procedures do not violate this rule.

If an input procedure is specified, the SORT passes control to the procedure before sorting file-name-1. The compiler inserts a return mechanism at the end of the last section of the input procedure; when control passes the last statement in the input procedure, the records that have been released to file-name-1 are sorted.

OUTPUT PROCEDURES. The output procedure, if present, must consist of one or more sections that appear contiguously in the source program and must not form any part of an input procedure. The output procedures are identified by section-name-3 and, optionally, section-name-4 in the format.

The output procedures must contain at least one RETURN statement in order to access the sequenced records in sort-file.

The output procedure may consist of any procedures needed to select, modify, or copy the records that are being returned one at a time in sorted order from the sort-file.

Output procedures — like input procedures — should be considered “slave” procedures designed exclusively for the use of the SORT statement. If the output procedures are not executed under the control of the SORT statement, the RETURN statement will cause the job to abort. Like the input procedures, output procedures may not contain any explicit transfers of control outside the output procedures. Transfers that occur under the control of Declarative procedures do not violate this rule.

When an output procedure is specified, control passes to it after file-name-1 has been sequenced by the SORT statement. The RETURN statement(s) in the output procedure request the next record from the sorted file.

The compiler inserts a return mechanism at the end of the last section of the output procedure. When control passes the last statement in the output procedure, this mechanism provides for termination of the sort and then passes control to the next statement after the SORT statement.

USING/GIVING Option

Instead of using input or output procedures, the user may name a USING file or a GIVING file, respectively. In effect, the compiler generates appropriate input and/or output procedures for the named files. Also, if the files are associated with Declarative procedures, the compiler generates appropriate linkage to the Declarative procedures. For this compiler, the USING/GIVING option is more efficient than the INPUT/OUTPUT PROCEDURE option.

If the same file is specified as the USING file (file-name-2) and the GIVING file (file-name-3), a :FILE command must be used prior to running the program to specify that the file is a shared file. For example, suppose a program named APPLICN contains the following statements:

```

SELECT IFILE ASSIGN TO "DATA,DA".
SELECT SFILE ASSIGN TO "DATA,DA".

.
.
.
FD IFILE.
  (Description of IFILE.)
FD SFILE.
  (Description of SFILE.)

.
.
.
SORT SFILE ON ASCENDING KEY DATA-ONE USING IFILE GIVING IFILE.

.
.
.

```

To run this program, the following commands are used:

```

:FILE DATA; SHR
:RUN APPLICN

```

MPE Considerations

Normally the file descriptions compiled into the object program (using information from the SELECT, FD, and SD statements) provide all the information required to execute the object program. However, if the user wishes to override the file descriptions compiled into the COBOL program, he may do so by supplying MPE commands when the program is executed. These commands are effective only for the current execution of the program.

If the USING and GIVING file names in a SORT statement refer to the same file, a :FILE command must be used as described above. A :FILE command may also be used to specify the file

SORT Statement

size if a file to be sorted is not a disc file and contains more than 10,000 records. In this case, a :FILE command is used to specify the number of records in the SORT-FILE (the file named in the SD statement). For example, if a program contains the following statements:

```
SELECT SORT-FILE ASSIGN TO "SORT,DA".
SELECT NAME-FILE ASSIGN TO "INTAPE,UT".
.
.
.
SD SORT-FILE RECORD CONTAINS 100 CHARACTERS.
.
.
SORT SORT-FILE DESCENDING KEY DATA-ONE USING NAME-FILE GIVING
OUT-FILE.
```

and the INTAPE magnetic tape file contains 15,000 records, the following :FILE command should be used before the program is executed:

```
:FILE SORT; DISC=15000 ← maximum number of records SORT is expected to process
```

This technique should also be used if an input procedure passes more than 10,000 records to SORT.

An alternate method for specifying the file size is to include the file size parameter in the SELECT statement defining the SORT-FILE (SD file-name).

For example:

```
SELECT SORT-FILE ASSIGN TO "SORT,DA,,,15000".
```

If neither of these techniques is used and the records to be sorted exceed 10,000, an error condition results and the SORT program aborts.

A disc file containing more than 10,000 records does not require a file size since SORT program-matically determines the file size.

SORT Statement

IDENTIFICATION DIVISION.
PROGRAM-ID.
 GRADEREPORT.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. HP-3000.
OBJECT-COMPUTER. HP-3000.
INPUT-OUTPUT SECTION.
FILE-CONTROL.

 SELECT CARD-FILE ASSIGN TO "\$STDIN,UR".
 SELECT PRINT-FILE ASSIGN TO \$STDLIST,UR".
 SELECT SORT-FILE ASSIGN TO "SORT,DA".

DATA DIVISION.
FILE SECTION.

FD CARD-FILE RECORD CONTAINS 80 CHARACTERS
 LABEL RECORD IS OMITTED
 DATA RECORD IS STUDENT-RECORD.

01 STUDENT-RECORD.
 02 STUDENT-NAME-1 PIC X(26).
 02 NUMERICAL-SCORE-1 PIC 999.
 02 FILLER PIC X(51).

SD SORT-FILE RECORD CONTAINS 80 CHARACTERS.

01 SORT-REC.
 02 STUDENT-NAME PIC X(26).
 02 NUMERICAL-SCORE PIC 999.
 02 FILLER PIC X(51).

FD PRINT-FILE RECORD CONTAINS 133 CHARACTERS
 LABEL RECORD IS OMITTED.

01 PRINT-RECORD.
 :
 :
 :

PROCEDURE DIVISION.
THE-FIRST SECTION.
SORT-PROCEDURE.

 SORT SORT-FILE DESCENDING KEY NUMERICAL-SCORE USING CARD-FILE
 OUTPUT PROCEDURE MAIN-LINE.
 STOP RUN.

MAIN-LINE SECTION.
BEGIN-JOB.
 OPEN OUTPUT PRINTFILE.
HEADER-ROUTINE

 :
 :
 :

```
READ-ROUTINE.  
    RETURN SORT-FILE AT END GO TO END-OF-FILE-ROUTINE.  
WRITE-ROUTINE.  
    .  
    .  
    WRITE PRINT-RECORD AFTER ADVANCING 2 LINES.  
    GO TO READ-ROUTINE.  
END-OF-FILE-ROUTINE.  
    CLOSE PRINT-FILE.  
EXIT-PARA.  
    EXIT.
```

In the sample program, notice that there are no OPEN, READ, WRITE, or CLOSE statements for any of the files associated with the SORT statement. All these functions are performed by the SORT statement.

USE Statement

The USE statement specifies procedures for input-output label and error handling procedures that are in addition to the standard procedures provided by the operating system. Notice that the USE statement itself is never executed. Instead, it defines the conditions calling for the execution of the procedures associated with the USE.

A USE statement, when present, must immediately follow a section header in the Declarative portion of the Procedure Division and must be terminated by a period followed by a space. The remainder of the section must consist of one or more paragraphs which are the procedures to be executed under the conditions defined by this USE statement.

FORMAT 1

USE AFTER STANDARD ERROR PROCEDURE ON

$$\left(\begin{array}{l} \text{file-name-1 [, file-name-2] . . .} \\ \underline{\text{INPUT}} \\ \underline{\text{OUTPUT}} \\ \underline{\text{I-O}} \end{array} \right)$$

FORMAT 2

USE $\left\{ \begin{array}{l} \underline{\text{BEFORE}} \\ \underline{\text{AFTER}} \end{array} \right\}$ STANDARD $\left[\begin{array}{l} \underline{\text{BEGINNING}} \\ \underline{\text{ENDING}} \end{array} \right]$ $\left[\begin{array}{l} \underline{\text{REEL}} \\ \underline{\text{FILE}} \\ \underline{\text{UNIT}} \end{array} \right]$ LABEL PROCEDURE ON

$$\left(\begin{array}{l} \text{file-name-1 [, file-name-2] . . .} \\ \underline{\text{INPUT}} \\ \underline{\text{OUTPUT}} \\ \underline{\text{I-O}} \end{array} \right)$$

Within a USE procedure, there must not be any statement which transfers control outside the current declarative procedure. However, in the non-declarative portion of the program, a PERFORM statement may name procedures in declarative sections. Only PERFORM statements may refer to procedures within a declarative section (Other statements, such as GO TO might interfere with the return mechanisms of the declarative procedures.)

The USE statement must never name a sort-file. This means that no file defined with an SD level entry may be named in a USE statement. However, USING and GIVING files named in a SORT statement may be named in a USE statement since these files are defined with FD level entries.

Operation of Format 1



Declarative procedures associated with Format 1 of the USE statement are executed following the completion of the standard input-output error handling routine.

The user must exercise care when the declarative procedures initiate an I/O operation for the same file that caused control to be passed to the declarative procedures. A subsequent error for the file would again pass control to the declarative procedures, resulting in a possible program loop.

The compiler inserts an exit mechanism after the last statement in the section. This statement must be executed in order to return to the main program.

Operation of Format 2

With this compiler Declarative procedures associated with Format 2 are executed for beginning user labels only. System labels are maintained by MPE and are never made available to the user. Therefore, the compiler treats the BEFORE, ENDING, UNIT, and REEL options as comments. If these options are specified, the compiler issues a diagnostic message.

User labels must be 80 characters in length. Up to eight user labels may be specified. Only one label is available at a time, and all labels for a given file share the same area of memory. When multiple Format 2 USE statements are included in the program, these statements must not cause simultaneous requests for execution. For example, if one USE statement specifies the INPUT option, and a second USE statement explicitly names a file, then that file must not be an input file. In such a case, both USE statements would request control at the same time.

The transfer of control to procedures associated with Format 2 of the USE statement varies according to the type of file as follows:

- Input files — control is transferred after the first user label has been read.
- Output files — control is transferred after the OPEN procedures have been performed. The user may create his first user label at this time.
- Input-Output files — control is transferred after the first user label has been read.

USE Statement

When a file is named explicitly in a Format 2 USE statement, the file must not be described with the LABEL RECORDS ARE OMITTED option. When the INPUT, OUTPUT, or I-O option is specified, the USE procedures do not apply to files described with the LABEL RECORDS ARE OMITTED clause.

The compiler inserts an exit mechanism after the last statement in each Declarative procedure. For output and input-output files, this mechanism automatically writes out the current user label. The user never reads or writes labels himself. With a single exception, all logical program paths within a Declarative procedure must lead to the exit point. However, when processing files that have more than one user label, a special exit may be specified by the statement GO TO MORE-LABELS.

The function of this statement varies according to the type of file as follows:

- Input files — Control returns to the software which reads an additional user label, and then transfers control back to the first statement of the USE procedure. The last statement in the USE procedure must be executed in order to terminate label processing.
- Output files — Control returns to the software which writes out the current user label, and then transfers control back to the first statement of the USE procedure so that additional user labels can be created. The last statement in the USE procedure must be executed in order to terminate label processing.
- Input-Output files — Control returns to the software which writes out the current label and then reads the next label. The software then transfers control back to the first statement of the USE procedure. The last statement in the USE procedure must be executed in order to write out the last user label and to terminate label processing.

If the GO TO MORE-LABELS statement does not appear in the USE procedures, then only one user label is processed for the file.

EXAMPLE

The following coding terminates the program because of input errors and issues rerun instructions to the operator.

```
PROCEDURE DIVISION.  
DECLARATIVES.  
BAD-DATA SECTION. USE AFTER STANDARD ERROR PROCEDURE ON TAPEIN.  
    DISPLAY "UNREADABLE MASTER TAPE — ABORTING PROGRAM".  
    DISPLAY "TO RECREATE, RERUN YESTERDAYS WORK".  
    STOP RUN.  
END DECLARATIVES.  
HOUSEKEEPING SECTION.  
    OPEN INPUT TAPEIN.  
    etc.
```

WRITE Statement

The WRITE statement releases a logical record to an output file. It also provides for vertical positioning within printed reports. For mass storage files such as disc, the WRITE statement specifies an imperative-statement to be performed if the ACTUAL KEY is found to be invalid.

This discussion of the WRITE statement will be in three parts, one for general rules and sequential files other than printer files, one for printer files, and one for random access files.

FORMAT (SEQUENTIAL ACCESS FILES)

WRITE record-name [FROM identifier-1]

Notice that the COBOL formats specify WRITE record-name and READ file-name. This is because a file may contain more than one record format. The WRITE statement must know which record is to be released to a file. By contrast, the READ statement can only access a file and present whatever record happens to be accessed.

Record-name must be the name of a record defined in the File Section of the Data Division. Record-name may be qualified.

An OPEN statement must be executed for the file prior to executing the first WRITE statement for that file.

FROM Option

If the FROM option is specified, the data is moved from the work area specified by identifier-1 to the output buffer. This move follows the rules for the MOVE statement without the CORRESPONDING option. Identifier-1 can be in the File Section or the Working-Storage Section of the Data Division. Any indexing or subscripting associated with identifier-1 is evaluated before the move.

Identifier-1 and record-name in the format must not refer to the same storage area.

Following execution of the WRITE . . . FROM statement, the record is no longer available in the output buffer area, but is still available in the work area specified by identifier-1.

FORMAT (PRINTER FILES)

WRITE record-name [FROM identifier-1]

$$\left[\left\{ \begin{array}{l} \underline{\text{BEFORE}} \\ \underline{\text{AFTER}} \end{array} \right\} \text{ADVANCING} \left\{ \begin{array}{l} \text{identifier-2 LINES} \\ \text{integer LINES} \\ \text{mnemonic-name} \end{array} \right\} \right]$$

The WRITE statement and FROM option rules stated above apply to this format of the WRITE statement.

WRITE Statement

Identifier-2 must be defined as an elementary numeric item without any positions to the right of the implied decimal point.

Integer must be a numeric literal from zero through 63 lines and must have no positions to the right of the decimal point.

The software is capable of advancing a printer form a maximum of 63 lines. If either integer or the contents of identifier-2 is greater than 63, the software divides the number by 63 and then uses the remainder as the forms advancement specification. For example, if integer specifies 77 lines, the form is advanced 14 lines. (77 divided by 63 equals 1 with a remainder of 14.)

Mnemonic-name must be a programmer-assigned name for the system functions TOP and/or NO SPACE CONTROL. The programmer assigns names to these functions in the Special-Names paragraph of the Environment Division. These system functions are individually explained below:

- TOP — The TOP functions causes the printer to skip to channel one of the carriage control tape. Channel one is usually aligned with the first line of a form; therefore, this function is normally used to skip to the top of a new page. However, the exact operation of this function depends on the carriage control tape mounted on the printer when the program is executed.
- NO SPACE CONTROL — The NO SPACE CONTROL function suspends normal carriage control conventions and causes the printed output to be single-spaced.

The effect of this function is as though all WRITE statements for the printer file included the AFTER ADVANCING 1 LINES clause.

BEFORE/AFTER ADVANCING OPTION

The BEFORE/AFTER ADVANCING option controls paper advancement for printer files. This option may be used only with printer files.

If identifier-2 is used, the printer page is advanced a number of lines equal to the current value contained in the identifier. If integer is specified, the page is advanced a number of lines equal to integer. If a mnemonic-name is specified, the paper is positioned according to the rules for the system functions TOP or NO SPACE CONTROL.

Care must be taken when BEFORE and AFTER ADVANCING clauses are intermixed as this may result in overprinting.

If the printer file is assigned to the standard listing device, \$STDLIST, at compile time (via the ASSIGN clause), the carriage-control default is CCTL; in all other cases, the default is NOCCTL. (CCTL and NOCCTL are explained in *MPE Commands Reference Manual*.) This action is illustrated by the following two general cases. In CASE I, the printer file is assigned to \$STDLIST at compile time. In CASE II, the printer file is assigned to the formal-file-designator at compile time, and the following MPE/3000 :FILE commands are issued at run time:

: FILE L; DEV = LP [; { CCTL
NOCCTL }]

: FILE formal - file - designator = *L

The results are as follows:

CASE I:

NO ADVANCING	The first character in the printer buffer is used as a carriage - control character.
ADVANCING	The full record is printed.

CASE II:

:FILE At Run-Time WRITE	NOCCTL or Default	CCTL
NO ADVANCING	The full record is printed, with single spacing.	The first character in the printer buffer is used as a carriage-control character.
ADVANCING	The full record is printed, with single spacing.	The full record is printed, with spacing as indicated in ADVANCING statement.

FORMAT (RANDOM ACCESS FILES)

WRITE record-name [FROM identifier]; INVALID KEY imperative-statement

The rules for record-name and the FROM option as stated above apply to this format of the WRITE statement.

This format of the WRITE statement may be used only with mass storage files such as disc. In addition, any file associated with this format of the WRITE statement must include the ACTUAL KEY clause in its SELECT statement in the Environment Division.

Prior to each execution of the WRITE . . . INVALID KEY statement for a random access file, the programmer must move location data for the record into the ACTUAL KEY. The system then uses the contents of ACTUAL KEY as a relative record number which determines the record's relative address within the file.

The SEEK function is implicit in the WRITE . . . INVALID KEY statement. When the statement is executed, the system locates the desired record address and then writes the record at that address. The system does not check to see if another record already occupies the desired location. Therefore, when inserting records into a random access file, it is a good practice to read the desired location before writing the new record. If the desired location contains a valid record, an alternate ACTUAL KEY should be assigned to the new record.

ACTUAL KEY Format

The ACTUAL KEY must be a double-word binary integer which corresponds to a relative record number. The USAGE of the ACTUAL KEY must be COMPUTATIONAL SYNCHRONIZED. The ACTUAL KEY may contain from five through nine digits. Thus the PICTURE for ACTUAL KEY must be as follows:

S9(5) through S9(9) USAGE COMP SYNCHRONIZED

The ACTUAL KEY must be a data item within the File Section or the Working-Storage Section of the Data Division.

The value moved to the ACTUAL KEY must correspond to a relative record number. The method used to generate the relative record number is the responsibility of the programmer.

WRITE Statement

An **ACTUAL KEY** is invalid under the following conditions:

- It contains a negative value.
- It is greater than the highest possible relative record number in the file.
- The value moved to the **ACTUAL KEY** contains more digits than can be stored in the **ACTUAL KEY**.

When a **WRITE . . . INVALID KEY** statement is executed and the **ACTUAL KEY** is found to be invalid, the record is not released to the file and is still available in the output buffer. The software transfers control to the imperative-statement specified for the **INVALID KEY** clause.

SECTION XI

Locking and Unlocking Files

Files may be locked and unlocked from COBOL programs by calling the COBOLLOCK and COBOLUNLOCK procedures provided in the COBOL Library. In order to use these procedures on a file, the file must be opened with dynamic locking enabled. There are two ways to enable dynamic locking:

- The LOCKING parameter of the \$CONTROL command enables dynamic locking for all files specified in an FD entry. The command may appear any place in the source program. (Refer to Appendix B).
- Alternatively, dynamic locking may be enabled for specific files by including the L parameter with the system-file name clause of the SELECT statement. For example, SELECT ORDERFL ASSIGN TO "FILEZ,,,DISC,500,,L" specifies dynamic locking for the file with formal file designator FILEZ and File Description file name ORDERFL. (Refer to SELECT (Sequential Files) in Section IV.)

COBOLLOCK

To lock a file that has been opened with dynamic locking enabled, use this procedure.

FORMAT

CALL "COBOLLOCK" USING file-name, lock-cond, cond-code.

- The file-name parameter specifies which file is to be locked. The name must be the same as the file name used in the File Description Entry.
- The lock-cond parameter specifies either conditional or unconditional locking. The parameter must be a data item defined with PIC S9(4) USAGE COMP. If the value of this data item is odd (TRUE), locking takes place unconditionally. If the file cannot be locked immediately, the process suspends until it can be locked. If the value of the data item is even (FALSE), control returns immediately if the file cannot be locked.
- The cond-code parameter returns a condition code indicating the results of the attempt to lock the file. The settings for this code are:
 - 1 (CCL) Request denied because this file was not opened with dynamic locking or the request was to lock more than one file and the calling process does not possess the Multiple RIN capability. (Multiple RIN capability is specified when the program is prepared by including the MR parameter in the capability list. To use the MR parameter, the log-on account and user name must have been granted Multiple RIN capability by the system manager and account manager.)
 - 0 (CCE) Request granted: file is locked.
 - +1 (CCG) Request denied because the file has been locked by another process. This value is not returned if the lock-cond parameter is odd (TRUE).

EXAMPLE

CALL "COBOLLOCK" USING UPDATE-FILE, LOCK-CONDU, COND-CODEU.

COBOLUNLOCK

This procedure may be used to unlock a file.

FORMAT

CALL "COBOLUNLOCK" USING file-name, cond-code.

- The file-name parameter specifies which file is to be unlocked. The name must be the same as the file name used in the File Description Entry.
- The cond-code parameter returns a condition code indicating the results of the request to unlock the file. The settings for this code are:
 - 1 (CCL) Request denied because the file was not opened with dynamic locking or the file is not open.
 - 0 (CCE) Request granted; file is unlocked.
 - +1 (CCG) Request denied because the file has not been locked by the calling process.

EXAMPLE

CALL "COBOLUNLOCK" USING UPDATE-FILE, COND-CODEU.

PART 4
Appendices

APPENDIX A

Using the COBOL/3000 Compiler

To the Multiprogramming Executive Operating System (MPE/3000), the COBOL compiler is simply a process which is allocated file devices and executed like any other process. The compiler, in turn, accepts source statements as input data, generates object code for the source statements, and produces a listing of the program with diagnostic messages inserted if the program contains errors. The compiler-generated object program is not immediately executable: It must first be prepared by the MPE/3000 Segmenter. The preparation step formats the object program so that it can be manipulated efficiently by MPE/3000 when the program is executed. After preparation, the program is ready for execution.

Any one of the three MPE/3000 commands summarized below may be used to invoke the COBOL compiler. These commands may be entered as part of the input stream in the batch mode, or from the user's terminal in a session.

MPE COMMAND	FUNCTION OF COMMAND
:COBOL	This command invokes the COBOL compiler, which then compiles the source program, produces a program listing with diagnostic messages (if any), and generates an object program. Because this command requires a minimum of computer time, it is especially useful for syntax checking or recompilations of programs with extensive changes.
:COBOLPREP	This command causes MPE to execute two separate processes: First, the COBOL compiler is invoked and operates as described above; then, MPE calls the preparation step which prepares the object program for execution. Because of the extra time required for the preparation step, this command should not be used for syntax checking or for programs that contain known errors.
:COBOLGO	This command causes compilation, preparation, and execution of the program. Any files required for execution of the object program must be specified before this command is invoked. This command is especially useful for testing purposes and in educational applications.

Certain MPE commands duplicate functions of the commands listed on the previous page. For example, a program compiled via the :COBOL command can be prepared using the MPE :PREP command. A program which has been compiled and prepared can be run using the MPE :RUN command. Also, a program which has been compiled can be prepared and executed using the MPE :PREPRUN command. Each of these commands is explained on the following pages.

Note: After the compiler has been invoked by one of the MPE commands, the user may control a number of compile-time options by using compiler subsystem commands. Subsystem commands and their formats are fully explained in Appendix B of this manual.

After a user initiates an MPE/3000 batch job or session, he can compile, prepare, and execute his COBOL/3000 programs through various MPE/3000 commands. These commands require references to files used for input and output. For instance, a command to compile a program references a file that contains source program input, another used for program listing output, and a third used for object program output. Because of the importance of file references as command parameters, some of the rules for specifying files are introduced before the compilation, preparation, and execution commands themselves. The complete rules concerning files are discussed in *MPE Commands Reference Manual*.

REFERENCING FILES

When compiling, preparing, or executing programs, the user can designate the files to be used for input and output in either of two ways:

1. By naming the files as positional parameters (called *actual file designators*) in the compilation, preparation, or execution command.
2. By omitting optional parameters from the compilation, preparation, or execution command, allowing MPE/3000 to assign standard *default file designators* for input or output.

Specifying Files as Command Parameters

The user can name the following types of files as parameters in a compilation, preparation, or execution command.

- System-Defined Files
- User Pre-Defined Files
- New Files
- Old Files



SYSTEM-DEFINED FILES. System-defined file designators indicate those files that MPE/3000 uniquely identifies as standard input/output devices for a job/session. They are referenced by the following actual file designators:

Actual File Designator	Device/File Referenced
\$STDIN	A filename indicating the standard job or session input device (that from which the job or session is initiated). For a job, this is typically a card reader. For a session, this indicates a terminal. Input data images in the \$STDIN file should not contain a colon in column 1, since this indicates the end of the source input. Use the :EOD command to indicate the physical end of a source program. The same command is used to indicate the end of a data file.
\$STDINX	Equivalent to \$STDIN, except that MPE/3000 command images (those with a colon in column 1) encountered in a data file, are read without indicating the end of data. (However, the commands :JOB, :DATA, :EOJ, and :EOD are exceptions that always indicate the end-of-data and are <i>never</i> read as data.)

Actual File Designator**Device/File Referenced****\$STDLIST**

A filename indicating the standard job or session listing device corresponding to the particular job or session input device being used. (For each potential job/session input device, the user with MPE/3000 System Supervisor capability designates a corresponding job/session listing device during system configuration.) The job or session listing device is customarily a printer for a batch job and a terminal for a session.

\$NULL

The name of a non-existent "ghost" file that is always treated as an empty file. When referenced as an input file by a program, that program receives only an end-of-data indication upon first access. When referenced as an output file, the associated write request is accepted by MPE/3000 but no physical output is actually performed. Thus, \$NULL can be used to discard unneeded output from a running program.

USER PRE-DEFINED FILES. A user pre-defined file is any file that was previously defined or redefined in a :FILE command, as discussed in *MPE Commands Reference Manual*. In other words, it is a back-reference to that :FILE command. In compilation, preparation, or execution commands, the actual file designator of this file is written as

**formaldesignator*

formaldesignator = The name used in the *formaldesignator* parameter of the :FILE command, preceded by an asterisk (*)

NEW FILES. New files are files that have not yet been created, and are being created/opened for the first time by the current batch job or timesharing session. New files can have the following actual file designators:

Actual File Designator**File Referenced****\$NEWPASS**

A temporary disc file that can be automatically passed to any succeeding MPE/3000 command within the same job/session, which references it by the filename \$OLDPASS. (Passing is explained in later examples.) Only one such file with this designation can exist in the job/session at any one time. (When \$NEWPASS is closed, its name is automatically changed to \$OLDPASS, and any previous file named \$OLDPASS in the job/session is deleted.)

Actual File Designator**File Referenced***filereference*

Any other new file to which the user has access. Unless the user specifies otherwise, this is a temporary file, residing on disc, that is destroyed upon termination of the program. If closed as a *job/session temporary file*, however, it is saved until the end of the job/session, when it is purged. If closed as a permanent file, it is saved until purged by the user. Typically, this format consists of a file name containing up to eight alphanumeric characters, beginning with a letter. In addition, other elements (such as a groupname, account name or lockword) can be specified. The complete rules governing the *filereference* format are presented in *MPE Commands Reference Manual*.

OLD FILES. Old files are existing files currently-resident in the system. They may be named by the following designators:

Actual File Designator**File Referenced**

\$OLDPASS

The name of the temporary file last closed as \$NEWPASS.

filereference

Any other old file to which the user has access. It may be a *job/session temporary file* created in this or a previous program in the current job/session, or a permanent file saved by any program in any job/session. The format is the same as *filereference*, noted above.

INPUT/OUTPUT SETS. All of the preceding actual file designators can be classified as those used as input parameters (*Input Set*) and those used as output parameters (*Output Set*). These sets, referred to frequently throughout this manual, are defined as follows:

Input Set

\$STDIN

The job/session input device.

\$STDINX

The job/session input device with commands allowed.

\$OLDPASS

The last file passed.

\$NULL

A constantly-empty file that will produce an end-of-file indication whenever it is read.

**formaldesignator*

A back-reference to a previously-defined file.

filereference

A file name, and perhaps account and group names and a lockword.

Output Set

\$STDLIST	The job/session listing device.
\$OLDPASS	The last file passed.
\$NEWPASS	A new temporary file to be passed.
\$NULL	A constantly-empty file.
<i>*formaldesignator</i>	A back-reference to a previously-defined file.
<i>filereference</i>	A file name, and perhaps account and group names and a lockword.

Specifying Files By Default

When a user omits an optional file parameter from a compilation, preparation, or execution command, MPE/3000 assigns one of the members of the input or output set by default. The file designator assigned depends on the specific command, parameter, and operating mode, as noted later in this section.

COMPILING/PREPARING/EXECUTING PROGRAMS

User source programs written in COBOL/3000 undergo the operational steps outlined below. In most cases, the details of these steps will be invisible to the user during normal compilation and execution. When necessary, however, the user can advance through each of these steps independently, completely controlling the specifics of each event along the way.

1. The source language program is *compiled* (translated by a compiler) into binary form and stored as one or more relocatable binary modules (RBM's) in a specially-formatted disc file called a user subprogram library (USL). There is one RBM for each program unit. (A program unit is a self-contained set of statements that is the smallest divisible part of a program or subprogram. In COBOL/3000, this is a section.) In USL form, however, the program is not yet executable.

The MPE Segmenter can be used to create new USL files or to move RBM's from one segment to another. For more information see below.

2. The USL is then *prepared* for execution by a process running the MPE/3000 Segmenter. This process binds the RBM's from the USL into linked, re-entrant code segments organized on a program file. (In preparation, only *one* USL can be used for RBM input to the program file.) At this point, the special segment for the input of user data (the stack) is also initially defined.
3. The program file is *allocated and executed*. In allocation, a process binds the segments from the program file to referenced external segments from the system, account, or group segmented library. Next, the first code segment to be executed, and the associated stack are moved into main memory and execution begins.

A particular program can be run by many user processes at the same time, because code in a program is inherently sharable.

When a request to execute a program is encountered, but before execution actually begins, MPE/3000 checks to determine if the file space used has exceeded the specified account and group limits. If a limit is exceeded, the program enters execution but is aborted if and when it requires allocation of a new disc *extent* (the integral number of contiguous disc sectors by which file space is allocated). If no limit is exceeded prior to execution, the program enters execution and continues to run even if the filespace limit is exceeded during execution; however, if a limit is exceeded during execution, a message indicating this is printed.

The compilation, preparation, and execution of a program can be requested by individual commands or by a single command that performs all three operations. In the following pages, all of these commands are described.

Manipulating USL Files with the MPE Segmenter

The Segmenter can be used to arrange Relocatable Binary Modules in a User Subprogram Library for presentation at program preparation time. However, the following restrictions apply:

- The only program units that may precede the outer block are dynamic COBOL subprograms and other procedures which do not use global storage.**
- These types of program units, if included, must follow the outer block:
 - non-dynamic COBOL subprograms
 - SPL procedures with TRACE, EXTERNAL, or OWN variables
 - FORTRAN procedures with DATA, COMMON, LABELED COMMON, or TRACE variables.

When a program unit or segment is added to the USL, the Segmenter inserts it at the top of the list. Therefore, the order in which program units are added to the USL file with the Segmenter COPY command should be carefully planned. This is also true when using the Segmenter NEWSEG command to move an RBM into a different segment within a USL file.

Refer to the *MPE Segmenter Reference Manual* for complete information about these commands. Here are a few examples to illustrate how USL files can be modified.

EXAMPLES

The USL file shown below was produced by compiling a main program containing two Sections in the Procedure Division.

```
USL FILE DOCUSL.COBOL.LANG
MAIN
  MAIN'          40  OB  A C N  Main Program
                    ↑ Outer Block Designation
                    | Initialization
SECONDSEC02'
  SECONDSEC02'  31  P  A C N R
FIRSTSEC00'
  FIRSTSEC00'   33  P  A C N R
FILE SIZE      377600
DIR. USED      166          INFO USED          377
DIR. GARB.     0          INFO GARB.           0
DIR. AVAIL     37212       INFO AVAIL.        337401
```

****Note:** *The base address for the main program data area must be DB+0 (the beginning of the global area). Since COBOL does not allocate Primary DB storage, the Secondary DB storage begins at DB+0, and the outer block must be the first program allocating Secondary DB storage. For more information about DB storage see the Systems Programming Language Reference Manual.*

If a non-dynamic subprogram has been compiled into another USL file, it can be moved into the master USL file shown above. The first step is to COPY the two subprogram segments into the master USL file using the Segmenter.

```

USL DOCUSL
AUXUSL DOCAUX
COPY SUB      (Copy subprogram initialization)
COPY SUB'     (Copy dynamic subprogram body)
LISTUSL
USL FILE DOCUSL.COBOL.LANG
SUB'
  SUB'          145 P   A C N R
SUB
  SUB           466 P   A C N R
MAIN
  MAIN'         40 OB   A C N
SECONDSEC02'
  SECONDSEC02'  31 P   A C N R
FIRSTSEC00'
  FIRSTSEC00'   33 P   A C N R

```

However, this leaves the USL file in an improper condition since a non-dynamic subprogram precedes the Outer Block. There are two ways to correct this situation. The first way is to move the Outer Block to the first segment by using a NEWSEG command as shown below.

```

NEWSEG SUB'.MAIN'
PURGERBM SEGMENT,MAIN
LISTUSL
USL FILE DOCUSL.COBOL.LANG
SUB'
  MAIN'         40 OB   A C N
  SUB'          145 P   A C N R
SUB
  SUB           466 P   A C N R
SECONDSEC02'
  SECONDSEC02'  31 P   A C N R
FIRSTSEC00'
  FIRSTSEC00'   33 P   A C N R
FILE SIZE      377600
DIR. USED      331          INFO USED      2045
DIR. GARB.     7          INFO GARB.     0
DIR. AVAIL.    37047      INFO AVAIL.    335733

```

The second way is to use the NEWSEG command to create a new segment for the Outer Block as shown below.

```

NEWSEG NEWMAIN,MAIN'
LISTUSL
USL FILE DOCUSL.COBOL.LANG
NEWMAIN
  MAIN'         40 OB   A C N
SUB'
  SUB'          145 P   A C N R
SUB
  SUB           466 P   A C N R
SECONDSEC02'
  SECONDSEC02'  31 P   A C N R
FIRSTSEC00'
  FIRSTSEC00'   33 P   A C N R

```

The USL file can now be prepared.

The COPY command can be used without any further action to add a dynamic subprogram or a procedure in another language (such as SPL or FORTRAN) which does not use global storage.

```

COPY SUBA'          (Copy subprogram initialization)
COPY SUBA          (Copy dynamic subprogram body)
LISTUSL
USL FILE DOCUSL.COBOL.LANG
SUBA
  SUBA          327 P   A C N R
SUBA'
  SUBA'        211 P   A C N R
NEWMAIN
  MAIN'        40  OB  A C N
SUB'
  SUB'         145 P   A C N R
SUB
  SUB          466 P   A C N R
SECONDSEC02'
  SECONDSEC02' 31  P   A C N R
FIRSTSEC00'
  FIRSTSEC00'  33  P   A C N R

```

The NEWSEG command can be used to combine initialization procedures, main program modules and subprogram compilations.

```

NEWSEG SUB,SUB'
PURGERBM SEGMENT,SUB'
NEWSEG SUBA',SUBA
PURGERBM SEGMENT,SUBA
LISTUSL
USL FILE DOCUSL.COBOL.LANG
SUBA'
  SUBA          327 P   A C N R
  SUBA'        211 P   A C N R
NEWMAIN
  MAIN'        40  OB  A C N R
SUB
  SUB'         145 P   A C N R
  SUB          466 P   A C N R
SECONDSEC02'
  SECONDSEC02' 31  P   A C N R
FIRSTSEC00'
  FIRSTSEC00'  33  P   A C N R

```

A main program segment can be moved into another segment.

```

NEWSEG SUBA',FIRSTSEC00'
PURGERBM SEGMENT,FIRSTSEC00'
LISTUSL
USL FILE DOCUSL.COBOL.LANG
SUBA'
  FIRSTSEC00'  33  P   A C N R
  SUBA          327 P   A C N R
  SUBA'        211 P   A C N R
NEWMAIN
  MAIN'        40  OB  A C N
SUB
  SUB'         145 P   A C N R
  SUB          466 P   A C N R
SECONDSEC02'
  SECONDSEC02' 31  P   A C N R
FILE SIZE      377600
DIR. USED      461      INFO USED      3154
DIR. GARB.     37      INFO GARB.      0
DIR. AVAIL.    36717   INFO AVAIL.    334624

```

Relocatable Libraries and Segmented Libraries

COBOL program units can be put into a Relocatable Library (RL) or a Segmented Library (SL) using the Segmenter. For more information about how to do this refer to the *MPE Segmenter Reference Manual*. Here is a summary of the types of program units that can reside in these types of libraries.

Relocatable Libraries may contain:

- COBOL subprograms
- Procedures coded in other languages
- Subprogram initialization procedures
- Main program modules

Relocatable Libraries may *not* contain:

- Outer block program units (Main program initialization)

Segmented Libraries may contain:

- Dynamic subprograms
- Subprogram initialization routines
- Procedures in other languages without global data.

Segmented Libraries may *not* contain:

- Outer block program units
- Non-dynamic subprograms
- Procedures in other languages with global data.

:COBOL

To compile a source-language program, the user enters a command of the following format:

`:_COBOL [textfile] [,uslfile] [,listfile] [,masterfile] [,newfile]]]]`

- textfile* Input file (device) from which the source program is to be read. This can be any ASCII-coded file from the input set. If omitted, the default file \$STDIN (the current input device) is used. (Optional parameter.)
- uslfile* The name of the USL file on which the object program is written. This can be any binary file from the output set. If omitted, the default file \$OLDPASS is assigned, unless no file is in the passed state (in which case \$NEWPASS is used). If the default value is not used, the user must create the USL file before he compiles. He does this in one of three ways:
1. By saving a USL file (through the MPE/3000 command :SAVE), created by a previous compilation where the default value was used for the *uslfile* parameter.
 2. By building the USL (through the MPE/3000 segmenter subsystem command — BUILDUSL).
 3. By creating a new file of USL type (through the MPE/3000 command :BUILD, with the decimal value 1024 used for the *filecode* parameter).
- (Optional parameter.)
- listfile* The name of the file to which the program listing is to be generated. This can be any ASCII file from the output set. If omitted, the default file \$STDLIST (typically the terminal in a session or the printer in a batch job) is assigned. (Optional parameter.)
- masterfile* The name of a file to be optionally merged with *sourcefile* and written onto a file named *newfile*, as discussed in the manuals covering compilers. If *masterfile* is omitted, the default file \$NULL is assigned. (Optional parameter.)
- newfile* The file on which the records from *sourcefile* and *masterfile* are optionally merged. When *newfile* is omitted, the default file \$NULL is assigned. (Optional parameter.)

An additional file, *copyfile*, may also be specified. When used, *copyfile* supplies pre-written text to be copied into, and compiled with, the source program. If such a file is used and its name is not COPYLIB (the default file designator for this file), then the name must be equated to COPYLIB through an MPE/3000 :FILE command issued prior to the :COBOL command. Further information on the *copyfile* and the copy-library can be found in Section VIII of this manual.)

EXAMPLES

The following command compiles a COBOL/3000 source program, entered as a batch job through the card reader (job input device), into an object program on a USL file named \$NEWPASS. It also transmits listing output to the line printer (job list device.) (If the next command is one to prepare an object program, and no other input file is designated as a command parameter, \$NEWPASS is passed to it as an input file re-named \$OLDPASS. Note that a file can only be passed between commands or programs within the same job or session.)

```
:COBOL
```

The following job illustrates the passing of a file between two successive compilations. In this job, the :COBOL command compiles a program onto the USL file \$NEWPASS (because the default value for *uslfile* is taken, and no pass file presently exists). The :SPL command compiles another program onto the same USL, passed to this command under the redesignation \$OLDPASS. (When no passed file exists, \$NEWPASS is used; when a passed file exists, that file (\$OLDPASS) is used.) After the second compilation, \$OLDPASS contains RBM's from both compilations:

```
:JOB
:COBOL
    (COBOL SOURCE PROGRAM)
    .
    .
:EOB
:SPL
    .
    .
    (SPL SOURCE PROGRAM)
:EOB
    .
    .
:EOJ
```

The following command, entered during a session, compiles a COBOL/3000 source program residing on the disc file SOURCE into an object program on the USL file OBJECT, with a program listing generated on the file LISTFL.

```
_:COBOL SOURCE, OBJECT, LISTFL
```

:COBOLPREP

To compile and prepare for execution a source-language program, the user enters a command in the following format. (The USL file created during compilation is a temporary file passed directly to the preparation mechanism; it is not accessible by the user.)

`:COBOLPREP [textfile] [, [progfile] [, [listfile] [, [masterfile] [, newfile]]]]]`

textfile Input file (device) from which the source program is read. This can be any ASCII-coded file from the input set. If omitted, the default file \$STDIN (the current input device) is used. (Optional parameter.)

progfile The name of the program file onto which the prepared program segments are to be written. This can be any binary file from the output set. The user must create this program file (unless the default is taken). He does this in one of two ways:

1. By creating a new file of program-file type (through the MPE/3000 :BUILD command with the decimal value 1029 used for the *filecode* parameter).
2. By specifying a non-existent file in the *progfile* parameter of the :COBOLPREP command, in which case the system creates a new file of the correct size and type.

If omitted, the default file \$NEWPASS is assigned. (Optional parameter.)

listfile The name of the file to which the program listing is written. This can be any ASCII file from the output set. If omitted, the default file assigned is \$STDLIST (usually the terminal in a session or the printer in a batch job). (Optional parameter.)

masterfile }
newfile } The same meanings described under *compilation*.

Note: If a copyfile with a name other than COPYLIB is used, then its name must be equated to COPYLIB through an MPE/3000 :FILE command issued prior to the :COBOLPREP command.

EXAMPLES

The following command compiles and prepares a COBOL/3000 program entered through the card reader (job input device) as a batch job. The resulting program file is named \$NEWPASS, and the program listing is printed on the line printer (job list device). (If the next command is one to execute the program, the file \$NEWPASS is referenced in the execute command under the name \$OLDPASS.)

:COBOLPREP

:COBOLPREP

The next command, entered during a session, compiles and prepares a COBOL/3000 source program input from a source file name **SFILE** into a program file named **\$NEWPASS**. The resulting program listing is printed at the terminal (session listing device).

_:COBOLPREP SFILE

:COBOLGO

To compile, prepare, and execute a program, a command of the following format is entered. (This command creates temporary USL and program files that are not accessible by the user.)

:COBOLGO [textfile] [, [listfile] [, [masterfile] [, newfile]]]]

where:

<i>textfile</i>	Input file (device) from which source program is read. This can be any ASCII-coded file from the input set. If omitted, the default file \$STDIN (the current input device) is assigned. (Optional parameter.)
<i>listfile</i>	The name of the file to which the program listing is transmitted. This can be any ASCII file from the output set. If omitted, the default file assigned is \$STDLIST (normally the terminal for sessions or printer for batch jobs). (Optional parameter.)
<i>masterfile</i> <i>newfile</i> }	The same meanings described under <i>compilation</i> .

Note: If a copyfile with a name other than COPYLIB is used, then its name must be equated to COPYLIB through an MPE/3000 :FILE command issued before the :COBOLGO command.

EXAMPLES

The command shown below compiles, prepares, and executes a COBOL/3000 program entered as part of a batch job through the card reader (job input device). The program listing is printed on the line printer (job list device).

:COBOLGO

The following command, entered during a session, compiles, prepares, and executes a COBOL/3000 program residing in the disc file SOURCE and transmits the program listing to the terminal:

:COBOLGO SOURCE

:PREP

If a user's source program has been compiled onto a USL file, he can prepare it for execution with the :PREP command. (When writing this command, recall that keyword parameters can appear in any order after the positional parameters.)

_:PREP uslfile,progfile

[;ZERODB]
[;PMAP]
[;MAXDATA=segsz]
[;STACK=stacksz]
[;DL=dlsz]
[;CAP=caplist]
[;RL=filename]

<i>uslfile</i>	The name of the USL file (from the input set) on which the program has been compiled. (Required parameter.)
<i>progfile</i>	The name of the program file onto which the prepared program segments are to be written. This can be any binary file from the output set. The user must create this program file. He does this in one of two ways: <ol style="list-style-type: none">1. By creating a new file of program-file type (through the MPE/3000 :BUILD command, with the decimal value 1029 used for the <i>filecode</i> parameter).2. By specifying a non-existent file in the <i>progfile</i> parameter of the :PREP command or in the <i>progfile</i> parameter of the segmenter subsystem command —PREPARE (in which case a file of the correct size and type is created). (Required parameter.)
<i>ZERODB</i>	An indication that the initially-defined user-managed (DL-DB) area, and uninitialized portions of the DB-Q (initial) area, will be initialized to zero. If this parameter is omitted, these areas are not affected. (Optional parameter.)
<i>PMAP</i>	An indication that a descriptive listing of the prepared program will be produced on the job/session listing device. If this parameter is omitted, the listing is not produced. (Optional parameter.)
<i>segsz</i>	Maximum stack area (Z-DL) size permitted, in <i>words</i> . This parameter is included if the user expects to change the size of the DL-DB or Z-DB areas during process execution. If omitted, MPE/3000 assumes that he will not change these areas. (Optional parameter.)
<i>stacksz</i>	The size of the user's initial local data area (Z-Q (initial)) in the stack, in <i>words</i> . This overrides the <i>stacksz</i> estimated by the segmenter, which applies if the <i>stacksz</i> parameter is omitted. (The default is a function of estimated stack requirements for each program unit in the program. Since it is difficult for the system to predict the behavior of the stack at run time, the user may want to override the default by supplying his own estimate with <i>stacksz</i> .) (Optional parameter.)

:PREP

dlsize The DL-DB area to be initially assigned to the stack. This area is mainly of interest only in programmatic applications, and is discussed in detail in Section II. If the *dlsize* parameter is omitted, a value estimated by the segmenter applies. (Optional parameter.)

caplist The *capability-class attributes* associated with the user's program; specified as two-character mnemonics. If more than one mnemonic is specified, each must be separated from its neighbor by a comma. The mnemonics are

IA = Interactive access	}	Standard Capabilities
BA = Local batch access		
PH = Process Handling		
DS = Data Segment Management		
MR = Multiple resource management		
RT = High-Priority (BS) subqueue		
CR = Very-High Priority (AS) subqueue		
PM = Privileged-mode operation		
NS = Non-standard subqueue access		

The user who issues the :PREP command can only specify capabilities that he himself possesses (through assignment by the system or account manager). If the user does not specify any capabilities, the IA and BA capabilities (if possessed by user himself) will be assigned to this program. (Optional parameter.)

filename The name of a relocatable procedure library (RL) to be searched to satisfy external references during preparation (this can be any permanent file of type RL). It need not belong to the log-on group, nor does it have a reserved, local name. This file yields a single segment that is incorporated into the segments of the program file prepared. If *filename* is omitted, no library is searched. (Optional parameter.)

EXAMPLES

The following command prepares a program from the USL file named USEFILE and stores it in a program file named PROGFILE. The optional parameters will be those assigned by default. The program's capability-class attributes will be the standard capabilities associated with the preparing user.

```
_:PREP USEFILE, PROGFILE
```

The next command will accomplish the same function as the previous command, except that the prepared program will be listed, a stacksize of 500 words will be established, and the program will be assigned only the batch-access capability.

```
_:PREP USEFILE, PROGFILE; PMAP; STACK=500; CAP=BA
```

:PREPRUN

Programs compiled in USL files can be prepared *and* executed with the :PREPRUN command. This prepares a temporary program file (not accessible by the user) and then executes the program in that file. The command format is:

```
_:PREPRUN uslfile[,entrypoint]  
    [_];NOPRIV  
    [_];PMAP  
    [_];LMAP  
    [_];ZERODB  
  
    [_];MAXDATA=segsz  
    [_];PARM=parameternum  
    [_];STACK=stacksize  
    [_];DL=dlsz  
    [_];RL=filename  
    [_];LIB=library  
    [_];CAP=caplist
```

where:

- | | |
|-------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>uslfile</i> | The name of the USL file (from the input set) on which the program has been compiled. (Required parameter.) |
| <i>entrypoint</i> | The program entry-point where execution is to begin. This may be the primary entry point of the program, or any secondary entry point in the program's outer block. If the parameter is omitted, the primary entry point is assigned by default. (Optional parameter.) |
| <i>NOPRIV</i> | A declaration that the program will be placed in a <i>non-privileged</i> mode; this parameter is intended for programs prepared with the privileged-mode capability. Normally, program segments containing privileged instructions are executed (run in privileged mode) <i>only</i> if the program was <i>prepared</i> with the privileged-mode (PM) capability-class. (A program containing legally-compiled privileged code, placed in the non-privileged mode, aborts when an attempt is made to execute it.) If NOPRIV is specified in the :PREPRUN command, all program segments are placed in the <i>non-privileged</i> mode, with NOPRIV overriding the PM capability class. (Library segments, however, are not affected, since their mode is determined independently.) (Optional parameter.) |
| <i>PMAP</i> | An indication that a listing describing the <i>prepared</i> program will be produced on the job/session listing device. If this parameter is omitted, the prepared program is not listed. (Optional parameter.) |
| <i>LMAP</i> | An indication that a listing of the <i>allocated</i> (loaded) program will be produced on the job/session listing device. If this parameter is omitted, the allocated program is not listed. (Optional parameter.) |

:PREPRUN

- ZERODB* An indication that the initially-defined user-managed (DL-DB) area, and initialized portions of the DB-Q-initial area, will be initialized to zero. If this parameter is omitted, these areas are not affected. (Optional parameter.)
- segsiz* Maximum stack area (Z-DL) size permitted. This parameter is included if the user expects to change the size of the DL-DB or Z-DB areas during process execution. If omitted, MPE/3000 assumes that he will not change these areas. (Optional parameter.)
- parameternum* A value that can be passed to the user's program as a general parameter for control or other purposes; when the program is executed, this value can be retrieved from the address Q (initial)-4, where Q (initial) is the Q-address for the outer block of the program. The value can be an octal number or a signed or unsigned decimal number. If *parameternum* is omitted, the Q (initial)-4 address is filled with zeros. (Optional parameter.)
- stacksize* The initial size of the user's initial local data area in the stack, in words, as described in the discussion of the :PREP command. (Optional parameter.)
- dlsiz* The initial DL-DB area size, as described in the discussion of the :PREP command. (Optional parameter.)
- filename* The name of a relocatable program library to be searched to satisfy external references during preparation, as described in the discussion of the :PREP command. (Optional parameter.)
- library* The order in which applicable segmented procedure libraries are searched to satisfy external references during allocation, where:
- G = Group library, followed by account public library, followed by system library.
- P = Account public library, followed by system library.
- S = System library.
- If no parameter is specified, *S* is assumed. (Optional parameter.)
- caplist* The capability-class attributes associated with the user's program, as described in the discussion of the :PREP command. (Optional parameter.)

EXAMPLES

The following command prepares and executes a program on the USL file USEF, with no special parameters declared. (All default values apply.)

```
_:PREPRUN USEF
```

The next command prepares and executes a program on the USL file UBASE, beginning execution at the entry-point RESTART, declaring a stacksize of 500 words, and specifying that the library LIBA will be searched to satisfy external references:

```
:PREPRUN UBASE, RESTART; STACK=500; RL=LIBA
```

:RUN

Programs that have been compiled and prepared, and that therefore exist on program files, can be executed by the :RUN command, entered as follows:

```
_:RUN progfile[,entrypoint]  
    [;NOPRIV]  
    [;LMAP]  
    [;MAXDATA=segsz]  
    [;PARM=parameternum]  
    [;STACK=stacksize]  
    [;DL=dlsz]  
    [;LIB=library]
```

<i>progfile</i>	The name of the program file (from the input set) that contains the prepared program. (Required parameter.)
<i>entrypoint</i>	The program entry-point where execution is to begin. This may be the primary entry-point of the program or any secondary entry-point in the program's outer block. If this parameter is omitted, the primary entry-point (where execution normally begins) is assigned by default. (Optional parameter.)
<i>NOPRIV</i>	A declaration that the program will be placed in <i>non-privileged</i> mode; this parameter is intended for use by programs prepared with the privileged-mode capability. Normally, program segments containing privileged instructions are executed (run in privileged mode) <i>only</i> if the program was <i>prepared</i> with the privileged-mode (PM) capability-class. (A program containing legally-compiled privileged code, placed in the non-privileged mode, aborts when an attempt is made to execute it.) If NOPRIV is specified in the :RUN command, all program segments are placed in the <i>non-privileged</i> mode, with NOPRIV overriding the PM capability class. (Library segments, however, are not affected, since their mode is determined independently.) (Optional parameter.)
<i>LMAP</i>	An indication that a listing of the <i>allocated</i> (loaded) program will be produced on the job/session listing device. If this parameter is omitted, the allocated program is not listed. (Optional parameter.)
<i>segsz</i>	Maximum stack area (Z-DL) size permitted, as described in the discussion of the :PREP command. (Optional parameter.)
<i>parameternum</i>	A value that can be passed to the user's program as a general parameter for control or other purposes; when the program is executed, this value can be retrieved from the address Q (initial)-4, where Q (initial) is the Q-address for the outer block of the program. The value can be an octal number or a signed or unsigned decimal number. If <i>parameternum</i> is omitted, the Q (initial) -4 address is filled with zeros. (Optional parameter.)

<i>stacksize</i>	The initial size of the user's initial local data area in the stack, in words, as described in the discussion of the :PREP command. (Optional parameter.)
<i>dlsize</i>	The initial DL-DB area size, as described in the discussion of the :PREP command. (Optional parameter.)
<i>library</i>	The order in which applicable segmented procedure libraries are searched to satisfy external references during allocation, where: G = Group library, followed by account public library, followed by system library. P = Account public library, followed by system library. S = System library. If no parameter is specified, <i>S</i> is assumed. (Optional parameter.)

If values for *segsize*, *stacksize*, and *dlsize* are explicitly specified in the :RUN command, they override such values assigned at preparation time (which are recorded in the program file). If these parameters are omitted, the values recorded in the program file take effect.

EXAMPLE



The following command executes a program existing on the program file PROG3, with no special parameters specified. (All default values are used.)

```
_:RUN PROG3
```

The next command executes a program existing on the program file PROG4, beginning at the entry-point SECLAB. This program, prepared with privileged-mode capability, will run in non-privileged mode.

```
_:RUN PROG4, SECLAB; NOPRIV
```


APPENDIX B

Compiler Subsystem Commands

In general, compiler options such as source input merging, listing format specification, or warning message suppression are determined by default settings assigned by the compiler. However, the user can override these settings and select different options by issuing *compiler subsystem commands*. These commands take effect only after access to the compiler is established. They are directed only to the compiler and are not effective during object program execution.

Compiler subsystem commands differ in both function and format from compiler language source statements, and thus are not considered true COBOL/3000 statements. (However, the compiler subsystem commands accepted by COBOL/3000 conform to the general formats for all other HP 3000 language translators. For each function performed by more than one translator, the same command name is used; in most cases, the same command parameters also apply. This feature helps users familiar with one translator subsystem to use another.)

SYNTAX AND FORMAT

In describing the syntax and format of compiler subsystem commands, these conventions are used for consistency and clarity:

1. Entries that always appear exactly as shown are designated by UPPERCASE CHARACTERS.
2. Variable entries (such as class names) are indicated by lowercase characters.
3. Optional information is indicated by [brackets]. (However, the user does not enter these brackets as part of the commands.) Where one member in a group of entries *may* be selected and entered by the user, the entire group is surrounded by [brackets].
4. Where one member in a group of entries *must* be selected and entered by the user, the group is surrounded by {braces }.
5. An item or group of items, within brackets or braces, that may be repeated an indefinite number of times is followed by ellipses (. . .).

The general form of a compiler subsystem command is:

`[$] commandname [parameterlist]`

The first dollar-sign (\$) is required, and identifies the command as a compiler subsystem command. It must be the first character in the text portion of the record containing the command. For commands directed to COBOL/3000, this dollar sign must appear in Position 7.

The second \$, optional, suppresses transmittal of the command to *newfile* (if a *newfile* is created during compilation). (For commands directed to COBOL/3000, this second \$ must appear in Position 8.)

The *commandname* specifies the function requested. It follows the first \$ (or second \$, if present), with no intervening spaces.

The optional *parameterlist*, if present, specifies various command options. The list is separated from the *commandname* by one or more spaces. Within the list, parameters are separated from each other by commas, optionally followed and/or preceded by spaces. The parameterlist may continue through Position 72 of the source record on which it appears. (Positions 73 through 80 of COBOL/3000 source records comprise the identification field which is not part of the compiler subsystem command.)

The sequence field (Positions 1 through 6) of a record containing a compiler subsystem command also is not part of the command; it may, however, be used for sequence-checking the record during editing and merging operations, as described later.

Note: Only upper-case letters, and numbers and special characters are used in entering compiler subsystem commands; when lower-case letters are input as part of a command, the compiler interprets them as their upper-case equivalents (except when the lower-case letters are contained within character strings as defined below.)

Parameters

Within the *parameterlist*, a parameter may be any of the following four items:

- A character string.
- A symbolic name
- A keyword.
- A keyword with a subparameter.

A *character string* consists of a quotation mark (“) that denotes its beginning, optionally followed by one or more alphanumeric characters, followed by another quotation mark that terminates the string. Blank characters (spaces) may be included in the string. Quotation marks within the string are written as double quotation marks (two adjacent quotation marks, “ ”) to distinguish them from the quotation marks that begin and end the string. A character string consisting only of beginning and terminating quote marks is called an *empty string*. (Note that quotation-mark delimiters in *subsystem commands* are not changed by the QUOTE=' parameter in the \$CONTROL subsystem command, discussed later — the QUOTE=' parameter only affects delimiters in *COBOL/3000 source-language statements*.)

A *keyword* is a reserved word (with respect to a given command) that consists of a letter followed by one or more letters and/or digits.

A *keyword with subparameter* is a keyword followed by an equal sign, followed by a subparameter consisting of a character string, a symbolic name, or a number. (A *number* consists of one or more decimal digits.) The equal sign can be preceded and/or followed by one or more spaces. The general format is:

keyword=subparameter

Comments

Within any command, *comments* may also be included. A *comment* is generally used to document the purpose of coding or to make notations about program logic. A *comment* is not interpreted as part of the command, and has no effect upon compilation. It is syntactically treated as a space, and can appear in any of these locations:

- Following the *commandname*, separated from it by at least one space.
- Preceding or following any parameter in the *parameterlist*.

A *comment* cannot be embedded *within* a parameter; for instance, it cannot appear within a keyword, preceding or following an equal sign, or within a quoted string. Furthermore, a *comment* cannot be continued from one record to the next.

A *comment* can contain any characters from the ASCII character set. The *comment* must begin with two adjacent less-than signs (<<) and terminate with two adjacent greater-than signs (>>) as delimiters. (Since adjacent greater-than signs terminate a *comment*, they cannot appear within the *comment* itself.) The *comment* may continue through Position 72 of the record on which it appears. (This *comment* feature is provided in addition to the *comment* features of the COBOL/3000 language.)

EXAMPLES

The following examples illustrate various ways in which comments can be included in compiler subsystem commands.

1. Following the command name (\$PAGE) in a command with no parameterlist.

```
$PAGE <<PAGE EJECT,NO TITLE CHANGE.>>
```

2. Following the last parameter in a parameterlist (where the comment effectively appears as a separate field).

```
$SET X1=ON,X2=ON,X3=ON <<SWITCHES 1-3 ON.>>
```

3. Embedded within the parameterlist (preceding the last parameter):

```
$SET X1=ON,X2=ON, <<LAST SW OFF>> X3=OFF
```

Continuation Records

When the length of a command exceeds one physical record (source card or entry line), the user can enter an ampersand (&) as the last non-blank character of this record and continue the command on the next record (called a *continuation record*). The text portion of the continuation record, in turn, must begin with a dollar sign in Position 7. (Even when a command begins with the double dollar sign (\$\$), its continuation records still begin only with a single dollar sign.)

Note: A subsystem command record must never be separated from its continuation record by a COBOL/3000 source record; nor must a COBOL/3000 source record be separated from its continuation record by a subsystem command record.

In continuing a command onto another record, the user cannot divide a primary command element (a *command name*, *keyword*, *subparameter* (including quoted strings), or *comment*) — no primary element is allowed to span more than one line.

When a command containing one or more continuation records is encountered by the compiler, each continuation record is concatenated (beginning with the character following the \$) to the preceding record; each \$ and continuation ampersand is replaced by a space.

EXAMPLE

The following \$CONTROL command is continued onto a second record:

```
$CONTROL LIST, SOURCE, WARN, MAP, &  
$CODE, LINES=36.
```

It is interpreted as:

```
$CONTROL LIST, SOURCE, WARN, MAP, CODE, LINES=36
```

Even though a comment cannot be divided over more than one line, extensive commentary text requiring several lines can be entered by enclosing it within separate comments that each occupy one line.

EXAMPLE

The following \$CONTROL command includes commentary text spread over three lines.

```
$CONTROL NOWARN <<WARNING MESSAGES ON TRIVIAL ERRORS>>&  
$ <<WILL NOT BE LISTED. BUT MESSAGES ON>>&  
$ <<FATAL ERRORS WILL APPEAR.>>
```

Effects of Commands

A command does not take effect until all of its parameters have been interpreted. Thus, a command that suppresses source listing output will not affect the listing of any continuation records within the command itself. Parameters are interpreted from left to right. In some cases, parameters may be redundant or supersede previous parameters within the same command. In other cases, certain parameters are allowed by only once within a command.

EXAMPLE

In the following \$CONTROL command, the redundant parameters LIST and NOLIST each appear twice:

```
$CONTROL LIST, NOLIST, NOLIST, LIST
```

Because the final redundant parameter in any \$CONTROL command always takes effect, the above command is equivalent to:

```
$CONTROL LIST
```

COMMAND SUMMARY

A summary of the compiler subsystem commands for COBOL/3000 appears in Table B-1. (Only the *commandnames* are shown; the *parameterlists* are described later.)

Table B-1. Compiler Subsystem Command Summary

Command	Purpose
\$CONTROL	Restricts access to listfile; suppresses source text, object code, and symbol table listing; suppresses warning messages; sets maximum number of lines listed per page; sets maximum number of severe errors allowed; establishes delimiting character for non-numeric literals; places COBOL/3000 in subprogram mode; allocates data relative to Q-register.
\$IF	Interrogates software switches for conditional compilation.
\$SET	Sets software switches for conditional compilation.
\$TITLE	Establishes or changes page title on listing.
\$PAGE	Establishes or changes page title, and ejects page.
\$EDIT	Specifies editing options during merging (omitting sections of old source program and/or re-numbering sequence fields).

LISTING AND COMPILATION OPTIONS (\$CONTROL COMMAND)

When the user invokes the compiler without specifying compiler subsystem commands, several listing control, error message, and object-file formulation options take effect by default:

1. The compiler is given unrestricted access to *listfile*.
2. All source records (passed to the compiler by its editor) are listed unless the *listfile* and primary input file (normally the *textfile*) are assigned to the same terminal.
3. Warning messages are listed.
4. Listing of the symbol table is suppressed.
5. Listing of the object code generated is suppressed.
6. The number of lines appearing on each printed page (output to *listfile*) is a maximum of 60.

7. The maximum number of severe errors allowed before compilation is terminated is 100.
8. The quotation mark (“) is recognized as the character used to delimit non-numeric literals in COBOL/3000.
9. COBOL/3000 is invoked in the *program* (rather than *subprogram*) mode.
10. All data storage for object programs is allocated relative to the DB-register setting (rather than the Q-register setting).

The above default options can be overridden by entering the \$CONTROL compiler subsystem command. This command allows the user to restrict the compiler's access to the *listfile*; suppress source record listings; produce object code and symbol table listings; re-specify the maximum number of lines per printed page; and otherwise alter the normal compiler control options. The format of the \$CONTROL command is:

\$[\$]CONTROL parameterlist

Each parameter in the *parameterlist* specifies a different option: the options are described below. Unless otherwise noted, each parameter can appear in a \$CONTROL command placed anywhere in the source input. Each parameter remains in effect until *explicitly* cancelled by an opposing parameter (for example, NOLIST cancelling LIST), or until access to the compiler terminates. In any \$CONTROL command, the *parameterlist* (containing at least one parameter) is always required. Within the *parameterlist*, the parameters can appear in any order. In the descriptions below, default parameters are shown in **boldface** type.

Parameter	Option Requested
LIST	Allows the compiler unrestricted access to <i>listfile</i> , permitting the SOURCE, MAP, CODE, and LINES listing parameters described below to take effect when issued. The LIST parameter remains in effect until a \$CONTROL command specifying the NOLIST parameter (described below) is encountered. When neither LIST nor NOLIST is specified at the beginning of the compilation, LIST takes effect by default.
NOLIST	Allows <i>only</i> source records that contain errors, appropriate error messages, and subsystem initiation and completion messages to be written to the <i>listfile</i> . NOLIST remains in effect until a \$CONTROL command specifying LIST appears.
SOURCE	Requests listing of all source records input (as edited by the compiler's editor) while LIST is also in effect. When the compiler is invoked with <i>listfile</i> and the primary input file assigned to the same terminal, NOSOURCE is initially the default. In all other cases, SOURCE is initially in effect as the default.
NOSOURCE	Suppresses the listing of source text, cancelling the effect of any previous SOURCE parameter. NOSOURCE remains in effect until SOURCE is subsequently encountered.

Parameter	Option Requested
WARN	Permits the reporting of doubtful minor error conditions in the source input. These reports are transmitted to <i>listfile</i> in the form of warning messages. The WARN parameter remains in effect until a \$CONTROL command specifying the NOWARN parameter (described below) is encountered. When neither WARN nor NOWARN has been specified, WARN takes effect by default. <i>Note: NOLIST does not suppress warning messages — they are suppressed solely by NOWARN.</i>
NOWARN	Suppresses warning messages; cancels the effect of any previous WARN parameter. The NOWARN parameter remains in effect until a \$CONTROL command specifying WARN appears.
MAP	Requests printing of user-defined symbol following the listing of the source text (if LIST is in effect). The MAP parameter remains in effect until the NOMAP PARAMETER (described below) is encountered. When neither MAP nor NOMAP is specified at the beginning of the compilation, NOMAP is assumed by default. (See Appendix F for a description of the Symbol Table Map.)
NOMAP	Suppresses printing of user-defined symbols, cancelling effect of any previous MAP parameter. The NOMAP parameter remains in effect until MAP is again encountered.
CODE	Requests listing of object code generated following the listing of the source text (if LIST is in effect). The CODE parameter remains in effect until the NOCODE parameter (described below) is encountered. When neither CODE nor NOCODE is specified, NOCODE is assumed by default.
NOCODE	Suppresses listing of object code, cancelling effect of any previous CODE parameter. The NOCODE parameter remains in effect until a CODE parameter is again encountered.
LINES=nnnn	Limits lines printed on <i>listfile</i> to <i>nnnn</i> lines per page. Whenever the next line sent to <i>listfile</i> would overflow the line count (<i>nnnn</i>), the page is ejected and the standard page heading and two blank lines are printed at the top of the next page, followed by the line to be transmitted. (A page heading and its following two blank lines are counted against the total line count, <i>nnnn</i> .) The subparameter <i>nnnn</i> is an integer ranging from 10 to 9999. The LINES- <i>nnnn</i> parameter remains in effect until another LINES=nnnn parameter appears. If this parameter is omitted, the default value assigned is: 60 lines per page, for listing output through devices other than terminals. 32767 lines per page, for listings output through terminals.
LOCKING	Enables dynamic locking capability. (File can be locked and unlocked with COBOLLOCK and COBOLUNLOCK procedures. See Section XI).



Parameter	Option Requested
ERRORS=nnn	Sets the maximum number of severe errors allowed during compilation to <i>nnn</i> ; if this limit is exceeded, compilation terminates and the <i>uslfile</i> is unchanged. (If the limit specified has already been exceeded when this ERRORS=nnn parameter is encountered, compilation terminates at this point.) If the ERRORS=nnn parameter is omitted, <i>nnn</i> is set to 100 by default.
USLINIT	Initializes the <i>uslfile</i> to empty status prior to generation of object code. If no <i>uslfile</i> is specified by the user (and is thus supplied by the compiler through default), or if the user supplies a <i>uslfile</i> whose contents are obviously incorrect, the compiler automatically initializes the <i>uslfile</i> to empty status whether or not USLINIT is specified.
QUOTE={“,”}	Defines the character to be used for delimiting non-numeric literals in the following source text. This character may be either a quotation mark (“) or an apostrophe (’). If the QUOTE parameter is not specified, the quotation mark is assumed. The QUOTE parameter can appear anywhere in the source input; typically however, it is entered near the beginning of the program.
SUBPROGRAM	Places COBOL/3000 in subprogram mode, with data storage allocated by DB-relative addressing. This condition is initially disabled when COBOL/3000 is invoked. If it is to be enabled, this must be done before the COBOL program’s Data Division is encountered. Once COBOL/3000 is placed in subprogram mode, this mode cannot be disabled.
DYNAMIC	Places COBOL/3000 in the subprogram mode, with data storage allocated by Q-relative addressing (rather than by the DB-relative addressing). The DYNAMIC condition is initially disabled when COBOL/3000 is invoked. If it is to be enabled, this must be done before the COBOL program’s Data Division is encountered. Once enabled, this mode cannot be disabled.
DEBUG	Requests COBOL/3000 to generate a call, in the initialization segment of the user’s main program, to the MPE/3000 intrinsic XCONTRAP. This intrinsic allows the user to transfer control to the DEBUG utility program, at any time during execution of his program, by pressing the CONTROL and Y keys on the terminal. The compiler will <u>not</u> generate calls to DEBUG unless a \$CONTROL command with the DEBUG parameter appears somewhere in the user’s program.
BOUNDS	Requests COBOL/3000 to generate code for validation of table indexes and subscripts during execution of the program unit. The bounds condition is initially disabled when COBOL/3000 is invoked. If it is to be enabled, this must be done before the program’s Procedure Division is encountered.
CHECKSYNTAX	Checks the syntax of program without producing an object program.

EXAMPLES

The following \$CONTROL command requests unrestricted access to *listfile*; listing of all source text, symbol table information, and object code; suppression of warning messages (but not *error* messages); and use of the apostrophe as a delimiter for non-numeric literals. By default, the maximum number of lines per printed page is limited to 60, the maximum number of errors allowed is 100, the *uslfile* supplied by the user is not initialized to empty status, and COBOL/3000 is not placed in subprogram mode.

```
$CONTROL LIST, SOURCE, MAP, CODE, NOWARN, QUOTE='
```

The following \$CONTROL command illustrates the default values for the command parameters; it produces the same effect as if no \$CONTROL command were entered:

```
$CONTROL LIST, SOURCE, WARN, NOMAP, NOCODE, LINES=60, ERRORS=100, QUOTE=""
```

The next two examples illustrate use of the LIST and NOLIST parameters with respect to the same program. First, assume that the user compiles a program (in batch job mode) with a lengthy data division. During this first compilation, he wants to list all user-defined symbols in the program, all object code produced, and all source records but those contained in the data division. To do this, he specifies \$CONTROL commands as shown below:

Source Records	Effect
\$CONTROL MAP, CODE	Requests unrestricted access to list device (LIST option) and generation of source records (SOURCE option) by default, from this point on. Requests listing of user-defined symbols and object code at end of COBOL program.
IDENTIFICATION DIVISION	Begins Identification Division (and COBOL program).
⋮	
ENVIRONMENT DIVISION	Begins Environment Division
⋮	
\$CONTROL NOSOURCE	Suppresses listing of source records.
DATA DIVISION	Begins Data Division. (This record is not listed.)
⋮	
\$CONTROL SOURCE	Resumes listing of source records.
PROCEDURE DIVISION	Begins Procedure Division
⋮	
<Symbol Map>	Symbol map and object code requested in the first
<Object Code>	\$CONTROL command appear here.

When the user compiles his program a second time, he decides that he does not need to list any source code, nor the symbol map nor object code. This time, he precedes the COBOL program with a \$CONTROL command *restricting* access to the *listfile*, thus negating the effects of the other three \$CONTROL commands in the program. This action relieves the user of the need to remove or change these \$CONTROL records themselves.

Source Records	Effect
\$CONTROL NOLIST	Restricts access to list device, overriding the effects of the other three \$CONTROL commands in the program.
\$CONTROL MAP, CODE	None of these records are actually listed.
IDENTIFICATION DIVISION	
:	
:	
ENVIRONMENT DIVISION	
:	
:	
\$CONTROL NOSOURCE	
DATA DIVISION	
:	
:	
\$CONTROL SOURCE	
PROCEDURE DIVISION	
	No symbol map nor object code listing appears.

CONDITIONAL COMPILATION (\$IF COMMAND)

Generally, when the user submits a program to the compiler, he wants to compile the entire program. But, he may occasionally wish to compile only one or more portions of his program. He can request such *conditional compilation* by delimiting the source code to be compiled (or omitted from compilation) with a series of \$IF compiler subsystem commands. These \$IF commands interrogate any of ten compiler toggle switches, named X0 through X9, inclusive. (These switches are set *on* or *off* by the \$SET compiler subsystem command described later.) Thus, a \$IF command can direct the compiler to compile or ignore all source code between *this* \$IF command and the next \$IF command encountered, *if* a particular relation is true or false.

The format of the \$IF command is:

$$\$[\$]IF [X_n = \left\{ \begin{array}{l} \text{OFF} \\ \text{ON} \end{array} \right\}]$$

The parameters and their resultant options are:

Parameter	Option
Xn	The letter <i>X</i> , followed by a digit (<i>n</i>) from 0 to 9 that specifies the name of the switch to be tested — for example, X3. Spaces between <i>X</i> and <i>n</i> are not allowed.
OFF } ON }	The state that the switch is to be tested for in determining if the statements following the \$IF command are to be compiled. If the relation is <i>false</i> (the switch is not in the state specified by this parameter), the following source records (except \$EDIT, \$PAGE and \$TITLE commands) are ignored until another \$IF command is encountered. If the relation is <i>true</i> , succeeding source records are compiled normally.

A \$IF command can appear anywhere in the source text. The appearance of a \$IF command always terminates the influence of any preceding \$IF command. When a \$IF command is entered and no parameter list is included, the following text is compiled in the normal way but the effect of any *previous* \$IF command is cancelled. Regardless of whether conditional compilation occurs or is suspended as a result of a \$IF command,

1. \$EDIT, \$PAGE, \$TITLE and \$IF commands within the range of this \$IF command are interpreted and executed.
2. Normally-listable source text (regardless of whether or not it is compiled) is listed if the \$CONTROL command options LIST and SOURCE are in effect.

The *textfile-masterfile* merging operation and transmission of merged/edited text to the *newfile* are not affected by \$IF commands. (Merging and editing are described in the discussion of the \$EDIT command.)

An example illustrating use of the \$IF command is presented under the discussion of the \$SET command below.

SOFTWARE SWITCHES FOR CONDITIONAL COMPILATION (\$SET COMMAND)

When the compiler is invoked, all ten software toggle switches are initially turned *off*. The user can turn them *on* (and *off* again) by using the \$SET command.

$$\$[\$]SET [X_n = \left\{ \begin{array}{l} \text{OFF} \\ \text{ON} \end{array} \right\} [, X_n = \left\{ \begin{array}{l} \text{OFF} \\ \text{ON} \end{array} \right\}] \dots]$$

The parameters and options are:

Parameter	Option
Xn	The letter X followed by a digit (n) from 0 to 9 that specifies the name of the switch to be set.
OFF } ON }	The state to which the switch is to be set. Either OFF or ON (but not both) must be specified for each Xn parameter.

A \$SET command can appear anywhere in the source text. If a \$SET command that contains no parameter list is encountered, all ten switches are turned off. If more than one parameter appears in a \$SET command, each parameter must be separated from its predecessor by a comma.

EXAMPLE

In the following source text, switches X4 and X5 are set on and interrogated, with the results indicated by the comments:

```
      .
      .
$SET  X4=ON, X5=ON    <<SETS SWITCHES X4 AND X5 ON.>>
      .
      .
$IF   X4=ON          <<REQUESTS COMPILATION OF SOURCE BLOCK 1.>>
      .
      .
      (SOURCE BLOCK 1)
$IF   X5=OFF         <<REQUESTS THAT SOURCE BLOCK 2 BE IGNORED>>&
      .
      .
      <<BY CANCELLING PREVIOUS $IF COMMAND.>>
      .
      .
      (SOURCE BLOCK 2)
      .
      .
$IF   <<CANCELS PREVIOUS $IF COMMANDS SO THAT>>&
$     <<SOURCE BLOCK 3 IS COMPILED.>>
      .
      .
      (SOURCE BLOCK 3)
```

PAGE TITLE IN STANDARD LISTING (\$TITLE COMMAND)

On each page of output listed during compilation, a standard heading appears. Positions 29 through 132 of this heading are reserved for a title (usually describing the page's content), optionally specified with the \$TITLE compiler subsystem command.

```
[$]TITLE [string [,string] . . . ]
```

Each *string* parameter is a character string (bounded by quotation marks) that is combined with any other strings specified to form the title. In forming the title, the strings are stripped of their delimiting quotation marks; they are then concatenated from left to right. The entire parameter list can specify up to 104 characters, including spaces within the string but excluding delimiters and spaces between the strings. If the title contains fewer than 104 characters, the unused portion is filled to the right with spaces. If no *string* parameters are present in the \$TITLE command, or if no \$TITLE command (or \$PAGE command with title specification, discussed below) is entered, the title portion of the heading is blank. When a new \$TITLE command is encountered, it supersedes any previously specified title from that point on.

When a \$TITLE command is interpreted and the NOLIST parameter of the \$CONTROL command is in effect, title specification or replacement occurs even when the \$TITLE command appears within the range of an \$IF command whose relation is evaluated as false.

EXAMPLE

Consider the following \$TITLE command, occupying two lines of code:

```
$TITLE "THE PROGRAM TITLE IS ", &  
$"RUN III."
```

This command results in the following entry in the title field of the standard page heading:

```
THE PROGRAM TITLE IS RUN III.
```

PAGE TITLE AND EJECTION (\$PAGE COMMAND)

The user can specify a program title (as with the \$TITLE command) together with page ejection by entering the \$PAGE command. This allows varied listing formats. For example, individual sections of the program can be listed starting on a new page, and each section can have its own descriptive title. The \$PAGE command format is:

```
[$]PAGE [string[,string] . . . ]
```

Each string parameter has the same format, meaning, result, and constraints as in the \$TITLE command. If no parameter is present in the \$PAGE command, the current title (assigned by a previous \$TITLE or \$PAGE command) or a blank title (if no previous \$TITLE or \$PAGE command occurred) remains in effect.

Following title specification or replacement, if the LIST parameter of the \$CONTROL command is in effect, this action occurs:

1. The current page is ejected.
2. The standard page heading (including the new title) is printed, followed by two blank lines.

If no *string* was specified in the \$PAGE command and LIST is in effect, the current page is ejected and the standard page heading (including the old title) is printed, followed by two blank lines.

If LIST is not in effect, specified title replacement occurs, but no printing or page eject takes place. Any new title will appear, however, after LIST is requested.

The \$PAGE command itself is never listed.

SOURCE TEXT MERGING AND EDITING (\$EDIT COMMAND)

The user can request the following merging and editing operations:

1. Merge corrections or additional source text (on *textfile*) with an existing source program and commands (on *masterfile*) to produce a new source program and commands. This new input is compiled and optionally copied to *newfile*, which can be saved for recycling through an MPE/3000 :FILE command.
2. Check source-record sequence numbers for ascending order.
3. Omit sections of the old source program during merging.
4. Re-number the sequence fields of the records in the new, merged source program.

The editing done by the compiler is limited to linear source text modification. Extensive or more sophisticated editing is possible with the HP 3000 Text Editor, EDIT/3000.

Merging

Merging is requested by simply equating actual file names to the *textfile*, *masterfile*, and (optionally) *newfile* formal designators (COBTEXT, COBMAST, and COBNEW, respectively). This equating is done by using the MPE/3000 :COBOL command when the compiler is invoked. Use of this command is described in detail in *MPE Commands Reference Manual* and in Appendix A. An example appears below:

EXAMPLE

To specify merging of a *textfile* (maintenance file) TFILE with a *masterfile* MFILE, the user could enter the following :COBOL command. The merged source text is copied to the *newfile* NFILE, with the object code and listing output written to the default files \$NEWPASS and \$STDLIST, respectively.

```
:COBOL TFILE, ,MFILE,NFILE
```

Prior to merging, the records in both *textfile* and *masterfile* must be arranged in ascending order as dictated by their sequence fields — that is, the value of the sequence field on any record, or it must be blank. (The order of sequencing is based on the ASCII Collating Sequence. There are no restrictions regarding blank sequence fields — the sequence fields of some or all of the records in either *textfile*, *masterfile*, or both files can be blank, and such records can appear anywhere in either file.

The merging operation is also based on ascending order of sequence fields, according to the ASCII Collating Sequence. During merging, the sequence fields of the records in both files are checked for ascending order. If their order is improper, the offending records are skipped during merging and appropriate diagnostic messages are sent to *listfile*. During each comparison step in merging, one record is read from each file and these records are compared, with one of three results:

1. If the values of the sequence fields of the *masterfile* and *textfile* records are equal, then the *textfile* record is compiled and (optionally) passed to *newfile*; the *masterfile* record is ignored; and one more record is read from each file for the next comparison.
2. If the value of the sequence-field of the *masterfile* record is less than that of the *textfile* record, the *masterfile* record is compiled and (optionally) passed to *newfile*; the *textfile* record is retained for comparison with the next *masterfile* record.
3. If the value of the sequence field of the *textfile* record is less than that of the *masterfile* record, the *textfile* record is compiled and (optionally) passed to *newfile*; the *masterfile* record is retained for comparison with the next *textfile* record.

During merging, a record with a blank sequence field is assumed to have the same sequence field as that of the *last* record with a non-blank sequence field read from the same file (or a null sequence field, if no record with a non-blank sequence field has yet been encountered in the file). Thus, a group of one or more records with blank sequence fields residing on *masterfile* are never replaced by records from *textfile*; they can only be deleted through use of the \$EDIT command, as explained later.

Records from *masterfile* that are replaced during merging and thus neither compiled nor sent to *newfile* are not listed during compilation.

When an end-of-file condition is encountered on either *textfile* or *masterfile*, merging terminates (except for the continuing influence of an unterminated VOID parameter in an \$EDIT command, as discussed later). At this point, the subsequent records on the remaining file are checked for proper sequence, compiled, and (optionally) passed to *newfile*. (However, *masterfile* records within the range of a VOID parameter are neither compiled nor sent to *newfile*.)

The sequence field values of records transmitted to *newfile* are not normally changed by the merging operation. However, the user can request the assignment of new sequence characters by using the \$EDIT command.

Checking Sequence Fields

The presence of a *masterfile* during compilation implicitly requests checking of source records for proper sequence. Thus, when the user specifies both *textfile* and *masterfile* as input files for the compiler, or when he specifies *masterfile* alone, sequence-checking is done automatically on both files. But when the user specifies *textfile* as the only input file, sequence checking does not occur. Therefore, when the user wants source input sequence-checked but does not require merging of two input files, he can transmit the input from either the *textfile* or the *masterfile* and equate the unused file to \$NULL.

EXAMPLE

The user could compile a program from the *textfile* SOURCE (with no *masterfile* input) and implicitly request sequence checking with the following command:

```
:COBOL SOURCE, , $NULL
```

Editing

Editing operations during merging consist of omitting sections of the old source program (residing on *masterfile*) and/or re-numbering the sequence fields of the new, merged source program (residing on *newfile*). Both of these operations are requested through the \$EDIT command, written in the following format:

```
[$]EDIT [VOID=sequencevalue]
        [ ,SEQNUM=sequencenumber ]
        [ ,NOSEQ
        [ ,INC=incnumber ]
```


The parameters and options are as follows; the parameters can be specified in any order.

Parameter	Option
VOID=sequencevalue	<p>Requests the compiler to bypass (during merging) all records on <i>masterfile</i> whose sequence fields contain a value less than or equal to <i>sequencevalue</i>, plus any subsequent records with blank sequence fields. This request remains in effect until a <i>masterfile</i> record with a sequence-field value higher than <i>sequencevalue</i> is encountered. The VOID request is initially disabled when the compiler is invoked.</p> <p>The <i>sequencevalue</i> subparameter can be either a legal sequence number or a character string. If <i>sequencevalue</i> is less than six characters, COBOL/3000 left-fills sequence numbers with ASCII zeros and sequence character strings with spaces.</p>
SEQNUM=sequencenumber	<p>Requests re-numbering of the merged source records on <i>newfile</i>, beginning with the value specified by the <i>sequencenumber</i> sub-parameter; this value replaces the sequence number of the next record sent to <i>newfile</i>. The sequence number of each succeeding record is incremented according to the value specified by the INC parameter (or its default), described below. If the <i>SEQNUM=sequencenumber</i> parameter is present but <i>newfile</i> does not exist, the re-numbering request is ignored. If this parameter is present, and <i>newfile</i> exists, the re-numbering request remains in effect until an \$EDIT command with the NOSEQ parameter is encountered. When the merged output is listed, records actually transmitted to <i>newfile</i> appear with the new sequence numbers but records not sent to <i>newfile</i> appear with blank sequence fields. The re-sequencing request is initially disabled when the compiler is invoked.</p> <p>The <i>sequencenumber</i> sub-parameter can be a legal sequence number of one to six digits. If less than six digits, COBOL/3000 left-fills this value with ASCII zeros.</p>
NOSEQ	<p>Suspend re-numbering of merged records on <i>newfile</i>; current sequence numbers are retained. If neither SEQNUM nor NOSEQ are specified, NOSEQ takes effect by default until superseded by SEQNUM.</p>

Parameter	Option
INC=incnumber	Sets increment by which records sent to newfile are re-numbered <i>if</i> SEQNUM is in effect. The increment is specified by <i>incnumber</i> , which can be a value ranging from 1 through 999999. Notice, however, that very large increments are of limited value, since they may cause the six-digit sequence number to overflow. Re-numbering only occurs if SEQNUM is specified (or if the last parameter is not overridden by a NOSEQ parameter) <i>and</i> a <i>newfile</i> exists. If SEQNUM is specified but INC is not, the sequence number is incremented by the default value of 100 for each succeeding record; this default value applies until an INC parameter specifying a new value is encountered.

\$EDIT commands are normally input from *textfile*. (Their input from *masterfile* is allowed, but is not recommended since any \$EDIT command containing a VOID parameter on *masterfile* could void its own continuation records.) \$EDIT commands themselves are never sent to *newfile*; thus, the command form \$\$EDIT , while permitted, is redundant.

While sequence fields are allowed (and usually necessary) on records containing \$EDIT commands, continuation records for such commands should have blank sequence fields.

During merging, a group of one or more *masterfile* records with blank sequence fields are never replaced by lines from *textfile*; they can only be deleted by an \$EDIT command with a *VOID=sequencevalue* parameter at least as great as the last non-blank sequence field preceding the group. In this case, the entire group of *masterfile* records with blank sequence number fields is deleted.

Since voided records are never passed to the *uslfile* or *newfile*, their sequence is never checked, and they never generate an out-of-sequence diagnostic message.

A VOID parameter does not affect records in *textfile*.

Any *masterfile* record replaced by a *textfile* record is treated as if voided, except that following records with blank sequence fields are not also voided. If a replaced record would have been out-of-sequence, the *textfile* record that replaces it produces an out-of-sequence diagnostic message.

In general, whenever a record sent to *newfile* has a non-blank sequence field lower in value than that of the last record with a non-blank sequence field, a diagnostic message is printed.

EXAMPLE

The user wants to merge text input from the standard input device (default value for *textfile*, designated by \$STDIN) with an old program on the file OLDPROG, creating new source input on the file NEWPROG. He wants to re-number the merged source records on NEWPROG beginning with the value 50, incrementing the sequence number of each subsequent record by 10. After logging on, the user enters:

```
:COBOL , , , OLDPROG, NEWPROG
.
.
.
$EDIT SEQNUM=50, INC=10 <<SPECIFIES EDITING PARAMETERS.>>
.
.
.
(New text or corrections to be merged with old program.)
.
.
.
```

APPENDIX C

COBOL Diagnostic Messages



This appendix contains two tables. Table C-1 defines the COBOL compiler diagnostic messages which can occur during compilation of a COBOL source language program. Table C-2 defines the subsystem diagnostic messages which can occur when a COBOL object program is being executed.

COBOL COMPILER DIAGNOSTIC MESSAGES

During compilation of a COBOL source program, the compiler diagnoses questionable or erroneous program elements and then reports them on the program listing. In most cases diagnostic messages are listed immediately following the questionable (or erroneous) source line and a position indicator (either a \wedge or a \uparrow), which is printed below and to the right of the element in question. There will be one position indicator and one diagnostic message for each questionable (or erroneous) element in a source line.

Format of COBOL Compiler Diagnostic Messages

The format of a COBOL compiler diagnostic message is

$$\left. \begin{array}{l} \dots \\ ??? \\ !!! \\ **** \end{array} \right\} \Delta \left\{ \begin{array}{l} \text{WARNING} \\ \text{ERROR} \end{array} \right\} \Delta \text{message-number} \Delta \text{message-text.} \Delta [\text{SEE sequence-number}]$$

where

. . . . equals a warning only code
???? equals a questionable construct code
!!!! equals a serious error code
**** equals a disasterous error code

WARNING is printed following warning or questionable construct codes

ERROR is printed following serious or disasterous error codes

message-number is the number of the message (See Table C-1)

message-text is the actual text of the message (See Table C-1)

SEE sequence-number is a phrase which indicates the line sequence number of the previous diagnostic message if any has been generated.

WARNING ONLY DIAGNOSTIC MESSAGES (. . .). A warning diagnostic message is given for minor violations of COBOL rules which have no effect upon the interpretation or meaning of the program being compiled (for example, missing spaces), or for compiler difficulties when interpreting a subsystem command (see Appendix B). Warning messages can be suppressed by using the **NOWARN** parameter in the **\$CONTROL** subsystem command. The total number of warnings issued (whether or not they have been listed) is given near the end of the program listing.

QUESTIONABLE CONSTRUCT DIAGNOSTIC MESSAGES (????). A questionable construct diagnostic message is given in those cases where the resulting object program will execute, but probably not in the exact way the programmer intended. These diagnostics are counted toward the maximum errors allowed count (as specified by the **ERRORS** parameter of the **\$CONTROL** command, described in Appendix B).

SERIOUS ERROR DIAGNOSTIC MESSAGES (!!!!). A serious error diagnostic message occurs when the grammatic rules of COBOL have been violated, or when the compiler cannot determine the semantics of the program. The program's compilation continues, but to check syntax only; program execution should not be attempted. These diagnostics are counted toward the maximum errors allowed count and the **MAP** and **CODE** parameters of the **\$CONTROL** command are ignored (see Appendix B).

DISASTEROUS ERROR DIAGNOSTIC MESSAGES (**).** When a disasterous error diagnostic message occurs, the compiler has found a condition which prevents continuation of the compilation (for example, a file error or memory space overflow). The compiler terminates the compilation; do not attempt program execution.

EXAMPLE:

The following short COBOL compiler listing extract gives examples of the form and positioning of the different compiler diagnostic messages.

PAGE 0001 HEWLETT-PACKARD 32213A.00.0 COBOL/3000 TUE, AUG 21, 1973, 9:03 AM

```
000100$CONTROL  ERRORS=3
000200 IDENTIFICATION DIVISION.
000300 PROGRAM-ID. DIAGNOSTIC-EXAMPLES.
                                     ^
.... WARNING 29  IMPROPER PROGRAM NAME.
000400 ENVIRONMENT DIVISION.
000500 DATA DIVISION.
000600 WORKING-STORAGE SECTION.
000800 77 B  PICTURE X(3)  VALUE  "ABCDEF".
                                     ^
.... WARNING 40  NON-NUMERIC VALUE EXCEEDS SIZE OF ITEM.  SEE 000300
000900 77 C  PICTURE 9(6).
001000 PROCEDURE DIVISION.
001100 P.  OPEN OUTPUT PRINTER.
                                     ^
!!!! ERROR 150  SYNTAX REQUIRES FILE NAME.  SEE 000800
001200      ADD A,B TO C.
                                     ^
???? ERROR 63  UNDEFINED WORD IGNORED.  SEE 001100
                                     ^
.... WARNING 6  MISSING SPACE.  SEE 001200
                                     ^
???? ERROR 100 ILLEGAL ARITHMETIC OPERAND.  SEE 001200
                                     ^
**** ERROR 1   TOO MANY ERRORS.  SEE 001200

CHECKED SYNTAX ONLY.
CPU TIME = 0:00:01.  WALL TIME = 0:00:07.
END COBOL/3000 COMPILATION. 004 ERRORS. 003 WARNINGS.  SEE 001200
```

OBJECT PROGRAM DIAGNOSTIC MESSAGES

During execution of a COBOL object program, the subsystem can produce diagnostic messages. These messages have the form

file-name

*** ERROR nnn message-text

where

file-name is the name of the file (appears only for file-related errors)

nnn is the object program diagnostic message number (see Table C-2)

message-text is the actual diagnostic message

Following each file error which occurs during object program execution (message numbers 520 through 631), MPE/3000 outputs a File Information Display. The File Information Display can be used to find the exact cause of the file error. Consult *MPE Commands Reference Manual* for a definition of the File Information Display.

Table C-1. COBOL Compiler Diagnostic Messages

Message Number	Type	Message Text and Explanation	Programmer Response
001	*	<p>TOO MANY ERRORS.</p> <p>In order to save computer time, the user may specify that the compilation is to terminate after a number of serious and questionable errors are found. This number is set using a compiler subsystem command. The compilation terminates.</p>	<p>Correct the errors and/or change the maximum error count with a subsystem command. (Recompiling without correcting known errors wastes computer time.)</p>
002	?	<p>ILLEGAL CHARACTER IGNORED.</p> <p>The indicated character is either an illegal ASCII character or is not a valid COBOL character in this context. The character is ignored.</p>	<p>Check the punchcard code.</p>
003	.	<p>FIELD A MUST BE BLANK.</p> <p>Field A (columns 8 - 11) must be blank in a continuation record. However, the Field is included in the record.</p>	<p>Observe proper source record format.</p>
004	.	<p>TOO MANY CHARACTERS.</p> <p>The indicated word, number, string, or non-numeric literal contains or specifies too many characters in the given context. In a COBOL source record, the item is truncated. In a compiler subsystem command, the parameter is ignored.</p>	<p>Correct and/or shorten the indicated item.</p>
005	.	<p>SHOULD BE HYPHEN OR ASTERISK.</p> <p>Column 7 contains a character other than a blank or one of the following:</p> <ul style="list-style-type: none"> - indicates a continuation line. * indicates a comment line. \$ indicates a compiler subsystem command. <p>The line is treated as a comment.</p>	<p>Check the source record for proper format.</p>
006	.	<p>MISSING SPACE.</p> <p>COBOL standards require a space in certain contexts, but the compiler is able to interpret the meaning correctly. No action is taken.</p>	<p>Insert a space.</p>

Table C-1. COBOL Compiler Diagnostic Messages (cont.)

Message Number	Type	Message Text and Explanation	Programmer Response
007	.	<p>ILLEGAL COMMAND.</p> <p>The subsystem command name was not recognized; the entire command is ignored.</p>	Review the subsystem commands available in COBOL.
008	?	<p>IMPROPER NUMERIC LITERAL.</p> <p>The indicated numeric literal was not formed properly; for example, it contains two decimal points or an alphabetic character.</p>	Review the rules for forming numeric literals; correct the literal.
009	.	<p>MISSING QUOTATION MARK.</p> <p>Terminal quotation mark is missing from a non-numeric literal or string constant in a subsystem command. May be caused by missing continuation record. Compiler assumes a quote.</p>	Supply quotation mark or continuation record.
010	*	<p>STATE-STACK OVERFLOW.</p> <p>The complexity of the indicated statement caused an internal compiler table to overflow, even though the statement may not contain any other specific error. Compilation terminates.</p>	Break the statement into a series of shorter, simpler COBOL statements.
011	!	<p>GRAMMATICAL ERROR ! EXPECTING ONE OF THE SYMBOLS BELOW.</p> <p>The indicated item appears in some context that the compiler cannot recognize. On the line following this message, the compiler lists the items expected at this point in the program. The compiler attempts to recover from this error (see message 67).</p>	Review statement format and correct. Also, notice that this statement may be correct, but cannot logically follow the preceding statement.
012	*	<p>UNEXPECTED DIVISION END.</p> <p>Division or program ended unexpectedly. This is frequently the result of a previous grammatical error. The compilation terminates.</p>	Correct grammatical errors.
013	*	<p>SPACE OVERFLOW.</p> <p>The compiler has used all the space available to it under MPE. The compilation terminates.</p>	Either subdivide the program using the Interprogram Communication feature, or change the maximum allowable data segment size in MPE.

Table C-1. COBOL Compiler Diagnostic Messages (cont.)

Message Number	Type	Message Text and Explanation	Programmer Response
014	?	<p>IMPROPER USE OF RESERVED SYMBOL.</p> <p>A reserved word or character has been used in the wrong context. (For example, a reserved word has been used as a data-name or a paragraph-name.) The compiler ignores the symbol.</p>	<p>Check the list of reserved words and then use a word or symbol that is not reserved.</p>
015	?	<p>IMPROPER LEVEL NUMBER.</p> <p>Level number is not in the range 01 to 49, 66, 77, or 88. Compiler assumes a likely number.</p>	<p>Use a proper level number.</p>
016	.	<p>NON-STANDARD LEVEL NUMBER.</p> <p>Assume that a data heirarchy has been established using level numbers 01, 02, and 04 when the compiler finds an 03 level number. The 03 level item represents a non-standard level number. The compiler treats the entry as though it were coded as an 02 level entry.</p>	<p>Change the level number to conform to the rules for level numbers. Also, check to be certain that subsequent data entries still reflect the desired data structure after the non-standard level number has been changed.</p>
017	.	<p>CLAUSE DUPLICATION.</p> <p>The same clause appears two or more times in the same entry. For example, a data description contains a USAGE COMP and a USAGE DISPLAY clause. The compiler uses the last-entered clause.</p>	<p>Eliminate redundant clauses. (This warning may be ignored if the program runs properly.)</p>
018	?	<p>IMPROPER INTEGER.</p> <p>The compiler has encountered an integer outside the range 0 to 32,767. The compiler assumes a number within the legal range.</p>	<p>Assign an integer within the legal range.</p>
019	?	<p>REDEFINED DATA-NAME NOT DEFINED.</p> <p>The item being redefined is either missing or was dropped from the program because of some other error. The compiler ignores the REDEFINES clause.</p>	<p>Be certain that the redefined item is correctly defined and then recompile.</p>

Table C-1. COBOL Compiler Diagnostic Messages (cont.)

Message Number	Type	Message Text and Explanation	Programmer Response
020	?	<p>ILLEGAL PICTURE.</p> <p>Typical causes of this message are:</p> <ul style="list-style-type: none"> ● illegal character picture string. ● illegal combination of picture string characters. ● more than 18 digits specified for numeric data. <p>The compiler replaces illegal picture strings as though they were coded PIC 9(1).</p>	<p>If the cause of the error is not obvious, read the section of the COBOL manual which deals with the PICTURE clause character-string. Correct the error and recompile.</p>
021	?	<p>REFERENCE NOT UNIQUE.</p> <p>The indicated reference is not uniquely qualified. The compiler uses one of available possibilities.</p>	<p>Make the reference unique either by adding sufficient qualification or by spelling it differently. If the reference is a data-name manipulated by the CORRESPONDING option, it must be qualified.</p>
022	!	<p>SECTION MISPLACED.</p> <p>FILE SECTION, WORKING-STORAGE SECTION, and LINKAGE SECTION must appear in that order when used.</p>	<p>Correct the sequence of the sections.</p>
023	?	<p>LEVEL NUMBER IN REDEFINE CLAUSE IS 66 OR 88.</p> <p>The compiler ignores this entry.</p>	<p>Correct the error and recompile.</p>
024	?	<p>REDEFINING A DIFFERENT LEVEL.</p> <p>The compiler ignores the REDEFINES entry.</p>	<p>Correct the level number of the REDEFINES entry and recompile.</p>
025	.	<p>REDEFINING A TABLE.</p> <p>The REDEFINES clause may not refer to an item defined with the OCCURS clause; however, this compiler allows redefinition of a table.</p>	<p>Ignore the warning diagnostic or correct the error by defining the item first and then redefining it with the table description.</p>

Table C-1. COBOL Compiler Diagnostic Messages (cont.)

Message Number	Type	Message Text and Explanation	Programmer Response
026	.	<p>REDEFINING AN ITEM SUBORDINATED TO A TABLE.</p> <p>The COBOL language specifies that the REDEFINES clause may not refer to an item subordinated to a table; however, this compiler allows such a redefinition.</p>	<p>Ignore the warning diagnostic or correct the error by defining the item first and then redefining it with the table description.</p>
027	?	<p>REDEFINES CLAUSE CONTAINS A VALUE CLAUSE.</p> <p>The values specified for the REDEFINES clause are compiled into the program. If the item being redefined also specified values, the original values are replaced.</p>	<p>Values should be associated with the original item rather than the REDEFINES clause.</p>
028	!	<p>REPORT WRITER MODULE NOT IMPLEMENTED.</p> <p>Although the REPORT WRITER module is not implemented, this compiler checks the syntax of REPORT WRITER statements so that the compiler can be more useful for educational purposes.</p>	<p>No object code is generated for REPORT WRITER statements; rewrite this portion of the program.</p>
029	.	<p>IMPROPER PROGRAM NAME.</p> <p>If a program-name is greater than 15 characters, it is truncated and this warning is generated. The truncated name is assigned to the program. If the program name does not begin with an alphabetic character, the default name PROGRAM' ID is assigned, and this warning is generated.</p>	<p>Correct the program-name.</p>
030	?	<p>MNEMONIC NAME MUST BE UNIQUE.</p> <p>A mnemonic assigned in the SPECIAL-NAME paragraph is defined elsewhere in the program. The compiler ignores this clause.</p>	<p>Assign a different, unique mnemonic name.</p>
031	?	<p>OCCURS CLAUSE CANNOT BE SPECIFIED ON AN 01 OR 77 LEVEL.</p> <p>The compiler accepts the level number as coded, but ignores the OCCURS clause.</p>	<p>For an 01 level entry: Add a dummy 01 level entry and change the level number of the OCCURS item to 02. For a 77 level entry: The OCCURS clause cannot be used at this level; recode the entry.</p>

Table C-1. COBOL Compiler Diagnostic Messages (cont.)

Message Number	Type	Message Text and Explanation	Programmer Response
032	?	<p>MULTIPLE REEL/UNIT NOT IMPLEMENTED.</p> <p>The compiler treats this clause as a comment.</p>	<p>Because MPE does not currently support multiple reel/unit files, the program must be rewritten.</p>
033	?	<p>RERUN OPTION NOT IMPLEMENTED.</p> <p>The compiler treats this option as a comment.</p>	<p>Because MPE does not currently support RERUN, the program must be rewritten.</p>
034	.	<p>FILE-LIMIT(S) CLAUSE NOT IMPLEMENTED.</p> <p>The compiler treats this clause as documentation.</p>	<p>Eliminate the FILE-LIMITS clause to suppress this warning message. (The program will operate in any case.)</p>
035	?	<p>SAME RECORD/SORT AREA FILE-NAME UNDEFINED.</p> <p>The compiler ignores the SAME clause.</p>	<p>Correct the error and recompile.</p>
036	.	<p>FILE-NAME APPEARS IN MORE THAN 1 SAME/RECORD/SORT AREA.</p>	<p>Correct the error and recompile.</p>
037	.	<p>FILE-NAME NOT IN SAME RECORD/SORT AREA CLAUSE.</p> <p>This message is issued under the following conditions:</p> <ul style="list-style-type: none"> ● A sort-file cannot be named in the SAME AREA clause unless the RECORD option is used. ● All files named in the SAME AREA clause must also appear in the SAME RECORD AREA clause. ● All the files named in the SAME AREA clause must also appear in the SAME SORT AREA clause. 	<p>Correct the error and recompile.</p>

Table C-1. COBOL Compiler Diagnostic Messages (cont.)


Message Number	Type	Message Text and Explanation	Programmer Response
038	?	<p>FORMAL FILE DESIGNATOR MORE THAN 8 CHARACTERS.</p> <p>The formal file designator – the name by which the file is known to MPE – can be no more than 8 characters in length. The compiler truncates longer names to 8 characters. Notice that the formal file designator refers only to the name assigned as part of the system-file-name in the ASSIGN clause; the system-file-name itself may be longer than 8 characters.</p>	<p>Change the name to be 8 characters or less in length. (Or use the truncated name assigned by the compiler.)</p> 
039	?	<p>FILE SIZE MORE THAN 9 DIGITS.</p> <p>The file size entry in the system-file-name may not exceed 9 digits. The compiler assigns a default value of 10,000.</p>	<p>Correct this problem in either of two ways: Either correct the source program and recompile, or use an MPE :FILE command to specify the desired file size when the program is run.</p>
040	.	<p>NON-NUMERIC VALUE EXCEEDS SIZE OF ITEM.</p> <p>The non-numeric string is truncated from left or right, depending on whether or not the JUSTIFIED clause is specified.</p>	<p>Correct the error and recompile the program.</p>
041	?	<p>CLASS NOT DA, UT, OR UR.</p> <p>The class field in the system-file-name of the ASSIGN clause is incorrect. The compiler ignores the rest of the ASSIGN clause.</p>	<p>Correct the system-file-name and recompile.</p>
042	?	<p>RECORDING MODE NOT A OR B.</p> <p>The recording mode field in the system-file-name of the ASSIGN clause is neither A nor B. The compiler assumes B (binary).</p>	<p>If recording mode B is acceptable, no action need be taken. If B is not acceptable, correct the system-file-name and recompile the program.</p>
043	?	<p>OCCURS SUBJECT MUST BE THE ONLY KEY.</p> <p>The compiler assumes the object of the OCCURS clause to be the only key and ignores all other items named in the KEY clause.</p>	<p>Correct the error and recompile the program.</p>

Table C-1. COBOL Compiler Diagnostic Messages (cont.)

Message Number	Type	Message Text and Explanation	Programmer Response
044	?	<p>FILE-NAME APPEARED IN MORE THAN 1 SELECT CLAUSE.</p> <p>The compiler assumes that the last SELECT statement is correct and compiles it into the program.</p>	<p>Either correct the error and recompile or use an MPE :FILE command when the program is run.</p>
045	.	<p>EMPTY PICTURE STRING.</p> <p>Either no PICTURE clause is given, or the PICTURE clause character-string is missing (possibly because of a grammatical error). The compiler assumes that this item is a group item. Notice that this assumption may cause a level number error on subsequent entries.</p>	<p>Correct the error and recompile the program.</p>
046	?	<p>A SAME-AREA FILE NOT APPEARED IN SAME-SORT AREA CLAUSE.</p>	<p>Correct the error and recompile the program.</p>
047	?	<p>DATA RECORD NOT 01-LEVEL.</p> <p>Data record definitions for a file must begin at the 01 level. The compiler changes the level number of this entry to 01 and attempts to continue the compilation. Notice that this assumption may cause a level number error on subsequent entries.</p>	<p>Correct the error by supplying the missing 01 level entry and recompiling the program.</p>
048	?	<p>88-LEVEL WITHOUT VALUE CLAUSE.</p> <p>The item is discarded.</p>	<p>Correct the error and recompile the program.</p>
049	?	<p>INTEGER-1 MUST BE < INTEGER-2 IN OCCURS CLAUSE.</p> <p>The compiler sets integer-2 to the value of integer-1.</p>	<p>Correct the error and recompile the program.</p>
050	?	<p>INTEGER-2 IN OCCURS CLAUSE MUST BE > 0.</p> <p>A negative literal is given for integer-2. The compiler assumes that a value of 0 is given. Notice that this assumption may cause message 049.</p>	<p>Correct the error and recompile the program.</p>

Table C-1. COBOL Compiler Diagnostic Messages (cont.)

Message Number	Type	Message Text and Explanation	Programmer Response
051	!	<p>TABLE EXCEEDS THREE DIMENSIONS.</p> <p>The COBOL language supports tables containing no more than three levels of OCCURS clauses. This compiler uses only the first three levels defined and ignores the remainder.</p>	<p>Rewrite the program using no more than three levels of OCCURS clauses.</p>
052	.	<p>EMPTY GROUP.</p> <p>An empty item is a group item which contains no elementary item definitions. For example, if two consecutive statements are both 01 level items without a PICTURE clause, the first 01 is an empty group. Such items cannot be manipulated within the program since the compiler has no way to calculate a length for the item.</p>	<p>Correct the error and re-compile the program.</p>
053	!	<p>SIZE OF DATA SEGMENT GREATER THAN 65K BYTES.</p> <p>The HP 3000 cannot handle data segments longer than 65K bytes. When this occurs, the compiler continues to check the source program syntax, but generates no object code.</p>	<p>The program must be re-written. Handling extremely large records can often be simplified using the Interprogram Communication feature with a \$CONTROL DYNAMIC parameter.</p>
054	?	<p>OCCURS. .DEPENDING ON WITHIN RANGE OF ANOTHER OCCURS.</p>	<p>Correct the error and re-compile the program.</p>
055	!	<p>FILE-NAME NOT APPEARED IN A SELECT CLAUSE.</p> <p>The compiler attempts to use the file-name as a formal file designator. However, this does not provide sufficient information for a successful compilation.</p>	<p>Correct the error and re-compile the program.</p>

Table C-1. COBOL Compiler Diagnostic Messages (cont.)

Message Number	Type	Message Text and Explanation	Programmer Response
057	?	<p>LABEL RECORD NON 01-LEVEL.</p> <p>The data-name in the LABEL RECORD IS clause has a level number other than 01. Although the compiler attempts to use the data-name as coded, this may cause other errors.</p>	<p>Correct the error and re-compile the program.</p>
058	?	<p>LABEL RECORD SIZE NOT 80 CHARACTERS.</p> <p>The compiler assumes a record size of 80 characters for the label.</p>	<p>Change the label record size to 80 characters and re-compile the program.</p>
059	.	<p>SYSTEM-FILE-NAME BEYOND THE 1ST TREATED AS COMMENT.</p> <p>Only the first system-file-name in a SELECT statement has meaning to the compiler.</p>	<p>To suppress the warning, correct the error and re-compile the program. To alter file characteristics at object time, use MPE :FILE commands.</p>
060	?	<p>USAGE CONFLICT BETWEEN GROUP AND ITS ELEMENTARY ITEMS.</p> <p>The compiler assigns the USAGE of the group to each of its elementary items.</p>	<p>Correct the error(s) and re-compile the program.</p>
061	?	<p>IMPROPER USE OF SYNC/JUST/PIC/VALUE/BLANK WHEN 0 CLAUSES.</p> <p>Common causes for this message are:</p> <ul style="list-style-type: none"> ● BLANK WHEN ZERO cannot appear with 88 level items. ● None of these clauses may appear with an index-data-item. ● VALUE, JUST, SYNC, or USAGE may not be subordinate to a group item assigned a value. 	<p>Correct the error and re-compile the program.</p>
062	?	<p>ILLEGAL CURRENCY SIGN.</p> <p>The selected symbol cannot be used as a currency sign. The compiler assumes \$.</p>	<p>Select another character and recompile the program.</p>

Table C-1. COBOL Compiler Diagnostic Messages (cont.)

Message Number	Type	Message Text and Explanation	Programmer Response
063	?	<p>UNDEFINED WORD IGNORED.</p> <p>The indicated data-name was not defined in the Data Division. The word may be undefined, dropped because of a previous error, or simply misspelled. The compiler ignores the undefined word.</p>	<p>Correct the error and re-compile the program.</p>
064	?	<p>LEVEL-77 MUST BE ELEMENTARY ITEM.</p> <p>The compiler assumes that the indicated item is a group with no elementary items.</p>	<p>Correct the entry by giving it a legitimate PICTURE character-string and re-compile the program.</p>
065	?	<p>MISSING PROGRAM-ID PARAGRAPH.</p> <p>COBOL standards require a PROGRAM-ID paragraph. However, this compiler will assume a program name PROGRAM' ID if the paragraph is missing. (See message 029.)</p>	<p>Supply a valid PROGRAM-ID paragraph and recompile the program.</p>
066	!	<p>FILE-NAME MUST BE UNIQUE.</p> <p>The compiler cannot generate a useful object program if this message appears. However, the compiler will accept the duplicate file-name and continue to check the source program syntax.</p>	<p>Supply a valid file-name and recompile the program.</p>
067	.	<p>ATTEMPTING SYNTAX RECOVERY FROM PRECEDING SYMBOL.</p> <p>This message is normally paired with message 011. Notice the positions of the arrows associated with this message and message 011. Because of grammatical errors, the compiler ignores all source text between the two arrows.</p>	<p>Correct the previous error to eliminate both this message and message 011.</p>
068	.	<p>NESTED COPY STATEMENT IGNORED.</p> <p>A COPY statement was encountered in a source record from the copyfile. The compiler ignores the COPY statement.</p>	<p>Remove the COPY statement from the copyfile. Any uncopied statements must be added to the program.</p>
069	?	<p>REFERENCE INCOMPLETE.</p>	<p>Correct the error and re-compile the program.</p>

Table C-1. COBOL Compiler Diagnostic Messages (cont.)

Message Number	Type	Message Text and Explanation	Programmer Response
070	?	<p>NAME MUST NOT QUALIFY ITSELF.</p> <p>A word has been used twice in the qualification of a data-name. The compiler will attempt to find the correct reference.</p>	<p>Correct the qualification of the data-name and recompile the program.</p>
071	!	<p>REDEFINING ITEM SIZE MUST BE <= REDEFINED ITEM SIZE.</p> <p>An item whose description includes a REDEFINES clause has a total length which is greater than the item being redefined. This makes an accurate location count impossible.</p>	<p>Correct the error by defining the longer item first and recompile the program.</p>
072	.	<p>SEQUENCE ERROR ON TEXTFILE.</p> <p>The sequence numbers (cols 1 - 6) in the textfile are not in ascending order.</p>	<p>Correct the sequence numbers in the textfile.</p>
073	.	<p>SEQUENCE ERROR ON MASTFILE.</p> <p>The sequence numbers (cols 1 - 6) in the master file are not in sequence.</p>	<p>Correct the sequence numbers in the master file.</p>
074	.	<p>SEQUENCE ERROR ON NEWFILE.</p> <p>The sequence number of the next record to be written to the NEWFILE is not greater than the previous non-blank sequence number for NEWFILE.</p>	<p>Correct the record sequence.</p>
075	.	<p>COMMAND CONTINUATION NOT FOUND.</p> <p>The record following a continued subsystem command did not contain a \$ in column 7. The subsystem command is not continued.</p>	<p>Supply the missing command continuation.</p>
076	.	<p>IMPROPER COMMAND PARAMETER.</p> <p>An error was found in a subsystem command parameter. The faulty parameter is ignored.</p>	<p>Correct the subsystem command.</p>
077	?	<p>INDEX NAME MUST BE UNIQUE.</p> <p>The compiler has found a duplicate index-name. The compiler ignores <i>all</i> the index-names associated with this OCCURS clause.</p>	<p>Change the duplicate index-name and recompile.</p>

Table C-1. COBOL Compiler Diagnostic Messages (cont.)

Message Number	Type	Message Text and Explanation	Programmer Response
078	?	<p>TABLE HAS MORE THAN 12 INDEX NAMES.</p> <p>A table is limited to no more than 12 index-names. The compiler ignores all but the first 12 indexes.</p>	<p>If the current version of the program does not use the excess indexes, the program should execute. Otherwise, delete the extra indexes and recompile.</p>
079	!	<p>THE FOLLOWING SORT/SEARCH KEY IS UNDEFINED.</p> <p>Following this message, the compiler will list the missing key. The SEARCH or SORT associated with the key cannot be generated.</p>	<p>Define the missing key(s) and recompile the program.</p>
080	!	<p>THE FOLLOWING ACTUAL KEY IS UNDEFINED.</p> <p>Following this message, the compiler will list the missing key. File access statements associated with the key cannot be generated.</p>	<p>Define the missing key(s) and recompile the program.</p>
081	!	<p>THE FOLLOWING DEPENDING-ON DATA-NAME IS UNDEFINED.</p> <p>Following this message, the compiler will list the missing data-name. Table handling statements associated with the data-name cannot be generated.</p>	<p>Define the data-name and recompile the program. Notice that this error may be caused by misspelling the name of a correctly defined item.</p>
082	?	<p>RENAMING AN ITEM NOT YET DEFINED.</p> <p>The compiler ignores this statement.</p>	<p>Correct the error and recompile the program.</p>
083	?	<p>RENAMING ITEM WITH LEVEL 01, 66, 77, OR 88.</p> <p>COBOL syntax rules do not permit renaming at these levels. The compiler ignores this statement.</p>	<p>Either RENAME at a legal level or delete the RENAMES entry, and then recompile the program.</p>

Table C-1. COBOL Compiler Diagnostic Messages (cont.)

Message Number	Type	Message Text and Explanation	Programmer Response
084	?	<p>RENAMING AN ITEM HAVING A RELATIVE WITH OCCURS CLAUSE.</p> <p>COBOL syntax rules do not permit renaming of items related to an OCCURS clause. (This would cause confusion as to whether the entire table or an element of the table is the renamed item.) The compiler ignores the RENAMES entry.</p>	Delete the RENAMES entry or redefine the table so that the desired data can be accessed with a data-name.
085	.	<p>RENAME CLAUSE NON-66 LEVEL.</p> <p>RENAMES can be specified only at the 66 level. The compiler ignores a RENAMES at any other level.</p>	Change the level number of the RENAMES entry to 66 and recompile the program.
086	?	<p>RENAMED START-POINT >= END-POINT.</p> <p>In a RENAMES . . . THRU clause, either data-name-2 logically follows data-name-3 in the record structure (and should not), or they name the same item. The compiler attempts to assign a reasonable length to the renamed entry, but this is not always possible.</p>	Program execution using the renamed item may not produce the desired result. To ensure the correct result, correct the error and recompile the program.
087	?	<p>CONDITION-NAME WITHOUT VARIABLE.</p> <p>The compiler ignores the 88 level entry.</p>	Give the condition-name a variable.
088	?	<p>EXPECTING NUMERIC LITERAL OR FIGURATIVE ZERO.</p> <p>A conditional variable is defined as a numeric item, but the value assigned to the item at the 88-level is either a non-numeric constant or a figurative constant other than ZERO. For example:</p> <p>02 TRANCODE PIC 9. 88 NEW-ACCOUNT VALUE "N".</p>	Correct the error and recompile the program.
089	?	<p>EXPECTING NON-NUMERIC LITERAL OR FIGURATIVE CONSTANT.</p> <p>This condition is the opposite of message 088: A conditional variable is defined as an alphabetic item, but the value assigned to the item at the 88-level is numeric.</p>	Correct the error and recompile the program.

Table C-1. COBOL Compiler Diagnostic Messages (cont.)

Message Number	Type	Message Text and Explanation	Programmer Response
090	.	RANGE OF VALUES NOT ALLOWED. The compiler accepts only the first literal.	Correct the error and re-compile the program.
091	?	OPTIONAL FILE MUST BE SEQUENTIAL. The OPTIONAL clause may not be used with random access files.	If a random access file is intended, remove the OPTIONAL clause from its description. If an optional sequential file is intended, its description must include the ACCESS IS SEQUENTIAL clause.
092	?	UNABLE TO LOCATE LIBRARY NAME ON COPYLIB. The COPY statement is ignored.	Add the appropriate source statements to the copy file and recompile, or add the statements to the source program and recompile after removing the COPY statement.
093	!	ACTUAL KEY REQUIRED IN RANDOM ACCESS. Random files cannot be accessed without an ACTUAL KEY.	Include an ACTUAL KEY IS data-name clause in the file's description and recompile the program.
094	.	ACTUAL KEY NOT ALLOWED IN SEQUENTIAL ACCESS. An ACTUAL KEY is redundant and is not used for a sequential file.	Remove the ACTUAL KEY clause from the file's description and recompile the program.

Table C-1. COBOL Compiler Diagnostic Messages (cont.)

Message Number	Type	Message Text and Explanation	Programmer Response
095	?	<p>NUMERIC LITERAL AND ITS PICTURE DISAGREE.</p> <p>When a numeric literal disagrees with its picture, the compiler aligns the literal according to the rules for Standard Data Positioning and supplies truncation or zero-fill on both the left and right, as needed. For example, PIC 99V99 VALUE 123.456 is compiled as 23.45.</p>	Correct the error and re-compile the program.
096	?	<p>ILLEGAL CHAR. IN NUMERIC LITERAL.</p> <p>When a numeric literal contains an illegal character, the compiler accepts the literal up to but not including, the illegal character. Thus, the literal 123G43 is compiled as 123.</p>	Correct the error and re-compile the program.
097	.	<p>COMMENT MUST BE CONTAINED ON ONE LINE OF THE COMMAND.</p> <p>Subsystem command is discarded.</p>	Correct the error and re-compile the program.
098	?	<p>JUSTIFIED CLAUSE IN NUMERIC OR NUMERIC-EDITED ITEM.</p> <p>Numeric and edited-numeric items are always aligned by decimal point. Thus the JUSTIFIED clause cannot be used with these items. The compiler ignores the clause.</p>	Correct the error and re-compile the program.
099	?	<p>ILLEGAL FORMAL FILE DESIGNATOR.</p> <p>The first character of a formal file designator must be a \$ or an alphabetic character. Subsequent characters must be a slash (/), period (.), an alphabetic character, or a digit.</p>	Correct the error and re-compile the program.
100	?	<p>ILLEGAL ARITHMETIC OPERAND.</p> <p>An operand in an arithmetic statement cannot be used (for example, it might be an alphabetic item). The compiler replaces the illegal operand with the TALLY register except when the statement includes the CORRESPONDING option. In this case, the compiler ignores the arithmetic statement.</p>	Correct the illegal operand and recompile the program.

Table C-1. COBOL Compiler Diagnostic Messages (cont.)

Message Number	Type	Message Text and Explanation	Programmer Response
101	?	<p>ILLEGAL ARITHMETIC REPLACEMENT IDENTIFIER.</p> <p>The identifier designated to store the results of an arithmetic operation is illegal. The compiler replaces the identifier with the TALLY register.</p>	<p>Change the identifier and recompile the program.</p>
102	?	<p>COMPOSITE OF OPERANDS > 18 DIGITS.</p> <p>The composite of operands must not contain more than 18 digits. This composite is the data item resulting from superimposition of all operands (except the identifier following the word GIVING in the ADD or SUBTRACT statement), aligned on their decimal points.</p>	<p>Correct the error and recompile the program.</p>
103	?	<p>ILLEGAL PROCEDURE-NAME.</p> <p>Common causes of illegal procedure-names are:</p> <ul style="list-style-type: none"> ● Name is also a data-name. ● Name is incorrectly qualified as when a paragraph-name instead of a section-name is used. ● A numeric name is not an integer. <p>When the illegal name is coded as a paragraph-name, the compiler ignores it; when the illegal name appears as a reference, the compiler assigns the dummy name ILL' BRANCH, which will cause the program to abort if executed.</p>	<p>Correct the name and recompile the program.</p>
104	?	<p>DEFINITION INCONSISTENT WITH REFERENCE.</p> <p>The indicated name was defined as a section-name, but was referenced as a paragraph-name. The compiler ignores the name.</p>	<p>Correct the error and recompile the program.</p>
105	?	<p>DUPLICATE PROCEDURE NAMES.</p> <p>The indicated procedure-name is multiply defined and is not adequately qualified. The compiler ignores the procedure-name.</p>	<p>Correct the error and recompile the program. Be certain that no paragraph-name is the same as a section-name.</p>

Table C-1. COBOL Compiler Diagnostic Messages (cont.)

Message Number	Type	Message Text and Explanation	Programmer Response
106	?	<p>SECTION ILLEGAL IN PROGRAM.</p> <p>The compiler has encountered a section-name within the Procedure Division of a program when the Procedure Division did not begin with a section-name. The section is treated as a paragraph.</p>	<p>If the Procedure Division is to contain a paragraph, then the entire division must be in sections. Correct the error by changing the erroneous section-name to a paragraph-name or by adding a section-name at the beginning of the Procedure Division.</p>
107	?	<p>ILLEGAL PRIORITY NUMBER.</p> <p>Either the priority number appears in the Declaratives Section, or it is greater than 99. The compiler sets the priority number to 0.</p>	<p>Change or remove the illegal priority number.</p>
108	?	<p>ILLEGAL PROGRAM CONSTRUCT.</p> <p>This message indicates an illegal section/paragraph construction or an illegal Declaratives/paragraph construction. Either the previous section was empty or contained no paragraphs. Or, the paragraph either begins or ends the Declaratives Section in such a way that its relationship to a USE statement is not clear. The compiler includes these statements in the program, but this may cause errors if the program is executed.</p>	<p>Correct the illegal construct and recompile the program.</p>
109	?	<p>ILLEGAL STATEMENT FORM.</p> <p>A GO TO statement without an operand appeared as other than the first statement in a paragraph, thus making the GO TO unavailable for use by an ALTER statement. The compiler substitutes the statement GO TO ILL' BRANCH, which will cause the program to abort if executed.</p>	<p>Make the GO TO statement the first in the paragraph, or remove it from the program. (Notice that removing the statement will probably require the removal of any ALTER statements that refer to it.)</p>
110	?	<p>SECTION IS ILLEGAL REFERENCE.</p> <p>An ALTER statement refers to a section-name rather than a paragraph-name. The compiler ignores the ALTER statement.</p>	<p>Correct the ALTER statement so that it refers to a GO TO statement which immediately follows a paragraph-name. Recompile the program.</p>

Table C-1. COBOL Compiler Diagnostic Messages (cont.)

Message Number	Type	Message Text and Explanation	Programmer Response
111	?	TABLE-REFERENCE MUST BE INDEXED OR SUBSCRIPTED.	Correct the error and re-compile the program.
112	!	NUMBER OF PARAMETERS >60. The number of identifiers in an ENTRY or CALL statement or in the USING option of the Procedure Division heading is greater than the maximum of 60. The compiler ignores identifiers after the 60th.	Decrease the number of identifiers and recompile the program.
113	!	FORMAL PARAMETER NOT IN LINKAGE SECTION. An identifier named in an ENTRY statement or in the USING option of the Procedure Division heading does not appear in the Linkage Section. The compiler ignores the identifier.	Add the identifier to the Linkage Section and re-compile the program.
114	!	ILLEGAL PARAMETER. An identifier named in an ENTRY statement or in the USING option of the Procedure Division heading is illegal. Such identifiers must have level number 01 or 77, and may not be subscripted or refer to special registers such as TALLY.	Correct the error and re-compile the program.
115		(this message number not used)	
116	?	ENTRY NOT ALLOWED IN DECLARATIVES PORTION. An ENTRY statement has been found in the Declaratives Section of the program. The compiler ignores the ENTRY statement.	Remove the ENTRY statement and recompile the program.
117	?	ALTER DOES NOT REFERENCE ALTER-ABLE GO TO. The first statement of a paragraph named in an ALTER statement is not a GO TO. The compiler ignores the ALTER statement.	Either make the GO TO statement the first in the paragraph, or remove the ALTER statement. Recompile the program.

Table C-1. COBOL Compiler Diagnostic Messages (cont.)

Message Number	Type	Message Text and Explanation	Programmer Response
118	?	<p>PARAMETER MUST START AT WORD BOUNDARY.</p> <p>All parameters named in a subprogram CALL must begin on a word boundary. (A word is two bytes – 16 bits – in length.) The compiler changes this reference to point to the word containing the desired item. This has the effect of adding one byte at the left of the desired item.</p>	<p>Either reposition the desired item so that it falls on a word boundary (remember to count slack bytes), or set up an 01 or 77 level field for the item in Working-Storage. (01 and 77 level items are always allocated starting on a word boundary.)</p>
119	!	<p>UNDEFINED LOCAL PARAGRAPHS.</p> <p>Following this message, the compiler lists any undefined local paragraphs in the previous section.</p>	<p>Define the listed paragraphs and recompile the program.</p>
120	!	<p>DEPENDING-ON IDENTIFIER IS NOT INTEGER.</p> <p>The compiler generates no code for the GO TO DEPENDING ON statement.</p>	<p>Change the identifier and recompile the program.</p>
121	?	<p>EXIT STATEMENT MUST BE ONLY ONE IN PARAGRAPH.</p> <p>COBOL syntax requires that the EXIT statement be the only statement in a paragraph. However, this compiler processes the statement normally.</p>	<p>Correct the error and recompile the program. (Notice that if this error is left in the program any code between the EXIT and the next paragraph-name is inaccessible when the preceding paragraph is performed.)</p>
122	?	<p>ILLEGAL IDENTIFIER IN CORRESPONDING.</p> <p>The compiler ignores the statement.</p>	<p>Check the rules for corresponding data items, then correct the error and recompile the program.</p>

Table C-1. COBOL Compiler Diagnostic Messages (cont.)

Message Number	Type	Message Text and Explanation	Programmer Response
123	?	<p>NO PAIRS FOUND IN CORRESPONDING.</p> <p>Since no corresponding items were found, no code is generated for the statement.</p>	<p>Check the rules for corresponding data-items, then correct the error and re-compile the program.</p>
124	?	<p>ILLEGAL CLASS-CONDITION IDENTIFIER.</p> <p>Identifiers named in a class-condition must be DISPLAY items. If the identifier has any other usage, the compiler ignores the statement. When the IF NUMERIC statement is applied to an alphabetic field or the IF ALPHABETIC statement is applied to a numeric field, the compiler generates the test indicated in the IF statement. Under certain conditions this may cause false results since certain packed or binary numbers may have bit configurations that happen to be identical to alphabetic characters.</p>	<p>Change the test if necessary.</p>
125	?	<p>ILLEGAL ABBREVIATED RELATION CONDITION.</p> <p>Abbreviated conditions cannot combine class conditions, sign conditions, and condition-name conditions. The compiler-generated code for the indicated condition will probably yield false results.</p>	<p>Complete the abbreviated condition to remove any ambiguity and recompile the program.</p>
126	?	<p>ILLEGAL RELATION CONDITION.</p> <p>Certain conditions and logical operators cannot be combined. These are indicated in the chart in the introduction to the Procedure Division in this manual. The compiler-generated code for these illegal conditions will probably yield false results.</p>	<p>Correct the illegal condition and recompile the program.</p>
127	?	<p>SHAKY RELATION CONDITION.</p> <p>The indicated condition compares a non-numeric item with a Computational or Computational-3 item. The compiler generates a non-numeric comparison. This may cause false results since certain packed or binary numbers may have bit configurations that happen to be identical to alphanumeric characters.</p>	<p>Some programmers use this type of condition to avoid the need for defining binary or Computational-3 constants. Because this technique defeats the self-documenting feature of COBOL, the condition should be changed in most cases.</p>

Table C-1. COBOL Compiler Diagnostic Messages (cont.)

Message Number	Type	Message Text and Explanation	Programmer Response
128	?	<p>ILLEGAL IDENTIFIER IN EXAMINE.</p> <p>The EXAMINE identifier is either undefined or TALLY. The compiler ignores this EXAMINE statement.</p>	Change the identifier and recompile the program.
129	?	<p>ILLEGAL LITERAL IN EXAMINE.</p> <p>The literal is illegal under the following conditions:</p> <ul style="list-style-type: none"> • literal is not a single character. • category of literal is inconsistent with EXAMINE identifier. • non-numeric literal used with ALL figurative constant. <p>The compiler ignores this EXAMINE statement.</p>	Change the literal and recompile the program.
130	?	<p>TOO MANY LEVELS IN PERFORM-VARYING.</p> <p>More than three levels appear in the PERFORM VARYING statement. The compiler ignores the additional levels.</p>	Remove the excess levels and recompile the program.
131	?	<p>NUMBER OF TIMES MUST BE POSITIVE INTEGER.</p> <p>The literal in a PERFORM. . .TIMES statement is less than or equal to zero, or the identifier does not specify an integer. The compiler ignores the PERFORM.</p>	Change the literal or identifier in the PERFORM statement and recompile the program.
132	.	<p>RECORDING MODE NOT IMPLEMENTED.</p> <p>The RECORDING MODE clause is treated as a comment by the compiler.</p>	Use the \$CONTROL NOWARN subsystem command to turn off the warning message output, if so desired.
133	?	<p>SUBPROGRAMS MUST NOT BE SEGMENTED.</p> <p>More than one priority number occurred in a subprogram. This indicates segmentation which is illegal in subprograms. The compiler assigns the first priority number to the entire subprogram.</p>	Make the priority numbers the same.

Table C-1. COBOL Compiler Diagnostic Messages (Cont.)

Message Number	Type	Message Text and Explanation	Programmer Response
134	?	<p>SUBPROGRAM MUST NOT BE COMPILED AS MAIN.</p> <p>The source program being compiled contains a Linkage Section and a USING option in either its Procedure Division heading or an ENTRY statement, but it is being compiled as a main program (it lacks a \$CONTROL SUBPROGRAM control command). The compiler assumes that the subprogram mode is desired.</p>	<p>If the program is to be a main program, remove the Linkage Section statements and the USING option, then recompile. If the program is to be a subprogram and there are no other errors, no other action is needed. Recompile the program as a subprogram to suppress this message.</p>
135	?	<p>STATEMENT NOT ALLOWED IN MAIN PROGRAM.</p> <p>An ENTRY statement appears in a program compiled as a main program. The compiler ignores the ENTRY statement.</p>	<p>Either remove the ENTRY statement or recompile as a subprogram.</p>
136	.	<p>MISSING COMPUTER PARAGRAPHS.</p> <p>COBOL standards require that the SOURCE COMPUTER and OBJECT COMPUTER paragraphs be present. This compiler assumes that both paragraphs specify the HP 3000.</p>	<p>Supply the missing paragraphs to suppress this message and to maintain compatibility with the COBOL language.</p>
137	.	<p>77 ITEM PRECEDED BY 01-49 ITEM OR 77 IN FILE SECTION.</p> <p>Level 77 items should be grouped at the beginning of the Working-Storage Section. The compiler assumes an 01 level for the item.</p>	<p>Correct the error and recompile the program if the 01 level number causes incorrect results.</p>
138	?	<p>ILLEGAL USAGE ASSOCIATED WITH GROUP HAVING VALUE CLAUSE.</p> <p>A condition-name cannot be associated with a group containing items whose descriptions include JUSTIFIED or USAGE other than DISPLAY. The compiler treats the group as though there were no USAGE or JUSTIFIED clause.</p>	<p>Correct the error and recompile the program.</p>

Table C-1. COBOL Compiler Diagnostic Messages (cont.)

Message Number	Type	Message Text and Explanation	Programmer Response
139	?	<p>LITERAL-1 AND LITERAL-2 ARE OF DIFFERENT DATA TYPES.</p> <p>In an 88 level condition-name using the THRU option, both literal-1 and literal-2 must both be either numeric or alphabetic. The compiler ignores the THRU option.</p>	<p>Correct the error and re-compile the program.</p>
140	?	<p>LITERAL-1 MUST BE < LITERAL-2.</p> <p>In an 88 level condition-name using the THRU option, literal-1 must be lower in the collating sequence than literal-2. Thus the construction "5 THRU 1" must be changed to "1 THRU 5". The compiler ignores the THRU option.</p>	<p>Correct the error and re-compile the program.</p>
141	?	<p>EXPECTING A NUMERIC LITERAL.</p> <p>A numeric data item has been assigned a non-numeric literal value (PIC 99 VALUE "AB"). The compiler assigns a value of zero to the item.</p>	<p>Correct the error and re-compile the program.</p>
142	?	<p>EXPECTING FIGURATIVE CONSTANT ZERO.</p> <p>A numeric data item has been assigned a figurative constant other than zero. The compiler assigns a value of zero to the item.</p>	<p>Correct the error and re-compile the program.</p>
143	.	<p>NUMERIC LITERAL SPECIFIED FOR AN ALPHANUMERIC ITEM.</p> <p>When a numeric literal is specified for an alphanumeric item (PIC X), the compiler builds the literal as ASCII characters and considers it to be non-numeric. This construction is a syntactical error since alphanumeric literals should be enclosed in quotation marks.</p>	<p>Correct the error and re-compile the program if this error affects the operation of the program.</p>
144	?	<p>EXPECTING A FIGURATIVE CONSTANT OR NON-NUMERIC LITERAL.</p> <p>A value specified for a group item must be either a figurative constant or a non-numeric literal. The compiler ignores the literal.</p>	<p>Correct the error and re-compile the program.</p>

Table C-1. COBOL Compiler Diagnostic Messages (cont.)


Message Number	Type	Message Text and Explanation	Programmer Response
145	?	<p>01-LEVEL DATA NAME NOT UNIQUE IN WORKING-STORAGE SECTION.</p> <p>All data references must be unique. The compiler accepts the duplicate name, but all Procedure Division references to the duplicate names will reference the one defined last in the Data Division.</p>	<p>Correct the error and re-compile the program.</p>
146	?	<p>ITEM MUST BE NUMERIC/NUMERIC EDITED ON BLANK WHEN ZERO.</p> <p>The compiler ignores the BLANK WHEN ZERO clause.</p>	 <p>Correct the error and re-compile the program.</p>
147	?	<p>RENAME CLAUSE MUST BE THE LAST IN A LOGICAL RECORD.</p> <p>A level-66 entry is followed by a level number other than 01, 77, or 66. The compiler changes the level number to 01.</p>	<p>Correct the error and re-compile the program.</p>
148	.	<p>AN 88-LEVEL MUST NOT FOLLOW AN INDEX DATA ITEM.</p> <p>The compiler ignores the 88-level item.</p>	<p>Correct the error and re-compile the program if this error affects the operation of the program.</p>
149	.	<p>AN 88-LEVEL MUST NOT FOLLOW A 66-LEVEL.</p> <p>The compiler ignores the 88-level item.</p>	<p>Correct the error and re-compile the program if this error affects the operation of the program.</p>
150	!	<p>SYNTAX REQUIRES FILE NAME.</p> <p>The indicated statement requires a file name. If a file name was given, it may be invalid for one of the following reasons:</p> <ul style="list-style-type: none"> ● the name is misspelled. ● There is no SELECT statement for the file name. ● There is no FD statement for the file name. <p>The compiler is unable to generate any object code for the statement.</p>	<p>Correct the error and re-compile the program.</p>

Table C-1. COBOL Compiler Diagnostic Messages (cont.)

Message Number	Type	Message Text and Explanation	Programmer Response
151	!	<p>CANNOT CLOSE A SORT FILE.</p> <p>The opening and closing of sort files are internal functions. The compiler stops generating object code with this statement.</p>	<p>Correct the error and re-compile the program.</p>
152	.	<p>REEL, UNIT, AND LOCK OPTIONS ALL TREATED AS COMMENT.</p> <p>The compiler ignores these options.</p>	<p>The indicated options can be removed to suppress this message, if desired.</p>
153	.	<p>INPUT, OUTPUT, OR I-O CAN BE SPECIFIED ONLY ONCE.</p> <p>The word INPUT (or OUTPUT, or I-O) can appear only once in an OPEN statement.</p>	<p>Correct the syntax by using more than one OPEN statement.</p>
154	!	<p>CANNOT OPEN A SORT FILE.</p> <p>The opening and closing of sort files are internal functions. The compiler stops generating object code with this statement.</p>	<p>Correct the error and re-compile the program.</p>
155	.	<p>WITH NO REWIND NOT IMPLEMENTED.</p> <p>The compiler ignores this option.</p>	<p>The REWIND option can be removed to suppress this message, if desired.</p>
156	.	<p>REVERSED NOT IMPLEMENTED.</p> <p>The compiler ignores this option.</p>	<p>The REWIND option can be removed to suppress this message, if desired.</p>
157	!	<p>SYNTAX REQUIRES A RECORD NAME.</p> <p>WRITE and RETURN statements require a record name; file names are not acceptable in these statements. The compiler stops generating object code with this statement.</p>	<p>Correct the statement by supplying an appropriate record name and recompile the program.</p>

Table C-1. COBOL Compiler Diagnostic Messages (cont.)

Message Number	Type	Message Text and Explanation	Programmer Response
158	!	<p>MNEMONIC NAME ERROR.</p> <p>Only the following mnemonic names are permitted in the Special Names paragraph: SYSIN, SYSOUT, CONSOLE, TOP, and NO SPACE CONTROL. The compiler stops generating object code with this statement.</p>	<p>Correct the error in the Special Names paragraph and recompile the program.</p>
159	.	<p>ENDING OPTION NOT IMPLEMENTED.</p> <p>The compiler ignores the ENDING option.</p>	<p>The ENDING option can be removed to suppress this message, if desired.</p>
160	!	<p>USE PROCEDURE CANNOT SPECIFY FOR SORT FILE.</p> <p>The COBOL language does not permit the use of declarative procedures for sort files. The compiler stops generating object code with this statement.</p>	<p>Remove the sort file name from the USE statement and recompile the program.</p>
161	!	<p>MULTIPLE DEFINED PROCEDURE FOR SAME FILE.</p> <p>The same file name may appear in more than one USE statement. However, the indicated file name appears in USE statements that would call for the execution of two different procedures at the same time. The compiler stops generating object code with this statement.</p>	<p>Correct the error and recompile the program.</p>
162	.	<p>BEFORE OPTION NOT IMPLEMENTED.</p> <p>The compiler ignores the BEFORE option in the USE statement.</p>	<p>This option is not currently available under MPE.</p>
163	!	<p>CAN'T USING A SORT-FILE.</p> <p>A sort file may not appear as a USING file in a SORT statement.</p>	<p>Correct the error and recompile the program.</p>
164	!	<p>ILLEGAL INTO-FROM OPTION OR MOVE OPERAND.</p> <p>The operand named as a receiving field for a MOVE, a READ or RETURN . . . INTO, or a WRITE or RELEASE . . . FROM causes an illegal move operation. The compiler stops generating object code with this statement.</p>	<p>Review legal and illegal moves in this manual; (the rules for MOVE also apply to the INTO and FROM options). Correct the error and recompile the program.</p>

Table C-1. COBOL Compiler Diagnostic Messages (cont.)

Message Number	Type	Message Text and Explanation	Programmer Response
165	?	<p>NULL STRING NOT PERMITTED.</p> <p>There must be at least one character in a string; (the compiler supplies one character).</p>	Correct the string.
166	!	<p>CAN'T GIVING A SORT FILE.</p> <p>A sort file may not appear as a GIVING file in a SORT statement.</p>	Correct the error and re-compile the program.
167	!	<p>SORT CANNOT APPEAR IN DECLARATIVES.</p> <p>The COBOL language does not permit the use of SORT within Declarative procedures. The compiler stops generating object code with this statement.</p>	All SORT procedures must be outside the Declarative portion of the program. However, the program can set a flag in the Declarative procedures to control a sort located elsewhere in the Procedure Division. The program must be recompiled.
168	!	<p>KEY FIELD MUST BE LOCATED WITHIN LOGICAL RECORD.</p> <p>The sort key named for the file cannot be found within the record to be sorted. The compiler stops generating object code with this statement.</p>	Correct the error and re-compile the program.
169	!	<p>FILE NAME MUST BE RANDOM ACCESS FILE.</p> <p>A READ. .INVALID KEY statement has been found for a sequential file. The compiler stops generating object code with this statement.</p>	Either redefine the file or drop the INVALID KEY option from the READ statement. Recompile the program.
170	!	<p>SYNTAX REQUIRES A SECTION NAME.</p> <p>Input and output procedures for a SORT must reference section names, not paragraph names. The compiler stops generating object code with this statement.</p>	Provide valid section names for input and output procedures. Recompile the program.
171	!	<p>CAN'T READ A SORT FILE.</p> <p>Opening, closing, reading, and writing of sort files are all functions internal to the SORT statement. The compiler stops generating object code with this statement.</p>	Use RETURN and/or RELEASE statements to access sort file records. Recompile the program.

Table C-1. COBOL Compiler Diagnostic Messages (cont.)

Message Number	Type	Message Text and Explanation	Programmer Response
172	!	<p>FILE-NAME MUST BE A SEQUENTIAL ACCESS FILE.</p> <p>The compiler has found a READ statement without the INVALID KEY option for a file whose access mode was defined as random. The compiler stops generating object code with this statement.</p>	<p>Either correct the READ statement or change the access mode for the file to sequential. Recompile the program.</p>
173	!	<p>INVALID-KEY CLAUSE MISSING.</p>	<p>Correct the error and recompile the program.</p>
174	.	<p>ACCEPT FROM CONSOLE CANNOT EXCEED 31 CHARACTERS.</p> <p>At most, 31 characters may be accepted through the console for each ACCEPT statement.</p>	<p>Change the program and recompile.</p>
175	!	<p>CAN'T WRITE ON A SORT-FILE.</p> <p>Opening, closing, reading, and writing of sort files are all functions internal to the SORT statement. The compiler stops generating object code with this statement.</p>	<p>Use RETURN and/or RELEASE statements to access sort file records. Recompile the program.</p>
176	!	<p>RECORD-NAME AND IDENTIFIER MUSTN'T REFER TO SAME AREA.</p> <p>The compiler has found a READ, WRITE, RELEASE, or RETURN statement with an INTO or FROM option that refers to the same area as the record-name reserved for the file. The compiler stops generating object code with this statement.</p>	<p>Correct the identifier or remove the INTO or FROM option. Recompile the program.</p>
177	!	<p>SYNTAX REQUIRES A SORT-FILE.</p> <p>The compiler has found a RELEASE or RETURN statement for a file that was not defined as a sort-file.</p>	<p>Each sort-file requires an SD statement in the Data Division. If this is not to be a sort-file, remove the RELEASE and/or RETURN statements and replace them with READ or WRITE statements. Recompile the program.</p>

Table C-1. COBOL Compiler Diagnostic Messages (cont.)

Message Number	Type	Message Text and Explanation	Programmer Response
178	!	<p>CAN'T APPEAR IN NON-DECLARATIVE SECTION.</p> <p>The compiler has found a GO TO MORE-LABELS statement in the non-declarative portion of the Procedure Division. This statement is reserved for use in the Declarative portion only. The compiler stops generating object code with this statement.</p>	<p>Remove the GO TO MORE-LABELS statement and re-compile the program.</p>
179	.	<p>LEFT TRUNCATION MAY OCCUR.</p> <p>The receiving field is shorter than the sending field. The compiler generates object code that may cause truncation at the left-hand end of the receiving field.</p>	<p>As a general rule, it is a poor programming technique to allow operations that could cause the loss of the most significant digits of an arithmetic field. However, this warning may be ignored for special applications.</p>
180	!	<p>RECORDS FOR FILE-NAME-2 OR -3 NOT EQUAL TO -1.</p> <p>The record length for a GIVING or USING file is not the same as that defined for the sort-file. The compiler stops generating object code with this statement.</p>	<p>The record lengths for USING, GIVING, and sort-files must all be equal. Recompile the program.</p>
181	.	<p>NOT ALL DATA RECORDS HAVE THE SAME SIZE.</p> <p>When the RECORD CONTAINS integer-2 characters clause is specified, all data records in the file must be the same size. The maximum size in the record description is used.</p>	<p>Correct the error and re-compile the program.</p>
182	.	<p>RECORD-SIZE < SIZE IN RECORD CONTAINS CLAUSE.</p> <p>The maximum size in the record description is used.</p>	<p>Correct the error and re-compile the program.</p>
183	?	<p>RECORD-SIZE > SIZE IN RECORD CONTAINS CLAUSE.</p> <p>The maximum size in the record description is used.</p>	<p>Correct the error and re-compile the program.</p>
184	.	<p>RECORD-SIZE < MINIMUM SIZE IN RECORD CONTAINS CLAUSE.</p> <p>The maximum size in the record description is used.</p>	<p>Correct the error and re-compile the program.</p>

Table C-1. COBOL Compiler Diagnostic Messages (cont.)

Message Number	Type	Message Text and Explanation	Programmer Response
185	?	<p>INVALID QUALIFICATION OF A DATA NAME.</p> <p>An undefined identifier or an invalid qualification of a data name is encountered by the compiler.</p>	Correct the error and recompile the program.
186	.	<p>NON-88 LEVEL IN FILE SECTION WITH VALUE CLAUSE.</p> <p>The VALUE clause may not appear in the File Section, except at the 88-level. This compiler accepts the VALUE entry.</p>	Correct the error and recompile the program.
187	?	<p>FORMS MESSAGE MUSE BE < = 56 CHARACTERS AND TERMINATED BY A.</p> <p>A maximum of 56 characters, including the terminating period, are allowed in the FORMS MESSAGE parameter of the ASSIGN clause.</p>	Correct the error and recompile the program.
188	.	<p>ARITHMETIC OVERFLOW MAY OCCUR.</p> <p>If each identifier appearing in an arithmetic statement has the maximum value associated with its PICTURE clause, arithmetic overflow will occur.</p>	If overflow occurs, correct the error and recompile the program.
189	!	<p>MOVE OR COMPARE >32K BYTES</p> <p>A move statement or condition clause exceeds the maximum character limit.</p>	Reduce the size of the data items or break them into smaller sizes.
190	.	<p>LOCKING PARAMETER NOT L . . . IGNORED</p> <p>A character other than an L is in the place of the locking parameter in the ASSIGN clause. The parameter is ignored.</p>	If locking of this file is desired, change the parameter to an L.
196	?	<p>ACTUAL KEY NOT ON WORD BOUNDARY</p> <p>Actual keys must be computational items aligned on a word boundary. The compiler aligns the key to a word boundary, thus causing it to be one digit smaller than the user specified.</p>	Redefine the ACTUAL KEY so that it falls on a word boundary. 01-and 77-level items are automatically assigned on word boundaries. Recompile the program.
197	?	<p>ACTUAL KEY NON-COMPUTATIONAL ITEM.</p> <p>The compiler assumed USAGE IS COMPUTATIONAL for the ACTUAL KEY.</p>	Correct the error and recompile the program.

Table C-1. COBOL Compiler Diagnostic Messages (cont.)

Message Number	Type	Message Text and Explanation	Programmer Response
198	!	<p>ACTUAL KEY SIZE LESS THAN 4 BYTES.</p> <p>The ACTUAL KEY must be a computational item aligned on a word boundary (SYNCHRONIZED) with a PICTURE of S9(5) through S9(9). Computational items within this range are assigned four bytes by the compiler. The compiler stops generating object code with this statement.</p>	Correct the ACTUAL KEY and recompile the program.
199	?	<p>ACTUAL KEY SIZE REDUCED TO 4 BYTES.</p> <p>The picture given for the ACTUAL KEY was larger than S9(9). The compiler assumes a picture of S9(9).</p>	Correct the ACTUAL KEY and recompile the program. (Notice that this may alter record lengths when the ACTUAL KEY is a part of the record.)
200	!	<p>USL FILE OVERFLOW.</p> <p>USL file overflow may occur under the following conditions:</p> <ul style="list-style-type: none"> • A \$CONTROL USLINIT command may be missing. • The default size of 1024 records assigned to \$NEWPASS may be too small. • There may not be enough records left in \$OLDPASS. <p>Additional information about the files manipulated by the compiler subsystem may be found in the appendix section of this reference manual.</p>	Check to be certain that the \$CONTROL USLINIT command is not missing. If that command is present, build a larger USL file and recompile.
201-211		(these messages are internal compiler errors.)	Submit to HP systems engineer.
212	!	<p>CODE SEGMENT EXCEEDS 16K</p> <p>The object code generated by the compiler exceeds the maximum segment size. The paragraph name where the overflow occurred is displayed.</p>	Use sections with priority numbers to segment your program.
213-225		(These message numbers are not used.)	
226	?	OCCURS. . . DEPENDING ON MUST BE THE LAST ITEM IN A RECORD.	Correct error and recompile.
227	.	<p>SIZE OF RECEIVING FIELD > 18 CHARACTERS.</p> <p>A numeric-to-alphanumeric MOVE results in data that is right-justified with zero fill on the left.</p>	Correct error and recompile.

Table C-1. COBOL Compiler Diagnostic Messages (cont.)

Message Number	Type	Message Text and Explanation	Programmer Response
228	.	<p>VALUE LITERAL LARGER THAN ITEM SIZE.</p> <p>The value literal contains more characters than the PICTURE clause specifies.</p>	Correct the error and recompile.
229	?	<p>ILLEGAL INDEXING OF KEY.</p> <p>The compiler has found a SEARCH ALL statement with a compound condition which specifies different indexing or subscripting for the different conditions. Since the SEARCH ALL statement manipulates the index or subscript in a non-sequential manner, the index or subscript settings must be identical in each of the conditions. The compiler assumes that the key references are identical within the SEARCH ALL statement compound conditions.</p>	Correct the conditions and recompile the program.
230	?	<p>UNDEFINED GOTO..</p> <p>Following this message, the compiler lists each paragraph-name where a GO TO. statement was found, but that had no related ALTER statement. For each of these GO TO. statements, the compiler generates a branch to an address which will cause the program to abort if executed.</p>	Supply ALTER statements for the paragraphs listed.
231	!	<p>SEARCH KEY MUST NOT BE A TABLE OR SUBORDINATE TO A TABLE.</p> <p>The compiler is unable to generate code for the illegal key. The compiler stops generating object code with this statement.</p>	Correct the error and recompile the program.
232	.	<p>SAME SORT AREA CLAUSE CONTAINS NO SORT FILE.</p> <p>The compiler treats the clause as a SAME AREA clause.</p>	This message may indicate that a sort file has been improperly defined. Correct the error and recompile if necessary.
233	?	<p>COMP SYNC ITEMS MUST BE ON WORD BOUNDARY.</p>	Correct the error and recompile the program.

Table C-1. COBOL Compiler Diagnostic Messages (cont.)

Message Number	Type	Message Text and Explanation	Programmer Response
234	?	<p>REDEFINES OBJECT NOT IMMEDIATELY PRECEDING SUBJECT.</p> <p>The redefinition of an item must immediately follow the original description of the item being redefined. The compiler ignores the illegal redefinition.</p>	<p>Correct the error and recompile the program.</p>
235	!	<p>UNDEFINED PROCEDURE NAMES.</p> <p>Following this message, the compiler lists any procedure-names that are mentioned in GO TO, PERFORM, or ALTER statements, but that do not appear as valid procedure-names within the program. The compiler stops generating object code with this message.</p>	<p>Misspelling a procedure-name is a common cause of this message. Correct the error and recompile the program.</p>
236	.	<p>ILLEGAL LEVEL NUMBER.</p> <p>The compiler has found an elementary item followed by another elementary item which has a higher level number (02, 05, 04, 02, for example). The compiler treats the item as though its level number were the same as the previous legal item (02, 05, 02, 02, in the example above).</p>	<p>This message may be ignored if it does not affect the operation of the program. Correct the error and recompile the program to suppress this message.</p>
237	!	<p>NO EXIT EXISTS IN PROGRAM.</p> <p>The program contains no valid STOP RUN, EXIT PROGRAM, or GOBACK statement. The program cannot be executed since the operator would have to manually abort the run.</p>	<p>Notice that EXIT PROGRAM is not a valid statement for terminating a main program. Correct the error by adding a STOP RUN or GOBACK statement to the program and recompile.</p>

Table C-1. COBOL Compiler Diagnostic Messages (cont.)

Message Number	Type	Message Text and Explanation	Programmer Response
238	?	<p>KEYS IN COMPOUND CONDITION ARE INCOMPLETE OR DUPLICATE.</p> <p>The compiler has found a set of illegal compound conditions in a SEARCH ALL statement. Each condition in the SEARCH ALL must name a key item as either subject or object of the condition. The same key may not appear in more than one condition. When multiple keys are named in the KEY clause and a key other than the first is named in a condition, then all preceding keys must also appear in the compound conditions associated with this SEARCH ALL statement. The compiler generates object code for compound conditions up to but not beyond the point where the error was found.</p>	<p>Correct the condition(s) and recompile the program.</p>
239	?	<p>INVALID KEY IN SEARCH-ALL CONDITION.</p> <p>A condition in a SEARCH ALL statement names a key that does not belong to the table to be searched. The compiler ignores the condition.</p>	<p>Correct the condition and recompile the program.</p>
240	!	<p>CONDITION NAME VALUE MUST CONTAIN SINGLE LITERAL.</p> <p>A condition-name in a SEARCH ALL condition specifies a range of values. When a condition-name condition is used in a SEARCH ALL statement, the condition-name can be associated with only a single literal. The compiler ignores the condition.</p>	<p>Correct the conditional statement and recompile the program.</p>
241	?	<p>TOO MANY CONDITIONS IN SEARCH-ALL (MAXIMUM = 12).</p> <p>Since each condition in a SEARCH ALL statement must name a key item and the maximum allowable number of keys is 12, there can be no more than 12 conditions named in the statement. The compiler ignores all conditions after the 12th.</p>	<p>Correct the conditions in the SEARCH ALL statement and recompile the program.</p>

Table C-1. COBOL Compiler Diagnostic Messages (cont.)

Message Number	Type	Message Text and Explanation	Programmer Response
242	?	<p>ILLEGAL CONDITION IN SEARCH-ALL STMT.</p> <p>A key item must appear as either the subject or object of each condition in a SEARCH ALL statement, but not both. The compiler ignores the illegal condition.</p>	Correct the condition and recompile the program.
243	.	<p>NON 88-LEVEL IN LINKAGE SECTION WITH VALUE CLAUSE.</p> <p>The VALUE clause may not appear in the Linkage Section except at the 88-level. The compiler ignores the VALUE entry.</p>	Correct the error and recompile the program.
244	!	<p>DEPENDING-ON ITEM MUST BE AN INTEGER.</p> <p>The item named in this DEPENDING ON clause is defined with digits to the right of the decimal point. DEPENDING ON items must not have decimal positions. The compiler stops generating object code with this statement.</p>	Correct the error and recompile the program.
245	?	<p>ILLEGAL SEARCH-IDENTIFIER.</p> <p>The compiler issues this message to indicate a number of different conditions; the compiler action varies according to the format of the SEARCH statement:</p> <ul style="list-style-type: none"> ● When the VARYING option names an item which is not an integer (contains no decimal positions), the compiler ignores the VARYING option. ● When the identifier named in a SEARCH or SEARCH ALL statement is undefined or is not a table or does not include an INDEXED BY clause, the compiler ignores the statement. ● When a table named by a SEARCH ALL statement does not contain any KEY items, the statement is ignored. 	Correct the error and recompile the program.
246	?	<p>ILLEGAL PAIR OF OPERANDS IN SET STATEMENT.</p> <p>The SET statement contains illegal subject-object pairs. The compiler ignores the statement</p>	Review the rules for the SET statement. Correct the error and recompile the program.

Table C-1. COBOL Compiler Diagnostic Messages (cont.)

Message Number	Type	Message Text and Explanation	Programmer Response
247	?	<p>ILLEGAL IDENTIFIER IN SET STATEMENT.</p> <p>The compiler ignores the illegal identifier.</p>	Correct the error and re-compile the program.
248	.	<p>VALUE CLAUSE IN AN ITEM SUB-ORDINATED TO AN OCCURS TABLE.</p> <p>The VALUE clause is not permitted within an entry subordinate to an OCCURS.</p>	If a value is desired, define the value first, then re-define it with the table description.
249	?	<p>NO OF INDICES DOES NOT MATCH NO OF DIMENSIONS.</p> <p>The number of indices or subscripts used to access an item in a table does not match the number of levels defined for the table. When too many indices are mentioned, the compiler simply ignores the excess indices. When too few indices are mentioned, the compiler assumes the first occurrence for each of the missing indices.</p>	Correct the error and re-compile the program.
250	?	<p>ILLEGAL NAME OF TABLE ELEMENT.</p> <p>A reference to a table item refers to an undefined item. The compiler changes the reference to TALLY.</p>	Correct the error and re-compile the program.
251	?	<p>ILLEGAL SUBSCRIPT OR INDEX.</p> <p>The following are causes of this message:</p> <ul style="list-style-type: none"> ● In relative indexing, the amount to be added to the index is greater than 65K. The compiler changes the amount to 0. ● An illegal item is used as a subscript; i.e., the subscript is not an integer or an index-data-item. The compiler assumes a constant subscript of 1. ● A constant subscript specifies an occurrence outside the bounds of the table. The compiler generates code for the subscript as though it were legal, thus accessing whatever happens to be at the location. ● The specified index does not belong to the same table as the table item referenced. The compiler accepts the index as though it were legal, thus yielding unpredictable results. 	Correct the error and re-compile the program.

Table C-1. COBOL Compiler Diagnostic Messages (cont.)

Message Number	Type	Message Text and Explanation	Programmer Response
252	.	<p>NUMERIC USAGE WITH NON-NUMERIC PIC. PIC 9 IS ASSUMED.</p> <p>The indicated item specifies a USAGE other than DISPLAY for X-type data. The compiler assumes PIC 9 with the specified usage for the item.</p>	Correct the error and re-compile the program.
253	?	<p>MIXED SUBSCRIPTING AND INDEXING.</p> <p>The COBOL language does not permit the mixing of subscripts and indices in a reference to a table item. This compiler will process the reference correctly.</p>	To maintain compatibility with the COBOL language, alter the reference so that it contains all subscripts or all indices.
254	?	<p>SUBSCRIPT MUST BE ONE-WORD INTEGER > 0.</p> <p>The indicated constant subscript is less than or equal to zero or greater than 65K. The compiler assumes a subscript of 1.</p>	Correct the error and re-compile the program.
255	?	<p>DEVICE NAME MORE THAN 8 CHARACTERS.</p> <p>The device name in the SELECT . . . ASSIGN clause is limited to 8 characters.</p>	Correct the error and re-compile the program.
401	*	<p>UNABLE TO OPEN FILE <i>filename</i>.</p> <p>The compiler was unable to open the file whose formal file designator is <i>filename</i>. Compilation terminates.</p>	Supply the file.
402	*	<p>UNABLE TO USE FILE <i>filename</i>.</p> <p>The attributes of the file whose formal file designator is <i>filename</i> did not match those needed by the compiler. A file information display is sent to \$STDLIST and compilation terminates.</p>	Compare the file attributes given in the file information display with those required.
403	*	<p>END-OF-FILE ON FILE <i>filename</i>.</p> <p>An unexpected end-of-file was encountered on the file whose formal file designator is <i>filename</i>. A file information display is sent to \$STDLIST and compilation terminates.</p>	Supply a larger file.

Table C-1. COBOL Compiler Diagnostic Messages (cont.)

Message Number	Type	Message Text and Explanation	Programmer Response
404	*	<p>READ/WRITE FAILURE ON <i>filename</i>.</p> <p>An error has occurred on the file whose formal file designator is <i>filename</i>. A file information display is sent to \$STDLIST and compilation terminates.</p>	<p>Use the file information display to determine the nature of the error.</p>
405	*	<p>UNABLE TO CLOSE FILE <i>filename</i>.</p> <p>The compiler was unable to close the file whose formal file designator is <i>filename</i>. A file information display is sent to \$STDLIST and compilation terminates.</p>	<p>Use the file information display to determine the reason for the error.</p>

Table C-2. Object Program Diagnostic Messages

Message Number	Message Text and Explanation	Programmer Response
520	<p>TRYING TO CLOSE AN UNOPENED FILE</p> <p>The job is aborted.</p>	<p>Correct error in program and recompile.</p>
521	<p>CLOSE UNSUCCESSFUL</p> <p>The job is aborted.</p>	<p>To find out why the CLOSE was unsuccessful, check in <i>MPE Intrinsic Reference Manual</i>; try to run the program again.</p>
540	<p>TRYING OPEN, FILE ALREADY OPEN</p> <p>The job is aborted.</p>	<p>Remove the redundant OPEN statement(s) or insert necessary CLOSE statement(s) and recompile the program.</p>
542	<p>OPEN SERVICE NOT GRANTED</p> <p>The job is aborted.</p>	<p>To find out why the service was not granted, check in <i>MPE Intrinsic Reference Manual</i>; try to run the program again.</p>
543	<p>AREA ALREADY IN USE BY SAME AREA CLAUSE</p> <p>The job is aborted.</p>	<p>Correct the error and recompile the program.</p>
550	<p>TRY TO READ CLOSED FILE, OR FILE NOT OPEN</p> <p>The job is aborted.</p>	<p>Correct the program.</p>
551	<p>READ SERVICE NOT GRANTED</p> <p>The job is aborted.</p>	<p>To find out why the service was not granted, check in <i>MPE Intrinsic Reference Manual</i>; try to run the program again.</p>

Table C-2. Object Program Diagnostic Messages (cont.)

Message Number	Message Text and Explanation	Programmer Response
610	<p>TRYING WRITE, FILE NOT OPEN YET</p> <p>The job is aborted.</p>	Correct the program.
611	<p>WRITE SERVICE NOT GRANTED</p> <p>The job is aborted.</p>	<p>To find out why the service was not granted, check in <i>MPE Intrinsic Reference Manual</i>; try to run the program again.</p>
630	<p>ERROR DURING WRITE OF USER LABEL</p> <p>The job is aborted.</p>	
631	<p>USER LABEL SPACE UNALLOCATED OR ATTEMPT TO WRITE BEYOND LABEL LIMIT</p> <p>The job is aborted. (When the file is input, this message means that the user has tried to read more user's labels than exist on the file; when the file is output, this message means that the user has tried to write more than 8 user's labels.)</p> <p>On output a user has attempted to write a label on a file which was created using a :BUILD command and has no label allocated in the file.</p>	
706	<p>UNDERFLOW IN EXPONENTIATE</p> <p>The result of an exponentiate falls in the range -10^{-28} to $+10^{-28}$</p>	Check operands.
707	<p>OVERFLOW IN EXPONENTIATE</p> <p>The result of an exponentiation exceeds $\pm 10^{56}$</p>	Check operands.
708	<p>UNDEFINED RESULT FROM EXPONENTIATE</p> <p>An attempt was made to compile a^b where $a \leq 0$ and b is not an integer.</p>	Check operands and rewrite program.
710	<p>ILLEGAL DECIMAL DIGIT IN <<mnemonic>>.</p> <p><<mnemonic>> is one of the following: ADDD, SUBD, CMPD, SLD, NSLD, SRD, DIVD, and MPYD. An illegal packed decimal digit was encountered in execution. The invalid data is printed on the list device and the program is aborted.</p>	Correct data and re-run program.

Table C-2. Object Program Diagnostic Messages (cont.)

Message Number	Message Text and Explanation	Programmer Response
711	<p>ILLEGAL SOURCE DIGIT IN CONVERSION <<mnemonic>>.</p> <p><<mnemonic>> is one of the following: CVDA, CVAD, CVDB. Replace illegal digit with zero and restart.</p>	<p>Correct data, if necessary, and re-run program.</p>
750	<p>DEPENDING-ON IDENTIFIER OUT OF BOUNDS</p> <p>This error only occurs when the user selects the BOUNDS option of the \$CONTROL compiler subsystem command.</p>	<p>Correct program by removing bounds violation.</p>
751	<p>SUBSCRIPT/INDEX OUT OF BOUNDS</p> <p>This error only occurs when the user selects the BOUNDS option of the \$CONTROL compiler subsystem command.</p>	<p>Correct program by removing bounds violation.</p>
752	<p>INVALID I/O TYPE</p> <p>(KSAM files only). I-O type specified as parameter in call to CKOPEN or CKOPENSHR is not 0, 1, or 2.</p>	
753	<p>ILLEGAL KSAM OPERATION</p> <p>(KSAM files only). An illegal operation attempted such as call to CKDELETE or CKREWRITE not preceded by a call to a read procedure; or operation not consistent with access type such as an attempt to read a file opened for output only or vice versa.</p>	

Table C-3. COBOL Status Parameter Values Returned by KSAM

STATUS VALUE	MEANING	ACTION
"00"	SUCCESSFUL COMPLETION – I/O operation was completed successfully.	None.
"02"	SUCCESSFUL COMPLETION, DUPLICATE KEY – Write or rewrite operation was successful; a duplicate key was written for a key that is allowed duplicates.	None required, returned for information only.
"10"	AT END – End-of-file or beginning-of-file reached during sequential or random read. There is no next logical record in ascending key order.	Usually none. This result is a signal to close the file or perform another end-of-file action.
"21"	INVALID KEY, SEQUENCE ERROR – Attempt was made to write a record with a primary key that is out of sequence when the file was opened for sequential access.	Check the primary key value in the record being written. If you don't want sequence checking, re-open the file for random or dynamic access.
"22"	INVALID KEY, DUPLICATE KEY – Attempt was made to write or rewrite a record with a key value that duplicates a key value in an existing record, and duplicates are not allowed.	Check the key values. If possible change them to avoid the duplication. If duplicate keys must be written, create the file again allowing duplicates for the key and then copy the old file to the new file with FCOPY.
"23"	INVALID KEY, NO RECORD FOUND – Attempt to access record identified by a key with CKSTART or CKREADBYKEY, but no record is found with the specified key value at the specified key location.	Check the <i>keyvalue</i> , <i>keylength</i> , and <i>keylocation</i> parameters in the call. Correct if necessary. If record that cannot be found should be in the file, you may want to list the data file with FCOPY.
"24"	INVALID KEY, BOUNDARY VIOLATION – An attempt was made to write beyond the externally defined end of file.	
"30"	LOCK DENIED – File was locked by another process.	Wait until process locking file unlocks it – try again or lock file with <i>lockcond</i> = 1.
"31"	UNLOCK DENIED – File was not locked by calling process.	Before calling CKUNLOCK to unlock a shared file it must have been locked by a call to CKLOCK.
"9n"	FILE SYSTEM ERROR - Where <i>n</i> is a binary number between 0 and 255 corresponding to a File System Error code. (Refer to the MPE Intrinsic Reference Manual, table 10-4.)	Within your program you can call CKERROR to convert the number to a displayable value and then display it. Look up the value in Intrinsic table and perform any suggested action.

APPENDIX D

COBOL/3000 Reserved Words

ACCEPT	COBOL	DIVISION
ACCESS	CODE	DOWN
ACTUAL	COLUMN	ELSE
ADD	COMMA	END
ADDRESS	COMP	ENDING
ADVANCING	COMP-3	ENTER
AFTER	COMPUTATIONAL	ENTRY
ALL	COMPUTATIONAL-3	ENVIRONMENT
ALPHABETIC	COMPUTE	EQUAL
ALTER	CONFIGURATION	ERROR
ALTERNATE	CONSOLE	EVERY
AND	CONTAINS	EXAMINE
ARE	CONTROL	EXIT
AREA	CONTROLS	FD
AREAS	COPY	FILE
ASCENDING	CORR	FILE-CONTROL
ASSIGN	CORRESPONDING	FILE-LIMIT
AT	CURRENCY	FILE-LIMITS
AUTHOR	CURRENT-DATE	FILLER
BEFORE	DATA	FINAL
BEGINNING	DATE-COMPILED	FIRST
BLANK	DATE-WRITTEN	FOOTING
BLOCK	DE	FOR
BY	DECIMAL-POINT	FROM
CALL	DECLARATIVES	GENERATE
CF	DEPENDING	GIVING
CH	DESCENDING	GO
CHARACTERS	DETAIL	GOBACK
CLOCK-UNITS	DISPLAY	GREATER
CLOSE	DIVIDE	GROUP

HEADING	MOVE	RECORD
HIGH-VALUE	MULTIPLE	RECORDING
HIGH-VALUES	MULTIPLY	RECORDS
I-O	NEGATIVE	REDEFINES
I-O-CONTROL	NEXT	REEL
IDENTIFICATION	NO	RELEASE
IF	NOT	REMAINDER
IN	NOTE	REMARKS
INDEX	NUMBER	RENAMES
INDEXED	NUMERIC	REPLACING
INDICATE	OBJECT-COMPUTER	REPORT
INITIATE	OCCURS	REPORTING
INPUT	OF	REPORTS
INPUT-OUTPUT	OFF	RERUN
INSTALLATION	OMITTED	RESERVE
INTO	ON	RESET
INVALID	OPEN	RETURN
IS	OPTIONAL	REVERSED
JUST	OR	REWIND
JUSTIFIED	OUTPUT	RF
KEY	PAGE	RH
KEYS	PAGE-COUNTER	RIGHT
LABEL	PERFORM	ROUNDED
LAST	PF	RUN
LEADING	PH	SAME
LEFT	PIC	SD
LESS	PICTURE	SEARCH
LIMIT	PLUS	SECTION
LIMITS	POSITION	SECURITY
LINE	POSITIVE	SEEK
LINE-COUNTER	PROCEDURE	SEGMENT-LIMIT
LINES	PROCEED	SELECT
LINKAGE	PROCESSING	SENTENCE
LOCK	PROGRAM	SEQUENTIAL
LOW-VALUE	PROGRAM-ID	SET
LOW-VALUES	QUOTE	SIGN
MEMORY	QUOTES	SIZE
MODE	RANDOM	SORT
MODULES	RD	SOURCE
MORE-LABELS	READ	SOURCE-COMPUTER
		SPACE

SPACES
SPECIAL-NAMES
STANDARD
STATUS
STOP
SUBTRACT
SUM
SYNC
SYNCHRONIZED
SYSIN

SYSOUT
TALLY
TALLYING
TAPE
TERMINATE
THAN
THEN
THROUGH
THRU
TIMES

TIME-OF-DAY
TO
TOP
TYPE
UNIT
UNTIL
UP
UPON
USAGE
USE

USING
VALUE
VALUES
VARYING
WHEN
WITH
WORDS
WORKING-STORAGE
WRITE
ZERO

ZEROES
ZEROS

APPENDIX E

COBOL Language Formats



This appendix contains the COBOL/3000 language formats. It is intended to display complete and syntactically correct formats. It is not intended to imply legal combinations of language elements.

In this presentation, bold-face upper-case type indicates the name of the paragraph, section, or verb for which syntax is provided. Bold-face upper- and lower-case type indicates the different formats that can be used with that syntax. The syntax itself follows these bold-face headings and sub-headings.

*Note: A * preceding a heading flags a Report Writer option which is not implemented in the COBOL/3000 compiler. Report Writer options are recognized by the parser and an error diagnostic will be generated by the compiler.*

IDENTIFICATION DIVISION

IDENTIFICATION DIVISION.

PROGRAM-ID. program-name.

[AUTHOR. [comment-entry] ...]

[INSTALLATION. [comment-entry] ...]

[DATE-WRITTEN. [comment-entry] ...]

[DATE-COMPILED. [comment-entry] ...]

[SECURITY. [comment-entry] ...]

[REMARKS. [comment-entry] ...]

ENVIRONMENT DIVISION

ENVIRONMENT DIVISION.

CONFIGURATION SECTION

CONFIGURATION SECTION.

SOURCE-COMPUTER

Format 1:

SOURCE-COMPUTER. COPY library-name

$$\left[\underline{\text{REPLACING}} \text{ word-1 } \underline{\text{BY}} \left\{ \begin{array}{l} \text{word-2} \\ \text{identifier-1} \\ \text{literal-1} \end{array} \right\} \right.$$
$$\left. \left[, \text{ word-3 } \underline{\text{BY}} \left\{ \begin{array}{l} \text{word-4} \\ \text{identifier-2} \\ \text{literal-2} \end{array} \right\} \right] \dots \right].$$

Format 2:

SOURCE-COMPUTER. computer-name.

OBJECT-COMPUTER

Format 1:

OBJECT-COMPUTER. COPY library-name.

$$\left[\underline{\text{REPLACING}} \text{ word-1 } \underline{\text{BY}} \left\{ \begin{array}{l} \text{word-2} \\ \text{identifier-1} \\ \text{literal-1} \end{array} \right\} \right.$$
$$\left. \left[, \text{ word-3 } \underline{\text{BY}} \left\{ \begin{array}{l} \text{word-4} \\ \text{identifier-2} \\ \text{literal-2} \end{array} \right\} \right] \dots \right].$$

Format 2:

OBJECT-COMPUTER. computer-name

[, MEMORY SIZE integer { WORDS
CHARACTERS }]

[, SEGMENT-LIMIT IS priority-number]

SPECIAL-NAMES

Format 1:

SPECIAL-NAMES. COPY library-name

[REPLACING word-1 BY { word-2
identifier-1
literal-1 }]

[, word-3 BY { word-4
identifier-2
literal-2 }] . . .] .

Format 2:

SPECIAL-NAMES. [function-name IS mnemonic-name] . . .

[, CURRENCY SIGN IS literal] [, DECIMAL-POINT IS COMMA] .

INPUT-OUTPUT SECTION

INPUT-OUTPUT SECTION.

FILE-CONTROL

Format 1:

FILE-CONTROL. COPY library-name

[REPLACING word-1 BY { word-2
identifier-1 }
literal-1]

[, word-3 BY { word-4
identifier-2 }] ...] .

Format 2:

FILE-CONTROL. { SELECT [OPTIONAL] file-name

ASSIGN TO [integer-1] system-file-name-1 [, system-file-name-2] ...

[FOR MULTIPLE { REEL
UNIT }] [, RESERVE { integer-2 }
NO] ALTERNATE [AREA
AREAS]]

[, { FILE-LIMIT IS
FILE-LIMITS ARE } { data-name-1 }
literal-1] THRU { data-name-2 }
literal-2]

[, { data-name-3 }
literal-3] THRU { data-name-4 }
literal-4] ...]

[, ACCESS MODE IS { SEQUENTIAL
RANDOM }]

[, PROCESSING MODE IS SEQUENTIAL]

[, ACTUAL KEY IS data-name-5] . } ...

Format 3:

FILE-CONTROL. { SELECT file-name ASSIGN TO system-file-name-1

[, system-file-name-2] ... OR system-file-name-3

[, system-file-name-4] ... [FOR MULTIPLE { REEL
UNIT }] . } ...

I-O-CONTROL

Format 1:

I-O-CONTROL. COPY library-name

[REPLACING word-1 BY { word-2
identifier-1
literal-1 }]

[, word-3 BY { word-4
identifier-2
literal-2 }] ...] .

Format 2:

I-O-CONTROL. [; RERUN [ON { file-name-1
system-file-name }]]

EVERY { [END OF] { REEL
UNIT } OF file-name-2 }
integer-1 RECORDS
integer-2 CLOCK-UNITS
condition-name } ...]

[; SAME [{ RECORD
SORT }] AREA FOR file-name-3 { , file-name-4 } ...] ...

[; MULTIPLE FILE TAPE CONTAINS file-name-5 [POSITION integer-3]

[, file-name-6 [POSITION integer-4]] ...]

Format 2:

FD file-name

[; BLOCK CONTAINS [integer-1 TO] integer-2 { RECORDS
CHARACTERS }]

[; DATA { RECORD IS
RECORDS ARE } data-name-1 [, data-name-2] . . .]

[; RECORDING MODE IS { F
V
U }]

; LABEL { RECORD IS
RECORDS ARE } { STANDARD
OMITTED
data-name-3 [, data-name-4] . . . }

[; RECORD CONTAINS [integer-3 TO] integer-4 CHARACTERS]

[; VALUE OF data-name-5 IS { data-name-6
literal-1 }]

[, data-name-7 IS { data-name-8
literal-2 }] . . .] .

*** Format 3:**

FD file-name

[; BLOCK CONTAINS [integer-1 TO] integer-2 { RECORDS
CHARACTERS }]

; LABEL { RECORD IS
RECORDS ARE } { STANDARD
OMITTED
data-name-1 [, data-name-2] . . . }

[; RECORD CONTAINS [integer-3 TO] integer-4 CHARACTERS]

; { REPORT IS
REPORTS ARE } report-name-1 [, report-name-2] . . .

[; VALUE OF data-name-3 IS { data-name-4
literal-1 }]

[, data-name-5 IS { data-name-6
literal-2 }] . . .] .

SORT DESCRIPTION

Format 1:

SD file-name; COPY library-name

[REPLACING word-1 BY { word-4
identifier-1 }
literal-1]

[, word-3 BY { word-4
identifier-2 }] . . .] .

Format 2:

SD file-name

[; DATA { RECORD IS
RECORDS ARE } data-name-1 [, data-name-2] . . .]

[; RECORD CONTAINS [integer-1 TO] integer-2 CHARACTERS] .

RECORD DESCRIPTION

Format 1:

01 data-name-1; COPY library-name

[REPLACING word-1 BY { word-2
identifier-1 }
literal-1]

[, word-3 BY { word-4
identifier-2 }] . . .] .

Format 2:

level-number { data-name-1 } [; REDEFINES data-name-2]
FILLER

[; BLANK WHEN ZERO]

[; { JUSTIFIED } RIGHT]
JUST

[; OCCURS { integer-1 TO integer-2 TIMES [DEPENDING ON data-name-3] }
integer-2 TIMES

[{ ASCENDING } KEY IS data-name-4 [, data-name-5] ...] ...
DESCENDING

[INDEXED BY index-name-1 [, index-name-2] ...]

[; { PICTURE } IS character-string]
PIC

[; { SYNCHRONIZED } [LEFT]]
SYNC RIGHT

[; [USAGE IS] { COMPUTATIONAL }]
COMP
DISPLAY
INDEX
COMPUTATIONAL-3
COMP-3

[; VALUE IS literal-3]

Format 3:

66 data-name-1; RENAMES data-name-2 [THRU data-name-3].

Format 4:

88 condition name

; { VALUE IS } literal-1 [THRU literal-2]
VALUES ARE

[, literal-3 [THRU literal-4]]

***REPORT GROUP DESCRIPTION**

Format 1:

01 data-name-1; COPY library-name

[REPLACING word-1 BY { word-2
 identifier-1
 literal-1 }]

[, word-3 BY { word-4
 identifier-2
 literal-2 }] . . .] .

***Format 2:**

01 [data-name-1]

[; LINE NUMBER IS { integer-1
PLUS integer-2
NEXT PAGE }]

[; NEXT GROUP IS { integer-3
PLUS integer-4
NEXT PAGE }]

; TYPE IS { REPORT HEADING
RH
PAGE HEADING
PH
 { CONTROL HEADING } { identifier-1
CH } { FINAL }
DETAIL
DE
 { CONTROL FOOTING } { identifier-2
CF } { FINAL }
PAGE FOOTING
PF
REPORT FOOTING
RF }

[; [USAGE IS] DISPLAY] .

***Format 3:**

level-number [data-name-1]

[; BLANK WHEN ZERO]

[; COLUMN NUMBER IS integer-1]

[; GROUP INDICATE]

[; { JUSTIFIED } RIGHT]
[; { JUST }]

[; LINE NUMBER IS { integer-2
PLUS integer-3 }]
[; NEXT PAGE]

[; { PICTURE } IS character-string]
[; { PIC }]

[; RESET ON { identifier-1 }]
[; FINAL]

{ ; SOURCE IS identifier-2
; SUM identifier-3 [,identifier-4] . . . [UPON data-name-2] }
[; VALUE IS literal-1]

[; [USAGE IS] DISPLAY] .

PROCEDURE DIVISION

PROCEDURE DIVISION [USING identifier-1 [, identifier-2] . . .] .

ACCEPT

ACCEPT identifier [FROM { SYSIN
mnemonic-name }]
CONSOLE]

ADD

Format 1:

ADD { identifier-1 } [, identifier-2] . . . TO identifier-m [ROUNDED]
[, literal-1] [, literal-2] . . .
[, identifier-n [ROUNDED]] . . .
[; ON SIZE ERROR imperative-statement]

Format 2:

ADD { identifier-1 } { identifier-2 } [, identifier-3] . . .
[, literal-1] [, literal-2] [, literal-3] . . .
GIVING identifier-m [ROUNDED]
[; ON SIZE ERROR impertive-statement]

Format 3:

ADD { CORRESPONDING } identifier-1 TO identifier-2 [ROUNDED]
{ CORR }
[; ON SIZE ERROR imperative-statement]

ALTER

ALTER procedure-name-1 TO [PROCEED TO] procedure-name-2
[, procedure-name-3 TO [PROCEED TO] procedure-name-4] . . .

CALL

CALL literal-1 [USING identifier-1 [identifier-2] . . .]

CLOSE

CLOSE file-name-1 [REEL / UNIT] [WITH { NO REWIND / LOCK }]
[, file-name-2 [REEL / UNIT] WITH [{ NO REWIND / LOCK }]] ...

COMPUTE

COMPUTE identifier-1 [ROUNDED] = { identifier-2 / literal-1 / arithmetic-expression }
[; ON SIZE ERROR imperative-statement]

COPY

COPY library-name

[REPLACING word-1 BY { word-2 / identifier-1 / literal-1 }
[, word-3 BY { word-4 / identifier-2 / literal-2 }]] ...] .

DECLARATIVES

DECLARATIVES.

{ section-name SECTION. declarative-sentence
{ paragraph-name. { sentence } ... } ... } ...

END DECLARATIVES.

DISPLAY

DISPLAY { literal-1 } [,literal-2] . . . [UPON { SYSOUT
CONSOLE
mnemonic-name }]

DIVIDE

Format 1:

DIVIDE { identifier-1 } INTO identifier-2 [ROUNDED]

[; ON SIZE ERROR imperative-statement]

Format 2:

DIVIDE { identifier-1 } INTO { identifier-2 }
literal-1 literal-2

GIVING identifier-3 [ROUNDED]

[; ON SIZE ERROR imperative-statement]

Format 3:

DIVIDE { identifier-1 } BY { identifier-2 }
literal-1 literal-2

GIVING identifier-3 [ROUNDED]

[; ON SIZE ERROR imperative-statement]

Format 4:

DIVIDE { identifier-1 } INTO { identifier-2 }
literal-1 literal-2

GIVING identifier-3 [ROUNDED] REMAINDER identifier-4

[; ON SIZE ERROR imperative-statement]

Format 5:

DIVIDE { identifier-1 } BY { identifier-2 }
 { literal-1 } { literal-2 }

GIVING identifier-3 [ROUNDED] REMAINDER identifier-4

[; ON SIZE ERROR imperative-statement]

ENTER

ENTER language-name [routine-name] .

ENTRY

ENTRY literal-1 [USING identifier-1 [, identifier-2] . . .]

EXAMINE

EXAMINE identifier

$$\left\{ \begin{array}{l} \left\{ \begin{array}{l} \underline{\text{TALLYING}} \\ \underline{\text{REPLACING}} \end{array} \right\} \left\{ \begin{array}{l} \underline{\text{UNTIL FIRST}} \\ \underline{\text{ALL}} \\ \underline{\text{LEADING}} \end{array} \right\} \text{literal-1 } [\underline{\text{REPLACING BY}} \text{ literal-2}] \\ \left\{ \begin{array}{l} \underline{\text{REPLACING}} \\ \underline{\text{REPLACING}} \end{array} \right\} \left\{ \begin{array}{l} \underline{\text{ALL}} \\ \underline{\text{LEADING}} \\ \underline{\text{[UNTIL] FIRST}} \end{array} \right\} \text{literal-3 } \underline{\text{BY}} \text{ literal-4} \end{array} \right\}$$

EXIT

EXIT [PROGRAM] .

***GENERATE**

GENERATE identifier

GO

Format 1:

GO TO [procedure-name-1]

Format 2:

GO TO procedure-name-1 [, procedure-name-2] . . .
[, procedure-name-n DEPENDING ON identifier]

GOBACK

GOBACK.

IF

IF condition ;THEN { statement-1
NEXT SENTENCE } [;ELSE { statement-2
NEXT SENTENCE }]

*INITIATE

INITIATE report-name-1 [, report-name-2] . . .

MOVE

Format 1:

MOVE { identifier-1
literal-1 } TO identifier-2 [, identifier-3] . . .

Format 2:

MOVE { CORRESPONDING
CORR } identifier-1 TO identifier-2

MULTIPLY

Format 1:

MULTIPLY { identifier-1 }
literal-1 } BY identifier-2 [ROUNDED]

[; ON SIZE ERROR imperative-statement]

Format 2:

MULTIPLY { identifier-1 }
literal-1 } BY { identifier-2 }
literal-2 }

GIVING identifier-3 [ROUNDED]

[; ON SIZE ERROR imperative-statement]

NOTE

NOTE character-string.

OPEN

OPEN { INPUT file-name-1 { REVERSED
WITH NO REWIND }
OUTPUT file-name-3 [WITH NO REWIND]
I-O file-name-5
[, file-name-2 [{ REVERSED
WITH NO REWIND }]] ... } ...
[, file-name-4 [WITH NO REWIND]] ... } ...
[, file-name-6] ...

PARAGRAPH NAMES

{ paragraph-name. { sentence}. . . } . . .

PERFORM

Format 1:

PERFORM procedure-name-1 [THRU procedure-name-2]

Format 2:

PERFORM procedure-name-1 [THRU procedure-name-2] { identifier-1 } TIMES
integer-1

Format 3:

PERFORM procedure-name-1 [THRU procedure-name-2] UNTIL condition-1

Format 4:

PERFORM procedure-name-1 [THRU procedure-name-2]

VARYING { index-name-1 } FROM { index-name-2 }
identifier-1 literal-2
identifier-2

BY { literal-3 } UNTIL condition-1
identifier-3

[AFTER { index-name-4 } FROM { index-name-5 }
identifier-4 literal-5
identifier-5]

BY { literal-6 } UNTIL condition -2
identifier-6

[AFTER { index-name-7 } FROM { index-name-8 }
identifier-7 literal-8
identifier-8]

BY { literal-9 } UNTIL condition-3]]
identifier-9

READ

READ file-name RECORD [INTO identifier]

{ ; AT END imperative-statement
; INVALID KEY imperative-statement }

RELEASE

RELEASE record-name [FROM identifier]

RETURN

RETURN file-name RECORD [INTO identifier]

; AT END imperative-statement

SEARCH

Format 1:

SEARCH identifier-1 [VARYING { index-name-1
identifier-1 }]

[; AT END imperative-statement-1]

; WHEN condition-1 { imperative-statement-2
NEXT SENTENCE }

[; WHEN condition-2 { imperative-statement-3
NEXT SENTENCE }] ...

Format 2:

SEARCH ALL identifier-1 [; AT END imperative-statement-1]

; WHEN condition-1 { imperative-statement-2
NEXT SENTENCE }

STOP

STOP { literal
RUN }

SUBTRACT

Format 1:

SUBTRACT { literal-1
identifier-1 } [, literal-2
[, identifier-2] . . .

FROM identifier-m [ROUNDED] [, identifier-n [ROUNDED]] . . .

[; ON SIZE ERROR imperative-statement]

Format 2:

SUBTRACT { literal-1
identifier-1 } [, literal-2
[, identifier-2] . . .

FROM { literal-m
identifier-m } GIVING identifier-n [ROUNDED]

[; ON SIZE ERROR imperative-statement]

Format 3:

SUBTRACT { CORRESPONDING
CORR } identifier-1 FROM identifier-2 [ROUNDED]

[; ON SIZE ERROR imperative-statement]

*TERMINATE

TERMINATE report-name-1 [, report-name-2] . . .

USE

Format 1:

USE AFTER STANDARD ERROR PROCEDURE on

$$\left(\begin{array}{l} \text{file-name-1 [, file-name-2] . . .} \\ \underline{\text{INPUT}} \\ \underline{\text{OUTPUT}} \\ \underline{\text{I-O}} \end{array} \right)$$

Format 2:

USE $\left\{ \begin{array}{l} \underline{\text{BEFORE}} \\ \underline{\text{AFTER}} \end{array} \right\}$ STANDARD $\left[\begin{array}{l} \underline{\text{BEGINNING}} \\ \underline{\text{ENDING}} \end{array} \right]$ $\left[\begin{array}{l} \underline{\text{REEL}} \\ \underline{\text{FILE}} \\ \underline{\text{UNIT}} \end{array} \right]$ LABEL PROCEDURE ON

$$\left(\begin{array}{l} \text{file-name-1 [, file-name-2] . . .} \\ \underline{\text{INPUT}} \\ \underline{\text{OUTPUT}} \\ \underline{\text{I-O}} \end{array} \right)$$

*Format 3:

USE BEFORE REPORTING identifier-1

WRITE

Format 1:

WRITE record-name [FROM identifier-1]

$$\left[\left\{ \begin{array}{l} \underline{\text{BEFORE}} \\ \underline{\text{AFTER}} \end{array} \right\} \text{ ADVANCING } \left\{ \begin{array}{l} \text{identifier LINES} \\ \text{integer LINES} \\ \text{mnemonic-name} \end{array} \right\} \right]$$

Format 2:

WRITE record-name [FROM identifier-1]

; INVALID KEY imperative-statement

APPENDIX F

Sample COBOL/3000 Program Output

This appendix presents an example of listings obtained during the compilation, preparation, and execution of a COBOL/3000 program. These listings appear at the end of the appendix; key elements are discussed below.

SYMBOL TABLE MAP

If the MAP option appears in a \$CONTROL command parameter list, the compiler turns on a software switch requesting generation of a symbol table map at the end of compilation. The symbol table map is suppressed if the switch is off. (The map switch is initially off when the compiler is invoked.) The symbol table map has two header lines: one for the file and data names, and the other for section and paragraph names. They are described below with range in print column numbers, header descriptor words, and meaning of contents shown under the header descriptors.

File And Data Names

The header lines for file and data names appear in the following format. Actual examples appear on the second page of the sample listings.

<u>Columns</u>	<u>Header Descriptor Word</u>	<u>Meaning</u>
1-4	LVL	Level number -- FD, SD, or two decimal digits: 01-49, 66, 77 or 88.
5-35	SOURCE NAME	The user-defined file name, data name, condition name, and index data name.
36-40	BASE	Data segment base: DB -- location is relative to DB+0. (This storage allocation is for the main program). OWN -- location is relative to the OWN base in the secondary DB area. (This storage allocation is for non-dynamic subprograms). LINK -- location is relative to the Q+3 address. (This storage allocation is for dynamic subprograms).

<u>Columns</u>	<u>Header Descriptor Word</u>	<u>Meaning</u>
		The base for a main program is DB+0, and the base for a subprogram (whether or not it is dynamic) is stored in Q+2. The base (a word address) for the program or subprogram currently in execution can always be found in DB-5.
41-46	DISPL	Displacement relative to the base -- a six-digit octal count in bytes.
49-54	SIZE	Size of the data item in number of bytes -- a six-digit octal number.
55-65	USAGE	The usage of the data item or type of file -- DISP - Display. COMP - Computational. COMP - SYNC - Computational - Synchronized. COMP - 3 - Computational - 3. INDEX - Index data item. SEQ - Sequential access file. RANDOM - Random Access File.
67-73	CATEGORY	The category of the data item: GROUP - a group item. DISP-N - Numeric Display. DISP-NS - Non-numeric display. A - Alphabetic. AN - Alphanumeric. NE - Numeric edited. N - Numeric. ANE - Alphanumeric edited.
75	R	REDEFINES - An "R" under this column indicates that the data-name redefines another data-name.
77	O	OCCURS - An "O" under this column indicates that an OCCURS clause has been specified for this data-name.
79	D	DEPENDING ON - A "D" under this column indicates that the data-name is the object, or contains the object, of the DEPENDING ON option of the OCCURS clause.
81	J	JUSTIFIED -- A "J" under this column indicates that a JUSTIFIED clause has been specified for this data-name.

<u>Columns</u>	<u>Header Descriptor Word</u>	<u>Meaning</u>
83-84	BZ	BLANK WHEN ZERO -- A "BZ" under these two columns indicates that a BLANK WHEN ZERO clause has been specified for this data-name.

Section And Paragraph Names

The header lines for section and paragraph names appear in the following format. Actual examples appear on the third page of the sample listing.

<u>Columns</u>	<u>Header Descriptor Word</u>	<u>Meaning</u>
1-31	SOURCE NAME	The programmer-defined section or paragraph name, or the compiler-created section name ILL'BRANCH. ILL'BRANCH is generated for either of the following two reasons: <ul style="list-style-type: none"> a) The last statement in the program is not an existing statement: STOP RUN, GOBACK, or EXIT PROGRAM. b) An alterable GOTO statement appears in the program.
32-34	S/P	Section or paragraph: S = Section P = Paragraph
36-49	INTERNAL NAME	Internal name assigned by the compiler and used by the segmenter (as described later in this appendix).
51-65	PB-RELATIVE LOC	PB - relative location with respect to its code module -- a six-digit octal word location.
68-80	PRIORITY NO.	The priority number of the current section or paragraph -- a two-digit (maximum) decimal number.

INTERNAL NAME ASSIGNMENT

The assignment of internal segment and entry point names is discussed below.

Segment Name

Each program unit contains an initialization segment which, at execution time, initializes various tables, and is always the first segment of the program unit.

The initialization segment name is assigned as follows:

For main programs: The first 15 characters of the PROGRAM-ID name with “-” stripped out.

For subprograms: The first 14 characters of the PROGRAM-ID name with “-” stripped out, followed by an apostrophe (’).

All consecutive sections which have the same priority-number constitute a program segment. The segment name is formed in this format:

xxxxxxxxxxxxdd’

where xxxxxxxxxxxx = the first section name within the segment, if the section name contains 12 or less characters; otherwise, it is the first 12 characters of the section name.

dd = the two-digit priority number for the section.

Entry Point Name

The entry point for the initialization segment is defined as the first 14 characters of the PROGRAM-ID name, with “-” stripped out, followed by the character ‘.

The entry point for the first executable code module is defined as follows:

For main program: The first 12 characters of the first section name (first paragraph name if there is no section), with “-” stripped out, followed by the two-digit priority number of the section, followed by the character ‘.

For subprograms: The first 15 characters of the PROGRAM-ID name with “-” stripped out.

The entry point for other code modules are assigned as follows:

For main program: The first 12 characters, with “-” stripped out, followed by the two-digit priority number of the section, followed by the character ‘.

For subprograms: The first 15 characters of the ENTRY names, with “-” stripped out.

EXAMPLES

Examples of internal name assignment are presented below:

For a Main Program:

PROGRAM-ID. CALLPROG.

S1 SECTION.

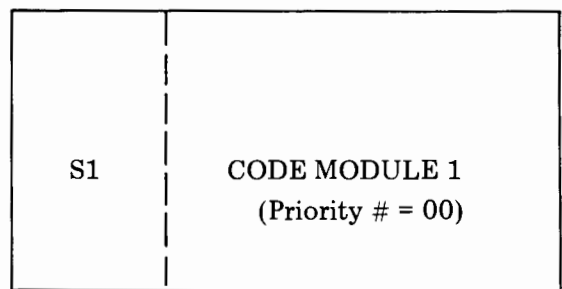
P1.

⋮

S2 SECTION 40.

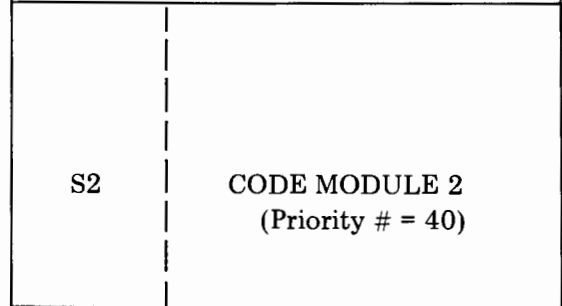
P2.

ENTRY POINT NAME
= S100'



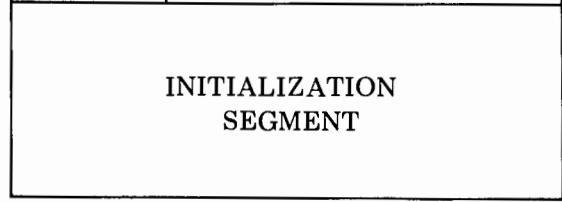
SEGMENT 2 NAME
= S100'

ENTRY POINT NAME
= S240'



SEGMENT 1 NAME
= S240'

ENTRY POINT NAME
= CALLPROG'



SEGMENT 0 NAME
= CALLPROG

For a Subprogram:

PROGRAM-ID. SUBPROG.

S1 SECTION.

⋮

P1.

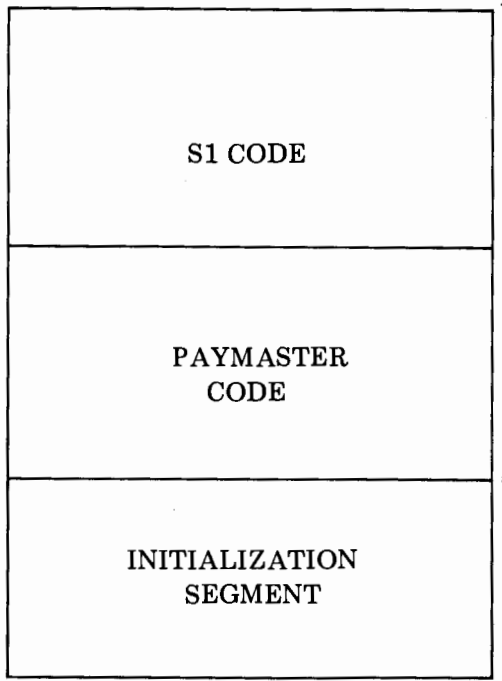
ENTRY "PAYMASTER"

⋮

ENTRY POINT NAME
= SUBPROG

ENTRY POINT NAME
= PAYMASTER

ENTRY POINT NAME
= SUBPROG'



SEGMENT 1 NAME
= SUBPROG

SEGMENT 0 NAME
= SUBPROG'

Sample Listings

The listings obtained from the compilation, preparation, and execution of the sample COBOL/3000 program, appear below:

```
000100*CONTROL LIST, MAP
000200 IDENTIFICATION DIVISION.
000300 PROGRAM-ID. TESTER.
000400 ENVIRONMENT DIVISION.
000500 DATA DIVISION.
000600 WORKING-STORAGE SECTION.
000700 01 A.
000800 02 FILLER PIC X(15) VALUE "HERE IT IS:".
000900 02 FILLER PIC X(30) VALUE "**START OF PROGRAM**".
001000 01 S9-AREA.
001100 05 S9 OCCURS 10 PIC S9.
001200 01 9-AREA.
001300 05 9X OCCURS 10 PIC 9.
001400 PROCEDURE DIVISION.
001500 MAINLINE SECTION.
001600 FIRST-PARA.
001700 DISPLAY A.
001800 MOVE "AIJRSZ19()" TO S9-AREA.
001900 DISPLAY S9-AREA.
002000 PERFORM SECOND-PARA
           VARYING TALLY FROM 1 BY 1 UNTIL TALLY > 10.
002100 DISPLAY 9-AREA.
002200 STOP RUN.
002300 SECOND-PARA.
002400 IF S9 (TALLY) NUMERIC
002500 DISPLAY TALLY "=" S9 (TALLY)
002600 MOVE S9 (TALLY) TO 9X (TALLY).
002700
```

Figure F-1. Source Program Listing

PAGE 0002		TESTER	SYMBOL TABLE MAP				CATEGORY R O D J BZ			
LVL	SOURCE, NAME		BASE DISPL	SIZE	USAGE					
WORKING-STORAGE SECTION										
01	A		DH 000042	000055		GROUP				
02	FILLER		DH 000042	000017	DISP	AN				
02	FILLER		DH 000061	000036	DISP	AN				
01	S9-AREA		DH 000120	000012		GROUP				
05	S9		DB 000120	000012	DISP	DISP-NS			0	
01	9-AREA		DH 000132	000012		GROUP				
05	9X		DB 000132	000012	DISP	DISP-N			0	

Figure F-2. Symbol Table Map

PAGE 0003 TESTER SYMBOL TABLE MAP

SOURCE NAME	S/P	INTERNAL NAME	PB-RELATIVE LOC	PRIORITY NO.
MAINLINE	S	MAINLINE00	000000	0
FIRST-PARA	P		000000	0
SECOND-PARA	P		000124	0
ILL*KRANCH	S		000251	0

DATA AREA IS 8000344 WORDS.
 CPU TIME = 0:00:06. WALL TIME = 0:00:27.
 END COBOL/3000 COMPILATION. NO ERRORS. NO WARNINGS.

Figure F-2. Symbol Table Map (Continued)

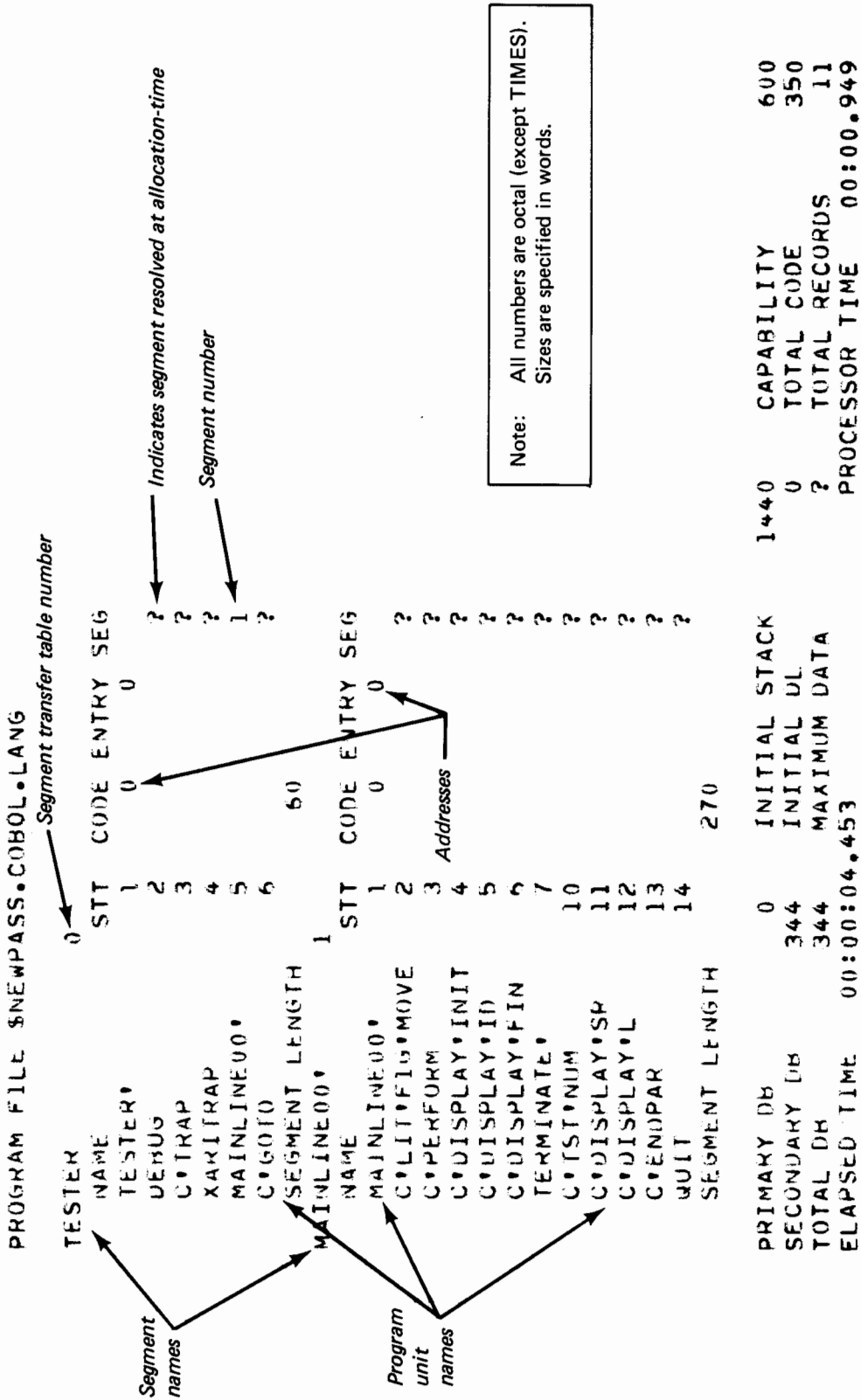


Figure F-3. P-MAP

```
HERE IT IS:      **START OF PROGRAM**
AIJRSZ19()
00001=A
00002=I
00003=J
00004=H
00007=1
00008=9
1919 19

END OF PROGRAM
```

Figure F-4. Object Program Output

APPENDIX G

Software Arithmetic Trap for Interprogram Communication

For a COBOL main program, the COBOL/3000 compiler emits code to arm the MPE/3000 software arithmetic trap mechanism so that activation of any of the following firmware traps causes a branch to the post-trap procedure C'TRAP:

- Decimal divide by zero
 - Invalid decimal digit
 - Invalid ASCII digit
 - Decimal overflow
 - Integer overflow
 - Integer divide by zero
- } trap.



The code emitted is the following call to the XARITRAP intrinsic:

```
XARITRAP (% 23422, @ C'TRAP, OLDMASK, OLDPLABEL);
```

(The XARITRAP intrinsic and its parameters are described in *MPE Intrinsic Reference Manual*.)

The C'TRAP procedure resides in the COBOL run-time library. This procedure determines from the conditions of the trap the appropriate response to COBOL.

For operating efficiency and compatibility with other HP programming languages, the COBOL 3000 compiler does not emit code to arm software traps for COBOL subprograms. It is the user's responsibility to arm the software arithmetic trap (as required by COBOL) before a COBOL subprogram is called by a program or another subprogram written in another language.

For the user's convenience, the following procedure (COBOLTRAP) resides in the COBOL run-time library. It can be used to arm the software arithmetic trap:

```
PROCEDURE COBOLTRAP;  
BEGIN  
  PROCEDURE C'TRAP; OPTION EXTERNAL;  
  INTRINSIC XARITRAP;  
  INTEGER OLDMASK;  
  INTEGER OLDPLABEL;  
  XARITRAP (% 23422, @ C'TRAP, OLDMASK, OLDPLABEL);  
END;
```

If the user desires, he may provide his own procedure for arming the software arithmetic trap. However, HP recommends use of COBOLTRAP to take advantage of any modifications made to the trap-handling software in COBOL as time goes by.

Recommended programming steps for trap-handling are described below:

COBOL PROGRAMS/SUBPROGRAMS CALLING ANOTHER COBOL SUBPROGRAM

Since the software arithmetic trap is armed in the COBOL main program, there is no need to arm this trap for the subprograms.

COBOL PROGRAMS/SUBPROGRAMS CALLING SUBPROGRAMS IN OTHER LANGUAGES

- Step 1: Arm or disarm the software trap mechanism as required by the other language. (For directions on how to do this, refer to the HP manual covering that language.)
- Step 2: Issue a CALL statement calling the other-language subprogram or procedure.
- Step 3: Arm the software trap as required by COBOL. This can be done in the calling program by using the statement:

CALL "COBOLTRAP"

PROGRAMS/SUBPROGRAMS IN OTHER LANGUAGES CALLING A COBOL SUBPROGRAM

- Step 1: Arm the software trap as required by COBOL, by calling the COBOLTRAP procedure.
- Step 2: Call the COBOL subprogram.
- Step 3: Arm or disarm the software trap mechanism required by the other-language program. (Refer to the HP manual covering that language.)

Examples

The following example shows how a COBOL/3000 program calling an SPL/3000 procedure could arm the software arithmetic trap.

Col.
7

IDENTIFICATION DIVISION.
PROGRAM-ID. COBOLPROF.
ENVIRONMENT DIVISION.

```

DATA DIVISION.
WORKING STORAGE SECTION.
77 PARM1 PIC S99V99 COMP.
77 PARM2 PIC X(18).
PROCEDURE DIVISION.
P. . . . .

```

```

* CALL THE PROCEDURE SPLTRAP TO DISARM SOFTWARE TRAP FOR SPL
* (STEP 1)
  CALL "SPLTRAP"
* CALL THE SPL PROCEDURE SPLPROGRAM (STEP 2)
  CALL "SPLPROGRAM" USING PARM1, PARM2
* ARM SOFTWARE TRAP FOR COBOL (STEP 3)
  CALL "COBOLTRAP"
.
.
.
STOP RUN.

```

The procedure SPLTRAP, which may be embedded in the procedure SPLPROGRAM, is coded as follows:

```

PROCEDURE SPLTRAP;
BEGIN
  INTRINSIC XARITRAP;
  INTEGER OLDMASK, OLDPLABEL;
  XARITRAP (0, 0, OLDMASK, OLDPLABEL);
  <<DISARM SOFTWARE TRAP >>
END;

```

The next example shows how an SPL/3000 program calling a COBOL/3000 subprogram could arm the software arithmetic trap.

```

PROCEDURE SPLPROG;
BEGIN
  PROCEDURE COBOLSUBP (PARM1, PARM2);
  INTEGER PARM1; ARRAY PARM2; OPTION EXTERNAL;
  PROCEDURE COBOLTRAP; OPTION EXTERNAL;
  INTEGER I;
  ARRAY A (0: 8);
  .
  .
  .
  COBOLTRAP; << ARM THE SOFTWARE TRAP REQUIRED BY COBOL (STEP 1) >>
  COBOLSUBP (I, A); <<CALL THE COBOL SUBPROGRAM (STEP 2) >>
  SPLTRAP; <<DISARM THE SOFTWARE TRAP USING THE SAME PROCEDURE
  NOTED IN THE FIRST EXAMPLE (STEP 3) >>
END;

```


Index

A

- A-Area and B-Area, 2-2, 3-2
- A character-string symbol, 6-20
- ACCEPT statement, 10-2
- ACCESS MODE clause, 4-7
- ACCESS MODE IS RANDOM, 4-9
- actual file designators, A-3, ff.
- ACTUAL KEY, 4-9
- ACTUAL KEY (in READ Statement), 10-13
- ACTUAL KEY (in WRITE Statement), 10-29
- ADD statement, 8-2
- AFTER ADVANCING; writing to printer files, 10-28
- ALL literal, 1-11
- ALTER statement, 8-4
- alphabetic data, 5-5
- alphanumeric data, 5-5
- alphanumeric edited, 5-5
- alphanumeric (without editing), 5-5
- AREA A and AREA B, 2-2, 3-2
- arithmetic expressions, 7-6
- ASCENDING options (in SORT Statement), 10-19
- ASCII character set, 1-8
- ASSIGN clause, 4-6
- AT END clause (in READ statement), 10-12

B

- B character-string symbol, 6-20
- BEFORE ADVANCING, writing to printer files, 10-28
- binary arithmetic operators, 7-6
- binary data items, 6-38
- BLANK WHEN ZERO clause, 6-9
- BLOCK CONTAINS clause, 6-3
- Boolean NOT, 7-20
- braces (in formats), 1-5
- brackets (in formats), 1-4

C

- calling program, 7-21
- CALL statement, 8-5
- CALL statement, operation of, 8-7
- character set, 1-6, 1-7
- CHECKSYNTAX parameter, B-9
- class condition, 7-9, 7-17
- classes of data, 5-5
- CLOSE statement, 10-5
- CLOSE statement, random access files, 10-6
- CLOSE statement, sequential access files, 10-5
- COBOL compiler diagnostic messages, C-1, C-5, ff.
- COBOL compiler-directing statements, 7-4
- COBOL compile-time options, B-1, ff.
- COBOL conditional statements, 7-4
- COBOL imperative statements, 7-4
- COBOL reserved words, D-1, ff.
- COBOLLOCK procedure, 11-2
- COBOLUNLOCK procedure, 11-3
- coding conventions, 2-4
- coding form, 2-1
- collating sequence, 1-7
- COMMENT-ENTRY paragraphs, 3-2
- comment lines, 2-3
- comments (in compiler subsystem commands), B-3
- compiler, 1-1
- compiler-directing statement, 1-2
- compiler subsystem command summary, B-6
- compiler subsystem commands,
 - syntax and format, B-1
- compile-time options, 7-22
- COMP option, 6-37
- compound condition, 7-9
- COMPUTATIONAL option, 6-38
- COMPUTE statement, 8-9
- COMP-3 option, 6-38
- COMPUTATIONAL-3 option, 6-38
- conditional compilation (\$IF), B-11
- conditional statement, 1-2
- condition-name condition, 7-18
- condition-names, 6-42
- condition object, 7-9
- condition subject, 7-9
- conditions, 7-9
- CONFIGURATION SECTION
 - (Environment Division), 4-2
- CONSOLE device, 4-2, 10-7
- constant, 1-9
- Continuation records (in
 - (in compiler subsystem commands), B-4
- continued lines, 2-3
- converting collating sequences, 1-9
- COPY command, Segmenter, A-7a
- copy-file, 8-10
- COPYLIB.group.account library name, 8-11
- copy library, 8-10
- COPY statement, 8-10
- COPY statement (in Data Division), 8-12
- COPY statement (in Environment Division), 8-12
- COPY statement (in Procedure Division), 8-12
- CORRESPONDING option, 7-25
- currency character-string symbol, 6-22
- currency sign clause, 4-3
- CURRENT-DATE register, 1-11

D

- data description entries, 2-3, 6-6
- Data Division, 1-2, 5-1
- data item description, 6-6
- DATA-NAME or FILLER clause, 6-8
- DATA RECORDS clause, 6-3
- DATE-COMPILED paragraph, 3-2
- decimal-point clause, 4-4
- Declarative procedure, 1-3
- declaratives, 7-2
- DEPENDING ON option, 6-14
- DESCENDING options (in SORT statement), 10-19
- direct indexing, 1-14
- disasterous error diagnostic messages, C-2
- DISPLAY option, 6-37
- DISPLAY statement, 10-7
- DIVIDE statement, 8-13
- division headings, 2-2
- dynamic locking, 4-6, 11-1, B-8
- dynamic subprogram, 7-3b, B-8

E

- editing conventions, 6-23
- elementary items, 5-3
- elementary move, 8-28
- ellipsis (in formats), 1-5
- ELSE NEXT SENTENCE (in IF statement), 8-25
- ENTER statement, 8-16
- ENTRY statement, 8-17
- equating copy files to COPYFILE, A-12
- Environment Division, 1-2
- evaluating conditions, 7-14
- EXAMINE statement, 8-18
- EXIT PROGRAM statement, 8-21
- EXIT statement, 8-20

F

- figurative constants, 1-10
- File Description Entries, 6-1
- File Information Display (MPE), C-4
- FILE-LIMITS clause, 4-7
- file-name (COBOL), 4-6, 9-1
 - passing to SPL subprogram, 8-6
- FILE SECTION, 5-1
- file size, sort, 10-21a
- FILLER, 6-8
- fixed insertion editing, 6-24
- floating insertion editing, 6-28
- FROM option (in WRITE statement), 10-27

G

- GOBACK statement, 8-22
- GO TO statement, 8-23
- group items, 5-3

H

- HIGH-VALUE, 1-10

I

- identification code, 2-3
- Identification Division, 1-2, 3-1
- identification field, 2-3
- identifier, 1-12
- IF statement, 8-25
- IF statements, nesting of, 8-26
- imperative statement, 1-2
- implied decimal point, 1-6
- INDEXED BY clause, 6-16
- index-data-item, 6-16
- index-data-items, comparison of, 7-12
- indexing, 1-14
- index-names, comparison of, 7-12
- INDEX option, 6-40
- initial values, 6-41
- INPUT (in OPEN statement), 10-9
- input/output file designators
 - (input set and output set), A-5, A-6
- interprogram communication, 7-21
- INTO option (in READ statement), 10-11, 10-13
- INVALID KEY (in READ statement), 10-12
- INVALID KEY (in WRITE statement), 10-30
- I-O-CONTROL paragraph, 4-10
- I-O conventions, 9-1
- I-O (in OPEN statement), 10-9

J

- JUSTIFIED clause, 6-11

K

- KEY option, 6-15
- key word, 1-3

L

- LABEL RECORDS clause, 6-4
- level 66, 5-4
- level 77, 5-4
- level 88, 5-4
- level indicator (FD and SD), 5-3
- level numbers, 5-3
- LINKAGE SECTION, 5-2
- listing and compilation options (\$CONTROL), B-6
- literal, 1-9
- local references, 7-3
- LOCK option (CLOSE statement), 10-5
- locking, dynamic, 11-1, B-8
- lowercase words (in formats), 1-4
- LOW-VALUE, 1-10

M

main programs, 7-3a
maintenance file, B-16
 see \$EDIT Map, symbol table, B-8, F-1
MOVE statement, 8-27
MPE command summary, A-1
MPE File Information Display, C-4
multi-dimensional tables, 8-51
MULTIPLE FILE clause, 4-11
MULTIPLE REEL/UNIT clause, 4-7
MULTIPLY statement, 8-32

N

natural memory boundary (16-bit word), 6-34
new files, A-4
NEWSEG Command, Segmenter, A-7a
NEXT SENTENCE (in IF statement), 8-25
non-elementary move, 8-29
nonnumeric literal, 1-10
nonnumeric operands, comparison of, 7-11
NO REWIND option (CLOSE statement), 10-6
NO SPACE CONTROL, 4-2
NOT condition, 7-20
NOTE statement, 8-34
numeric data, 5-5
numeric edited, 5-5
numeric literal, 1-9
numeric operands, comparison of, 7-11

O

object program, 1-1
object program diagnostic messages, C-4, C-44, ff.
OCCURS clause, 6-12
OCCURS clause, rules for, 6-13
old files, A-5
OPEN statement, 10-9
operands, comparison of, 7-11
OPTIONAL clause, 4-6
outerblock, 7-3a
OUTPUT (in OPEN statement), 10-9
overpunches, zoned, 6-37

P

P character-string symbol, 6-20
page title and ejection (\$PAGE), B-14
page title in standard listing (\$TITLE), B-14
paragraph, 1-2, 7-3
paragraph-names, 2-2, 7-3
parameters (in compiler subsystem commands), B-2
PERFORM statement, 8-35
PERFORM statement, nesting of, 8-37

PERFORM statement, segmentation
 considerations, 8-37
PERFORM VARYING option
 (in PERFORM statement), 8-39
PERFORM VARYING, one identifier, 8-40
PERFORM VARYING, three identifiers, 8-42
PERFORM VARYING, two identifiers, 8-40
permissible moves, 8-30
permissible relational comparisons, 7-13
PICTURE clause, 4-3, 6-17
PICTURE clause character-string, 6-19
PICTURE clause data items, 6-17, ff.
precedence of symbols used in the
 PICTURE clause, 6-30
priority number, 7-3
procedural statements, common options, 7-22, ff.
Procedure Division, 1-2
Procedure Division conventions, 7-2
Procedure Division header, 7-2
Procedure Division segmentation, 7-3
procedure-name, 1-3
procedures, 7-3
PROCESSING MODE clause, 4-7
PROGRAM-ID paragraph, 3-1
punctuation of statements, 1-6

Q

qualification, 1-12
questionable construct diagnostic messages, C-2
QUOTE, 1-10

R

READ statement, 10-11
READ statement, random access files, 10-12
READ statement, sequential access files, 10-11
RECORD CONTAINS clause, 6-5
RECORDING MODE clause, 6-5
REDEFINES clause, 6-31
REEL/UNIT option (CLOSE statement), 10-5
referencing files, A-3
relation condition, 7-9, 7-15
relation conditions, abbreviating, 7-16
relational operator, 7-9
relative indexing, 1-15
RELEASE statement, 10-15
relocatable binary module, (RBM), A-7
relocatable library (RL), 7-3a, A-7a
RENAMES clause, 6-32
REPLACING (in EXAMINE statement), 8-18
RERUN clause, 4-10
RESERVE clause, 4-7
reserved words, 1-3, D-1, ff.
RETURN statement, 10-16
RESERVED (in OPEN statement), 10-9
ROUNDED option, 7-23

S

- S character-string symbol, 6-20
- SAME AREA clause, 4-10
- SAME RECORD AREA, 4-10
- scaling position, 6-20
- SEARCH statement, 8-47
- SEARCH ALL statement, 8-47
- section, 1-2, 7-3
- SEEK statement, 10-17
- segmented library, 7-3a, A-7a
- segmenter, 7-3a, A-7a
- segments, 7-3a
- SELECT (Random files), 4-9
- SELECT (sequential files), 4-5
- SELECT (Sort files), 4-7
- sentence, 1-2
- sequence numbers, 2-2
- sequence of character-string
 - symbols (in PICTURE clause), 6-29
- serious error diagnostic messages, C-2
- SET statement, 8-52
- sign condition, 7-19
- signed packed decimal format, 6-39
- simple conditions, 7-9
- simple insertion editing, 6-23
- SIZE ERROR option, 7-24
- SL (segmented library), 7-3a, A-7a
- slack byte, 6-34, 6-35, 6-40
- software switches for conditional compilation (\$SET), B-12
- SORT statement, 10-18
- SORT statement input procedures, 10-20
- SORT statement output procedures, 10-20
- source text merging and editing (\$EDIT), B-15
- SPACE, 1-10
- special characters (in formats), 1-5
- special insertion editing, 6-24
- SPECIAL-NAMES paragraph, 4-2
- specifying files as command parameters, A-3
- statement, 1-2
- STOP RUN statement, 8-55
- STOP statement, 8-55
- subprogram, 7-3b, 7-21, B-8
- subprogram, using ENTRY statements in, 8-5
- subscripting, 1-15
- substituting " with ', 1-9
- SUBTRACT statement, 8-57
- symbol table map, B-8, F-1
- SYNCHRONIZED clause, 6-34
- SYSIN device, 4-2, 10-2
- SYSOUT device, 4-2, 10-7
- system-defined files, A-3
- system-file-name (MPE), 4-6, 9-1
- system-function, 4-2
- system labels (in USE statement), 10-25

T

- TALLYING (in EXAMINE statement), 8-18
- TALLY register, 1-11
- THRU option, 6-33
- TIME-OF-DAY register, 1-11
- TIMES option (in PERFORM statement), 8-39
- TOP, 4-2

U

- unary operators, 7-6
- UNTIL option (in PERFORM statement), 8-39
- uppercase words (in formats), 1-3
- USAGE clause, 6-36
- user labels (in USE statement), 10-25
- user pre-defined files, A-4
- user subprogram library (USL), 7-3a, A-7
- USE statement, 10-24
- USING/GIVING option (in SORT statement), 10-21
- USING option, CALL statement, 8-6

V

- V character-string symbol, 6-21
- VALUE clause, 6-41
- VALUE OF clause, 6-5
- VARYING option (SEARCH statement), 8-50

W

- warning only diagnostic messages, C-2
- WITH NO REWIND (in OPEN statement), 10-9
- word, 6-34
- WORKING-STORAGE SECTION, 5-2
- WRITE statement, 10-27
- WRITE statement, for printer files, 10-27
- WRITE statement, random access files, 10-30
- WRITE statement, sequential access files, 10-27

X

- X character-string symbol, 6-21

Z

- Z character-string symbol, 6-21
- ZERO, 1-10
- zero suppression editing, 6-26
- zoned overpunches, 6-37

. . . . (warning), C-2
. character-string symbol, 6-22
???? (questionable construct), C-2
!!!! (serious error), C-2
**** (disasterous error), C-2
* character-string symbol, 6-22
, character-string symbol, 6-21
+, -, CR, DB character-string symbols, 6-22
^ character (caret), 1-6
Δ character, 1-6

0 character-string symbol, 6-21
9 character-string symbol, 6-21
66, 5-4
77, 5-4
88, 5-4

\$CONTROL subsystem command, B-6
\$EDIT subsystem command, B-15
\$IF subsystem command, B-11
\$OLDPASS, A-5
\$STDIN, A-3
\$STDINX, A-3
\$STDLIST, A-4
\$NEWPASS, A-4
\$NULL file, 10-10, A-4
\$PAGE subsystem command, B-14
\$SET subsystem command, B-12
\$TITLE subsystem command, B-14
:COBOL command (MPE), A-1
:COBOL command parameters, A-8
:COBOLGO command (MPE), A-1
:COBOLGO command parameters, A-12
:COBOLPREP command (MPE), A-1
:COBOLPREP command parameters, A-10
:FILE command (MPE), 10-10
:PREP command parameters, A-13
:PREPRUN command parameters, A-15
:RUN command parameters, A-18

Printed in U.S.A. 7/75
Update No. 1 Incorporated 3/77
PART NO. 32213-90001

HEWLETT  PACKARD
Sales and service from 172 offices in 65 countries.
5303 Stevens Creek Blvd., Santa Clara, California 95050