HP 3000 Computer Systems

COBOL II/3000

Reference Manual

19447 PRUNERIDGE AVE., CUPERTINO, CALIFORNIA 95014

Part No. 32233-90001
Product No. 32233A

Printed in U.S.A. 12/79

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.

Copyright © 1979 by 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. Within the manual, any page changed since the last edition is indicated by printing the date the changes were made on the bottom of the page. Changes are marked with a vertical bar in the margin. If an update is incorporated when an edition is reprinted, these bars are removed but the dates remain. No information is incorporated into a reprinting unless it appears in a prior update.

First Edition. . . . . Dec 1979
Update Package No. 1 . . . . July 1980

Changed Pages	Effective Date
iii, iv	July 1980
xviii	July 1980
2-5	July 1980
3-8	July 1980
3-9	July 1980
4-6	July 1980
5-4	July 1980
7-4	July 1980
7-9	July 1980
7-11	July 1980
7-23	July 1980
7-29	July 1980
7-34	July 1980
7-36	July 1980
7-46	July 1980
7-50	July 1980
7-54	July 1980
9-10	July 1980
9-20	July 1980
9-22	July 1980
9-35	July 1980
9-38	July 1980
9-41	July 1980
9-43	July 1980
9-52	July 1980
9-57	July 1980
9-58	July 1980
9-62	July 1980
9-66	July 1980
10-2	July 1980
10-6	July 1980
10-11	July 1980
10-18	July 1980
11-1	July 1980
11-2	July 1980
11-3	July 1980
11-5	July 1980
11-16	July 1980
11-43	July 1980
11-53	July 1980
11-59	July 1980
11-61	July 1980
11-65	July 1980
11-83	July 1980
11-85	July 1980
11-94	July 1980
11-99	July 1980
11-103	July 1980
11-106	July 1980
11-108	July 1980
11-109	July 1980
11-115	July 1980
11-116	July 1980
11-119	July 1980
11-120	July 1980
11-127	July 1980
12-5	July 1980
12-6	July 1980
12-8	July 1980
12-9	July 1980
12-10	July 1980
12-11	July 1980
12-12	July 1980
13-14	July 1980
14-19	July 1980
14-28	July 1980
A-7	July 1980
A-9	July 1980
A-24	July 1980
A-28	July 1980
A-29	July 1980
C-12	July 1980
C-13	July 1980
C-13a	July 1980
D-1 to D-27	July 1980
G-1	July 1980
H-9	July 1980
H-17	July 1980
J-7	July 1980
J-13	July 1980
J-23	July 1980
K-1	July 1980
2	July 1980
9	July 1980

PRINTING HISTORY

New editions are complete revisions of the manual. 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 of the manual changes only when a new edition is published. When an edition is reprinted, all the prior updates to the edition are incorporated. No information is incorporated into a reprinting unless it appears as a prior update. The edition does not change.

The software product part number printed alongside the date indicates the version and update level of the software product at the time the manual edition or update was issued. Many product updates and fixes do not require manual changes, and conversely, manual corrections may be done without accompanying product changes. Therefore, do not expect a one to one correspondence between product updates and manual updates.

First Edition......Dec 1979......32233A.00.00

Update Package #1..July 1980.....32233A.00.03

PREFACE

This manual is one of a set of four manuals that document the HP 3000 COBOL programming language, named COBOL II/3000. COBOL II/3000 is an implementation of ANSI COBOL for the HP 3000 computer systems. The standard for this language is American National Standard COBOL X3.23-1974 as approved by the American National Standards Institute (ANSI).

This manual explains how to use COBOL II/3000. Specifically, it shows how to write source programs in the COBOL II/3000 programming language, compile them into object programs with the COBOL II/3000 compiler, and prepare and execute them.

This manual is a reference book rather than a tutorial text for new programmers. To use it effectively, you should understand the fundamental techniques of programming and have some experience with the COBOL language. If you are an experienced programmer without knowledge of COBOL, you may use this manual to learn the language; however, you will find this easier if you first read a good introductory or tutorial text, such as:

> McCracken, Daniel D. and Garbassi, Umberto, A GUIDE FOR COBOL PROGRAMMING, 2nd Edition, New York, Wiley Interscience, 1970.

In addition, some familiarity with the HP 3000 Computer System is recommended.

The entire set of manuals produced by Hewlett-Packard for the documentation of COBOL II/3000 is shown below.

- COBOL II/3000 REFERENCE MANUAL
  (part number 32233-90001)
- COBOL II/3000 POCKET GUIDE
  (32233-90002—available mid 1980)
- USING COBOL II
  (32233-90003—available mid 1980)
- COBOL/3000 TO COBOL II/3000 CONVERSION GUIDE
  (32233-90005)

The first three of the above manuals are intended for all COBOL II/3000 installations. The conversion guide, however, is intended only for those installations where systems are being converted from COBOL/3000 (an HP implementation of ANSI COBOL '68) to COBOL II/3000.

Introductory information on the use of various software features of your HP 3000 Computer System appears in the following manuals:

- USING THE HP 3000
  (03000-90121)
- USING FILES
  (30000-90102)

Further information about compiling, preparing, and running programs appears in documentation covering the HP 3000 Multiprogramming Executive (MPE) Operating System. In particular, you may wish to refer to the following manuals:

- MPE COMMAND REFERENCE MANUAL
  (30000-90009)
- MPE INTRINSICS REFERENCE MANUAL
  (30000-90010)

ACKNOWLEDGMENT

At the request of the American National Standards Institute (ANSI), the following acknowledgment is reproduced in its entirety:

Any organization interested in reproducing the COBOL standard and specifications in whole or in part, using ideas from this document 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 the following acknowledgment paragraphs in their entirety as part of the preface to any such publication (any organization using a short passage from this document, such as in a book review, is requested to mention "COBOL" in acknowledgment of the source, but need not quote the acknowledgment):

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

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

The authors and copyright holders of the copyrighted material used herein 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.

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. F 28-8013, copyrighted 1959 by IBM; FACT, DSI 27A5260-2760, copyrighted 1960 by Minneapolis-Honeywell

CONVENTIONS USED IN THIS MANUAL

This manual presents descriptions of the syntax and format for all COBOL program entries. The format descriptions are intended to guide you in writing each entry according to the rules of the COBOL programming language. In these format descriptions, the symbolic notation discussed below is used to clarify meaning. (In most cases, the discussion of each symbol is keyed to an item in the format description example appearing at the end of this list of symbols.)

NOTATION SYMBOL	DISCUSSION
[ ]	An element inside brackets is optional. Brackets are often used to indicate optional multiple operands. In the example at the end of this section, for instance, *identifier-n* (Item 1) is a possible additional operand following *identifier-m*. Several elements stacked within a pair of brackets means that you may select any one or none of the elements. Thus, in the example, you may select either *identifier-2* or *literal-2*, or omit both (Item 2).

NOTE

The brackets are NOT part of the COBOL language and thus are NOT actually entered in your program.

NOTATION SYMBOL	DISCUSSION
{ }	When several elements are stacked within braces, you must select one of these elements. Therefore, in the example on page x, you must include either *identifier-1* or *literal-1* (Item 3).

NOTE

The braces are NOT part of the COBOL language and thus are NOT actually entered in your program.

NOTATION SYMBOL	DISCUSSION
UPPERCASE WORDS	All uppercase words are reserved words that have particular meaning to the COBOL compiler. They must be spelled exactly as shown. Uppercase words that are <u>underlined</u> are keywords that are always required when the clause or statement in which they appear is used in your program. But uppercase words that are NOT underlined are optional, and may be included or omitted at will; they have no effect on program execution and serve only to make source program listings more readable and easy to understand. In the example, when the optional entry ; ON SIZE ERROR imperative-statement is used in your program, SIZE and ERROR must be entered (Item 4) but ON may be either included or omitted as you wish (Item 5).

italicized words All lowercase words are either variables (user-defined words) or literals whose values you must supply. User-defined words include names, such as program-names, data-names, paragraph-names, and section-names. Literals include directly-specified numbers and character strings. In the example below, you might supply the value DATA-1 for the user-defined word *identifier-1* (Item 6) or the number 20 for the literal *literal-1* (Item 7).

underlining in dialog When it is necessary to distinguish user input from computer output, the input is underlined.

Example: `NEW NAME?` ALPHA

... A horizontal ellipsis indicates that a previous bracketed element may be repeated, or that elements have been omitted from the description. In the example, the ellipsis shows that the preceding identifier/literal construct may be repeated indefinitely (Item 8).

NOTE

The ellipsis is NOT part of the COBOL language and thus is NOT actually entered in your program.

<, >, and = are symbols used in conditional statements to represent the keywords LESS THAN, GREATER THAN, and EQUAL TO, respectively. Although these symbols represent keywords, they are NOT underlined.

; The semicolon is used only to improve readability and is always optional.

, The comma also is used only to improve readability, and is always optional.

. The period is a terminator or delimiter that is always required where shown; it must always be entered at the end of every division name, section name, paragraph name, and sentence.

Δ The delta is sometimes used for clarity in examples to denote a space character; it is NOT actually part of the COBOL language.

‸ The caret is sometimes used in examples to represent an implied decimal point in computer memory.

Superscript c	The superscript c indicates a control character.
	Example: Y^c ("CONTROL Y")
return	return in italics indicates a carriage return.
linefeed	linefeed in italics indicates a linefeed.

Example

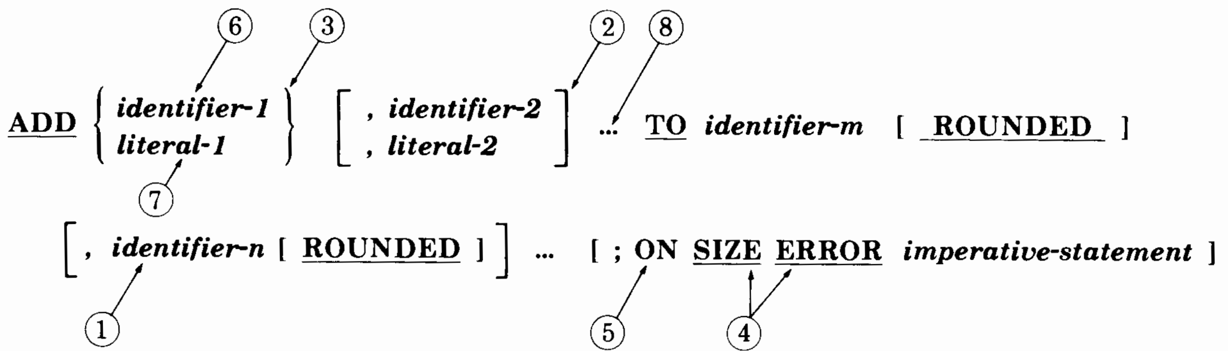


TABLE OF CONTENTS

SECTION I INTRODUCING COBOL II/3000

APPLICATIONS	1-2
FEATURES	1-2
STANDARD CAPABILITIES	1-3
EXTENDED CAPABILITIES	1-5
CONVERSION CONSIDERATIONS	1-5
USING COBOL II/3000	1-5
SYSTEM REQUIREMENTS	1-7

SECTION II PROGRAM STRUCTURE

STRUCTURAL HIERARCHY	2-2
DIVISIONS	2-2
SECTIONS	2-6
SECTION FORMAT	2-6
PARAGRAPHS	2-7
SENTENCES, STATEMENTS, AND CLAUSES	2-9

SECTION III PROGRAM ELEMENTS

CHARACTER STRINGS	3-2
WORDS	3-2
Reserved Words	3-2
Key Words	3-3
Optional Words	3-3
Connectives	3-3
Special Register Words	3-4
Figurative Constant Words	3-6
Special-Character Words	3-8
User-Defined Words	3-9
SYSTEM NAMES	3-14
LITERALS	3-15
Numeric Literals	3-15
Non-Numeric Literals	3-16
PICTURE CHARACTER STRINGS	3-18
COMMENT ENTRIES	3-18
Comment Lines	3-19
Separators	3-19
Character Set	3-20

SECTION IV DESCRIBING AND REFERENCING DATA

FILES	4-1
RECORDS	4-1
Logical Versus Physical Records	4-2
RECORD DESCRIPTIONS	4-2
DATA ITEMS	4-5
Data Classes and Categories	4-5
Algebraic Signs	4-6
Operational Signs	4-6
Editing Signs	4-6
DATA ALIGNMENT	4-7
UNIQUENESS OF REFERENCE	4-8
QUALIFIERS	4-8
SUBSCRIPTS	4-11
SUBSCRIPTED DATA-NAME FORMAT	4-12
INDEXES	4-12
CONDITION-NAMES	4-15

SECTION V CODING SOURCE PROGRAMS

USING EDIT/3000	5-1
USING A CODING FORM	5-4
CODING RULES	5-4
Sequence Numbers	5-4
Program Text	5-5
Continuation Lines	5-5
Comment Lines	5-6
Identification Code	5-6
CODING CONVENTIONS	5-6

SECTION VI IDENTIFICATION DIVISION

DIVISION FORMAT	6-1
Division Syntax Rules	6-1
PARAGRAPHS	6-3
PROGRAM-ID Paragraph	6-3
DATA-COMPILED Paragraph	6-2
Other Paragraphs	6-2

SECTION VII ENVIRONMENT DIVISION

DIVISION FORMAT	7-1
Division Syntax Rules	7-1
CONFIGURATION SECTION	7-2
SOURCE-COMPUTER Paragraph	7-4
OBJECT-COMPUTER Paragraph	7-5
MEMORY-SIZE Clause	7-5
PROGRAM COLLATING SEQUENCE Clause	7-6
SEGMENT-LIMIT Clause	7-6
SPECIAL-NAMES Paragraph	7-7
FUNCTION-NAME Clause	7-9
Software Switches	7-11
ALPHABETIC-NAME Clause	7-14
STANDARD-1 and NATIVE Phrases	7-15
EBCDIC Phrase	7-15
LITERAL Phrase	7-16
CURRENCY SIGN IS Clause	7-20
DECIMAL-POINT IS COMMA Clause	7-21
INPUT-OUTPUT SECTION	7-22
FILE-CONTROL PARAGRAPH	7-23
Sequential Files	7-23
Random-Access Files	7-23
Relative Files	7-24
Indexed Files	7-25
SORT-MERGE Files	7-27
File Status	7-27
FILE CONTROL CLAUSES	7-28
SELECT Clause	7-31
OPTIONAL Phrase	7-31
ASSIGN Clause	7-32
ACCESS MODE Clause	7-34
ACTUAL KEY Clause	7-36
ALTERNATE RECORD KEY Clause	7-37
DUPLICATE Phrase	7-37
FILE LIMIT Clause	7-38
FILE STATUS Clause	7-39
ORGANIZATION Clause	7-45
RECORD KEY Clause	7-46
DUPLICATES Phrase	7-46
RESERVE Clause	7-47
PROCESSING MODE Clause	7-48
FILE-CONTROL Paragraph Example	7-49
I-O CONTROL PARAGRAPH	7-51
SAME Clause	7-52
SAME AREA Clause	7-52
SAME RECORD AREA Clause	7-53
SAME SORT and SAME SORT-MERGE Clauses	7-53
MULTIPLE FILE Clause	7-54

SECTION VIII DATA DIVISION

DIVISION FORMAT 8-2
Division Syntax Rules 8-3
FILE SECTION 8-4
File Section Format 8-5
WORKING STORAGE SECTION 8-6
Working Storage Section Format 8-6
LINKAGE SECTION 8-7
Linkage Section Format 8-7

SECTION IX DATA DIVISION CLAUSES

FILE DESCRIPTION CLAUSES 9-2
File Description Clause Formats 9-3
Level Indicators 9-4
BLOCK CONTAINS Clause 9-5
CODE-SET Clause 9-8
DATA RECORDS Clause 9-9
LABEL RECORD Clause 9-10
LINAGE Clause 9-11
RECORD CONTAINS Clause 9-16
RECORDING MODE Clause 9-17
VALUE OF Clause 9-19
DATA DESCRIPTION ENTRIES 9-22
Data Description Entry Formats 9-24
DATA-NAME or FILLER Clause 9-26
BLANK WHEN ZERO Clause 9-27
JUSTIFIED Clause 9-28
OCCURS Clause 9-29
PICTURE Clause 9-34
REDEFINES Clause 9-53
SIGN Clause 9-56
SYNCHRONIZED Clause 9-58
USAGE Clause 9-60
VALUE Clause 9-67
RENAMES Clause 9-69
CONDITION NAMES 9-71

SECTION X PROCEDURE DIVISION

PROCEDURE DIVISION HEADER 10-1
USING Clause 10-2
GENERAL FORMAT OF THE PROCEDURE DIVISION BODY 10-3
Declarative Sections 10-3
Procedures 10-5
Sections and Section Headers 10-5
Segmentation 10-5
PROCEDURE DIVISION STATEMENTS AND SENTENCES 10-7
Conditional Statements and Sentences 10-7
COMPILER DIRECTING Statements and Sentences 10-8
IMPERATIVE Statements and Sentences 10-8
Categories of Statements 10-10
ARITHMETIC STATEMENTS 10-12
Arithmetic Statements 10-13
Hierarchy of Operations 10-13
Valid Combinations in Arithmetic Expressions 10-15
CONDITIONAL EXPRESSIONS 10-16
Simple Conditions 10-16
Sign Condition 10-17
Class Condition 10-18
Switch-Status Condition 10-21
Relation Conditions 10-21
Condition-Name Conditions 10-26
Intrinsic Relation Conditions 10-25
Complex Conditions 10-27
Condition Evaluation Rules 10-30
Abbreviated Combined Relation Conditions 10-32
COMMON PHRASES 10-34
COMMON FEATURES OF ARITHMETIC STATEMENTS 10-37
TABLES 10-39

SECTION XI PROCEDURE DIVISION STATEMENTS

ACCEPT STATEMENT	11-1
ADD STATEMENT	11-8
ALTER STATEMENT	11-10
CALL STATEMENT	11-11
CANCEL STATEMENT	11-12
CLOSE STATEMENT	11-13
COMPUTE STATEMENT	11-18
DELETE STATEMENT	11-19
DISPLAY STATEMENT	11-22
DIVIDE STATEMENT	11-24
ENTER STATEMENT	11-29
ENTRY STATEMENT	11-30
EXAMINE STATEMENT	11-31
EXCLUSIVE STATEMENT	11-33
EXIT STATEMENT	11-35
EXIT PROGRAM STATEMENT	11-37
GO TO STATEMENT	11-38
GOBACK STATEMENT	11-39
IF STATEMENT	11-40
INSPECT STATEMENT	11-42
MOVE STATEMENT	11-50
MULTIPLY STATEMENT	11-56
OPEN STATEMENT	11-58
PERFORM STATEMENT	11-63
READ STATEMENT	11-79
REWRITE STATEMENT	11-85
RELEASE STATEMENT	11-88
RETURN STATEMENT	11-89
SEARCH STATEMENT	11-90
SEEK STATEMENT	11-97
SET STATEMENT	11-98
START STATEMENT	11-100
STOP STATEMENT	11-103
STRING STATEMENT	11-104
SUBTRACT STATEMENT	11-110
UN-EXCLUSIVE STATEMENT	11-113
UNSTRING STATEMENT	11-114
USE STATEMENT	11-121
WRITE STATEMENT	11-125

SECTION XII INTERPROGRAM COMMUNICATION

Transfer of Control	12-2
Reference to Common Data	12-3
Types of Subprograms	12-4
CALL STATEMENT	12-6
Calling Intrinsics	12-7
Calling Programs	12-9
USING Phrase	12-9
GIVING Phrase	12-10
Overflow Conditions	12-11
CANCEL STATEMENT	12-12
ENTRY STATEMENT	12-13
EXIT PROGRAM STATEMENT	12-16
GOBACK STATEMENT	12-17

SECTION XIII SORT-MERGE OPERATIONS

MERGE STATEMENT	13-2
COLLATING SEQUENCE Phrase	13-4
GIVING and OUTPUT PROCEDURE Phrase	13-5
Segmentation Considerations	13-5
RELEASE STATEMENT	13-6
RETURN STATEMENT	13-7
INTO Phrase	13-8
AT END Phrase	13-8
SORT STATEMENT	13-9
ASCENDING and DESCENDING Phrases	13-13
COLLATING SEQUENCE Phrase	13-13
USING and INPUT PROCEDURE Phrases	13-14
GIVING and OUTPUT PROCEDURE Phrases	13-15

SECTION XIV COBOL LIBRARIES AND THE COBEDIT PROGRAM

Copy Libraries	14-1
COBEDIT PROGRAM	14-2
BUILD COMMAND	14-4
COPY COMMAND	14-7
EDIT COMMAND	14-8
EXIT COMMAND	14-12
HELP COMMAND	14-14
KEEP COMMAND	14-15
LIBRARY COMMAND	14-17
LIST COMMAND	14-20
PURGE COMMAND	14-21
SHOW COMMAND	14-23
COPY STATEMENT	14-24
REPLACING PHRASE	14-25

APPENDIX A PREPROCESSOR AND MODIFICATION COMMANDS
Types of Processes A-1
PREPROCESSOR PROGRAMMING LANGUAGE A-2
$COMMENT Preprocessor Command A-2
MACRO Definition and Use A-5
$DEFINE Command A-5
Formal Parameters A-8
MACRO Calls A-9
CONDITIONAL COMPILATION A-13
$SET Command A-13
$IF Command A-13
FILE INSERTION, AND MERGING AND EDITING OPERATIONS A-15
$INCLUDE Command A-15
Merging Files and the $EDIT Command A-17
Merging Files A-17
$EDIT Command A-20
COMPILER DEPENDENT OPTIONS A-26
$PAGE Command A-23
$TITLE Command A-24
$CONTROL Command A-24

APPENDIX B MPE COMMANDS AND FILES
MPE COMMANDS B-1
Compiling, Preparing, Executing Programs B-3
Manipulating USL Files with the Segmenter B-3
Using the RUN Command B-4

APPENDIX C COBOL II EXAMPLE PROGRAM

APPENDIX D ERROR MESSAGES

APPENDIX E EXTENSIONS TO ANS '74 COBOL

APPENDIX F COBOL CODING FORM

APPENDIX G ASCII AND EBCDIC CHARACTER SETS

APPENDIX H COBOL GLOSSARY

APPENDIX I COBOLLOCK AND COBOLUNLOCK
COBOLLOCK I-3
COBOLUNLOCK I-2

APPENDIX J COMPOSITE LANGUAGE SKELETON

APPENDIX K COBOL RESERVED WORD LIST

Computer
Museum

INTRODUCING COBOL II/3000

SECTION I

COBOL, which stands for COmmon Business Oriented Language, is a high-level procedure-oriented programming language that is widely-used throughout industry and government in commerical data-processing applications.

COBOL allows business data-processing procedures to be precisely described to a computer in a standard form that closely resembles the English language. COBOL statements, like everyday English language statements, include commonly-recognized nouns, verbs, connectives, and conventional arithmetic symbols. Structurally, they are organized into paragraphs, sentences, and clauses very similar to those written in English. Because of this similarity, you can use COBOL to write instructions for a computer in much the same way that you would prepare a set of directions for another person to follow. As an example, a typical COBOL sentence might appear as follows:

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

The similarity of COBOL to English not only simplifies the writing of programs for you, but also makes it easy for other programmers to understand the source code that you write. Thus, others may easily work with you on the same program or maintain your programs after you are no longer responsible for their upkeep. This ability to facilitate clear and easily understood source code makes COBOL essentially a self-documenting programming language.

The rules for writing COBOL statements, however, are much more rigorous than those for writing English language statements, and must be followed in detail to produce usable COBOL programs.

Once you write a COBOL program to solve a particular problem, it must be translated from source language into a set of machine instructions (an object program) that can be executed by the computer. On HP 3000 Computer Systems, this translation is performed by the COBOL II/3000 compiler. As a matter of fact, the term COBOL is typically used to refer to the compiler as well as the source language. On the HP 3000, once a source program is compiled into an object program, it may then be prepared and executed to perform the functions you desire.

The COBOL compiler and the object programs it produces all operate on any HP 3000 Computer System under control of the most recent version of the HP 3000 Multiprogramming Executive Operating System, MPE-III.

Applications

In the business community, COBOL is used in many different kinds of applications. These may range from the preparation and maintenance of simple mailing lists to the computerized management of large payroll, inventory, accounting, or management-information systems. A typical COBOL application handles:

- Large volumes of data.
- Extensive file processing.
- Many repetitive operations.
- Printing of reports.
- Flexible processing environment in which systems, processing options, programmers, and users may all change.

Features

COBOL, in general, is a widely-used, easy-to-learn, self-documenting language offered on many different computer systems produced by most major mainframe manufacturers. Because most COBOL compilers follow a standard established by the American National Standards Institute (ANSI), COBOL language programs written for one system are generally compatible with most other systems and require little or no modification for transfer between systems. Because COBOL is a business-problem oriented (rather than a machine oriented) language, it allows you to concentrate fully on solving your data-processing problems rather than worrying about computer system requirements.

COBOL II/3000 offers several important features, among which are:

- Compatibility with American National Standards Institute (ANSI) COBOL at all levels of all COBOL modules, except for the Report Writer, Communication, and Debug Modules. (For further explanation of modules, see STANDARD CAPABILITIES below.)
- Communication with programs written in other languages, such as RPG, FORTRAN, and the HP 3000 Systems Programming Language (SPL/3000).
- Communication with SORT-MERGE/3000 (the HP 3000 sorting and merging subsystem) through the SORT verb. SORT-MERGE/3000 provides many powerful sorting and merging capabilities, including multiple file inputs, a wide variety of data types and user-defined collating sequences.
- Communication with IMAGE/3000 (the HP 3000 data base management subsystem) and HP VIEW/3000 (the HP data entry and screen design and management facility). IMAGE/3000 eliminates redundant data, promotes consistency in data references, and reduces data maintenance by allowing you to consolidate several data files into a single data base. HP VIEW/3000 provides high level facilities for terminal handling and data entry that aid you in designing and coding front ends to transaction processing applications. For further information about these subsystems, please see QUERY/3000 Reference Manual (part no. 30000-90042) and IMAGE/3000 Data Base Management System Reference Manual (part no. 32215-90003), and HP VIEW/3000 Reference Manual (part no. 32209-90001).
- Processing of indexed files with the Keyed Sequential Access Method (KSAM/3000) subsystem, which is accessed automatically through COBOL's Indexed I/O Module. This subsystem allows you to create and manage files that can be accessed both sequentially and randomly by any COBOL program. It offers great file-processing flexibility and efficiency.

Standard Capabilities

The standard capabilities of COBOL were originally developed and defined by a national committee of computer manufacturers and users known as the Conference On DAta SYstems Languages (CODASYL). In 1960, under the guidance of this committee, the first official version of COBOL was designed; this was designated COBOL 60. Since then, subsequent versions were developed that significantly extended the power of the language. The later versions, beginning with COBOL 68, followed standards published and sanctioned by ANSI.

The current version of COBOL, designated COBOL II, was developed in accordance with ANSI Standard X3.23-1974. This version is organized on the basis of one nucleus and eleven functional processing modules. These elements are summarized in table 1-1.

Each module contains either two or three functional levels. In all cases, the lower levels are proper subsets of the higher levels within the same module. The lowest levels supply elements needed for basic or elementary operations; the higher levels supply more extensive or sophisticated capabilities.

The full ANSI COBOL is composed of the highest level of the nucleus and of each functional processing module. The minimum ANSI COBOL is composed of the lowest level of the nucleus and of each module.

NOTE

> COBOL II/3000 is a full level two implementation of the nucleus and all modules except the Report Writer, Communication, and Debug modules at the highest level. The Communication, Report Writer, and Debug modules are not present in COBOL II/3000.

Table 1-1.
ANS COBOL II ORGANIZATION

Nucleus/Module	Function
Nucleus	Contains language elements necessary for internal processing.
Table-Handling Module	Contains language elements needed for definition of tables; identification, manipulation, and use of indices; and references to items within tables.
Sequential I/O Module	Provides language elements for definition and access of sequentially-organized external files.
Relative I/O Module	Provides capability of defining and accessing mass storage files in which records are identified by relative record number.
Indexed I/O Module	Provides capability of defining and accessing mass storage files in which records are identified by the value of a key and accessed through an index.
Sort-Merge Module	Allows for inclusion of one or more sort/merge operations in a COBOL program.
Report Writer Module	Provides for semi-automatic production of printed reports. (Not implemented in COBOL II/3000.)
Segmentation Module	Provides for overlaying of Procedure Division sections at object time.
Library Module	Provides for inclusion into a program of predefined COBOL text.
Debug Module	Offers a means by which users can specify a debugging algorithm—the conditions under which data or procedure items are monitored during execution of the program. (Not implemented COBOL II/3000.)
Inter-Program Communication Module	Allows a program to communicate with other programs.
Communication Module	Provides ability to access, process, and create messages or portions of messages, and to communicate through a message control system with local and remote communication devices. (not implemented in COBOL II/3000.)

Extended Capabilities

In addition to the capabilities of COBOL as defined by ANSI Standard X3.23-1974, COBOL II/3000 offers several other features that extend and enhance the power of the COBOL language. The most significant of the newest extensions are:

- Free-field data entry (through the ACCEPT FREE statement).
- Ability to call MPE intrinsics and SPL procedures directly from COBOL, without writing special routines in SPL/3000 for this purpose.
- Ability to pass parameters both by value, byte-pointer, and by word-pointer to non-COBOL subprograms.
- EXCLUSIVE and NON-EXCLUSIVE use of files (for locking and unlocking files to provide data integrity during on-line updating operations conducted simultaneously by many users).
- Skipping of carriage-control tape channels on the lineprinter. Thus, programs can refer to tape channels rather than line numbers when directing output to the line-printer.
- Octal literals.

NOTE

> Most features implemented in COBOL/3000 that are not part of ANS Standard X3.23-1974 are retained in COBOL II/3000 for upward compatibility.

A complete list of the extensions of COBOL II/3000 to ANS COBOL '74 appears in Appendix E.

Conversion Considerations

Users upgrading their HP 3000 systems from COBOL to COBOL II may take advantage of the special conversion aids provided by HP. For further information, please read the COBOL/3000 TO COBOL II/3000 Conversion Guide (part no. 32233-90005).

Users transferring COBOL source programs and data to HP 3000 systems from other systems should consult their HP 3000 Systems Engineer for assistance and advice.

Using COBOL II/3000

In using COBOL II/3000 to solve any programming task, you normally follow the seven steps outlined below and illustrated in Figure 1-1.

Step 1: Analyze the Task

Carefully consider the data to be input and its format, the operations to be performed on this data, and the data to be output and its format. You must also decide what devices to use for your input and output files.

Step 2: Plan the Program

Plan the general steps your program must execute to produce the desired output. In doing this for a large program, you may want to prepare an overall flow-chart or function diagram of your program.

Step 3: Code the Source Program

Write the source program from which your object program will be compiled. For convenience, you may wish to use a standard COBOL coding sheet for this purpose. A small COBOL program written on a coding sheet appears in figure 1-2. Appendix F contains a blank coding form.

Step 4: Enter the Source Program

Enter your program into the HP 3000 Computer System. Most COBOL programmers perform this step by logging on at an interactive terminal and using the HP 3000 Editor (EDIT/3000) for source-code entry. In some cases, however, programmers have their programs keypunched onto cards and entered through a card reader.

NOTE

> COBOL programmers with substantial experience sometimes combine steps 3 and 4 by composing their programs as they use EDIT/3000 to enter them through a terminal.

The use of EDIT/3000 with COBOL is further discussed in Section V.

Along with your COBOL program, you can also submit special instructions to the compiler to request such options as source-code merging, listing format specification, or warning-message suppression. These instructions, called preprocessor subsystem commands, are discussed in Appendix A.

Step 5: Compile the Source Program

Using the COBOL II/3000 compiler, compile your source program. The compiler works in conjunction with MPE to assign disc storage areas and create routines to handle input/output, and to translate the source program into HP 3000 machine code. stored in a user subprogram library (USL) file. At your option, the compiler provides source program and object program listings, a symbol table map, and a symbol cross-reference listing. Examples of these listings appear in figures 1-3 through 1-6. The listings are explained fully in Appendix C. The compiler also produces compile-time error messages concerning any errors encountered during compilation. These error messages are discussed in Appendix D.

The listings and messages generated by the compiler help you correct any errors detected. If corrections are necessary, compile the program again until all errors are resolved.

NOTE

> COBOL II/3000 allows input/output on many devices. Specifically, you can enter your source program from a disc, terminal, card reader, or magnetic tape unit; you can write listings to a printer, terminal, disc, or magnetic tape unit. (However, you must direct your object program to disc only.)
>
> MPE allows you to request compilation (step 5), preparation (step 6, below), and execution (step 7, below) as separate, discrete operations using three separate MPE commands; as a collective sequence of operations using a single command; or as several combinations of operations using various other commands. These MPE commands are listed in Appendix B and are fully described in the MPE Commands Reference Manual (part no. 30000-90009).

Step 6: Prepare the Object Program for Execution

The program in the USL file is not directly executable. Instead, you must prepare it for execution by linking the program to any subprograms required for its support. Prepare the program by entering a command directed to the MPE Segmenter Subsystem, as discussed in Appendix B. When preparation is complete, the object code and subprogram linkages reside in an executable object program file on disc.

NOTE

> Typically, steps 4 through 7 are all done within a single session at a terminal. Large programs generally are submitted as a job stream.

Step 7: Execute the Object Program

Request program execution through the MPE RUN command. In response, MPE links the code in the program file to the appropriate supporting code and the object program begins execution. The object program processes your input, performs the operations you requested, and produces the output required. At this time, run-time diagnostic messages may appear if program errors are encountered. (These messages, like compile-time messages, are discussed in Appendix D.)

Your object program can use various devices on the system for input/output, as you desire. MPE allows you to specify these devices at program-execution time, without requiring changes to the source or object program. An example of object program output appears in figure 1-7. (This output is part of an interactive dialog between a user at a terminal and the program shown in figure 1-2. For clarity in this and all later examples of interactive processing, the user's input is underlined but the program's output is not.)

System Requirements

The COBOL II/3000 compiler can be installed as a disc-resident program and operated on any HP 3000 Computer System run under control of the MPE-III operating system. For its operational support, COBOL II/3000 also requires the KSAM/3000 and SORT-MERGE/3000 software subsystems and the COBOL II/3000 firmware.

The object programs generated by COBOL II/3000 will support any input/output devices supported by MPE-III. These include terminals, card readers, card punches, and line printers.

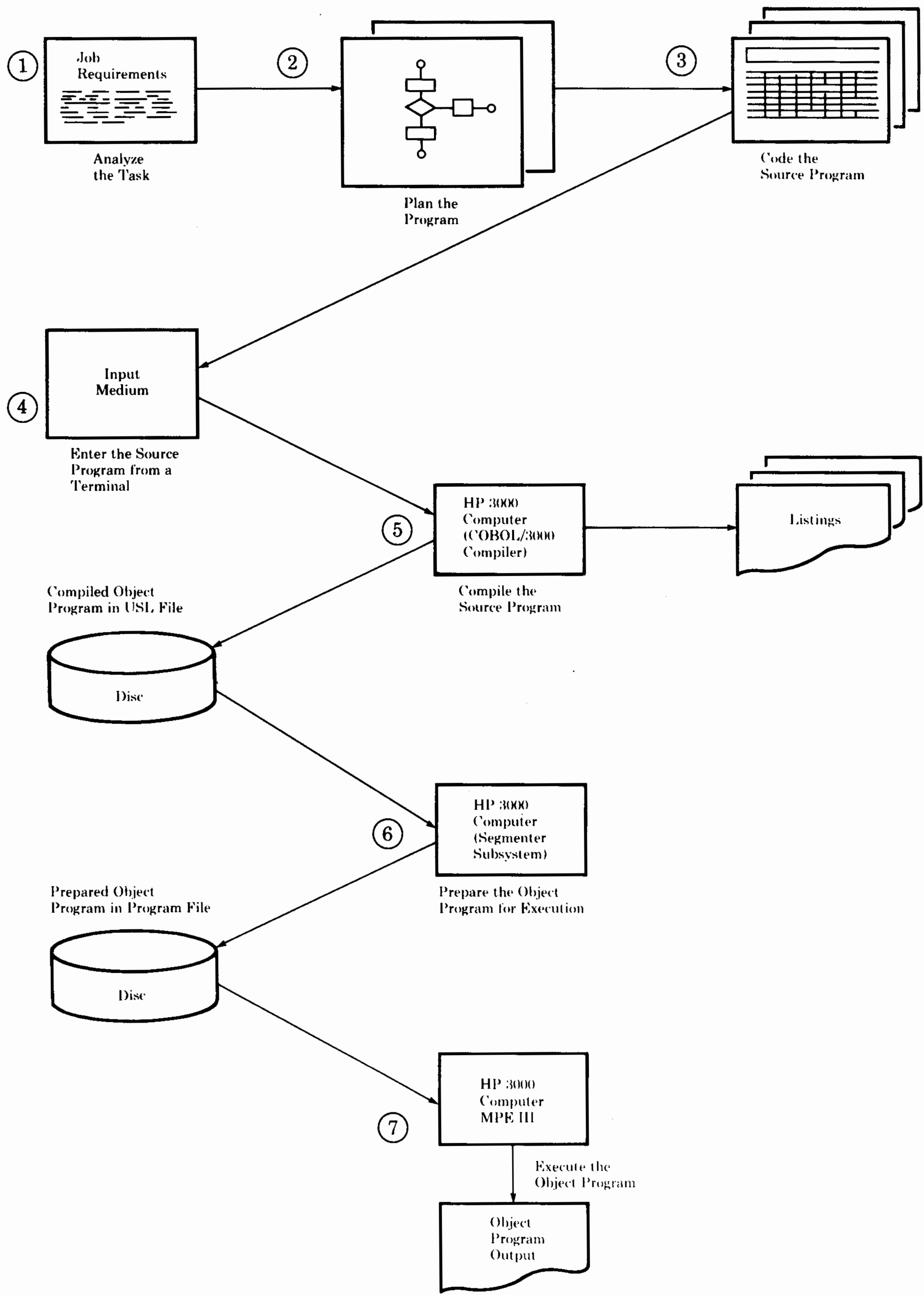


Figure 1-1. STEPS FOR PREPARING AND RUNNING A COBOL PROGRAM.

HEWLETT-PACKARD **COBOL** CODING FORM

PROGRAMMER: JONES, ARTHUR L. | DATE: 12/18/79 | PROGRAM: COBOL-FID3 | PAGE 1 OF 1

SEQUENCE (PAGE) (SER.) 1 3 4 6 | CONT. 7 | A 8 | B 12 ... COBOL STATEMENT ... 72 | IDENTIFICATION 73 80

```
ØØØØØ5$CONTROL USLINIT,MAP,CODE,CROSSREF,STDWARN,SOURCE
ØØ1ØØØ IDENTIFICATION DIVISION.
ØØ1ØØ5 PROGRAM-ID. COBOL-FID3.
ØØ1Ø1Ø AUTHOR. ARTHUR JONES.
ØØ2ØØØ ENVIRONMENT DIVISION.
ØØ3ØØØ DATA DIVISION.
ØØ3ØØ5 WORKING-STORAGE SECTION.
ØØ3Ø1Ø 77  EDIT-FIELD           PIC $Z,ZZ9.99.
ØØ3Ø15 77  TOTAL-COST           PIC 999V99.
ØØ3Ø2Ø 77  COST-OF-SALE         PIC 99V99.
ØØ3Ø25 77  TAX                  PIC 99V99.
ØØ3Ø3Ø 77  Y-N                  PIC X.

ØØ4ØØØ PROCEDURE DIVISION.
ØØ4ØØ5 ENTER ROUTINE.
ØØ4Ø1Ø     MOVE ZEROS TO TOTAL-COST.
ØØ4Ø15     DISPLAY SPACE.
ØØ4Ø2Ø     DISPLAY "ENTER COST OF SALE (BEFORE TAX) NO DECIMAL PT".
ØØ4Ø25     DISPLAY "(4 DIGITS MAX) INCLUDE LEADING ZEROS!".
ØØ4Ø3Ø     ACCEPT COST-OF-SALE.
ØØ4Ø35     COMPUTE TAX = COST-OF-SALE * .Ø6.
ØØ4Ø4Ø     ADD COST-OF-SALE, TAX TO TOTAL-COST.
ØØ4Ø45     MOVE TOTAL-COST TO EDIT-FIELD.
ØØ4Ø5Ø     DISPLAY "TOTAL COST OF PURCHASE = " EDIT-FIELD.
ØØ4Ø55     DISPLAY "ARE YOU FINISHED? (Y OR N)".
ØØ4Ø6Ø     ACCEPT Y-N.
ØØ4Ø65     IF Y-N = "N" GO TO ENTER-ROUTINE.
ØØ4Ø7Ø     STOP RUN.
```

0 = ZERO   O = ALPHA O   1 = ONE   I = ALPHA I   2 = TWO   Z = ALPHA Z

FOR 029 KEYPUNCHERS: ASCII [= 029 ¢   ASCII \ = 029 0-8-2   ASCII] = 029!   ASCII ^ = 029 ¬   ASCII ! = 029 |

5951-4465

Figure 1-2

COBOL II Source Program on Coding Sheet

```
PAGE 0001   HEWLETT-PACKARD 32233A.00.00  COBOL II/3000 WED, NOV 28, 1979,  9:57 AM
(C) HEWLETT-PACKARD CO. 1979
!
00001           001000$CONTROL USLINIT,MAP,CODE,CROSSREF,VERBS,STDWARN,SOURCE
00003           001100 IDENTIFICATION DIVISION.
00004           001200 PROGRAM-ID. COBOL-F1D3.
00005           001300 AUTHOR. ARTHUR-JONES.
00006           001400 ENVIRONMENT DIVISION.
00007           001500 DATA DIVISION.
00008           001600 WORKING-STORAGE SECTION.
00009           001700 77  EDIT-FIELD     PIC $Z,ZZ9.99.
00010           001800 77  TOTAL-COST     PIC 999V99.
00011           001900 77  COST-OF-SALE   PIC 99V99.
00012           002000 77  TAX            PIC 99V99.
00013           002100 77  Y-N            PIC X.
00014           002200
00015           002300 PROCEDURE DIVISION.
00016           002400 ENTER-ROUTINE.
00017           002500     MOVE ZEROS TO TOTAL-COST.
00018           002600     DISPLAY SPACE.
00019           002700     DISPLAY ``ENTER COST OF SALE (BEFORE TAX) NO DECIMAL PT''.
00020           002800     DISPLAY ``(4 DIGITS MAX) INCLUDE LEADING ZEROS!''.
00021           002900     ACCEPT COST-OF-SALE.
00022           003000     COMPUTE TAX = COST-OF-SALE * .06.
00023           003100     ADD COST-OF-SALE, TAX TO TOTAL-COST.
00024           003200     MOVE TOTAL-COST TO EDIT-FIELD.
00025           003300     DISPLAY ``TOTAL COST OF PURCHASE = '' EDIT-FIELD.
00026           003400     DISPLAY ``ARE YOU FINISHED? (Y OR N)''.
00027           003500     ACCEPT Y-N.
00028           003600     IF Y-N = ``N'' GO TO ENTER-ROUTINE.
00029           003700     STOP RUN.
```

Figure 1-3

COBOL II Source Program Listing

```
00000  170400 051604 031400 041402 022736 170003 010201 140004   00010  030060 030060 030000 021005 020043 041401 022402 021002
00020  000000 021040 000000 000000 041401 022402 021002 000007   00030  041405 004500 010301 170002 140031 000005 026505 047124
00040  042522 020103 047523 052040 047506 020123 040514 042440   00050  024102 042506 047522 042440 052101 054051 020116 047440
00060  042105 041511 046501 046040 050124 021030 020003 000000   00070  000045 041401 022402 021002 000045 041405 004500 010301
00100  170002 140025 000005 022450 032040 042111 043511 052123   00110  020115 040530 024440 044516 041514 052504 042440 046105
00120  040504 044516 043440 055105 051117 051441 021024 020003   00130  000041 000041 040265 041401 022601 000000 000600 171700
00140  041405 021004 041402 022744 021004 020602 020625 004500   00150  012120 000600 021006 020570 021401 167406 041402 022750
00160  021004 041405 021004 021022 071405 021005 021402 177406   00170  021002 020604 021402 020630 021022 071405 021004 001603
00200  020626 004000 020723 000600 171700 041405 021004 041402   00210  022744 021004 020602 020625 004500 012120 000600 171700
00220  041405 021004 041402 022750 021004 020602 020625 004500   00230  012120 001100 021401 167406 041402 022736 021005 041405
00240  021006 041402 022736 021005 020602 021022 071405 021005   00250  021402 177406 021002 020604 020631 021022 071405 021005
00260  001603 020626 004000 020723 041405 040133 041402 022736   00270  040131 020461 170003 010201 140005 142061 156062 020716
00300  021360 041402 022724 041405 021006 020462 000600 020470   00310  041401 022402 021002 000217 041405 004500 010301 170002
00320  140017 000005 014524 047524 040514 020103 047523 052040   00330  047506 020120 052522 041510 040523 042440 036440 021016
00340  020003 000211 004500 041402 022724 021011 000000 000216   00350  041401 022402 021002 000040 041405 004500 010301 170002
00360  140020 000005 015101 051105 020131 047525 020106 044516   00370  044523 044105 042077 020050 054440 047522 020116 024504
00400  021017 020003 000041 000034 040016 041401 022605 000252   00410  021754 156402 022116 141510 041401 043700 032001 100000
00420  001006 001005 100001 000000 000600 000000

00000  041604 025404 055000 171400 000500 055000 140006 000000   00010  027054 000444 000542 000000 000000 040406 004500 010201
00020  035013 041401 004300 045000 045001 100000 004641 031000   00030  051403 027210 055001 071403 055000 040425 051403 021044
00040  051404 041402 070431 051405 041401 022655 051406 041402   00050  070436 051407 041401 051411 041401 022546 051412 021402
00060  041401 022611 041401 022567 004500 061702 141610 041402   00070  077701 057701 043700 026665 002000 140411 000200 025001
00100  041402 020477 000043 020474
```

Figure 1-4

COBOL II Source Program Listing in Octal Code

```
PAGE 0002/COBTEXT  COBOL-F1D3       SYMBOL TABLE MAP
 LVL SOURCE NAME                     BASE DISPL   SIZE   USAGE      CATEGORY R O J BZ
-
-
 WORKING-STORAGE SECTION
-
 77  EDIT-FIELD                      Q+2: 000324  000011 DISP       NE
 77  TOTAL-COST                      Q+2: 000336  000005 DISP       N
 77  COST-OF-SALE                    Q+2: 000344  000004 DISP       N
 77  TAX                             Q+2: 000350  000004 DISP       N
 77  Y-N                             Q+2: 000354  000001 DISP       AN


PAGE 0003/COBTEXT  COBOL-F1D3       SYMBOL TABLE MAP
 LINE #    PB-LOC       PROCEDURE NAME/VERB                   INTERNAL NAME
-
 00016     000003         ENTER-ROUTINE                     ENTERROUTINE00'
 00017     000003             MOVE
 00018     000015             DISPLAY
 00019     000024             DISPLAY
 00020     000071             DISPLAY
 00021     000132             ACCEPT
 00022     000136             COMPUTE
 00023     000203             ADD
 00024     000264             MOVE
 00025     000310             DISPLAY
 00026     000350             DISPLAY
 00027     000404             ACCEPT
 00028     000410             IF
 00028     000414             GO TO
 00029     000423             STOP
```

Figure 1-5

Symbol Table Map

```
PAGE 0004/COBTEXT  COBOL-F1D3      SYMBOL TABLE MAP
COBOL CROSS REFERENCE LISTING
:

-
COST-OF-SALE
00011    00021    00022    00023
-
EDIT-FIELD
00009    00024    00025
-
ENTER-ROUTINE
00016    00028
-
TAX
00012    00022    00023
-
TOTAL-COST
00010    00017    00023    00024
-
Y-N
00013    00027    00028
```

```
PAGE 0005/COBTEXT  COBOL-F1D3      SYMBOL TABLE MAP
:
USAGE OF NONSTANDARD COBOL:
:
 LINE #  SEQ #    COL  ERROR  SVRTY  TEXT OF MESSAGE
:
 00007  001500    12    501     N     CONFIGURATION SECTION REQUIRED IN STANDARD COBOL.
:
0 ERRORS, 0 QUESTIONABLE,  0 WARNINGS, 1 USE OF NONSTANDARD (EXTENDED) COBOL
-
```

Figure 1-6
Symbol Table Cross Reference Listing And Standard Warning Messages

```
ENTER COST OF SALE (BEFORE TAX) NO DECIMAL PT
(4 DIGITS MAX) INCLUDE LEADING ZEROS!
TOTAL COST OF PURCHASE = $    9.01
ARE YOU FINISHED? (Y OR N)

ENTER COST OF SALE (BEFORE TAX) NO DECIMAL PT
(4 DIGITS MAX) INCLUDE LEADING ZEROS!
TOTAL COST OF PURCHASE = $   78.22
ARE YOU FINISHED? (Y OR N)

ENTER COST OF SALE (BEFORE TAX) NO DECIMAL PT
(4 DIGITS MAX) INCLUDE LEADING ZEROS!
TOTAL COST OF PURCHASE = $   23.32
ARE YOU FINISHED? (Y OR N)

ENTER COST OF SALE (BEFORE TAX) NO DECIMAL PT
(4 DIGITS MAX) INCLUDE LEADING ZEROS!
TOTAL COST OF PURCHASE = $   63.60
ARE YOU FINISHED? (Y OR N)

END OF PROGRAM
```

Figure 1-7

Object Program Input-Output

PROGRAM STRUCTURE

SECTION II

COBOL is similar to the English language in both structure and content. Structurally, for example, COBOL programs are made up of such familiar constructs as paragraphs, sentences, statements, and clauses. These constructs, in turn, contain such elements as words, names, verbs, and symbols. Program constructs are discussed in this section of the manual; program elements are described in section III.

Within the context of COBOL, constructs and elements all have very specific meanings. In this manual, all such terms are defined at or near the point where they are introduced. For additional convenience, their definitions also appear in the glossary in Appendix H.

Also discussed in this section is the concept of program modules. These modules make up a kind of super-set into which all other constructs fall; they contain almost all of the program constructs.

Structural Hierarchy

All COBOL programs are organized in a structure that consists of divisions, sections, paragraphs, sentences, statements, clauses, and phrases. This structure is hierarchical—that is, as a general rule, a division is made up of sections; a section is made up of paragraphs; a paragraph is made up of either sentences or clauses (depending upon the division in which it appears); a sentence may contain one or more statements; a statement or clause may contain one or more phrases. The general hierarchy appears schematically in figure 2-1. Those COBOL constructs with English-language counterparts: paragraphs, sentences, clauses, and phrases—generally resemble their corresponding counterparts. From the standpoint of the compiler, each construct is treated as a logical entity within your program.

In discussing the COBOL constructs, this manual begins with the highest level construct within a program, and proceeds to the lowest level.

Divisions

A division is the first-level (highest) construct in a COBOL program. All COBOL programs are partitioned into the following four divisions, which serve the functions noted:

- **IDENTIFICATION DIVISION**—specifies the program name and other items used to uniquely identify the program.
- **ENVIRONMENT DIVISION**—describes the computer and peripheral devices used to compile and execute the program, and the data files used by the program.
- **DATA DIVISION**—describes and defines the data items referenced by the program, including their names, lengths, decimal-point locations (if applicable), and storage formats.
- **PROCEDURE DIVISION**—specifies the operations that the program must perform, describing how the data defined in the Data Division should be processed.

These four divisions always appear in the order listed above; all are required in every COBOL program.

More information about the functions of these divisions appears later in the manual, where the divisions are discussed individually.

DIVISION FORMAT

Each division begins with a header entry, which is sometimes followed by one or more sections (in the Environment, Data, and Procedure Divisions) or paragraphs (in the Identification Division) called the division body. A division is terminated by the next division header in the program, or by the end of the program in the Procedure Division. In any COBOL program, the headers for all divisions are required although the bodies for all but the Identification Division may be optional. (The Identification Division requires a body that specifies the name of the program.)

DIVISION HEADER FORMAT

The division header consists of the division name, followed by the word DIVISION, followed by a period and a space. In the Procedure Division only, the optional USING phrase may also appear in the header between the word DIVISION and the period. In any COBOL program, only the following division headers are allowed:

IDENTIFICATION DIVISION.

ENVIRONMENT DIVISION.

DATA DIVISION.

PROCEDURE DIVISION [USING *data-name-1* [, *data-name-2*] ...].

NOTE

The conventions for all syntax and format notations in this manual, such as the upper and lowercase letters, underlines, brackets, and ellipses used in the headers shown above, are explained in CONVENTIONS USED IN THIS MANUAL (near the front of this book).

Always remember that the terminating periods shown in construct format descriptions must be included—otherwise, the compiler generally misinterprets the construct.

EXAMPLES

The Identification, Environment, Data, and Procedure Divisions, including appropriate headers, appear in figure 2-2 as Items 1 through 4, respectively.

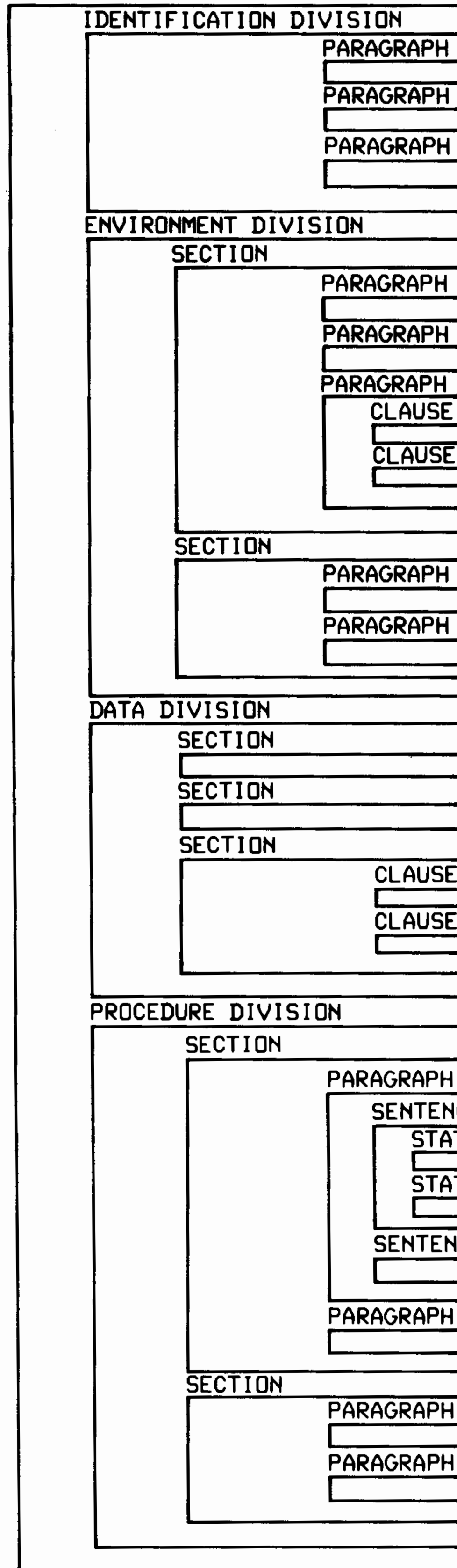


Figure 2-1.

COBOL PROGRAM STRUCTURE HIERARCHY

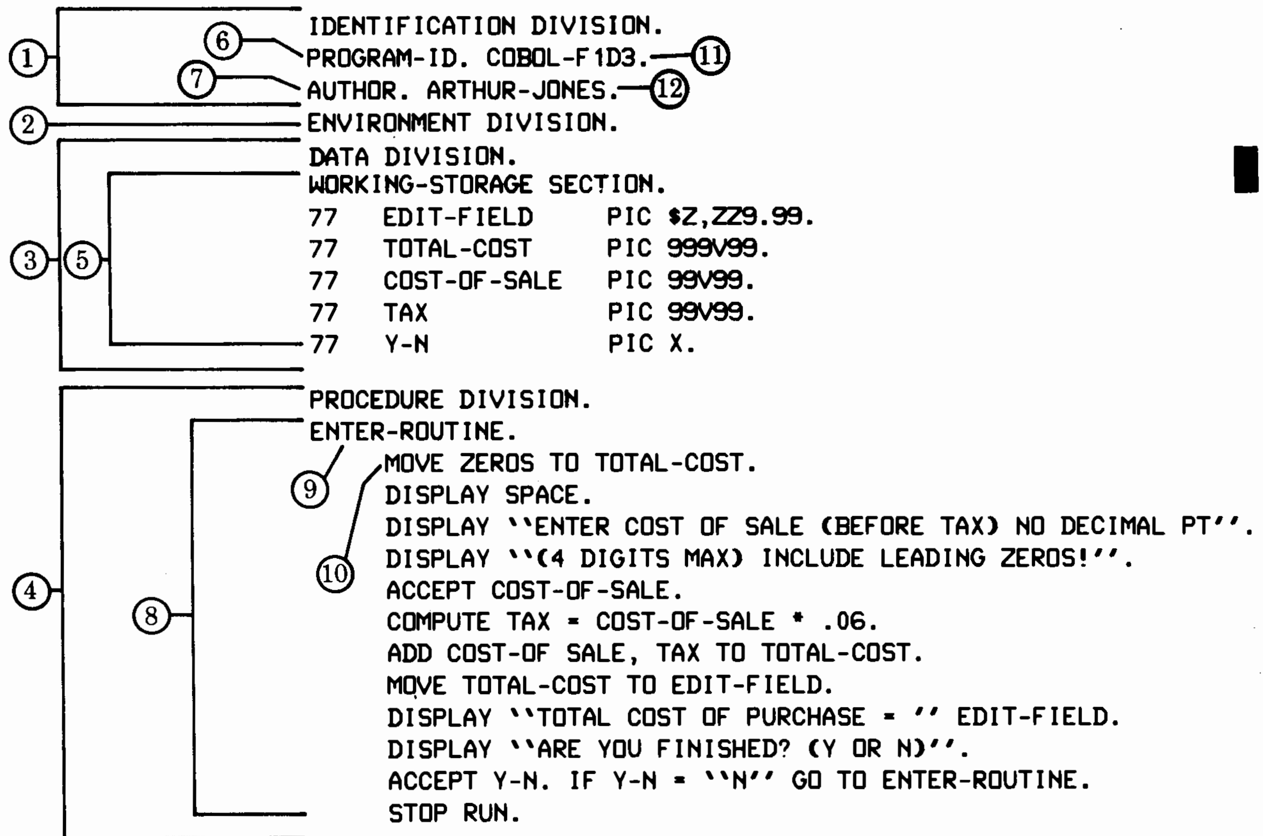


Figure 2-2.
PROGRAM STRUCTURE EXAMPLE

Sections

A section is the second-level construct in a COBOL program. In the source program, sections allow you to group logically related items together within a division. As the object program is compiled, sections can be grouped into segments—the basic units of code that are swapped to and from main memory during program execution. In the Procedure Division, you can organize logically related functions into the same sections in such a way that often-used routines reside in main memory for longer periods of time than routines used infrequently. This minimizes the total number of input/output operations that MPE must perform on the code segments belonging to the program. It also facilitates program debugging. In other divisions, the features that a program uses determine which sections must be specified in the program.

Sections are optional in the Procedure Division. At least one section is required in the Environment Division and in the Data Division if a division body is present. Sections are prohibited, however, in the Identification Division. If you do not specify sections in a division, the entire division is treated as a single section.

SECTION FORMAT

Each section begins with a header entry that is optionally followed by zero, one, or more paragraphs (in the Environment or Procedure Division) or clauses (in the Data Division). The paragraphs or clauses comprise the section body. A section is terminated by the next section header, the next division header, the END DECLARATIVES keywords (in the declarative portion of the Procedure Divison), or the physical end of the program.

SECTION HEADER FORMAT

In the Environment and Data Divisions, the section header consists of a COBOL reserved word that identifies the section, followed by the word SECTION, followed by a period and a space. In the Environment Division, only the following section headers are permitted:

CONFIGURATION SECTION.

INPUT-OUTPUT SECTION.

In the Data Division, only these section headers are allowed:

FILE SECTION.

WORKING-STORAGE SECTION.

LINKAGE SECTION.

COMMUNICATION SECTION.

REPORT SECTION.

In the Procedure Division, the section header consists of a user-defined section name that identifies the section, followed by the word SECTION, followed by an optional segment number, followed by a period and a space. In the Procedure Division, unlike the Environment and Data Divisions, names are not restricted to specific words—you can supply any section names you desire. Examples of section headers that might be used in the Procedure Division are:

INITIALIZATION SECTION.

INPUT-GENERATION SECTION.

CALCULATION SECTION.

HOUSEKEEPING SECTION 3. ←——— Segment number

The segment number is used in program segmentation (partitioning of a program into discrete code segments) as discussed in section X.

For clarity, programmers usually write a section header on a line by itself, although they are not formally required to do so.

EXAMPLES

In figure 2-2, the WORKING-STORAGE SECTION (Item 5) appears in the Data Division. Because no section is specified in the Procedure Division, the whole division is regarded as a section by the compiler.

Paragraphs

A paragraph is the third-level construct in a COBOL program. Paragraphs allow you to break down your program into even more elementary units. One paragraph (the PROGRAM-ID paragraph) is required in the Identification Division. Paragraphs are optional in the Environment and Procedure Divisions. They are not used in the Data Division.

Paragraph Format

In the Identification and Environment Divisions, each paragraph begins with a header entry, optionally followed by one or more words or clauses that comprise the paragraph body. In the Procedure Division, a paragraph begins with a paragraph name, optionally followed by one or more sentences comprising the paragraph body. In any division, a paragraph is terminated by one of the following:

- The next paragraph header or name
- The next section or division header
- The physical end of the program in the Procedure Division
- The words END-DECLARATIVES in the Procedure Division.

PARAGRAPH HEADER/NAME FORMAT

The paragraph header, used in the Identification and Environment Divisions, consists of a COBOL-reserved word identifying the paragraph, followed by a period and a space. In the Identification Division, only the following headers are permitted:

PROGRAM-ID.

AUTHOR.

INSTALLATION.

DATE-WRITTEN.

DATE-COMPILED.

SECURITY.

In the Environment Division, only these headers are allowed:

SOURCE-COMPUTER.

OBJECT-COMPUTER.

SPECIAL-NAMES.

FILE-CONTROL.

I-O-CONTROL.

The paragraph name, used in the Procedure Division, is a user defined word that identifies the paragraph, and is always terminated by a period and a space. It must be unique within a section, or in a program if no sections are defined. If sections are used, however, the same paragraph-name may appear in different sections. When referencing such a paragraph, you can use the section name to qualify the paragraph-name, and you must do so if you are referencing to it from within a section other then the section in which it is defined.

The paragraph header or name must be the first item on a coding line, but may be followed by other items on the same line.

EXAMPLES

In figure 2-2, the Identification Division contains paragraphs identified by the headers PROGRAM-ID (Item 6) and AUTHOR (Item 7). The Procedure Division includes one paragraph (Item 8) identified by the user-defined name ENTER-ROUTINE (Item 9).

Sentences, Statements, and Clauses

Within a paragraph, sentences (in the Procedure Division) and clauses (in the Identification, Environment, and Data Divisions) may appear. Within a sentence, in turn, one or more statements can be written. These items provide a further syntactic breakdown of your program. In structure, they closely resemble their English-language counterparts.

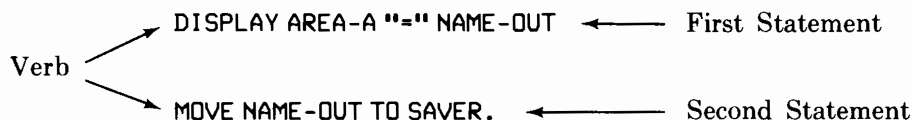
A *sentence* is a sequence of one or more statements, with the last statement terminated by a period followed by a space.

A *statement* is a syntactically-valid combination of words and symbols, beginning with a verb such as ADD, READ, or DISPLAY.

A *clause* is an ordered set of character strings (sequences of characters) that specify an attribute of an entry in the program.

All of these items are explained further in the discussions of the divisions in which they appear.

In figure 2-2, the Procedure Division begins with a sentence that contains a single statement—MOVE ZEROS TO TOTAL COST. (Item 10). As a further example, the sentence shown below contains two statements:



In figure 2-2, the Identification Division contains the clauses COBOL-F1D3 (Item 11) and ARTHUR-JONES (Item 12).

Statements and clauses may also include phrases. A phrase is a sequence of one or more consecutive character strings that form a portion of a statement or clause. In the example below, the characters RED-DATA OF COLOR-DATA form a phrase:

```
MOVE RED-DATA OF COLOR-DATA TO FORM-DATA.
```

PROGRAM ELEMENTS

SECTION III

All COBOL language constructs are made up of basic elements comprised of character strings and separators. A *character string* is a character or sequence of characters that forms a COBOL word, literal, PICTURE character string, or comment entry (as defined later in this section). Every character string is delimited at its beginning and end by a separator which is either a single special character (such as a period, comma, semicolon, or blank) or a sequence of special characters. The characters used in character strings and separators are selected from the COBOL character set, described at the end of this section.

Character Strings

Character strings may form:

- Words
- Literals
- PICTURE character strings
- Comment-entries

WORDS

In COBOL, a *word* is generally the name of some entity such as a function, paragraph, register, section, data-item, constant, or other syntactical term used in a format description. There are three types:

- Reserved words
- User-defined words
- System names

Each word is limited to a maximum length of 30 characters. Certain types of words, such as user-defined words, may be restricted to a shorter length.

Reserved Words

A COBOL *reserved word* belongs to a specific list of words (shown in Appendix K) that always have consistent meaning within all COBOL programs. Reserved words, therefore, are always interpreted in the same way by the compiler. For instance, the reserved word SECTION always denotes a COBOL section header. As a programmer, you may not define new reserved words. The following types of reserved words may appear in your program:

- Key words
- Optional words
- Connectives
- Special register words
- Figurative constant words
- Special character words

KEY WORDS

A *key word* is a reserved word that is required to convey the meaning of a statement or clause in whose format description it appears. In other words, if your program contains a statement or clause whose definition includes the key word, you must enter the key word. In the division header below, the words PROCEDURE, DIVISION, and USING are all keywords. (In the format descriptions in this manual, all keywords are denoted by underlined uppercase letters.) The USING clause is optional, as indicated by the brackets. But if you use this clause in your program, you must include the keyword USING.

PROCEDURE DIVISION [**USING** ***data-name-1*** [, ***data-name-2***] ...] .

OPTIONAL WORDS

An *optional word* is a reserved word that may be included in, or omitted from, a statement or clause at will. It has no effect on program execution and serves only to make source program listings more readable and easy to understand. In the following clause, the words MODE and IS are optional. (In the format descriptions, all optional reserved words are denoted by uppercase letters that are not underlined.)

```
[ ; ACCESS MODE IS { SEQUENTIAL } ]
[                  { RANDOM     } ]
[                  { DYNAMIC    } ]
```

CONNECTIVES

A *connective* is a reserved word that is used for one of these reasons:

1. To associate a data-name, paragraph-name, condition-name, or text-name with a qualifier word. (These names are specifically defined later; typically, they include the connectives OF and IN.) For instance, the connective OF associates the data-name PARK with the qualifier word CITY:

   ```
   PARK OF CITY
   ```

2. To separate two or more consecutive operands in an expression. In such cases, connectives such as a comma (,) or semicolon (;) may be used. For instance, in the following example, the comma is used to separate the operands A+1 and B+3:

   ```
   TABX (A+1, B+2)
   ```

3. To indicate the relationship between elements in conditional statements. There are two of these conditional connectives—AND and OR. As an example, the connective AND appears in the following statement:

   ```
   IF A AND B; GO TO ROUTINE-1; ELSE GO TO ROUTINE-2.
   ```

SPECIAL REGISTER WORDS

A special register is a storage area in main memory that contains information primarily used in connection with specific COBOL features. The content of this area is generated automatically by the compiler. In a COBOL program, such an area is referenced by a *special register word*. Those special register words that are part of ANSI COBOL 74 are indicated in table 3-1. The COBOL II/ 3000 special register words that represent extensions to ANSI COBOL are denoted in table 3-2.

Table 3-1.
SPECIAL REGISTER WORDS

WORD	CONTENTS
LINAGE-COUNTER	An eight-digit unsigned number used to keep track of the number of lines written to each page of a printed report. It is generated for each output file whose description in the Data Division contains a LINAGE clause (which defines the number of lines permitted per page). The register is initialized to zero and then updated each time a line is written with the WRITE statement in the Procedure Division. When this value exceeds the number specified by integer-1 or data-name-1 in the LINAGE clause, the program skips to the next page and re-sets the register to 1.
LINE-COUNTER	An eight-digit unsigned integer that is used to determine when to print page and overflow headings and trailers, and to control spacing. NOTE This register is not implemented in COBOL II/3000.
PAGE-COUNTER	An eight-digit unsigned integer used by the program to number the pages of a printed report. NOTE This register is not implemented in COBOL II/3000.
DEBUG-ITEM	A data-item used in support of the COBOL DEBUG facility. The compiler automatically generates one DEBUG-ITEM register for each program. NOTE This register is not implemented in COBOL II/3000.

Table 3-2.

EXTENSIONS TO SPECIAL REGISTER WORDS

WORD	CONTENTS
TALLY	A five-digit unsigned integer typically used to store information produced by the EXAMINE statement in the Procedure Division. (This statement counts the occurrences of a particular character within a data-item and optionally replaces all instances of that character with another character.) This register may also be used as a data-name for an unsigned numeric value with no decimal positions—for instance, as a subscript.
CURRENT-DATE	An eight-digit alphanumeric item used only as the transmitting field in a MOVE or DISPLAY statement in the Procedure Division. (These statements transmit data to another field or to an output device, respectively.) This item is always stored in this format: *mm/dd/yy* Here, *mm* indicates the month, *dd* indicates the day of the month, and *yy* indicates the last two digits of the year. The slash- marks are automatically included in the data, and should not be inserted by the programmer.
WHEN-COMPILED	An eight-character alphanumeric item that represents the date that the program is compiled. It may be used only in MOVE and DISPLAY statements of the Procedure Division. This field is automatically stored as follows, with slash-marks inserted: *mm/dd/yy* As in the CURRENT-DATE format, *mm* means current month, *dd* means day of month, and *yy* indicates the year.
TIME-OF-DAY	A six-character numeric item accessed only as the transmitting field of a MOVE or DISPLAY statement in the Procedure Division to access the time of day. This data may be used to determine the clock-time required to run a COBOL program; this is done by printing the contents of this register at the beginning and end of the program. Remember however, that clock-time in a multiprogramming environment is not necessarily related to the central-processor time used by the program—it varies according to the current mix of active programs. The data is always stored in this format: *hhmmss* *hh* indicates the current hour, *mm* the current minute, and *ss* the current second, relative to midnight. The data is unedited. However, the statement DISPLAY TIME-OF-DAY results in the edited format: *hh:mm:ss*

FIGURATIVE CONSTANT WORDS

Figurative constants are values that have been used so often that they have been assigned fixed data-names within the COBOL language. These data-names are called *figurative constant words.* For example, the figurative constant consisting of a string of zeros is referenced by the figurative constant word ZERO. The constants are generated automatically by the compiler and do not require definition within your program. The figurative constant words are shown in table 3-3; the singular and plural forms of these words may be used interchangeably.

Table 3-3.
FIGURATIVE CONSTANT WORDS

WORD	CONSTANT VALUE
ALL *literal*	The character string denoted by the variable *literal.* This string may be either a non-numeric literal (as defined later in this section), or another figurative constant (such as ZERO). If a literal is used, it must be enclosed in quotation marks. If a figurative constant is used, the word ALL is redundant.
HIGH-VALUE HIGH-VALUES	One or more occurrences of the character with the highest possible value in the ASCII Collating Sequence (all eight bits on). The ASCII equivalent of this character is not used on the HP 3000, but this bit configuration is equivalent to the hexadecimal character FF.
LOW-VALUE LOW-VALUES	One or more occurrences of the character with the lowest possible value in the ASCII Collating Sequence (all eight bits off). This is the nonprinting character NULL.
QUOTE QUOTES	One or more quotation marks. This constant is used to code the quotation mark as a literal in statements such as MOVE QUOTES. However, the word QUOTE or QUOTES cannot be used in place of an explicit quotation mark (") to delimit a nonnumeric literal—thus, QUOTE ABD QUOTE cannot be substituted for the non-numeric literal "ABD".
SPACE SPACES	One or more spaces.
ZERO ZEROS ZEROES	One or more occurrences of the digit 0.

Figurative constant words must not be bounded by quotation marks or apostrophes.

The length of the character string represented by a figurative constant word is determined by the size of the field to which the constant is moved or with which it is associated, as follows:

1. When the constant is associated with another data-item, as in a VALUE clause or when the constant is moved to or compared with another item, the constant assumes the same length as the associated item. The string of characters represented by the constant is repeated, character by character, until the size of the resultant string equals that of the associated data-item. Thus, when the constant word group ALL literal is used, the literal specified is repeated until the associated data-item is filled with the value of literal. For example, if FIELD-A is defined as a ten-character item, the statement MOVE ALL "123" TO FIELD-A produces the following result:

   1231231231

2. When the constant is not associated with some other item, as when used in a DISPLAY, STRING, UNSTRING, EXAMINE, or STOP statement, then the constant assumes a length of one character. In such cases, the figurative constant ALL may not be used.

A figurative constant may be referenced wherever a literal appears in a format decription, except that literals restricted to numeric characters only may be replaced by the figurative constant words ZERO, ZEROS, or ZEROES only.

Use of figurative constant words is demonstrated by the following MOVE and DISPLAY statements, which transmit the value of the constants referenced to the storage areas denoted by STORE-n:

Example	Comment
MOVE QUOTES TO STORE-1	Suppose STORE-1 is an area six character-positions long. When this statement is executed, STORE-1 contains: """"""
MOVE ALL "NEGATIVE" TO STORE-2	Suppose STORE-2 is twelve positions long. It contains: NEGATIVENEGA
MOVE SPACES TO STORE-3	Suppose STORE-3 is nine positions long. It contains all spaces.
DISPLAY QUOTE "BETA" QUOTE UPON STORE-4	Suppose STORE-4 is six positions long. It contains: "BETA" Note that quotation marks delimit the literal value BETA in the program. These, however, are not transmitted to STORE-4. Instead, the quotation marks in STORE-4 are supplied by the figurative constant QUOTE.

SPECIAL-CHARACTER WORDS

A *special-character* word is a reserved word, grouping of reserved words, or character that represents an arithmetic or relational operator. These words are listed in table 3-4.

Table 3-4.
SPECIAL-CHARACTER WORDS

Arithmetic		Relational	
Operator	**Meaning**	**Operator**	**Meaning**
+	Addition.	IS [NOT] GREATER THAN IS [NOT] >	Greater-than or not greater-than.
–	Subtraction.	IS [NOT] LESS THAN IS [NOT] <	Less-than or not less than.
*	Multiplication.	IS [NOT] EQUAL TO IS [NOT] =	Equal-to or not equal-to.
/	Division.		
**	Exponentiation.		

User-Defined Words

A *user-defined word* is a word that the programmer must supply, to satisfy the format of a statement or clause. Such words act as arbitrary variables that name or identify various program items. These words include such elements as program-names, section-names, paragraph-names, and data-names. In the Procedure Division header below, *data-name-1* and *data-name-2* are user-defined words. (In the format descriptions, all user-defined words are denoted by italic lowercase letters.)

PROCEDURE DIVISION [USING *data-name-1* [, *data-name-2*] ...] .

When you enter the Procedure Division header shown above, you might arbitrarily supply the data-names ALPHA, BETA, and GAMMA in the format, as follows:

```
PROCEDURE DIVISION  USING ALPHA, BETA, GAMMA.
```

User-defined program-names may contain up to 15 characters. User-defined data-names, procedure-names, and section-names may contain up to 30 characters. These include letters (A through Z), digits (0 through 9), and the hyphen (-). Except for paragraph-names, section-names, segment-numbers, and level-numbers, all user-defined words must contain at least one alphabetic character (letter). However, user-defined words cannot begin with a hyphen, include an embedded space, or have the same spelling as any reserved word.

In specific formats, the rules covering user-defined words may be more restrictive. Where such rules apply, they are explained in the discussion of the statement or clause in which the word appears.

In ANSI COBOL, 16 types of user-defined words are permitted. These are defined in table 3-5. In COBOL II/3000, 14 of these types are implemented; two (report-name and routine-name) are not, although they are accepted by the compiler and treated as comments.

Table 3-5.
USER-DEFINED WORD TYPES

WORD TYPE	DEFINITION
Alphabet-name	Word that identifies (names) a specific character set or collating sequence to be used by the program. Defined in Special Names paragraph of Environment Division; used in CODE SET clause of Data Division and in Collating Sequence phrase of SORT and MERGE statements in Procedure Division. Must also be named in the Program Collating Sequence clause of the Configuration Section in the Environment Division in order to specify a collating sequence to be used throughout your program.

Condition-name	Word that identifies a specific value, or subset or range of values, within a complete set of values that a data-item may assume. (This data-item is called a conditional variable.) It may be defined in the Data Division, where the condition-name is preceded by the level number 88 and followed by a VALUE clause. In the following example, the condition names FIRST-CONST, SECOND-CONST, and THIRD-CONST appear:
	`02 CONST PICTURE 99.`
	`88 FIRST-CONST VALUE IS 10.`
	`88 SECOND-CONST VALUE IS 20.`
	`88 THIRD-CONST VALUE IS 30.`
	A condition-name may also appear in the Special Names paragraph of the Environment Division, where it is assigned to denote the status of switches, or used in the RERUN clause or as an abbreviation for specific conditions.

Data-name	Word that identifies a data-item. Used in data-description entries in the Data Division. Cannot be subscripted, indexed, or qualified unless specifically permitted by the format description in which it appears.

Table 3-5 (Continued).
USER-DEFINED WORD TYPES

WORD TYPE	DEFINITION
File-name	Word that identifies a data file. Used in File Description entry or Sort/Merge File Description entry in Data Division.
Index-name	Word that identifies an index associated with a specific table, and used to select an item from that table. Used in Data and Procedure Divisions.
Level-number	Word that indicates the position of a data- item in the hierarchical structure of a logical record, or that indicates special properties of a data-description entry. Level numbers 1 through 49 indicate the position in a record structure; level numbers 66, 77, and 88 identify special properties. In the example that appears in the condition-name discussion, above, level numbers 02 and 88 are used. In single-digit level numbers, leading zeros may be optionally added. Used in the Data Division.
Library-name	Word that identifies a COBOL library (containing source text) used as input by the compiler during a particular compilation. Used in all divisions.
Mnemonic-name	Word equated to a name that identifies a special feature of the computer system on which the program is compiled or run. This relationship is established in the Special Names paragraph of the Environment Division.
Paragraph-name	Word that identifies and begins a paragraph in the Procedure Division.
Program-name	Word that identifies a COBOL-language source program. Used in the Identification Division.

Table 3-5 (Continued).
USER-DEFINED WORD TYPES

WORD TYPE	DEFINITION
Record-name	Word that identifies a logical record in a data file. Used in a Record Description entry in the Data Division and WRITE statement in the Procedure Division.
Report-name	Word that identifies a printed report not implemented in COBOL II/3000. Used in Report Section of Data Division, but COBOL II/3000 treats it as a comments.
Routine-name	Word that identifies a procedure written in a language other than COBOL. Used in Procedure Division in ANSI COBOL, but COBOL II/3000 treats it as a comment.
Section-name	Word that identifies and begins a section in the Procedure Division.
Segment-number	Word that classifies sections in the Procedure Division for purposes of program segmentation. Must be one of the numbers 0 through 99; leading zeros are optional.
Text-name	Word that identifies text within a source library. Used in all divisions.

Within any COBOL program, 12 of the 14 types of user-defined words must be regarded as belonging to 10 disjoint sets. These sets are:

- Alphabet-names.
- Condition-names, data-names, and record-names.
- File-names.
- Index-names.
- Library-names.
- Mnemonic-names.
- Paragraph-names.
- Program-names.
- Section-names.
- Text-names.

All user-defined words within the same program, except segment-numbers and level-numbers, can belong to only one of these disjoint sets—that is, if the word TEST-1 is used as a program name, it cannot also be used as a routine-name. Furthermore, all such words must be unique within a disjoint set, either because no other user-defined word in the set is spelled and punctuated the same way or because uniqueness is ensured by qualification. In other words, a program cannot include two paragraphs both named PAR-A unless special qualification is made.

NOTE

> The general term procedure-name is often used to refer to either a section-name or a paragraph-name in the Procedure Division.

Further examples of valid user-defined names are:

```
END-OF-SCHOOL-AVERAGE

PAGECTR

A-B-C

HDR-1

123B
```

System Names

A *system name* is a word that is used to define the operating environment in which the COBOL program is compiled or run. It permits communication between the program and this environment. There are two types of system names:

- Computer-name, used to identify the computer on which the program is to be compiled or run. This name appears in the Configuration Section of the Environment Division.
- Language-name, used to specify the language in which the program is written. This name is used in the ENTER statement of the Procedure Division.

In COBOL II/3000, all system names are purely optional. When included, they are treated as comments—that is, they appear on source program listings but do not affect compilation or execution. Nevertheless, when present, system names may contain only letters (A through Z), digits (0 through 9), and/or one hyphen. The first character of a system name must be alphabetic; also, if a hyphen is used, it must not be the last character in the name.

Literals

A *literal* is a character string that defines itself, rather than representing some other value. In other words, the value of the literal is the character string composing the literal.

Literals are always constants—values that cannot be changed during program execution. Because literals are self-defining, you do not define them in the Data Division. Instead, you simply code them directly into your program.

In COBOL, two types of literals are used: numeric and non-numeric.

NUMERIC LITERALS

A *numeric literal* is essentially a number (numeric value) that is specified directly in a program. It is comprised of characters selected from the digits 0 through 9, the plus sign (+), the minus sign (-), and/or a decimal point (.). As an example, the literal 1 appears in the ADD statement below:

```
ADD 1 TO PAGE-NUMBER.
```

The specific value of the literal is the algebraic quantity represented by the characters that compose the literal. In the format descriptions, literals are indicated by the word literal-n, shown in lowercase letters—for instance, literal-1, literal-2, and so forth. Every numeric literal must contain:

- At least one digit.
- No more than 18 digits.
- No more than one arithmetic sign (+ or -). If a sign is used, it must appear as the leftmost character in the literal. If no sign is used, the literal is treated as a positive value by the compiler.
- No more than one decimal point. If a decimal point is used, it may appear anywhere in the literal except as the rightmost character. (Any decimal used as the rightmost character is interpreted as a period that terminates a sentence.) If no decimal point is used, the literal is treated as an integer.

NOTE

> If a character string follows the rules for the formation of a numeric literal but is enclosed in quotation marks, it is treated by the compiler as a non-numeric literal. (See the discussion below.) As such, it cannot be used in arithmetic operations.

Further examples of numeric literals are:

```
1670037627

3.415

+2

-30.06
```

As an extension to ANSI COBOL, COBOL II/3000 allows the use of octal numeric literals. These are always immediately preceded by a percent sign (%), as shown in the following examples:

```
% 17
% 37777777777
% 456
```

NON-NUMERIC LITERALS

A *non-numeric literal* is a character string containing letters, digits, and/or special characters that is coded directly into a program. It is formed by entering:

- A quotation mark or apostrophe that denotes the beginning of the literal,
- The character string that comprises the literal, and
- A quotation mark or apostrophe that delimits the end of the literal.

NOTE

> The delimiting quotation marks or apostrophes are not considered part of the literal.
>
> All punctuation marks within a non-numeric literal are treated as ordinary punctuation marks, not as delimiters or separators.

Note from the above description that you may use either quotation marks or apostrophes as delimiters. There is no restriction on which set of delimiters may be used at a given time. This allows you a great amount of freedom in forming non-numeric literals. For example, to display the message,

```
PLEASE ENTER "AGE UNDETERMINED" IF UNSURE
```

on your terminal screen, you can use the following DISPLAY statement:

```
DISPLAY 'PLEASE ENTER "AGE UNDETERMINED" IF UNSURE'
```

As an example of invalid usage, the character string, 'I DON'T KNOW', is interpreted by the COBOL II/3000 compiler as being the string, 'I DON', followed by the characters, T KNOW'. In this case, a syntax error would be generated. Since you can use quotation marks and apostrophes throughout your program to delimit non-numeric literals, the above string could be made valid by using quotation marks:

```
DISPLAY "I DON'T KNOW"
```

Note that you may use two consecutive quotation marks, or two consecutive apostrophes, within the characters of a non-numeric literal to represent a single quotation mark or apostrophe. Thus, for example, the DISPLAY statement above could also have been written as shown below:

```
DISPLAY 'I DON''T KNOW'
```

The results of executing this statement would be the message,

```
I DON'T KNOW
```

on your terminal screen (or line printer; see the DISPLAY statement). If you use double apostrophes in a non-numeric literal, and the literal is bounded by quotation marks, then both apostrophes are used as part of the literal. The opposite is also true. To illustrate:

```
DISPLAY "DOUBLE APOSTROPHES, '', ARE PART OF THIS LITERAL"
```

results in

```
DOUBLE APOSTROPHES, '', ARE PART OF THIS LITERAL
```

and

```
DISPLAY 'DOUBLE QUOTES, "", ARE PART OF THIS LITERAL'
```

results in

```
DOUBLE QUOTES, "", ARE PART OF THIS LITERAL
```

A character-string may be from 1 to 132 characters long, and may consist of any of the characters from the ASCII collating sequence. See Appendix G for a list of these characters.

Note that the figurative constant words QUOTE and QUOTES cannot be used to supply delimiting quotation marks for non-numeric literals.

Picture Character Strings

The PICTURE character string appears in the PICTURE clause of the Data Division. This clause describes the characteristics and editing requirements of data that is typically destined for some external output device such as a terminal or line printer. Specifically, the PICTURE clause determines the appearance of the field that is actually output by specifying:

- The size of the field.
- The class (type) of data that can be written into the field—alphabetic, numeric, and alphanumeric.
- Whether or not a numeric sign appears in the field.
- The position of the decimal point, if any.
- The editing required to insert, suppress, or replace characters in the field.

The PICTURE clause supplies currency-signs, leading or trailing zeros, commas, plus or minus signs, and other punctuation stated in the PICTURE string. Often, for example, it is used to suppress leading zeros on checks, replacing them with asterisks or spaces. As an example, if the data to be printed was comprised of the digits 8765432123 and you wished to print it as a dollar-value with appropriate punctuation, you could specify the following PICTURE character string in the PICTURE clause:

```
$99,999,999.99
```

In this string, the 9-digits are used as special symbols—they specify the character positions that will be filled with numeric data. The dollar-sign, commas, and decimal-point indicate the positions of the punctuation characters. The PICTURE character string is superimposed on the output data so that the following information is printed:

```
$87,654,321.23
```

Complete details about PICTURE character strings appear in section IX.

COMMENT-ENTRIES

A *comment-entry* is used in the Identification Division to include comments or remarks in the program. These comments appear on the source program listing but do not affect program compilation or execution. They denote such items as: program author, installation name, date written, date compiled, security requirements, and other general remarks. They may include any printable character from the ASCII Character Set.

All comment-entries are optional. When included, however, they must also conform to the rules for paragraph and sentence structure discussed in this manual.

Examples of comment-entries appear in the boxed area in the figure below.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. PAYROLL.

AUTHOR:   DON R. SMITH.
INSTALLATION.  HP 3000 SITE II.
DATE-WRITTEN.  3/29/80
DATE-COMPILED.  3/29/80
SECURITY.  LEVEL-I.
```

Comment Lines

A *comment line* is any line with an asterisk in the continuation indicator area (column 7) of the line.

A comment line can appear anywhere in your program following the Identification Division header. It can be made up of any combination of characters from the ASCII collating sequence, with all characters (except the asterisk in column 7) contained in columns 8 through 72 of the line.

Additionally, you may use a special form of comment line to cause page ejection prior to printing the comment. This special comment line is the same as the general one described above, except that a slash character (/) is placed in column 7 instead of an asterisk.

Separators

A *separator* is a punctuation character that delimits a character string. Separators include:

- Spaces (one or more).
- Commas, semicolons, and periods, when immediately followed by a space. These may only appear where specifically permitted by format descriptions, definitions, and rules.
- Left and right parentheses. These must only appear as balanced pairs used to delimit subscripts, indices, arithmetic expressions, or conditions.
- Quotation marks or apostrophes. These delimit non-numeric literals, and must appear as balanced pairs (except as noted under Continuation Lines in section V). An opening quotation mark or apostrophe must be immediately preceded by a space or left parenthesis. A closing quotation mark or apostrophe must be immediately followed by a space, comma, semicolon, period, or right parenthesis.
- Sets of two contiguous equal signs (==), used to delimit pseudo-text. (Pseudo-text is text incorporated into, or replaced in, a COBOL program by the COPY statement.) An opening delimiter must be immediately preceded by a space; a closing delimiter must be immediately followed by a space, comma, semicolon, or period. These delimiters must appear in balanced pairs.

Any of the above separators may, at your option, be immediately preceded by one or more spaces, except if specifically prohibited by format rules. (A space preceding a closing quotation mark is treated as part of the literal enclosed by this and the preceding quotation mark.)

Any of the above separators except the opening quotation mark may be optionally followed immediately by one or more spaces. (A space following an opening quotation mark is considered as part of the literal enclosed by this and the next following quotation mark.)

NOTE

> The above rules do not apply to punctuation characters within non-numeric literals, comment-entries, comment lines, or PICTURE character strings. Those characters are not regarded as separators.

Character Set

Most character strings and all delimiters in a COBOL program are formed from characters selected from the COBOL Character Set. This character set includes 51 characters—letters (A through Z), digits (0 through 9), and certain special characters:

Special Character	Meaning	
+	Plus sign	
-	Minus sign	
*	Asterisk	
/	Slash	
=	Equal sign	
$	Currency sign	
,	Comma	
;	Semicolon	
.	Period (decimal point)	
"	Quotation mark	
'	Apostrophe	
(	Left parenthesis	
)	Right parenthesis	
>	Greater-than symbol	
<	Less-than symbol	
	Space	
%	Percent sign	COBOL II/3000 Extensions to ANSI COBOL (applies to %, @, \)
@	At-sign	
\	Back-slash	

In the case of non-numeric literals, comment-entries, and comment-lines, additional characters may be used. These characters are selected from the ASCII Character Set listed in Appendix G, (which includes the COBOL Character Set).

NOTE

When lowercase letters appear outside of literals, the COBOL II/3000 compiler automatically converts them to uppercase letters.

Each character in the ASCII Character Set has a unique octal value that establishes its order in the collating sequence of this character set. In Appendix G, the characters are listed from top to bottom, in order of ascending value.

NOTE

When COBOL programs originally written for other systems are run on an HP 3000 Computer System, variations between the collating sequences for both systems may cause variations in output. For example, the Binary Coded Decimal (BCD) and Extended Binary Coded Decimal Information Interchange Code (EBCDIC) both collate the letters of the alphabet before (lower than) the digits 0 through 9. This differs from the ASCII collating sequence, where digits are collated before letters. In addition, several special characters collate differently in various sequences. To permit valid processing when a different collating sequence is used, you may specify the appropriate sequence in the ALPHABET-NAME and PROGRAM COLLATING SEQUENCE clause of the Environment Division.

DESCRIBING AND REFERENCING DATA

SECTION IV

The data used by a COBOL program is defined and described in the Data Division, and referenced and operated upon in the Procedure Division. This data is stored in, read from, and written to files (collections of information) that reside on various peripheral devices. For instance, a payroll-processing program might accept input from a file that contains wage and salary information for all employees on the company payroll; this program might also write new output to this same file during updating operations.

Within a file, all information is organized into units of related data called logical records. These records are similar in form, purpose, and content. For example, in the payroll file, each logical record could contain the wage and salary data related to a particular employee; there would be one record for each employee.

Within each record, individual elements of data, or groups of such elements, are called data-items. As an example, a payroll record for an employee might contain the following data items: the employee's name, social-security number, marital status, gross pay, tax exemptions, individual deductions, and net pay. The individual deductions data-item might itself contain subordinate data-items, such as federal income tax, state income tax, insurance premiums, bond payments, and charity contributions.

Files

In COBOL, a file is a collection of records that is identified by a unique name and is currently recognizable by your program. The name allows you to reference the file in your program. Other specifications define how records are organized within the file with respect to the physical device on which the file is stored. The specifications for the file are defined in the Input-Output Section of the Environment Division and the File Description Section of the Data Division.

Records

Each logical record constitutes a group of related information, uniquely identifiable and treated as a unit. A record is actually the most inclusive data-item in a file—that is, each input/output statement in the Procedure Division accesses one logical record, although it may also extract subordinate data-items from that record.

LOGICAL VERSUS PHYSICAL RECORDS

A physical record is one or more logical records, and is commonly called a block. A block is the physical unit used by the MPE file system to read data from a file, or write data to it; it is the basic unit transferred between the device on which the file resides and main memory each time a program executes an input or output operation.

You may use the BLOCK CONTAINS clause to specify the size of (that is, the number of logical records contained in) a physical record.

For files on magnetic tape or disc, a block consists of either one logical record or a group of several logical records—for instance, 2, 16, or 256 logical records could be grouped into one block. (For tape files, blocking is normally done to improve execution time or to conserve file space by reducing the number of interrecord gaps on the tape.) For files on card readers and punches, line printers, and terminals, each block is identical to each logical record, and its length is determined by the type of device. Thus, each block/ logical record read from a card reader consists of one 80 character punched card; each block/logical record written to a line printer consists of one line of print—typically 132 characters. The size of a block has no relation to the size of the data file contained on the device to or from which the block is transferred.

A single storage device may hold one or more logical records.

NOTE

In this manual, the term record refers to logical records unless the term block or physical record is specifically used.

COBOL allows you to define logical records in main memory as well as in files stored on peripheral devices. This is done through the Working Storage Section of the Data Division (see Section VIII).

Record Descriptions

Each record in a file is defined by a Record Desciption entry in the Data Division. This entry, in turn, consists of one or more Data-Description entries that collectively define the characteristics of the record. Each Data-Description entry consists of these elements, in the order listed:

- **Level-number** that indicates a subdivision or portion of the logical record—a data-item.
- **Data-name** that allows you to identify and reference the data-item.
- **Independent** clauses that describe the attributes of the data-item.

To reference portions of the information in a logical record, you must subdivide the record into corresponding data-items; you must also identify each data-item that you wish to reference with a name. Once you specify any data-item, you may further subdivide it into subordinate data-items to permit more detailed data-reference.

The level-number indicates the hierarchical order of a data-item within the record structure. Examples appear in figure 4-1. Since a record is the most inclusive data-item your program can reference, it is assigned the level-number 01. Less-inclusive data-items are assigned numerically higher level-numbers, ranging from 02 through 49. These numbers need not be successive. The most basic subdivisions of a record—those data-items that have no further subdivisions—are called elementary items. Items with subdivisions are called group items, or simply groups. Within the Record Description entry, each group includes all following group and elementary items until an item with a level-number less than or equal to the level-number of that group is encountered.

A record is considered a single elementary item if it is not subdivided; otherwise, it is regarded as a sequence of elementary items that may or may not be organized into groups. Because of the hierarchical structure of the record, a basic element may belong to more than one group—its immediate group and higher-level groups that contain that group. In the Procedure Division, your program may refer to the entire record, or to any group of any level within that record, or to an elementary item.

In figure 4-1, a record named PERSONNEL-RECORD (Item 1) is defined in the Data Division. This record is divided into the various group items:

- Two main group items, named EMPLOYEE-ID (Item 2) and ADDRESS (Item 3).
- The EMPLOYEE-ID group item is subdivided into two subordinate group items—EMPLOYEE-NUMBER (Item 4) and SOCIAL-SECURITY (Item 5). The ADDRESS group item is subdivided into the two group items STREET (Item 6) and LOCATION (Item 7).
- Another group item, ZIP (Item 8), of the same level as EMPLOYEE-NUMBER, SOCIAL-SECURITY-NUMBER, STREET, and LOCATION, is also defined.
- The LOCATION group item is further subdivided into two subordinate groups—CITY (Item 9) and STATE (Item 10).

In this example, the following group items are all elementary items: EMPLOYEE-NUMBER, SOCIAL-SECURITY-NUMBER, STREET, CITY, STATE, and ZIP. (All but two are assigned level-number 05; the two exceptions, CITY and STATE, are assigned level-number 07.) If your program accesses the group item ADDRESS, it implicitly accesses STREET, LOCATION, CITY, STATE, and ZIP.

Notice that the level-numbers used in this example are not successive, and that the descriptions of all elementary items include PICTURE clauses. The first entry in this example begins with the word FD, which is a level-indicator that indicates the entire file; this entry is a file-description entry, which must always precede any group of record-description entries in the File Section. (File Description entries are discussed completely in section VIII.)

Figure 4-1.
RECORD DESCRIPTION ENTRY

```
            .
            .
            .
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).
            .
            .
            .
```

In addition, programs may contain special level-numbers that do not actually apply to hierarchical levels. Instead, they indicate special properties of entries in the Data Division. These level-numbers are:

Level-Number	Purpose
66	Specify group or elementary items introduced by a RENAMES clause. This clause permits the re-grouping of data.
77	Specify non-contiguous data-items that are not subdivisions of other items, and are not themselves subdivided. These items are defined in the Working Storage Section and typically reference internal counters and accumulators.
88	Specify condition-names associated with particular values of a conditional variable.

Specific rules for coding all of the above entries appear in the discussion of the Data Division in Section IX.

Data Items

Data-items in a COBOL program are specified and referenced in a very precise way. The various restrictions governing data-items are outlined below.

DATA CLASSES AND CATEGORIES

In COBOL, three general classes of data-items are recognized:

- **Alphabetic**, which may contain letters (A through Z) or spaces, in any combination.
- **Numeric**, which may contain digits (0 through 9) in any combination, optionally preceded by a plus (+) or minus (−) sign. This is the only class of data-item that can be used in arithmetic operations.
- **Alphanumeric**, which may contain any characters from the ASCII Character Set, in any combination.

NOTE

> For complete compatibility with all ANSI COBOL compilers, use only members of the COBOL Character Set.

These three classes are subdivided into five categories:

- **Alphabetic**, which is synonymous with alphabetic class.
- **Numeric**, which is synonymous with numeric class.
- **Alphanumeric**, which may contain any characters from the ASCII Character Set in any combination, not edited by a PICTURE clause.
- **Alphanumeric edited**, which may contain any characters from the ASCII Character Set in any combination, plus editing symbols supplied by a PICTURE clause.
- **Numeric edited**, which may contain any digits (0 through 9), plus editing symbols supplied by a PICTURE clause.

NOTE

> More precise definitions of these categories appear under the discussion of the PICTURE clause in Section IX.

These classes and categories are independent of the external or internal storage formats of the data-items. The relation of classes to categories are summarized in table 4-1. For alphabetic and numeric data-items, the classes and categories are synonymous. For alphanumeric data-items, the relation of class to category depends on the level (group or elementary) of the data-item within the record structure. Every elementary item (except an index data-item) belongs to one of these classes and categories. During program execution, every group item is treated as an alphanumeric item regardless of the class of the elementary items subordinate to that group item.

Table 4-1.
DATA-ITEM CLASSES AND CATEGORIES

LEVEL OF ITEM	CLASS	CATEGORY
Elementary	Alphabetic	Alphabetic
	Numeric	Numeric
	Alphanumeric	Numeric Edited Alphanumeric Edited Alphanumeric
Group	Alphanumeric	Alphabetic Numeric Numeric Edited Alphanumeric Edited Alphanumeric

ALGEBRAIC SIGNS

With numeric data, two types of algebraic signs may be associated: operational signs and editing signs.

OPERATIONAL SIGNS

These signs are associated with signed numeric data-items and signed numeric literals, to indicate their algebraic properties. In regular (default) internal format, they are represented in storage as noted in the discussion of the USAGE clause of the Data Division (section IX). However, you may optionally override this format by explicitly specifying the location of the signs. You do this by using the SIGN clause of the Data Division.

NOTE

> Using the SIGN clause to force operational signs into a representation different from the regular (default) format typically causes the compiler to create less efficient object code.

EDITING SIGNS

These signs typically appear on edited reports, and are used to denote the positive or negative value of data. They are inserted into the data through sign/control symbols in the PICTURE clause.

Data Alignment

In a COBOL program, data is moved from one area of storage (sending data-item) to another (receiving data-item) through use of MOVE, ACCEPT, STRING, UNSTRING, arithmetic, or other statements in the Procedure Division. When aligning the data within the receiving item, the compiler follows specific rules. These rules depend upon the category of that item, as noted below. Examples appear in table 4-2.

1. For data-items in the numeric category, the compiler aligns the data by the decimal point and places it in the receiving data-item. The compiler also truncates excess characters on either end of the sending data-item, and fills unused positions in the receiving data-item with zeros.

   NOTE

   The decimal-point is never actually stored in a numeric data-item; instead, the compiler defines and keeps track of an assumed decimal point that appears in the data-item only when it is read or written or output. Any stored data-item that contains a combination of digits and editing characters such as the decimal point, comma, and so forth, does not belong to the numeric category and cannot be used in arithmetic operations except as a receiving field.

   When your program does not explicitly specify a decimal point for a numeric data-item, the compiler defines an assumed decimal point immediately after the right most digit and aligns the item as described above.

2. For data-items in the numeric-edited category, the compiler aligns the data by decimal point with zero-fill or truncation at either end (as with numeric data-items), except where editing replaces leading zeros with another character.

   NOTE

   The decimal point in a numeric edited data-item, unlike that in a numeric data-item, is actually stored in the item.

3. For data-items in the alphabetic, alphanumeric, and alphanumeric edited categories, the compiler aligns the data at the leftmost character position; it also truncates excess characters at the right of the sending item and fills unused positions at the right of the receiving item with spaces.

   NOTE

   If your program specifies the JUSTIFIED clause for the receiving data-item in the Data Division, the above rules are modified as directed by that clause.

Level-01 (record) and level 77 data-items are always aligned on word boundaries.

In the example in figure 4-2, space characters are represented by deltas (Δ) and assumed decimal-point positions are represented by carets (‸). The receiving data-items for alphabetic and alphanumeric data-items are each 11 positions long. The receiving data-item for the alphanumeric edited data-item is six positions long. The receiving data-items for numeric and numeric edited data-items may each contain up to 18 digits plus a decimal point. (The decimal point is assumed for numeric items and actually present for numeric edited items.) The specifications for the PICTURE format are explained in section VIII.

Table 4-2.
DATA ALIGNMENT

Category	Data To Be Stored	Receiving Item Before Transfer	Receiving Field After Transfer	
			PICTURE format	Content
Alphabetic	ABC ABCDEFGHIJK ABCDEFGHIJKLMN	PQRSTUVWXYZ PQRSTUVWXYZ PQRSTUVWXYZ	A(11) A(11) A(11)	ABC ABCDEFGHIJK ABCDEFGHIJK
Numeric	1‸2 123‸456 12345‸67890	987‸654 987‸654 987‸654	9(3)V9(3) 9(3)V9(3) 9(3)V9(3)	001‸200 123‸456 345‸678
Numeric Edited	1.2 123.456 12345.67890	987.654 987.654 987.654	9(3).9(3) 9(3).9(3) 9(3).9(3)	001.200 123.456 345.678
Alphanumeric Edited	ABCDE	ZZZ/ZZZ	XXX/XXX	ABC/DEΔ
Alphanumeric	A2C A2C?E!G*I5K A2C?E/G*I5K@M7	PQRSTUVW123 PQRSTUVW123 PQRSTUVW123	X(11) X(11) X(11)	A2CΔΔΔΔΔΔΔΔ A2C?E!G*I5K A2C?E!G*I5K

Uniqueness of Reference

To ensure that the basic elements defined in the various program divisions can be properly referenced in the Procedure Division, COBOL places several restrictions on some of these elements. These restrictions cover qualifiers, subscripts, indexes, and identifiers as they apply to data-names, condition-names, paragraph-names, and text-names.

Qualifiers

Each data-name, condition-name, paragraph-name, and text-name must be unique within the program in which it appears. Such a name is unique if either of these two conditions apply:

1. No other name in the program has the same spelling and hyphenation.

2. The If the same name is used for two different elements in a program, it must be made unique through qualification. For instance, if two paragraphs are both identified by the name PAR-MSG, they must be members of different sections, called perhaps SEC-1 and SEC-2. In the Procedure Division, you might then reference one of these paragraphs by specifying:

```
PAR-MSG OF SEC-1.
```

In a hierarchy of names such as this, the higher-level names are called qualifiers. Specifically, a qualifier is one of the following:

- A data-name used in a reference together with another data-name or with a condition-name at a lower-level in the same hierarchy.
- A section-name used in a reference together with a paragraph-name specified in that section.
- A library-name used in a reference together with a text-name associated with that library.

In a program, you qualify a data-name, condition-name, paragraph- name, or text-name by entering one or more phrases composed of: the name to be qualified, followed by the reserved word IN or OF, followed by a qualifier. Three formats are possible, depending on the type of name:

Format 1

```
{ data-name-1    }  [ { OF }  data-name-2 ] ...
{ condition-name }  [ { IN }              ]
```

Format 2

```
paragraph-name  [ { OF }  section-name ]
                [ { IN }               ]
```

Format 3

```
text-name  [ { OF }  library-name ]
           [ { IN }               ]
```

NOTE

In these format descriptions, the keywords OF and IN are logically equivalent and may be used interchangeably.

In all cases, you must use sufficient qualifiers to make the name unique. It is not always necessary, however, to mention all levels in a particular hierarchy.

Within the Data Division, all data-names used as qualifiers must be associated with a level-indicator or level-number. Thus, you cannot specify two identical data-names as subordinate entries in a group item unless you can make them unique through qualification. In the qualification hierarchy, names associated with a level indicator are the most significant; names associated with level number 01 are the next most significant; names associated with level numbers 02 through 49 are then ranked in descending order of significance.

In the Procedure Division, two identical paragraph-names must not appear in the same section. A section-name is the highest and only qualifier for a paragraph-name.

NOTE

> The most significant name in any hierarchy must be unique and cannot be qualified.

Subscripted or indexed data-names and conditional variables may be made unique through qualification. The name of a conditional variable can be used as a qualifier for any of its condition-names. Regardless of the available qualification, no name can be both a data-name and a procedure-name.

In using qualification, the following specific rules apply:

- For all names:

Each qualifier must belong to a successively higher level, and fall within the same hierarchy, as the name it qualifies.

You may not use the same name at two or more levels in the same hierarchy.

If more than one combination of qualifiers ensures uniqueness, you may use any of these combinations to reference the name.

If a name does not require qualification, you may still qualify it.

- For data-names and condition-names:

If you assign a data-name or condition-name to more than one data-item, you must qualify this name each time it is referenced in the Environment, Data, and Procedure Divisions.

NOTE

> This rule does not apply to the REDEFINES clause of the Data Division, where qualification is unnecessary and, in fact, prohibited.

- For data-names:

You may not subscript a data-name that is used as a qualifier.

You may not specify, as the complete set of qualifiers for one data-name, a partial set of qualifiers used for another data-name.

- For paragraph-names:

You may not duplicate a paragraph-name within a section.

When a paragraph-name is qualified by a section-name, you may not use the reserved word SECTION in the qualification.

In any section, you need not qualify a paragraph-name that is referenced in the same section.

- For text-names:

If more than one COBOL library is used during compilation, you must qualify all text-names each time they are referenced in the program.

To illustrate qualification, the name DATA-GRAY is duplicated in the program in figure 4-2, where it actually refers to two different data-items. In each case, the name can be qualified as DATA-GRAY OF DATA-BLACK or DATA-GRAY OF DATA-GREEN.

Figure 4-2.
QUALIFICATION OF DATA-NAMES

```
          .
          .
          .
01 RECORD-1.
   03 DATA-BLACK.
      05 DATA-GRAY.
         10 DATA-BLUE.
         10 DATA-YELLOW.
      05 DATA-BROWN.
   03 DATA-WHITE.
      05 DATA-GREEN.
         10 DATA-GRAY.
          .
          .
          .
```

Subscripts

Elements in a table of like elements can be uniquely referenced through subscripts. A table is a list of logically/consecutive data-items. A subscript is an integer that corresponds to a specific element in the table. You may only use subscripts for elements that have not been assigned individual data-names. The lowest possible subscript value is 1, which identifies the first element in the table. The next ascending values 2, 3, 4 . . . point to the second, third, and fourth elements, and so forth. The highest possible value is specified by the OCCURS clause of the Data Division.

The table is identified by a table element data-name or referenced by a condition-name. Individual elements in the table are identified by one, two, or three subscripts. The subscript, or set of subscripts, is enclosed in a pair of balanced parentheses following the table element data-name. (Any data-name written in this format is called a subscripted data-name.) When two or three subscripts are used, they are written in order of successively decreasing inclusiveness.

Subscripted Data-Name Format

$$\left\{\begin{array}{l}\textit{data-name}\\ \textit{condition-name}\end{array}\right\}\left(\textit{subscript-1}\left[, \textit{subscript-2}\ [\ , \textit{subscript-3}\ ]\right]\right)$$

The subscript can be represented by a numeric literal or a data-name. In either case, it must represent an integer, optionally preceded by a plus (+) sign. If a data-name is used, it must specify an elementary item; the data-name may be qualified but may not itself be subscripted.

The table-element data-name must be separated from the left parenthesis by a space. When more than one subscript appears, each must be separated from its predecessor by a comma followed by a space.

Examples of subscripted data-items are:

```
DAY (1)

DATE (1, 2)

BENCHMARK (ALPHA, BETA, GAMMA)

TANDY (20, FILEZ-01, +3)
```

Indexes

Elements in a table can be referenced through indexes as well as subscripts. An index is a special register containing a binary value that is the byte offset from the beginning of the table to which it is associated.

The index is defined for the table and assigned an index-name through the INDEXED BY phrase in the table definition in the Data Division. In the Procedure Division, you use the index-name to reference the index. Before you may use the index as a table-reference, however, you must assign the index an initial value. You do this by using the SET, SEARCH ALL, or PERFORM statement.

Two indexing techniques are available:

- Direct indexing, where the element desired is specified by the contents of the index.
- Relative indexing, where the element desired is specified by the contents of the index plus or minus a specific value.

Direct indexing is specified by using an index-name in the form of a subscript after the table element data-name. For example:

```
DATA-1 (INDEX-A)
```

Relative indexing is specified by using the index-name, followed by a plus or minus sign, followed by an unsigned integer specified as a numeric literal, all enclosed in balanced parentheses and following the table element data-name.

The string comprised of the table-element data-name and the index is called the index data-name.

When two or more index-names are used, they are written in order of successively decreasing inclusiveness in the data organization. Each is separated from its predecessor by a comma followed by a space.

The format for referencing an index in the Procedure Division is:

$$\left\{\begin{array}{l}\textit{data-name}\\ \textit{condition-name}\end{array}\right\}\left(\left\{\begin{array}{l}\textit{index-name-1}\ \left[\left\{\begin{array}{c}+\\-\end{array}\right\}\textit{literal-2}\right]\\ \textit{literal-1}\end{array}\right\}\right.$$
$$\left.\left[,\left\{\begin{array}{l}\textit{index-name-2}\ \left[\left\{\begin{array}{c}+\\-\end{array}\right\}\textit{literal-4}\right]\\ \textit{literal-3}\end{array}\right\}\left[,\left\{\begin{array}{l}\textit{index-name-3}\ \left[\left\{\begin{array}{c}+\\-\end{array}\right\}\textit{literal-6}\right]\\ \textit{literal-5}\end{array}\right\}\right]\right]\right)$$

In this format, the data-name or condition-name must be separated from the left-parenthesis by a space. Also, the plus or minus sign must be bounded by a space on either side.

The compiler determines the sequential location of the element in the table by incrementing (for the plus sign) or decrementing (for the minus sign) the value in the index by the value of the literal.

When a program executes a statement that refers to an indexed table element through direct indexing, the value of the index must not be either less than one or greater than the highest-numbered element permitted in the table. When a program uses relative indexing, this same restriction applies to the value resulting from relative indexing.

The following index data-names demonstrate direct indexing:

```
TABLE-1 (ALPHA)

TABLE-2 (20, 10)

TAB-M (BETA, GAMMA)

LIST-2 (10, 5, 2)

MAXVAL (EPSILON, 30, ZETA)
```

In the first of the above examples, if TABLE-1 is the index table-name and the value of ALPHA is 15, the program accesses the 15th element in the table. In the second of these examples, the program accesses element 20, 10 in a two dimensional table named TABLE-2.

The next examples illustrate relative indexing:

```
TABLE-1 (BETA + 1)

TABLE-2 (THETA - 1, MU)

ARRAY-3 (STORE-1 + 1, STORE-2 + 1, STORE-3 + 1)
```

In the first of the above examples, if TABLE-1 is the index table name and the value of BETA is 15, the program accesses the 16th element. The second example demonstrates that both direct and relative indexing are permitted in the same index data-name.

Identifiers

Data-names that are uniquely identified by qualifiers, subscripts, or indexes, are known collectively as identifiers. Their overall syntax is summarized in two general formats:

Format 1

data-name-1 [{ **OF** / **IN** } *data-name-2*] ... [(*subscript-1* [, *subscript-2*

[, *subscript-3*]])]

Format 2

data-name-1 [{ **OF** / **IN** } *data-name-2*] ... [({ *index-name-1* [{±} *literal-2*] / *literal-1* }

[, { *index-name-2* [{±} *literal-4*] / *literal-3* } [, { *index-name-3* [{±} *literal-6*] / *literal-5* }]])]

Certain restrictions govern the use of identifiers:

1. In formats 1 and 2 above, you may use only positive numeric integers for the following literals: literal-1, literal-3, and literal-5. Additionally, you may use only unsigned numeric integers for the following literals: literal-2, literal-4, and literal-6.

2. You may not index or subscript a data-name that itself is used as an index or subscript, or as a qualifier.

3. Where subscripting is prohibited, indexing is also prohibited.

4. You may alter the value of an index only by using the SET, SEARCH, and PERFORM statements in the Procedure Division.

5. You may store values referenced by index-names into data-names that are described by the USAGE IS INDEX clause of the Data Division.

Condition-Names

Condition-names made unique through qualification, indexing, or subscripting have the same overall syntax as Formats 1 and 2 for identifiers, above. In these formats, however, the user-defined word *condition-name-1* replaces *data-name-1.*

The restrictions that apply to the combined use of qualification, subscripting, and indexing of identifiers also apply to condition-names. In addition, these further restrictions apply:

1. If a condition-name is made unique through qualification, you may use as the first qualifier either the hierarchy of names associated with the related conditional variable or the conditional variable itself.

2. If references to a conditional variable require indexing or subscripting, you must use the same indexing or subscripting for references to any condition name associated with that variable.

NOTE

> In the format descriptions appearing throughout this manual, condition-name refers to a condition-name that may be qualified, indexed, or subscripted as necessary.

CODING SOURCE PROGRAMS

SECTION V

To code a COBOL source program and enter it into the system, one may employ either of two methods:

- Create and enter the source program at an on-line terminal, using the HP 3000 Editor (EDIT/3000). Then save the program in an ASCII file on disc.
- Write the program on a COBOL coding form. Then, have it keypunched onto Hollerith cards and entered into the system through a card reader.

Using EDIT/3000

On the HP 3000, most COBOL programmers write their programs on-line using EDIT/3000. This is generally the most convenient method, for it allows them to bypass both coding sheets and punched cards. To use this method, follow the directions shown below and keyed to the example in figure 5-1. (In this and later examples of interactive processing, user input is underlined to distinguish it from system output.)

1. At your terminal, enter an MPE :HELLO command to begin an on-line dialog (session) with the system.
2. Enter the MPE :EDITOR command to access EDIT/3000. (See the EDIT/3000 Reference Manual for further information about using the HP 3000 Editor.)
3. Enter an EDIT/3000 SET command to place the Editor in COBOL mode. In this mode, the source code you enter is written to a disc file of appropriate format for input to the COBOL/3000 compiler. Furthermore, as you enter each line of code, EDIT/3000 increments its line number by .1 and also automatically supplies, in columns 1 through 6 of each record written to the disc file, a COBOL sequence number. The compiler creates these sequence numbers by multiplying the Editor line numbers by 1000. Thus, the compiler would provide the sequence number 002500 for the record you enter on line 2.5 The sequence numbers appear on source-code listings produced by the compiler, but not on listings displayed by EDIT/3000. (Further information about sequence numbers appears later in this section.) The compiler also assumes, by default, a line-length of 80 characters.

   NOTE

   A COBOL program created with EDIT/3000 never has blank sequence-number fields. But a program copied from cards or tape may contain some lines without sequence numbers. If you wish to edit such a program, you must first modify it as directed in USING FILES.

4. Enter an EDIT/3000 ADD command to create a temporary ASCII file on disc to which you may add source records, and to initiate the add operation.
5. If you decide to use any of the compiler options other than those supplied by default, enter a $CONTROL compiler subsystem command to specify those options. Begin this command in column 7. In figure 5-2, the $CONTROL command:

- Initializes the object-code (USL) file before the code is written to it, as requested by the USLINIT parameter.
- Prints a symbol table listing for debugging purposes, as requested by the MAP parameter.

More information about the $CONTROL command appears in Appendix A.

NOTE

In COBOL mode, EDIT/3000 always skips to column 7 as the left margin for source-text entry. Columns 1 through 6 are reserved for sequence numbers.

6. Enter your COBOL program, beginning with the Identification Division header and terminating with the STOP RUN statement. Begin the division header in column 8. Begin all other entries in the columns specified later in this section, under CODING RULES.
7. Enter a double slash-mark (//) or Control Y (Y^c) to stop the addition operation.
8. If you wish to check the source code for errors, enter an EDIT/3000 LIST command to display the source program on your terminal screen.
9. Enter an EDIT/3000 KEEP command to save the file containing your program as a permanent file. In figure 5-2, the permanent file is named SOURCEFL.

NOTE

Do not use the UNNUMBERED parameter with the KEEP command, otherwise, the sequence numbers are omitted and the source text is written to the wrong columns.

10. If you wish a hard-copy listing of your program, enter an EDIT/3000 LIST command to direct the listing to the line printer.
11. Enter an EDIT/3000 END command to terminate communication with EDIT/3000. You may next perform other functions such as compiling, preparing, and editing your program, or you may log off (by entering the MPE :BYE command).

NOTE

For information on other editing functions, see the EDIT/3000 Reference Manual and USING COBOL. Such functions may include:

- Correcting mistakes in your source code.
- Inserting, changing, or deleting entries.
- Setting tabs and printing column-number headings on the terminal screen.

Figure 5-1.

ENTERING A COBOL SOURCE PROGRAM WITH EDIT/3000

```
:HELLO JOANNE.MARKDEPT
HP3000 / MPE III B.00.02. WED, APR 30, 1980, 4:19 PM
:EDITOR
HP32201A.7.04 EDIT/3000 WED, APR 30, 1980, 4:19 PM
(C) HEWLETT-PACKARD CO. 1978
/SET FORMAT=COBOL
/ADD
   1        $CONTROL USLINIT
   1.1      $CONTROL MAP
            IDENTIFICATION DIVISION.
          .
          .
          .
   6.7      STOP RUN.
   6.8  //
/LIST ALL
   1        $CONTROL USLINIT
   1.1      $CONTROL MAP
            IDENTIFICATION DIVISION.
          .
          .
          .
   6.7      STOP RUN.
/KEEP SOURCEFL
/LIST ALL, OFFLINE
*** OFF LINE LISTING BEGUN. ***
/EXIT
END OF SUBSYSTEM
          .
          .
          .
```

Using a Coding Form

Those programmers who do not code their programs on-line generally write them on a COBOL coding form or other convenient medium. The coding form, introduced in section I, is designed with 80 columns to facilitate transfer onto punched cards. A blank coding form appears in Appendix F.

The coding form contains a heading in which you may enter your name, the date, the program name, and the current page number of the coding form. The remainder of the form is for the program itself, which you must enter according to the rules described below.

Coding Rules

Whether entering your program through EDIT/3000 or onto a coding form, follow these rules carefully:

SEQUENCE NUMBERS (COLUMNS 1 THROUGH 6)

Sequence numbers appear in columns 1 through 6 of the source record. They identify the order of the record with respect to other records in your program. When you enter your program through EDIT/3000, the sequence numbers are supplied automatically by that subsystem. When you write the program on a coding form, however, these numbers are optional. When you choose to use them, you enter them in columns 1 through 6 of the coding sheet. These numbers may range from 000000 through 999999. They may not contain letters or special characters.

When you compile your program, you may request the compiler to use the sequence numbers to check the sequence of the source statements in the program. You may also request the compiler to renumber these statements. You select these options with the $CONTROL command.

Sequence numbers are especially useful for punched-card input; if the deck is dropped, they enable you to easily place it in proper order again.

Sequence numbers are also useful if you must recompile the program. They allow you to merge new text with the originally compiled text stored on disc, according to sequence number. Thus, you need to enter only additional or changed statements for this compilation.

NOTE

> If you intend to use this feature, increment the sequence numbers by 10 or 100 to allow space for possible new statements.

PROGRAM TEXT (COLUMNS 8 THROUGH 72)

Program text appears in columns 8 through 72. This group of columns is divided into Areas A and B. The coding sheet provides both column and area headings.

Program text appears in columns 8 through 72. This group of columns is divided into Areas A and B. The coding sheet provides both column and area headings. All area A and area B coding conventions presented throughout this manual represent the ANSI standard specifications for COBOL. In many instances the COBOL II/3000 compiler allows variations from these standards. In order to enhance the readability of source programs, and ensure compatibility with standard ANSI COBOL, programmers are encouraged to follow the rules presented herein.

In Area A (columns 8 through 11), you begin all division headings, section-headers, paragraph-headers, paragraph-names, level indicators FD and SD, and level numbers 01 and 77. These entries may, where necessary, be continued into Area B.

In Area B (columns 12 through 72), you enter all other COBOL text. For example, the following elements must appear in Area B: all sentences and procedural statements; data-description entries (including their names), whether or not associated with level indicators and numbers; and all level numbers other than 01 and 77.

CONTINUATION LINES

Any sentence or entry that requires more than one line must be continued in Area B of the next line. Your program may contain any number of continuation lines. When a word or numeric literal is broken from one line to the next, you must enter a hyphen (-) in column 7 of the continuation line. This hyphen indicates that the first non-space character in Area B is part of the word or literal broken on the previous line. When a non-numeric literal is broken from one line to the next, you again must place a hyphen in column 7; furthermore, in this case, you enter a quotation mark before the continuation of the literal. In any case, the continuation of the word or literal may begin anywhere within Area B of the continuation line. All spaces at the end of the continued line are considered part of the end of the word or literal. In any continuation line, area A must always contain only spaces. Blank lines cannot appear between continued and continuation lines.

COMMENT LINES

You may enter explanatory comment lines anywhere within your program, after the Identification Division header. You denote these by entering an asterisk (*) in column 7 and entering the comment in columns 8 through 72. The comment line appears on the source program listing but does not affect compilation or execution. Comment lines may not appear between a continued line and its following continuation line.

You may request the line-printer to eject a page prior to printing a comment, by using a slash-mark (/) in place of the asterisk.

IDENTIFICATION CODE (COLUMNS 73 THROUGH 80)

An optional identification code may appear in columns 73 through 80. You may use this feature for identifying different versions of a program. It also serves as the library name for source statements placed in a COBOL copy-library; all statements in such a library require an identification code. See Section XIV, COBOL LIBRARIES AND THE LIBEDIT PROGRAM, for more information on libraries and their use.

Coding Conventions

To improve the readability of your programs, and simplify debugging, the following coding conventions are recommended:

- Skip lines on the terminal or coding form. This both improves legibility and leaves room for inserting additional records.
- In the Data Division, indent entries of differing levels-the higher the level, the greater the indentation. This also aids in graphically representing data structures.
- Enter only one statement per line.
- Indent all continuation lines.

IDENTIFICATION DIVISION

SECTION VI

Every COBOL II program begins with the Identification Division. This division specifies information that identifies both the source program and related listings produced by the COBOL II/3000 compiler. Among this information, you must always include the name of your program. In addition, you may optionally identify:

- The author of the program.
- The installation where the program is compiled.
- The date that the program is written.
- The date that the program is compiled.
- Any security restrictions governing the program.

IDENTIFICATION DIVISION Format

The Identification Division has the following format:

```
{ ID             }
{                } DIVISION.
{ IDENTIFICATION }

PROGRAM-ID. program-name.

[ AUTHOR. [ comment-entry ] ... ]

[ INSTALLATION. [ comment-entry ] ... ]

[ DATE-WRITTEN. [ comment-entry ] ... ]

[ DATE-COMPILED. [ comment-entry ] ... ]

[ SECURITY. [ comment-entry ] ... ]

[ REMARKS. [ comment-entry ] ... ]
```

In this format, the paragraph headers identify the kind of information that each paragraph contains. Thus, in the PROGRAM-ID paragraph, you specify the name of your program; in the AUTHOR paragraph, you generally enter your own name.

IDENTIFICATION DIVISION SYNTAX RULES

The PROGRAM-ID paragraph is always required, but all other paragraphs are optional. When any optional paragraphs are included, they must always appear in the order shown in the format description.

Begin the division header in Area A of the first line on the terminal screen or coding sheet. Begin each paragraph header in Area A of a new line. In each paragraph, begin the paragraph body (***program-name*** or ***comment-entry***) either on the same line as the paragraph header, or in Area B of a new line following the header. When you must continue a lengthy entry, begin the continuation in Area B of the next available line. The ***program-name*** and each ***comment-entry*** must be terminated by a period followed by a space.

Paragraphs

In the Identification Division, the various paragraphs serve the functions noted below.

PROGRAM-ID PARAGRAPH

This paragraph must appear in every program and must include the program's name. This name identifies your source program and appears on the listings associated with it. It must be a unique name with respect to all program units (COBOL II main program or subroutines) compiled in a particular instance. Thus, if you compile a main program and two subroutines onto the same USL file, each must have a unique name with respect to the others. The name must begin with a letter and cannot contain more than 15 characters, excluding hyphens. Additional characters are truncated and all hyphens are deleted by the compiler. Non-numeric literals are permitted as program names.

DATE-COMPILED PARAGRAPH

When you enter the DATE-COMPILED paragraph in your source program, the compiler prints the current date and time on the second line of this paragraph as it appears on the source program listing. Generally, you include only the paragraph header (and do not specify a body) when you enter the source program. On the source listing, the date and time appears in this format:

```
            .
            .
            .
   Source-code entry
001400 DATE-COMPILED. FRI, JUL 28, 1980,  5:27 PM
                      Date and time supplied by compiler
            .
            .
            .
```

No additional data, either on the same line as the paragraph header or on subsequent lines, is printed on the source listing.

OTHER PARAGRAPHS

In the other paragraphs of the Identification Division, all optional, the paragraph bodies are treated as comments. Thus, you may enter in any of them any information you wish. For example, you may use the AUTHOR paragraph for data other than someone's name.

To continue any of these comment-entries onto two or more lines, simply enter the information you wish in columns 8 through 72 of the necessary lines, but do not, in this case, enter the hyphen continuation indicator in column 7.

EXAMPLE

An example of a complete Identification Division, showing all required and optional paragraphs, appears below.

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  EXAMPLE-ID-SECTION.
AUTHOR. MANUAL-J-WRITER.
INSTALLATION. GSD-CUST-TRAINING-AND-DOC-GROUP.
DATE-WRITTEN. 07/11/80.
DATE-COMPILED. NOT-YET.
SECURITY.  NONE.
```

ENVIRONMENT DIVISION

SECTION VII

In every COBOL II program, the Identification Division is followed by the Environment Division. The Environment Division allows you to define those aspects of your data processing application that depend on the physical characteristics of your processing environment. This division consists of two sections:

- **Configuration Section**, for specifying the hardware characteristics of the system on which you will compile and run your program.
- **Input-Output Section**, for specifying the data files used by your program, and various input/output control elements.

Each of these sections may contain several paragraphs.

DIVISION Format

The Environment Divison has the following general format:

```
ENVIRONMENT DIVISION.
[ CONFIGURATION SECTION.
  SOURCE-COMPUTER. source-computer-entry
  OBJECT-COMPUTER. object-computer-entry
  [ SPECIAL-NAMES. special-names-entry ] ]
[ INPUT-OUTPUT SECTION.
  FILE-CONTROL. {file-control-entry} ...
  [ I-O-CONTROL. input-output-control-entry ] ]
```

DIVISION SYNTAX RULES

The Configuration and Input-Output sections are both optional. When included, however, they must appear in the order shown in the format description.

NOTE

Although the Configuration Section is required in ANSI '74 COBOL, it is optional in COBOL II/3000.

Within each section, optional paragraphs are enclosed in brackets; required paragraphs are not.

Begin the division header in Area A of the first line on the terminal screen or coding sheet. Begin each section or paragraph header in Area A of a new line. In each paragraph, begin the paragraph body either on the same line as the paragraph header, or in Area B of a new line following the header; enter the remainder of the paragraph on successive lines within Area B.

CONFIGURATION SECTION

The Configuration Section may include the following paragraphs:

- **SOURCE-COMPUTER** paragraph, for defining the characteristics of the computer system on which your COBOL II source program will be compiled.
- **OBJECT-COMPUTER** paragraph, for specifying the characteristics of the computer system on which the resulting object program will be run.
- **SPECIAL-NAMES** paragraph, for specifying the relationship of special fraction-names to mnemonic names appearing in the source program.

Although these paragraphs are optional, the use of one requires the use of others. That is, if you specify the SOURCE-COMPUTER paragraph, you must specify the OBJECT-COMPUTER paragraph as well. The opposite is also true. Furthermore, if you specify the SPECIAL-NAMES paragraph, then both the SOURCE-COMPUTER and the OBJECT-COMPUTER paragraphs must be specified.

CONFIGURATION SECTION Format

The Configuration Section of the Environment Division has the following format.

```
CONFIGURATION SECTION.

SOURCE-COMPUTER.  computer-name  [ WITH DEBUGGING MODE ] .

OBJECT-COMPUTER.  computer-name

  [                              { WORDS      } ]
  [ , MEMORY SIZE  integer       { CHARACTERS } ]
  [                              { MODULES    } ]

  [, PROGRAM COLLATING SEQUENCE IS alphabet-name ]

  [, SEGMENT-LIMIT IS segment-number ] .

[ SPECIAL-NAMES.
                                                                                   ]
   [ , function-name                                                            ]  ]
   [ { IS mnemonic-name  [    , ON STATUS IS condition-name-1  ]                 } ]  ]
   [ {                   [ [, OFF STATUS IS condition-name-2 ] ]                 } ]  ]
   [ {                                                                         } ]  ]
   [ { IS mnemonic-name  [    , OFF STATUS IS condition-name-2 ]               } ] ...]
   [ {                   [ [, ON STATUS IS condition-name-1 ]  ]               } ]  ]
   [ {                                                                         } ]  ]
   [ { ON STATUS IS condition-name-1 [, OFF STATUS IS condition-name-2 ]       } ]  ]
   [ { OFF STATUS IS condition-name-2 [, ON STATUS IS condition-name-1 ]       } ]  ]
                                                                                      ]
   [                     { STANDARD-1                                          } ]    ]
   [                     { NATIVE                                              } ]    ]
   [ , alphabet-name IS  { EBCDIC                                              } ] ...]
   [                     {                                                     } ]    ]
   [                     {           [ { { THROUGH } literal-2 } ]             } ]    ]
   [                     { literal-1 [ { { THRU    }           } ]             } ]    ]
   [                     {           [ { ALSO literal-3        } ]             } ]    ]
   [                     {           [ { [, ALSO literal-4 ] ... } ]           } ]    ]
   [                     {                                                     } ]    ]
   [                     {  [             [ { { THROUGH } literal-6 } ] ]      } ]    ]
   [                     {  [ literal-5   [ { { THRU    }           } ] ] ...  } ]    ]
   [                     {  [             [ { ALSO literal-7        } ] ]      } ]    ]
   [                     {  [             [ { [, ALSO literal-8 ] ... } ] ]    } ]    ]
                                                                                      ]
  [, CURRENCY SIGN IS literal-9 ]                                                     ]
                                                                                      ]
  [, DECIMAL-POINT IS COMMA ] .                                                       ]
```

Each paragraph of this section is discussed on the following pages.

SOURCE-COMPUTER PARAGRAPH

COBOL II/3000 does not require a SOURCE-COMPUTER paragraph. In ANSI '74 COBOL, the SOURCE-COMPUTER paragraph denotes the computer system on which you plan to compile your source program, and whether or not to set the debugging mode switch on at compile time. When used, this paragraph has the following syntax.

SOURCE-COMPUTER Paragraph Format

SOURCE-COMPUTER. ***computer-name*** **[WITH DEBUGGING MODE] .**

Where computer-name is any valid user-defined COBOL word. That is, any combination of alphanumeric characters and hyphens you choose, with the restriction that the first character must be alphabetic, and that there must be no blanks between the first and last characters.

The DEBUGGING MODE clause, if specified, is treated as a comment by this compiler, since the DEBUG module of ANSI COBOL '74 was not implemented in this release of COBOL II/3000.

COBOL II/3000 assumes that all programs will be compiled on an HP computer system. Therefore, it does not require this paragraph, unless you wish to use the OBJECT-COMPUTER paragraph. If you specify a *computer-name* in the SOURCE-COMPUTER paragraph, the compiler treats this name as a comment.

OBJECT-COMPUTER PARAGRAPH

In ANSI COBOL, the OBJECT-COMPUTER paragraph denotes the computer system on which the object program is executed. Since COBOL II/3000 assumes that all COBOL programs will be executed on an HP computer system, this paragraph is optional. However, if you wish to specify the SOURCE-COMPUTER paragraph, or the SPECIAL-NAMES paragraph in your program, the OBJECT-COMPUTER paragraph must be used.

The only clause in the OBJECT-COMPUTER paragraph which is not treated as a comment is the Program Collating Sequence clause.

OBJECT-COMPUTER Paragraph Format

```
OBJECT-COMPUTER.  computer-name

  [                         { WORDS      } ]
  [ , MEMORY SIZE integer   { CHARACTERS } ]
  [                         { MODULES    } ]

  [, PROGRAM COLLATING SEQUENCE IS alphabet-name ]

  [, SEGMENT-LIMIT IS segment-number ] .
```

Where

computer-name	is any combination of alphanumeric characters and hyphens you choose, with the restriction that the first must be allphabetic, and that there must be no blanks between the first and the last characters in the name.
integer	is any positive integer.
alphabet-name	is any name you choose, with the same rules and restrictions as *computer-name* above. This name can be used in the alphabet clause of the SPECIAL-NAMES paragraph.
segment-number	is any non-negative integer in the range 1 to 49.

MEMORY-SIZE Clause

In ANSI '74 COBOL, this clause specifies the amount of main memory required by your program. In COBOL II/3000, however, memory is allocated automatically through MPE. Thus, any entry in this clause is treated as a comment.

PROGRAM COLLATING SEQUENCE Clause

On an HP 3000 computing system, the following operations are performed on the basis of the ASCII collating sequence:

- Determining the truth value of non-numeric comparisons explicitly specified in relation or condition-name conditions.
- Using non-numeric sort or merge keys (unless the COLLATING SEQUENCE phrase of the respective SORT or MERGE statement is specified in the Procedure Division, and the alphabet name used in it specifies a non-ASCII collating sequence).

The Collating Sequence clause can be used in relation with the SPECIAL-NAMES paragraph to define a different collating sequence to be used in these operations.

That is, in the SPECIAL-NAMES paragraph, you can relate *alphabet-name* to the specific collating sequence desired.

Note that if you specify an *alphabet-name* in the Program Collating Sequence clause, but do not use it in the SPECIAL-NAMES paragraph, you cannot reference it anywhere else in your program.

An example of the Collating Sequence clause is shown under the SPECIAL-NAMES paragraph later in this section.

The program collating sequence clause applies only to the program in which it appears. If you omit this clause, the ASCII collating sequence is used.

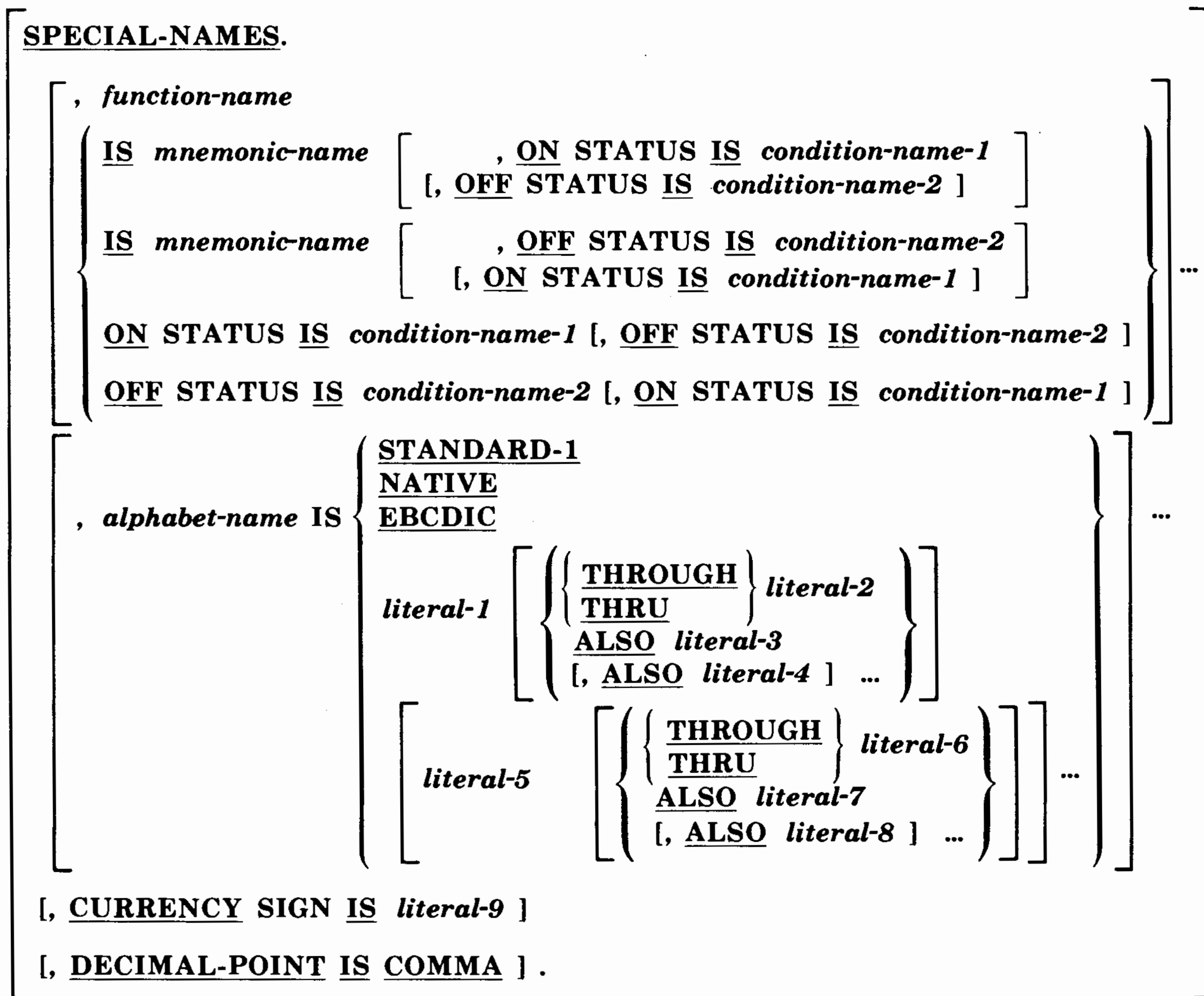
SEGMENT-LIMIT Clause

In ANSI COBOL '74, the Segment-Limit clause is used to define the number of permanent segments in a COBOL program. However, since the concept of a permanent segment has no meaning on an HP computer system, this clause, if specified, is treated as a comment.

SPECIAL-NAMES PARAGRAPH

The SPECIAL-NAMES paragraph allows you to relate certain COBOL supplied functions to mnemonic names; then, by specifying these mnemonic names in your program, you may invoke these functions. It also allows you to relate the alphabet-name specified in the program collating sequence clause to a particular collating sequence or character set, and to define other collating sequences with different alphabet names.

SPECIAL-NAMES Paragraph Format



Where

function-name	is a COBOL reserved word having a specific meaning. All of these names are shown, and their meanings described, in table 7-1 on the following pages.
mnemonic-name	is a name chosen by you to represent *function-name* within your program. This name can be any valid user-defined COBOL word.
condition-name-1 and ***condition-name-2***	are each any valid user-defined COBOL words.
alphabet-name	is either the name (if any) specified in the Program Collating Sequence clause of the OBJECT- COMPUTER paragraph, or any valid user-defined COBOL word.
literal-1 through ***literal-8***	are each alphabetic or numeric literals. If any such literal is a numeric literal, it must be a positive integer from the range 1 to 256. If any non-numeric literal is used in a THROUGH or ALSO phrase, it must be only one character long. Note that no ASCII character can be specified more than one time as a literal in any given alphabet-name clause.
STANDARD-1	represents the ASCII collating sequence.
NATIVE	is currently defined as representing the ASCII collating sequence. However, it may be changed to represent another character set (such as, for example, the KATAKANA character set) at a later date.
EBCDIC	specifies that the EBCDIC collating sequence is to be used. Note that this does not enable any conversion of data. It only allows data to be (for example) sorted or merged according to the EBCDIC collating sequence.
literal-9	is a single character, chosen from a specific set. This set is shown later in this section, when the Currency Sign clause is discussed.
THROUGH and **THRU**	are equivalent, and may be used interchangably.

Each clause of the SPECIAL-NAMES paragraph is discussed on the following pages.

FUNCTION-NAME CLAUSE

In the Function-Name clause, you relate various COBOL functions to mnemonic names used in your program. You do this by equating the specific function-name to the desired mnemonic name.

FUNCTION-NAME Clause Format

```
[ , function-name
  { IS mnemonic-name [    , ON STATUS IS condition-name-1   ]
                     [ [, OFF STATUS IS condition-name-2 ]  ]
    IS mnemonic-name [    , OFF STATUS IS condition-name-2  ]
                     [ [, ON STATUS IS condition-name-1 ]   ]        } ...
    ON STATUS IS condition-name-1 [, OFF STATUS IS condition-name-2 ]
    OFF STATUS IS condition-name-2 [, ON STATUS IS condition-name-1 ] } ]
```

Where

function-name	is a COBOL reserved word having a specific meaning. All of these names are shown, and their meanings described, in table 7-1 on the following pages.
mnemonic-name	is a name chosen by you to represent *function-name* within your program. This name can be any valid user-defined COBOL word.
condition-name-1 and *condition-name-2*	are each any valid user-defined COBOL words.

The function names and corresponding functions are listed in table 7-1. All functions except the software switches (SW0 through SW9) and the CONDITION-CODE function may be referenced by using the assigned *mnemonic-name* in the ACCEPT, DISPLAY, SEND, or WRITE statements of the Procedure Division. As an example, in the Environment Division in figure 7-1, the function names SYSIN and SYSOUT (referencing the terminal for input and output, respectively, during an interactive session, are both related to the *mnemonic-name* USER-TERMINAL. Then, in the Procedure Division, USER-TERMINAL is used to reference the source of input and the destination of output. (The ACCEPT statement reads the input, and the DISPLAY statement writes the output.)

The CONDITION-CODE function is used in relation to the special MPE intrinsic relation condition. See Section X, RELATION CONDITIONS, for more information.

Table 7-1.
COBOL II FUNCTION NAMES AND FUNCTIONS

FUNCTION NAME	FUNCTION
TOP	When included in the ADVANCING clause of the WRITE statement, the mnemonic name assigned to TOP causes the line printer to perform a page eject. This is equivalent to specifying PAGE, which uses an %61 in the FWRITE intrinsic.
NOSPACE CONTROL	Alters the effect of COBOL II WRITE statements that direct output to line-printer files. This function prints the full data record (using single spacing) rather than using the first character of the record for carriage-control and suppressing the printing of that character. COBOL II implements this change by re-setting the control parameter of the MPE FWRITE intrinsic, implicitly called by the WRITE statement, from 1 to 0.
SYSIN	Refers to the MPE standard input device, indicated by the MPE name $STDIN. In an interactive session, this is your terminal. In a batch job, it is either the card reader or operator's console.
SYSOUT	Refers to the MPE standard output device, indicated by the MPE name $STDLIST. In an interactive session, this is your terminal. In a batch job, it is the line printer.
CONSOLE	Refers to the operator's console.
C01 through C12	Used in the ADVANCING clause of the WRITE statement for sequential files. Each directs the line printer to skip to a particular channel (1 through 12) on the carriage-control tape. See Section XI, the WRITE statement, for details.
SW0 through SW9	Refer to software switches associated with *condition-names*.
CONDITION-CODE	Refers to condition codes returned by MPE intrinsics when they have been called through the CALL statement.

Software Switches

Each of the ten software switches (SW0 through SW9) used in a given program must have at least one condition-name associated with it.

A *condition-name* is the only means by which you can reference one of these switches. Thus, although you can specify a *mnemonic-name* for a switch in the Function-Name clause, the *mnemonic-name* is treated as a comment by the compiler.

To illustrate this,

```
                    .
                    .
                    .
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. HP3000SIII.
OBJECT-COMPUTER. HP3000SIII.
SPECIAL-NAMES.
    SW0 IS SORT-SWITCH, ON STATUS IS SORT-ON.
               .

PROCEDURE DIVISION.
                    .
                    .
                    .
    IF SORT-ON
       THEN CALL "SORTER"
       ELSE NEXT SENTENCE.
                    .
                    .
                    .
```

In the above example, the *mnemonic-name* associated with SW0 is treated as a comment; in this case, it serves the sole purpose of documenting the use of SW0 in your program. The actual reference to SW0 is done in the IF statement in the Procedure Division, where SORT-ON is tested in a condition-name condition. (Section X has information on condition-name conditions.) If SORT-ON is in an "on" condition, the subprogram, SORTER, is called. If it is in an "off" condition, the next executable sentence is executed.

SETTING SOFTWARE SWITCHES

Software switches are always considered to be "off" during the execution of a program unless you turn them on before execution begins.

To accomplish this, you can use the PARM= parameter of the MPE RUN command that executes your program. This parameter is assigned an octal value corresponding to a single bit in a 16-bit word.

This bit is the actual switch corresponding to one of the software switches, depending upon its position in the word. Thus, to set SW2 on, you can assign the octal value, %20000, to the PARM= parameter. This sets bit 2 of the switch word to a value of 1, which is interpreted by the COBOL II program as "on".

To illustrate:

Initial switch word:

```
Bit 0                                          Bit 15
↓                                              ↓
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
```

(All switches are initially "off".)

`:RUN MYPROG;PARM=%20000`

Switch word following the above `:RUN` command:

```
Bit 0  Bit 2                                   Bit 15
↓      ↓                                       ↓
0  0  1  0  0  0  0  0  0  0  0  0  0  0  0  0

        %  0  2  0  0  0  0
```

(SW2 is "on".)

To set several switches to "on", you must assign the appropriate value to the PARM= parameter. For example, to set switches SW0, SW3, and SW5 to "on", assign the value, %112000 to PARM=.

The following illustrates this.

Initial switch word:

```
Bit 0                                          Bit 15
↓                                              ↓
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
```

(All switches are initially "off".)

`:RUN MYPROG;PARM=%112000`

Switch word following the above `:RUN` command:

```
Bit 0     Bit 3  Bit 5                         Bit 15
↓         ↓      ↓                             ↓
1  0  0  1  0  1  0  0  0  0  0  0  0  0  0  0

        %  1  1  2  0  0  0
```

A final note about software switches: The minimum and maximum values you can use for the PARM= parameter are %100 (to set switch SW9 "on"), and %177700 (to set all switches, from SW0 to SW9, "on). Note that the right most set of six bits are never used. If a value is placed into them, thus turning them "on", they are ignored.

LINE PRINTER FUNCTIONS

The TOP, NOSPACE CONTROL, and C01 through C12 switches are all related to line printer control. Since there is no way for a COBOL II program to explicitly check for the condition of a line printer, you can not use condition-names with these functions. You must, however, specify mnemonic-names for these functions if you intend to reference them later in your program.

To illustrate:

```
                          .
                          .
                          .
SOURCE-COMPUTER.   HP3000SIII.
OBJECT-COMPUTER.   HP3000SIII.
SPECIAL-NAMES.
     TOP IS TOP-OF-FORM, C09 IS TO-END-OF-FORM.
                          .
                          .
                          .
PROCEDURE DIVISION.
                          .
                          .
                          .
    WRITE REC-OUT FROM FOOT-NOTE
       AFTER ADVANCING TO-END-OF-FORM.
                          .
                          .
                          .
    WRITE REC-OUT FROM TITLE
       AFTER ADVANCING TOP-OF-FORM.
                          .
                          .
                          .
```

CONDITION-CODE FUNCTION

The CONDITION-CODE function allows you to check the condition code returned by MPE intrinsics. This function is itself a form of conditional variable, with an integer value. Thus, no condition-name can be associated with it, and a mnemonic-name must be if you wish to check condition codes returned by intrinsics called from your program. See Section X, RELATION CONDITIONS, for an example and more information.

SYSIN, SYSOUT, AND CONSOLE FUNCTIONS

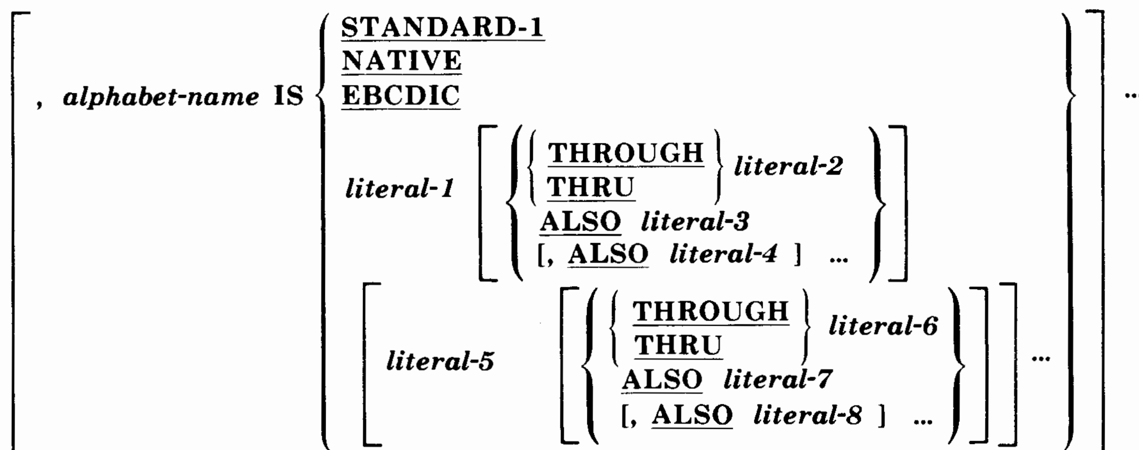
The SYSIN, SYSOUT, and CONSOLE functions are used in the ACCEPT and DISPLAY statements of the Procedure Division. Since these names refer to terminals, line printers, card readers, and the operator's console, you cannot associate condition-names with them. Also, because of the formats of the ACCEPT and DISPLAY statements, you need not specify mnemonic-names for them in the SPECIAL-NAMES paragraph. You may, however, choose to do so. See the ACCEPT and DISPLAY statements in Section XI for more information.

ALPHABET-NAME CLAUSE

The Alphabet-Name clause serves up to three functions. These functions are:

- To define a program collating sequence to be used in sort and merge operations, and non-numeric comparisons (including comparisons used in the INSPECT and EXAMINE statements).
- To define an alphabet-name, and relate this name to either the ASCII or the EBCDIC collating sequence. This *alphabet-name* can later be used in the CODE-SET clause of the Data Division to specify whether records of a sequential file are written in ASCII or EBCDIC code.
- To define an alternate collating sequence separate from the program collating sequence to be used (optionally) in sort and merge operations.

ALPHABET-NAME Clause Format



Where

alphabet-name is either the name (if any) specified in the Program Collating Sequence clause of the OBJECT-COMPUTER paragraph, or any valid user-defined COBOL word.

literal-1 through ***literal-8*** are each alphabetic or numeric literals. If any such literal is a numeric literal, it must be a positive integer from the range 1 to 256. If any non-numeric literal is used in a THROUGH or ALSO phrase, it must be only one character long. Note that no ASCII character can be specified more than one time as a literal in any given alphabet-name clause.

STANDARD-1 represents the ASCII collating sequence.

NATIVE is currently defined as representing the ASCII collating sequence. However, it may be changed to represent another character set (such as, for example, the KATAKANA character set) at a later date.

EBCDIC specifies that the EBCDIC collating sequence is to be used. Note that this does not enable any conversion of data. It only allows data to be (for example) sorted or merged according to the EBCDIC collating sequence.

THROUGH and **THRU** are equivalent, and may be used interchangably.

To define a program collating sequence, you must relate the *alphabet-name* specifed in the Program Collating Sequence clause of the OBJECT-COMPUTER paragraph to either the words NATIVE, STANDARD-1, or EBCDIC, or a list of literals.

Note that since STANDARD-1 refers to the ASCII collating sequence, which is used on all HP computer systems, you need not specify an *alphabet-name* in either the Program Collating Sequence clause or the SPECIAL-NAMES paragraph if this is your choice for a program collating sequence.

STANDARD-1 and NATIVE Phrases

To specify the ASCII collating sequence used on HP computer systems, enter either the STANDARD-1 or NATIVE phrase in the *alphabet-name* clause. As an example:

```
          .
          .
          .
SPECIAL-NAMES.
   ALPHA-NAME IS STANDARD-1.
          .
          .
          .
```

Note, however, that although NATIVE is currently defined as being the ASCII collating sequence, it may be changed in future releases of COBOL II/3000. Thus, to avoid the possibility of having to change your program in the future, you should always use the STANDARD-1 phrase rather than the NATIVE phrase.

EBCDIC Phrase

To specify the EBCDIC collating sequence, enter EBCDIC in the alphabet-name clause:

```
          .
          .
          .
SPECIAL-NAMES.
   ALPHA-NAME IS EBCDIC.
```

LITERAL Phrase

The literal phrase allows you to rearrange the ASCII collating sequence to suit your needs. However, if you specify the literal phrase in an *alphabet-name* clause, you may not reference that name in a CODE-SET clause.

To define your own collating sequence, you must follow the rules listed below.

The collating sequence is defined according to the following rules:

1. If you specify a non-numeric literal, it represents the equivalent character in the ASCII collating sequence. For instance, the literal "A" represents an ASCII A in the following *alphabet-name* clause:

```
          .
          .
          .
ALPHA-NAME IS "A".
          .
          .
          .
```

   If the literal consists of several characters, each character is assigned successive ascending positions in the collating sequence, beginning with the leftmost character. As an example, the *alphabet-name* clause:

```
          .
          .
          .
ALPHA-NAME IS "0A9B8C7D".
```

   Results in the collating sequence:

```
          0
          A
          9
          B
          8
          C
          7
          D
          .
          .
          .
```

2. If the literal is numeric, it represents the ordinal number of the corresponding character in the ASCII collating sequence. Thus, if you specify 65, this represents the 65th character in the ASCII collating sequence — an uppercase A.

```
          .
          .
          .
ALPHA-NAME IS 65
          .
          .
          .
```

3. The order in which the literals appear in the *alphabet-name* clause determines, in ascending sequence, the ordinal numbers of the corresponding characters in the new collating sequence.

4. Any ASCII characters not explicitly specified in the literal phrase assume positions in the new collating sequence numerically higher than any explicitly specified characters. The relative order of the unspecified characters, with respect to each other, is the same as in the ASCII collating sequence.

5. If the THROUGH phrase is used, the new collating sequence will consist of contiguous characters from the ASCII character set, beginning with the value of *literal-1* and ending with the value of *literal-2*. These characters are assigned successive ascending positions in the new character set. As an example, the following alphabet-name clause creates a collating sequence consisting of all uppercase alphabetic characters (letters) from the ASCII collating sequence:

```
          .
          .
          .
ALPHA-NAME IS "A" THROUGH "Z".
          .
          .
          .
```

   With the THROUGH phrase, you may also specify and assign ASCII characters in descending sequence. For example:

```
          .
          .
          .
ALPHA-NAME IS "Z" THROUGH "A".
          .
          .
          .
```

   Note that if a non-numeric literal is used in a THROUGH phrase, it must be only one character long.

6. If the ALSO phrase is used, the ASCII characters specified by *literal-1, literal-3, literal-4,...*, are assigned to the same relative positions in the new collating sequence. If a non-numeric literal is used in an ALSO phrase, it must be only one character long.

7. The character with the highest ordinal position in the new collating sequence may be referenced by the figurative constant, HIGH-VALUE. If more than one character shares the highest ordinal position, the last character specified in the alphabet-name clause is referenced by HIGH-VALUE.

8. The character with the lowest ordinal position in the new collating sequence may be referenced by the figurative constant LOW-VALUE. If more than one character shares the lowest ordinal position, the first character specified in the alphabet-name clause is referenced by LOW-VALUE.

To illustrate the alphabet-name clause:

```
                                 .
                                 .
ENVIRONMENT DIVISION.
SOURCE-COMPUTER.  HP3000SIII.
OBJECT-COMPUTER.  HP3000SIII.
     PROGRAM COLLATING SEQUENCE IS ASCII.
SPECIAL-NAMES.
     ASCII IS STANDARD-1,
     IBMCODE IS EBCDIC,
     SORT-SEQ IS "A" THROUGH "Z"
                 "a" THROUGH "z".
                                 .
                                 .
DATA DIVISION.
FILE SECTION.
FD EBCDICIN;
LABEL RECORDS ARE OMITTED;
CODE-SET IS IBMCODE.
                                 .
                                 .
FD EBCDOUT;
LABEL RECORDS ARE OMITTED;
CODE-SET IS IBMCODE.
                                 .
                                 .
PROCEDURE DIVISION.
                                 .
                                 .
     SORT SFILE ON ASCENDING LEFT-CHAR
          COLLATING SEQUENCE IS SORT-SEQ
          USING INFILE
          GIVING OUTFILE.
                                 .
                                 .
     SORT SFILE ON ASCENDING LEFT-CHAR
          COLLATING SEQUENCE IS IBMCODE
          USING EBCDICIN
          OUTPUT PROCEDURE IS SORT-OUT-PARA THROUGH END-OUT.

                                 .
                                 .
SORT-OUT-PARA.
GET-NEXT-REC.  RETURN SFILE INTO CHK-REC.
     IF DATA-FIELD-2 OF CHK-REC IS NOT ALPHABETIC
        THEN DISPLAY "ERROR IN SORTED RECORD, "
             DISPLAY CHK-REC
             PERFORM GET-RESPONSE-PARA
        ELSE MOVE CHK-REC TO EBCDOUT.
     GO GET-NEXT-REC.
     END-OUT.
```

In the above example, the program collating sequence is specified as STANDARD-1. Since this is the default, the Program Collating Sequence clause and the ASCII IS STANDARD-1 phrase serve only as documentation.

The *alphabet-name*, IBMCODE, is used in two file description entries to indicate that the records of the files, EBCDICIN and EBCDOUT are in EBCDIC code. Thus, when the records are read in, they are translated to ASCII, and when they are written out, they are translated back to EBCDIC.

Note the use of IBMCODE in the second SORT statement. This causes the ASCII records of SFILE (obtained from EBCDICIN) to be sorted using the EBCDIC collating sequence. Since EBCDOUT also names IBMCODE as its code set, the sorted records are translated back from ASCII to EBCDIC when they are written to EBCDOUT. Thus, the result of this sorting operation is an EBCDIC file sorted with the EBCDIC collating sequence. Note that if you had not wanted to be able to display an erroneous record, there would have been no necessity to translate the records form EBCDIC to ASCII or from ASCII to EBCDIC, since the results of sorting the records without using a translation would have been the same. However, since, in the output procedure, any erroneous record is displayed, and since it would have appeared as the ASCII equivalent of EBCDIC characters, the CODE-SET clause was required in the file description of EBCDICIN to translate the records into ASCII.

In the other SORT statement, the collating sequence was specified as SORT-SEQ. The result of this is that the records of OUTFILE are arranged in such a way that all records containing alphabetic characters in their left most positions precede records containing non-alphabetic characters in corresponding positions. This is different from the standard ASCII collating sequence, since, in the standard sequence, all numerals and 55 other characters precede the letters of the alphabet.

CURRENCY SIGN IS CLAUSE

In this clause, you may specify a literal whose value will later be referenced in the PICTURE clause of the Data Division to denote a currency symbol.

CURRENCY SIGN IS Clause Format

[, CURRENCY SIGN IS *literal-9*]

Where

literal-9 is a single character, chosen from a specific set. This set is shown later in this section, when the Currency Sign clause is discussed.

The literal must be a single character not selected from the following group:

- The digits 0 through 9.
- The letters A,B,C,D,L,P,R,S,V,X,Z
- The special characters:

  * (asterisk)
  + (plus sign)
  - (minus sign)
  , (comma)
  . (period or decimal point)
  ; (semicolon)
  ((left-parentheses)
) (right-parentheses)
  " (quotation-mark)
  / (slash-mark)
  = (equal sign)
  (space)

If the CURRENCY SIGN IS clause is omitted, the dollar sign ($) must be used as the currency symbol in the PICTURE clause.

To specify the percent sign (%) as the currency symbol, enter the following clause:

```
      .
      .
      .
CURRENCY SIGN IS "%"
      .
      .
      .
```

DECIMAL-POINT IS COMMA CLAUSE

With this clause, you can request exchange of the function of the comma and decimal-point (or period) in numeric literals or PICTURE-clause character strings.

DECIMAL-POINT IS COMMA Clause Format

[, DECIMAL-POINT IS COMMA] .

This clause has some effect on PICTURES for edited data items. See Section IX, the PICTURE CLAUSE, for details. It may also have an effect on the ACCEPT FREE verb.

INPUT-OUTPUT SECTION

The Input-Output Section allows you to specify information needed to control the transmission and handling of data between the object program and various input/output devices. Specifically, it permits you to define the names of data files, and the devices on which they reside, and special control techniques to be used in the object program.

The Input-Output Section may include the following paragraphs:

- FILE-CONTROL paragraph, for specifying the names of files used by your program and other file-related information.
- I-O CONTROL paragraph, for defining special storage techniques.

INPUT-OUTPUT Section Format

The Input-Output Section has the following format:

```
INPUT-OUTPUT SECTION.

FILE-CONTROL.

   { file-control-entry } ...

[ I-O-CONTROL.
    [          [ RECORD     ]                                              ]
    [ ; SAME   [ SORT       ]  AREA FOR file-name-1 {, file-name-2 } ...   ] ...
    [          [ SORT-MERGE ]                                              ]

    [ ; MULTIPLE FILE TAPE CONTAINS file-name-3 [ POSITION integer-1 ]
                    [, file-name-4 [ POSITION integer-2 ] ] ... ] ... .     ]
```

Each paragraph is discussed separately on the following pages.

FILE-CONTROL PARAGRAPH

The FILE-CONTROL paragraph is used to name files to be used in your program, and to define certain properties of these files which are necessary for their use by your program. Each file named in the FILE-CONTROL paragraph must be described in the Data Division of your program. Conversely, each file described in the Data Division must be named exactly once in the FILE-CONTROL paragraph.

An overview of the types of files which can be used in COBOL II is presented on the following pages; following this overview, the various clauses of the FILE-CONTROL paragraph are discussed.

In COBOL II/3000, you have the ability to access and use files in five different ways. These are:

- Sequential access
- Random access
- Relative access
- Indexed access
- Sort/merge access

Each type of file discussed must be named, and certain features specified, in the Input-Output Section. The organization of the files and their logical records must be described in an SD (for sort/merge files), or an FD (for any other type of files) entry in the File Section of the Data Division.

Sequential Files

Sequential access files are generally files residing on, or being written to a paper tape, punched cards, or serial access device (such as a magnetic tape or a serial disc). Of course it is possible to access disc files sequentially also.

A sequentially accessed file means the records of that file can only be accessed in the order that the records were written to the file. Because of the nature of serial access devices, these types of files can only be written to or read from in a single operation. However, on direct access discs, files being accessed sequentially can be read from and written to at the same time, as well as have a record brought in, modified, and returned to the same storage area.

Random-Access Files

Random-access files must reside on disc. Through the use of a key, you can read or write a record anywhere within a random access file, regardless of whether data has been written on previous records.

The only limitation on where you can write a record is the externally defined boundaries of the file. Thus, for example, if a random-access file has been defined to contain a maximum of three thousand records, you cannot write data to record number 3010, although you can write data to record number 2000 without having written data to any preceding records.

Record access of a random-access file is controlled by the data-item defined in the ACTUAL KEY clause of the FILE-CONTROL paragraph. This data-item is later described in the Working Storage Section as an unsigned integer of from five to nine digits whose USAGE should be described as COMPUTATIONAL SYNCHRONIZED.

The ACTUAL KEY data-item is used by placing a number into it which corresponds to the logical record number in the file. Logical records in a random-access file begin with record 0. Thus, to access the tenth record in a random-access file, your program must move the integer, 9, into the ACTUAL KEY data-item, and then execute the input/output statement.

Execution of an input/output statement for random-access files does not update the ACTUAL KEY data-item. For example, if the ACTUAL KEY data-item contains a value of 1 before reading or writing takes place, the second record is accessed when the READ or WRITE statement is executed; following execution of the statement, any subsequent WRITE statement or READ statement (without the NEXT phrase) will also access record 1 unless the value in the ACTUAL KEY data-item has been changed.

When your program writes data to a record of a random-access file, and previous record areas have had no data written to them, these records are filled with blanks or zeroes. Blank fill is used when the records of the file are designated as ASCII records. Zero fill is used when the records are designated as binary records.

The implication of this blank/zero filling is that you can read records of the file for which no WRITE statement has been executed. In such a case, the data moved into your program will be either a blank record, or a zero-filled record. This capability does not exist for any other type of file.

Relative Files

Relative files are similar to random-access files in that you access records of such a file through the use of a record number. The only real difference in the two keys used for random-access and relative files is that record numbers on a relative file begin with 1, rather than with 0 as in random-access files.

The major functional difference between random-access and relative files is that you can always reuse record areas in a random-access file by simply writing new data in them. In relative files, you must use the DELETE statement to purge data from record areas; once a record has been deleted, you can no longer access the area it occupied except to write a record into it again.

Relative files open in dynamic mode use a data-item named in the RELATIVE KEY clause of the FILE-CONTROL paragraph to access records. This data-item must be described in the File or Working Storage section of your program as an unsigned integer of from five to nine digits whose USAGE is COMPUTATIONAL SYNCHRONIZED. It must not be part of a record description entry for the file with which it will be used.

Relative files open in sequential mode do not need to use the RELATIVE KEY data-item to access records. You can simply execute input/output operations on them as though they were sequential files. If, however, you wish to position the file by using the START statement, you must specify the RELATIVE KEY data-item, since the START statement uses this data-item to find the record you want.

In COBOL II/3000, your program can access a relative file in one of two ways. These two ways are described below.

- **Sequential Access.**

  Sequential WRITE statements for a relative file release data to the file, starting with the first record on the file, and proceeding to the second, third, and so forth in turn. If the RELATIVE KEY data-item has been specified, it is updated each time a sequential WRITE statement is executed. Note that even though you may already have data on these records, a sequential WRITE statement will cause the new data to replace it.

Sequential READ statements for relative files start with a particular record, read it, and proceed to the next existing record.

The record which is read is dependent upon the type of the last input/output statement executed before the READ statement is encountered. If a DELETE or READ statement was executed, the READ statement will read the next existing record which follows the record just read or deleted. If an OPEN statement was executed, the first existing record is read. If a successful START statement was executed, the record pointed to is read.

Sequential DELETE statements require the use of the READ statement to position the file to the record to be deleted. This READ statement must have been the last input/output operation to have been performed on the file before the DELETE statement was encountered.

Sequential REWRITE statements require the use of the READ statement to position the file to the record to be rewritten. This READ statement must have been the last input/output statement to have been performed on the file before the REWRITE statement was encounted.

- **Dynamic Access.**

A relative file open in dynamic access mode allows you to access your file in either random or sequential mode.

Random access input/output statements use the required RELATIVE KEY data-item to select the record for the READ, WRITE, REWRITE, and DELETE statements. Thus, to perform a random access input/output operation on a relative file, you must place the number of the record to be accessed into the RELATIVE KEY data-item before you execute the input/output statement.

When the random access input/output operation is executed, an implicit SEEK statement is performed to find the record, and the specified input/output operation is performed if possible.

The only permissable sequential access which can be performed on a relative file open in dynamic access is the READ NEXT form of the READ statement. This statement allows you to access the records of the file, starting with the record pointed to (if valid) by the current record pointer. If the record is invalid (that is, has been deleted), the next valid record in the file is read.

Indexed Files

Indexed files use data-items which are integral parts of the records to control accessing of the records.

For a given indexed file, each record contains a single prime record key, and zero or more alternate record keys. Each key must be described as alphanumeric within the record description entry of the associated indexed file. To indicate which key is the prime record key, and which, if any, are the alternate keys, you must specify their names in the RECORD KEY and ALTERNATE RECORD KEY clauses respectively, in the FILE-CONTROL paragraph of the Environment Division.

The prime record key is used in writing, deleting, or updating records of an indexed file. Alternate record keys are used only in reading records.

The values of both prime and alternate record keys may be duplicated within an indexed file. Note however, that since the prime record key must be used for all input/output operations except reading, you should be very careful to make sure that if you are deleting a record whose prime key has duplicates, it is the record you wish to delete, and not some other record which has the same prime record key value.

When a prime record key has duplicates, and you use such a key to access a record on an indexed file, the file is searched in a chronological order. That is, the file is searched according to which record was written first. Thus, the record accessed is the first record containing the specified record key value which is still active.

As with relative files, an indexed file can be accessed in random or sequential mode. The actions taken for a specific access mode and a specific input/output operation are listed below.

- **Sequential Access.**

A sequential access READ statement for an indexed file uses the current record pointer to read records from the file. The record selected to be read is determined in essentially the same way as for relative files open in sequential mode. See the discussion on sequential access of relative files, above.

A sequential access WRITE statement for an indexed file uses the prime record key to place records on the file. Records must be written in ascending order according to these keys. Since the prime record key (and all alternate keys as well) must be alphanumeric, this means that the records must be written using the ASCII collating sequence to determine ascending order.

A sequential access REWRITE or DELETE statement requires that a READ statement be the last executed input/output statement for the file being referenced.

- **Dynamic Access.**

When dynamic access is used for indexed files, you can use either the READ NEXT form of the READ statement, or you can use the READ ... KEY IS form.

The READ NEXT form is used when you wish to read records in a sequential manner. This statement allows you to access the records of the file starting with the record pointed to by the current record pointer (if valid). If the record is invalid, the next valid record in the file is read.

The READ ... KEY IS form requires that you place a key value into one of the record key data-items (prime or alternate). This data-item is then specified in the READ statement, and the indexed file is searched until a record having the same value in the same record key is found. This record is then brought into your program.

A dynamic access WRITE, REWRITE, or DELETE statement uses the contents of the RECORD KEY data-item to select the record to be written or deleted.

SORT-MERGE Files

The ability to arrange records in a particular order is often required in COBOL applications. This ability is provided by the sort and merge features of COBOL II/3000.

The sort facility allows you to arrange the records of one or more files in a specified sequence. The merge facility merges two already sorted files.

Sort/merge files are the files acted upon by the sort and merge operations. These files can never be accessed directly, except in input and output routines associated with a SORT or MERGE statement.

When a SORT or MERGE statement is issued, you must name a file or a procedure from which records are inserted in the specified sort/merge file, and you must name a file or procedure into which the sorted or merged records are to be placed.

See Section XIII, SORT-MERGE OPERATIONS IN COBOL II/3000, for more information on sort/merge files and operations.

FILE STATUS

Every type of file discussed above may have a file status data item associated with it. This data-item can be used to check on the success or failure of an input/output operation involving the file with which it is associated. File status data-items are discussed in more detail later in this section.

FILE-CONTROL CLAUSES

The various clauses of the FILE-CONTROL paragraph can be specified in five separate formats, depending upon the type of file being described. These formats are shown below.

Format 1 - For Sequential Files

```
SELECT [ OPTIONAL ] file-name
   ASSIGN TO "file-info-1" [, "file-info-2" ] ...

  [            { integer-1 } [ AREA  ] ]  [                       { REEL } ]
  [ ; RESERVE  {           } [       ] ]  [ FOR MULTIPLE          {      } ]
  [            { NO        } [ AREAS ] ]  [                       { UNIT } ]

  [; ORGANIZATION IS SEQUENTIAL ]

  [; ACCESS MODE IS SEQUENTIAL ]

  [; FILE STATUS IS stat-item ].
```

Format 2 - For Relative Files

```
SELECT file-name
   ASSIGN TO "file-info-1" [, "file-info-2" ] ...

  [                          [ AREA  ] ]
  [ ; RESERVE  integer-1     [       ] ]
  [                          [ AREAS ] ]

   ; ORGANIZATION IS RELATIVE

  [                     { SEQUENTIAL [, RELATIVE KEY IS data-name-1 ]      } ]
  [                     { { RANDOM  }                                      } ]
  [ ; ACCESS MODE IS    { {         } , RELATIVE KEY IS data-name-1        } ]
  [                     { { DYNAMIC }                                      } ]

  [; FILE STATUS IS stat-item ] .
```

Format 3 - For Random-Access Files

```
SELECT file-name
    ASSIGN TO [ integer-1 ] "file-info-1" [, "file-info-2" ] ...

  [ ; RESERVE { integer-1 } [ AREA  ] ]
              { NO        } [ AREAS ]

  ; ACCESS MODE IS RANDOM

  [; PROCESSING MODE IS SEQUENTIAL ]

  ; ACTUAL KEY IS data-name-1

  [; FILE STATUS IS stat-item ] .

  [ ; { FILE-LIMIT IS    } { data-name-1 } THRU { data-name-2 }
      { FILE-LIMITS ARE  } { literal-1   }      { literal-2   }

      [ , { data-name-3 } THRU { data-name-4 } ] ... ]
          { literal-3   }      { literal-4   }
```

Format 4 - For Indexed Files

```
SELECT file-name
    ASSIGN TO "file-info-1" [, "file-info-2" ] ...

  [ ; RESERVE integer-1 [ AREA  ] ]
                        [ AREAS ]

  ; ORGANIZATION IS INDEXED

  [ ; ACCESS MODE IS { SEQUENTIAL } ]
                     { RANDOM     }
                     { DYNAMIC    }

  ; RECORD KEY IS data-name-1 [ WITH DUPLICATES ]

  [; ALTERNATE RECORD KEY IS data-name-2 [ WITH DUPLICATES ]]

  [ ; FILE STATUS IS stat-item ].
```

Format 5 - For Sort-Merge Files

SELECT ***file-name***

ASSIGN TO " file-info-1" [, ***" file-info-2"***] ...

In the FILE-CONTROL paragraph body, the SELECT clause must be specified first. The remaining clauses may appear in any order.

Each of the clauses of the FILE-CONTROL paragraph are discussed in alphabetical order on the following pages, except the SELECT and ASSIGN clauses. Since these clauses must be specifed first for any file type, they are discussed first.

SELECT CLAUSE

The SELECT clause is used to identify a file to be used in your program. It has two formats:

Format 1 - For Sequential Files

SELECT [**OPTIONAL**] ***file-name***

Format 2 - For All Other Files

SELECT ***file-name***

Where

file-name is any valid user-defined COBOL word.

file-name is the name you use later in OPEN, CLOSE, USE, and other statements in the Procedure Division. It must also be used in an FD or an SD entry in the Data Division.

OPTIONAL Phrase

The OPTIONAL phrase can only be used with sequential files.

The purpose of the OPTIONAL phrase is to allow you to specify an input file in the SELECT statement which may not be present during a particular execution of the program in which it is named. If the file is not present, and the OPTIONAL phrase has been specified, then when the first READ statement naming that file is executed, the imperative statement in the associated AT END phrase, if any, is executed. If no AT END phrase has been specified, then a USE procedure must have been defined, either explicitly or implicitly, and this procedure is executed.

ASSIGN CLAUSE

The ASSIGN clause associates a file with the storage medium on which the file resides. It has the following two formats:

Format 1 - For Random-Access Files

<u>ASSIGN</u> TO [*integer-1*] "*file-info-1*" [, "*file-info-2*"] ...

Format 2 - For All Other Files

<u>ASSIGN</u> TO "*file-info-1*" [, "*file-info-2*"] ...

integer-1 is entry for random-access files, if specified, is treated as a comment by COBOL II/3000. It is provided only for ease of conversion from programs written in '68 COBOL/3000.

File-info-1 is of the form,

name* [, [*class*] [, [*recording mode*] [, [*device
[(CCTL)]] [, [*filesize*] [, [*formsmessage.*] [, L]]]]]]

Only the file designated by *file-info-1* is defined. Succeeding file names, such as *file-info-2*, are treated as comments. The meanings of each parameter in the format of *file-name-1* are discussed below.

name is the MPE file designator of the form,

file/lockword.group.account

This is the fully qualified form of a file name. If the file exists in your log-on group, then you need specify only the *file* parameter and any *lockword* assigned to it. If the file exists in a group other than your log-on group, you must specify the *file* (and *lockword*, if any), and the *group* in which the file resides. Finally, if the file exists in a group and account other than your log-on, you must use the fully qualified file name. Note that in the case that the file resides in a group and/or account other than your own, you must have access to the file. For details, see the System Mangager, System Supervisor Reference Manual, part number 3000-90014, for details on file security and access.

class is the class of device on which the file resides. This parameter is not used by the MPE file system, and is ignored if it is specified. However, if specified, the class parameter can be one of the following three mnemonics:

DA, implying a mass-storage device.

UT, implying a utility device such as a tape drive.

UR, implying a unit-record device, such as a card reader.

If the class parameter is omitted, DA is assigned by default.

recording mode is the recording mode of the file. It may be either ASCII or binary. If the file is an ASCII file, recording mode must be A. If the file is binary, the recording mode must be B.

If the recording mode parameter is omitted, ASCII is assigned by default.

device is the type of device on which the file resides. This allows you to select an MPE-defined device type (such as DISC, TAPE, or LP) or a private volume. If you omit this parameter, the MPE default is assumed. This default is currently DISC. If this parameter is followed by a left-parenthesis, the CCTL option described below is assumed.

CCTL is the carriage-control option for an output file, indicating that carriage-control directives are supplied in write-operations referencing line-printer files. If omitted, your program uses the MPE default for the device or file.

filesize is the number of records in the file. This value may be a maximum of nine digits long. If omitted, a file size of 10,000 records is assigned by default.

forms-message. is, for a listing device, a request for the operator to provide special forms, such as blank checks or inventory report forms, on the printer. For any other device, this parameter is ignored. This entry may contain a maximum of 49 characters and must be terminated with a period.

L is not a required parameter if the EXCLUSIVE and UN-EXCLUSIVE statements are used. Otherwise, it enables your program to dynamically lock and unlock a disc file. If any entry other than this is specified in this position, COBOL II/3000 displays a warning message and dynamic locking/unlocking remains disabled until the $CONTROL LOCKING compiler subsystem command is encountered in your program.

This feature is provided only to assist in the conversion of '68 COBOL/3000 programs to COBOL II/3000. It is recommended that the EXCLUSIVE and UN-EXCLUSIVE statements be used instead for file locking and unlocking.

ACCESS MODE CLAUSE

The ACCESS MODE clause specifies the way in which your program is to access the associated file.

There are four formats of this clause, depending upon the type of file being described. Note that this clause cannot be used for sort/merge files. The four formats are shown below.

Format 1 - For Sequential Files

```
[; ACCESS MODE IS SEQUENTIAL ]
```

Format 2 - For Relative Files

```
[                   { SEQUENTIAL [, RELATIVE KEY IS data-name-1 ]  } ]
[ ; ACCESS MODE IS  { { RANDOM  }                                  } ]
[                   { {         } , RELATIVE KEY IS data-name-1    } ]
[                   { { DYNAMIC }                                  } ]
```

Format 3 - For Random-Access Files

```
; ACCESS MODE IS RANDOM
```

Format 4 - For Indexed Files

```
[                   { SEQUENTIAL } ]
[ ; ACCESS MODE IS  { RANDOM     } ]
[                   { DYNAMIC    } ]
```

Where

data-name-1 is a data-item which must not be defined in the record description entry for the relative file being described. Furthermore, the item must be defined as an unsigned integer.

For example, such a data item might be defined as:

USAGE COMPUTATIONAL SYNCHRONIZED PIC 9(***n***),

where ***n*** is an integer in the range 5 to 9.

Note that even if you wish to access a relative file sequentially, you must specify the RELATIVE KEY phrase if you want to reference the associated file in a START statement.

For three of the four file types to which this clause pertains, (sequential, relative, and indexed) if you do not specify an ACCESS MODE clause, SEQUENTIAL access is assumed. You must specify the ACCESS MODE clause exactly as it is shown for random-access files.

In ANSI COBOL '74, the RANDOM and DYNAMIC access modes are distinct; however, on an HP computer system, they are equivalent, and can therefore be used interchangably.

The two access modes are defined as follows:

- Sequential access means that existing records are accessed in ascending order. The relative key is used for relative files, and a prime or alternate record key is used for indexed files. Random-access files may not be accessed sequentially.

- Dynamic access means that your program may alternate between sequential and random access modes, by selectively using different forms of various input/output statements. This type of access may only be used for relative, indexed, and random-access files.

Random access means that records are accessed directly, by using a record key data-item (for indexed files), or by using the relative record numbers of records (for relative and random-access files).

For details on how sequential and dynamic access is performed on the various file types, refer to the overview on preceding pages of this section.

ACTUAL KEY CLAUSE

The ACTUAL KEY clause applies only to random-access files.

The ACTUAL KEY clause names the data-item to be used in accessing records of a random-access file. It has the following format:

ACTUAL-KEY Clause Format

; ACTUAL KEY IS *data-name-1*

Where

data-name-1	is an integer item of from one to nine digits.

For greatest efficiency the variable *data-name-1* should have a PICTURE clause of the form of the example below.

```
PIC S9(5) USAGE COMPUTATIONAL SYNCHRONIZED
```

This data-item must be defined in either the File Section or the Working Storage Section of the Data Division. It corresponds to a relative record number. Record numbers in random-access files begin with 0. To insure the accessibility of all records, the data-item must be large enough to contain the greatest record number in the file.

ALTERNATE RECORD KEY CLAUSE

Each use of the ALTERNATE RECORD KEY clause names an alternate record key for an indexed file. The number of alternate keys for a file must be the same as the number used when the file was created.

ALTERNATE RECORD KEY Clause Format

[; **ALTERNATE RECORD** KEY IS *data-name-2* [WITH **DUPLICATES**]]

Where

data-name-2 is the name of a data-item described as alphanumeric in a record description entry for the file with which it is associated. It must not reference an item whose first character begins in the same position as the first character of any other key, whether alternate or prime.

DUPLICATES Phrase

The DUPLICATES phrase specifies that the named alternate key may be duplicated within any of the records of the file. If this phrase is unused, the contents of the data-item referenced by *data-name-2* must be unique among records of the file.

FILE LIMIT CLAUSE

The FILE LIMIT clause can be used only in the description of a random-access file; it is treated as a comment by COBOL II/3000, and is included only as a carry-over from '68 COBOL/3000.

FILE STATUS CLAUSE

The FILE STATUS clause allows you to name a data-item to be used in obtaining information about the success or failure of input/output operations performed using the file being described.

This clause is optional, and may be used for all types of files except sort/merge files.

FILE STATUS Clause Format

[; FILE STATUS IS *stat-item*].

Where

stat-item	is a two character alphanumeric data-item defined in the Working-Storage Section of the Data Division.

When an input or output operation has been performed on a file which has a FILE STATUS *data-item* associated with it, the *data-item* is updated with two characters which indicate the status of the operation. The left most character of this *data-item* is called ***status-key 1***. The right most character is called ***status-key 2***. The values which can be placed in *status-key 1* and *status-key 2*, and their meanings, are shown in the following tables.

Table 7-1.

STATUS-KEY 1 SETTINGS FOR SEQUENTIAL FILES

Setting	Meaning
0	Successful completion. The requested input or output operation was successfully completed.
1	At end. A sequential READ statement failed to execute successfully for one of these reasons: 1. The program attempted to read a record following the last record in the file. 2. The program attempted to read a record from a file that was described with the OPTIONAL clause, but that was not available when the associated OPEN statement was executed.
3	Permanent error. The input-output statement failed to execute successfully because of a boundary violation for the file or an input-output error (such as a parity or transmission error). When this value is placed in status-key 1, it is strongly recommended that you use the CALL statement to call the MPE intrinsic, PRINTFILEINFO. This intrinsic will print a file information display which can help you to determine the cause of the error. File information displays are discussed in Section X of the MPE Intrinsics Reference Manual.
9	Other error. The input-output statement failed because of some other error.

STATUS-KEY 1 SETTINGS FOR RELATIVE, RANDOM-ACCESS, AND INDEXED FILES

Setting	Meaning
0	Successful completion. The requested input or output operation was successfully completed.
1	At end. A sequential read statement failed to execute because the program attempted to read beyond the last record in the file.
2	Invalid Key. The input/output operation failed because a duplicate key existed, a boundary violation occurred, the record sought could not be found, or a sequence error (for indexed files only) occurred.
3	Permanent error. The input/output operation failed because of a physical input/output error, such as a data-check, parity, or transmission error. As with sequential files, you should use the intrinsic, PRINTFILEINFO, to obtain more information about what error has occurred.
9	Other error. The input/output operation failed because of some other error.

Table 7-2.

STATUS-KEY 2 SETTINGS FOR SEQUENTIAL FILES

Setting	Meaning
0	No additional status information is available.
4	When status key 1 is set to 3, a value of 4 in status key 2 denotes a boundary violation. That is, the program attempted to write beyond the externally- defined boundaries of a sequential file. Any other setting denotes an input-output error.
*	When status-key 1 is set to 9, an unexpected error has occurred. In this case, the value placed in status-key 2 is a binary integer quantity corresponding to an MPE file system error number. Since this quantity can range from 0 to 255, and since the status-key item is alphanumeric, your program will interpret this integer as some character from the ASCII collating sequence. To obtain the MPE error number to which this item corresponds, you can, and should, use the CKERROR intrinsic (by calling it through the CALL statement). An example of this technique is given at the end of the discussion on the FILE-CONTROL paragraph.

STATUS-KEY 2 SETTINGS FOR RELATIVE AND RANDOM ACCESS FILES

Setting	Meaning
0	No additional status information is available.
2	When status key 1 contains 2 (for invalid key), and status key 2 contains 2, an attempt was made to write a record that would create a duplicate key in a relative file.
3	When status key 1 contains 2 (for invalid key), and status key 2 contains 3, and attempt was made to access (by key) a record that does not exist in the file.
4	When status key 1 contains 2 (for invalid key) and status key 2 contains 4, an attempt was made to write beyond the externally defined boundaries of a relative file.
*	When status-key 1 is set to 9, an unexpected error has occurred. In this case, the value placed in status-key 2 is a binary integer quantity corresponding to an MPE file system error. Since this quantity can range from 0 to 255, and since the status-key item is alphanumeric, your program will interpret this integer as some character from the ASCII collating sequence. To obtain the MPE error number to which this item corresponds, you can, and should, use the CKERROR intrinsic (by calling it through the CALL statement). An example of this technique is given at the end of the discussion on the FILE-CONTROL paragraph.

Table 7-2 (continued)

STATUS-KEY 2 SETTINGS FOR INDEXED FILES

Setting	Meaning
0	No further information is available.
1	A Sequence error has occurred during a sequential access of the file. Either the primary record key value has been changed between a READ and the following REWRITE, or the ascending sequence requirements of successive key values have been violated.
2	A duplicate key exists. For status-key 1 set to 0, this implies that for a READ statement, the current key is the same as the key in the next record within the current key of reference. For a WRITE or REWRITE statement, this implies that the record just written created a key value that duplicates the key value of one or more records. Since status-key 1 is 0, the DUPLICATES phrase must have been used for the key value. If status-key 1 is set to 2, a value of 2 for statuskey 2 indicates that an attempt was made to write or rewrite a record that would create a duplicate key.
3	No record was found. An attempt was made to access a record, identified by key, and that record does not exist.
4	A Boundary Violation occurred. An attempt was made to write beyond the externally defined boundaries of the file.
*	When status-key 1 is set to 9, an unexpected error has occurred. In this case, the value placed in status-key 2 is a binary integer quantity corresponding to an MPE file system error. Since this quantity can range from 0 to 255, and since the status-key item is alphanumeric, your program will interpret this integer as some character from the ASCII collating sequence. To obtain the MPE error number to which this item corresponds, you can, and should, use the CKERROR intrinsic (by calling it through the CALL statement). An example of this technique is given at the end of the discussion on the FILE-CONTROL paragraph.

The valid combinations of settings for status keys 1 and 2 are shown in table 7-3 with each combination indicated by an X at the appropriate intersection.

Table 7-3.

VALID COMBINATIONS FOR STATUS KEYS 1 AND 2 FOR SEQUENTIAL FILES

Status-key 1 / Status-key 2	Successful Completion (0)	At End (1)	Permanent Error (4)	Other Error (9)
No Further Information (0)	X	X	X	
Boundary Violation (4)			X	
File System Error Number (*)				X

VALID COMBINATIONS FOR STATUS KEYS 1 AND 2 FOR RELATIVE AND RANDOM-ACCESS FILES

Status-key 1 / Status-key 2	Successful Completion (0)	At End (1)	Invalid Key (2)	Permanent Error (3)	Other Error (9)
No Further Information (0)	X	X		X	
Duplicate Key (2)			X		
No Record Found (3)			X		
Boundary Violation (4)			X		
File System Error Number (*)					X

Table 7-3 (Continued)

VALID COMBINATIONS FOR STATUS KEYS 1 AND 2 FOR INDEXED FILES

Status-key 1 / Status-key 2	Successful Completion (0)	At End (1)	Invalid Key (2)	Permanent Error (3)	Other Error (9)
No Further Information (0)	X	X		X	
Sequence Error (1)			X		
Duplicate Key (2)	X		X		
No Record Found (3)			X		
Boundary Violation (4)			X		
File System Error Number (*)					X

ORGANIZATION CLAUSE

The ORGANIZATION clause specifies the logical structure of the file being described. It can be used in sequential, relative, and indexed files.

The three formats of the ORGANIZATION clause are shown below.

Format 1 - For Sequential Files

[; <u>ORGANIZATION</u> IS <u>SEQUENTIAL</u>]

Format 2 - For Relative Files

; <u>ORGANIZATION</u> IS <u>RELATIVE</u>

Format 3 - For Indexed Files

; <u>ORGANIZATION</u> IS <u>INDEXED</u>

The ORGANIZATION clause is required for relative and indexed files. It is optional for sequential files. Note that this clause cannot be used for sort/merge and random-access files.

RECORD KEY CLAUSE

The RECORD KEY clause is used for indexed files only. It is required, since it provides the means (*data-name-1*) by which the associated indexed file is accessed.

RECORD KEY Clause Format

; RECORD KEY IS *data-name-1* [WITH DUPLICATES]

Where

data-name-1 is the name of an alphanumeric data-item defined in a record description entry associated with the file being described. *Data-name-1* is the prime record key for the file.

The data description for *data-name-1*, and its relative position within a record must be the same as that used when the file was created.

DUPLICATES Phrase

The DUPLICATES phrase specifies that the named prime record key may be duplicated within any of the records of the file. If, however, you do not specify that duplicates may exist, then the value of the key must be unique among records of the file.

RESERVE CLAUSE

The RESERVE clause allows you to indicate the number of input/output buffers to be allocated for the file being described. Its use is optional. If you do not specify it, the number of buffers allocated is two (the MPE default).

This clause may be used for sequential, relative, indexed, and random-access files. It may not be used for sort/merge files.

The two formats for the RESERVE clause are shown below.

Format 1 - Sequential Files:

```
[          { integer-1 } [ AREA  ]] [                  { REEL } ]
[ ; RESERVE {           } [       ]] [ FOR MULTIPLE     {      } ]
[          { NO        } [ AREAS ]] [                  { UNIT } ]
```

Format 2 - Relative, Random-Access, and Indexed Files:

```
[                           [ AREA  ]]
[ ; RESERVE  integer-1      [       ]]
[                           [ AREAS ]]
```

Where

integer-1 is a non-negative integer in the range 0 to 16.

Note, if you specify 0, MPE will still allocate two (the default number). Also, although you can specify more than two, any more than three buffers does not usually increase input/output efficiency. See the INPUT/OUTPUT BUFFERS paragraph in Section VI of the MPE Commands Reference Manual for more information on buffers.

In the RESERVE clause for sequential files, the NO option is equivalent to specifying one buffer for input/output operations. This option is not part of ANSI COBOL '74. It is a carry-over from '68 COBOL/3000. The MULTIPLE REEL phrase is also a carry-over from COBOL/3000. It is treated as a comment by COBOL II/3000.

The words, AREA and AREAS are equivalent, and can be used interchangably.

PROCESSING MODE CLAUSE

The PROCESSING MODE clause applies to random-access files only.

This clause is a carryover from '68 COBOL/3000. It is treated as a comment if included in a COBOL II/3000 program.

PROCESSING MODE Clause Format

[; PROCESSING MODE IS SEQUENTIAL]

FILE-CONTROL PARAGRAPH EXAMPLE

The following example shows a FILE-CONTROL paragraph for an indexed and a sequential file, as well as important data-items associated with the files.

```
                 .
                 .
                 .
ENVIRONMENT DIVISION.
                 .
                 .
                 .
INPUT-OUTPUT SECTION.
FILE-CONTROL.
SELECT INDXFILE
  ASSIGN TO "KFILE,DA,A,DISC,,5000,,L"
  RESERVE 3 AREAS
  ORGANIZATION IS INDEXED
  ACCESS MODE IS DYNAMIC
  RECORD KEY IS FIRST-CHARS
   ALTERNATE RECORD KEY IS SECOND-CHARS WITH DUPLICATES
  FILE STATUS IS CHECK-KFILE.

SELECT TAPEIN
  ASSIGN TO "READTAPE,UT,ASCII,TAPE,,,HANG TAPE 001."
  RESERVE 2 AREAS
  FILE STATUS IS CHECK-TAPE.
                 .
                 .
                 .
DATA DIVISION.
FILE SECTION.
FD INDXFILE
    LABEL RECORDS ARE OMITTED
01 RECORD-IN.
   02 FIRST-CHARS                      PIC X(8).
   02 SECOND-CHARS                     PIC X(24).
                 .
                 .
                 .
FD TAPEIN
    LABEL RECORDS ARE OMITTED.
01 TAPE-REC                            PIC X(80).
                 .
                 .
WORKING-STORAGE SECTION.
01 CHECK-KFILE.
   02  STAT-KEY-1   PIC X.
   02  STAT-KEY-2   PIC X.
01 CHECK-TAPE.
   02  STAT-KEY-1   PIC X.
   02  STAT-KEY-2   PIC X.

77 CATCHNUM         PIC 9(4)  USAGE DISPLAY.
```

The example below shows how CKERROR can be used to find the error number returned in STAT-KEY-2 when STAT-KEY-1 contains the value 9.

```
                    .
                    .
                    .
PROCEDURE DIVISION.
START-SECTION.
OPEN-PARAGRAPH.
OPEN INPUT TAPEIN, I-O INDXFILE.
                    .
                    .
                    .
READ-TAPE SECTION.
  READ-TAPE-PARA-1.
  READ TAPE-REC; AT END PERFORM READ-COMPLETE.
  IF STAT-KEY-1 OF CHECK-TAPE IS EQUAL TO "9"
     CALL "CKERROR" USING CHECK-TAPE, CATCHNUM
     DISPLAY '"9" ERROR IN TAPE READ. SECOND VALUE IS ', CATCHNUM
     PERFORM CHOOSE-ACTION-ROUTINE.
                    .
                    .
                    .
```

Assuming that an error has occurred which causes the FILE STATUS data-item of TAPEIN to contain the characters, 9/, the call to CKERROR would return the value 47 to CATCHNUM. If you then used this number to consult the File System Error Codes table in the MPE Error Messages and Recovery Manual, you would find the number to represent the message, I/O ERROR ON FILE LABEL. This should indicate to you that the magnetic tape upon which TAPEIN resides has a faulty label.

Note that the value in STAT-KEY-2 of CHECK-TAPE could have been, for instance, an ASCII 7. This is not an error number, and must be translated using the CKERROR intrinsic just as the slash character was. When this value is translated, the error message found to correspond to the value in CATCHNUM (in this case, 55) is DEVICE UNAVAILABLE. This should tell you that the magnetic tape unit upon which TAPEIN was to be loaded is unavailable because it is out of service, it is being used by some other process, or the operator determined there was an error.

I-O-CONTROL PARAGRAPH

The I/O CONTROL Paragraph specifies the areas of memory (buffers) to be shared by different files, and the locations of files on a multi-file tape reel. This paragraph has the following format:

I-O-CONTROL Paragraph Format

```
I-O-CONTROL.
  [         [RECORD    ]                                              ]
  [ ; SAME  [SORT      ]  AREA FOR file-name-1 {, file-name-2 } ...   ] ...
  [         [SORT-MERGE]                                              ]

  [ ; MULTIPLE FILE TAPE CONTAINS file-name-3 [ POSITION integer-1 ]
                     [, file-name-4 [ POSITION integer-2 ] ] ... ] ... .
```

This paragraph is optional.

SAME CLAUSE

The SAME clause has three formats:

Format 1

<u>SAME</u> AREA FOR ***file-name-3*** {, ***file-name-4*** } ...

Format 2

<u>SAME</u> <u>RECORD</u> AREA FOR ***file-name-3*** {, ***file-name-4*** } ...

Format 3

<u>SAME</u> { **<u>SORT</u>** / **<u>SORT-MERGE</u>** } **AREA FOR** ***file-name-3*** {, ***file-name-4*** } ...

The meanings and restrictions of these formats are discussed below.

SAME AREA Clause

The SAME AREA clause allows you to conserve main memory space by permitting two or more non-sort/merge files to use the same area of main memory for processing the file.

Because the area shared includes all storage areas assigned to the files specified, only one file can be open at any given time.

Also, a file name can appear in only one SAME AREA clause within a program. This does not preclude the possibility, however, of the file name appearing in a SAME RECORD AREA or SAME SORT and SAME SORT-MERGE AREA clause.

The restrictions on file names appearing in more than one SAME RECORD, SAME SORT, or SAME SORT-MERGE AREA are listed on the following pages, under the headings, SAME RECORD AREA CLAUSE and SAME SORT AND SAME SORT-MERGE AREA CLAUSES.

The files referenced in the SAME AREA clause need not all have the same organization or access.

To specify that FILEA, FILEB, and FILEC share the same processing area, enter:

```
                .
                .
                .
     SAME AREA FOR FILEA, FILEB, FILEC.
                .
                .
```

SAME RECORD AREA Clause

The SAME RECORD AREA clause specifies that two or more files (of any kind) be allowed to share the same area of main memory for processing the current logical record.

If none of the files in this clause appear in a SAME AREA clause, then all of the files may be open at the same time.

A logical record in this shared record area is considered a logical record of each open output file named in the SAME RECORD AREA clause. It is also considered a record of the most recently opened input file named in the SAME RECORD AREA clause. This is equivalent to an implicit redefinition of the shared area. That is, records are aligned on the left most character position.

If any file appears in the SAME RECORD AREA clause and in the SAME AREA clause, then all other files appearing in the SAME AREA clause must also appear in the SAME RECORD AREA clause. Of course, files not named in the SAME AREA clause may also appear in the SAME RECORD AREA clause. Because of this restriction, and since only one file named in a SAME AREA clause may be open at any given time, if any file in the SAME RECORD AREA clause also appears in a SAME AREA clause, then the rule that only one file at one time may be open takes precedence over the rule that all files named in the SAME RECORD AREA clause may be open at the same time.

As with files named in the SAME AREA clause, files named in the SAME RECORD AREA clause may have different organizations and access modes. Also, if a file name appears in a SAME RECORD AREA clause, it may not appear in any other SAME RECORD AREA clause within the program.

SAME SORT And SAME SORT-MERGE AREA Clauses

The SAME SORT AREA and SAME SORT-MERGE AREA clauses are equivalent.

Both specify that the area used in main memory for sorting or merging sort/merge files be shared.

No sort or merge file may appear in a SAME AREA clause; however, not all files named in a SAME SORT or SAME SORT-MERGE AREA clauses need be sort or merge files. Only one must be. Furthermore, any sort or merge file which appears in a SAME SORT or SAME SORT-MERGE AREA clause may not appear in another SAME SORT or SAME SORT-MERGE AREA clause within the same program.

If a non-sort or non-merge file appears in a SAME AREA clause and in one or more SAME SORT or SAME SORT-MERGE AREA clauses, then all files named in that SAME AREA clause must appear in the SAME SORT or SAME SORT-MERGE AREA clauses.

During the execution of a SORT or MERGE statement that refers to a file named in a SAME SORT or SAME SORT-MERGE AREA clause, those files which are not sort or merge files, but are named in the SAME SORT or SAME SORT-MERGE AREA clauses, must not be open.

The files named in a SAME SORT or SAME SORT-MERGE AREA clause need not have the same organization or access mode.

MULTIPLE FILE CLAUSE

The MULTIPLE FILE clause has the following format.

MULTIPLE FILE Clause Format

[; <u>**MULTIPLE**</u> <u>**FILE**</u> **TAPE CONTAINS** ***file-name-3*** [<u>**POSITION**</u> ***integer-1***]
[, ***file-name-4*** [<u>**POSITION**</u> ***integer-2***]] ...] ... .

When the file referenced by your program shares a labeled tape with other files, you must enter the MULTIPLE FILE clause. Regardless of the number of files on the reel, you need specify only those used by your program. If you specify the files in chronological order, you need only enter the file names in the MULTIPLE FILE clause:

```
                    .
                    .
                    .
       MULTIPLE FILE TAPE CONTAINS FILEA, FILEC, FILEF
                    .
                    .
                    .
```

But if you specify the files in random order, you must indicate their positions by using the POSITION clause:

```
                    .
                    .
                    .
       MULTIPLE FILE TAPE CONTAINS FILEC POSITION 3
           FILEF POSITION 6, FILEA POSITION 1
```

(The first position, in relation to the beginning of the reel, is position 1.)

No more than one file on the same tape device may be open at the same time.

NOTE

The MULTIPLE FILE clause applies to labeled sequential files only. This is because of the intrinsically sequential nature of magnetic tape devices.

DATA DIVISION

SECTION VIII

In every COBOL program, the Data Division follows the Environment Division.

The Data Division describes all data that your program is to use. That is, it describes any data to be read from or written to a file, data developed internally and held in temporary or working storage, and any constants that are used.

There are five sections within the Data Division:

- **File Section**, used to define the structure of data files.

- **Working Storage Section**, used to describe data items used as constants by the object program, as well as records and non-contiguous data items which are developed and used internally, and are not part of external data files.

- **Linkage Section**, used to describe data items within subprograms that are referenced by both the calling and the called program. This section appears only in programs that will be called by some other program. Its format is the same as the format of the Working Storage Section.

- **Communication Section**, used to describe the data item in the source program which serves as the interface between the message control system and your program. This section is not implemented in COBOL II/3000.

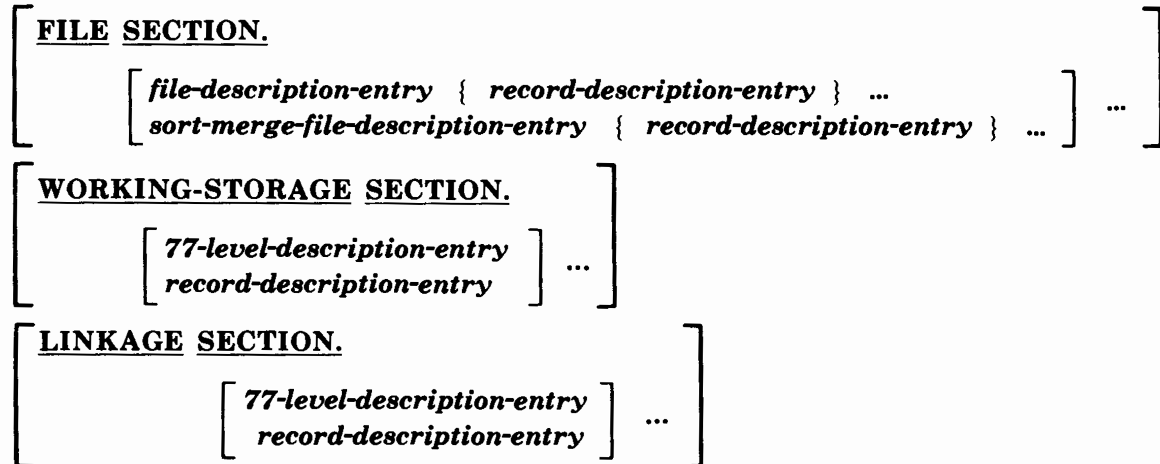
  Thus, although you may enter the complete format for the Communication Section in an COBOL II/3000 program, the COBOL II/3000 compiler only performs a syntax check on the section, and once your program is compiled, treats the entries in the Communication Section as comments.

- **Report Section**, used to describe the format of reports to be generated. This section is not implemented in COBOL II /3000. Thus, as with the Communication Section, you may enter the complete format of the Report Section, but it is only checked for syntax, and is thereafter treated as a comment.

The formats of the Communication and Report sections are shown in Appendix J of this manual, but are not discussed in any detail within this manual.

DIVISION Format

The Data Division has the general format,



The format of each section is discussed separately on the following pages, followed by a discussion of record description entries.

DIVISION SYNTAX RULES

All sections within the Data Division are optional. When included, however, they must appear in the order shown in the format description.

Each Data Division entry begins with a level indicator or a level-number, followed by a space, the name associated with the level indicator or level-number, and a sequence of independent description clauses.

Each clause except the last may be terminated by either a semicolon or a comma. The last must be terminated by a period.

The division header must begin in Area A (the eighth through eleventh character positions of each record). It consists of the words, DATA DIVISION, followed by a period.

FILE SECTION

The File Section defines the structure of any sequential, indexed, relative, random-access, or sort/merge file appearing in your program.

Each file is defined by a file description entry and one or more record descriptions.

A file description entry always begins with the letters SD (for sort/merge files), or FD (for any other type of file as mentioned above) in area A. It is followed by a space, the name of the file being described, and a sequence of input/output description clauses.

An FD file description furnishes information concerning the physical structure, identification, and record names pertaining to any type of file except a sort/merge file.

An SD file description provides information about the size and names of the data records in a file to be sorted or merged. Because you cannot control label procedures, and since the blocking for the internal storage of sort/merge records is controlled by the SORT and MERGE operations, the only two clauses allowed in a file description for a SORT/MERGE file are the RECORD CONTAINS and DATA RECORD IS clauses.

A record description always begins with a level-number which, in the File Section, may be any number from 1 to 49, or the numbers 66 and 88. Level numbers of 66 and 88 have special usages associated with them, as discussed in Section IV. Record description entries are discussed in Section IX.

Note that record description entries can be used in every section, and must be used in the File Section, following each FD or SD level description entry appearing in that section.

FILE SECTION Format

The complete format of the File Section is:

```
FILE SECTION.
FD file-name
[ , BLOCK CONTAINS [ integer-1 TO ] integer-2 { RECORDS    } ]
                                              { CHARACTERS }

                                 { F }
[ ; RECORDING MODE IS            { V } ]
                                 { U }
                                 { S }

[ ; RECORD CONTAINS [ integer-3 TO ] integer-4 CHARACTERS ]
data-name-9 [ , data-name-10 ] ...
  ; LABEL { RECORD IS    } { STANDARD }
          { RECORDS ARE  } { OMITTED  }
[ ; VALUE OF label-info-1 IS { data-name-1 }
                             { literal-1   }
     [ , label-info-2 IS { data-name-2 } ] ... ]
                         { literal-2   }
[ ; DATA { RECORD IS   } data-name-3 [ , data-name-4 ] ... ]
         { RECORDS ARE }
[ ; LINAGE IS { data-name-5 } LINES [ , WITH        { data-name-6 } ]
              { integer-5   }         FOOTING AT    { integer-6   }
     [ , LINES AT TOP { data-name-7 } ] [ , LINES     { data-name-8 } ] ]
                      { integer-7   }     AT BOTTOM   { integer-8   }
[ ; CODE-SET IS alphabet-name ]
{ record-description-entry } ...
SD file-name
[ ; RECORD CONTAINS [ integer-1 TO ] integer-2 CHARACTERS ]
[ ; DATA { RECORD IS   } data-name-1 [ , data-name-2 ] ... ] .
         { RECORDS ARE }
{ record-description-entry } ...
```

The File Section must begin in Area A with the words, FILE SECTION, followed by a period. This header is followed, in turn, by a level indicator (either FD or SD) and the name of the file being described. One or more record description entries must follow each FD or SD level description entry.

The clauses which follow a level indicator are discussed in Section IX.

Several of the clauses do not apply to sort/merge files (that is, to SD level indicator entries). Those that do not apply are noted as they are described.

WORKING STORAGE SECTION

The Working-Storage Section consists of a section header (WORKING-STORAGE SECTION.), and one or more data description entries for non-contiguous data items, as well as record description entries. The general format of the WORKING-STORAGE Section is:

Working Storage Section Format

WORKING-STORAGE SECTION.

[[*77-level-description-entry* | *record-description-entry*] ....]

Each data description entry within Working Storage allocates memory space for the data item and associates a data-name with the item.

You may use the Working Storage Section to assign initial values to data items, define report headings, set up tables with initial values, define counters and accumulators, and so forth.

Level 77 and other data description entries are discussed in the following section.

LINKAGE SECTION

The Linkage Section consists of a header, "LINKAGE SECTION.", and one or more data description entries for non-contiguous data items, as well as record description entries.

The Linkage Section in a program is meaningful if and only if the object program is to be called from another object program through a CALL statement, and the CALL statement in the calling program contains a USING phrase. See Section II, Program Modules, for more information on how calling and called programs operate.

The Linkage Section is used to describe the data that is available from the calling program, but is to be used in both the calling and the called program.

No space is allocated in the program containing a Linkage Section entry for the data items described in such an entry. Procedure Division references to these data items are resolved at object time by equating the reference in the called program to the location used in the calling program.

In the case of index-names, no such correspondence is established. Index-names are always local to their programs. Thus, such names in the called and calling program always refer to separate indices. If index-names are to be passed between programs, they should first be saved in a data-item described in the Working Storage Section as an index data-item. A SET statement in the Procedure Division accomplishes this. Next, the index data-item should be passed to the called program by the USING option of the CALL statement in the Procedure Division. Once the index-data-item is passed, a SET statement in the called program can be used to set the index value to the the index name declared in the Linkage Section of the called program.

Of course, data items defined in the Linkage Section of the called program may be referenced within the Procedure Division of the called program only if they are specified as operands of the USING phrase of the Procedure Division header or are subordinate to such operands, and the object program is under the control of a CALL statement that specifies the USING phrase.

All data description clauses may be used in the Linkage Section, with the restriction that a VALUE clause can only be used in a condition-name (level 88) entry.

The VALUE clause cannot be used in any other type of entry because no memory space is allocated for Linkage Section data items.

LINKAGE SECTION Format

```
LINKAGE SECTION.
-------  -------

[ [ 77-level-description-entry ]   ........ ]
[ [   record-description-entry ]            ]
```

Both types of descriptions are discussed in Section IX.

DATA DIVISION CLAUSES

SECTION IX

There are two distinct sets of clauses in the Data Division. They are file description clauses, and data description clauses.

File description clauses apply only to data files (FD level entries) and sort/merge files (SD level entries).

Data description clauses can be used in any of the three sections allowed by COBOL II/3000 within the Data Division. They must, in fact, be used in conjunction with each SD or FD level description entry.

File description clauses are discussed first, followed by data description clauses.

FILE DESCRIPTION CLAUSES

Each clause described on the following pages, if used, must be part of an FD or SD level description entry. However, several of the clauses do not apply to sort/merge files (that is, to SD level indicator entries). Those that do not apply to SD level entries are noted as they are described.

All file description clauses are shown on the following page, and are discussed in alphabetical order following the descriptions of the FD and SD level indicators.

A file description clause is terminated when a record description entry, a new FD or SD level indicator, a section header, or the Procedure Division header is encountered.

File Description Clause Formats

```
FD file-name

  [ , BLOCK CONTAINS [ integer-1 TO ] integer-2 { RECORDS    } ]
                                                { CHARACTERS }

  [ ; RECORDING MODE IS { F } ]
                        { V }
                        { U }
                        { S }

  [ ; RECORD CONTAINS [ integer-3 TO ] integer-4 CHARACTERS ]
  data-name-9 [ , data-name-10 ] ...

  ; LABEL { RECORD IS    } { STANDARD }
          { RECORDS ARE  } { OMITTED  }

  [ ; VALUE OF label-info-1 IS { data-name-1 } ]
                               { literal-1   }

      [ [ , label-info-2 IS { data-name-2 } ] ... ]
                            { literal-2   }

  [ ; DATA { RECORD IS   } data-name-3 [ , data-name-4 ] ... ]
           { RECORDS ARE }

  [ ; LINAGE IS { data-name-5 } LINES [ , WITH                { data-name-6 } ] ]
                { integer-5   }       [     FOOTING AT        { integer-6   } ]

     [ , LINES AT TOP { data-name-7 } ] [ , LINES              { data-name-8 } ] ]
                      { integer-7   } ] [     AT BOTTOM        { integer-8   } ] ]

  [ ; CODE-SET IS alphabet-name ]

  { record-description-entry } ... ] ...

SD file-name

  [ ; RECORD CONTAINS [ integer-3 TO ] integer-4 CHARACTERS ]

  [ ; DATA { RECORD IS   } data-name-3 [ , data-name-4 ] ... ] .
           { RECORDS ARE }
```

Level Indicators

FD LEVEL INDICATOR - FOR DATA FILE DESCRIPTIONS

The FD level indicator names the data file being described. It must be the first clause in a data file description entry. The data file description clause has the following format:

FD ***file-name***

Where

FD	indicates that the clauses which follow are data file description clauses.
file-name	is the name of the data file being described.

The characters, FD, must begin in area A. These characters are followed by a space, and the name of the data file being described. Following the name, several clauses, mostly optional, are used to describe the file.

SD LEVEL INDICATOR - FOR SORT FILE DESCRIPTIONS

An SD level indicator names the file to be sorted or merged. It must be the first entry in a sort/merge file description entry.

The format of the SD level indicator is:

SD ***sort-file-name***

Where

SD	indicates that the clauses which follow are used to describe a file to be sorted or (and) merged.
sort-file-name	is the name of the file being described.

The characters, SD, must begin in area A (columns 8 through 11 of a record). They must be followed by a space, and the name of the file being described. The name may then be followed by a RECORD CONTAINS, or a DATA RECORDS clause, and must be followed by at least one record description entry.

BLOCK CONTAINS CLAUSE

The BLOCK CONTAINS clause allows you to specify the blocking factor of the file being described.

This clause should be used if the actual blocking factor of the file being described cannot be determined by MPE.

For example, this clause is optional when the physical record contains exactly one complete logical record, or, for sequential files only, when the physical device associated with the file is a unit record device (such as a card reader).

When the BLOCK CONTAINS clause is omitted, the default MPE blocking factor is automatically assigned.

This clause does not apply to sort/merge files (that is, SD level descriptions).

BLOCK CONTAINS Clause Format

```
[ , BLOCK CONTAINS [ integer-1 TO ] integer-2 { RECORDS    } ]
                                              { CHARACTERS } 
```

Where

Integer-1 is optional. It must be positive, and refers to the minimum blocking factor size, or to the minimum size of the physical record, depending upon whether the keyword, RECORDS or CHARACTERS, is used. Due to the way in which MPE determines file attributes, this phrase, if used, is treated as a comment.

Integer-2 is required and must be positive. If used in conjunction with *integer-1*, it specifies the maximum size of the physical record; used alone, however, it specifies the exact size of the physical record, or the exact blocking factor.

RECORDS specifies that the physical record size of the file is determined by its blocking factor.

CHARACTERS when specified, is used to determine the blocking factor by dividing the value of *integer-2* by the logical record size. See the RECORD CONTAINS clause.

When the word CHARACTERS is used, the physical record size should be specified as a multiple of the maximum logical record size. Note that this logical record size must include any slack bytes generated by the compiler. See the SYNCHRONIZED CLAUSE, presented later in this section.

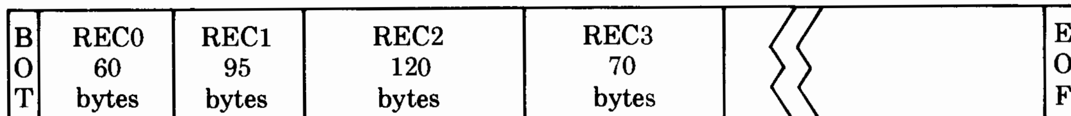
If logical records of differing size are grouped into one physical record, they are treated differently, according to the file's organization:

- Sequential or indexed file —The size of a logical record is variable, and is equal to the size of the record currently being acccessed.

Random and relative files require fixed length records.

To illustrate the use of the BLOCK CONTAINS clause:

1. A magnetic tape file contains variable length logical records with a maximum of 120 characters, blocked with a minimum of four logical records per physical record. There is a maximum of 480 characters (bytes) per physical record, and a minimum of 60 bytes per physical record.



In this case, you can use either

```
BLOCK CONTAINS 4 RECORDS.
```

or

```
BLOCK CONTAINS 60 TO 480 CHARACTERS.
```

NOTE

The figures specified above are only used as estimates by MPE. The actual blocking factor varies depending on the logical records, and the physical record size for variable records contains control information. Thus, it is larger than specified in the BLOCK CONTAINS clause. This applies to relative files as well. Refer to the MPE Commands Reference Manual for more information on blocking factors.

2. A random access disc file contains fixed length logical records of 206 bytes (103 words) each. Thus, there are 50 unused bytes per sector. To minimize this waste, a blocking factor of 6 records can be used:

```
BLOCK CONTAINS 6 RECORDS.
```

or

```
BLOCK CONTAINS 1236 CHARACTERS.
```

3. A serial access disc file contains variable length logical records, ranging from 256 to 2560 bytes per record. The blocking factor is 10.

   You can use either

```
BLOCK CONTAINS 256 TO 2560 CHARACTERS.
```

or

```
BLOCK CONTAINS 10 RECORDS.
```

CODE-SET CLAUSE

The CODE-SET clause specifies whether you are using ASCII or EBCDIC character codes to represent data stored in sequential files.

This clause is optional, with ASCII being the default if it is not specified. Also, this clause applies only to sequential files.

CODE SET Clause Format

[CODE-SET IS ***alphabet-name***]

Where

alphabet-name is a previously defined name related to either EBCDIC, STANDARD-1, or NATIVE. It must have been specified in the alphabet-name clause of the SPECIAL-NAMES paragraph in the Environment Division.

If the CODE-SET clause is specified, *alphabet-name* indicates the character code convention used to represent the data on the related file. It further indicates the conversion routine to be used in translating the data into ASCII (when reading it) and translating data back into its original character code (when writing it from your program).

Note that records written to an EBCDIC file are placed in an intermediate area, and then translated. This preserves the record in the record area. However, this intermediate area increases the size of your data stack.

When the CODE-SET clause is used for a file, all data in that file must be described as USAGE IS DISPLAY, and any signed numeric data must be described with the SIGN IS SEPARATE clause.

Note that FCOPY, an HP utility program, can be used to translate EBCDIC files containing records with elements whose usage is other than DISPLAY. See the *FCOPY Reference Manual*, part number 03000-90064.

DATA RECORDS CLAUSE

For any type of file (random, sequential, sort/merge, and so forth) the DATA RECORDS clause serves only to document the names of data records associated with a file. This clause is, therefore, optional for either an FD or SD level file description.

DATA RECORDS Clause Format

```
[ ; DATA { RECORD IS    } data-name-1  [ , data-name-2 ] ... ]
         { RECORDS ARE  }
```

Where

Data-name-1 to ***data-name-n***	Are the names of data records. Each name used in this clause must be used in a 01 level description following the file description in which it appears.

Use of more than one data-name in this clause indicates that the file contains more than one type of data record. They might, for instance, be of different size or format.

LABEL RECORD CLAUSE

The LABEL RECORD clause specifies whether one or more labels exist on the file, and, optionally, the names of the records containing the label.

In ANSI COBOL '74, this clause is required for every file description entry using an FD level-indicator; it does not, however, apply to sort/merge files. The clause is optional in COBOL II/3000.

LABEL RECORD Clause Format

```
[          { RECORD IS    }  { STANDARD                                  } ]
[ ; LABEL  {              }  { OMITTED                                   } ]
[          { RECORDS ARE  }  { data-name-1 [ , data-name-2 ] ...         } ]
```

Where

OMITTED specifies that no explicit labels exist for the file or the device to which the file is assigned. If this parameter is used, the VALUE OF clause is not applicable.

STANDARD specifies that labels exist for the file or the device to which it has been assigned, and that the labels conform to the MPE label specification.

Data-name-1* to *Data-name-n are names of label records which must be described in a record description entry associated with the file. These records must not appear in the DATA RECORDS clause associated with the file. Use of this option indicates that user labels, as well as standard labels, are to be processed. All Procedure Division references to these names, or to any subordinate items, must appear within USE procedures. Label records for *all* files share the same area of memory.

With COBOL II/3000, it does not matter whether you specify that labels are STANDARD or OMITTED, since the MPE file system processes standard labels before making the associated file available to your COBOL program. The data-name form is a COBOL II/3000 extension to ANSI COBOL '74.

LINAGE CLAUSE

The purpose of the LINAGE clause is to describe the format of a logical page. It is used in conjunction with sequential files opened for output. Although there is not necessarily any relation between a logical and a physical page, it is advisable (particularly when writing a logical page to the line printer) to consider the size of a physical page when you are defining a logical one.

The LINAGE clause applies only to sequential files. It has no meaning for relative, random, indexed, or sort/merge files.

Its use is optional with sequential files, but it must be used if you intend to write records to the file using the END-OF- PAGE (or EOP) phrase of the WRITE statement.

A logical page consists of a top margin, page body, footing area, and bottom margin.

Within a file, logical pages are contiguous, with no spaces separating them.

Figure 9-1 shows the concept of a logical page.

LINAGE Clause Format

```
[ ; LINAGE IS { data-name-1 } LINES [ , WITH          { data-name-2 } ]
              { integer-1   }         FOOTING AT    { integer-2   } ]

    [ , LINES AT TOP { data-name-3 } ] [ , LINES     { data-name-4 } ] ]
    [                { integer-3   } ] [   AT BOTTOM { integer-4   } ] ]
```

Where

data-name-1 through ***data-name-3***	each reference an elementary unsigned integer data item.
integer-1	is greater than 0.
integer-2	is greater than or equal to 0 and not greater than integer-1.
integer-3 and ***integer-4***	are each greater than or equal to 0.

The LINAGE clause uses *data-name-1* or *integer-1* to define the number of lines in the page body. The page body is the area of the logical page in which lines can be written or spaced. Since a page size is being defined, *integer-1* (or the value associated with *data-name-1*) must be greater than 0.

FOOTING PHRASE

The FOOTING phrase is optional. If specified, however, it uses *integer-2* or *data-name-2* to define the FOOTING AREA of the page body.

The entire page body can be the footing area. That is, *integer-2* (or, the value of *data-name-2*) may be 1, in which case the footing area is all of the page body.

The footing area is used in conjunction with the END-OF-PAGE phrase of the WRITE statement. It signifies that the end of the logical page has been reached.

If you do not use the, FOOTING phrase, then the only way that an end of page condition can occur is for a WRITE statement to attempt to write a record beyond the end of the logical page body. (That is, when a page overflow condition exists.)

LINES AT TOP AND LINES AT BOTTOM PHRASES

The LINES AT TOP and LINES AT BOTTOM phrases are optional. They are used to specify a top margin and a bottom margin, respectively, for the logical page. If neither phrase is used, the margins are assumed to be 0.

If THE LINES AT TOP phrase is specified, it uses *integer-3* or *data-name-3* to specify the number of lines in the top margin.

If THE LINES AT BOTTOM phrase is specified, it uses *data-name-4* or *integer-4* to specify the number of lines in the bottom margin.

The top and bottom margins are distinct from the page body; thus, no data may be written into them.

USE OF DATA NAMES VERSUS USE OF INTEGERS

The use of integers in a LINAGE clause allows for less flexibility than does the use of data names.

When an integer (either *integer-1*, *integer-2*, or *integer-3*) is specified, it is used when the file associated with the LINAGE clause is opened for output. This value is used for every logical page written for the file, and cannot change during a particular execution of the program in which it appears.

The values of data names, on the other hand, can vary during the execution of a program. Thus, the values are checked and used not only when the associated file is opened for output, but also whenever a WRITE statement containing an ADVANCING PAGE phrase is executed, or a WRITE statement is executed and a page overflow condition occurs.

Taking each of these cases in turn:

- When the file is opened for output, the current values of *data-name-1*, *data-name-2*, *data-name-3*, *data-name-4* are used to define their associated sections of the FIRST logical page only.
- When a WRITE statement is executed, and the ADVANCING PAGE phrase is activated, the current values of *data-name-1*, *data-name-3*, and *data-name-4* are used to define the page body, top and bottom margins of the next logical page.
- If a footing area has been defined, the ADVANCING PAGE phrase is activated when the WRITE statement in which it appears attempts to write data into the footing area. In this case, the data is written into the footing area of the current logical record and the current value of *data-name-2* is then used to define the footing area for the next logical page.
- When a WRITE statement is executed and a page overflow condition occurs, thus forcing an end-of-page condition, the current values of *data-name-1*, *data-name-3*, and *data- name-4* are used to define their associated parts of the next logical page.

This type of end-of-page condition implies that either the value of *data-name-2* is the same as that of *data-name-1*, or that a footing area was not defined (the two are equivalent).

In either case, the data to be written is placed in the first available line of the next logical record (depending upon whether the BEFORE or AFTER ADVANCING phrase was used in the WRITE statement).

If a footing area has been defined, the current value of *data-name-2* is then used to define the footing area of this logical record.

LINAGE-COUNTER

Any time a LINAGE clause is specified for a file, a LINAGE-COUNTER is supplied for the file.

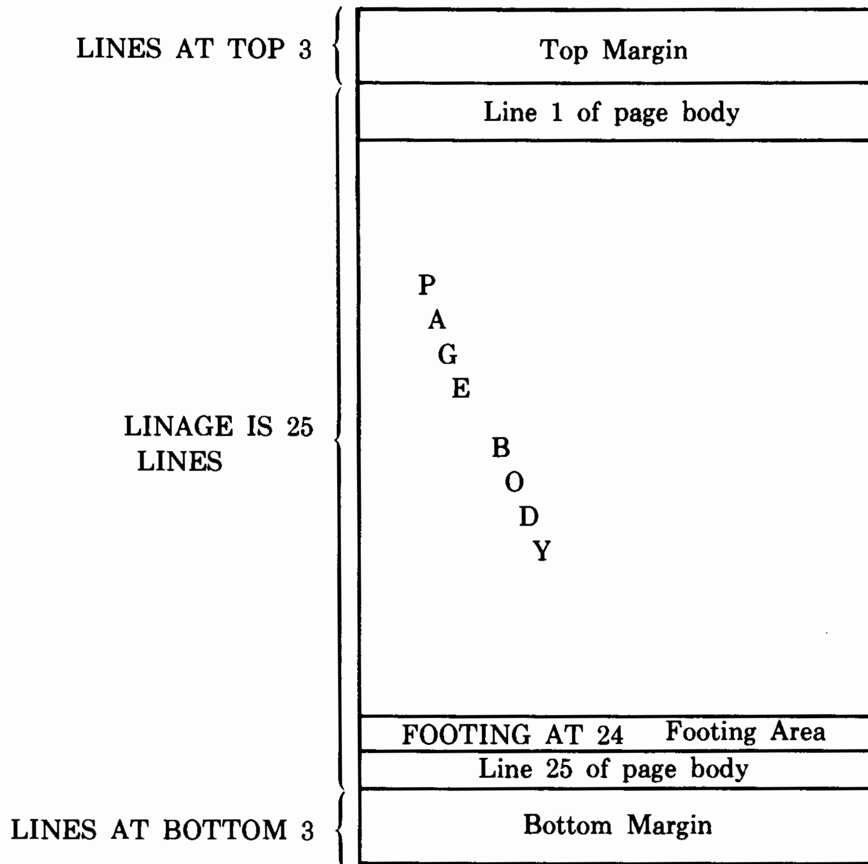
Since you can have more than one file whose description contains a LINAGE clause, you must qualify the LINAGE-COUNTER of each file by using the file name.

The value of a LINAGE-COUNTER at any given time is the current line number of the associated page body. This value ranges from 1, for the first line of a page body, to *integer-1* (or the value of *data-name-1*). You can reference a LINAGE-COUNTER in the Procedure Division, but cannot change it.

Each time a record is written to a logical page, the associated LINAGE-COUNTER is incremented according to the following rules:

- When the file associated with LINAGE-COUNTER is first opened, LINAGE-COUNTER is set to 1.
- If the ADVANCING phrase of the WRITE statement is not specified, LINAGE-COUNTER is incremented by 1 when the WRITE statement is executed.
- If the ADVANCING phrase is used with a WRITE statement, and
    1. is of the form, ADVANCING integer or ADVANCING *identifier-2*, LINAGE-COUNTER is incremented by *integer* (or the value of *identifier-2*) when the WRITE statement is executed.
    2. is of the form ADVANCING PAGE, LINAGE-COUNTER is reset to 1.
- If a new logical page is to be written upon, LINAGE-COUNTER is reset to 1.

Figure 9-1.
EXAMPLE OF THE LINAGE CLAUSE AND ITS LOGICAL REPRESENTATION



LINAGE IS 25 LINES, WITH FOOTING AT 24,

LINES AT TOP 3, LINES AT BOTTOM 3

RECORD CONTAINS CLAUSE

The RECORD CONTAINS clause specifies the size, in characters, of data records in a file. Since each data record of a file is completely defined in a record description entry, this clause is optional for any file description entry.

RECORD CONTAINS Clause Format

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

Where

integer-1	is optional, provided all data records in the file have the same size. It represents the minimum number of characters (bytes) in the smallest data record of the file.
integer-2	used by itself represents the exact number of characters in each record of the file. Thus, it can be used by itself only if all data records are of the same length. Used in conjunction with *integer-1*, *integer-2* specifies the maximum number of characters (bytes) in the largest record of the file.

The size of a record is determined by taking the sum of the numbers of all characters in all fixed length elementary items, and adding to that sum the maximum number of characters in any variable length item subordinate to the record.

This sum may differ from the actual size of the record because of slack bytes. See the SYNCHRONIZED and USAGE clauses appearing later in this section.

RECORDING MODE CLAUSE

This clause does not apply to sort/merge files.

The RECORDING MODE clause specifies how logical records are contained in the file, and whether or not the logical record being read or written spans more than one physical record (generally because of hardware restraints or for I/O efficiency).

RECORDING MODE Clause Format

```
[ ; RECORDING MODE IS { F } ]
                      { V }
                      { U }
                      { S }
```

Where

F specifies fixed length logical records. This implies that no OCCURS DEPENDING on clause can be associated with any record description entry for the file. Also, if more than one record description entry is supplied for the file, all record lengths calculated from the record descriptions must be the same. This option is the only one that is valid for random-access and relative files.

V specifies variable length logical records.

U specifies undefined length logical records. This kind of file cannot be blocked. Thus, the BLOCK CONTAINS clause need not be used for this kind of file.

S enables the MULTIRECORD (or more accurately, "multiblock") option. This option allows the reading or writing of a single logical record across more than one physical record.

In this case, an implied F is also supplied by COBOL.

Logical records are contained in files as either fixed, variable, or undefined in length.

A fixed length record file contains logical records whose lengths are all the same.

A variable length record file contains logical records whose lengths may vary. In such a file, each record is preceded by a one-word byte count, which specifies the length of that particular record.

An undefined length record file contains logical records of undetermined length. In such a file, each logical record is equivalent to one physical record, and a physical record is as long as the longest possible logical record in the file.

To clarify the case of logical records spanning more than one physical record, assume you want to read logical records of 128 characters each from a card reader.

Each card represents a physical record of 80 characters. Thus, to read one logical record, you must read two physical records.

in such a case, you must specify the recording mode as equal to S, the MPE multi-record option.

VALUE OF CLAUSE

The VALUE OF clause does not apply to sort/merge files.

The VALUE OF clause allows you to access existing files on labeled tapes, or to create a new labeled tape. A label contains identification, whether the label is in IBM or ANSI standard format, the expiration date of the file protection, and the position of the file on the tape.

VALUE OF Clause Format

```
[ ; VALUE OF label-info-1 IS { data-name-1 }
                             { literal-1   }

     [ , label-info-2 IS { data-name-2 } ] ... ]
                         { literal-2   }
```

Where

Label-info-1 to ***Label-info-n***	are each one of the fields, VOL, LABELS, SEQ, or EXDATE. Each of these fields is described in table 9-1 on the following pages.
Data-name-1 to ***Data-name-n***	must be described in the Working-Storage section. These names may be qualified, but cannot be subscripted, indexed, or described with the USAGE IS INDEX clause in a data description. Each is used to specify the value of the associated label-info entry. The possible values are shown in table 9-1 on the following pages.
Literal-1 to ***Literal-n***	are COBOL literals or are figurative constants.

The data-name associated with VOL can specify a data item of any category, but must consist of a maximum of six characters or digits. The data-name associated with SEQ can also specify a data item of any category, but must consist of a maximum of four digits or characters. The data-names associated with LABELS and EXDATE must name alphanumeric data items, and must be three and eight characters long, respectively.

Table 9-1.

VALUES OF THE LABEL INFO AND DATA NAME PARAMETERS IN THE VALUE OF CLAUSE

label-info-n	MEANING	data-name-n or literal-n
VOL	volume identification	Any combination of one to six characters from the set A through Z, and 0 through 9.
LABELS	ANSI standard or IBM format	ANS or IBM
SEQ	Relative position of file on a magnetic tape	0 to 9999 NEXT, or ADDF
EXDATE	Date when file may be written over. Until that time, the file is protected.	Date, in the form month/day/year default is 00/00/00

See the FOPEN intrinsic in the *MPE Intrinsics Reference Manual*, part number 30000-90010, for more information on these values.

To illustrate the VALUE OF clause:

```
                .
                .
                .
    DATA DIVISION.
                .
                .
                .
    FD TAPEFL
                .
                .
    LABEL RECORDS ARE LAB1, LAB2;
    VALUE OF VOL IS "JTAPE1", LABELS IS "ANS",
             SEQ IS 10, EXDATE IS "2/25/80";
                .
                .
    FD NEW-TAPE
                .
                .
    VALUE OF VOL IS "JTAPE2", LABELS IS "ANS",
             SEQ IS "ADDF", EXDATE IS "2/25/80";
                .
                .
                .
    PROCEDURE DIVISION.
        DISPLAY "PLEASE MOUNT NEW TAPE FOR JTAPE2" UPON CONSOLE.
        OPEN OUTPUT NEW-TAPE, INPUT TAPEFL.
                .
                .
                .
```

Assuming that NEW-TAPE names a new file, when the OPEN statement above is executed, the information given in the VALUE OF clause for it is used to write a label for the tape volume.

When TAPEFL is opened, the specification of 10 for the SEQ value causes the tape to automatically be placed at the beginning of the tenth file on the volume named JTAPE1.

Note that a message requesting that the volumes, JTAPE2, and JTAPE1 be mounted is displayed on the operator's console. Since JTAPE2 is a new volume, the DISPLAY statement above was used to tell the operator that JTAPE2 does not already exist.

DATA DESCRIPTION ENTRIES

A data description entry is composed of a level number followed by a data name, and then followed by a set of data clauses.

The level numbers can be 01 to 49 for record description entries, 77 for unrelated data items, 66 for alternative groupings of elementary items in the preceding record description entry, and 88 for condition names.

A level number is required for each data description entry, and must be the first element of such an entry.

The level numbers 01 and 77 must begin in Area A (columns 8 through 11 of a source record). Other level numbers may begin in either Area A or Area B.

All data description entries are discussed in the following pages, beginning with level 77 entries.

77-Level Description Entries

Noncontiguous data items are data items or constants which are not subdivided, and bear no hierarchical relationship to one another. That is, noncontiguous data items are unrelated data items.

Such items are always defined only in the Working Storage and Linkage Sections, and have level numbers of 77. Recall, however, that those items defined in the Linkage Section do not have any storage allocated for them.

Each name used for a noncontiguous data item must be unique since it cannot be qualified.

Examples of level 77 entries:

```
WORKING-STORAGE SECTION.

    77 COUNTER              PICTURE 9(4) VALUE 0.
    77 MID-TOTALS           PICTURE 9(6)V99 VALUE 0.
```

COUNTER is an accumulator, and MID-TOTALS is an intermediate storage variable. Since they are not subdivided, and they have no immediate relationship to any other data items, they are described using level 77 entries.

Record Description Entries

At least one record description entry must follow each FD or SD file description entry. This discussion of record description entries is applicable to either type of file description, as well as for the Linkage and Working Storage Sections.

A record description consists of one or more data description entries.

Associated with each data description entry is a level number chosen from the set 01 to 49, 66, or 88. The level number for the first data description entry of any record description must be 01 (or simply 1). Succeeding level numbers of data description entries for the same record description may range from 01 to 49, or may be 66. If, however, multiple 01 entries are used for a given level indicator (FD, or SD) they represent implicit redefinitions of the same area. The 02 to 49 level numbers are used to specify subsets of the characters of a record.

The level number 66 is used only when the data description uses the RENAMES clause to regroup data items (see the RENAMES clause, below).

The level number 88 is used only when the data description is of a condition name. It is always associated with a VALUE clause (see CONDITION NAMES-FORMAT 3 OF DATA DESCRIPTION ENTRIES, later in this section).

Data names subordinate to record names can be nonunique, provided they can be made unique by qualification.

DATA DESCRIPTION ENTRY FORMATS

There are three general formats for data description entries. These formats are shown below.

Format 1

```
level-number { data-name-1 }
             { FILLER      }

  [ ; REDEFINES data-name-2 ]

  [ ; { PICTURE }  IS character-string ]
      { PIC     }

  [ ; [ USAGE IS ] { COMPUTATIONAL-3 } ]
                   { COMP-3          }
                   { COMPUTATIONAL   }
                   { COMP            }
                   { DISPLAY         }
                   { INDEX           }

  [ ; [ SIGN IS ] { LEADING  }  [ SEPARATE CHARACTER ] ]
                  { TRAILING }

  [ ; 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 ] ... ]

  [ ; { SYNCHRONIZED } [ LEFT  ] ]
      { SYNC         } [ RIGHT ]

  [ ; { JUSTIFIED }  RIGHT ]
      { JUST      }

  [ ; BLANK WHEN ZERO ]

  [ ; VALUE IS literal ] .
```

Format 2

66 *data-name-1* ; **RENAMES** *data-name-2* [{ **THROUGH** / **THRU** } *data-name-3*] .

Format 3

88 *condition-name* ; { **VALUE** IS / **VALUES** ARE } *literal-1* [{ **THROUGH** / **THRU** } *literal-2*]

[, *literal-3* [{ **THROUGH** / **THRU** } *literal-4*]] ... .

General Rules

In the first format, the level number may be any number from 01 to 49, or 77. It cannot be 66 or 88.

The data-name or FILLER clause must immediately follow the level number, and the REDEFINES clause, if used, must immediately follow the data-name-1 clause.

Except for these two conditions, all other clauses, if used, may be written in any order.

The PICTURE clause must be used for every elementary item except an index data item. A PICTURE clause must not, in fact, be used for an index data item.

Data elements and constants bearing a definite hierarchic relationship to one another must be grouped into records.

The initial value of any data item except an index data item is specified by using the VALUE clause in the description of that item. Since the VALUE clause does not apply to index data items, the initial value of any such item is unknown.

You may not use the SYNCHRONIZED, PICTURE, JUSTIFIED, or BLANK WHEN ZERO clauses for any but elementary data items.

Except for the data-name or filler clause, which is discussed first, all other data description clauses are discussed in alphabetical order on the following pages.

DATA-NAME OR FILLER CLAUSE

A DATA-NAME clause specifies the name of the data being described. The word FILLER implies that you are specifying an elementary item of the logical record being described that cannot be referenced explicitly.

In the File, Working Storage, and Linkage Sections, a data-name or the word FILLER, must be the first word following the level number in each data description entry.

Data-Name or FILLER Clause Format

level-number { *data-name-1* | FILLER }

Where

data-name must be a valid user-defined COBOL word.

Although you may not refer to a FILLER item explicitly, the keyword, FILLER, may be used as a conditional variable (format 3) because the use of it in this manner does not require explicit reference to the FILLER item, but to its value.

BLANK WHEN ZERO CLAUSE

The BLANK WHEN ZERO clause is optional. If used, however, it refers only to a numeric or numeric edited elementary data item.

BLANK WHEN ZERO Clause Format

BLANK WHEN ZERO

The BLANK WHEN ZERO clause causes a numeric or numeric edited data item to be filled with spaces when the value of the data item is zero.

When this clause is used with a numeric data item, the category of the data item is considered to be numeric edited.

NOTE

This clause cannot be used for a numeric edited data item whose PICTURE uses asterisks for zero suppression and replacement.

JUSTIFIED CLAUSE

The JUSTIFIED clause is optional. It allows you to right-justify alphabetic or alphanumeric data items. It cannot be used with numeric or edited data items, and only applies to elementary data items being used to receive data.

JUSTIFIED Clause Format

```
[ ; { JUSTIFIED }  RIGHT ]
    { JUST      }
```

Where

JUST is an abbreviation for JUSTIFIED.

Data is moved from a sending data item to the right justified receiving data item starting with the right most character of the sending data item. This right most character is placed in the right most character of the receiving data item. The next right most data item of the sending data item is then moved to the next right most character of the receiving data item. This process continues until either all of the sending data item has been moved, or the receiving data item is full. Note that a space in the sending data item is considered a valid character, no matter where it is within the sending data item. That is, spaces are not stripped from the sending item, even if they are in the right most positions of the sending item.

When a receiving data item is described using this clause, and the sending data item is larger than the receiving item, the left most characters are truncated.

When the receiving data item is longer than a sending item, the data is aligned at the right most character position in the receiving field, and unused characters to the left are filled with spaces.

Of course, if the JUSTIFIED clause is not used, standard rules for aligning data within an elementary item are used.

To illustrate the effect of the JUSTIFIED clause:

Sending data item: HEWLETT-PACKARD COBOL II/3000ΔΔ

Receiving data item: `01 INFIRST  PIC X(35)  JUSTIFIED RIGHT.`

Resulting data item: ΔΔΔΔHEWLETT-PACKARD COBOL II/3000ΔΔ

Assuming the same sending data item, but using a new receiving data item:

Receiving data item: `01 NEWIN PIC X(20) JUSTIFIED RIGHT.`

Resulting data item: KARD COBOL II/3000ΔΔ

OCCURS CLAUSE

The OCCURS clause is used to define one, two, and three dimensional tables. Its use eliminates the need for separate entries to describe repeated data items, and provides information required for the application of subscripts or indices.

See Section X, Tables, for a description of the formation and use of tables.

This clause has two general formats as shown below.

Format 1:

```
[ ; OCCURS      integer-2 TIMES

    [ { ASCENDING  }  KEY IS data-name-4  [ , data-name-5 ] ... ] ...
      { DESCENDING }

    [ INDEXED BY index-name-1 [ , index-name-2 ] ... ]                ]
```

Format 2:

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

    [ { ASCENDING  }  KEY IS data-name-2  [ , data-name-3 ] ... ] ...
      { DESCENDING }

    [ INDEXED BY index-name-1 [ , index-name-2 ] ... ]                ]
```

Each format is discussed separately following the discussion of general rules for the OCCURS clause.

General Rules

The OCCURS clause cannot be specified in a data description entry having a 01, 66, 77, or 88 level-number.

It cannot be used in a description entry for an item whose size is variable. The size of an item is variable if the data description of any subordinate item contains FORMAT 2 of the OCCURS clause. Said in a different way, this restriction means that no OCCURS clause using the DEPENDING ON phrase can be used in the description of an item subordinate to an item which also uses either format of the OCCURS clause.

To illustrate,

```
01  ROAD.
    02  SURFACE OCCURS 1 TO 12 TIMES
        DEPENDING ON SIZER
       03 SIDE OCCURS 10 TIMES PIC X(9).
01  ROAD
    02  SURFACE OCCURS 10 TIMES
       03 SIDE OCCURS 4 TIMES PIC X(9).
```

are both allowed, whereas

```
01  ROAD.
    02  SURFACE OCCURS 10 TIMES
        03 SIDE OCCURS 1 TO 8 TIMES
            DEPENDING ON SIZER PIC X(9).
```

and

```
01 ROAD.
    02  SURFACE OCCURS 1 TO 10 TIMES
            DEPENDING ON SIZER
        03 SIDE OCCURS 1 TO 8 TIMES
            DEPENDING ON SIZEUP PIC X(9).
```

are both unacceptable.

The name of the date-item being described must be either subscripted or indexed whenever it is referred to in any other than the SEARCH or USE FOR DEBUGGING statements. Also, if the name of the data-item being described is the name of a group item, then all data names belonging to the group must be subscripted or indexed whenever they are used as operands, except as the object of a REDEFINES clause.

Except for the OCCURS clause itself, all data description clauses associated with an item whose description includes an OCCURS clause apply to every occurrence of the item being described.

OCCURS - FORMAT 1

The first format of the OCCURS clause,

```
[ ; OCCURS        integer-2 TIMES

    [ { ASCENDING  }  KEY IS data-name-4  [ , data-name-5 ] ... ] ...
      { DESCENDING }

    [ INDEXED BY index-name-1 [ , index-name-2 ] ... ]              ]
```

is used to define a table of fixed length data items.

Where

integer-2 specifies the exact number of occurences of the item being described.

KEY IS phrase indicates that the repeated data is arranged in ASCENDING or DESCENDING order according to the values contained in *data-name-2*, *data-name-3*, and so forth.

data-name-2 must either be the name of the entry containing the OCCURS clause, or the name of an entry subordinate to the entry containing the OCCURS clause.

***data-name-3*, *data-name-4*,** and so forth must be subordinate to the group item which is the subject of the OCCURS clause.

INDEXED BY phrase specifies one or more index-names to be used when reference to the subject of this entry or items subordinate to it is done by indexing.

***index-name-1*, *index-name-2*,** and so forth are not defined elsewhere in a program, and cannot be associated with any data hierarchy. Index-names must be unique within a given program.

The data-names must be listed in their descending order of significance. If *data-name-2* is not the same name as the item being described, then all of the items identified by data-names in this phrase must be within the group item which is the subject of this entry, and must not contain an OCCURS clause.

Data-name-2, *data-name-3*, and so forth, may be qualified.

There must not be any entry containing an OCCURS clause between the items identified in the KEY IS phrase and the subject of this entry.

An index-name occupies one word (16 bits) in memory. It contains a positive binary value representing the actual displacement, in characters, from the beginning of the table. This displacement is associated with an occurence number in the table.

OCCURS - FORMAT 2

The second format of the OCCURS clause,

OCCURS ***integer-1*** **TO** ***integer-2*** **TIMES DEPENDING ON** ***data-name-1***

[{ ASCENDING | DESCENDING } KEY IS ***data-name-2*** **[,** ***data-name-3*** **] ...]**

[INDEXED BY ***index-name-1*** **[,** ***index-name-2*** **] ...]**

is used to define a variable length table.

Where

integer-1 must be less than *integer-2*. It represents the minimum number of occurences of the subject of the OCCURS clause.

integer-2 represents the maximum number of occurences.

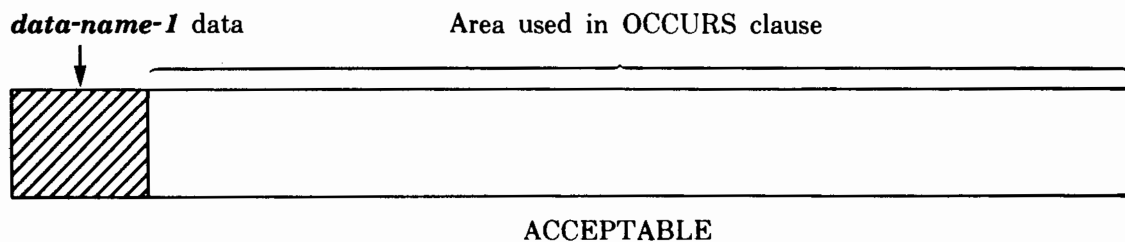
Note that this implies only that the number of occurences, and not the length of the data item is variable.

data-name-2 represents a positive integer used to determine the number of occurences of data items within the table. Thus, the current value of the data item referenced by *data-name-1* represents the number of occurences.

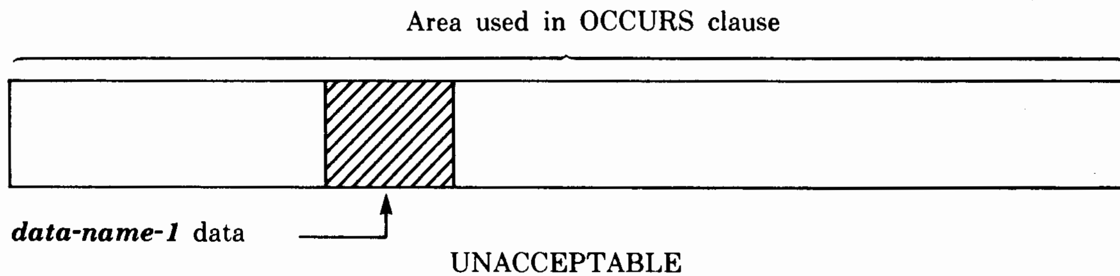
A data description entry containing this format of the OCCURS clause may only be followed, within the one record description, by data description entries subordinate to it.

This data item must not be contained in that part of the record being described which starts at the first character of the first element of the table and continues to the end of the record.

To illustrate, the two representations of records below show an allowable, and an unacceptable placement of the item referenced by *data-name-1* in a record.



ACCEPTABLE



Data-name-1, and its contents, may be described in a separate data description entry, and may be qualified.

Since the current value of the data item referenced by *data-name-1* represents the number of occurrences of a data item, it must be an integer in the range of values from *integer-1* to *integer-2*.

If the value of the integer represented by *data-name-1* is reduced from a previous value, the contents of those data-items whose occurrence numbers exceed this new value are unpredictable.

When a group item has a subordinate entry which specifies format 2 of the OCCURS clause, only that part of the group item specified by the value of the item moved by *data-name-1* is used.

The KEY IS and INDEXED BY phrases follow the same rules as for format 1.

PICTURE CLAUSE

The PICTURE clause describes the general characteristics and the editing requirements of an elementary item. It applies only to an elementary item, and must be used for every elementary data item except an index data item, for which its use is not allowed.

PICTURE Clause Format

[; { **PICTURE** / **PIC** } IS *character-string*]

Where

PICTURE and **PIC** — are equivalent.

Character-string — is a set of up to 30 characters, arranged in certain allowable combinations. These combinations determine the category of the elementary item.

The five categories of data are:

- alphabetic
- numeric
- alphanumeric
- alphanumeric edited
- numeric edited

Alphabetic Data

Alphabetic data is data that consists of upper and lower case letters from the English alphabet, and one or more blanks. When you wish to define the characteristics of an alphabetic data item, the character-string must consist of a combination of the letters, A and B, and, optionally, one or more non-negative integers in parentheses.

The symbol A represents a character of the alphabet or a space, whereas B represents where a space is to be inserted in the data item.

The integers are repetition factors, and are used to specify one or more occurrences of A or B in the picture.

For example, to describe an alphabetic data item that consists of six alphabetic characters, three spaces, and then twelve more alphabetic characters (or blanks), the following PICTURE clauses could be used, and are equivalent

```
          .
          .
          .
PICTURE IS AAAAAABBBAAAAAAAAAAAA
          .
          .
          .
PIC    A(6)B(3)A(12)
```

Numeric Data

Numeric data is data that consists of a combination of the Arabic numerals, 0,1,2,3,4,5,6,7,8, and 9, optionally including the positive or negative sign, or a representation of an operational sign as defined in the SIGN clause. Note that a decimal point is not part of the possible set of characters allowed in forming numeric data for a COBOL program.

You must enter the data without a decimal point, and use the V character of the PICTURE clause to indicate where the decimal point belongs.

The size of a numeric data item can be from 1 to 18 digits long counting the positive or negative sign, if it used.

When you wish to define the characteristics of a numeric data item, the picture clause you use must consist of only the symbols 9,P,S,V, and (in conjunction with the 9 and P symbols only) one or more repetition factors as described under the heading ALPHABETIC DATA, above.

The symbol 9 represents the character position which is to contain a numeral.

The symbol S indicates the presence of, but not necessarily the position within a numeric data-item, of an operational sign. It must be the leftmost character in the picture of the data item being described, and can appear only once in a given picture clause.

The S symbol is not counted in determining the size of an elementary data item unless the data item is subject to a SIGN clause using the SEPARATE CHARACTER phrase.

The symbol V is used to indicate the location of the assumed decimal point in numeric data. Like the S symbol, it may only appear once in any given picture clause.

Since the V does not represent a character position, it is not counted in the size of the elementary data item. Note also that if the V appears as the right-most character in a character string, it is redundant.

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 appearing in the data item.

That is, for each appearance of a P in the PICTURE of a data item, the data item is assumed to be multiplied by 10, or by one-tenth, depending upon whether the P symbol is on the right, or on the left of the character-string used to define the field.

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 0 whenever the item is accessed. When the data 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.

One or more P symbols can only be placed at either the left or the right end of a character-string. Since P represents a decimal position, the use of the P symbol and the V symbol as the left or right most symbol in the same string is redundant.

Thus, if the V symbol is used with the P symbol, it is meaningful only if it does not appear as the left or right most symbol in the character string.

The P symbol and the insertion character, period (.), cannot be used in the same PICTURE character-string.

To illustrate the use of the P symbol:

Input data: 241000

PICTURE clause: 999P(3)

Data stored as: 241

Data accessed as: 241000

Using the following PICTURE clause for the number 00000241

```
PICTURE  P(5)999
```

results in the data being stored as 241, but being accessed as 00000241.

A repetition factor may be used with the P symbol. Thus,

PPPPP99 is equivalent to P(5)99.

All numeric literals used 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 PPP999 are .000000 through .000999.

Alphanumeric Data

Alphanumeric data is data made up of any valid character used on an HP computing system.

To define an alphanumeric data item, you can use the symbols A, 9, or X. The PICTURE clause for this type of data, however, must contain at least one X symbol, or a combination of the A and 9 symbols to indicate that it represents an alphanumeric data item.

The A symbol can be used to represent alphabetic characters or a space, while the 9 symbol can be used to represent numerals. However, the entire PICTURE clause is treated as if it consists entirely of X symbols, where each X symbol can represent any single character used on an HP computing system.

Repetition factors may be used with all these symbols.

To illustrate,

```
77 FINISHER   PICTURE A(5)9(8).
```

is equivalent to

```
77 FINISHER   PICTURE X(13).
```

Alphanumeric Edited Data

Alphanumeric data is data consisting of any set of characters available on an HP computing system.

This type of data can be edited by specifying where one or more spaces, strokes, or zeros are to appear as part of the receiving data item.

To define a data-item to receive the edited alphanumeric data item you can use the A,X,9,B,0, and / symbols.

The A,X,9, and B symbols are discussed under the headings, Alphabetic Data, Numeric Data, and Alphanumeric Data, respectively.

The 0 symbol is used to specify where in the character string the numeral 0 is to be inserted.

The / (stroke) symbol represents where in the character string a stroke symbol is to be inserted.

All of the symbols used in an alphanumeric edited PICTURE clause may use a repetition factor.

In order to distinguish this form of data from alphanumeric (non-edited) data, one of the following conditions, minimally, must appear in the PICTURE clause:

At least one B and one X

or one 0 and one X

or one / and one X

or one 0 and one A

or one / and one A

To illustrate,

Sending data item: 010680

PICTURE of receiving data item: 99/99/99

Receiving data item: 01/06/80

Sending data item: NAMEADDRESSPHONENUMBERZIPCODE

PICTURE of receiving data item: `PICTURE A(4)B(15)A(7)B(15)A(5)BA(6)/A(7)`

Receiving data item:

NAMEΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔADDRESSΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔPHONEΔNUMBER/ZIPCODE

Numeric Edited Data

Numeric edited data consists, in standard data format, of a combination of the numerals 0,1,2,3,4,5,6,7,8, and 9, and an optional decimal point.

Editing of this type of data in COBOL II consists of leading zero suppression, filling or replacement, placement and alignment of a decimal point and a currency symbol, insertion of a sign, commas, blanks, or strokes.

This is accomplished through use of the symbols:

P,V,9,B,/,0,Z,(,),(.),*,+,-,CR,DB

and the currency symbol as defined in the CURRENCY SIGN clause of the SPECIAL-NAMES paragraph in the Environment Division. To distinguish this type of data from unedited numeric data, at least one of the above symbols (except the 9) must appear in the PICTURE clause.

A maximum of 18 digit positions may be represented in this type of PICTURE.

The first three of the above symbols are discussed under the heading, Numeric Data, and the second set of three are discussed under the heading, Alphanumeric Edited Data. The remaining symbols are discussed below.

The Z symbol is used for the suppression of leading zeros in the receiving data item. It can only be used to represent the leftmost leading numeric character positions.

The comma symbol (,) represents where in the character string of the receiving data item a comma is to be inserted. It cannot be the rightmost character in the PICTURE clause.

The period symbol (.) represents the decimal point for aligning the sending and receiving data items and also represents a character position into which a period (decimal point) is to be inserted. It may not be used if the V symbol is used, and may only appear once in a given PICTURE clause if the DECIMAL-POINT IS COMMA clause is not specified in the SPECIAL-NAMES paragraph. Also, it may not appear as the rightmost element in the PICTURE clause.

If the DECIMAL-POINT clause is specified, the roles of the commas and period symbols are reversed. Thus, in such a case, only one comma symbol may appear in a numeric edited PICTURE clause, but several periods may appear.

The plus (+), minus (-), CR (for CRedit), and DB (for DeBit) are used as editing sign control symbols. Only one of these symbols may appear in any given PICTURE clause, and when used, specify the position in the receiving data item into which the editing sign control symbol will be placed.

The asterisk (*) symbol is used for replacing leading zeros. Each leading zero in the sending data item is replaced in the receiving data item by an asterisk if there is an asterisk in the PICTURE clause for the receiving data item whose position corresponds to the position of the zero in the sending data item.

This symbol may not appear in a PICTURE clause for a data item which has the BLANK WHEN ZERO clause specified for it.

The appearance of a currency symbol in a PICTURE clause represents the position into which a currency symbol is to be placed. If you do not specify an alternative currency symbol through the CURRENCY SIGN clause of the SPECIAL-NAMES paragraph, the dollar ($) symbol is used.

This symbol must appear as the leftmost symbol in the character string, except that it may be preceded by a plus (+) or minus (-) symbol.

With the exception of the symbols, V, CR, DB, and period (.), all of the symbols discussed above may be specified using a repetition factor.

To illustrate numeric edited data:

Sending data item: -1234‸59

PICTURE clause of the receiving data item: `PICTURE 99,999.99DB`

Receiving data item: 1,234.59DB

Note in the second example that the DECIMAL-POINT IS COMMA clause is assumed.

Sending data item: 345777‸78

PICTURE clause of the receiving data item: `PICTURE +$ZZZ,ZZZ,ZZZ,ZZZ.99`

Receiving data item: +$345.777,78

Size Of Elementary Data Items

The size of an elementary data item is defined as being the number of character positions occupied by the elementary item in standard data format.

This size is determined by counting the number of allowable symbols used to represent character positions within a PICTURE clause for that item.

With the exception of the S, V, period (.), CR and DB symbols, all symbols discussed in the PICTURE clause may use repetition factors.

These repetition factors are represented by an integer enclosed by parentheses following the symbol to which they pertain, and indicate the number of consecutive occurrences of the symbol.

Furthermore, only the V symbol, and the S symbol if the SEPARATE CHARACTER phrase of the SIGN clause is not specified, do not participate in the count when you are determining the size of the data item.

DB and CR each represent two character positions.

When you count occurrences of characters in an elementary data item description, you must only count those symbols which appear without repetition factors, and add them to the sum of all integers appearing in repetition factors for that PICTURE clause.

Examples:

The size of the data item represented by the following PICTURE clause is 11 characters.

```
PICTURE ZZZ,999V99CR
```

The size of the data item represented by the next PICTURE clause is 17 characters.

```
PICTURE A(10)B(5)XX
```

In the next PICTURE clause, we assume that the SIGN IS SEPARATE clause is NOT specified for the data item represented by the PICTURE clause below.

```
PICTURE S9(5)V99
```

The size of the item described above is 7 characters.

Editing Rules

There are two general methods for performing editing in the PICTURE clause.

They are insertion, and zero suppression and replacement.

Editing takes place only when data is moved into an elementary data item whose PICTURE clause specifies editing (that is, whose PICTURE clause is alphabetic, alphanumeric edited or numeric edited. Thus, for example, data moved into a numeric field is not edited.

You may perform simple insertion editing for an item belonging to the alphabetic or alphanumeric edited categories.

Three other types of insertion, as well as suppression and replacement, may be performed on numeric edited data.

The other types of insertion are special, fixed, and floating insertion.

Table 9-2 below summaries the type of editing permitted for each category.

Table 9-2.
ALLOWABLE TYPES OF EDITING FOR CATEGORIES OF DATA ITEMS

CATEGORY	TYPE OF EDITING
Alphabetic	Simple insertion 'B' only
Numeric	None
Alphanumeric	None
Alphanumeric Edited	Simple insertion '0',',' , 'B' and '/'
Numeric Edited	All

SIMPLE INSERTION EDITING

The comma (,), space (B), zero (0), and stroke (/) symbols are used in simple insertion editing. These insertion characters are counted in the size of the receiving item, and represent the position in the receiving item into which the character is inserted.

Simple insertion editing is so called because, other than inserting the particular symbol, no other editing is done, and the data sent is unaffected except for the placement of the simple insertion characters between, before, or after the other characters received from the sending data item.

An example of simple insertion editing is shown in the illustration under the heading, ALPHANUMERIC EDITED DATA, on the preceding pages.

SPECIAL INSERTION EDITING

The period (.) is used in special insertion editing. In addition to being an insertion character, it is also used for alignment purposes when the sending data item is numeric and contains a decimal point. The result of this form of editing is the appearance of the period in the receiving item in the same position as it appears in the PICTURE clause for the item.

When data is moved to an item defined with the special insertion character, COBOL automatically provides truncation and zero fill to both the left and the right of the decimal point. However, if zero suppression or floating insertion editing are included in the PICTURE clause of the receiving data item, zero fill normally produced by special insertion editing is overriden.

To illustrate special editing,

> Sending data item: 12345‸678
>
> PICTURE clause of receiving data item: 9(5).99
>
> Receiving data item: 12345.67

Note that the right-most digit, 8, was truncated. This was caused by the alignment of decimal points.

Another example:

> Sending data item: ‸001
>
> PICTURE clause of receiving data item: 9,999.9999
>
> Receiving data item: 0,000.0010

Finally:

> Sending data item: 658456‸995
>
> PICTURE clause of receiving data item: 999.99
>
> Receiving data item: 456.99

FIXED INSERTION EDITING

Fixed Insertion Editing uses the currency symbol and the editing sign control symbols, +, -, CR, and DB.

Only one currency symbol and one editing sign control symbol can be used in a given PICTURE clause when you wish to use fixed insertion editing. When the CR or DB symbol is used, each represents two characters, and must be in the rightmost character positions counted in determining the size of the receiving item.

When the + or - symbol is used, it must be either the leftmost or rightmost character in the PICTURE clause for the receiving item, and is counted in the size of that item.

The currency symbol must be the leftmost character position to be counted in the size of the item, except that it may be preceded by a + or a - symbol.

Fixed insertion editing results in the insertion character occupying the same character position in the receiving item as it does in the character string used in the PICTURE clause.

The sign control symbols produce different results, depending upon whether the sending data item is positive or negative. These differing results are shown in table 9-3 below.

TABLE 9-3.
EFFECTS OF SIGN CONTROL SYMBOLS ON RECEIVING ITEMS

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

FLOATING INSERTION EDITING

Floating insertion editing uses the currency, +, or - symbol. As such, each is mutually exclusive of the others when you wish to perform this type of editing.

Also, zero suppression and replacement cannot be used in the same character string of a PICTURE clause using floating insertion editing.

You can represent floating insertion editing in one of two ways. The first is to represent any or all of the leading numeric positions to the left of the decimal point with your chosen floating insertion character. The second way is to represent every numeric character position, on both sides of the decimal point, with the insertion character.

Floating insertion editing is indicated by an occurrence of the same symbol used at least twice in the same string. This is the major distinction between fixed and floating insertion editing.

Between or to the right of this floating insertion string may be any of the simple insertion characters. If such is the case, the simple insertion characters are a part of the floating insertion character string.

The bounds of the floating insertion string (including the simple insertion characters as noted above) are formed by the leftmost and the rightmost elements of the floating string.

Non-zero numeric data can be stored in the receiving data item starting at the first character to the right of the leftmost character in the floating string, and proceeding through the entire floating string.

If the floating insertion characters are only to the left of the decimal point, insertion takes place in a fashion analogous to the following algorithm:

1. The leading character of the sending data item is checked to see if it has a zero value. If it is zero, the floating insertion character is inserted in the corresponding character position of the receiving data item and the preceding character of this data item is replaced with a space.
2. The next character of the sending data item is then checked for a zero value, and if it is zero, the action described in step 1 is repeated.
3. The process continues either until no numeral in the sending data item is non-zero (in which case all the positions corresponding to the floating insertion string in the receiving data item are replaced with spaces), or some non-zero numeral is found in the sending data item and this numeral appears to the left of the decimal place.

If any simple insertion character appears as part of the floating insertion string, and no non-zero character is encountered in the sending data item before the next floating insertion character position is considered, one simple insertion character is replaced by the floating insertion character, and the preceding floating insertion character is replaced by a space.

When a non-zero numeral is encountered in the data item, that numeral and all following it are replaced in the positions corresponding to their positions in the floating insertion string.

If the floating insertion characters correspond to every numeric character position, including those to the right of the decimal point, the algorithm is the same as above, with one exception.

The exception is when the original data item is 0. In this case, the result of floating insertion editing is that the data item referenced by the PICTURE clause contains only spaces.

Note that to avoid truncation, your character-string in the PICTURE clause for the receiving data item must be, minimally, the size of the number of characters in the sending data item, plus the number of non-floating insertion characters being inserted into the receiving data item, plus one for the first floating insertion character.

To illustrate floating insertion editing, we use a sending data item which is, in standard data format, 0012345, and use the PICTURE clause,

```
PICTURE    $$$,$$$.99
```

to describe the receiving data item.

Using the algorithm described above, the steps taken appear as follows:

1. First character equal to 0? Yes. Thus, receiving data item appears as Δ$
2. Second character equal to 0? Yes. Thus, receiving data item appears as ΔΔΔ$ (Since the comma preceded the first occurrence of a non-zero numeral).
3. Third character equal to 0? No. Thus, receiving data item appears as $123.45

Result: $123.45

Note that if the PICTURE clause had been of the form,

```
PICTURE    $$,$$$.99
```

the result would be $12.34 because of truncation of the sending data item to allow for insertion of the floating character, $, in the receiving data item.

ZERO SUPPRESSION EDITING

Zero suppression editing allows you to replace leading zeros of the sending data item with either spaces or asterisks in the receiving data item.

You can replace one or more leading zeros with spaces by placing a Z in the corresponding positions of the PICTURE character string used to represent the receiving data item.

If you wish to replace leading zeros with asterisks (*), you use a string of asterisks rather than Z's in the PICTURE for the receiving data item.

you may use either the Z symbol or the * symbol, but not both, in any one PICTURE clause.

The algorithm used in zero suppression and replacement is essentially the same as for floating insertion editing.

That is, any simple insertion symbols may appear between the first and last symbol, or to the right of the last suppression symbol, and they are included as part of the suppression string.

Furthermore, if the suppression symbols appear only to the left of the decimal point, any leading zero in the sending data item which corresponds to a suppression symbol is replaced by that suppression symbol, and suppression terminates with the first occurrence of a non-zero numeral in the sending data item, or with the decimal point, whichever occurs first.

If all numeric character positions are represented by suppression symbols and the sending data item is 0, the entire receiving data item will consist of spaces (if Z's are used), or asterisks, except for the decimal point (if asterisks are used).

If all numeric characters are represented in the receiving data item by suppression or replacement symbols, and the sending data item is not 0, suppression and replacement take place in the same manner as if the suppression symbols appeared only to the left of the decimal place.

Note that you may not use floating insertion editing in the same PICTURE clause in which you are using zero suppression and replacement.

To illustrate zero suppression and replacement:

Sending data item in standard data format: 00405367

Picture of receiving data item: `PICTURE $ZZZ,ZZZ.ZZ`

Result: $ΔΔ4,053.67

Using the same sending data item, but with the following picture of the receiving data item,

```
PICTURE $***,**9.99
```

results in $**4,053.67

Precedence Rules

Table 9-4 below shows the order of precedence when using insertion and suppression/replacement symbols in a character string of a PICTURE clause.

An X at an intersection indicates that the symbol or symbols at the top of the column may precede the symbol or symbols at the left of the row.

Note that those symbols appearing in braces are mutually exclusive, and that "CS" indicates the currency symbol.

Also note that the + and -, and the Z and * symbols appear twice in the Non-Floating and Floating insertion symbols section of the chart. The left-most (columns) and the topmost (rows) appearance of each of these pairs represents the use of the symbol or symbols to the left of the decimal point.

The right-most (columns) and lower (rows) appearance represents the use to the right of the decimal point.

TABLE 9-4.

PICTURE Character Precedence Chart

Second Symbol \ First 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
B	x	x	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	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
CR DB	x	x	x	x	x				x	x	x			x	x	x			x	x	x
cs						x															
Z *	x	x	x	x		x			x	x											
Z *	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
cs	x	x	x	x		x								x							
cs	x	x	x	x	x	x								x	x				x		x
9	x	x	x	x	x	x			x	x		x		x		x	x	x	x		x
A X	x	x	x													x	x				
S																					
V	x	x	x	x		x			x	x		x		x		x		x		x	
P	x	x	x	x		x			x	x		x		x		x		x		x	
P						x			x									x	x		x

REDEFINES CLAUSE

The REDEFINES clause allows you to define the same storage area in main memory for different data items whose lengths are not described as variable in an OCCURS clause.

REDEFINES Clause Format

level-number { ***data-name-1*** }

[; **REDEFINES** ***data-name-2***]

Where

level-number and ***data-name-1***	are the level number and data name of the data item being described. These are not part of the REDEFINES clause; however, when the REDEFINES clause is used in a data description entry, it must be immediately preceded by *level-number* and *data-name-1.*
Data-name-2	is the data name used in a different data description entry. It must have the same level number associated with it as does *data-name-1.* however, the level number must not be 66 or 88. Nor can it be 01, if the REDEFINES clause is used in the File Section.

Redefinition of storage area begins at the *data-name-1* entry and continues until a level number less than or equal to that of *data-name-1* (or *data-name-2* since they are the same) is found.

Since *data-name-1* is a redefinition of the storage area for *data-name-2*, no entry having a level number numerically lower than the level number of *data-name-2* may occur between the data description entries of *data-name-2* and *data-name-1.*

Furthermore, the description entry for *data-name-2* cannot contain a REDEFINES or an OCCURS clause, but may be subordinate to an entry which does contain one of these clauses.

If the data description entry for an item to which *data-name-2* is subordinate contains an OCCURS clause, the reference to *data-name-2* in the REDEFINES clause must not be subscripted or indexed.

If the level number of *data-name-1* and *data-name-2* is other than 01, the description of *data-name-2* must specify the same number of character positions as specified for the data item referenced by *data-name-2.*

Also, multiple redefinitions of the same character positions are permitted in COBOL II. However, multiple redefinitions of the same character positions must all use the data-name of the entry that was originally used to define the area, and the entries providing the new descriptions must immediately follow the entries used to define the area currently being redefined. The new entries must not (except in the case of condition name entries) contain any VALUE clauses.

To illustrate the REDEFINES clause:

```
FD FILEN.
       01 RECORD-IN      PICTURE X(80).
       01 RECORD-PARTS REDEFINES RECORD-IN.
          02  NAME            PICTURE X(30).
          02  STREET          PICTURE X(20).
          02  CITY            PICTURE X(20).
          02  STATE           PICTURE X(10).

       01 PARTS-TABLE.
          02 PART OCCURS 35 TIMES.
             03 NAME                PIC X(10).
             03 QUANTITY            PIC 9(04).
             03 UNIT-PRICE          PIC 9(06).
             03 LOCALE              PIC X(10).
             03 SITE-INFO REDEFINES LOCALE.
                04 BUILDING-NO      PIC X(03).
                04 FLOOR-NO         PIC X(02).
                04 SECTION-NO       PIC X(02).
                04 BIN-NO           PIC X(03).
```

The above two uses of the REDEFINES clause are permissible, whereas the following two are not.

```
01 VOCABULARY OCCURS 2000 TIMES    PIC X(100).
      01 WORDLIST REDEFINES VOCABULARY.
         02 INITIAL                       PIC X(20).
         02 SECOND                        PIC X(30).
         02 THIRD                         PIC X(30).
         02 FOURTH                        PIC X(20).

      01 RECORD-IN.
         02 FIRST-FIELD.
            03 SUB-AA                     PIC X(15).
            03 SUB-AB                     PIC X(05).
         02 SECOND-FIELD.
            03 SUB-BB REDEFINES SUB-AA    PIC X(15) VALUE SPACES.
            03 SUB-BB1                    PIC X(05).
```

The first unacceptable usage above is because of the use of an OCCURS clause in the description of VOCABULARY. The second is unacceptable because of the 02 level entry between SUB-AA and SUB-BB. Also, the new entry may not contain any value clauses.

SIGN CLAUSE

The SIGN clause is only used with a signed numeric data description whose usage is DISPLAY, or a group item containing at least one such data description entry.

It states the position of the sign, whether leading or trailing, as well as whether the sign was formed by overstriking in the first or last character of the data item. (see the USAGE IS DISPLAY clause, above), or was formed separately.

SIGN Clause Format

```
[ ; [ SIGN IS ] { LEADING  } [ SEPARATE CHARACTER ] ]
                { TRAILING }
```

Where

LEADING and **TRAILING** — indicates that the sign is at the front, or is at the back, respectively, of the item.

SEPARATE — indicates that the sign is not overstruck; thus, that the sign exclusively occupies the first or last character of this item.

Only one sign clause may be used per given numeric data description entry.

Also, if the CODE-SET clause is specified in a file description, any signed numeric data description entry associated with that file must be described with the SIGN IS SEPARATE clause.

Valid signs for data items, and their representations when overstriking is used, are shown in the table under the USAGE IS DISPLAY heading on the following pages.

Of course, for a SIGN IS SEPARATE designation, the two valid operational signs (whether LEADING or TRAILING) are + and - for positive and negative quantities, respectively.

For a signed numeric data description entry having no SIGN clause associated with it, the default is equivalent to SIGN IS TRAILING. That is, the sign is assumed to be overstruck in the last column of the item.

In either the default case, or the case when the optional SEPARATE CHARACTER phrase is not used, the letter "S" in the PICTURE clause is not counted in determining the size of the item when represented in standard data format.

To illustrate the SIGN clause:

1. The data to be entered is `123489F`

   In this case, no SIGN clause is required, since the default is SIGN IS TRAILING. However, note that if the PICTURE clause for this data item is `PICTURE S9(7)`, the size of the data item is seven characters.

2. The data to be entered is `+1409748`

   In this case the SIGN clause should be: `SIGN IS LEADING SEPARATE CHARACTER`

   Also, since the sign is separate, the PICTURE clause for this data item, `PICTURE S9(7)` defines the data item to be eight characters long in order to hold the separate sign.

SYNCHRONIZED CLAUSE

The SYNCHRONIZED clause is used to align items defined as USAGE IS COMPUTATIONAL on word boundaries in order to facilitate arithmetic operations. (A word is two characters (bytes) long; that is, it consists of 16 bits.)

All other items are aligned on byte boundaries. Since the character (byte) 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. It is treated as a comment for items described as DISPLAY, INDEX, or COMPUTATIONAL-3.

SYNCHRONIZED Clause Format

```
[ ; { SYNCHRONIZED } [ LEFT  ] ]
[   { SYNC         } [ RIGHT ] ]
```

Because of the word structure used on HP computer systems, the LEFT and RIGHT options are irrelevant, and are treated as comments by the compiler.

The words SYNCHRONIZED and SYNC are equivalent.

The compiler always aligns all level 01 and level 77 items on a word boundary.

When the SYNCHRONIZED clause is specified for a data item whose description also contains an OCCURS clause, or in a data description entry of a data item subordinate to a description entry containing an OCCURS clause, each occurrence of the data item is synchronized, and any implicit filler (see Slack Bytes below) generated for other data items within that same table are generated for each occurrence of those data items.

Slack Bytes

The SYNCHRONIZED clause specifies that the data being described is to be aligned on word boundaries. If the SYNCHRONIZED item does not fall naturally on a word boundary, the compiler assigns the item the next highest boundary address. Since a word consists of two bytes, the item will thus fall on a word boundary.

The effect of adding this byte is equivalent to providing an extra FILLER character, known as a slack byte, just before the SYNCHRONIZED item.

This slack byte is not used for any other data item and is not counted in the size of the items. It is, however, included in the size of any group item or items to which the elementary item belongs, and is included in the character positions redefined when the SYNCHRONIZED item is the object of a REDEFINES clause. Thus, when you use the REDEFINES clause in a data description that also contains a SYNCHRONIZED clause, you must insure that the redefined item has the proper boundary alignment for the item that redefines it.

Whenever a SYNCHRONIZED item is referenced in your program, the original size of the item, as shown in the PICTURE clause, is used in determining any action that depends on size. Such actions include justification, truncation, or overflow.

If the SYNCHRONIZED clause is not used, no space is reserved for slack bytes, and when a computation is performed on a data item described as COMPUTATIONAL, the compiler provides the code and space required to move the data item from its storage area to a work area. This work area has the alignment required to perform the computation.

As an example of slack bytes, consider the following data description entries:

```
01 ITEM-LIST.
       02  ITEM-NUMBER     PICTURE X(3).
       02  ITEM-1          PICTURE X(4).
       02  ITEM-2 REDEFINES ITEM-1 PICTURE S9(6) USAGE COMP SYNC.
```

The above is an example of the programmer NOT taking into account the slack byte required because of the REDEFINES clause. To correct it, the description of ITEM-LIST should include an extra byte prior to ITEM-1:

```
01 ITEM-LIST.
       02  ITEM-NUMBER      PICTURE X(3).
       02  SLACK-BYTE       PICTURE X.
       02  ITEM-1           PICTURE X(4).
       02  ITEM-2 REDEFINES ITEM-1 PICTURE S9(6) USAGE COMP SYNC.
```

This change is all that was needed, since all 01 level entries are aligned on word boundaries.

USAGE CLAUSE

The USAGE clause specifies how the data item being described is stored internally, and in one case (index), specifies that the data item being described contains a value equal to the value of an index-name associated with an occurrence number for a table element.

USAGE Clause Format

```
┌                ┌                   ┐ ┐
│                │ COMPUTATIONAL-3   │ │
│                │ COMP-3            │ │
│ ; [ USAGE IS ] │ COMPUTATIONAL     │ │
│                │ COMP              │ │
│                │ DISPLAY           │ │
│                │ INDEX             │ │
└                └                   ┘ ┘
```

Where

COMPUTATIONAL-3	and COMP-3 are equivalent. They specify packed decimal format.
COMPUTATIONAL	and COMP are equivalent. They specify twos complement binary integer format.
DISPLAY	is the default usage if no USAGE clause is specified. It specifes that data is to be stored internally as ASCII characters.

This clause is optional, with a USAGE IS DISPLAY being used by your program for any data item having no USAGE clause as part of its description, or as part of the description of a data item to which it is subordinate.

If you choose to use the USAGE clause, you may do so at any level of organization. However, if written at a group level, the USAGE clause applies to each elementary item in the group, and any USAGE clause specified at the elementary level must be the same as at the group level.

Although a USAGE clause does not affect the use of a data item, some of the statements in the Procedure Division may restrict the USAGE clause of the operands used.

USAGE IS DISPLAY

When usage of a data item is defined (implicitly or explicitly) as DISPLAY, the data is stored internally as ASCII characters.

This means that each character of data is stored as an eight-bit byte.

If you are using the data item in a non-computational manner (that is, printing or displaying it), this is the appropriate type of usage to be specified.

However, for optimum use of your COBOL program, you should specify USAGE is COMPUTATIONAL or COMPUTATIONAL-3 for data items intended for use in computations.

This is because data items described as USAGE IS DISPLAY must be be converted to two's-complement binary or packed-decimal format before they can be used in computations, and this conversion takes time. Also, if you intend to use signed numeric data items for computational purposes, you must specify a sign (by using the S symbol) in the PICTURE clause for that item (see USAGE IS COMPUTATIONAL on the following page), whether its usage is specified as DISPLAY or otherwise.

An unsigned numeric data item whose description specifies the USAGE IS DISPLAY clause is assumed to be positive.

Numeric DISPLAY items without a SIGN IS SEPARATE clause associated them are represented in ASCII coded (eight bits) decimal digits (0 through 9) except for the units digit which carries the sign of the data item. The units digit, with the sign of its associated number being positive, negative or no sign (absolute value) respectively, is represented in ASCII code as shown in table 9-5.

Note that using signed numeric DISPLAY data items for computational purposes is more efficient than using unsigned numeric data items.

Table 9-5.
UNITS DIGIT FOR ASCII CODED DECIMAL NUMBERS

Units digit	Internal representation (ASCII)		
	Positive	Negative	No sign
0	{	}	0
1	A	J	1
2	B	K	2
3	C	L	3
4	D	M	4
5	E	N	5
6	F	O	6
7	G	P	7
8	H	Q	8
9	I	R	9

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. If no sign is desired, only the digits need be punched.

Zone signs cause the signed digit to have the same punch configuration as certain other characters. This is the purpose of the S symbol in the PICTURE clause; it informs the compiler that the last digit in the field is to be interpreted as a number and a sign, and not as the character that it would otherwise represent.

Table 9-5 shows the data character equivalents to each possible right-most digit sharing a zone sign.

> NOTE: If a zoned overpunch occurs in a position other than specified by the SIGN clause, a warning is issued and the data is converted to a numeric value by stripping the zoned punch. For example, if you used zoned overpunching to form the characters, 1GH3D on a punched card, it is interpreted as 1783D.

USAGE IS COMPUTATIONAL

When usage of a data item is defined as COMPUTATIONAL, the data must be numeric, and is stored in twos-complement binary integer form, consisting of either one, two, or four 16-bit words each. The number of words used depends upon the size of the data item, as shown in table 9-6 below.

Table 9-6.
NUMBER OF WORDS USED TO CONTAIN A COMPUTATIONAL DATA ITEM

PICTURE	NUMBER OF WORDS ASSIGNED
S9 to S9(4)	1
S9(5) to S9(9)	2
S9(10) to S9(18)	4

A data item whose usage is defined as COMPUTATIONAL must have an unedited numeric PICTURE clause associated with it. It may contain up to 18 digits plus a sign. Also, if a group item is described as COMPUTATIONAL, all of the elementary items in the group are computational, and may thus be used in computations. However, the group item itself may not be used in computations, as it is considered alphanumeric. A numeric data item that does not have a sign associated with it is assumed to be positive.

As with numeric DISPLAY data items, a signed numeric data item whose USAGE is COMPUTATIONAL is more efficient than an unsigned numeric data item with the same USAGE.

USAGE IS COMPUTATIONAL-3

Data items described as COMPUTATIONAL-3 are subject to the same restrictions and are used in the same way as data items described as COMPUTATIONAL. Such items are, however, stored in 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. Table 9-7 illustrates the bit configurations used to represent signs in packed decimal fields. Notice that the bit configuration 1100 specifies a positive value; the four-bit configuration 1111 represents the unsigned (assumed positive) value when an unsigned picture is specified. For negative values, the four-bit configuration is 1101.

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 are effectively in a hexadecimal format.

Table 9-7
COMPUTATIONAL-3 SIGN CONFIGURATION

SIGN	BIT CONFIGURATION	HEXADECIMAL VALUE
+	1100	C
–	1101	D
unsigned	1111	F

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		01	23	4C
+12345‸	S99999		12	34	5C
12345‸	99999		12	34	5F
−1‸2	S999V999	00	01	20	0D
−‸5	S999V999	00	00	50	0D
+1‸22172	S999V999	00	00	12	2C
−12345	99999		12	34	5F

Note that the last number in the table was stored as an unsigned (assumed positive) number because the receiving field is unsigned according to its PICTURE.

USAGE IS INDEX

An elementary data item whose usage is defined as INDEX is called an index data item. Its purpose is to hold the contents of a table index while the table is being processed. Thus, any value within the index data item must correspond to an occurrence number of an element in a table. Both internally and externally, an index data item is stored as a synchronized unsigned computational integer, one word (16 bits) long.

An index data item cannot be a conditional variable, and can only be referenced explicitly in a SEARCH or SET statement, a relation condition, the USING phrase of the Procedure Division header, or USING phrase of a CALL statement.

Do not use the SYNCHRONIZED, JUSTIFIED, PICTURE, VALUE, and BLANK WHEN ZERO clauses to describe a group of elementary items whose usage is defined as INDEX. Index data items are automatically SYNCHRONIZED.

In ANSI COBOL '74, if a group item is described with a USAGE IS INDEX clause, all of its elementary items are index data items, but the group itself is not an index data item and cannot be used in the SEARCH or SET statements, or in an alphanumeric comparison in the Procedure Division. COBOL II/3000, however, does allow a group item described with USAGE IS INDEX to be used in an alphanumeric comparison.

VALUE CLAUSE

In a format 1 data description entry, the VALUE clause is used to define the values of constants, and to initialize the values of Working Storage data items. In format 3 of data description entries, it is used to define the values associated with condition-names. Format 3 data description entries are discussed at the end of this section.

VALUE Clause Format (Format 1 data description)

VALUE IS ***literal***

Where

literal is the value assigned to the data item being described.

The VALUE clause for a format 1 data description entry can only be used in the Working Storage section.

If used, the VALUE clause causes the item to which it is associated to assume the specified value at the start of the object program, irrespective of any BLANK WHEN ZERO or JUSTIFIED clause. If the VALUE clause is not used in an item's description, the initial value of the item is undefined.

Restrictions On The Use Of The VALUE Clause

The restrictions below apply to the use of a VALUE clause in a format 3 data description entry as well as the use of a VALUE clause in a format 1 data description entry.

The VALUE clause cannot be used for items whose size is variable, in a data description entry containing an OCCURS or REDEFINES clause (except when used with a condition-name) , or in an entry subordinate to an entry containing such a clause. Nor can it be used for a group containing items with descriptions including JUSTIFIED, SYNCHRONIZED, or USAGE (except USAGE IS DISPLAY).

The VALUE clause must not conflict with any other clauses in the data description of the item, or in the data description within the hierarchy of the item.

Literals In VALUE Clauses

The literals used in the VALUE clause are subject to the following rules:

- Figurative constants may be substituted for literals.
- A signed numeric literal must have a signed numeric PICTURE and character-string associated with it.
- All numeric literals must have a value within the range of values indicated by the PICTURE clause, and must not have a value which would require truncation of non-zero digits. Nonnumeric literals must not exceed the size indicated by the PICTURE clause.
- If the category of the item being described is numeric, all literals in the VALUE clause must be numeric. If the literal defines the value of a working storage item, the literal is aligned in the data item according to the standard alignment rules.
- If the category of the item being described is any other than numeric, all literals in the VALUE clause must be nonnumeric. The literal is aligned in the data item as if the data item had been described as alphanumeric. Editing characters in the PICTURE clause are included in determining the size of the data item, but have no effect on initialization. Thus, the VALUE for an edited item must be presented in an edited form.
- If the VALUE clause is used in an entry at the group level, the literal must be a figurative constant or a nonnumeric literal, and the group area is initialized without consideration for the individual elementary or group items contained within this group. A VALUE clause cannot be used for elements of a group which has a VALUE clause assigned to it at the group level.

RENAMES CLAUSE

The RENAMES clause permits alternative, possibly overlapping groupings of elementary items.

This clause is always associated with a 66 level entry, and, therefore, makes up the second general format of a data description entry.

RENAMES Clause Format (FORMAT 2 OF DATA DESCRIPTION ENTRIES)

66 ***data-name-1*** **;** **RENAMES** ***data-name-2*** [{ **THROUGH** / **THRU** } ***data-name-3***] .

Of course the level number, *data-name-1*, and semicolon are not part of the RENAMES clause, but are used to clarify the purpose of the clause.

THROUGH
and
THRU — are equivalent.

Data-name-1 — is the name used to rename the item or items referenced by *data-name-2* and *data-name-3*. It cannot be a qualifier, and can only be qualified by the names of the associated 01, FD, or SD level entries.

Data-name-2
and
data-name-2 — must be names of elementary items or groups of elementary items in the same logical record. They must not be the same name, and neither may have an OCCURS clause in its data description, or be subordinate to an item that has an OCCURS clause in its description. Furthermore, no item within the range of the portion of the logical record being renamed can be variable in size, or can contain such an item. *Data-name-2* and *data-name-3* may be qualified.

If *data-name-2* is used alone (that is, the optional THROUGH phrase is unused), *data-name-2* can be either a group or an elementary item.

When *data-name-2* is a group item, *data-name-1* is treated as a group item; when *data-name-2* is an elementary item, *data-name-1* is treated as an elementary item.

If the THROUGH phrase is used, *data-name-1* is a group item which includes all elementary items starting from *data-name-2* (if elementary) or the first elementary item in *data-name-2* (if a group item), and concluding with *data-name-3* (if elementary), or with the last elementary data item in *data-name-3* (if *data-name-3* is a group item).

Because of the way in which *data-name-1* is defined, there are restrictions on the area described by *data-name-2* and *data-name-3*. That is, the area described by *data-name-3* must not begin to the left of the first character in the area described by *data-name-2*, and it must end to the right of the last character of the area of *data-name-2*.

Note that this implies that *data-name-3* cannot be subordinate to *data-name-2*.

You can use more than one RENAMES entry for a single logical record; however, all RENAMES entries referring to data items within a given logical record must immediately follow the last data description entry of the associated record description entry. You cannot use a level 66 entry to rename another level 66 entry or a 77, 88, or 01 level entry.

CONDITION NAMES

A *condition-name* is always subordinate to another data item, called the *conditional variable*. The characteristics of a condition-name are implicitly those of its conditional variable. Thus, for example, if a conditional variable is described as PIC 9(5) USAGE COMP-3, then the condition-name is itself implicitly a COMPUTATIONAL-3 data item consisting of five digits. This must be reflected in the value or values assigned to the condition-name.

A condition-name is assigned one or more values in a format 3 data description entry. The condition-name can later be used in a comparison with the conditional variable (see Section X, The Procedure Division, for information on condition-name conditions).

Since a condition-name is always subordinate to another data item, the level number 88 is used to describe it. This makes up the third format of a data description entry, as shown below.

CONDITION NAMES Format (FORMAT 3 OF DATA DESCRIPTION ENTRIES)

```
                      { VALUE IS   }             [ { THROUGH }             ]
88 condition-name ;   { VALUES ARE }  literal-1  [ { THRU    }  literal-2  ]

   [              [ { THROUGH }             ] ]
   [ , literal-3  [ { THRU    }  literal-4  ] ] ...   .
```

Where

condition-name	is any valid user-defined COBOL word.
literal-1 ***literal-2,*** and so forth	are the values assigned to the condition-name.
THROUGH	is equivalent to THRU; they may be used interchangably.

The VALUE clause and the *condition-name* itself are the only two clauses permitted in a format 3 data description entry.

In a format 3 data description entry, the VALUE clause can be used in any of the sections of the Data Division, and must be used for condition-names.

Wherever the THROUGH phrase (format 2) is used, *literal-1* must be less than *literal-2*, *literal-3* must be less than *literal-4*, and so forth.

When a VALUE clause is used in a level 88 entry, you can specify no more than 127 ranges of values for the related *condition-name*. A range of values is either a single literal, or two literals related by the THROUGH (or THRU) key word.

Additional rules applying to the VALUE clause of a format 3 data description entry are discussed under the headings, RESTRICTIONS ON THE USE OF THE VALUE CLAUSE, and LITERALS IN VALUE CLAUSES, earlier in this section.

The condition-name entries for a particular conditional variable must follow the entry describing the item with which the condition-name is associated (that is, the conditional variable).

Each condition-name in your program must have a separate level-88 entry associated with it.

A condition-name cannot be associated with any data description entry containing a level-number 66, another condition-name, or a group item with descriptions including JUSTIFIED, SYNCRONIZED, or USAGE (other than DISPLAY).

An example of a format 3 data description:

```
01 CONDVAR          PIC 9(5)      USAGE DISPLAY.
    88 COND1  VALUE 10 THRU 25, 100 THRU 250.

  01 ALPHAVAR         PIC A.
    88 ALPHACOND     VALUE "A" , "M" THROUGH "Z".
```

PROCEDURE DIVISION

SECTION X

The Procedure Division is the division in which your program performs the operations for which it was designed.

It must be included in every COBOL program, and may contain declaratives as well as nondeclarative procedures.

Generally, statements in the Procedure Division are executed sequentially in the order in which they were entered by the compiler.

You can, however, alter this sequential flow by using the IF statement, the PERFORM statement, or the GO TO statement.

Also, through the use of the DECLARATIVES keyword coupled with the END DECLARATIVES keywords, you can specify procedures to be executed only under special circumstances.

PROCEDURE DIVISION HEADER

The Procedure Division header has the form:

```
PROCEDURE DIVISION [ USING data-name-1
               [ , data-name-2 ] ... ] .
```

The Procedure Division header begins in area A of your program. Note that the header must be terminated by a period followed by a space.

USING Clause

The USING clause in the Procedure Division header is required if and only if the program in which it appears is to be called by another COBOL program through the CALL statement, and the CALL statement itself includes a USING clause.

That is, the USING clause in a Procedure Division header identifies the program in which it appears as being a subprogram which references data common to the program that calls it.

The data-names in the USING clause must follow the rules listed below.

1. Each data item named in the USING phrase of a Procedure Division header must be described in the Linkage Section of that program, and must have a level-number of either 01 or 77.

2. Data items are processed according to their descriptions in the called program, and not according to their descriptions in the calling program. Note that although this implies that common data may have different usages, the data must, as a general rule, have the same usage. Results may be undefined if usages are mixed. This is because data sharing is done by passing an address of the data item, with no conversion from one data type to another.

3. The descriptions of data common to both programs must define an equal number of character positions.

4. Data is passed from one program to another according to the position of its name in the USING phrase, and not by its name. Thus, data in the calling program may be known to the calling program by a completely different name. Also, the same name can appear in the USING phrase of the CALL statement, but each name in the USING phrase of a Procedure Division header must be unique with respect to other names in that phrase.

5. No more than 57 data names may be listed in the USING phrase.

For a more general overview of passing data to and from two COBOL programs, see Section XII, INTERPROGRAM COMMUNICATION.

GENERAL FORMAT OF THE PROCEDURE DIVISION

The body of the procedure division has two general formats:

Format 1

```
{ paragraph-name. [ sentence ] ... } ...
```

Format 2

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

 { section-name SECTION [ segment-number ].
[ paragraph-name. [ sentence ] ... ] ... } ...
```

The first format of the Procedure Division body can be used when you wish to use no section names in your program. In such a case, only paragraph names are used to define procedures. This is generally not the best way to write a COBOL program, since it does not allow for USE procedures, segmentation of the object program, or sort/merge input or output procedures. It may be beneficial, however, if you are writing a very short, simple program.

The second format of the Procedure Division body is used when you wish to allow for segmentation of your program, define sort/merge input or output procedures, or define declarative procedures.

In the second format, section names may be used to define procedures, with paragraph names being used as subsections. If any part of the Procedure Division is written using a section name, then the entire Procedure Division must be written using section names. Thus, if a section name is used, either the entire Procedure Divison is a single section, or the Procedure Division consists of several sections.

DECLARATIVE Sections

Declarative sections are optional; if used, however, they may appear in COBOL subprograms as well as main programs.

When you define declarative sections, they must be the first sections within the Procedure Division, and must be preceded by the key word, DECLARATIVES, followed by a period and a space. To indicate where declarative sections end and the remainder of the Procedure Division begins, you must use the keywords, END DECLARATIVES, followed by a period and a space.

USE procedures consist of a section name followed by a space and the key word, SECTION, followed by an optional segment number, a period and a space, a declarative sentence, and one or more optional paragraphs.

A declarative sentence is one which contains a USE statement. The USE statements themselves are not executed. They simply define the conditions calling for the execution of the USE procedures. The DECLARATIVE procedures are the optional paragraphs following the declarative sentence.

A single USE procedure is terminated in a source program by either a new section name, which indicates the beginning of another Declarative procedure, or by the key words, END DECLARATIVES, which indicate the end of the list of Declarative sections.

As is implied in the preceding paragraph, you must define a new section for each USE statement entered in your source program.

Declarative procedures must not reference non-declarative procedures, although you may use a PERFORM statement in the non-declarative portion of a program to refer to procedures associated with a USE statement.

Below is an example of the declaratives portion of a COBOL program:

```
        .
        .
PROCEDURE DIVISION.
DECLARATIVES.
IN-FILE-ERR SECTION.
    USE AFTER STANDARD ERROR PROCEDURE ON IN-FILE.
REPORT-ERR-PARA.
    DISPLAY "ERROR-IN IN-FILE."
    DISPLAY "FILE-STATUS ITEM IS  ", FILE-STAT.
    DISPLAY "WHAT ACTION?"
    DISPLAY "ENTER C OR A FOR CONTINUE OR ABORT"
    ACCEPT DECISION.
    IF DECISION IS EQUAL TO "A" MOVE "ON" TO STOP-IT.
FILE-LABEL SECTION.
    USE AFTER STANDARD BEGINNING FILE LABEL
    PROCEDURE ON OUT-FILE.
WRITE-LABEL-PARA.
    MOVE LBL-DATE TO LABEL-DATE;
    MOVE "ANS" TO LABEL-TYPE;
    MOVE "0" TO LABEL-SEQ;
    MOVE "03/31/80" TO LABEL-EXPDATE.
END DECLARATIVES.
        .
        .
        .
```

In the above example, if an error occurs during execution of an OPEN, CLOSE, READ, WRITE, or REWRITE statement referencing FILE- IN, and no AT END phrase was used in the statement, the USE procedure, IN-FILE-ERR, is executed.

When OUTFILE is opened, the second USE procedure, FILE-LABEL, is executed. This procedure creates a user label, and as an implicit part of its operation, writes the label on the file.

For more information on USE procedures, see the USE statement in section XI.

Procedures

A procedure consists either of one or more paragraphs, or of one or more sections.

This implies that if one paragraph appears in a section, then all must appear in sections. That is, either the procedure consists entirely of sections, or entirely of paragraphs, but not both.

A procedure-name is a word chosen by you to refer to a paragraph or section in the source program in which it occurs. A procedure-name consists of a section-name, or a paragraph-name (which may be qualified).

The physical end of the Procedure Division is that physical position in the source program after which no further procedures appear.

Sections And Section Headers

A section consists of a section header followed by zero or more successive paragraphs. A paragraph consists of a paragraph name followed by a period, a space, and zero or more successive sentences. (Paragraphs, sentences, statements, and so forth are described in Section II).

The paragraphs (if any), that make up a section should consist of closely related operations designed to collectively perform a particular function.

In the Procedure Division, a section header consists of a section-name of your choosing followed by a space, the word SECTION, another space, an optional segment number, a period, and a space, in that order. For example,

```
UNIT-1 SECTION 0.
```

is a valid section header, whereas

```
UNIT-0.
```

missing the key word, SECTION, is not.

The segment-number appearing in a section header is used to segment the Procedure Division.

Segmentation

Segmentation is the method available for physically separating the Procedure Division of an object program into pieces (segments) of logically related code.

This method is very useful on HP computer systems, as it allows you to load the minimum amount of code possible into memory during execution of a run unit.

It also allows you, through the use of the Segmenter Subsystem to activate, deactivate, add, or delete segments from a given object program. (See the Using COBOL Guide, part number 32233- 90003, and the Reference Manual, part number 30000- 90011 for more details.)

You can create segments in a COBOL program by using segment-numbers in section headers of the Procedure Division. All source paragraphs containing the same segment-number in their section headers form a single segment when the source file is compiled.

Since segment numbers range from 00 to 99, it is possible to segment any COBOL object program into a maximum of 100 segments. However, MPE limits program files to 63 segments. The remaining segments, if any, must be placed in a segmented library.

When you wish to use segmentation in a program, you must code the entire Procedure Division in sections. Although ANSI standard COBOL II allows for fixed permanent, fixed overlayable, and independent segments, such segment types have no meaning on HP computer systems. Thus, these terms are neither defined nor used here.

Segment Numbers

The segment-number appearing in a section header of the Procedure Division specifies that the section in which it appears is part of a segment whose number is the value specified by segment- number. Thus, all sections with the same segment number constitute a program segment. If no segment-number is specified, COBOL assumes it to be 0. Sections in the DECLARATIVES portion of a Procedure Division must have segment numbers less than 50. Although sections with the same segment-numbers are a part of the same segment, they need not be physically contiguous in the source program.

The term "initial state" refers to the original setting of GO TO statements before they are modified at run time by the ALTER statement (see the ALTER statement, Section XI).

A segment with a segment-number from 0 to 49 is in its initial state only when it is first used in a given run unit. Upon subsequent entries into such a segment, its state is the same as when it was exited from a previous usage.

With three exceptions, a segment with a segment number from 50 to 99 is always in its initial state whenever control is transferred to that section. The first exception concerns the appearance of a SORT, MERGE, or PERFORM statement, or any statement that implicitly calls a USE procedure, in a section whose segment number is greater than 49. When one of these statements implicitly transfers control to a procedure outside of the segment in which it appears, the segment is reentered in its last used state following the execution of the procedure.

The second exception is when a subprogram is called from a section whose segment number is greater than 49. In this case if the EXIT PROGRAM or GOBACK statement is executed in the called program, the calling program is reentered at the statement following the CALL statement. If this statement is within the same segment as the CALL statement, the segment is in its last used state when it is reentered.

The third exception is when a PERFORM statement of the form, PERFORM...n TIMES, is executed. If the sections or paragraphs named in the PERFORM statement have segment numbers greater than 49, then the segment of which they are a part is in its initial state the first time it is executed. It remains in its last used state for all subsequent (n-1) executions. Of course, following the completion of the PERFORM ...n TIMES, the associated segment is entered again in its initial state.

Note that since segments with segment numbers greater than 49 are always (with the noted exceptions) in their initial states when used, you must initialize each such section when control is passed to it, thus lengthening execution time. Modifying a GO TO statement in such a section from *outside* by using an ALTER statement in another section is impossible, since all GO TO statements in that section will be set to their initial state once control is passed to it.

Sections frequently used in a program should be given segment numbers less than 50, and sections which frequently communicate with other sections should be put in the same segment.

PROCEDURE DIVISION STATEMENTS AND SENTENCES

There are three types of statements, and three types of corresponding sentences in the Procedure Division. The three types are:

- Conditional statements and sentences;
- Compiler directing statements and sentences;
- Imperative statements and sentences.

CONDITIONAL Statements And Sentences

A conditional statement specifies that a condition is to be tested, and, depending upon the truth value of the condition, determines the subsequent action of the object program.

There are seventeen conditional statements in COBOL II. They are:

- An IF, SEARCH, or RETURN statement;
- A READ statement specifying the AT END or INVALID KEY phrase;
- A WRITE statement specifying the INVALID KEY or END-OF-PAGE phrase;
- A START, REWRITE, or DELETE statement specifying the INVALID KEY phrase;
- An arithmetic statement (ADD, COMPUTE, DIVIDE, MULTIPLY, or SUBTRACT) specifying the SIZE ERROR phrase;
- A RECEIVE statement specifying a NO DATA phrase;
- A STRING, UNSTRING, or CALL statement specifying the ON OVERFLOW phrase.

A conditional sentence is a conditional statement, optionally preceded by an imperative statement, terminated by a period followed by a space.

COMPILER DIRECTING Statements And Sentences

A compiler directing statement consists of a compiler directing verb (either COPY or USE) followed by the verb's operands. It causes the compiler to take a specific action during compilation.

A compiler directing sentence is a single compiler directing statement terminated by a period followed by a space.

IMPERATIVE Statements And Sentences

An imperative statement indicates a specific unconditional action to be taken by the object program. Any statement in COBOL that is neither a conditional statement nor a compiler directing statement is an imperative statement.

An imperative statement is a sequence of one or more imperative statements, each possibly separated from the other by a separator.

Note that when the phrase, imperative-statement, appears in a format, it refers to that sequence of consecutive imperative statements that must be ended in one of three ways. These three ways are:

- By a period;
- By an ELSE phrase associated with a previous IF statement;
- By a WHEN phrase associated with a previous SEARCH statement.

An imperative sentence is an imperative statement terminated by a period followed by a space.

Verbs used in forming imperative statements are shown in table 10-1 on the following page.

TABLE 10-1.
IMPERATIVE VERBS

ACCEPT	EXIT	SEEK
ADD (1)	GO	SET
ALTER	INSPECT	SORT CALL (3)
MERGE	START (2)	CANCEL
MOVE	STOP CLOSE	MULTIPLY (1)
STRING (3)	COMPUTE (1)	OPEN
SUBTRACT (1)	DELETE (2)	PERFORM
UNSTRING (3)	DISPLAY	READ (4)
WRITE (5)	DIVIDE (1)	RELEASE
EXAMINE	REWRITE (2)	

(1) Without the optional SIZE ERROR phrase.
(2) Without the optional INVALID KEY phrase.
(3) Without the optional ON OVERFLOW phrase.
(4) Without the optional AT END phrase or INVALID KEY phrase.
(5) Without the optional INVALID KEY phrase.

Categories Of Statements

All COBOL II/3000 statements fall into ten categories. These categories, and the verbs used in them, are listed in table 10-2.

Table 10-2.
CATEGORIES OF STATEMENTS

CATEGORY	VERBS
ARITHMETIC	ADD COMPUTE DIVIDE INSPECT (TALLYING) EXAMINE (TALLYING) MULTIPLY SUBTRACT
COMPILER DIRECTING	COPY USE
CONDITIONAL	ADD (SIZE ERROR) CALL (OVERFLOW) COMPUTE (SIZE ERROR) DELETE (INVALID KEY) DIVIDE (SIZE ERROR) IF MULTIPLY (SIZE ERROR) READ (END or INVALID KEY) RETURN (END) REWRITE (INVALID KEY) SEARCH START (INVALID KEY) STRING (OVERFLOW) SUBRTACT (SIZE ERROR) UNSTRING (OVERFLOW) WRITE (INVALID KEY or END-OF-PAGE)
DATA MOVEMENT	ACCEPT (DATE, DAY, or TIME) EXAMINE (REPLACING) INSPECT (REPLACING) MOVE STRING UNSTRING
ENDING	STOP STOP RUN GOBACK (in main program)

Table 10-2 (continued)

CATEGORY	VERBS
INPUT-OUTPUT	ACCEPT (identifier) CLOSE COBOLLOCK COBOLUNLOCK DELETE DISPLAY EXCLUSIVE OPEN READ REWRITE SEEK START STOP (literal) UN-EXCLUSIVE WRITE
INTERPROGRAM COMMUNICATION	CALL CANCEL ENTRY EXIT PROGRAM GOBACK
ORDERING	MERGE RELEASE RETURN SORT
PROCEDURE BRANCHING	ALTER CALL EXIT GOBACK GO TO PERFORM
TABLE HANDLING	SEARCH SET

ARITHMETIC EXPRESSIONS

Arithmetic expressions are used in the COMPUTE statement and relation conditions. They provide you with the ability to use exponentiation, as well as the addition, subtraction, multiplication division, and negation operations which can be performed using arithmetic statements.

Arithmetic expressions allow you to combine arithmetic operations without the restrictions on "composites of operands", and receiving data items that exist for arithmetic statements.

The maximum number of digits in arithmetic expressions is 28. If, in an intermediate result, this number of digits is exceeded, the result is truncated on the left by the number of digits required to attain a 28 digit number.

The number of decimal places used in evaluating an arithmetic expression is determined by the maximum number of decimal places within the expression and within the operand of a COMPUTE statement intended to receive the result.

An arithmetic expression can be any of the following:

- An identifier of a numeric elementary item;
- A numeric literal;
- Identifiers and literals as described above separated by arithmetic operators;
- Two arithmetic expressions separated by an arithmetic operator;
- An arithmetic expression enclosed in parentheses.

Any arithmetic expression may be preceded by a unary operator.

Arithmetic Operators

There are five binary and two unary arithmetic operators. Each is represented by a specific character or characters. When an arithmetic operator is used, it must be preceded and followed by a space.

The binary operators are:

+	symbolizing addition;
−	symbolizing subtraction;
*	symbolizing multiplication;
/	symbolizing division;
**	symbolizing exponentiation.

The unary operators are:

+	which is equivalent to multiplying by +1;
−	which is equivalent to multiplying by −1.

An arithmetic expression may only begin with a left parenthesis, a plus or minus sign, or an identifier or numeric literal. It may only end with an identifier or numeric literal, or with a right parenthesis.

Also, there must be a one-to-one correspondence between left and right parentheses, and each left parenthesis must be to the left of its corresponding right parenthesis.

Hierarchy Of Operations

When parentheses are not used, or entirely enclose an arithmetic expression, the order in which the various operands are applied in evaluating the expression is determined in the following manner:

A. Any unary operator used (+ or −) is executed first.

B. Following the execution of a unary operator, any exponentiation specified in the expression is executed.

C. Next, if multiplication or division is specified, the multiplication or division is executed. If consecutive multiplications and/or divisions are specified, each operation is performed in turn, starting from the left and continuing until the right most multiplication or division has been performed.

D. Following the execution of multiplication or division operations, any addition or subtraction specified in the expression is performed next. As with multiplication and division, if any consecutive combination of these operators is used, evaluation begins with the left most, and terminates with the execution of the right most operator in the consecutive list.

In general, when the sequence of execution of an arithmetic expression is not specified by parentheses, the order of execution of consecutive operations of the same hierarchical level is from left to right.

For example, the arithmetic expression,

```
-5+3**2*4+7-21
```

is evaluated as follows:

−5 is evaluated, resulting in −5.

3**2 is evaluated, resulting in 9.

9*4 is evaluated, resulting in 36.

−5+36 is evaluated, resulting in 31.

31+7 is evaluated resulting in 38.

38−21 is evaluated, resulting in 17.

Thus,

```
-5+3**2*4+7-21
```

is equivalent to

```
-5+36+7-21.
```

USE OF PARENTHESES

Parentheses may be used in arithmetic expressions to specify the order in which elements are to be evaluated. They are always used in pairs, with a compile time error message being issued if they are not.

Expressions within parentheses are evaluated first, and within nested parentheses, evaluation begins with the innermost set of parentheses, and continues outward until the expression contained in the outermost set is evaluated.

Use of parentheses allows you to eliminate ambiguities in logic where consecutive operations of the same hierarchical level appear, or to modify the normal hierarchical sequence of execution in an arithmetic expression.

To illustrate the use of parentheses, we use the arithmetic expression from the preceding example.

```
-5+3**2*4+7-21
```

is exactly equivalent to

```
(((-5+((3**2)*4))+7)-21)
```

both result in 17.

However,

```
-5+3**(2*4)+7-21
```

is equivalent to

```
-5+3**8+7-21
```

which results in 6542, and

```
(-5+3)**2*4+7-21
```

is equivalent to

```
-2**2*4+7-21
```

which results in 2.

Valid Combinations In Arithmetic Expressions

The ways in which operators, variables, and parentheses may be combined in an arithmetic expression are summarized in table 10–3 below.

The letter "P" indicates a permissible pair of symbols; the bar (–) indicates an invalid pair.

The word, "variable" indicates a numeric literal or an identifier of a numeric elementary item.

Table 10-3.
COMBINATION OF SYMBOLS IN ARITHMETIC EXPRESSIONS

FIRST SYMBOL	SECOND SYMBOL				
	Variable	* / ** – +	Unary + or –	(	)
Variable	–	P	–	–	P
* / ** + –	P	–	P	P	–
Unary + or –	P	–	–	P	–
(	P	–	P	P	–
)	–	P	–	–	P

CONDITIONAL EXPRESSIONS

Conditional expressions identify conditions to be tested in order to enable your object program to select alternate paths of control. This selection is determined by the truth value of a condition.

Conditional expressions are used in the IF, SEARCH, and PERFORM statements.

There are two categories of conditions associated with conditional expressions.

The first is simple conditions; the second is complex conditions.

You can enclose either category within any number of paired parantheses without changing its category.

Simple Conditions

There are five simple conditions:

- Sign condition;
- Class condition;
- Switch-status condition;
- Relation condition;
- Condition-name condition.

SIGN CONDITION

The sign condition tests whether or not the algebraic value of an arithmetic expression is less than, greater than, or equal to zero.

Sign Condition Format

arithmetic-expression IS [NOT] { POSITIVE | NEGATIVE | ZERO }

Where

arithmetic-expression is any valid arithmetic expression, as described on the preceding pages. It must contain at least one reference to a variable.

NOT coupled with one of the next key words, means to algebraically test *arithmetic expression*, and

If NOT POSITIVE is specified, return a value of 'true' if *arithmetic-expression* is negative or equal to zero; return a value of 'false' otherwise.

If NOT NEGATIVE is specified, return a value of 'true' if *arithmetic-expression* is equal to zero or positive; return a value of 'false' otherwise.

If NOT ZERO is specified, return a value of 'true' if *arithmetic-expression* is positive or negative; return a value of 'false' if *arithmetic-expression* is equal to zero.

POSITIVE NEGATIVE and ZERO Each used without the NOT key word means to algebraically test *arithmetic-expression* and

If POSITIVE is specified, return a value of 'true' if *arithmetic-expression* is greater than zero; return a value of 'false' otherwise.

If NEGATIVE is specified, return a value of 'true' if *arithmetic-expression* is less than zero; return a value of 'false' otherwise.

If ZERO is specified, return a value of 'true' if *arithmetic-expression* is equal to zero; return a value of 'false' otherwise.

To illustrate the sign condition, we assume that the variable A identifies the numeric value, -5.

```
IF A IS ZERO NEXT SENTENCE;
     ELSE DIVIDE A INTO SUMS.
```

In this example, since A is not zero, the statement, DIVIDE A INTO SUMS is executed. If A were zero, the sentence immediately following the condition sentence would be executed.

CLASS CONDITION

The class condition determines whether an operand consists entirely of characters selected from the set 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, and an operational sign (that is, whether the operand is numeric), or consists entirely of characters selected from the English alphabet (upper and lower case), and a space (that is, alphabetic characters).

Class Condition Format

identifier IS [NOT] { NUMERIC | ALPHABETIC }

Where

identifier names the operand to be tested. It must be described (implicitly or explicitly) as USAGE IS DISPLAY. Other restrictions apply if the key word NUMERIC is used. As an extension to the ANSI standard, COBOL II/3000 allows COMPUTATIONAL-3 items to be tested for NUMERIC.

NOT coupled with one of the next keywords is evaluated as follows:

if NOT NUMERIC is specified, a value of 'true' is returned if the operand contains some character other than the numerals 0 through 9 and a single operational sign; a value of 'false' is returned otherwise.

if NOT ALPHABETIC is specified, a value of 'true' is returned if the operand contains some character other than a space and letters of the English alphabet; a value of 'false' is returned otherwise.

ALPHABETIC means a value of 'true' is returned if the operand consists entirely of characters selected from the English alphabet (upper and lower case) and a space; a value of 'false' is returned otherwise.

NUMERIC means a value of 'true' is returned if the operand consists entirely of numerals selected from the set 0 through 9 and a single operational sign.

You cannot use a NUMERIC test if the operand has a data description defining it as alphabetic, or as a group item composed of elementary items whose data descriptions indicate the presence of an operational sign or signs.

If the data description of the operand does not indicate the presence of an operational sign, the operand is considered numeric only if it consists of numerals, and has no operational sign.

If the data description of the operand does indicate an operational sign, the operand is considered numeric only if it consists of numerals from the set 0 through 9, and a single valid operational sign.

Valid operational signs are determined by the presence or absence of the SIGN IS SEPARATE clause in the data description of the operand.

If the SIGN IS SEPARATE clause is present, the valid operational signs are the standard data format characters, + and −.

If the SIGN IS SEPARATE clause is not present, the valid operational signs in standard data format are shown in table 9-5 of section IX under the heading, USAGE IS DISPLAY.

The ALPHABETIC test cannot be used with an operand whose data description describes it as numeric.

To illustrate the class condition, we use an operand which, in standard data format, is 35798D.

Data description of operand:

```
01 FIRST-NUMBER  PIC S9(6)  SIGN IS TRAILING.
```

Condition test:

```
FIRST-NUMBER IS NUMERIC
```

In this case, the test returns a value of 'true', since D is a valid operational sign. D has the value +4, thus making the numeral 35798D equivalent to +357984.

SWITCH-STATUS CONDITION

A switch-status condition determines the on or off status of a defined switch. The function-name and the on or off value associated with the condition must be specified in the SPECIAL-NAMES paragraph of the Environment Division.

Switch-Status Condition Format

condition-name

Where

condition-name is the name associated with the function-name in the SPECIAL-NAMES paragraph of the Environment Division.

The result of the test is true if the switch is set to the specified position corresponding to *condition-name*.

To illustrate:

```
ENVIRONMENT DIVISION.
        .
        .
        .
SPECIAL NAMES.
    SW0, OFF STATUS IS NOADD, ON STATUS IS ADDONE.
        .
        .
        .
PROCEDURE DIVISION.
        .
        .
        .
PRINT ROUTINE.
     IF NOADD THEN PERFORM OTHER-ACTION.
        .
        .
        .
```

In the above example, if the switch-status of the condition-name, NOADD, is "off", then a routine named OTHER-ACTION is performed. If the switch-status of the NOADD is "on", then OTHER-ACTION is not performed, and control passes to the next executable statement.

RELATION CONDITIONS

There are two types of relation conditions in COBOL II/3000. One is ANSI standard; the other is used for condition code checking after intrinsic calls. The intrinsic relation condition is discussed following the discussion of ANSI standard relation conditions.

ANSI Standard Relation Conditions

A relation condition compares two operands, each of which may be a data item referenced by an identifier, a literal, or the value resulting from an arithmetic statement.

If a specified relation exists between the two operands, the relation condition value is "TRUE."

You may compare two numeric operands, regardless of their respective usages; however, if you want to compare two operands, and one of them is not numeric, then both must have the same usage. Note that since group items are treated as alphanumeric, nonnumeric comparison rules apply.

A relation condition must contain at least one reference to a variable.

General Format

```
                           { IS [NOT] GREATER THAN }
{ identifier-1           } { IS [NOT] LESS THAN    } { identifier-2           }
{ literal-1              } { IS [NOT] EQUAL TO     } { literal-2              }
{ arithmetic-expression-1} { IS [NOT] >            } { arithmetic-expression-2}
                           { IS [NOT] <            }
                           { IS [NOT] =            }
```

NOTE

The required relational characters '>', '<', and '=' are not underlined to avoid confusion with other symbols such as '≥' (greater than or equal to).

Where

Identifier-1 or ***literal-1*** or ***arithmetic-expression-1***	is the subject of the condition.
Identifier-2 or ***literal-2*** or ***arithmetic-expression-2***	is the object of the condition.
[NOT] **GREATER THAN**	is equivalent to [NOT] >
[NOT] **LESS THAN**	is equivalent to [NOT] <
[NOT] EQUAL	is equivalent to [NOT] =
NOT	coupled with the next key word or relation character has the following meaning: NOT GREATER or NOT > means less than or equal; NOT LESS or NOT < means greater than or equal; NOT EQUAL or NOT = means greater than or less than.

The relational operator specifies the type of comparison to be made. A space must precede and follow each reserved word comprising the relational operator.

COMPARISON OF NUMERIC OPERANDS

For operands belonging to the numeric class, a comparison is made with respect to the algebraic values of the operands. The number of digits in an operand is not significant. Also, no distinction is made between a signed or unsigned value of zero.

Comparison of numeric operands is not affected by their usages. Unsigned numeric operands are considered to be positive when they are used as operands in a comparison.

COMPARISONS USING INDEX-NAMES AND INDEX DATA ITEMS

Relation tests may be made using index-names and index data items. Those which are permissible are:

1. Two index-names. The result is the same as if the corresponding occurrence numbers were compared.
2. An index-name and a data item (other than an index data item) or literal. The occurrence number corresponding to the value of the index-name is compared to the data item or literal.
3. An index data item and an index-name or another index data item. The actual values are compared without conversion.

An index data item should only be compared with another index data item or an index-name. Comparison of an index data item with any other data item or a literal gives an undefined result.

COMPARISON OF NONNUMERIC OPERANDS

For nonnumeric operands, or one nonnumeric operand and one numeric operand, a comparison is made with respect to the ASCII character set collating sequence.

If one of the operands is numeric, it must be an integer data item or an integer literal, and it must have the same usage as the nonnumeric operand.

If the nonnumeric operand is an elementary data item or a literal, the numeric operand is treated as though it were moved to an elementary alphanumeric data item of the same size as the numeric data item, and the contents of this alphanumeric data item were then compared to the nonnumeric operand.

If the nonnumeric operand is a group item, the numeric operand is treated as though it were moved to a group item of the same size as the numeric data item, and the contents of this group item were then compared to the nonnumeric operand. Recall that a group item is always classified as alphanumeric.

NOTE

> In the previous paragraphs, "the same size as the numeric data item" means the size of the numeric data item in standard data format. If the P character of the PICTURE clause is included in the description for the numeric operand, recall that it must not be included in determining the size of the operand.

The size of an operand is the total number of standard data format characters in the operand.

When operands are of unequal size, comparison proceeds as though the shorter operand were extended on the right by sufficient spaces to make the operands of equal size.

When operands are of equal size (or have been adjusted as described in the preceding paragraph), comparison proceeds on a character-by-character basis, starting with each leftmost character, and continuing until either the last character of each operand have been compared, or a pair of unmatched characters is found.

The operands are considered equal if each pair of characters match, from the leftmost to the rightmost.

The first time a pair of characters is found to be unequal (that is, do not match), their positions in the program collating sequence are located, and the character having the numerically larger index in the collating sequence is considered to be greater than the other character.

To illustrate, we use the following data items:

```
01 SUBJECT   PIC X(06) VALUE  'FLAXEN'.
01 OBJECT    PIC X(07) VALUE  'FLATTER'.
```

The relative condition is

```
SUBJECT IS EQUAL TO OBJECT
```

The comparison takes place as follows:

```
F L A X E N
↕ ↕ ↕ ↕
F L A T T E R
```

F matches F; therefore, proceed.
L matches L; therefore, proceed.
A matches A; therefore, proceed.
X does not match T; therefore find the indices of each
in the ASCII collating sequence:

Index of X: 88
Index of T: 84

X is greater than T; thus, FLAXEN is greater than FLATTER,

and the relation condition above returns a 'false' value.

INTRINSIC RELATION CONDITIONS

Intrinsic relation conditions are used only to test the condition codes returned by MPE intrinsics and SPL procedures after they have been called through the use of CALL statements.

Intrinsic Relation Condition Format

$$\textit{mnemonic-name} \quad [\ \text{NOT}\ ] \left\{ \begin{array}{c} < \\ = \\ > \end{array} \right\} \ \mathbf{0}$$

NOTE: The required relational characters '>', '<', and '=' are not underlined to avoid confusion with other symbols.

Where

mnemonic-name is a name chosen by you to represent the CONDITION-CODE function, and must be defined in the SPECIAL-NAMES paragraph of the Environment Division.

If *mnemonic-name* is equal to 0, execution of the intrinsic was successful.

If *mnemonic-name* is not 0, then some error probably occurred. Specific meanings are numerous, since they vary from intrinsic to intrinsic. Refer to the MPE Intrinsics Reference Manual, part number 30000-90010, for meanings of the values returned.

CONDITION-NAME CONDITIONS

In a condition-name condition, a conditional variable is tested to determine whether or not its value is equal to one of the values associated with condition-name.

Condition-Name Condition Format

condition-name

Where

condition-name is an identifier described under an 88 level data description entry in the Data Division.

If *condition-name* is associated with a range of values, then the conditional variable is tested to determine whether or not its value falls in this range, including the end values.

The rules for comparing a conditional variable with a condition-name are the same as those specified for relation conditions.

The result of such a comparison is true if one of the values of *condition-name* equals the value of its associated conditional variable.

To illustrate condition-name conditions:

```
      .
      .
DATA DIVISION.
      .
      .
      .
01 CON-VAR         PICTURE 999.
88 CON-NAME1;
     VALUES ARE 001 THRU 100.
             .
             .
             .
PROCEDURE DIVISION.
             .
             .
             .
    IF CON-NAME1 THEN PERFORM UNDER-VALUE;
     ELSE NEXT STATEMENT.
             .
             .
             .
```

In the above example, the test is performed to see if CON-VAR has a value of 1, 100, or any number between 1 and 100. If it does, a procedure named UNDER-VALUE is performed. Otherwise, the next sentence is executed.

COMPLEX CONDITIONS

A complex condition is formed by using the logical operators, AND and OR, to combine simple conditions, combined conditions, and/or complex conditions, or by negating these conditions with the logical negation operator, NOT.

The truth value of a complex condition, regardless of the use of parentheses, is that truth value which results from the interaction of all the stated logical operators on the individual truth values of simple conditions, or the intermediate truth values of conditions logically connected or negated.

The meanings of the three logical operators are listed below.

LOGICAL OPERATOR	MEANING
AND	Logical conjunction; the truth value is 'true' if both of the conjoined conditions are true; 'false' if one or both of the conjoined conditions is false.
OR	Logical inclusive OR; the truth value is 'true' if one or both of the included conditions is true; 'false' if both included conditions are false.
NOT	Logical negation or reversal of truth value; the truth value is 'true' if the condition is false; 'false' if the condition is true.

When a logical operator is used, it must be preceded and followed by a space.

Negated Simple Conditions

You can use the NOT operator to negate simple conditions.

A negated simple condition has a value of "true" if and only if the value of the simple condition is "false". Conversely, a negated simple condition has a value of "false" if and only if the simple condition itself has a value of "true".

Negated Simple Condition Format

NOT ***simple-condition***

Where

simple-condition is as described on the preceding pages.

To illustrate a negated simple condition, we write a statement which is equivalent to the IF statement used in our last example.

```
IF CON-NAME1 THEN PERFORM UNDER-VALUE;
        ELSE NEXT STATEMENT
```

is equivalent to

```
IF NOT CON-NAME1 THEN NEXT STATEMENT;
        ELSE PERFORM UNDER-VALUE.
```

Combined Conditions

You can form a combined condition by connecting conditions with one of the logical operators, AND and OR.

Combined Condition Format

$$\left\{ \textit{condition} \quad \left\{ \begin{matrix} \underline{\textbf{AND}} \\ \underline{\textbf{OR}} \end{matrix} \right\} \quad \textit{condition} \right\} \ldots$$

Where *condition* is one of five possible types of conditions:

- A simple condition;
- A negated simple condition;
- A combined condition;
- A negated combined condition (a negated combined condition is a combined condition enclosed by parentheses and preceded by the NOT logical operator);
- Combinations of any of the four types listed above, as specified in table 10-4 on the following page.

Although you need not use parentheses when you form a combined condition using just AND or just OR, you may use parentheses to clarify, and to affect the final result of a combined condition.

Table 10-4 indicates the ways in which conditions and logical operators may be combined and parenthesized.

There must be a one-to-one correspondence between left and right parentheses, and any left parenthesis must be to the left of its corresponding right parenthesis.

Table 10-4.
COMBINATIONS OF CONDITIONS, LOGICAL OPERATORS, AND PARENTHESES

Given the following element	Location in conditional expression		In a left-to-right sequence of elements:	
	First	Last	Element, when not first, may be immediately preceded by only:	Element, when not last, may be immediately followed by only:
simple-condition	Yes	Yes	OR, NOT, AND, (	OR, AND,)
OR or AND	No	No	simple-condition,)	simple-condition, NOT, (
NOT	Yes	No	OR, AND, (	simple-condition,
(	Yes	No	OR, NOT, AND, (	simple-condition, NOT, (
)	No	Yes	simple-condition,	OR, AND,)

As an illustration of the use of the table, note that the pair, OR NOT is acceptable, whereas NOT OR is not.

CONDITION EVALUATION RULES

You may use parentheses to specify the order in which individual conditions of complex conditions are to be evaluated when you wish to modify the precedence rules (as listed below) for evaluating such conditions.

Conditions within parentheses are evaluated first. Within nested parentheses, the condition bounded by the inner-most set is evaluated first, followed by the condition within the next innermost set, and continuing until the condition within the outermost set of parentheses is evaluated.

When parentheses are not used, or when they completely contain a condition, the following rules are used to determine the truth value:

1. Arithmetic expressions are evaluated.
2. Truth values for simple conditions are established next, in the following order:
   a) Relation conditions;
   b) Class conditions;
   c) Condition-name conditions;
   d) Switch-status conditions;
   e) Sign condition.
3. Truth values for negated simple conditions are established next.
4. Truth values for combined conditions are established next, using the AND operator first, and the OR operator second.
5. Finally, truth values for negated combined conditions are established.

When a sequence of evaluation is not completely specified by parentheses, the order of evaluation of consecutive operations in the same hierarchical level is from left to right.

For example, the expression,

```
A IS NOT ALPHABETIC AND A IS NOT GREATER THAN B*3 OR (NOT C AND NOT D)
```

is evaluated in the following way:

1. The class condition,

   ```
   A IS NOT ALPHABETIC
   ```

   is tested. Assuming this condition is true, the result is

   "true" `AND A IS NOT GREATER THAN B*3 OR (NOT C AND NOT D).`

2. The relation condition,

   ```
   A IS NOT GREATER THAN B*3
   ```

   is evaluated. Since B*3 is an arithmetic expression, it must be evaluated before evaluating the entire relation condition. Thus, assuming that B*3 results in a value, v, the relation condition becomes

   ```
   A IS NOT GREATER THAN v.
   ```

   Assuming that this value is again true, the result is

   "true" and "true" OR (NOT C AND NOT D)

3. The expression inside parantheses is evaluated next. Thus,

   ```
   NOT C AND NOT D
   ```

   is evaluated. Since both NOT C and NOT D are condition-name conditions, NOT C is evaluated first, followed by NOT D. Assuming that both are true, this results in the compound condition,

   "true" AND "true"

   which in turn results in "true".

4. The final step is to evaluate the compound condition resulting from the previous evaluations. That is,

   "true" AND "true" OR "true"

   must be evaluated.

   "true" AND "true" are evaluated first, giving "true".

   Finally, "true" OR "true" is evaluated, giving "true". Thus, under the above assumptions, the original expression gives a "true" result.

ABBREVIATED COMBINED RELATION CONDITIONS

If you combine simple or negated simple relation conditions with logical connectives (AND and OR) in a consecutive sequence in such a way that

A. no parentheses are used in the consecutive sequence, and

B. a succeeding relation condition contains the same subject as the preceding relation condition, or

C. a succeeding relation condition contains the same subject and the same relational operator,

then you can abbreviate any relation condition, except the first, within the consecutive sequence.

There are two ways by which you can abbreviate such relation conditions.

The first is by omitting the subject of the relation condition; the second is by omitting the subject and the relational operator of the relation condition.

Abbreviated Combined Relation Condition Format

Relation-condition { **AND** | **OR** } [**NOT**] [***relational-operator***] ***object*** ...

The effect of using such abbreviations is as if the last preceding stated subject were inserted in place of the omitted subject, and the last stated relational operator were inserted in place of the relational operator.

In order to assure that an abbreviated relation condition is valid, insert the omitted subject and relational operator. If, after insertion, the combined relation condition is valid according to the rules in table 10-4, the abbreviated relation condition is valid.

The end of an abbreviated relation condition is signified by the first occurrence of a complete simple condition within a complex condition.

The word NOT can be used in two different ways; that is, as part of a relational operator, and as the logical negation operator. The rules for its usage in an abbreviated combined relation condition are:

a. if the word immediately following NOT is one of GREATER, LESS, EQUAL, or one of the equivalent symbols (>, <, =), then NOT participates as part of the relational condition;

b. if the word immediately following NOT is not one of those listed in the above paragraph, it is considered to be the negation operator. Thus, it negates only the first occurrence of the abbreviated relation condition.

The examples below serve to clarify abbreviated combined relation conditions.

```
A > B AND NOT < C OR D
```

is equivalent to

```
((A > B) AND (A NOT < C)) OR (A NOT < D)
```

```
A NOT EQUAL B OR C
```

is equivalent to

```
(A NOT EQUAL B) OR (A NOT EQUAL C)
```

```
NOT A = B OR C
```

is equivalent to

```
(NOT (A = B)) OR (A = C)
```

```
NOT (A GREATER B OR < C)
```

is equivalent to

```
NOT ((A GREATER B) OR (A < C))
```

```
NOT (A NOT > B AND C AND NOT D)
```

is equivalent to

```
NOT ((((A NOT > B) AND ( A NOT > C)) AND (NOT (A NOT > D))))
```

COMMON PHRASES

Several statements described in the next section use the ROUNDED, SIZE ERROR, and CORRESPONDING phrases.

In order to avoid describing each of these phrases each time they appear in a particular statement, we describe them now.

In the discussion that follows, the term resultant-identifier means the identifier associated with a result of an arithmetic operation.

ROUNDED PHRASE

The ROUNDED phrase consists entirely of the keyword, ROUNDED.

In an arithmetic operation, if, after decimal-point alignment, there are more decimal places in the fraction of the result than is specified for the resultant-identifier, truncation is performed on the result. The number of digits truncated is dependent upon the number of decimal places specified for the fractional part of resultant-identifier.

If you want to round the result before truncation occurs, you can use the ROUNDED option.

If the ROUNDED phrase is specified in an arithmetic operation, the absolute value of resultant-identifier is increased by 1 whenever the most significant digit of the excess portion of the result is greater than or equal to 5. The excess portion is then truncated.

When the low-order integer positions in a resultant-identifier are represented by the P character in the PICTURE clause of that resultant-identifier, rounding occurs relative to the right most integer position for which storage is allocated.

SIZE ERROR Phrase

The SIZE ERROR phrase has the format,

ON SIZE ERROR ***imperative-statement***

Where *imperative-statement* is one or more imperative statements.

If, after decimal point alignment, the number of digits in a result exceeds the number of digits specified for the associated resultant-identifier, a SIZE ERROR condition exists.

The *imperative-statement* is executed if A SIZE ERROR condition occurs.

The SIZE ERROR condition applies only to the final result of most arithmetic operations; it applies to intermediate results, however, when the MULITPLY and DIVIDE statements are used.

Note that division by 0 (zero) always forces a SIZE ERROR condition.

If the ROUNDED phrase is specified in an arithmetic operation, rounding is done before a SIZE ERROR check is performed.

When a SIZE ERROR condition occurs, and the SIZE ERROR phrase is not specified, the values of any resultant-identifiers affected are undefined.

If other resultant-identifiers are involved in a particular arithmetic operation for which a SIZE ERROR condition occurs, their values are unaffected; only the resultant-identifiers for which the SIZE ERROR occurs have undefined values.

Thus, for example, if the arithmetic operation,

```
ADD A TO B, C
```

forces a SIZE ERROR condition for B, but not for C, only B has an undefined value.

When the SIZE ERROR phrase is specified for an arithmetic operation, and a SIZE ERROR condition exists for the values of one or more of the resultant-identifiers involved, their values remain as they were before the operation was executed.

Values of other resultant-identifiers involved in the operation are unaffected by size errors. Thus, their values are changed according to the arithmetic operation specified.

The SIZE ERROR phrase includes an *imperative-statement* following the words, SIZE ERROR. This statement is executed following the occurrence of a size error in an arithmetic statement for which the SIZE ERROR phrase is specified.

To illustrate:

```
          .
          .
WORKING-STORAGE SECTION.

01 SIZE-ERR.
   02 NOTIFY    PIC X(10) VALUE 'SIZE ERROR'.
   02 PARAMETERS.
      03 PARM-1    PIC Z(18)  VALUE 0.
      03 PARM-2    PIC Z(18)  VALUE 0.
      03 PARM-3    PIC Z(18)  VALUE 0.
          .
          .
          .
PROCEDURE DIVISION.
          .
          .
          .
    ADD A, B TO C, D; ROUNDED;
       ON SIZE ERROR PERFORM NOTIFICATION.
          .
          .
          .
          .
NOTIFICATION.
    MOVE C TO PARM-1.
    MOVE D TO PARM-2.
    WRITE SIZE-ERR AFTER ADVANCING 1 LINE.
          .
          .
          .
```

If an ADD or a SUBTRACT statement uses the CORRESPONDING phrase as well as the SIZE ERROR phrase, and an operation produces a size error condition, the imperative statement in the SIZE ERROR phrase is not executed until all individual additions or subtractions are completed.

CORRESPONDING Phrase

The CORRESPONDING phrase consists entirely of the word CORRESPONDING, or of the equivalent abbreviation, CORR.

The purpose of the CORRESPONDING phrase is to allow you to add, subtract, or move a data item subordinate to a group item to a data item subordinate to some other group item.

Two data items are said to correspond if three conditions are met. For purposes of discussion, assume that D1 and D2 are group items.

A data item from D1 is said to correspond to a data item from D2 if:

1. Both of the data items have the same name, the name is not FILLER, and both have the same qualifiers up to, but not including D1 and D2.

2. When the CORRESPONDING phrase is being used in a MOVE statement, at least one of the data items is an elementary data item; when the CORRESPONDING phrase is used in an ADD or SUBTRACT statement, both data items are elementary data items.

3. The descriptions of D1 and D2 do not contain a 66, 77, or 88 level-number, and do not contain a USAGE IS INDEX clause.

Any data item which is a candidate for use in a CORRESPONDING phrase is ignored if, although it meets the three conditions above, it contains a REDEFINES, RENAMES, OCCURS, or USAGE IS INDEX clause. Furthermore, any data items subordinate to such a data item are also ignored.

These restrictions do not apply to D1 and D2, except as noted in condition 3, above.

To illustrate correspondence and its usage:

```
01  FIRST-DATA.
    02  ENTRY-1.
        03  ENTRY-1A      PIC 9(5)V99.
        03  ENTRY-1B      PIC 9(3)V99.
    02  ENTRY-2           PIC X(30)
                .
01  SECOND-DATA.
    02  ENTRY-1.
        03  ENTRY-1A      PIC 99V99.
        03  ENTRY-1B      PIC 999.
    02  FINISH            PIC X(20)
```

ENTRY-1A of FIRST-DATA corresponds to ENTRY-1A of SECOND-DATA, and ENTRY-1B of FIRST-DATA corresponds to ENTRY-1B of SECOND DATA.

ENTRY-1 of FIRST DATA does not correspond to ENTRY-1 of SECOND-DATA because of the second condition of correspondence.

The ADD statement below uses the CORRESPONDING phrase to add ENTRY-1A of FIRST-DATA to ENTRY-1A of SECOND-DATA, and ENTRY-2A of FIRST-DATA to ENTRY2A of SECOND-DATA. The results are stored in ENTRY-1A and ENTRY-2A of SECOND-DATA.

```
ADD CORRESPONDING FIRST-DATA TO SECOND-DATA.
```

COMMON FEATURES OF ARITHMETIC STATEMENTS

The five arithmetic statements, ADD, SUBTRACT, MULTIPLY, DIVIDE, and COMPUTE have several features in common. These features are:

1. The data descriptions of operands in an arithmetic statement need not be the same.

   If operands are of mixed types, the compiler generates any data conversion routines necessary to format the data. Note that this does, however, increase the size of your data stack.

   If the operands are already defined as computational synchronized, the compiler does not have to generate conversion routines. This reduces your object program size and its execution time. Thus, to maximize efficiency of arithmetic operations, you should define the operands as being COMPUTATIONAL SYNCHRONIZED, or at least with a usage of COMPUTATIONAL-3. (COMPUTATIONAL-3 is faster than COMPUTATIONAL if the number of digits in each operand is greater than 9.)

2. The maximum size of each operand is eighteen digits; however, the composite of operands must not contain more than eighteen decimal digits.

   The composite of operands is the hypothetical data item resulting from the superimposition of specified operands in an arithmetic statement after the operands have been aligned on their decimal points.

   For example, in format 1 of the ADD statement, the composite of operands is determined by using all of the operands in a given statement.

   Thus, given A = 1234567, B = 12359, and C = 1034077, the composite of operands of the statement, ADD A,B TO C is 103402359.

   This number was arrived at by selecting the operand with the greatest number of digits to the right of the decimal point (in this case, 12359), and then the operand with the greatest number of digits to the left of the decimal point (which is 1034077).

   These two operands were then superimposed, with the larger number to the left or right of the decimal point masking the smaller.

3. Arithmetic statements may have multiple results. For example, the ADD statement,

   ```
   ADD A TO B,C,D.
   ```

   gives the multiple results, A+B, A+C, and A+D.

   Such statements behave as though they had been written in the following way:

   a. A statement was first written which performs the specified arithmetic operation, and stores the results in a temporary location.

b. A sequence of statements was then written which transfers or combines the value in the temporary location with each of the single data items specified as a result in the original arithmetic statement. This hypothetical sequence of statements was written to perform the transfer or combining of the temporary value in the same left-to-right sequence as the multiple results are listed.

   To illustrate, the result of the statement,

```
ADD A,B,C  TO  C,D,E
```

   is equivalent to the statements,

```
ADD A,B,C GIVING TEMP
         ADD TEMP  TO C
         ADD TEMP  TO D
         ADD TEMP  TO E
```

   where "TEMP" is an intermediate result item.

Overlapping Operands and Incompatible Data

When a sending and a receiving data item in an arithmetic statement or an INSPECT, MOVE, SET, STRING, or UNSTRING statement share a part of their storage areas, the result of the execution of such a statement is undefined.

Furthermore, except for a class condition, when the contents of a data item are referenced in the Procedure Division and the contents of that data item are not compatible with the class specified by its PICTURE clause, the result of such a reference is undefined.

TABLES

Quite frequently in business applications, data is arranged in the form of tables. This often comes about from the logical arrangement of data, and, because it is easier to both describe and select elements of a table than it is to write all the components of the table as one record.

Tables composed of contiguous data items are defined in COBOL II/3000 by using the OCCURS clause in a record description entry.

In short, the OCCURS clause states how many elements there are in a table, gives these elements a common name, tells whether the elements are arrayed in ascending or descending order, and whether you want to use subscripting or indexing to access elements of the table.

You must use subscripting or indexing to access table elements because they share the same name.

Defining a Table

In COBOL II/3000, you may define a table of one, two, or three dimensions. This is accomplished by using the OCCURS clause one, two, or three times, respectively, within different level numbers (other than 01) of the description.

To define a one dimensional table, you use an OCCURS clause as part of the data description for the table itself. If you do not use the OCCURS clause as part of the first level description following the table name, the elements described before the OCCURS caluse are not part of the table. For example:

```
01 TABLE-1.
   02 TABLE1-HEADER         PIC X(20) VALUE "TABLE ONE".
   02 ELEMENT OCCURS 100 TIMES.
      03 TIME-ADUSTER                PIC X(10).
      03 DATE-ADJUSTER                PIC X(10).
```

TABLE1-HEADER is not an element of TABLE-1, whereas TIME and DATE are.

When you wish to define a two dimensional table, you must build it from a one dimensional table.

That is, to define a two dimensional table, you must use an OCCURS clause twice; once in an element of the first table, and a second time in the description of a group item containing that table element. To illustrate:

```
01 SHOW-TABLE.
   11 FIRST-DIM OCCURS 10 TIMES.
      22 DIM1-HEAD              PIC X(20).
      22 SECOND-DIM OCCURS 10 TIMES.
         25 TWODIM-ELEMENTS.
            30 MORE-ONE         PIC X(10).
            30 MORE-TWO         PIC X(07).
            30 MORE-THREE       PIC X(16).
```

The table element, SECOND-DIM, of the first table (FIRST-DIM) uses the OCCURS clause to define a second dimension of SHOW-TABLE while the group item, FIRST-DIM, to which SHOW-TABLE is subordinate defines the first dimension.

Defining a three dimensional table is analogous to defining a two dimensional table. You must simply extend the table elements of the two dimensional table to include an element which uses the OCCURS clause. To illustrate:

```
01 SALES-ORGANIZATION-TABLE.
   11 REGION-TABLE  OCCURS 4 TIMES.
      22 SALES-REGION                PIC X(05).
      22 STATE-TABLE OCCURS 13 TIMES.
         33 STATE          PIC X(20).
         33 REP-TABLE OCCURS 4 TIMES.
            34 REP-INFO.
               35 REPRESENTATIVE     PIC X(20).
               35 LOCATION-INFO      PIC X(60).
```

The table above, named SALES-ORGANZATION-TABLE, has one dimension for REGION-TABLE, two for STATE-TABLE, and three for REP-TABLE.

SALES-ORGANIZATION-TABLE contains 472 data items. There are four for REGION-TABLE, 52 for STATE-TABLE (4 times 13), and 208 for REP-TABLE (52 times 4 twice; one for each element name).

Referencing Table Items

As was mentioned earlier, there are two ways of accessing table items. You can use indexing, or subscripting.

The simplest of these two methods is subscripting. Subscripting uses an occurrence number (that is, the number of where in a particular dimension an element occurs) for each dimension of a table. To illustrate, using the three dimensional table from our previous example,

```
REPRESENTATIVE (4, 1, 3)
```

accesses the third occurrence of REPRESENTATIVE in the first state of the fourth sales region.

Thus, if the fourth sales region is the western sales region, and the first state in that region is California, then the name of the third representative for that state is accessed by the above subscripted reference.

Of course, if you wish only to access a state entry, you can do so by using only two subscripts. For example:

```
STATE (4, 12)
```

might reference Wyoming, the twelfth state in the western sales region.

Note that data names could as easily have been used to perform all or just part of the subscripting above. Generally, these data items are defined in working storage, and have no restrictions on them except that they cannot be index data names.

The second method of accessing table items is indexing. This method requires more coding in the OCCURS clause, since you must specify at least one name to be used for the indexing. Furthermore, if any one of the dimensions in a table uses indexing, then all dimensions must.

To illustrate this technique, we re-code the three dimensional table used above.

```
01 SALES-ORGANIZATION-TABLE.
   11 REGION-TABLE OCCURS 4 TIMES
        INDEXED BY RINDX.
          22 SALES-REGION                          PIC X(05).
          22 STATE-TABLE OCCURS 13 TIMES
               INDEXED BY SINDX.
                 33 STATE            PIC X(20).
                 33 REP-TABLE OCCURS 4 TIMES
                      INDEXED BY RPINDX.
                        34 REP-INFO.
                             35 REPRESENTATIVE      PIC X(20).
                             35 LOCATION-INFO       PIC X(60).
```

Once the index names have been defined, you must use the SET statement of the Procedure Division to initialize the index-names to a value within the range of from 1 to the highest occurence number associated with the dimension in which the index-name was defined. It is good practice also to reset the value of an index-name each time you use it, since, under certain conditions, its value may be undefined after use. Keeping track of the last value of an index-name is facilitated through the use of an index data name with the SET statement. See the USAGE clause in Section IX for more details.

Once an index-name has been set, you may use it to access table elements. For example, assuming that RPINDX has been set to 2,

```
REP-INFO(4, 1, RPINDX)
```

accesses the information about the second representative in the first state of the fourth sales region. Referring back to the use of this same data base in our subscript example, we see that this accesses the information about the second sales representative in California.

Note that with indexing, you must use index-names or literals to specify indices, but, unlike subscripting, you must not use data names that are not index-names.

You can use index-names in conjunction with the SEARCH statement of the Procedure Division to search for occurrences of table items within a given table. For information and restrictions on searching tables, see the SEARCH statement in Section XI.

PROCEDURE DIVISION STATEMENTS

SECTION XI

All statements that may be used in the Procedure Division are discussed in alphabetical order on the following pages.

ACCEPT STATEMENT

The ACCEPT statement can be used for low volume input from a specified device. It has three general formats, as shown below.

Format 1

```
                                     ┌        ┌ SYSIN         ┐ ┐
ACCEPT identifier [ FREE ]           │ FROM   │ CONSOLE       │ │
                                     └        └ mnemonic-name ┘ ┘
```

Format 2

```
                               ┌        ┌ SYSIN         ┐ ┐
ACCEPT identifier FREE         │ FROM   │ CONSOLE       │ │
                               └        └ mnemonic-name ┘ ┘

      ; ON INPUT ERROR imperative-statement
```

Format 3

```
                                          ┌ DATE ┐
ACCEPT identifier [ FREE]  FROM           │ DAY  │
                                          └ TIME ┘
```

Where

identifier is a valid data-name; it receives the data entered by the execution of the ACCEPT statement.

SYSIN is, in a batch or stream job, the card reader or input stream file, respectively; in a session, this name indicates the terminal used to initiate execution of your program. Note that there is no indication of a pending user response; thus, you should use the DISPLAY statement immediately before the ACCEPT statement to indicate that the ACCEPT statement is awaiting input.

CONSOLE is the operators console. When an ACCEPT statement is issued against this device, the message, "AWAITING REPLY" is displayed at the console.

mnemonic-name is a name assigned by you in the SPECIAL-NAMES paragraph of the Environment Division. It must be a name for either SYSIN or CONSOLE, and has the same effect as the device name to which it is equated.

imperative-statement is one or more imperative statements. The INPUT ERROR phrase in which it appears can only be used if the FREE phrase is used.

DATE is composed of the year of the century, month of year, and day of month, in that order. Thus, for example, February 16, 1980 is transmitted as 800216. COBOL moves this data as an unsigned elementary numeric integer data item six digits in length.

DAY is composed of the year of the century and day of the year, in that order. Thus, for example, February 16, 1980 is accessed as 80047. COBOL moves this data as an unsigned elementary numeric integer data item five digits in length.

TIME is the time of day, taken from a 24 hour clock, in hours, minutes, seconds and tenths of a second. The minimum value of time is 00000000, and the maximum is 23595990. COBOL moves this data as an unsigned elementary numeric integer data item eight digits long.

Accept Statement - Formats 1 And 2

When formats 1 and 2 are used, data is accepted from either a card reader or input spool file (if your program is running in batch mode), the terminal from which your program is executed (if it is running in session mode), or from the operator's console (if the CONSOLE option is used). This data is then used to replace the contents of the data-item named by *identifier.*

If you do not use the FROM phrase in your ACCEPT statement, the compiler assumes you are referencing the SYSIN device.

Free And Input Error Phrases

The FREE phrase allows you to use free-field format to enter data. This is an HP extension to ANSI COBOL'74.

The INPUT ERROR phrase may also be used if the FREE phrase has been specified. It may not, however, be specified if the FREE phrase is not. This is the distiction between formats 1 and 2 of the ACCEPT statement.

Free-field format uses the pound sign, (#), to indicate the end of data. The ampersand (&), if used as the last non-blank character in a record, indicates a continuation of data from one record or line to another. An ampersand takes precedence over the pound sign. Thus, for example, if you enter the characters, ABC##&, a single pound sign is treated as part of the data, and the ampersand is assumed to indicate a continuation of the data to the next line.

If the ACCEPT statement is issued against a terminal (operators console or otherwise), the pound sign is not required to terminate data. The pound sign need only be used to indicate the end of data on a terminal when the last non-blank character of data to be read is an ampersand. Otherwise, simply pressing the RETURN key on the terminal will indicate the end of data.

If you wish to enter a pound sign as part of your data, you must use two consecutive pound signs, in which case, your program takes a single pound sign as a data character.

In free-field format, alphanumeric data is left justified (or right justified if JUSTIFIED [RIGHT] is specified in the PICTURE clause for the receiving data item), with blank fill for any unused character positions. Numeric data is aligned on the decimal point, with zero fill for unused character positions.

If the identifier named in the ACCEPT statement names a numeric or numeric edited data item, the input must be a numeric value, with an optional sign. Any necessary conversion takes place automatically as in elementary moves (see the MOVE statement).

In any case other than numeric or numeric edited data, input is assumed to be alphanumeric. No conversion takes place, but justification and space filling is performed as described above.

If you use the FREE phrase, you can also specify the ON INPUT ERROR phrase. This phrase, if specified, allows you to handle three input error conditions. These conditions are:

A. An illegal digit or illegal sign in a numeric item;

B. A physical I/O error or an end-of-file error;

C. Too long an input string for the receiving field.

When such an error condition occurs, and the ON INPUT ERROR phrase is specified, control is passed to the imperative statement of that phrase.

Format 1 Of The Accept Statement Without The Free Phrase

If a format 1 ACCEPT statement is used without the FREE phrase, and the receiving data item requires less characters than the hardware imposed maximum, then when the input data is transferred, and it is the same length as the receiving data item, no problems arise.

If a hardware device is not capable of transferring data of the same size as the receiving data item, two cases must be considered.

First, if the size of the receiving data item exceeds the size of the transmitted data, the transmitted data is stored in the leftmost characters of the receiving data item. Additional data is then requested. The next group of data elements transmitted (if any) is aligned to the right of the rightmost character already occupying positions in the receiving data item. This process continues until either the receiving data item is full, or the characters :EOJ or :EOD are transmitted as the first four characters of a sending record.

Note that you can use the linefeed key to continue the transmission of characters from your screen after you have reached the right margin. This allows you to enter up to 256 characters per line before you press the RETURN key. In most cases, this avoids the necessity of sending only part of the characters required to fill the receiving data item at a given time.

In the second case, if the size of the transferred data exceeds the size of the receiving data item, or of the portion of the receiving data item not yet occupied, only the leftmost characters of the transferred data are stored in the area available in the receiving data item. The remaining characters are ignored.

Free-field format has its own rules for data transmission termination, as described on the following pages.

Format 3 Of The Accept Statement

The third format of the ACCEPT statement,

ACCEPT *identifier* [**FREE**] **FROM** { **DATE** | **DAY** | **TIME** }

is used to transmit the date, day, or time from the internal clock of the system to the identifier named in the ACCEPT statement.

The words, DATE, DAY and TIME are all reserved in COBOL II and cannot, therefore, be described in a COBOL program.

To illustrate the ACCEPT statement:

```
        .
        .
        .
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SPECIAL-NAMES.
     FROM-TERMINAL IS SYSIN.
        .
        .
        .
DATA DIVISION.
        .
        .
        .
WORKING-STORAGE SECTION.
01   NUMBER-IN            PIC 999V99.
01   DATE-IN
     02  YR               PIC X(2).
     02  MO               PIC X(2).
     02  DY               PIC X(2).
01   DATE-OUT
     02  MONTH-OUT        PIC X(2).
     02  FILLER           PIC X      VALUE '/'.
     02  DAY-OUT          PIC X(2).
     02  FILLER           PIC X      VALUE '/'.
     02  YEAR-OUT         PIC X(2).
         .
         .
         .
PROCEDURE DIVISION.
         .
         .
         .
     ACCEPT DATE-IN FROM DATE.
     MOVE DY TO DAY-OUT.
     MOVE MO TO MONTH-OUT.
     MOVE YR TO YEAR-OUT.
     WRITE DATE-OUT AFTER ADVANCING 1 LINES.
         .
         .
         .
     ACCEPT NUMBER-IN.
     IF NUMBER-IN IS LESS THAN 125.50 THEN PERFORM BILL-LOW.
```

To illustrate the free-field format:

```
                .
                .
                .
     ENVIRONMENT DIVISION.
                .
                .
                .
     DATA DIVISION.
                .
     01   IN-DATA            PICTURE X(19) VALUE SPACES.
                .
                .
    PROCEDURE DIVISION.
                .
                .
                .
        ACCEPT IN-DATA FREE;
             ON INPUT ERROR DISPLAY "DATA TOO LONG".
                .
                .
                .
```

User response:

```
DOUBLE&
TROUBLE &
BUBBLE GUM
```

Result:

```
DOUBLETROUBLE BUBBL

DATA TOO LONG
```

Another response:

```
ADD & GET ## OF SUM#
```

Result:

```
ADD & GET # OF SUM Δ
```

supplied by compiler to fill IN-DATA.

In the first response above, the message, DATA TOO LONG, was returned because the user response exceeded nineteen characters. Note that the data stored did not include the five characters, EΔGUM. If the ON INPUT ERROR had not been specified, there would have been no indication that the data had been truncated.

ADD STATEMENT

The ADD statement computes the sum of two or more operands and stores the result. It has three formats:

Format 1

```
ADD  { identifier-1 }  [ , identifier-2 ]  ... TO  identifier-m
     { literal-1    }  [ , literal-2    ]

     [ ROUNDED ] [, 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 ]
            [, identifier-n [ ROUNDED ] ] ...
     [; ON SIZE ERROR imperative-statement ]
```

Format 3

```
ADD  { CORRESPONDING }   identifier-1 TO identifier-2
     { CORR          }

     [ ROUNDED ] [; ON SIZE ERROR imperative-statement ]
```

In format 1 and 2, *identifier-1*, *identifier-2*, and so forth must refer to elementary numeric items, except that in format 2, each identifier following the word GIVING may also be an edited numeric data item. Also, the word, *literal*, means numeric literal.

In format 3, both identifiers must refer to group items.

When you use format 1, the values of all identifiers and literals to the left of the key word, TO, are added together, and the resulting sum is added to the current contents of *identifier-m*. The results are then stored into *identifier-m*. This process of adding the resulting sum to an identifier and then storing the results into the identifier is continued until all identifiers to the right of the TO keyword have been used.

When you use format 2, all literals and values of identifiers to the left of the GIVING keyword are added, and the result is stored into each identifier named to the right of the GIVING keyword.

When you use format 3, data items in *identifier-1* are added to corresponding data items in *identifier-2*. The results are stored in corresponding data items of *identifier-2*. Thus, format 3 is equivalent to using format 1 for each pair of corresponding data items.

See section X for details on the ROUNDED, SIZE ERROR, and CORRESPONDING phrases.

The composite of operands (see Arithmetic Expressions, section X) must not exceed 18 digits. It is calculated, in format 1, by using all of the operands in the statement; in format 2, it is calculated using all of the operands to the left of the GIVING phrase; in format 3, the composite of operands is calculated using pairs of corresponding data items.

Note that during execution the compiler always insures that enough places are carried to avoid losing any significant digits.

For an example of format 3 usage, see the paragraph titled CORRESPONDING PHRASE, in section X. For an example of format 1 usage, see the paragraphs under the heading ARITHMETIC STATEMENTS, in SECTION X.

To illustrate format 2 of the ADD statement:

Operands and assumed values:

```
01    SUM-IT              PICTURE 9(9)V999.
```
(assumed value is 329‸182)

```
01    SUM-AT              PICTURE 999V9.
```
(assumed value is 203‸9)

Receiving data items:

```
01    TAKE-1              PICTURE 9(9)V999.
01    TAKE-2              PICTURE 9(8)V9.
```

ADD statement:

```
ADD SUM-IT, SUM-AT GIVING TAKE-1,
 TAKE-2 ROUNDED; ON SIZE ERROR PERFORM
  REPORT-IT.
```

Composite of operands: 329‸182

Results:

TAKE-1 has the value 533‸082

TAKE-2 has the value 533‸1 because of the ROUNDED phrase.

ALTER STATEMENT

The ALTER statement allows you to modify a predetermined sequence of operations.

Format

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

Where

procedure-name-1, procedure-name-3, and so forth	are paragraphs containing a single sentence consisting of a GO TO without the DEPENDING phrase.
procedure-name-2, procedure-name-4, and so forth	are paragraphs or sections in the Procedure Division.

Execution of an ALTER statement modifies the GO TO statement in the specified paragraphs so that subsequent executions of the modified GO TO statements cause transfer of control to the sections or paragraphs named by *procedure-name-2*, *procedure-name-4*, and so forth.

For example, the paragraph,

```
GO-PARA.
   GO TO CHECK SECTION.
```

is altered to be equivalent to the paragraph,

```
GO-PARA.
   GO TO FINISH UP.
```

by the ALTER statement,

```
ALTER GO-PARA TO PROCEED TO FINISH-UP.
```

Segmentation Considerations

The ALTER statement must not refer to a GO TO statement which appears in a section whose segment-number is greater than 49 unless the ALTER statement is in the same segment.

CALL STATEMENT

The CALL statement is described in Section XII, INTERPROGRAM COMMUNICATION.

CANCEL STATEMENT

The CANCEL statement is described in Section XII, INTERPROGRAM COMMUNICATION.

CLOSE STATEMENT

The CLOSE statement terminates the processing of sequential, random, relative, and indexed files. It can only be executed for an open file.

It has two formats, depending upon whether you wish to close a sequential, or one of the other three types of files. The formats are:

Format 1 - Sequential Files

```
                     ┌ { REEL }  [ WITH NO REWIND ] ┐
CLOSE file-name-1    │ { UNIT }  [ FOR REMOVAL    ] │
                     │                              │
                     │ WITH { NO REWIND }           │
                     └      { LOCK      }           ┘

    ┌                  ┌ { REEL }  [ WITH NO REWIND ] ┐ ┐
    │ , file-name-2    │ { UNIT }  [ FOR REMOVAL    ] │ │ ...
    │                  │                              │ │
    │                  │ WITH { NO REWIND }           │ │
    └                  └      { LOCK      }           ┘ ┘
```

General Rules

Rules applying to a CLOSE statement for any type of file are described below.

A CLOSE statement can only be issued for a file which has already been opened, and has not yet been closed.

If a CLOSE statement has been successfully executed for a file, no other statement can be executed that references the closed file, either explicitly or implicitly, unless an intervening OPEN statement for that file is executed. There is one exception to this rule. A sequential file which has been closed may be referred to in SORT and MERGE statements which use the USING or GIVING phrases. In this case, the file or files named in the USING and GIVING phrases must not be open.

Following the successful execution of the CLOSE statement (without the REEL or UNIT phrase in the case of sequential files), the record area associated with the name of the closed file is no longer available.

If a CLOSE statement is unsuccessful in its execution, the availability of the record area for the specified file is undefined.

If a CLOSE statement has not been issued for an open file when a STOP RUN statement (or a GOBACK statement in a main program) is executed, the file is automatically closed by the MPE file system.

If a called program has been cancelled through the use of the CANCEL statement, and if a file has been opened but not closed by that program before it was cancelled, then when the same program is called and attempts to open the file, a run time error will occur.

If the file being closed is a new file or a temporary file, it is closed in the temporary file domain. If it is a permanent file, it remains in the permanent file domain when it is closed.

The FILE STATUS data item, if any, specified for the file named in the CLOSE statement is updated to indicate the success or failure of the closing operation. See section VII, the FILE STATUS CLAUSE, for valid combinations of status keys 1 and 2.

Format 1 - Sequential Files

```
                         ┌ ┌ REEL ┐ ┌ WITH NO REWIND ┐ ┐
                         │ └ UNIT ┘ └ FOR REMOVAL    ┘ │
CLOSE file-name-1        │                             │
                         │ WITH ┌ NO REWIND ┐          │
                         └      └ LOCK      ┘          ┘

  ┌                      ┌ ┌ REEL ┐ ┌ WITH NO REWIND ┐ ┐ ┐
  │                      │ └ UNIT ┘ └ FOR REMOVAL    ┘ │ │
  │ , file-name-2        │                             │ │ ...
  │                      │ WITH ┌ NO REWIND ┐          │ │
  └                      └      └ LOCK      ┘          ┘ ┘
```

Using a format 1 CLOSE statement, as shown above, allows you to terminate the processing of files whose organization is sequential. It also provides you with the options of placing the serial access device at its physical beginning, and of locking the file so that it cannot be opened again during the execution of the current run unit.

REEL/UNIT And REMOVAL Phrases

The REEL/UNIT phrase and the REMOVAL phrase are treated as comments in this format of the CLOSE statement. Furthermore, if the REEL/UNIT phrase is specified in a format 1 CLOSE statement, the entire CLOSE statement is treated as a comment. Thus, the file specified in the CLOSE REEL/UNIT statement remains open.

Each of the remaining optional phrases are discussed below.

If no optional phrases are used, that is, if the format 1 CLOSE statement consists entirely of the statement,

CLOSE *file-name-1.*

then the system's closing operations are executed, no matter what kind of operations (input, input-output, output or extend) the file was opened for. If the file resides on a magnetic tape, the reel is rewound when the file is closed.

NO REWIND Phrase

The NO REWIND phrase applies only to labeled magnetic tape files.

Used without either a REEL or UNIT phrase, the NO REWIND phrase alters the execution of the system's standard closing procedure. The tape device, instead of being rewound when the file is closed, remains in its current position.

This phrase should be used in the closing of a file only if another file residing nearer the end of the same tape is to be opened later in the program. Upon completion of the object program the tape is rewound by the operating system.

If the file resides on a device which allows no rewinding, such as a line printer, the NO REWIND phrase is ignored when specified for that file in a CLOSE statement; it has no effect on the file.

WITH LOCK Phrase

The WITH LOCK phrase can be used in the CLOSE statement to assure that the file being closed cannot be opened again during the execution of the current run unit.

This locking is accomplished by the program, following the successful closing of the file.

FORMAT 2 - Random, Relative And Indexed Files

The second format of the CLOSE statement,

CLOSE *file-1* [WITH LOCK] [, *file-2* [WITH LOCK]] ... .

closes the files named by ***file-1***, ***file-2***, and so forth, and optionally locks the files so that they cannot be opened again during execution of the current run unit.

The files named by ***file-1***, ***file-2***, and so forth need not all have the same organization or access.

When a CLOSE statement without the LOCK phrase is issued for a relative, random, or indexed file, the MPE file system closing procedures are used to close the file or files specified, no matter how the files are used (i.e. input, input-output, or output).

Additionally, if the LOCK phrase is used with a relative, random- access, or indexed file, the compiler insures that the file cannot be opened again during execution of the current run unit.

As an example of a format 2 CLOSE statement,

```
              .
              .
              .
ENVIRONMENT DIVISION.
              .
              .
              .
INPUT-OUTPUT SECTION.

   FILE-CONTROL.
   SELECT INDEXER
      ASSIGN TO ``FILE-INDX, DA, A, DISC''
         ORGANIZATION IS INDEXED
         ACCESS MODE IS DYNAMIC
         RECORD KEY IS INDX-FOR-FL.

   SELECT RNDM-FL
      ASSIGN TO ``RANDUM''
         ACCESS MODE IS RANDOM
         PROCESSING MODE IS SEQUENTIAL
         ACTUAL KEY IS DATA-5.
              .
              .
PROCEDURE DIVISION.
              .
              .
              .
CLOSE INDEXER WITH LOCK, RNDM-FL WITH LOCK.
              .
              .
              .
```

In the above CLOSE statement, the files named INDEXER and RNDM-FL are closed and locked so that they may not be opened again during execution of the run unit.

COMPUTE STATEMENT

The COMPUTE statement evaluates an arithmetic expression (see Section X, Arithmetic Expressions), and assigns the result to one or more data items.

COMPUTE ***identifier-1*** **[ROUNDED] [, *identifier-2* [ROUNDED]] ...**

= *arithmetic-expression*
[; ON SIZE ERROR *imperative-statement*]

Where

identifier-1, identifier-2, and so forth	each refer to either an elementary numeric, or an elementary numeric edited data item.
arithmetic-expression	is any valid COBOL arithmetic expression.

The ROUNDED and SIZE ERROR phrases are discussed in Section X, as are multiple results and other information pertaining to arithmetic statements.

The COMPUTE statement allows you to combine arithmetic operations without the restrictions on composites of operands or receiving data items imposed by the arithmetic statements, ADD, SUBTRACT, MULTIPLY, and DIVIDE.

When the COMPUTE statement executes, the arithmetic expression is evaluated, and all of the identifiers to the left of the equal sign are assigned the value of the result. Rounding is done where specified and necessary.

Note that if *arithmetic-expression* is just a single identifier or numeric literal, the COMPUTE statement provides a means of setting each specified identifier equal to a single value.

An example of the COMPUTE statement:

```
COMPUTE DAILY-RTE-1, DAILY-RTE-2 = (INT - RTE / 360) * DAYS
   ON SIZE ERROR PERFORM RATE-ERROR-RTNE.
```

In the above statement, a daily interest rate is calculated, and the results are stored in the two data items, DAILY-RTE-1 and DAILY-RTE-2. If a size error occurs, no data is stored in the two receiving data items, and the error handler, RATE-ERROR-RTNE, is performed.

DELETE STATEMENT

The DELETE statement logically removes a record from a relative or indexed file.

```
DELETE file-name RECORD
    [ ; INVALID KEY imperative-statement ]
```

Where

file-name — is the name of the file from which the record is to be deleted. The file must be an indexed or relative file opened in input-output mode.

imperative-statement — is one or more imperative statements.

If the specified file is being used in sequential access mode, the INVALID KEY clause must not be specified. However, if the file is being used in any other access mode, and there is no applicable USE procedure, the INVALID KEY clause must be specified.

After a successful execution of a DELETE statement, the selected record (see below) has been logically removed from the file, and can no longer be accessed. Note that "logically removed" means that the record has been marked for deletion, and not physically removed. Hence, the contents of the record area associated with the specified file are unaffected. The current record pointer is also unaffected.

The execution of the DELETE statement causes the value of the FILE STATUS data item, if specified for the file, to be updated. See section VII, the FILE STATUS CLAUSE, for valid combinations of status keys 1 and 2.

Selection of the record to be deleted is accomplished in one of two ways, depending upon what access mode is specified for the file.

For files being used in sequential access mode, the record to be deleted is the last record to have been read by a successfully executed READ statement. The READ statement must have been the last input-output operation performed on the file prior to execution of the DELETE statement.

For files in dynamic or random access mode, the record removed is that record identified by the contents of the RELATIVE KEY or RECORD KEY data item associated with the specified file. If the file does not contain the record specified by the key, an INVALID KEY condition exists.

If the INVALID KEY phrase has been specified, and the execution of a DELETE statement causes an INVALID KEY condition, the imperative statement (or statements) specified in the INVALID KEY phrase is executed. Any USE procedure specified for the file is ignored.

Note that for indexed files, if the primary record key has duplicates specified for it, you should use the DELETE statement only when the file is open in sequential access mode. This is because a DELETE statement for such files open in dynamic or random access mode will delete the first record written to the file which has the same primary key value as the value placed in the RECORD KEY data item. This first occurrence of the duplicate value might not be the record you wish to delete.

An example of use of the DELETE statement:

```
       .
       .
       .
ENVIRONMENT DIVISION.
       .
       .
       .
INPUT-OUTPUT SECTION.
FILE-CONTROL.
SELECT REL-FILE
    ASSIGN TO "DATAFL1"
      ORGANIZATION IS RELATIVE
      ACCESS MODE IS SEQUENTIAL
      FILE STATUS IS CHK-REL-FILE.
SELECT INDX-FILE
    ASSIGN TO "DATAFL2"
      ORGANIZATION IS INDEXED
      ACCESS MODE IS RANDOM
      RECORD KEY IS KEY-DATA
      FILE STATUS IS CHK-INDX-FILE.
        .
        .
        .
DATA DIVISION.
FILE SECTION.
FD REL-FILE.
        .
        .
        .
01 REL-DATA.
   02  DATA-1                    PIC 99.
        .
        .
        .
FD INDX-FILE.
        .
        .
        .
01 INDX-DATA.
   02 KEY-DATA                   PIC X(5).
        .
        .
        .
```

```
PROCEDURE DIVISION.
        .
        .
        .
    OPEN I-O REL-FILE, INDX-FILE.
    READ REL-FILE RECORD; AT END STOP RUN.
    IF DATA-1 IS EQUAL TO 0 THEN DELETE REL-FILE RECORD.
        .
        .
        .
    READ INDX-FILE RECORD.
    IF KEY-DATA IS EQUAL TO "MEYER"
       THEN DELETE INDX-FILE RECORD;
        INVALID KEY PERFORM CHECK-OUT.
        .
        .
        .
CHECK-OUT.
    DISPLAY "VALUE OF CHK-INDX-FILE IS", CHK-INDX-FILE.
    DISPLAY "WHAT ACTION TO BE TAKEN?"
    ACCEPT ACTION-ITEM.
        .
        .
        .
```

DISPLAY STATEMENT

The DISPLAY statement can be used to transfer low volume data to the operators console, a terminal, or the line printer. If more than one name is specified, each data item is listed in the order specified by the DISPLAY statement.

Format

```
           { identifier-1 }   , [ identifier-2 ]         [      { SYSOUT        } ]
DISPLAY    {              }   ; [              ] ...     [ UPON { CONSOLE       } ]
           { literal-1    }   , [ literal-2    ]         [      { mnemonic-name } ]
```

Where

identifier-1, literal-1 and so forth	are identifiers of data items, unsigned numeric integer literals, the special registers, TALLY, TIME-OF-DAY, and CURRENT-DATE, and any figurative constant except ALL.
SYSOUT	is, in batch mode, the line printer; in session mode, it is the terminal from which the COBOL program was initiated. This is the default if the UPON phrase is not used.
CONSOLE	is the operator's console.
mnemonic-name	is a name specified by you, and defined under the SPECIAL-NAMES paragraph of the Environment Division as either SYSOUT or CONSOLE.

The DISPLAY statement displays numeric data as it appears in memory. Thus, for example, -987, if entered as a zoned-overpunch number, is DISPLAYed as 98P. Also, if an item is described as USAGE COMPUTATIONAL or COMPUTATIONAL-3, the compiler translates it into a USAGE DISPLAY item for purposes of displaying it.

If TIME-OF-DAY is used as an identifier, the time is displayed in edited form. That is, in the form, HH:MM:SS where HH is the hour taken from a 24 hour clock; MM is the number of minutes after the hour, and SS is the number of seconds after the minute.

If a figurative constant is specified as an operand, only one occurrence of the constant is displayed.

When a DISPLAY statement contains more than one operand, the size of the data to be transmitted is the sum of the sizes of all the operands. The values of the operands are transferred in the sequence in which the operands are listed.

As with the ACCEPT statement, hardware record sizes determine the display of the data specified in the DISPLAY statement. The following methods are used, depending upon whether the size of the sending item is equal to, shorter than, or longer than the hardware device designated to receive the data.

1. If the sending item is the same length, no problem arises, and the data is transmitted.

2. If the sending item is shorter than the device, the transferred data is displayed beginning with the left-most position of the device, and continuing to the right until all data characters have been displayed.

3. If the sending item is longer than can be displayed on one line of the device, the first line of the device is filled with as many characters as possible, the next line is then filled, and so forth until the entire sending item has been displayed. The order in which the sending data is displayed is the same as the order in which it is transmitted.

   An example:

```
       .
       .
       .
WORKING-STORAGE SECTION.
01 BEGIN-MSG  PIC X(21) VALUE "  PROGRAM BEGINNING  ".
       .
       .
       .
PROCEDURE DIVISION.
      DISPLAY CURRENT-DATE, BEGIN-MSG, TIME-OF-DAY
         UPON SYSOUT.
       .
       .
       .
```

When the DISPLAY statement above is executed, assuming the date to be Tuesday, February 26,1980 at exactly 10:45 a.m. the message,

```
02/26/80  PROGRAM BEGINNING  10:45:00
```

is displayed on the screen of the terminal from which the program was run.

DIVIDE STATEMENT

The DIVIDE statement divides one numeric data item into one or more others, assigns the result to a data item, and optionally assigns a remainder to another data item.

There are five formats of the DIVIDE statement:

Format 1

```
DIVIDE { identifier-1 } INTO identifier-2 [ ROUNDED ]
       { literal-1    }

   [, identifier-3   [ ROUNDED ]] ... [ ; ON SIZE ERROR imperative-statement ]
```

Format 2

```
DIVIDE { identifier-1 }  INTO { identifier-2 } GIVING identifier-3  [ ROUNDED ]
       { literal-1    }        { literal-2    }

   [, identifier-4   [ ROUNDED ]] ... [ ; ON SIZE ERROR imperative-statement ]
```

Format 3

```
DIVIDE { identifier-1 }  BY { identifier-2 } GIVING identifier-3  [ ROUNDED ]
       { literal-1    }      { literal-2    }

   [, identifier-4   [ 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*]

Format 5

DIVIDE { *identifier-1* | *literal-1* } **BY** { *identifier-2* | *literal-2* } **GIVING** *identifier-3* [**ROUNDED**]

REMAINDER *identifier-4* [; ON **SIZE** **ERROR** *imperative-statement*]

Where

identifier-1, ***identifier-2,*** and so forth	are names of elementary numeric items, except that those associated with a GIVING or REMAINDER phrase may be elementary numeric edited items.
literal-1, ***literal-2,*** and so forth	are numeric literals.

The ROUNDED and SIZE ERROR phrases are discussed under the heading, COMMON PHRASES, in Section X.

The composite of operands for the DIVIDE statement is determined using all of the receiving data items of a particular statement except the data item associated with the REMAINDER phrase. This composite must not exceed 18 digits. See Section X, the ARITHMETIC EXPRESSIONS paragraph, for details on how to determine the composite of operands.

When you use format 1 of the DIVIDE statement, each identifier following the INTO keyword is divided, in turn, by the identifier or literal to the left of the INTO keyword. Each result is rounded if specified and necessary, and is then stored in the data item referenced by the identifier which acted as the dividend in that particular division.

When you use format 2, the literal or data item specified by the identifier between the key words INTO and GIVING is divided by the literal or data-item specified by *identifier-1*, and the result is stored in each identifier listed in the GIVING phrase.

When you use the third format of the DIVIDE statement, the data item specified by *identifier-1* or *literal-1* is divided by *literal-2* or the contents of *identifer-2*. The result is then stored in each identifier following the GIVING phrase, with rounding being used where specified and needed.

You can use formats 4 and 5 when you wish to obtain a remainder from a division operation. In COBOL, the remainder is defined as the difference between the product of the quotient and the divisor and the dividend.

For example, in the (format 4) DIVIDE statement,

```
DIVIDE A INTO B GIVING C REMAINDER D
```

The remainder, D, has the value determined by multiplying C times A and subtracting this product from B. Thus, if A=7 and B=16, then C=2, and D=2 since 16-7*2=2.

If *identifier-3* (the quotient) is defined as numeric edited, the quotient used to calculate the remainder is an internal, intermediate field containing the unedited quotient.

Also, if the ROUNDED phrase is specified, the quotient used to calculate the remainder is kept in an intermediate field, and is truncated rather than rounded.

Appropriate decimal alignment and truncation are performed on the remainder as needed.

When the SIZE ERROR phrase is specified for a format 4 or 5 DIVIDE statement, and a size error condition occurs for the quotient, the contents of data items referenced by *identifier-3* and *identifier-4* are unchanged. However, if the size error condition occurs for the remainder, and not the quotient, only the remainder is unchanged. *Identifier-3* will still contain the new quotient.

To illustrate the DIVIDE statement:

```
           .
           .
           .
WORKING-STORAGE SECTION.
RATE            PIC 99      VALUE ZERO.
CHECK           PIC V99     VALUE ZERO.
           .
           .
           .
FILE SECTION.
FD PAY-FILE.
01 PAY-INFO OCCURS 1 TO 150 TIMES
      DEPENDING ON EMP-SIZE.
   02 EMP-NAME           PIC X(30).
   02 EMP-NUM            PIC X(9).
   02 PAY                PIC 999V99.
   02 HOURS              PIC 99.
           .
           .
           .
PROCEDURE DIVISION.
           .
           .
           .
     DIVIDE PAY BY HOURS GIVING RATE REMAINDER CHECK;
        ON SIZE ERROR PERFORM SIZE-ERR.
           .
           .
```

```
SIZE-ERR.
     IF RATE = 0 THEN
        DISPLAY "SIZE ERROR IN RATE USING " PAY, HOURS;
     ELSE
        DISPLAY "SIZE ERROR IN CHECK".
        .
        .
        .
```

The DIVIDE statement above uses format 5. If a size error occurs, the SIZE-ERR routine is performed, and a check is made to determine whether the size error occurred because of RATE or CHECK.

ENTER STATEMENT

In ANSI COBOL'74, this statement provides a means of allowing the use of more than one language in the same program. It is, however, not allowed in HP COBOL II/3000. Thus, if specified in your program, it is treated as a comment. The format is listed below for information only.

Format

ENTER *language-name* [*routine-name*].

ENTRY STATEMENT

The ENTRY statement is discussed in Section XII, INTERPROGRAM COMMUNICATION.

EXAMINE STATEMENT

The EXAMINE statement is not a part of COBOL II. It was deleted and replaced by the INSPECT statement, covered later in this section.

Although the EXAMINE statement can be used in HP COBOL II/3000, it is advisable that you use the INSPECT statement for coding new source programs.

The EXAMINE statement replaces or counts the number of occurrences of a given character in a data item.

Format

```
EXAMINE identifier
  { TALLYING { UNTIL FIRST   } literal-1 [ REPLACING BY literal-2 ] }
  {          { ALL           }                                      }
  {          { LEADING       }                                      }
  {                                                                 }
  { REPLACING { ALL                } literal-3 BY literal-4         }
  {           { LEADING            }                                }
  {           { [ UNTIL ] FIRST    }                                }
```

Where

identifier	names a data item whose usage is DISPLAY. It is this data item which is examined.
literal-1, ***literal-2,*** and so forth	are each a single character whose data type is the same as identifier. Any or all of these literals may be any figurative constant except ALL.

When the EXAMINE statement is executed, it acts differently depending upon whether *identifier* names a numeric or a nonnumeric data item.

If *identifier* is a nonnumeric data item, examination begins with the leftmost character, and proceeds to the right. Each character is examined in turn.

If *identifier* is a numeric data item, the data item may contain a sign, and examination proceeds on a digit by digit basis, starting with the left-most digit and proceeding to the right. Note that a sign, if included in the data item being examined, is ignored, regardless of its physical location.

TALLYING Phrase

When the TALLYING phrase is used in an EXAMINE statement, a count is placed in the special COBOL II/3000 register named TALLY. This count is an integer, and represents a value which is dependent upon the key words following the word TALLYING.

If TALLYING UNTIL FIRST is specified, the integer in the TALLY register after execution of an EXAMINE statement is the number of occurrences of characters in *identifier* before the first occurrence of *literal-1*.

If TALLYING ALL is specified, every occurrence of *literal-1* is counted, and the result of this counting is placed in the TALLY register.

If TALLYING LEADING is specified, only those occurrences of *literal-1* which precede any other characters in the data item named by *identifier* are counted. Thus, for example, if the first character of *identifier* is not *literal-1*, the EXAMINE statement ceases execution immediately.

If the REPLACING phrase is used in conjunction with the TALLYING phrase, then, depending upon which key words are used with the TALLYING phrase, those occurrences of *literal-1* which participate in the tallying are replaced by *literal-2*. Thus, for example, if the EXAMINE statement,

```
EXAMINE ABMASK TALLYING ALL A REPLACING BY B.
```

is executed, and ABMASK contains the value, ABBBBABBABAAAB, before execution, then when execution of the EXAMINE statement is complete, the value in the TALLY register is 6, and BITMASK now contains the value, BBBBBBBBBBBBBB.

REPLACING Phrase

The REPLACING phrase acts in the same manner as the REPLACING verb in the TALLYING phrase. However, since no tallying takes place, the TALLY register remains unchanged. Specifically,

- If REPLACING ALL is specified, all occurrences of *literal-3* in *identifier* are replaced by *literal-4*.
- If REPLACING LEADING is specified, each occurrence of *literal-3* is replaced by *literal-4* until the first occurence of a character other than *literal-3* or the right-most character of the data item is examined.
- If REPLACING UNTIL FIRST is specified, every character of the data item represented by *identifier* is replaced by *literal-4* until *literal-3* is encountered in the data item. If *literal-3* does not appear in the data item, the entire data item is filled with *literal-4*.
- If REPLACING FIRST is specified, only the first occurrence of *literal-3* is replaced by *literal-4*. If *literal-3* does not appear in the data item represented by *identifier*, the data item is unchanged after execution of the EXAMINE statement.

EXCLUSIVE STATEMENT

The EXCLUSIVE statement provides you with a method for locking a file which has been opened for shared access.

Note that this "locking" does not stop anyone from accessing the file. Locking and unlocking files must be done on a cooperative basis. That is, if all users who intend to access a shared file agree to attempt to lock the file before accessing its records, then no problems arise. However, since this form of "locking" only sets a flag on the file, if other users do not check to see if the flag is set (by attempting to lock it themselves), then they may do anything with the file that other file security mechanisms allow.

A file which has been locked remains locked until an UN-EXCLUSIVE statement is issued which names the file.

Format

EXCLUSIVE ***file-name*** [**CONDITIONALLY**]

Where *file-name* is the name of the file you wish to lock. It must be open before the EXCLUSIVE statement is executed. Also, the file must have a USE procedure associated with it, in case an error occurs during execution of the EXCLUSIVE statement. If an error does occur, the USE procedure is executed.

The EXCLUSIVE statement indirectly calls the FLOCK intrinsic. If used without the CONDITIONALLY option, the EXCLUSIVE statement causes the FLOCK intrinsic to continue to try to lock the file until it succeeds. If the file is already locked (for example, by another user), this means your program will pause until the FLOCK intrinsic succeeds.

To avoid this possibility, you can use the CONDITIONALLY option. This tells the FLOCK intrinsic to attempt to lock the file, and if unsuccessful, to return control immediately to your COBOL program.

The FILE STATUS data item, if any, associated with the file named in the EXCLUSIVE statement is updated to indicate whether or not the attempt to lock the file was successful. Thus, if you use the CONDITIONALLY option, you can use the FILE STATUS data item to discover whether the file was locked.

To illustrate the EXCLUSIVE statement:

```
                         .
                         .
                         .
    ENVIRONMENT DIVISION.
                         .
                         .
                         .
    FILE-CONTROL.
         SELECT CUSTFILE ASSIGN TO "CUSTDATA"; FILE STATUS IS CHECKER.
                         .
                         .
    PROCEDURE DIVISION.
                         .
                         .
                         .
        OPEN I-O CUSTFILE.
        EXCLUSIVE CUSTFILE CONDITIONALLY.
        IF CHECKER IS EQUAL TO "00" PERFORM CUSTOMER-UPDATE;
             ELSE PERFORM FIND-WHY.
                         .
                         .
                         .
```

EXIT STATEMENT

The EXIT statement provides a common end point for a series of procedures.

Format

paragraph-name.
    EXIT.
paragraph/section-name.

Paragraph-name and *paragraph/section-name* are not a part of the EXIT statement. They are shown to clarify the fact that

A. EXIT must appear in a sentence by itself, and

B. EXIT must be the only sentence in a paragraph.

An EXIT statement serves only to enable you to terminate a procedure and has no other effect on the compilation or execution of the program.

To illustrate the use of the EXIT statement:

```
          .
          .
          .
PROCEDURE DIVISION.
          .
          .
          .
     PERFORM FIX-IT THRU OUT.
          .
          .
          .
     PERFORM EXCESS THRU OUT.
          .
          .
          .
     FIX-IT.
          IF CHARS IS ALPHABETIC THEN GO TO OUT.
          .
          .
          .
     EXCESS.
          IF OVER-AMT IS EQUAL TO 0 THEN GO TO OUT.
          .
          .
          .
     OUT. EXIT.
          .
          .
          .
```

In the above illustration, both of the IF statements are the first lines of procedures executed by PERFORM statements. If the condition in either of the IF statements returns a "true" value, the statement branches to the OUT paragraph, the EXIT statement is executed, and control passes to the statement following the PERFORM statement which called the procedure.

EXIT PROGRAM STATEMENT

The EXIT PROGRAM statement is discussed in Section XII, INTERPROGRAM COMMUNICATION.

GO TO STATEMENT

The GO TO statement transfers control from one part of the Procedure Division to another. It has two formats:

Format 1

GO TO [*procedure-name-1*]

Format 2

GO TO *procedure-name-1* [, *procedure-name-2*] ...
, *procedure-name-n* DEPENDING ON *identifier*

Where

procedure-name-1 through ***procedure-name-1***	each are names of procedures within the Procedure Division of your program.
identifier	is the name of a numeric elementary data item which has no positions to the right of the decimal point.

Format 1 of the GO TO statement transfers control to the procedure named by *procedure-name-1*, or, if no procedure is named, to the procedure specified in a previously executed ALTER statement. Note that an ALTER statement must be issued for this type of GO TO statement before it is executed if no procedure is named in it. This also implies that a GO TO statement without a procedure name specification must make up the only sentence in a paragraph. See the ALTER statement for other restrictions.

If format 1 of the GO TO statement does specify a procedure-name, and it appears in a sequence of imperative statements within a sentence, it must be the last statement in that sentence.

In format 2 of the GO TO statement, *identifier* must be the name of a numeric elementary item described with no positions to the right of the decimal point. It is used to determine which procedure is to be executed. If the contents of *identifier* is an integer in the range 1 to n, where n is the number of procedure names appearing in the GO TO statement, then control passes to the procedure in the position corresponding to the value of *identifier*. Otherwise, no transfer occurs, and control passes to the next statement following the GO TO statement.

Examples of the GO TO statement:

```
        .
        .
        .
WORKING-STORAGE SECTION.
01 SELECTOR             PIC 9(3).
        .
        .
        .
```

```
PROCEDURE DIVISION.
          .
          .
          .
     ALTER GO-PARA TO PROCEED TO WHICH.
          .
          .
          .
GO-PARA.  GO TO.
AFTER-GO PARA.
          .
          .
          .
     GO TO UNDER, OVER, EXACT DEPENDING ON SELECTOR.
     DISPLAY "SELECTOR OUT OF RANGE - VALUE IS  ", SELECTOR.
          .
          .
          .
UNDER.
          .
          .
          .
OVER.
          .
          .
          .
EXACT.
          .
          .
          .
WHICH.
          .
          .
          .
```

In the above illustration, the GO TO statement in the GO-PARA paragraph is equivalent to GO TO WHICH because of the ALTER statement preceding it.

The second GO TO statement branches to UNDER, OVER, or EXACT, depending upon whether SELECTOR has a value of 1, 2, or 3 respectively. If SELECTOR has any other value, the DISPLAY statement is executed.

GOBACK STATEMENT

The GOBACK statement is discussed in Section XII, INTERPROGRAM COMMUNICATION.

IF STATEMENT

The IF statement evaluates a condition, and, depending upon the truth value of the condition, determines the subsequent action of the object program.

Format

```
                          { statement-1   }  ;  { ELSE NEXT SENTENCE }
IF condition ; THEN       { NEXT SENTENCE }  ;  { ELSE statement-2   }
```

Where

statement-1 and ***statement-2***	each are imperative or conditional statements, optionally followed by a conditional statement.
condition	is any valid COBOL condition as described under CONDITIONS, Section X.

You may omit the ELSE NEXT SENTENCE phrase if it immediately precedes the period used to terminate the sentence.

When an IF statement is executed, the following transfers of control occur:

a. If the truth value of the condition is "true", and *statement-1* is specified, then if *statement-1* is a procedure branching or conditional statement, control is explicitly transferred according to the rules for that statement. If *statement-1* does not contain such a statement, then *statement-1* is executed, and control passes to the next executable sentence.

   If the truth value of the condition is "true", and the NEXT SENTENCE phrase is used instead of *statement-1* control immediately passes to the next executable sentence.

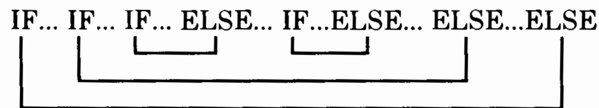
b. If the truth value of the condition is "false", and if *statement-2* is specified, then if *statement-2* is a procedure branching or conditional statement, control is explicitly transferred according to the rules for that statement. If *statement-2* is not such a statement, then *statement-2* is executed and control passes to the next executable sentence.

If the truth value of the condition is "false", and *statement-2* is not specified, then the ELSE NEXT SENTENCE phrase, if specified, causes transfer of control to the next executable sentence. If neither *statement-2* nor the ELSE NEXT SENTENCE phrase is present, the COBOL compiler assumes that the ELSE NEXT SENTENCE phrase is specified.

Note that *statement-1* or *statement-2* may be or may contain an IF statement, according to their description in the previous paragraphs. This is called nested IF statements.

Nested IF statements are considered as paired IF and ELSE combinations, proceeding from left to right. Thus, any ELSE encountered is considered to apply to the immediately preceding IF that has not already been paired with an ELSE.

To clarify, the IF/ELSE pairing is shown in the following illustration:



An example:

```
BEGIN SECTION.
DATA-IN.
  READ REC-FILE RECORD INTO DATA-REC.
      .
      .
      .
  IF DATA-REC IS NOT ALPHABETIC;
     THEN
          IF DATA-REC IS NOT NUMERIC;
             PERFORM ILLEGAL-CHARACTER;
          ELSE NEXT SENTENCE;
  ELSE PERFORM ALPHA-TYPE.
      .
      .
      .
```

The IF statements above are used to check that the data read into DATA-REC is either all alphabetic or all numeric.

The first IF statement consists of the IF/ELSE pair,

```
IF DATA-REC IS NOT ALPHABETIC...; ELSE PERFORM ALPHA-TYPE.
```

The second IF/ELSE pair is

```
IF DATA-REC IS NOT NUMERIC;...; ELSE NEXT SENTENCE.
```

Thus, if DATA-REC has any nonnumeric character or characters not from the English alphabet, the procedure, ILLEGAL-CHARACTER, is performed.

INSPECT STATEMENT

The INSPECT statement can be used to perform one of three actions:

1. It can count the number of occurrences of a given character or character substring within a data item.
2. It can replace a given character or characters within a specified data item with another character or set of characters.
3. It can perform both of the functions described in 1 or 2 in a single operation.

Through the use of the phrases, LEADING, BEFORE, and AFTER, the INSPECT statement can be used to replace only certain occurrences of characters within a data item.

Also, through the use of CHARACTERS, you can tally and replace every character (or subset of characters when used in conjunction with LEADING, BEFORE, and AFTER) in a data item.

The three formats of the INSPECT statement (each of which corresponds to the above delineations of capabilities) are shown on the following page.

Format 1

```
INSPECT identifier-1 TALLYING

{ , identifier-2 FOR { , { { ALL     } { identifier-3 } }
                          { { LEADING } { literal-1    } }
                          { CHARACTERS                   }

    [ { BEFORE } INITIAL { identifier-4 } ] } ... } ...
    [ { AFTER  }         { literal-2    } ]
```

Format 2

```
INSPECT identifier-1 REPLACING

{ CHARACTERS BY { identifier-6 } [ { BEFORE } INITIAL { identifier-7 } ]          }
{               { literal-4    } [ { AFTER  }         { literal-5    } ]          }
{                                                                                 }
{ { , { ALL     } { , { identifier-5 } BY { identifier-6 }                        }
{ {   { LEADING } {   { literal-3    }    { literal-4    }                        }
{ {   { FIRST   } {                                                               }
{                                                                                 }
{       [ { BEFORE } INITIAL { identifier-7 } ] } ... } ...                       }
{       [ { AFTER  }         { literal-5    } ]                                   }
```

Format 3

INSPECT *identifier-1* **TALLYING**

{ , *identifier-2* **FOR** { , { { **ALL** / **LEADING** } { *identifier-3* / *literal-1* } / **CHARACTERS** }

[{ **BEFORE** / **AFTER** } INITIAL { *identifier-4* / *literal-2* }] } ... } ...

REPLACING

{ **CHARACTERS BY** { *identifier-6* / *literal-4* } [{ **BEFORE** / **AFTER** } INITIAL { *identifier-7* / *literal-5* }]

{ , { **ALL** / **LEADING** / **FIRST** } { , { *identifier-5* / *literal-3* } **BY** { *identifier-6* / *literal-4* }

[{ **BEFORE** / **AFTER** } INITIAL { *identifier-7* / *literal-5* }] } ... } ... }

Where

identifier-1 is a variable representing either a group item or any category of elementary item described implicitly or explicitly as USAGE IS DISPLAY. This is the data item to be inspected.

identifier-2 names an elementary numeric data item. It is used to contain the results of tallying occurrences of a character or characters in the data item represented by identifier-1. Identifier-2 is not initialized by the INSPECT statement; thus, if you want it initializd, you must do so programmatically before the INSPECT statement is executed.

identifier-3 through ***identifier-n*** each must reference either an elementary alphabetic, alphanumeric, or numeric data item described implictly or explicitly as USAGE IS DISPLAY.

literal-1 through ***literal-n*** are each nonnumeric literals. Each may be any figurative constant except ALL.

If, in formats 1 and 3, either *literal-1* or *literal-3* is a figurative constant, the constant implicitly refers to a single character constant.

In formats 2 and 3, the size of the data referenced by *literal-4* or *identifier-6* must be equal to the size of the data item referenced by *literal-3* or *identifier-5.*

If *literal-4* is a figurative constant, its size is implicitly equal to the size of *literal-3,* or the size of the data item referenced by identifier-5.

If *literal-3* is a figurative constant, the data referenced by *literal-4* or *identifier-6* must be a single character.

When the CHARACTERS phrase is used, *literal-4* (or *literal-5*), or the size of the data item referenced by *identifier-6* (or *identifier-7*) must be one character in length.

When inspection takes place, the data items referenced by *identifier-1,* and identifiers 3 through 7 are all considered to be character strings, regardless of whether they are alphanumeric, alphanumeric edited, numeric edited, unsigned or signed numeric.

For any data item except signed numeric or alphanumeric, this is accomplished by inspecting the items as though they have been redefined as alphanumeric, and the INSPECT statement written to reference the redefined data item.

For signed numeric data items, this is accomplished by treating the data item as if it had been moved to an unsigned numeric data item, and then inspecting it as described in the preceding paragraph.

Comparison Operation

In order to facilitate the following discussion on the comparison operation, the various groups of identifiers and literals are renamed. The names used, and the identifiers or literals they represent are:

Searchchars used to represent the literals 1 and 3, or the contents of identifiers 3 and 5.

Initchars used to represent the literals 2 and 5, or the contents of identifiers 4 and 7.

Replacechars used to represent literal-4, or the contents of identifier-6.

Data used to represent the contents of identifier-1.

When inspection takes place, the elements of data are compared to searchchars, and for each properly matched occurrence of searchchars,

1. For formats 1 and 3, tallying occurs, using *identifier-2* to contain the results;
2. For formats 2 and 3, each set of properly matched characters in *data* is replaced by *replacechars*.

When the INSPECT statement is used in its simplest form, that is, without the LEADING, BEFORE, and AFTER phrases, the inspection algorithm is as follows:

The first set of *searchchars* is compared with an equal number of characters in *data*, starting with the left-most character of *data*.

If no match occurs, the second set of *searchchars* is compared with an equal number of characters in *data*, again starting with the left-most character of *data*.

This process continues for each set of *searchchars* until either a match occurs, or all sets of *searchchars* are exhausted.

If all sets of *searchchars* are exhausted, and no matches have occurred, comparison begins again, using the first set of *searchchars*, but starting this time with the character immediately to the right of the left-most character of *data*.

Again, comparison proceeds as described above until either a match occurs, or all sets of *searchchars* are used.

If all sets are used, and no matches have occurred, comparison begins again, starting with the character of *data two to the right of the left-most character of data.*

This continuing cycle of shifting one character to the right in the characters of *data* and using all of the sets of *searchchars* is terminated when, if no matches have occurred, the right-most character of *data* has been used in a comparison with the last set of *searchchars*.

If a match does occur for some set of *searchchars*, and TALLYING is specified, *identifier-2* is incremented by 1. If REPLACING is specified, the matched characters of *data* are replaced by the *replacechars* that correspond to the set of *searchchars* being compared when the match occurred.

Inspection continues in the manner described above until the right-most character of *data* has been used as the first character in a comparison with the last set of *searchchars*, or has been matched.

Note that if tallying and replacing are both specified in the INSPECT statement, two completely separate comparisons, as described above, take place. The first comparison is used for tallying, and the second is used for replacing.

To illustrate this form of inspection, we use the INSPECT statement,

```
INSPECT WORD TALLYING COUNTER FOR ALL "X"
           REPLACING ALL "EE" BY "0X", "9" BY "E".
```

Where WORD contains the alphanumeric data, YEEXE9XY, and COUNTER is the variable used to hold the tally. Since INSPECT does not initialize COUNTER, we assume it is initialized to 0 before the INSPECT statement is encountered.

The illustration below shows the step-by-step comparisons that take place using the above algorithm.

```
INSPECT WORD TALLYING COUNTER FOR ALL "X"
         REPLACING ALL "EE" BY "0X", "9"  BY "E".
```

(Initially, WORD="YEEXE9XY" and COUNTER=0)

```
YEEXE9XY
=
X
```

No match occurred; thus, begin comparison again, starting at the second character of YEEXE9XY:

```
YEEXE9XY
 =
 X
```

No match occurred; thus, begin comparison again, starting at the third character of YEEXE9XY:

```
YEEXE9XY
  =
  X
```

No match occurred; thus, begin comparison again, starting at the fourth character of YEEXE9XY:

```
YEEXE9XY
   =
   X
```

A match occurred; thus, increment COUNTER by 1, and begin comparison again, starting with the fifth character of YEEXE9XY:

```
YEEXE9XY
    =
    X
```

No match occurred; thus, begin comparison again, starting with the sixth character of YEEXE9XY:

```
YEEXE9XY
     *
     X
```

No match occurred; thus, begin comparison again, starting with the seventh character of YEEXE9XY:

```
YEEXE9XY
      *
      X
```

A match occurred; thus, increment COUNTER by 1, and begin comparison again, starting with the eighth character of YEEXE9XY:

```
YEEXE9XY
       *
       X
```

No match occurred, and the last character of YEEXE9XY has been used as the first element in a comparison; thus, the tallying cycle is complete. Begin the replacing cycle:

```
YEEXE9XY → YEEXE9XY
**         *
EE         9
```

No match occurred; thus, begin comparison again, starting with the second character of YEEXE9XY:

```
YEEXE9XY
 **
 EE
```

A match occurred; thus change "EE" to "OX", and begin comparison again, starting with the fourth character of YOXXE9XY:

```
YOXXE9XY → YOXXE9XY
   **         *
   EE         9
```

No match occurred; thus, begin comparison again, starting with the fifth character of YOXXE9XY:

```
YOXXE9XY → YOXXE9XY
    **         *
    EE         9
```

No match occurred; thus, begin comparison again, starting with the sixth character of YOXXE9XY:

```
YOXXE9XY → YOXXE9XY
     **         *
     EE         9
```

A match occurred; thus, replace "9" by "E" in YOXXE9XY, and begin comparisons again, at the seventh character of YOXXEEXY:

```
YOXXEEXY → YOXXEEXY
     ==          =
     EE          9
```

No match occurred; thus, begin comparison again, starting with the eighth character of YOXXEEXY:

```
YOXXEEXY → YOXXEEXY
      ==          =
      EE          9
```

No match occurred, and the last character of YOXXEEXY has been used as the first character in a comparison. Thus, the comparison cycle for REPLACING is complete. This ends execution of the INSPECT statement. The result of this INSPECT statement is summarized by the fact that WORD now contains the character string, YOXXEEXY, and COUNTER now contains the integer, 2.

Before and After Phrases

The comparison operation described on the preceding pages is affected by the BEFORE and AFTER phrases in the following way:

If the BEFORE phrase is used, the associated *searchchars* are used only in those comparison cycles that make comparisons of characters of *data* to the left of the first occurrence of the associated *initchars*.

If *initchars* does not appear in *data*, the BEFORE phrase has no effect upon the comparison operation.

If the AFTER phrase is used, the associated searchchars are used only in those comparison cycles that make comparisons of characters of *data* to the right of the first occurrence of the associated *initchars*.

If *initchars* does not appear in *data*, the associated *searchchars* are never used in the comparison cycle; thus, in such a case, it is equivalent to not using the clause in which the AFTER phrase appears.

Leading Phrase

If the LEADING phrase is used in an INSPECT statement, it causes identifier-2 (the variable used to hold the tally) to be incremented by 1 for each contiguous matching of searchchars with a character of data, provided that the matching begins with the left-most character of the characters that make up data.

For replacing, the LEADING phrase has the effect of replacing each contiguous occurrence of matched characters to be replaced by *replacechars*, provided that the matching begins with the left-most character of *data*.

If the first character (or characters) of *data* is not the same as *searchchars*, the clause in which the LEADING phrase appears has no effect upon the data, or the variable used to hold the tally.

All Phrase

When used in tallying, the ALL phrase causes the contents of *identifier-2* to be incremented by 1 for each occurrence of *searchchars* within *data*. When used in replacing, the ALL phrase causes each set of characters in *data* matched with the *searchchars* to be replaced.

Characters Phrase

When the CHARACTERS phrase is used in tallying, the contents of *identifier-2* are incremented by 1 for each character in the set of characters used in the comparison cycle. Note that this does not necessarily imply that all characters will be tallyed, since the BEFORE and AFTER phrases can limit comparison to only part of *data*.

When the CHARACTERS phrase is used in replacement, each character in the set of characters used in the comparison cycle is replaced by replacechars, regardless of what the character in data is. This phrase can be used, for example, to initialize a data item, by not using the BEFORE or AFTER phrases to limit the part of data to be acted upon.

First Phrase

When the FIRST phrase is used in replacement, the left-most occurrence in *data* of *searchchars* is replaced by the associated *replacechars*.

Examples of the INSPECT statement:

Assuming that the variable, THISONE, has the data, "WARNING", in it, the INSPECT statement,

```
INSPECT THISONE REPLACING ALL "N" BY "P" BEFORE INITIAL "I".
```

results in THISONE having the data "WARPING" in it.

```
INSPECT REC TALLYING ACUM FOR ALL "F", REPLACING ALL "G" BY "R".
```

If REC is "JIMGIRAFFEEGAVEGUMDROPS" and ACUM is 0 initially, the above INSPECT statement results in ACUM being 2, and REC containing "JIMRIRAFFEERAVERUMDROPS".

```
INSPECT PETE REPLACING LEADING "BV" BY "AT",
      CHARACTERS AFTER INITIAL "F" BY "0".
```

If PETE contains "CBVFEET" before execution of the above INSPECT statement, then following execution, PETE contains "CATF000".

MOVE STATEMENT

The MOVE statement transfers data to one or more data areas in accordance with the rules of editing. It has two general formats:

Format 1

```
       { identifier-1 }
MOVE   { spec-reg     }   TO identifier-2 [ , identifier-3 ] ...
       { literal      }
```

Format 2

```
MOVE   { CORRESPONDING }   identifier-1 TO identifier-2
       { CORR          }
```

Where

identifier-l ***spec-reg,*** and ***literal***	are the sending areas. The special registers, TALLY, TIME-OF-DAY, CURRENT-DATE, and WHEN-COMPILED may be used as sending items.
identifier-2, ***identifier-3,*** and so forth	are the receiving areas.
CORR	is an abbreviation for CORRESPONDING.

An index data item cannot be used as an operand of a MOVE statement.

If you use format 2, both identifiers must be group items. Selected items are moved from within *identifier-1* to selected items within *identifier-2.* The results are the same as if you referred to each pair of corresponding identifiers in separate MOVE statements. The rules governing correspondence are discussed in Section X, under the heading, CORRESPONDING PHRASE.

If you use format 1, the data designated by *literal,* or by *identifier-1* is moved, in turn, to identifier-2, then *identifier-3* and so forth. Any subscripting or indexing associated with identifiers to the right of the keyword, TO, is evaluated immediately before the data is moved to the respective data items.

Any subscripting or indexing associated with *identifier-1* is evaluated only once, immediately before the data is moved to the first of the receiving operands.

For example, the result of the statement

```
MOVE A(B)        TO B, C(B)
```

is equivalent to

MOVE A(B) TO *temp*
MOVE *temp* TO B
MOVE *temp* TO C(B)

Where temp is an intermediate result item used internally by the system. Note that the move of A(B) to B affects the element of C to which A(B) is moved. That is, if B is initially 1, and A(B) is 9, then after 9 is moved to B, A(1) is moved to C(9), and not C(1).

Rules For Moving Data

All data is moved according to the rules for moving elementary data items to elementary data items. This is called an elementary move. Valid and invalid moves are determined by the categories of the sending and receiving data items. See the PICTURE clause in Section IX for a description of the various categories.

Any move that is not an elementary move is treated exactly as if it were a move from one alphanumeric elementary data item to another, except that there is no conversion from one form of internal representation to another. In such a move, the receiving data item is filled without respect to the individual elementary or group items contained in either the sending or receiving area, except when the sending or receiving data item contains a table whose OCCURS clause uses the DEPENDING ON clause. In this case, only the area specified by the DEPENDING ON clause is filled or moved. If the move is from a group to an elementary item, justification takes place if specified in the receiving item.

RULES FOR ELEMENTARY MOVES

The following rules apply to an elementary move between data items belonging to one of the five categories of data:

1. All numeric literals and the figurative constant, ZERO, belong to the numeric category; all nonnumeric literals, and all figurative constants except SPACE and ZERO belong to the alphanumeric category; SPACE belongs to the alphabetic category.

2. A numeric edited, alphanumeric edited, or alphabetic data item cannot be moved to a numeric or numeric edited data item.

3. A numeric or numeric edited data item cannot be moved to an alphabetic data item.

4. A non-integer numeric literal or non-integer data item cannot be moved to an alphanumeric or alphanumeric edited data item.

5. All other elementary moves are valid, and are performed according to the rules listed below.

Any necessary conversion from one internal representation to another takes place during valid elementary moves, as does any editing specified for the receiving data item.

ALPHANUMERIC OR ALPHANUMERIC EDITED RECEIVING ITEM

When an alphanumeric edited or alphanumeric item is a receiving data item, alignment and any necessary space filling takes place as defined under DATA ALIGNMENT in Section IV. If the size of the sending item is larger than the receiving item, the excess characters are truncated on the right after the receiving data item is filled.

If the sending item is a signed numeric item, the sign is not moved, regardless of whether the sign is separate or not. If the sign is separate, however, the sending item is considered to be one character shorter than its actual size.

NUMERIC OR NUMERIC EDITED RECEIVING ITEM

When a numeric or numeric edited item is the receiving item, alignment by decimal point and any necessary zero-filling is performed as defined under DATA ALIGNMENT in Section IV. The exception to this rule is when zeroes are replaced because of editing requirements of the receiving data item.

For signed numeric receiving items, the sign of the sending item is placed in the receiving item. Note that an unsigned numeric sending item causes a positive sign to be generated for the receiving item. Also, any conversion of the representation of the sign, such as from a zoned overpunched sign to a separate sign, is done as necessary.

For an unsigned numeric receiving item, the absolute value of the sending item is moved, and no operational sign is generated for the receiving item.

For an alphanumeric sending item, data is moved as if the sending item were described as an unsigned numeric integer.

ALPHABETIC RECEIVING ITEM

When a receiving field is described as alphabetic, justification and any necessary space filling is performed as specified under DATA ALIGNMENT in Section IV. If the size of the sending data item is larger than the receiving data item, the excess characters are truncated to the right after the receiving item is filled.

Table 11-1 summarizes the rules presented above.

Table 11-1.
PERMISSIBLE MOVES

Receiving Field / Source Field	Group	Alphabetic	Alphanumeric	External Decimal	Binary	Numeric Edited	Alphanumeric Edited	Internal Decimal
Group	Y	Y	Y	Y1	Y1	Y1	Y1	Y1
Alphabetic	Y	Y	Y	Δ	Δ	Δ	Y	Δ
Alphanumeric	Y	Y	Y	Y4	Y4	Y4	Y	Y4
External Decimal (DISPLAY)	Y1	Δ	Y2	Y	Y	Y	Y2	Y
Binary (COMP)	Y1	Δ	Y2	Y	Y	Y	Y2	Y
Numeric Edited	Y	Δ	Y	Δ	Δ	Δ	Y	Δ
Alphanumeric Edited	Y	Y	Y	Δ	Δ	Δ	Y	Δ
Zero; (numeric or alphanumeric)	Y	Δ	Y	Y3	Y3	Y3	Y	Y3
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	Y5	Y5	Y5	Y	Y5
Numeric Literal	Y1	Δ	Y2	Y	Y	Y	Y2	Y
Nonnumeric Literal	Y	Y	Y	Y5	Y5	Y5	Y	Y5
Internal Decimal (COMP-3)	Y1	Δ	Y2	Y	Y	Y	Y2	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.

Examples of the MOVE statement:

```
FILE SECTION.
FD FILE-IN.
01 FILE-REC.
   02 EMP-FIELD.
      03 NAME        PIC X(20).
      03 AGE         PIC 99.
      03 EMP-NO      PIC X(9).
   02 LOCALE         PIC X(35).
      .
      .
      .
WORKING-STORAGE SECTION.
01 FIELD.
   02 SUB-F1      PIC BBXX       VALUE SPACES.
   02 SUB-F2      PIC XX/XX/XX   VALUE SPACES.
01 NUM-IN         PIC S9(3)V99    VALUE -12099.
01 CARD-NUM       PIC S9(3)V99    SIGN IS TRAILING VALUE ZERO.

01 INFO-OUT.
   02 EMP-FIELD.
      03 NAME        PIC X(20)BBB          VALUE SPACES.
      03 AGE         PIC XXBBB             VALUE SPACES.
      03 EMP-NO      PIC XXXBXXBXXXXBBB    VALUE SPACES.
   02 EXEMPTIONS     PIC 99                VALUE ZERO.
      .
      .
      .
PROCEDURE DIVISION.
      .
      .
      .
```

Given the fields described above, the MOVE statement,

```
MOVE NUM-IN TO FIELD
```

gives the result,

```
1209/9Δ/ΔΔ
```

Note that the spaces to the left were part of the editing requirements for SUB-F1, but that the spaces to the right were inserted to fill the positions required by the description of FIELD. Also note that the negative sign of NUM-IN was not moved.

The statement, `MOVE NUM-IN TO SUB-F2`, gives the result,

```
12/09/9Δ
```

Again, the space to the right was supplied in order to fill the field, and no operational sign was moved.

Assuming that the current contents of FILE-REC are, in order

NAME	JASONΔPENNYΔΔΔΔΔΔΔΔΔ
AGE	39
EMP-NO	585241215
LOCALE	WASHINGTONΔDISTRICTΔOFΔCOLUMBIAΔΔΔΔ

and the current contents of INFO-OUT are all spaces for NAME, AGE, and EMP-NO, and ZEROS for EXEMPTIONS, then the statement, MOVE CORRESPONDING FILE-REC TO INFO-OUT, gives the results, in INFO-OUT,

NAME	JASONΔPENNYΔΔΔΔΔΔΔΔΔ
AGE	39ΔΔΔ
EMP-NO	585Δ24Δ1215
EXEMPTIONS	00

Finally, the MOVE statement,

```
MOVE NUM-IN TO CARD-NUM
```

results in the contents of CARD-NUM being 1209R, since R is the zoned overpunch character for 9 in a negative number.

MULTIPLY STATEMENT

The MULTIPLY statement multiplies a number by one or more other numbers and stores the result in one or more locations. It has two formats:

Format 1

```
MULTIPLY { identifier-1 } BY identifier-2 [ ROUNDED ]
         { literal-1    }

      [ , identifier-3 [ ROUNDED ] ] ...
                    [ ; ON SIZE ERROR imperative-statement ]
```

Format 2

```
MULTIPLY { identifier-1 } BY { identifier-2 }
         { literal-1    }    { literal-2    }

    GIVING identifier-3 [ ROUNDED ] [ , identifier-4 [ ROUNDED ] ] ...
     [ ; ON SIZE ERROR imperative-statement ]
```

Where

identifier-1, identifier-2, and so forth	are each numeric elementary items, except that in format 2, each identifier following the GIVING keyword may be numeric edited elementary items.
literal-1 and ***literal-2***	are each any numeric literal.

The composite of operands (that is, the hypothetical data item resulting from the superimposition of all receiving data items aligned on their decimal points) in any given MULTIPLY statement must not contain more than eighteen digits.

The ROUNDED and SIZE ERROR phrases, as well as multiple results and overlapping operands, are discussed in Section X.

When you use format 1 of the MULTIPLY statement, *literal-1* or the contents of *identifier-1* is multiplied, in turn, by the identifiers following the BY keyword. The result of each multiplication is stored in each of the identifiers following the BY keyword immediately after that particular product is determined.

When format 2 is used, the value of *identifier-1* or *literal-1* is multiplied by the value of *identifier-2* or *literal-2*, and the resulting product is stored in each identifier following the GIVING keyword.

Examples of the MULTIPLY statement:

```
MULTIPLY 2 BY ROOT, SQ-ROOT, ROOT-SQUARED.
```

If ROOT has the value, 2, SQ-ROOT the value square root of 2, and ROOT-SQUARED the value 4, then the above MULTIPLY statement results in ROOT now having the value 4, SQ-ROOT, the value the square root of 2, and ROOT-SQUARED, the value 8.

Assuming ROOT to be 2, the MULTIPLY statement,

```
MULTIPLY ROOT BY ROOT GIVING ROOT-SQUARED
```

assigns the value 4 to ROOT-SQUARED.

OPEN STATEMENT

The OPEN statement opens a specified file or files. It also performs checking and writing of labels, and other input or output operations.

Format

```
      { INPUT file-name-1 [ REVERSED     ] [ , file-name-2 [ REVERSED     ] ] ... }
      {                   [ WITH NO      ] [               [ WITH NO      ] ]     }
      {                   [   REWIND     ] [               [   REWIND     ] ]     }
      {                                                                           }
OPEN  { OUTPUT file-name-3  [ WITH NO REWIND ]                                    } ...
      {   [ , file-name-4 [ WITH NO REWIND ] ] ...                                }
      {                                                                           }
      { I-O file-name-5  [ , file-name-6 ] ...                                    }
      {                                                                           }
      { EXTEND file-name-7  [ , file-name-8 ] ...                                 }
```

Where *file-name-1* through *file-name-8* are the files to be opened.

The NO REWIND and EXTEND phrases can only be used for sequential files. They have no meaning for indexed, random, or relative files, and must not be used for such files.

The REVERSED phrase is not implemented in COBOL II/3000. If used, it is treated as a comment.

You can use a single OPEN statement to open several files. The files to be opened need not have the same organization or access. However, each file must have a description equivalent to the description used for it when it was created.

Prior to the successful execution of an OPEN statement for a file, the file must not be referenced, implicitly or explicitly, by the execution of a statement. The exception to this rule is the referencing of a sequential file in a SORT or MERGE statement specifying a USING or GIVING phrase.

When a file has been successfully opened, its associated record area is made available to the program. However, execution of an OPEN statement does not obtain or release any data record of the opened file. The various input/output statements must be used to do this.

The INPUT, OUTPUT, I-O, or EXTEND phrases must only be used once in an OPEN statement for any given file. You may open a file with any of the phrases in the same program. However, the file must be closed each time using a CLOSE statement without the LOCK phrase (without the REEL/UNIT phrase in the case of sequential files) before another OPEN statement can be issued for that file.

The INPUT phrase opens a file for input operations. If the OPTIONAL phrase is specified in the SELECT clause of a sequential file, and the file is not present, the first READ statement for the file causes an AT END condition.

The OUTPUT phrase creates a file if it does not already exist, and opens it for output operations. When the output file is opened, it contains no data records. The file created is a job/ session temporary file using the formal file designator specified in the SELECT clause. The key file name for an INDEX file is the formal name plus an alphabetic "K". If the file name is already 8 characters long then the last character is replaced by "K".

The I-O phrase permits the opening of a file for both input and output operations. Since this phrase implies the existence of the associated file, it cannot be used if the file is being initially created, as it would be if the file either does not exist or if the file resides on a mass storage device.

When files are opened with the INPUT or I-O phrase, the OPEN statement (without the EXTEND or NO REWIND phrases in the case of sequential files) sets the current record pointer to the first record currently existing within the file. (Indexed files use the prime record key to determine the first record to be accessed.) If no records exist in the file, the current record pointer is set in such a manner that the next executed format 1 READ statement for the file results in an AT END condition.

Label Records

The LABEL RECORDS clause of the file description entry for a file indicates whether label records are present in the file.

If label records are present, then

A. When the INPUT phrase is used in the OPEN statement, standard labels are checked in accordance with the conventions for input label checking. Any user labels specified for the file are processed according to the procedure specified by a format 2 USE statement.

B. When the OUTPUT phrase is used in the OPEN statement, standard labels are written in accordance with the conventions for output label writing. Any user labels specified for the file are written according to the procedure specified by a format 2 USE statement.

C. When the I-O phrase is used in an OPEN statement, standard labels are checked in accordance with the conventions for input-output label checking. New standard labels are written in accordance with input-output label writing. Any user labels specified for the file are processed according to the procedure specified by a format 2 USE statement.

When label records are specified, but are not present, and the file was opened using the INPUT phrase, an input-output routine error results.

When label records are present, but not specified, and the file was opened using the INPUT phrase, the label records are ignored.

Extend, Reverse, And No Rewind Phrases

The EXTEND, REVERSE, and NO REWIND phrases apply only to sequential files.

The REVERSE and NO REWIND phrases are not recognized by the COBOL II/3000 compiler, and are treated as comments if specified.

The EXTEND phrase, when specified in an OPEN statement, positions the file immediately following the last logical record of the file. Subsequent WRITE statements for the file add records at the end of the file as though the file had been opened with the OUTPUT phrase.

Note that if you specify the EXTEND phrase for multiple file reels, a compilation error diagnostic will appear; this is done to conform to ANSI COBOL'74 standards. The object program on an HP computer system will allow the EXTEND operation to execute, even for multiple file reels. However, any files following the referenced file are written over, and are made inaccessible.

When the EXTEND phrase is specified, and the LABEL RECORDS clause of the file description for the file indicates the existence of label records, the execution of the OPEN statement includes the following steps:

A. The beginning file labels are processed only if the file resides on a single reel or unit. Any user labels specified for the file are processed according to the procedure specified by a format 2 USE statement.

B. Processing then proceeds as though the file had been opened with the OUTPUT phrase.

Permissible Statements

The following tables indicate the statements permitted to be executed for a file of a given organization opened in a given open mode.

Table 11-2
SEQUENTIAL ORGANIZATION

STATEMENT	OPEN MODE			
	Input	Output	Input-Output	Extend
READ	X		X	
WRITE		X	X	X
REWRITE			X	

WRITE with Input-Output mode is a COBOL II/3000 extension to ANSI COBOL '74.

Table 11-3
RELATIVE AND INDEXED ORGANIZATION

FILE ACCESS MODE	Statement	OPEN MODE		
		Input	Output	Input-Output
Sequential	READ	X		X
	WRITE		X	X
	REWRITE			X
	START	X		X
	DELETE			X
	SEEK(Rel only)	X		X
Random	READ	X		X
	WRITE		X	X
	REWRITE			X
	START	X		X
	DELETE			X
	SEEK(Rel only)	X		X
Dynamic	READ	X		X
	WRITE		X	X
	REWRITE			X
	START	X		X
	DELETE			X
	SEEK(Rel only)	X		X

Table 11-4
RANDOM ORGANIZATION

STATEMENT	OPEN MODE		
	Input	Output	Input-Output
SEEK	X		X
READ	X		X
REWRITE	X		
WRITE		X	X

File Status Data Item

If the file named in the OPEN statement has a FILE STATUS data item associated with it, the FILE STATUS data item is updated following the execution of the OPEN statement to indicate whether or not the attempt to open the file was successful. Refer to section VII, the FILE STATUS CLAUSE, for valid combinations of status keys 1 and 2.

PERFORM STATEMENT

The PERFORM statement transfers control explicitly to one or more procedures, and returns control implicitly when execution of the specified procedure or procedures is complete.

There are four general formats of the PERFORM statement:

Format 1

```
PERFORM procedure-name-1 [ { THROUGH } procedure-name-2 ]
                         [ { THRU    }                  ]
```

Format 2

```
PERFORM procedure-name-1 [ { THROUGH } procedure-name-2 ] { identifier-1 } TIMES
                         [ { THRU    }                  ] { integer-1    }
```

Format 3

```
PERFORM procedure-name-1 [ { THROUGH } procedure-name-2 ] UNTIL condition-1
                         [ { THRU    }                  ]
```

Format 4

```
PERFORM procedure-name-1 [ { THROUGH } procedure-name-2 ]
                         [ { THRU    }                  ]

VARYING { identifier-2 } FROM { identifier-3 }
        { index-name-1 }      { index-name-2 }
                              { literal-1    }

     BY { identifier-4 } UNTIL condition-1
        { literal-2    }
[ AFTER { identifier-5 } FROM { identifier-6 }                 ]
[       { index-name-3 }      { index-name-4 }                 ]
[                             { literal-3    }                 ]
[                                                              ]
[    BY { identifier-7 } UNTIL condition-2                     ]
[       { literal-4    }                                       ]
[ [ AFTER { identifier-8 } FROM { identifier-9 }             ] ]
[ [       { index-name-5 }      { index-name-6 }             ] ]
[ [                             { literal-5    }             ] ]
[ [                                                          ] ]
[ [    BY { identifier-10 } UNTIL condition-3                ] ]
[ [       { literal-6     }                                  ] ]
```

Where

procedure-name-1 and ***procedure-name-2***	are names of procedures within the Procedure Division of the program in which the PERFORM. statement appears. If one of the procedures is a declarative procedure, then both must be.
identifier-1 through ***identifier-10***	are all numeric elementary items described in the Data Division.
literal-1 through ***literal-6***	are all numeric literals.
condition-1 through ***condition-3***	are each any valid COBOL condition. See Section X for a discussion of conditions.
THRU	is equivalent to THROUGH.

Rules Of The Perform Statement

When a PERFORM statement is executed, control is transferred to the first statement of the procedure named *procedure-name-1.* This transfer occurs only once for each execution of a PERFORM statement.

An implicit transfer of control to the next executable statement following the PERFORM statement is established as follows:

A. If *procedure-name-1* is a paragraph name, and *procedure-name-2* is not specified, then the return occurs after the last statement of *procedure-name-1.*

B. If *procedure-name-1* is a section name, and *procedure-name-2* is not specified, then the return occurs after the last statement of the last paragraph of the section named by *procedure-name-1.*

C. If *procedure-name-2* is specified as well as *procedure-name-1*, and *procedure-name-2* is a paragraph name, then the return occurs after the last statement of the paragraph.

D. If *procedure-name-2* is specified, and is the name of a section, the return occurs after the last statement of the last paragraph in the section.

No relationship is necessary between *procedure-name-1* and *procedure-name-2* except that a consecutive sequence of operations is to be executed beginning at the procedure named *procedure- name-1*, and ending with the execution of the procedure named by *procedure-name-2.*

GO TO and PERFORM statements may appear between *procedure-name-1* and the end of *procedure-name-2*, provided that the logical paths developed by these statements eventually lead back to the last statement in *procedure-name-2*.

If there are two or more logical paths to the return point of a PERFORM statement, *procedure-name-2* may be a paragraph consisting solely of the EXIT statement, in which case all alternate paths must end at this paragraph.

Note that this implies that a GO TO statement may not be the last statement in *procedure-name-2*, or in *procedure-name-1*, if *procedure-name-2* is unused, even if the GO TO statement eventually leads to the statement immediately following the PERFORM statement.

If the procedures referenced by *procedure-name-1* or *procedure-name-2* are executed as a result of any other than a PERFORM statement, there need be no concern about returning to the statement following the PERFORM statement. In this case, control passes from the last statement of the procedure to the next executable statement, just as if no PERFORM statement mentioned *procedure-name-1* or *procedure-name-2*.

Nested Perform Statements

In ANSI COBOL '74, if a procedure or sequence of procedures is executed as the result of a PERFORM statement, and a PERFORM statement is included in the statements comprising the procedure (or sequence of procedures), then the procedure or procedures associated with the included PERFORM statement must be either entirely enclosed by, or entirely excluded from the logical sequence of procedures referred to by the original PERFORM statement.

However, COBOL II/3000 extends the range of nested PERFORM statements by allowing for a common exit point for PERFORM statements, and for the exit point of one PERFORM statement to be contained within the range of another, subsequently called, PERFORM statement.

Thus, in COBOL II/3000, an active PERFORM statement whose execution begins within the range of another active PERFORM statement may allow control to pass beyond the exit of the other active PERFORM statement, or may share its exit with the other active PERFORM statement.

To illustrate this,

```
    .
    .
    .
PERFORM ALPHA THRU NEWT.
    .
    .
    .
ALPHA.
    .
    .
    .
PERFORM BETA THRU NEWT.
    .
    .
    .
BETA.
    .
    .
    .
OMEGA.
    .
    .
    .
NEWT.
    .
    .
    .
PERFORM DELTA THRU EPSILON.
    .
    .
    .
DISPLAY "END OF ALPHA THRU NEWT PERFORM"
GAMMA.
    .
    .
    .
DELTA.
    .
    .
    .
EPSILON.
    .
    .
    .
DISPLAY "END OF DELTA THRU EPSILON PERFORM"
```

The above use of nested PERFORM statements is valid, with two PERFORM statements within the range of the first. The following illustration is also valid.

```
        .
        .
        .
PERFORM ALPHA THRU BETA.
        .
        .
        .
ALPHA.                                                  )
        .                                               |
        .                                               |
        .                                               |
PERFORM GAMMA THRU DELTA.                               |
        .                                               |
        .                                               |
        .                                               |
GAMMA.                                                  } range of first PERFORM.
        .                                               |
        .                                               |
        .                                               |
BETA.                                                   |
        .                                               |
        .                                               |
        .                                               |
DISPLAY "END OF ALPHA THRU BETA PERFORM"                )
EPSILON.
        .
        .
        .
DELTA.
        .
        .
        .
DISPLAY "END OF GAMMA THRU DELTA PERFORM".
```

Range of second PERFORM (GAMMA through DISPLAY "END OF GAMMA THRU DELTA PERFORM"), which crosses over the exit for the first.

FORMAT 1 PERFORM Statement

PERFORM *procedure-name-1* [{ **THROUGH** / **THRU** } *procedure-name-2*]

Format 1 of the PERFORM statement is the basic PERFORM statement. A procedure (or procedures) is executed once as described on the preceding pages, and control then passes to the next executable statement following the PERFORM statement.

FORMAT 2 PERFORM Statement

PERFORM *procedure-name-1* [{ **THROUGH** / **THRU** } *procedure-name-2*] { *identifier-1* / *integer-1* } **TIMES**

A format 2 PERFORM statement allows you to perform the named procedure or procedures the number of times specified by *integer-1* or the numeric integer named by *identifier-1*. Following the execution of the procedures the specified number of times, control is passed to the next executable statement following the PERFORM statement.

Note that if *identifier-1* is a negative integer or zero (0) at the time of execution of the PERFORM statement, control immediately passes to the next executable statement following the PERFORM statement. Thus, a negative integer or 0 value for *identifier-1* at the time of execution is equivalent to not having the PERFORM statement in the code at all.

If *identifier-1* is referenced during execution of the PERFORM statement, it does not change the number of times the procedures are executed as specified by the initial value of *identifier-1*.

Note that if a procedure within the range of a format 2 PERFORM statement is contained in a section (or is a section) whose section number is greater than 49, the section in which it is contained is in its initial state only the first time the section is entered. Each time the section is subsequently entered as a result of the PERFORM statement, it is in its last used state. Of course, after the procedure has been executed the specified number of times, it is entered in its initial state the next time the procedure is referenced.

FORMAT 3 PERFORM Statement

PERFORM ***procedure-name-1*** [{ **THROUGH** / **THRU** } ***procedure-name-2***] **UNTIL** ***condition-1***

A format 3 PERFORM statement uses *condition-1* to control the number of times the specified procedure or procedures are performed. If *condition-1* has a truth value of "true" when the PERFORM statement is entered, control is immediately passed to the next executable statement after the PERFORM statement; *procedure-name1* through *procedure-name-2* is not executed in this case.

If *condition-1* has a truth value of "false" when the PERFORM statement is entered, the specified procedures are performed until *condition-1* has a truth value of "true".

Note that you are responsible for assuring that, within the specified procedures, *condition-1* eventually has a truth value of "true".

FORMAT 4 PERFORM Statement

```
PERFORM procedure-name-1 [ { THROUGH } procedure-name-2 ]
                           { THRU    }

VARYING { identifier-2 } FROM { identifier-3 }
        { index-name-1 }      { index-name-2 }
                              { literal-1    }

     BY { identifier-4 } UNTIL condition-1
        { literal-3    }

[ AFTER { identifier-5 } FROM { identifier-6 }
        { index-name-3 }      { index-name-4 }
                              { literal-3    }

     BY { identifier-7 } UNTIL condition-2
        { literal-4    }

  [ AFTER { identifier-8 } FROM { identifier-9 }
          { index-name-5 }      { index-name-6 }
                                { literal-5    }

     BY { identifier-10 } UNTIL condition-3 ] ]
        { literal-6     }
```

A format 4 PERFORM statement is used to augment the values referenced by one or more identifiers or index-names in an orderly fashion during the execution of a PERFORM statement.

In this format, any literal used in the BY phrase, and data items referenced by *identifier-4*, *identifier-7*, and *identifier-10* must not have a ZERO value. Also, if an index-name is used in the VARYING or AFTER phrase then,

A. If an identifier or a literal is specified in the associated FROM phrase, the data item referenced by the identifier or the value of the literal must be a positive integer.

B. If an identifier is specified in the associated BY phrase it must name an integer data item. If a literal is specified, it must be a positive integer.

If an index-name is specified in the FROM phrase, then

A. If an identifier is used in the associated VARYING or AFTER phrase, it must name an integer data item.

B. If an identifier or literal is used in the associated BY phrase, the literal or the data item referenced by the identifier must be an integer.

When an index-name appears in a VARYING or an AFTER phrase, it is initialized and subsequently augmented according to the rules of the SET statement. When an index-name appears in a FROM phrase, any identifier appearing in an associated VARYING or AFTER phrase is initialized according to the rules of the SET statement. It is subsequently augmented using the SET statement rules. Subsequent augmentation is performed as described below.

To simplify the following discussion, every reference to *parameter-n*, where n is an integer, is equivalent to referencing one of the identifiers, index-names or literals occupying the same position in the general format.

VARIATION OF A SINGLE IDENTIFIER

Variation of a single identifier is accomplished by using a format 4 PERFORM statement in the form,

```
PERFORM procedure-name-1 [ { THROUGH } procedure-name-2 ]
                           { THRU    }
VARYING parameter-1  FROM  parameter-2  BY  parameter-3
                     UNTIL condition-1
```

In this form, *parameter-1* is set to the value of *parameter-2* when the PERFORM statement is initially executed. If the truth value of *condition-1* is "true", control is transferred to the next executable statement following the PERFORM statement.

Otherwise, the sequence of procedures, *procedure-name-1* through *procedure-name-2*, is executed once.

The value of *parameter-1* is then augmented by the increment or decrement specified by *parameter-3*, and *condition-1* is evaluated. If *condition-1* is false, the cycle of executing the specified procedures, augmenting *parameter-1*, and evaluating *condition-1* is repeated. This cycle continues until *condition-1* is true, at which point control is passed to the first executable statement after the PERFORM statement.

The flowchart in Figure 11-1 illustrates variation of a single identifier.

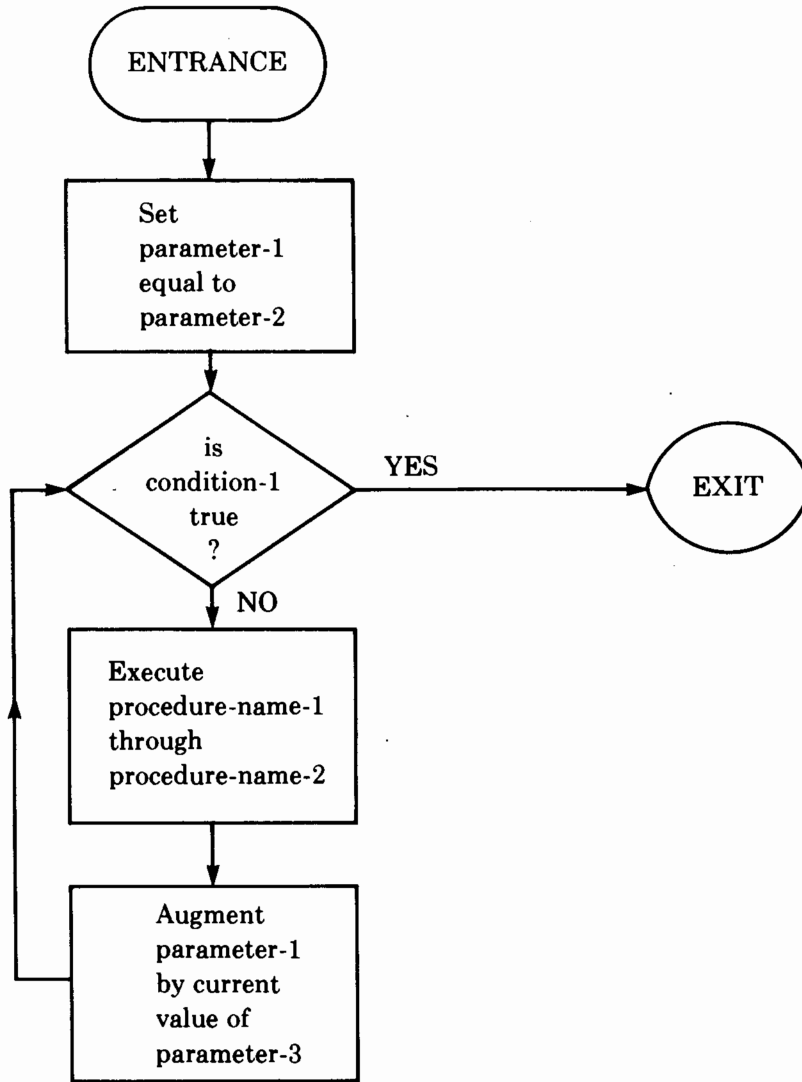


Figure 11-1
VARIATION OF A SINGLE IDENTIFIER

VARIATION OF TWO IDENTIFIERS

Variation of two identifiers is accomplished by using format 4 of the PERFORM statement in the form,

<u>**PERFORM**</u> ***procedure-name-1*** [{ <u>**THROUGH**</u> / <u>**THRU**</u> } ***procedure-name-2***]

condition-1 <u>**AFTER**</u> ***parameter-4*** <u>**FROM**</u> ***parameter-5*** **BY** ***parameter-1***
<u>**UNTIL**</u> ***condition-2***

In this case, *parameter-1* and *parameter-4* are set to the current values of *parameter-2* and *paramater-5* respectively. Following this, *condition-1* is tested, and if true, causes the PERFORM statement to cease execution, with control being transferred to the next executable statement following the PERFORM statement. If *condition-1* is false, *condition-2* is tested, with the same consequences as *condition-1* if the result is true.

If *condition-2* is false, the specified procedures are executed once, *parameter-4* is augmented by *parameter-6*, and *condition-2* is tested. This cycle continues until *condition-2* is true, at which point, *parameter-4* is set to the value of *parameter-5, parameter-1* is augmented by *parameter-3*, and *condition-1* is evaluated again. If *condition-1* is true, the PERFORM statement is complete. If it is false, the cycle using *parameter-4* and *condition-2* is repeated.

These two cycles are repeated until, after the cycle using *parameter-4* and *condition-2* is complete, *condition-1* is true.

Note that during execution of the procedures, any change in the values of *parameter-1, parameter-2, parameter-3*, or *parameter-4* is taken into consideration, and will affect the operation of the PERFORM statement.

At the end of this type of PERFORM statement, *parameter-4* has the current value of *parameter-5*, and *parameter-1* has a value differing from its last used setting by the value of *parameter-3*. This is always true except when *condition-1* is true initially, in which case, *parameter-1* has the value of *parameter-2*.

The flowchart for varying two identifiers is shown in Figure 11-2.

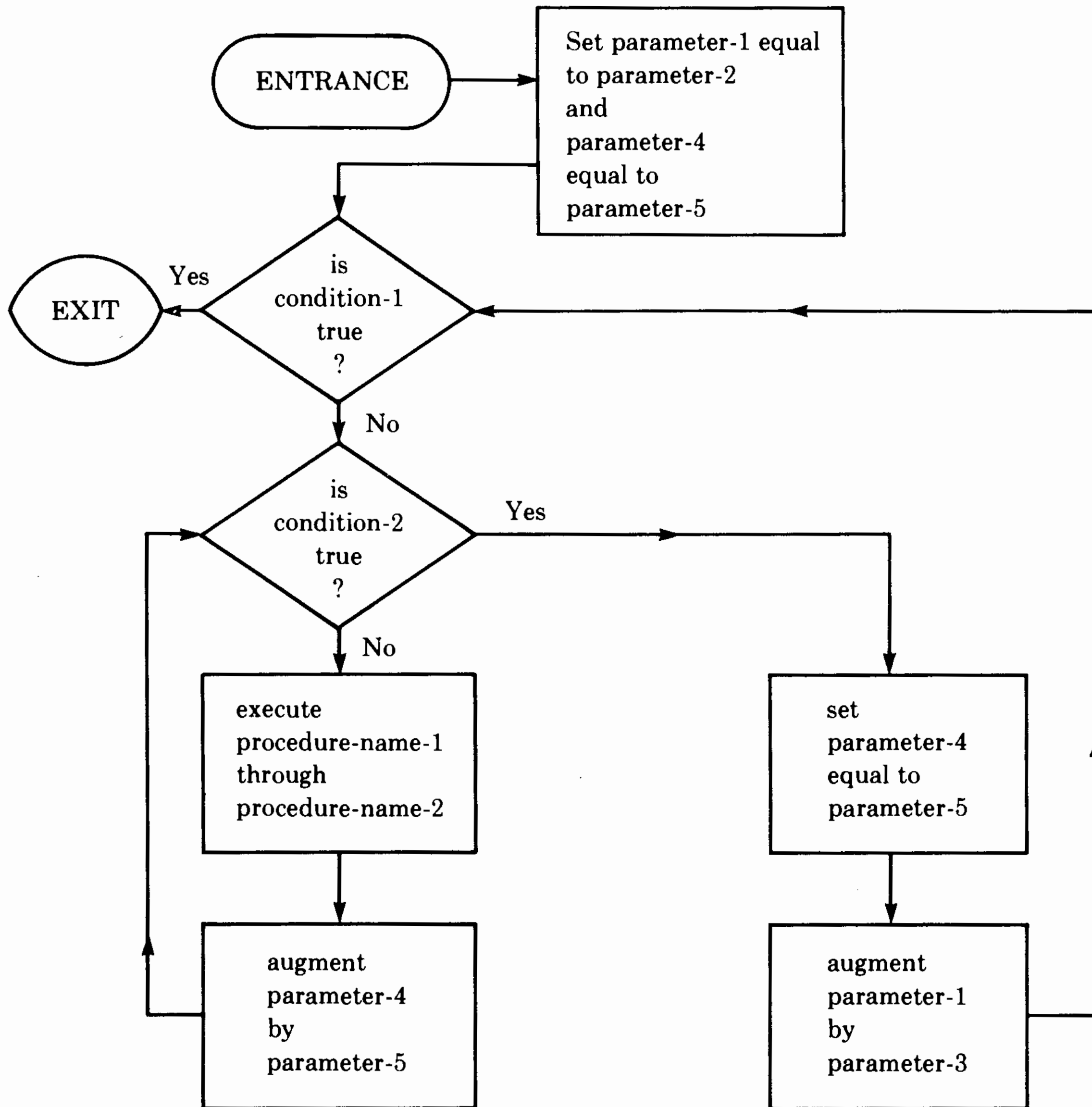


Figure 11-2
FLOWCHART FOR VARYING TWO IDENTIFIERS

VARIATION OF THREE IDENTIFIERS

Variation of three identifiers uses format 4 of the PERFORM statement in the form,

```
                                     [ { THROUGH }                    ]
PERFORM procedure-name-1             [ {         }  procedure-name-2  ]
                                     [ { THRU    }                    ]

   VARYING parameter-1 FROM parameter-2 BY parameter-3
       UNTIL condition-1
   AFTER   parameter-4 FROM parameter-5 BY parameter-6
       UNTIL condition-2
   AFTER   parameter-7 FROM parameter-8 BY parameter-9
       UNTIL condition-3
```

The mechanism for variation of three identifiers is exactly analogous to the mechanism for variation of two identifiers.

The differences are that *parameter-7* is initially set to the value of *parameter-8, condition-3* is also initially checked, and *parameter-7* goes through a complete cycle each time that *parameter-4* is augmented, which in turn goes through a complete cycle each time *parameter-1* is augmented.

After completion of a format 4 PERFORM statement, *parameter-4* and *parameter-7* have the current value of *parameter-5* and *parameter-8* respectively. *Parameter-1* has a value exceeding its preceding value by the value of *parameter-3.* The only exception to this is when *condition-1* is initially true, in which case, the value of *parameter-1* is the value of *parameter-2.*

The flow chart for varying three identifiers is shown in Figure 11-3.

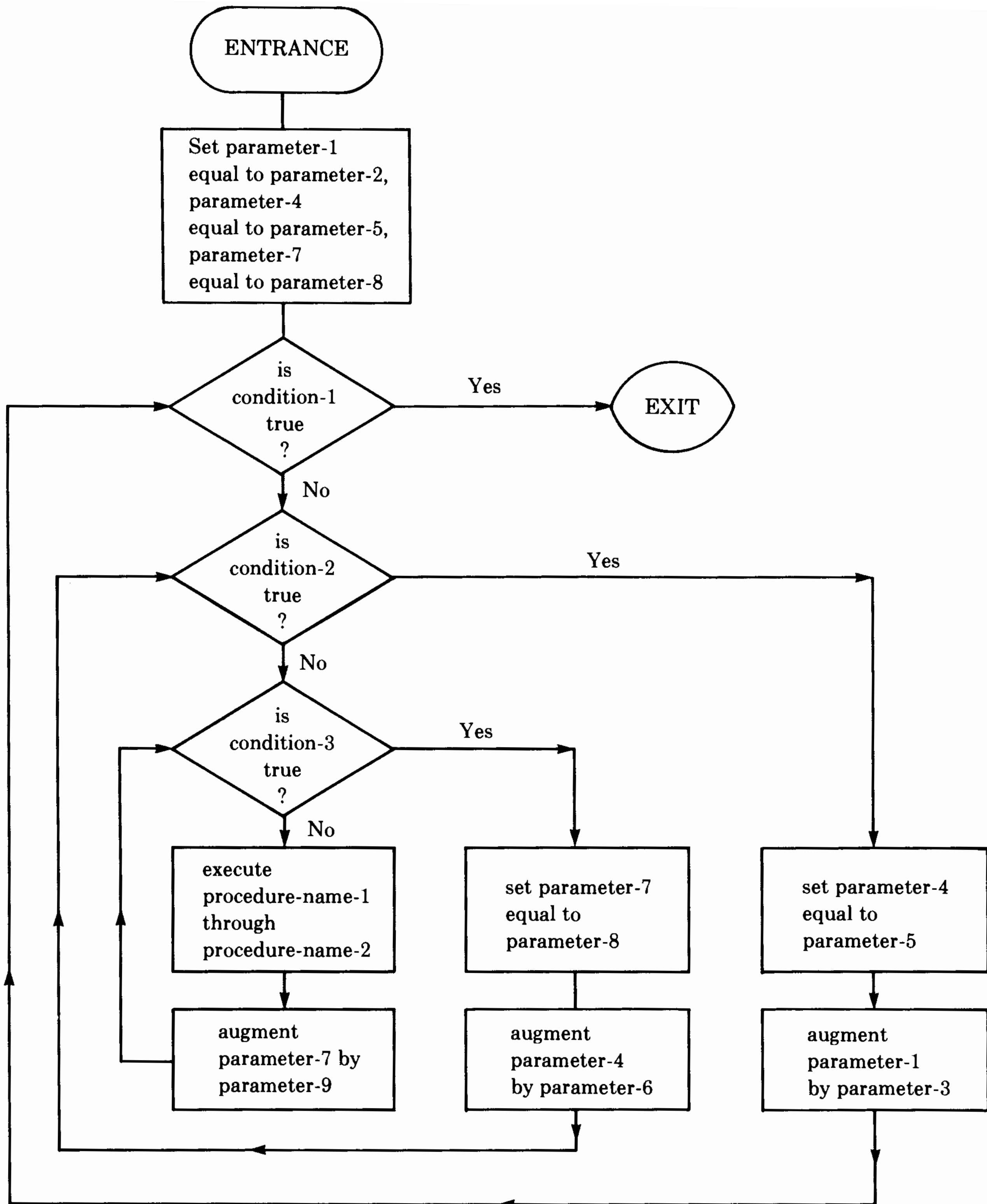


Figure 11-3
FLOW CHART FOR VARYING THREE IDENTIFIERS

Various examples of the PERFORM statement are shown below.

```
01  DISKIN-RECORD.
    03  DI-DATA                 OCCURS 4 TIMES.
        05  DI-NAME             PIC  X(20).
        05  DI-ADDRESS          PIC  X(20).
        05  DI-CTY-ST           PIC  X(20).
        05  DI-ZIP              PIC  9(05).
        05  DI-AMOUNT           PIC  9(03)V99 OCCURS 3 TIMES.
     .
     .
     .
01  WS-FILE-CTRL.
    03  WS-AMOUNT               PIC  9(06)V99 COMP VALUE 0.00.
    03  SUB-1                   PIC  9(01).
    03  SUB-2                   PIC  9(01).
    03  WS-DISKIN-CTRL          PIC  9(01)  VALUE 0.
        88  WS-DISKIN-TO-OPEN               VALUE 0.
        88  WS-DISKIN-OPEN                  VALUE 1.
        88  WS-DISKIN-EOF                   VALUE 2.
     .
     .
     .
PROCEDURE DIVISION.
200-PROGRAM-START.

** EXAMPLE OF PERFORM USING THRU OPTION *****************
    PERFORM 300-GET-DISKIN-RTN THRU 300-GET-DISKIN-EXIT.
```

```
** EXAMPLE OF PERFORM USING UNTIL OPTION *****************
     PERFORM 250-PROCESS-RECORD UNTIL WS-DISKIN-EOF.
     STOP RUN.
 250-PROCESS-RECORD.

** EXAMPLE OF PERFORM USING VARYING & UNTIL OPTIONS ******
     PERFORM 260-UNBLOCK-RECORD
        VARYING SUB-1 FROM 1 BY 1
            UNTIL SUB-1 GREATER 5.
 260-UNBLOCK-RECORD.
     MOVE DI-NAME (SUB-1)    TO P-NAME.
     MOVE DI-ADDRESS (SUB-1) TO P-ADDRESS.
     MOVE DI-CTY-ST (SUB-1)  TO P-CTY-ST.
     MOVE ZERO               TO SUB-2.

** EXAMPLE OF PERFORM USING # OF TIMES OPTION ************
     PERFORM 270-ACCUMULATE-AMOUNTS 3 TIMES.
 270-ACCUMULATE-AMOUNTS.
     ADD 1                        TO SUB-2.
     ADD DI-AMOUNT (SUB-1,SUB-2) TO WS-AMOUNT.
 300-GET-DISKIN-RTN.
     IF  WS-DISKIN-TO-OPEN
         OPEN INPUT DISKIN-FILE
         MOVE 1 TO WS-DISKIN-CTRL.
     READ DISKIN-FILE
         AT END
         CLOSE DISKIN-FILE
         MOVE 2 TO WS-DISKIN-CTRL.
 300-GET-DISKIN-EXIT.
     EXIT.
```

READ STATEMENT

The READ statement makes a record of a file available to your program. It has four formats, depending upon the type of organization of the file from which a record is made available.

Formats

Format 1 - Sequential files

<u>READ</u> ***file-name*** **RECORD [<u>INTO</u>** ***identifier*** **]**
[; AT <u>END</u> ***imperative-statement*** **]**

Format 2 - Relative, random, and indexed files

<u>READ</u> ***file-name*****[<u>NEXT</u>] RECORD [<u>INTO</u>** ***identifier*****]**
[; AT <u>END</u> ***imperative-statement*** **]**

Format 3 - Relative and random files

<u>READ</u> *file-name* RECORD [<u>INTO</u> *identifier*]
[; <u>INVALID</u> KEY *imperative-statement*]

Format 4 - Indexed files

<u>READ</u> ***file-name*** **RECORD [<u>INTO</u>** ***identifier*** **] [<u>KEY</u> IS** ***data-name*** **]**
[; <u>INVALID</u> KEY ***imperative-statement*** **]**

Where

file-name is the name of the file to be read.

identifier is a data item described in the Working Storage or File Section.

imperative-statement is one or more imperative statements, executed when an INVALID KEY or AT END condition occurs.

data-name is the name of a data item specified as a record key of the associated file. It may be qualified.

Common Rules

The following rules apply to any of the formats of the READ statement. Each format is discussed separately following these common rules.

When a READ statement is executed for a file, the file must be open in the INPUT or I-O mode.

The execution of a READ statement causes the FILE-STATUS data item (if specified) of the file being read to be updated. See section VII, the FILE STATUS CLAUSE, for valid combinations of status keys 1 and 2.

When the logical records of a file are described with more than one record description, these records automatically share the same storage area. This is equivalent to an implicit redefinition of the area. Those character positions of the current data record not filled in by the READ statement contain data items that are undefined at completion of the execution of the READ statement.

If the INTO phrase is specified, the input file must not contain logical records whose sizes vary according to their record descriptions. Also, the storage area associated with identifier, and the record area associated with the file being read must be distinct from one another. That is, these areas must not be the same storage areas.

When the INTO phrase is used, the record being read is placed into the input record area of the file, and the data item contained in the input record area is copied into the storage area of identifier according to the rules of the MOVE statement (without the CORRESPONDING phrase). This implied MOVE does not occur if the execution of the READ statement is unsuccessful.

Any subscripting or indexing associated with identifier is evaluated after the input record has been read, and immediately before it is moved into the storage area of identifier.

Format 1 Read Statement

A format 1 READ statement may be used for sequential files only.

If the file is being read for the first time following an OPEN statement, the record pointed to by the current record pointer is made available.

If one or more successful READ statements have been executed for the file, the current record pointer is updated to point to the next existing record in the file, and that record is made available.

If the end of a labeled magnetic tape reel is found during the execution of a READ statement, and the logical end of the file is not encountered, a reel swap is performed, and the first record of the new reel is made available.

Execution of a format 1 READ statement may be unsuccessful for one of three reasons:

A. The position of the current record pointer is undefined;

B. No next logical record exists in the file (that is, the end-of-file has been encountered);

C. The OPTIONAL phrase was used in the SELECT statement for the file, and the file was not present at the time the file was opened. In such a case, the standard end-of- file procedures are not performed.

Following the execution of an unsuccessful READ statement, the contents of the associated record area and the position of the record pointer are undefined. An unsuccessful READ statement as described in B and C above causes an AT END condition to occur.

When an AT END condition occurs because of an unsuccessful READ, a value is placed into the FILE STATUS data item (if specified; see the SELECT clause) to indicate an AT END condition. Next, if the AT END phrase is specified in the READ statement, control is transferred to the imperative statement or statements following the AT END keywords. Any USE procedure specified for the file is ignored. Note that if no USE procedure is specified for the file, an AT END phrase must be used for the READ statement.

If the AT END phrase is not specified, a USE procedure must be specified for the file, either explicitly or implicitly, and that procedure is executed.

When an AT END condition has occurred, no READ statement can be executed for the file without first executing a successful CLOSE statement followed by a successful OPEN statement for that file.

In the case of unlabeled magnetic tapes the AT END condition indicates an EOF mark was read. Subsequent READ statements will cause reading of the next file on the tape. Since this may cause reading off the end of the tape, the exact number of files on the tape must be known.

Format 2 Read Statement

A format 2 READ statement may be used for relative, random, or indexed files. It must be used for relative or indexed files whose access mode is sequential (see the SELECT clause); it must also be used, including the NEXT phrase, when the access mode for a relative or indexed file is dynamic and records of the file are being accessed sequentially, and to read records sequentially from a random-access file.

For a relative or indexed file being accessed sequentially, the record to be made available by a format 2 READ statement is determined as follows:

A. The record pointed to by the current record pointer is made available provided that the current record pointer was positioned by a SEEK, START or OPEN statement, and the record is still accessible through the path indicated by the current record pointer. If the record is no longer available, the current record pointer is updated to point to the next existing record (within the established key of reference for indexed files), and that record is then made available.

Note that a record may not be available because it was never written (for random-access files), it was deleted (for relative files), or because an alternate record key has been changed (for indexed files).

B. If the current record pointer was positioned by the execution of a previous READ statement, the current record pointer is updated to point to the next existing record (with the established key of reference for indexed files), and that record is made available.

A format 2 READ statement for a random-access file, or for a relative or indexed file whose access mode is dynamic, uses the NEXT phrase to position the current record pointer to the next logical record of the file. The record made available is then determined as for files opened in sequential access mode.

If the position of the current record pointer is undefined, the execution of the READ statement is unsuccessful.

If no next logical record exists for the file, and a READ statement attempts to execute for that file, the READ statement is unsuccessful, and an AT END condition occurs. The steps taken by the program in such a situation are essentially the same as for a sequential file.

That is,

A. The value of any FILE STATUS data item specified for the file is changed to indicate an AT END condition.

B. If an AT END phrase is specified in the READ statement, control is transferred to the imperative-statement following the AT END key words, and any USE procedure specified for the file is ignored.

If no AT END phrase is specified in the READ statement, a USE procedure must be specified either implicitly or explicitly for the file, and that procedure is executed.

Note that if no USE procedure is defined for the file, the AT END phrase must be used in the READ statement.

Following the unsuccessful execution of any READ statement, the contents of the associated record area and the position of the current record pointer are undefined. For indexed files, the key of reference is also undefined.

When the AT END condition exists, a format 2 READ statement for the file must not be executed without first executing one of the following:

A. A successful CLOSE statement followed by a successful OPEN statement for that file;

B. A successful START statement for that file;

C. A successful format 3 READ statement for a relative or random-access file, or a format 4 READ statement for a indexed file.

If the RELATIVE KEY phrase is specified in the SELECT clause for a relative file, successful execution of a format 2 READ statement updates the contents of the RELATIVE KEY data item so that it contains the relative record number of the record made available.

For a random-access file, the ACTUAL KEY data item is udpated to contain the relative record number of the record just read.

For an indexed file being sequentially accessed, records having the same duplicate value in an alternate record key being used for the key of reference are made available in the same order in which they are released by WRITE statements, or by execution of REWRITE statements which create duplicate values.

Format 3 Read Statement

Format 3 of the READ statement can be used for relative files whose access mode is random, or whose access mode is dynamic when records are to be retrieved randomly. It can also be used for random-access files.

When a format 3 READ statement is executed for a relative file, the current record pointer is set to the record whose relative record number is contained in the data item named in the RELATIVE KEY phrase for the file (see the SELECT clause). This record is then made available.

If the file does not contain such a record, an INVALID KEY condition exists, and the READ statement is unsuccessful.

When an INVALID KEY condition exists, two actions are performed. First, the data-item specified by the FILE STATUS clause, if used, is updated to reflect the condition.

Next, if there is a USE procedure for the file, and an INVALID KEY condition exists but the INVALID KEY phrase was not specified in the READ statement, the procedure named in the USE statement is executed.

If there is no USE procedure specified for the file, the INVALID KEY phrase must be used in the READ statement; when an INVALID KEY condition exists control is passed to the imperative-statement in the INVALID KEY phrase.

When a format 3 READ statement is executed for a random-access file, it is equivalent to executing a SEEK statement for the file, followed by the READ statement.

The contents of the data-name specified in the ACTUAL KEY clause of the SELECT statement are used to set the current record pointer to the record to be read. The record is then made available unless an INVALID KEY condition exists.

An INVALID KEY condition exists for a random-access file if the contents of the ACTUAL KEY data item do not point to a record within the file. When this occurs, the imperative-statement in the INVALID KEY phrase is executed.

Format 4 Read Statement

Format 4 of the READ statement is used for indexed files whose access mode is random, or is dynamic, and when records are to be retrieved randomly.

If the KEY phrase is specified, data-name is established as the key of reference for this retrieval. Also, if the access mode of the file is dynamic, this key of reference is used for retrievals by any subsequent executions of format 2 READ statements for the file until a different key of reference is established for the file.

If the KEY phrase is not specified, the prime record key, as specified by the RECORD KEY clause of the SELECT statement, is established as the key of reference for retrieval. It acts in the same manner as an alternate key when format 3 READ statements are subsequently issued.

Execution of a format 4 READ statement causes the value of the key of reference to be compared with the value contained in the corresponding data item of the stored records in the file.

The records used in the comparison are selected according to the ascending values of their keys, and not by the order in which the records were written (called "chronological order").

The comparison continues either until the first record having the same value is found, or until no such value is found.

If a value is found which matches the key of reference, the current record pointer is positioned to the record containing the matched value, and that record is made available to your program.

If the comparison fails, an INVALID KEY condition exists, and execution of the READ statement is unsuccessful.

When an INVALID KEY condition occurs, and no INVALID KEY phrase is specified, a USE procedure must be specified, and is executed.

When an INVALID KEY condition occurs, and the INVALID KEY phrase is specified, control is transferred to the imperative-statement appearing in the INVALID KEY phrase; any USE procedure that was specified for the file is ignored.

REWRITE STATEMENT

The REWRITE statement logically replaces an existing record in a sequential, relative, random, or indexed mass storage file. Note that the current record pointer is unaffected by the execution of a REWRITE statement.

There are two formats of the REWRITE statement:

Format 1 - Sequential Files

REWRITE *record-name* [FROM *identifier*]

Format 2 - Relative, Random, and Indexed Files

REWRITE *record-name* [FROM *identifier*]
[; INVALID KEY *imperative-statement*]

Where

record-name	is the name of a logical record in the File Section of the Data Division. *Record-name* may be qualified. It may not refer to the same storage area as that referred to by *identifier*.
identifier	is the name of a data item in the program. It may be described in any section of the Data Division, but must not refer to the same storage area as that referred to by *record-name*.
imperative-statement	is one or more imperative statements.

The file associated with *record-name* must be a mass storage file, and must be opened in I-O mode at the time of execution of the REWRITE statement.

The number of character positions in the record referenced by *record-name* must be equal to the number of character positions in the record being replaced.

For sequential files, and for relative or indexed files open in sequential access mode (see the SELECT statement), the last input-output statement executed for the associated file prior to the execution of the REWRITE statement must have been a READ statement. The record replaced by the REWRITE statement is the record accessed by the READ.

For indexed files this is accomplished by using the primary key. Thus, the value contained in the primary record key data item of the record to be replaced must be equal to the value of the primary record key of the last record read from the file. Note that if an indexed file has the DUPLICATES phrase specified for its primary record key, the REWRITE statement should be used only when the indexed file is in sequential access mode. This is because a REWRITE statement issued for such a file whose access mode is dynamic or random will only rewrite the first record having the duplicate primary key.

If the primary record key data item of the record to be replaced is not equal to the value of the primary record key of the last record read from the file, an INVALID KEY condition exists. In such a case, the REWRITE operation fails, and the record which was to be replaced is unaffected. Also, if the INVALID KEY phrase is specified, control is passed to the *imperative- statement* of that phrase, whether a USE procedure is specified for the file or not. If a USE procedure is not specified for the file, the INVALID KEY phrase must be specified. However, if a USE procedure is specified, and an INVALID KEY phrase is not, the USE procedure is executed when an INVALID KEY condition exists for the file.

Note that if a relative file is open in sequential access mode, the INVALID KEY phrase must not be used.

If a random file is open, or a relative file is open in random or dynamic access mode, the record to be logically replaced is specified by the contents of the RELATIVE KEY or ACTUAL KEY data item associated with the file. If the file does not contain the record specified by the key, an INVALID KEY condition exists. Thus, the operation does not succeed, and the data in the record area is unaffected. Also, if no USE procedure has been defined for the file, the INVALID KEY phrase must be specified, and when an INVALID KEY condition exists, control is transferred to the *imperative-statement* of that phrase. If a USE procedure has been defined, and the INVALID KEY phrase is not specified, the USE procedure is executed when an INVALID KEY condition exists. However, if both a USE procedure and an INVALID KEY phrase are specified, the USE procedure is ignored, and control is transferred to the *imperative-statement* of the INVALID KEY phrase.

For indexed files open in dynamic or random access mode, the record to be replaced is specified by the primary record key data item. An INVALID KEY condition exists for this type of REWRITE if the value of the primary record key in the record to be rewritten does not equal that of any record stored in the file, or when the value contained in an alternate record key data item equals the value of an alternate record key of another record, and the DUPLICATES clause has not been specified for that key in the SELECT statement for that file. The action taken for the occurrence of an INVALID KEY condition when an indexed file is open in random or dynamic access is the same as for when the indexed file is open for sequential access.

When an indexed file is the object of a REWRITE statement, the contents of alternate record key data items of the record being rewritten may differ from those of the record being replaced. These alternate keys are used during the execution of the REWRITE statement in such a way that subsequent access of the record may be made based upon any of the specified record keys.

The logical record written by a successful REWRITE statement is no longer available in the record area (memory) unless the associated file (whether indexed, relative or sequential) is named in the SAME RECORD AREA clause.

If the file is named in the same RECORD AREA clause, the written logical record is available to your program as a record of other files named in the clause, as well as to the file associated with the record to be replaced. That is, it remains in memory.

When a REWRITE statement completes execution, whether successfully or not, the value of the FILE STATUS data item, if any, associated with the file being accessed in the REWRITE statement is updated. See section VII, the FILE STATUS CLAUSE, for valid combinations of status keys 1 and 2.

From Phrase

If the FROM phrase is used in a REWRITE statement, execution of the statement is equivalent to the execution of the MOVE statement,

MOVE ***identifier-1*** **TO** ***record-name***

followed by the execution of the same REWRITE statement without the FROM phrase. The contents of the record area prior to the execution of the implicit MOVE have no effect upon the execution of the REWRITE statement.

The REWRITE statement executed following the implicit MOVE follows the rules and restrictions listed above.

Release Statement

The RELEASE statement is discussed in Section XIII, SORT/MERGE OPERATIONS IN COBOL II.

RETURN STATEMENT

The RETURN statement is discussed in Section XIII, SORT/MERGE OPERATIONS IN COBOL II.

SEARCH STATEMENT

The SEARCH statement is used to search a table for an element satisfying a specified condition, and to set the index associated with the index-name of the table to the value of the occurrence number of the element. The two formats of the SEARCH statement are:

Format 1

```
SEARCH identifier-1 [ VARYING { identifier-2   } ]
                              { index-name-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 ]

                                                 { identifier-3              }
    ; WHEN { data-name-1 { IS EQUAL TO }        { literal-1                 } }
           {             { IS =        }        { arithmetic-expression-1   } }
           { condition-name-1                                                 }

        [       {                                 { identifier-4            } } ]
        [ AND   { data-name-2 { IS EQUAL TO }     { literal-2               } } ] ...
        [       {             { IS =        }     { arithmetic-expression-2 } } ]
        [       { condition-name-2                                          } ]

        { imperative-statement-2 }
        { NEXT SENTENCE          }
```

Where

identifier-1	is the name of a table. It must not be subscripted or indexed, and must contain an OCCURS and an INDEXED BY clause in its description. If used in format 2, it must also contain a KEY IS phrase in its OCCURS clause.
identifier-2	if used, must be described either as USAGE IS INDEX, or as a numeric elementary item with no positions to the right of the assumed decimal point.
condition-1, condition-2, and so forth	may be any condition as described under the heading, CONDITIONAL EXPRESSIONS, in Section X.
condition-name-1, condition-name-2, and so forth	are condition-names whose descriptions (level 88) list only a single value. The data-name associated with a condition-name must appear in the KEY IS clause of *identifier-1.*
data-name-1, data-name-2, and so forth	may each be qualified, and must each be indexed by the first index-name associated with *identifier-1* as well as any other indices or literals as required. Each must also be referenced in the KEY IS clause of *identifier-1.*
identifier-3 and ***identifier-4*** and so forth	must not be referenced in the KEY IS clause of *identifier-1* or indexed by the first index-name associated with *identifier-1.*
arithmetic-expression-1, arithmetic-expression-2, and so forth	can be any arithmetic expression as described under the heading, ARITHMETIC EXPRESSIONS, in Section X. However, any identifiers appearing in any arithmetic expression in a SEARCH statement are subject to the same restrictions as *identifier-3* and *identifier-4.*
imperative-statement, imperative-statement-1, and so forth	are each one or more imperative statements.

If *identifier-1* is a data item subordinate to a data item containing an OCCURS clause (that is, is a two or three dimensional table), an index-name must be associated with each dimension of the table represented by *identifier-1.* This is done through the INDEXED BY phrase of the OCCURS clause for *identifier-1.* Only the setting of the index-name associated with identifier-1 (and the data item named by *identifier-2* or *index-name-1* if used) is modified by the execution of the SEARCH statement.

To search an entire two or three dimensional table, it is necessary for you to execute a SEARCH statement several times. SET statements must be executed whenever it is necessary to adjust index-names to appropriate settings. adjusted to appropriate settings.

Format 1 of the Search Statement

When you use a format 1 SEARCH statement, a serial search is performed, starting with the current index setting of the indexname associated with *identifier-1.*

If the index-name associated with *identifier-1* contains a value corresponding to an occurrence number greater than the highest possible occurrence number of *identifier-1,* the SEARCH statement is terminated immediately, and if an AT END phrase is specified, the *imperative-statement* within the phrase is executed. If an AT END phrase is not specified, control passes to the next executable statement following the SEARCH statement.

If the index-name associated with *identifier-1* contains a value corresponding to an occurrence number within the range of *identifier-1,* the SEARCH statement evaluates each condition in the order that it is written. If none of the conditions are satisfied, the index-name for *identifier-1* is incremented to reference the next occurrence. The process of evaluating each condition is then repeated using the new index-name settings, provided the value of the index-name is within the permissible range of occurrences for *identifier-1.* If the new setting is not within range, the search terminates in the manner described in the preceding paragraph.

If one of the conditions is satisfied when it is evaluated, the search terminates and the *imperative-statement* associated with that condition is executed. The index-name retains the value at which it was set when the condition was satisfied.

After execution of the *imperative-statement,* if it is not a GO TO statement, control passes to the next executable sentence.

VARYING PHRASE

If you use the VARYING phrase, *index-name-1* may or may not appear in the INDEXED BY phrase of *identifier-1.* If it does, *index- name-1* is the index-name used in the search. If it does not, or if you specified *identifier-2* instead, the first (or only) indexname given in the INDEXED BY phrase of *identifier-1* is used for the search.

If you specify VARYING *index-name-1,* and *index-name-1* is associated with a different table (that is, does not appear in the INDEXED BY phrase of *identifier-1),* the occurrence number represented by *index-name-1* is incremented by the same amount as, and at the same time as, the occurrence number represented by the index-name associated with *identifier-1.*

If you specify VARYING *identifier-2,* and *identifier-2* is an index data-item, the data item represented by *identifier-2* is incremented in the same way as for *index-name-1* in the preceding paragraph.

If *identifier-2* is not an index data-item, the data item referenced by *identifier-2* is incremented by one (1) at the same time that the index referenced by the index-name associated with *identifier-1* is incremented.

If you do not specify the varying phrase, the index-name used for the search operation is the first (or only) index-name appearing in the INDEXED BY phrase of *identifier-1.* Any other index-name for *identifier-1* remains unchanged.

The flowchart in figure 11-4 shows the execution of a format 1 SEARCH statement specifying two WHEN phrases.

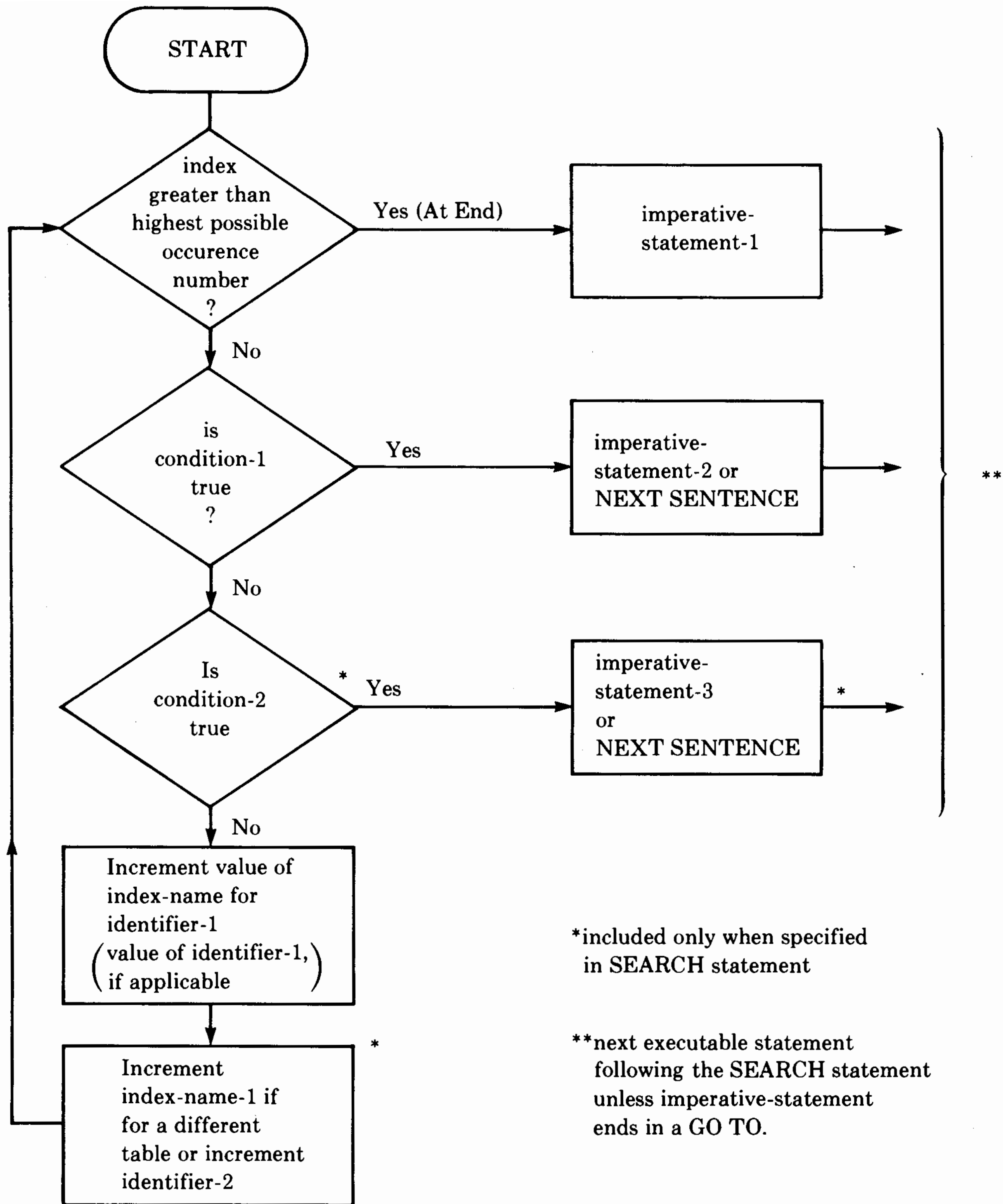


Figure 11-4

EXECUTION OF FORMAT 1 SEARCH STATEMENT

FORMAT 2 SEARCH Statement

If you use a format 2 SEARCH statement, a binary search is used.

The index-name used in the search is the first (or only) index-name appearing in the INDEXED BY phrase of identifier-1. Any other index-names for identifier-1 remain unchanged.

The results of a format 2 SEARCH statement are predictable only under two conditions:

A. The data in the table is ordered in the same manner as described in the ASCENDING (or DESCENDING) KEY clause associated with the description of *identifier-1.*

B. The contents of the key or keys referenced in the WHEN clause are sufficient to identify a unique table element.

When a data-name in the KEY clause of *identifier-1* is referenced, or when a condition-name associated with a data-name in the KEY clause of *identifier-1* is referenced, all preceding data-names in the KEY clause of *identifier-1* or their associated condition- names must also be referenced. The maximum number of reference keys is 12.

When the search begins, the initial setting of the index-name associated with *identifier-1* is ignored and its setting is varied during the search in such a way as to make the search as fast as possible. Of course, it is not set to a value outside of the possible range of occurrence numbers for the table.

If there is no possible valid setting for the index-name associated with *identifier-1* which satisfies all the conditions specified in the WHEN clause, control passes to the imperative statement specified in the AT END phrase if it has been specified. If the AT END phrase is not specified, and the conditions cannot all be satisfied, control passes to the next executable sentence. In either case, the final setting of the index is not predictable.

If all conditions in the WHEN phrase can be satisfied, the index indicates the occurrence that allows the conditions to be satisfied, and control passes to *imperative-statement-2.*

After execution of *imperative-statement-2*, providing that *imperative-statement-2* does not contain a GO TO statement, control passes to the next executable sentence.

The example below uses both formats of the SEARCH statement. In the format 1 SEARCH statement, a search is performed to find out if a carton is part of the inventory. If it is part of the inventory, its first dimensions (there are five possible) as well as its part number are displayed.

The second SEARCH statement is a format 2 SEARCH. It is used to find a container of a given volume whose height and width both equal 5.

```
        .
        .
        .
    01  PARTS-TABLE.
        02  PARTS-INFO OCCURS 10 TIMES INDEXED BY PT-INDX.
            03  PART-NAME           PIC X(20).
            03  PART-NUMBER         PIC X(10).
            03  MEASURES OCCURS 5 TIMES
                ASCENDING KEY IS VOLUME, HEIGHT, WIDTH
                         INDEXED BY IND-T2.
                          04  HEIGHT     PIC 999V999.
                          04  WIDTH      PIC 999V999.
                          04  LGTH       PIC 999V999.
                          04  VOLUME     PIC 999V999.
                              .
                              .
                              .
```

Computer
Museum


```
WORKING-STORAGE SECTION.
77 NEXT-NUM   PIC 99 VALUE 1.
77 INCRE  PIC 9  VALUE 1.
01  CONTAINER-INFO.
    02  PT-NAME                    PIC X(20).
    02  FILLER                     PIC X(5)  VALUE SPACES.
    02  PT-NO                      PIC X(10).
        03  FILLER                     PIC X(3).
        03  DIMENSIONS.
            04 H                           PIC Z(5).999.
            04 W                           PIC Z(5).999.
            04 L                           PIC Z(5).999.
            04 V                           PIC Z(5).999.
                         .
                         .
                         .
PROCEDURE DIVISION.
                         .
                         .
                         .
    SET IND-T2, PT-INDX TO 1.
    SEARCH PARTS-INFO; AT END PERFORM UNFOUND-RTN
         WHEN PART-NAME, (PT-INDX)  = 'CARTON' PERFORM FOUND-IT.
                         .
                         .
                         .
 FOUND-IT.
              MOVE MEASURES (PT-INDX, IND-T2) TO DIMENSIONS.
              MOVE PART-NAME (PT-INDX) TO PT-NAME
              MOVE PART-NUMBER (PT-INDX) TO PT-NO
               DISPLAY HEADER.
               DISPLAY CONTAINER-INFO.
               DISPLAY SPACES.
                         .
                         .
                         .
         SET PT-INDX TO NEXT-NUM.
         SET IND-T2 TO INCRE.
```

```
CONTAINER-SELECTION SECTION.
      DISPLAY "THIS SECTION SEARCHES FOR A CONTAINER"
      DISPLAY "OF A SPECIFIED VOLUME WHOSE HEIGHT AND"
      DISPLAY "WIDTH BOTH ARE FIVE.  PLEASE SPECIFY THE VOLUME"
      DISPLAY "REQUIRED (IN AN EVEN NUMBER OF CUBIC FEET.)"
      ACCEPT NEEDED-VOLUME.
      COMPUTE NEXT-NUM, INCRE = 1.
      SET PT-INDX = 1.
      SEARCH-PARTS-TABLE.

   SEARCH ALL MEASURES; AT END PERFORM NO-SUCH
       WHEN VOLUME (PT-INDX, IND-T2)
            IS EQUAL TO NEEDED-VOLUME
   AND WIDTH (PT-INDX, IND-T2) = 5 AND HEIGHT (PT-INDX, IND-T2) = 5
      PERFORM FOUND-IT.
                    .
                    .
                    .
      NO-SUCH.
      IF NEXT-NUM IS LESS THAN 10
         ADD 1 TO NEXT-NUM
         SET PT-INDX TO NEXT-NUM
         SET IND-T2 TO 1
         PERFORM SEARCH-PARTS-TABLE;
         ELSE DISPLAY "NO CONTAINER MEETS THESE REQUIREMENTS".
```

SEEK STATEMENT

The SEEK statement initiates access to a relative file whose access mode is dynamic or random, and to a random-access file, prior to execution of a format 3 READ statement.

SEEK Statement Format

SEEK ***file-name*** **RECORD**

The SEEK statement is valid only for input files; it is treated as a comment for output files.

The file specified in a SEEK statement must be opened prior to the first SEEK statement.

The SEEK statement causes a transfer of the physical record containing the logical record to be read in a subsequent READ statement from storage into main memory.

Using the SEEK statement before a format 3 READ statement may improve the performance of your program. However, because the SEEK function is implicit in the READ statement, its use is not mandatory in this case.

Two SEEK statements may logically follow each other. However, it is poor programming practice for SEEK statements to do so because of the time expended on input-output operations for data that will not be accessed.

For relative files, the SEEK statement uses the value of the data item named in the RELATIVE KEY clause of the file SELECT statement to find the desired record. Thus, before a SEEK statement is executed for a relative file, the relative record number of the desired record must be moved to the RELATIVE KEY data-item.

Similarly, for random-access files, the SEEK statement uses the value of the data-item named in the ACTUAL KEY clause of the file's SELECT statement to find the desired record.

Thus, you must move the ACTUAL KEY value for the desired record to the data-item named by the ACTUAL KEY clause before executing a SEEK statement for the file.

In either case, if the value moved to the data-item named in a RELATIVE KEY or ACTUAL KEY clause is invalid, the SEEK statement is ignored. If a subsequent format 3 READ statement is executed for the file, an INVALID KEY condition exists, and the appropriate action is taken.

SET STATEMENT

The SET statement establishes reference points for table handling operations by setting index-names associated with table elements.

The two formats of the SET statement are:

Format 1

```
      { identifier-1  [ , identifier-2 ] ...  }      { identifier-3 }
SET   {                                       }  TO  { index-name-3 }
      { index-name-1  [ , index-name-2 ] ...  }      { integer-1    }
```

Format 2

```
                                               { UP BY   }  { identifier-4 }
SET  index-name-4 [ , index-name-5 ] ...       {         }  {              }
                                               { DOWN BY }  { integer-2    }
```

Where

identifier-1, ***identifier-2,*** ***identifier-3***	must each name either index data items or elementary items described as integers.
identifier-4	must be described as an elementary numeric integer.
integer-1 and ***integer-2***	may be signed, with the restriction that integer-1 must be positive.
index-name-1, ***index-name-2,*** and so forth	are each related to a given table.

NOTE

The value of the index associated with an index-name may be undefined following the execution of a SEARCH or PERFORM statement in which it is used. Also, when a sending and a receiving item share a part of their storage areas, the result of execution of a SET statement using them is undefined.

Format 1 SET Statement

If you use *index-name-3,* the value of the associated index before execution of the SET statement must correspond to an occurrence number of an element in the table whose index is being set.

Index-name-1, if used, is set to a value causing it to refer to the table element whose occurrence number corresponds to the occurrence number of the table element referenced by *index-name- 3, identifier-3,* or *integer-1.*

The value to which *index-name-1* is set must correspond to an occurrence number of an element in the table to which *index-name- 1* is associated.

If *identifier-3* is an index data item, or if *index-name-3* is related to the same table as *index-name-1,* no conversion takes place.

If *identifier-1* is an index data item, it may be set equal to the contents of either *index-name-3* or *identifier-3* if *identifier-3* is an index data item. No conversion takes place. *Identifier-1* may not be set equal to *integer-1* in this case.

If *identifier-1* is not an index data item, only *index-name-3* may be used. The value to which *identifier-1* is set is, in this case, an occurrence number that corresponds to the value of *index-name-3.*

Any indexing or subscripting associated with *identifier-1* is evaluated immediately before the value of the indexed or subscripted data item is changed.

Any remaining identifiers or index-names are set in the same way, with the same restrictions as *identifier-1* or *index-name-1.* If *index-name-3* or *identifier-3* has been specified, its associated value is used as it was at the beginning of the execution of the SET statement.

Table 11-5 shows the validity of various operand combinations in the SET statement. An asterisk indicates that no conversion takes place, and two asterisks indicates no conversion takes place if row sizes are equal.

Table 11-5
SET STATEMENT OPERAND COMBINATIONS

Sending Item	Receiving Item		
	Integer Data Item	Index-name	Index Data Item
Integer Literal	No	Valid	No
Integer Data Item	No	Valid	No
Index-Name	Valid	** Valid	* Valid
Index Data Item	No	* Valid	* Valid

Format 2 Set Statement

When format 2 is used, the value of the index associated with an index-name to be set must correspond, before and after execution of the SET statement, to an occurrence number of an element in the associated table.

The value of *index-name-4* is incremented (if UP BY is used) or decremented (if DOWN BY is used) by the value of *integer-2* or the integer named by *identifier-4.*

If other index-names are specified, each is set up or down, as specified, just as the first was. The value of *identifier-4* is unchanged, and is used as it was at the beginning of the execution of the SET statement.

START STATEMENT

The START statement provides a basis for logical positioning within a relative or indexed file for subsequent retrieval of records.

START Statement Format

```
                          ┌      ┌ IS EQUAL TO         ┐           ┐
                          │      │ IS =                │           │
START file-name           │ KEY  │ IS GREATER THAN     │ data-name │
                          │      │ IS >                │           │
                          │      │ IS NOT LESS THAN    │           │
                          └      └ IS NOT <            ┘           ┘

     [ ; INVALID KEY imperative-statement ]
```

NOTE

The required relational characters '>', '<', and '=' are not underlined to avoid confusion with other symbols such as '≥' (greater than or equal to).

Where

file-name is the name of a relative or indexed file. The file must be open in INPUT or I-O mode when the START statement is executed.

data-name for a relative file, must be the data item named in the RELATIVE KEY phrase of the SELECT statement for the file; for an indexed file, *data-name* may reference a data item specified as a record key associated with the named index file, or it may reference any alphanumeric data item subordinate to the data-name of a data item specified as a record key of the named file, provided that its leftmost character position corresponds to the leftmost character position of that record key data item. *Data-name* may be qualified.

imperative-statement is one or more imperative statements.

The INVALID KEY phrase must be used in the START statement if no applicable USE procedure is specified for the file.

When the START statement executes, a comparison is made between a key associated with *file-name*, and a data item.

If the KEY phrase is unused, the relational operator, "IS EQUAL TO" is assumed.

The data item used in the comparison depends upon whether the file named in the START statement is a relative or an indexed file.

If the file is a relative file, the comparison uses the data item referenced in the RELATIVE KEY clause of the file's SELECT statement. This data item is always used, whether the KEY phrase is specified or not. Thus, for example, the statements,

```
MOVE 5 TO REL-KEY.
          START REL-FILE.
```

cause the current record pointer to be positioned to the fifth record of REL-FILE if REL-KEY is the name specified in the RELATIVE KEY phrase of the SELECT statement for the relative file named REL-FILE.

If the file named in the START statement is an indexed file, the data item used in the comparison depends upon whether the KEY phrase is used.

If the KEY phrase is not used for an indexed file, the primary key, that is, the data item named in the RECORD KEY clause of the file, is used.

If the KEY phrase is used, the comparison uses the data item referenced by *data-name*. Thus, this data item must be either a primary or alternate key for the file, or must be a data item whose first character is the first character of one of the keys for the file.

If the key associated with a record of an indexed file differs from the size of the data item used in the comparison, the comparison proceeds as though the longer of the two were truncated on the right so that its length is equal to the length of the shorter. A nonnumeric comparison is then performed following all the rules for such comparisons. The PROGRAM COLLATING SEQUENCE, however, is not used for the comparison, even if it was specified. The ASCII collating sequence is always used on an HP computer system.

When comparison takes place for either type of file, the current record pointer is positioned to the first logical record currently existing in the file whose key satisfies the comparison.

If the comparison is not satisfied by any record of the file, an INVALID KEY condition exists, the execution of the START statement is unsuccessful, and the *imperative-statement* of the INVALID KEY phrase (if specified) is executed. If the INVALID KEY phrase is not specified, then a USE procedure must be, and that procedure is executed. The position of the current record pointer is, in such a case, undefined.

The execution of a START statement causes the value of the FILE STATUS data item, if any, associated with the file to be updated. See section VII, the FILE STATUS CLAUSE, for valid combinations of status keys 1 and 2.

Upon completion of a successful START statement for an indexed file, a key of reference is established and used in subsequent format 2 READ statements.

If the KEY phrase is not specified in an indexed file START statement, the primary key is established as the key of reference. If the KEY phrase is specified, and *data-name* is any record key (primary or alternate) for the file, that record key becomes the key of reference.

If the KEY phrase is specified, and *data-name* is not a record key of the file, then the first character of the data-item contained in *data-name* is the same as the first character of some key for the file, and that key becomes the key of reference.

If a START statement for an indexed file is unsuccessful, then the key of reference and the current record pointer are undefined.

STOP STATEMENT

The STOP statement provides a means of temporarily suspending execution of your object program, as well as a means of stopping it completely.

STOP Statement Format

```
STOP { RUN     }
     { literal }
```

Where

RUN if specified, causes the entire run unit to cease execution when it is encountered, regardless of whether the STOP RUN statement is in a subprogram or a main program. Control is then returned to MPE.

literal may be numeric or nonnumeric, or may be any figurative constant except ALL. If the literal is numeric, it must be an integer. Use of a literal in a STOP statement temporarily suspends the object program.

If the STOP RUN statement is used in a consecutive sequence of imperative statements within a sentence, it must appear as the last statement in this sequence.

If STOP literal is used, the object program suspends and literal is displayed at the operators console. A system generated message is then displayed, followed by the message, TYPE GO TO RESUME.

When the operator responds with "GO", program execution resumes with the next executable statement following the STOP literal statement.

To illustrate:

```
STOP "MOUNT TAPE COBTEST"
OPEN INPUT COBTEST.
```

The use of the STOP statement above is to instruct the operator to mount a magnetic tape to be used as an input file for the program. After the operator mounts the tape and makes appropriate console responses, the program resumes execution when he types the word, "GO."

STRING STATEMENT

The STRING statement juxtaposes (concatenates) the partial or complete contents of two or more data items into a single data item.

STRING Statement Format

```
                                                                    { identifier-3 }
STRING { identifier-1 } [ , identifier-2 ] ... DELIMITED BY { literal-3    }
       { literal-1    } [ , literal-2    ]                  { SIZE         }

  [     { identifier-4 } [ , identifier-5 ]                   { identifier-6 } ]
  [ ,   { literal-4    } [ , literal-5    ] ... DELIMITED BY  { literal-6    } ] ...
  [                                                           { SIZE         } ]

  INTO identifier-7 [ WITH POINTER identifier-8 ]

  [ ; ON OVERFLOW imperative-statement ]
```

Where

literal-1, ***literal-2,*** and so forth	are each any figurative constant except ALL; none of them may be numeric literals.
identifier-1, ***identifier-2,*** through ***identifier-6***	are each described implicitly or explicitly as USAGE IS DISPLAY. If any of these identifiers represent an elementary numeric data item, it must be described as an integer without the 'P' symbol in its PICTURE character-string.
identifier-7	must represent an elementary alphanumeric data item without editing symbols or JUSTIFIED clause, and with a USAGE IS DISPLAY (implied or explicit) in its description.
identifier-8	must represent an elementary numeric integer data item of sufficient size to contain a value equal to the size, plus 1, of the area referenced by *identifier-7*. The symbol P must not be used in the PICTURE of *identifier-8*.
imperative-statement	is one or more imperative statements.

Description of the String Statement

All references to *identifier-1*, *identifier-2*, *identifier-3*, *literal-1*, *literal-2*, and *literal-3* apply equally to *identifier-4*, *identifier-5*, *identifier-6*, *literal-8*, *literal-5*, and *literal-6*, respectively, and all recursions thereof. Thus, to aid in the description of the STRING statement, we rewrite the format as follows:

```
STRING first sending items DELIMITED BY { delimiter-1 }
                                        { SIZE        }

[ second sending items  DELIMITED BY { delimiter-2 } ] ....
[                                    { SIZE        } ]

[ INTO identifier-7 [ WITH POINTER identifier-8 ]

  [ ; ON OVERFLOW imperative-statement ]
```

First sending items and *second sending items* represent the groups of literals and the data items named by the identifiers appearing between the STRING and DELIMITED key words, or between a delimeter (or the keyword SIZE) and the next use of the DELIMITED keyword. These are the items which are juxtaposed into *identifier-7*, the receiving data item.

Delimiter-1, and *delimiter-2* indicate the character or characters delimiting the characters moved from *first sending* items, and *second sending items*, respectively. If the SIZE phrase is used, the complete group of sending items is moved.

Note that if a figurative constant is used as a delimiter, it stands for a single character numeric literal, whose USAGE is DISPLAY. If a figurative constant is used for a literal in a group of sending items, it refers to an implicit one character data item whose USAGE is DISPLAY.

Execution of the String Statement

When the STRING statement executes, the characters in the *first sending items* are transferred to the contents of *identifier-7* in accordance with the rules of alphanumeric to alphanumeric moves, except that no space filling takes place.

If the DELIMITED phrase is specified using *delimiter-1*, the contents of the *first sending items* are moved to the contents of *identifier-7* in the sequence specified in the STRING statement, starting with the leftmost character and continuing until all character positions of *identifier-7* have been filled, or until the character or characters that make up *delimiter-1* are encountered. The characters of delimiter-1 are not transferred.

If *delimiter-1* is found in the *first sending items* before *identifier-7* is filled, the *second sending items* are processed in the same way as the first were, and transferring ceases in the same way, using *delimiter-2*, rather than *delimiter-1*.

If the DELIMITED phrase contains the word SIZE rather than *delimiter-1* or *delimiter-2*, then either all characters in the sending items are transferred to *identifier-7*, or as many characters as is possible are transferred before the end of the data area reserved for *identifier-7* has been reached.

The POINTER phrase is available for you to define the starting position in *identifier-7* to which data is to be moved. Thus, for example, if the phrase, WITH POINTER COUNT, is used, and the value of COUNT is 10, the first character transferred from the sending items is placed in the tenth character position (from the left) of *identifier-7*. Not using the POINTER phrase is equivalent to specifying WITH POINTER 1.

After a character is moved into the data item referenced by *identifier-7*, the pointer value is incremented by 1. Thus, the value of *identifier-8* at the end of a STRING statement is equal to its initial value plus one (1), plus the number of characters transferred.

If the value of *identifier-8* is less than 1, or exceeds the number of character positions in the data item referenced by *identifier-7*, and execution of the STRING statement is not complete, an overflow condition occurs. At this point, regardless of whether or not any data has already been moved to the data item referenced by *identifier-7*, no more data is moved.

Furthermore, if the ON OVERFLOW statement is specified in the STRING statement, the *imperative-statement* in the phrase is executed.

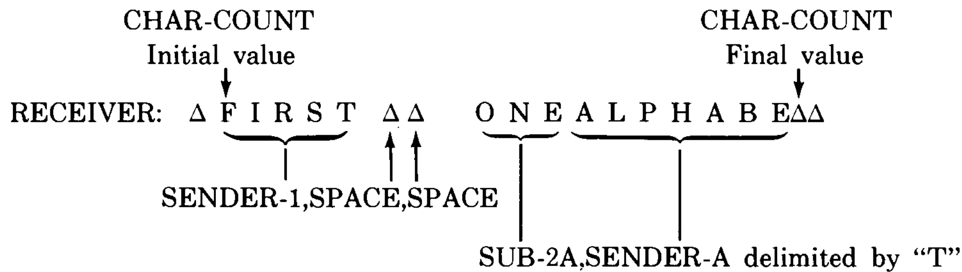
If the ON OVERFLOW phrase is not specified when an overflow condition is encountered, control is transferred to the next executable statement following the STRING statement.

At the end of execution of the STRING statement, only the portion of the data item referenced by *identifier-7* that was referenced during the execution of the STRING statement is changed. All other portions of the data item contain the data that was present before this execution of the STRING statement.

To illustrate the STRING statement:

```
    .
    .
    .
WORKING-STORAGE SECTION.
01 RECEIVER          PIC X(20)  VALUE SPACES.
01 SENDER-1          PIC X(5)   VALUE 'FIRST'.
01 SENDER-2.
   02  SUB-2A        PIC A(3)   VALUE 'ONE'.
   02  SUB-2B        PIC 99V99  VALUE ZERO.
01 SENDER-A          PIC X(15)  VALUE 'ALPHABETICALLY'.
77 CHAR-COUNT        PIC 99     VALUE 1.
77 LIMITER           PIC X      VALUE "T".
    .
    .
    .
PROCEDURE DIVISION.
    .
    .
    .
     ADD 1 TO CHAR-COUNT.
     STRING SENDER-1, SPACE, SPACE DELIMITED BY SIZE,
         SUB-2A, SENDER-A DELIMITED BY LIMITER
         INTO RECEIVER WITH POINTER CHAR-COUNT;
         ON OVERFLOW DISPLAY "OVERFLOW IN RECEIVER",
         " VALUE OF COUNTER IS  ", CHAR-COUNT.
    .
    .
    .
```

With the definitions of data names as described in the Working Storage section, and with CHAR-COUNT set to 2, the STRING statement fills RECEIVER as follows:



If the three statements below are used instead, an overflow condition is caused.

```
MOVE SPACES TO RECEIVER.
MOVE 10 TO CHAR-COUNT
STRING SENDER-A DELIMITED BY SIZE;
      INTO RECEIVER WITH POINTER CHAR-COUNT;
      ON OVERFLOW DISPLAY "OVERFLOW IN RECEIVER";
                  DISPLAY "VALUE OF COUNTER IS ", CHAR-COUNT.
```

The STRING statement now fills RECEIVER as follows:

```
                    CHAR-COUNT          CHAR-COUNT
                   Initial value      value at overflow
                         ↓                   ↓
RECEIVER:  ΔΔΔΔΔΔΔΔΔ  A L P H A B E T I C A
```

When this overflow occurs, the message,

```
OVERFLOW IN RECEIVER
VALUE OF COUNTER IS 21
```

is sent to the terminal from which the program was initiated.

SUBTRACT STATEMENT

The SUBTRACT statement subtracts one or more numeric data items from one or more numeric data items and stores the result in one or more data items.

The SUBTRACT statement has three formats:

Format 1

```
SUBTRACT { identifier-1 } [ , identifier-2 ] ... FROM identifier-m [ ROUNDED ]
         { literal-1    } [ , literal-2    ]

    [ , identifier-n [ ROUNDED ] ] ...  [ ; ON SIZE ERROR imperative-statement ]
```

Format 2

```
SUBTRACT { identifier-1 } [ , identifier-2 ] ... FROM { identifier-m }
         { literal-1    } [ , literal-2    ]          { literal-m    }

    GIVING identifier-n [ ROUNDED ] [ , identifier-o [ ROUNDED ] ] ...

    [ ; ON SIZE ERROR imperative-statement ]
```

Format 3

```
SUBTRACT { CORRESPONDING } identifier-1  FROM identifier-2 [ ROUNDED ]
         { CORR          }

    [ ; ON SIZE ERROR imperative-statement ]
```

Where

identifier-1, identifier-2, and so forth	are each elementary numeric data items, or, if to the right of the dey word, GIVING, may be elementary numeric edited data items. The exception is in format 3, where identifier-1 and identifier-2 must be group items.
literal-1, literal-2, and so forth	are each numeric literals.
CORR	is an abbreviation for CORRESPONDING.

The compiler always insures that enough places are carried in order to avoid losing significant digits.

The ROUNDED, SIZE ERROR, and CORRESPONDING phrases, as well as rules applying to multiple results and overlapping operands are discussed in section X.

The composite of operands, a hypothetical data item obtained by the superimposition of data items, must not exceed 18 digits. This composite is determined for each format as follows:

Format 1:	Composite is determined by using all of the operands in a given statement.
Format 2:	Composite is determined using all of the operands in a given statement except those following the GIVING phrase.
Format 3:	Composite is determined using each pair of corresponding data items separately.

When you use format 1 of the SUBTRACT statement, all literals or values of identifiers preceding the FROM phrase are added together, and the resulting sum is subtracted from the value of each identifier specified in the FROM phrase. As each subtraction is completed, the result is stored in the identifier of the FROM phrase used as operand.

When you use format 2, all literals of values of identifiers are added together, and then subtracted from *literal-m* or the value of *identifier-m.* The result of this subtraction is then stored in each identifier following the GIVING keyword.

When you use format 3 of the SUBTRACT statement, data items in *identifier-1* are subtracted from and stored in corresponding data items of *identifier-2.*

Examples of SUBTRACT statements:

```
          .
          .
          .
SUBTRACT FIRST-YR,SECOND-YR FROM THIRD-YR.
          .
          .
          .
SUBTRACT HIRE-DATE FROM AGE GIVING YEARS-OF-SERVICE.
          .
          .
          .
```

In the first example above, the value of FIRST-YR is added to the value of SECOND-YR, and this sum is subtracted from, and stored in of THIRD-YR.

In the second example, the value of AGE is subtracted from the value of HIRE-DATE, and the results are stored in YEARS-OF- SERVICE.

In the example below, QTY-1, QTY-2, and QTY-3 of PARTS-OUT are subtracted from QTY-1, QTY-2, and QTY-3 of CURRENT-PARTS, and the results are stored in QTY-1, QTY-2, and QTY-3 of CURRENT- PARTS.

```
          .
          .
          .
FILE SECTION.
FD INVFILE.
01 PARTS-INV.
   02  PARTS-OUT.
       03  PARTS-OUT.
           04  PARTS-NUM-1              PIC X(10).
           04  QUANTITY                 PIC 9(6).
           04  SUB-PARTS-OUT.
               05  QTY-1                PIC 9(6).
               05  QTY-2                PIC 9(6).
               05  QTY-3                PIC 9(6).
        03  CURRENT-PARTS.
           04  PART-NUM-1               PIC X(10).
           04  QUANTITY                 PIC 9(6).
           04  CURRENT-SUB-PARTS.
               05  QTY-1                PIC 9(5).
               05  QTY-2                PIC 9(5).
               05  QTY-3                PIC 9(5).
          .
          .
          .
PROCEDURE DIVISION.
          .
          .
          .
     SUBTRACT CORRESPONDING SUB-PARTS-OUT FROM
        CURRENT-SUB-PARTS.
```

UN-EXCLUSIVE STATEMENT

The UN-EXCLUSIVE statement releases a file which has been previously locked by the EXCLUSIVE statement.

UN-EXCLUSIVE Statement Format

UN-EXCLUSIVE ***file-name***

Where *file-name* is the name of a file which has been locked using the EXCLUSIVE statement.

If you use the EXCLUSIVE statement to lock a file, it is not necessary to unlock the file before closing it. An implicit UN-EXCLUSIVE statement is performed when you close the file. Note, however, that if a user has issued an unconditional EXCLUSIVE statement naming the file which you have locked, his program will suspend execution until the file is available to be locked. Thus, you should use the UN-EXCLUSIVE statement to unlock the file as soon as your program has finished accessing it.

The UN-EXCLUSIVE statement calls the MPE intrinsic, FUNLOCK. The condition code returned by the FUNLOCK intrinsic is reflected in the FILE STATUS data item, if any, of the file named in the UNEXCLUSIVE statement. Thus, you can check to insure that the file has been unlocked. If the UN-EXCLUSIVE statement was successful, the contents of the FILE STATUS data item are the characters, "10". If it was not successful, another pair of numerals will be stored in the FILE STATUS data item. The meanings of these numerals are listed in the MPE Intrinsics Manual, part number 30000-90010, under the heading, INTRINSIC CALL ERRORS, in section I.

A USE procedure must be specified for the file being unlocked.

UNSTRING STATEMENT

The UNSTRING statement divides data in a sending field, and places the segments of the data into multiple receiving fields.

UNSTRING Statement Format

```
UNSTRING identifier-1

 [ DELIMITED BY [ ALL ] { identifier-2 } [ , OR [ ALL ] { identifier-3 } ] ... ]
                        { literal-1    }                 { literal-2    }

   INTO identifier-4 [ , DELIMITER IN identifier-5 ] [ , COUNT IN identifier-6 ]

     [ , identifier-7 [ , DELIMITER IN identifier-8 ]
       [ , COUNT IN identifier-9 ] ] ...

 [ WITH POINTER identifier-10 ] [ TALLYING IN identifier-11 ]

 [ ; ON OVERFLOW imperative-statement ]
```

Where

literal-1, ***literal-2,*** and so forth	are all nonnumeric literals, and may be any figurative constants without the optional word, ALL.
identifier-1, ***identifier-2,*** ***identifier-3,*** ***identifier-5,*** and ***identifier-8***	must be described implicitly or explicitly as alphanumeric data items.
identifier-4 and ***identifier-7***	may be described as alphabetic, alphanumeric, or numeric data items. However, the B symbol may not be used in an alphabetic description, and the P symbol may not be used in a numeric description for these items. Note: Edited receiving fields are not permitted.
identifier-6, ***identifier-9,*** ***identifier-10,*** and ***identifier-11***	must be described as elementary numeric integer data items. The P symbol may not be used in their descriptions.

No identifier may name a level 88 entry.

If the DELIMITED BY phrase is not specified, the DELIMITER IN and COUNT IN phrases must not be used.

To facilitate the discussion which follows, we rewrite the format as:

```
UNSTRING sending-item

    [ DELIMITED BY [ ALL ] delimiter-1 [ OR [ ALL ] delimiter-2 ]  ...  ]

    INTO receiver-1 [ , DELIMITER IN  delim-receiver-1 ]
                    [ COUNT IN count-receiver-1 ]  ...

         receiver-2 [ , DELIMITER IN  delim-receiver-2 ]
                    [ COUNT IN count-receiver-2 ]  ...

    [ WITH POINTER identifier-10 ] [ TALLYING IN identifier-11 ]

    [ ; ON OVERFLOW  imperative-statement ]
```

In the first format, *identifier-1* represents the sending item. In the rewritten format above, this item is denoted by the words, *sending-item.*

Identifier-4, identifier-7, and any other recursions of equivalent identifiers represent the receiving data items. In the rewritten format above, these are represented by *receiver-1* and *receiver-2.*

Identifier-2 or its associated literal, and *identifier-3* or its associated literal represent delimiters on the sending item. They are represented by *delimiter-1* and *delimiter-2* above. If a figurative constant is used as a delimiter, it stands for a single character nonnumeric literal.

A delimiter may be any character available in the ASCII collating sequences. Also, when a delimiter contains two or more characters, all of the characters must be in contiguous positions of the *sending-item*, and be in the order specified to be recognized as a delimiter.

When more than one delimiter is specified, each delimiter is compared to the *sending-item* in turn. If a match occurs in one of these comparisons, examination of the sending field ceases.

Note that delimiters may not overlap. That is, no character or characters in the *sending-item* can be considered part of more than one delimiter.

When the ALL keyword is used in the DELIMITED phrase, and an associated delimiter is encountered while examining the *sendingitem*, each contiguous occurrence of the delimiter, beginning at the point where the delimiter first occurs, is considered as part of that delimiter. If the DELIMITER IN phrase has been specified, the entire string of contiguous delimiters is moved to the appropriate *delim-receiver*.

After a single delimiter (or a string of delimiters as described in the preceding paragraph) has been found, if the next character or set of characters is a delimiter, the current receiving item is space or zero filled, depending upon how the receiving item is described. If the DELIMITER IN phrase is specified for that particular receiving item, the delimiter is then moved to the corresponding *delim-receiver*.

Identifier-5 and *identifier-8*, represented respectively by *delimreceiver-1* and *delim-receiver-2* above, name receiving items for the delimiters specified by *delimiter-1* and *delimiter-2*.

Identifier-6 and *identifier-7*, represented by *count-receiver-1* and *count-receiver-2* respectively in the above format, are used to hold the count of the number of characters of the sending item moved to *receiver-1* and *receiver-2*, respectively. These values do not include the counts of the delimiter character or characters.

The data item represented by *identifier-10* contains an integer used to indicate the first character, counting from the left most character of the sending item, to be examined. You are responsible for setting the initial value of this item. If it is less than 1, or is greater than the number of characters in the sending item when the UNSTRING statement is initiated, an overflow condition exists.

When an UNSTRING statement completes execution, if you specified the POINTER phrase, the value of *identifier-10* is equal to the initial value plus the number of characters examined in the data item referenced by *identifier-1*.

The data item referenced by *identifier-11* is a counter that records the number of receiving items acted upon during the execution of an UNSTRING statement. As with *identifier-10*, you must initialize the value of *identifier-11*. When the UNSTRING statement completes execution, if you have specified the TALLYING phrase, the value of *identifier-11* is the initial value of *identifier-11* plus the number of data receiving items acted upon.

Execution of the UNSTRING Statement

When the UNSTRING statement is initiated, the current receiving area is *receiver-1.*

If the POINTER phrase is specified, the *sending-item* is examined beginning with the character position indicated by the contents of the data item referenced by *identifier-10.* If this phrase is not used, examination begins with the left-most character of the *sending-item.*

If the DELIMITED BY phrase is specified, the examination proceeds left to right until either a delimiter is found, or no delimiters are found, and the last character of the *sending-item* is examined.

If the DELIMITED BY phrase is not specified, the number of characters in *receiver-1* is used to determine how many characters of the *sending-item* are to be examined. That is, the number of characters examined is equal to the size of *receiver-1.* Note, however, that if *receiver-1* is a numeric data item described with the SIGN IS SEPARATE clause, the number of characters examined is one less than the size of *receiver-1.*

When examination is complete, the examined characters, excluding any delimiters encountered, are treated as an elementary alphanumeric data item, and are moved into *receiver-1* according to the rules for an alphanumeric MOVE (see the MOVE statement).

If the DELIMITER IN phrase is specified for *receiver-1,* and a delimiter was encountered, the character or characters making up the delimiter are treated as an elementary alphanumeric data item, and are moved into *delim-receiver-1* according to the rules of the MOVE statement. If the examination of the sending item ceased for a reason other than that a delimiter was encountered, *delim-receiver-1* is filled with spaces.

If the COUNT IN phrase is specified for *receiver-1,* a value equal to the number of characters examined, excluding any delimiter characters, is moved to *count-receiver-1* according to the rules for an elementary move. This completes the initial phase of execution of the UNSTRING statement.

If all characters of the *sending item* (beginning from the position specified by *identifier-10* if the POINTER phrase is specified) have been examined, the UNSTRING statement is complete, and control passes to the next executable statement.

If all characters have not been used, and *receiver-2* is specified, examination of the *sending-item* begins again. This second examination begins with the character immediately to the right of the delimiter (if any) which caused termination of the initial examination. If no delimiter was specified, meaning that examination ceased because the number of characters in *receiver-1* had been examined, examination begins with the character immediately to the right of the last character transferred.

This second phase of the UNSTRING statement is executed in the same way as for *receiver-1* (except that examination does not, of course, begin with the character position specified by the value of *identifier-10*).

The data items affected are *receiver-2, delim-receiver-2,* and *count-receiver-2.*

A new phase of examination and transfer is executed for each receiving item specified, or until all characters in the sending item have been examined.

The contents of the data item referenced by *identifier-10* are incremented by 1 for each character examined in the sending-item.

Overflow Conditions

An overflow condition is caused by one of two situations.

The first, described under DESCRIPTION OF PARAMETERS above, is caused by an invalid value for the data item represented by identifier-10.

The second situation is when all receiving items have been acted upon, but there remain unexamined characters in the sending-item.

When an overflow condition occurs, execution of the UNSTRING condition ceases.

If the ON OVERFLOW phrase is specified, and an overflow condition occurs, the imperative-statement in the ON OVERFLOW phrase is executed.

If the ON OVERFLOW phrase is not specified, control is passed to the next executable statement following the UNSTRING statement.

Subscripting or Indexing of IDENTIFIERS

Any subscripting or indexing associated with *identifier-1, identifier-10,* or *identifier-11* is evaluated only once, immediately before any data is transferred as the result of the initial phase of the UNSTRING statement.

Any subscripting or indexing associated with any other identifier is evaluated immediately before data is transferred to a receiver. This occurs for each phase of execution of the UNSTRING statement.

Example of the UNSTRING statement:

```
01 ID-INFO                     PIC X(35).
     .
     .
     .
01 EMPLOYEE-TABLE.
   02  EMPLOYEE-STATS OCCURS 30 TIMES.
       03  NAME                PIC X(40).
       03  BIRTH-DATE          PIC XX/XX/XX.
       03  HAIR-COLOR          PIC X(12).
       03  EYE-COLOR           PIC X(12).
       03  HEIGHT              PIC X/X.
01 SUBSCRIPTOR                 PIC XX  VALUE "1".
01 SUBSCRIPT                   PIC 99  VALUE 1.
01 INCREMENT                   PIC XX  VALUE "2".
     .
     .
     .
UNSTRING ID-INFO
    DELIMITED BY "," OR INCREMENT
INTO NAME (SUBSCRIPT), BIRTH-DATE (SUBSCRIPT)
,HAIR-COLOR (SUBSCRIPT), EYE-COLOR (SUBSCRIPT),
HEIGHT (SUBSCRIPT) DELIMITER IN SUBSCRIPTOR
WITH POINTER CHARS TALLYING COMPLETE-INFO;
ON OVERFLOW PERFORM FIND-CAUSE.
```

If ID-INFO is, in standard data format,

WILSON JAMES, 030250,BLONDE,BLUE,592

and the initial value of CHARS is 1, and of COMPLETE-INFO is 0, then when the UNSTRING statement above is executed it goes through the phases described below.

PHASE 1:

```
  WILSON JAMES,
  ↑           ↑
Initial pointer    delimiter found
```

Move "WILSON JAMES" into NAME(1) filling in spaces to the left of the rightmost character. Increment the value of CHARS by 13, giving 14.

PHASE 2:

```
WILSON JAMES,030250,
            ↑      ↑
  new pointer      delimiter found
```

Move 030250 into BIRTH-DATE(1). Increment the value of CHARS by 7, giving 21.

PHASE 3:

```
WILSON JAMES,030250,BLONDE,
                   ↑      ↑
         new pointer      delimiter found
```

Move "BLONDE" into HAIR-COLOR(1), filling in spaces to the left of the right-most character. Increment the value of CHARS by 7, giving 28.

PHASE 4:

```
WILSON JAMES,030250,BLONDE,BLUE,
                          ↑    ↑
                new pointer    delimiter found
```

Move "BLUE" into EYE-COLOR(1).

Increment the value of CHARS by 5, giving 33.

PHASE 5:

```
WILSON JAMES,030250,BLONDE,BLUE,592
                               ↑
                     new pointer  first delimiter
                                   not found.

WILSON JAMES,030250,BLONDE,BLUE,592
                               ↑ ↑
                     new pointer second delimiter
                                 found.
```

Move "59" to HEIGHT(1), and "2" to SUBSCRIPTOR.
Increment the value of CHARS by 3, giving 36.

Since all receiving items have been used, this completes the execution of the UNSTRING statement. The value of CHARS is 35, and the value of COMPLETE-INFO is 5, since five receiving items were acted upon.

USE STATEMENT

The USE statement specifies procedures for input-output error handling and for user label processing. These error handling and label processing procedures are in addition to the standard procedures provided by the input-output control system.

There are two general formats of the USE statement:

Format 1 - Error Handling Procedures

```
                                { EXCEPTION }
USE AFTER STANDARD              { ERROR     }   PROCEDURE

          { file-name-1 [ , file-name-2, ... ] }
          { INPUT                              }
ON        { OUTPUT                             }  .
          { I-O                                }
          { EXTEND                             }
```

Format 2 - User Label Procedures

```
USE AFTER STANDARD BEGINNING [ FILE ]

                              { file-name-1 [ , file-name-2 ] ... }
                              { INPUT                             }
    LABEL PROCEDURE ON        { OUTPUT                            }  .
                              { I-O                               }
                              { EXTEND                            }
```

Where

file-name-1, file-name-2, and so forth	are the names of files to be acted upon byh the appropriate procedures when an input-output error has occurred, or a user-label is to be processed. These names must not name sort-merge files.
ERROR and **EXCEPTION**	are synonymous, and may be used interchangeably.

General Rules

The rules below apply to both formats of the USE statement.

A USE statement, when specified, must immediately follow a section header in the declaratives section, and must be followed by a period and a space. The remainder of the section must consist of zero, one, or more procedural paragraphs that define the procedures to be used. These paragraphs make up the procedures which are executed when required. The USE statement itself is never executed. It merely defines the conditions calling for the execution of the USE procedures.

Within a USE procedure, there must not be any reference to any nondeclarative procedures. Conversely, in the nondeclarative portion there must not be any reference to procedure-names in the declarative portion, except that PERFORM statements may refer to a USE statement or to the procedures associated with such a USE statement.

Within a USE procedure, no statement can be executed which would cause the execution of a USE procedure that has been previously invoked, but has not yet returned control to the invoking routine.

In a USE statement, the files affected by the procedures related to the USE statement may be explicitly or implicitly specified.

They are explicitly specified by using their names in the USE statement itself.

To implicitly specify a file, the words, INPUT, OUTPUT, I-O, and EXTEND are used. For example, if a USE statement uses the word, INPUT, then any file opened for input, as opposed to output or input-output, is implicitly specified in that USE statement. Note that the word, EXTEND, may be used only for files whose organization is sequential.

The same file-name may appear in different USE statements. However, the appearance of a file-name in a USE statement may not cause the simultaneous request for execution of more than one USE procedure.

This applies to implicit as well as explicit references to a file. Thus, for example, a file open for input-output operations, and specified by name in a USE statement, will cause a compile-time error if a USE statement implicitly references the file through the use of the I-O keyword.

Format 1 USE Statement

The first format of the USE statement is for error handling procedures.

The files implicitly or explicitly referenced in a format-1 USE statement need not have the same organization or access.

When a format 1 USE statement is specified for a file, and an input or output error occurs, the input-output system performs any applicable USE procedures either after completing the standard input-output error routine, or upon recognition of the INVALID KEY or AT END condition.

When an INVALID KEY or AT END condition occurs, the appropriate USE procedures are executed provided only that an AT END or INVALID KEY phrase has not been specified in the input-output statement which generated the error.

Format 2 USE Statement

The format 2 USE statement is for reading and writing user labels on a file, starting immediately after the MPE system label at the beginning of a file.

There may be as many as eight user labels following an MPE system label, and each label may consist of 80 characters.

When user labels are read by your program, only one location is made available for them in memory. Thus, only one may be read at a time, requiring only a single description of a label record data item per file.

Whether a file is explicitly or implicitly referenced in a format 2 USE statement, the format 2 USE statement does not apply, and is ignored, if the referenced file includes the LABEL RECORDS ARE OMITTED clause in its description.

The transfer of control to format 2 USE procedures occurs when a file is opened, as follows:

- INPUT, I-O, and EXTEND - control is passed to the appropriate USE procedure after a beginning input label check procedure is executed.
- OUTPUT - control is transferred after a beginning output label is created, but before it is written.

Using the INPUT keyword in a format 2 USE statement allows you to read labels, while using I-O allows you to both read and write them. The use of the OUTPUT or EXTEND keywords only allows you to write user labels.

Using a file-name in a format 2 USE statement allows you to either read or write, or both, depending upon how the file is opened. Thus, for example, a file opened in I-O mode allows you to read and write user labels.

Note that common label items used in a format 2 USE procedure need not be qualified by file name when the USE statement for the procedure specifies the INPUT, I-O, OUTPUT, or EXTEND keywords rather than a file-name.

A common label item is an elementary data item that appears in every label record of the program, but does not appear in any data record of the program. It must have the same name, description, and relative position in every label record.

Within a format 2 USE procedure, no statements are explicitly written in order to read or write a user label.

For an input or input-output file, a corresponding USE procedure automatically reads the user label into the label record of the file. The USE procedure may then be written to check that the label has the form and content desired.

All format 2 USE procedures have an exit mechanism appended to them by the compiler. This exit mechanism follows the last statement of the procedure, and is used to write user labels out to the appropriate file.

Thus, a format 2 USE procedure for a file opened in OUTPUT, I-O, or EXTEND mode may use the label record data item to define the contents of a label. When the exit mechanism is reached, the label is automatically written to the file.

With a single exception, all logical paths within a declarative procedure must lead to this exit point, thus terminating the procedure. This implies that with one execution of a format 2 USE procedure, only a single user label may be read or written.

The purpose of the exception mentioned above is to allow you to read or write more than one user label (up to eight). The exception is the use of the GO TO MORE-LABELS statement within a format 2 USE procedure.

The function of this statement varies according to how the file was opened:

- 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 and EXTEND 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.

WRITE STATEMENT

The WRITE statement releases a logical record. To use the WRITE statement for a sequential file, the file must be opened in the OUTPUT or EXTEND mode; to use the WRITE statement with an indexed, relative, or random file, the file must be opened in OUTPUT or I-O mode.

For sequential files, the WRITE statement may additionally be used for vertical positioning of lines within a logical page (see the LINAGE clause in the File Description Entry of the Data Division).

There are two formats for the WRITE statement:

Format 1 Sequential Files

```
WRITE record-name [ FROM identifier-1 ]

[ { BEFORE }              { { identifier-2 }  [ LINE  ] } ]
[ { AFTER  }  ADVANCING   { { integer      }  [ LINES ] } ]
[                         {                             } ]
[                         { { mnemonic-name }           } ]
[                         { { PAGE          }           } ]

[ ; AT  END-OF-PAGE     imperative-statement ]
[       EOP                                  ]
```

Format 2 Relative, Indexed, or Random-Access Files

WRITE ***record-name*** [**FROM** ***identifier-1***]

[; **INVALID KEY** ***imperative-statement***]

Where

record-name	is the name of a logical record in the File Section of the Data Division. It may be qualified. *Record-name* must not reference the same storage area as *identifier-1*. Additionally for random-access files, *record-name* must not be part of a SORT file.
identifier-1	is the name of a data item described within the Data Division. It must not reference the same storage area as *record-name*.
identifier-2	is the name of an elementary integer data item. The value of the data item mustbe greater than or equal to 0.
integer	is a non-negative integer.
mnemonic-name	is a name related to the functions, TOP, NOSPACE CONTROL, and C01 through C12. This relation is provided by the SPECIAL NAMES clause to the Configuration Section of the Environment Division.
END-OF-PAGE and **EOP**	are equivalent.
imperative-statement	is one or more imperative statements.

General Rules

The rules discussed below apply to both formats of the WRITE statement.

The successful execution of a WRITE statement releases a logical record to the operating system. The number of character positions on a mass storage device required to store a logical record may or may not be equal to the number of character positions defined by the logical description of that record in the program.

If the file's records are longer than the data being written to it, the file is padded with blanks, or with zeroes, depending upon whether the file is an ASCII or binary file respectively.

If the file's records are shorter than the data being written to it, the data being written is truncated.

Whether execution of the WRITE statement was successful or not, the FILE STATUS data item, if any, is updated following the execution of the WRITE statement. See section VII, the FILE STATUS CLAUSE, for valid combinations of status keys 1 and 2.

The logical record released by the execution of the WRITE statement is no longer available in memory unless the associated file is named in a SAME RECORD AREA clause, or the execution of the WRITE statement was unsuccessful because of a boundary violation (for sequential files) or an INVALID KEY condition (for relative, indexed or random-access files).

The logical record is also available to the program as a record of other files referenced in that SAME RECORD AREA clause.

From Phrase

The results of executing a WRITE statement using the FROM phrase is equivalent to executing the statement,

MOVE *identifier-1* TO *record-name*

and then executing the same WRITE statement without the FROM phrase.

Unlike the record area for record-name, the data in *identifier1* always remains in memory and is available after execution of the WRITE statement, regardless of whether a SAME RECORD AREA clause was used for any file in which *identifier-1* names a data-item.

Note that the maximum record size of a file is established at the time the file is created, and must not subsequently be changed.

Format 1 WRITE Statement

A format 1 WRITE statement, used for sequential files only, allows you to use vertical positioning of a line within a logical page.

This is done through the ADVANCING and END-OF-PAGE phrases.

Either or both of these phrases may be used in a format 1 WRITE statement. However, if the END-OF-PAGE phrase is used, the LINAGE clause must appear in the file description entry for the associated file. Also, if the LINAGE phrase is present in the file's description, and the ADVANCING phrase is used, it cannot be used in the form ADVANCING *mnemonic-name*.

If neither phrase is used, automatic advancing equivalent to AFTER ADVANCING 1 LINE is provided.

Whenever the execution of a given format 1 WRITE statement cannot be fully accommodated within the current page body, an automatic page overflow condition occurs. If neither the ADVANCING nor the END-OF-PAGE phrase is specified, and a page overflow condition occurs, the WRITE statement uses an implicit AFTER ADVANCING PAGE to position the data on the next logical page.

ADVANCING PHRASE

When the ADVANCING phrase is used in a format 1 WRITE statement, the line to be written is presented to the page either before or after the representation of the page is advanced. Whether it is presented before or after advancing the logical page is determined by the use of the BEFORE or AFTER keyword respectively.

The amount of advancement of the logical page is determined by *integer, identifier-2,* PAGE, and *mnemonic-name* as follows:

- *Integer* causes the representation of the logical page to be advanced the number of lines equal to the value of integer.
- *Identifier-2* causes the representation of the logical page to be advanced the number of lines equal to the current value of the data item represented by *identifier-2.*
- PAGE causes the logical page to be advanced to the next logical page. If the record to be written is associated with a file whose description includes a LINAGE clause, the repositioning is to the first line that can be written on the new logical page as specified by the LINAGE clause. If the record to be written is associated with a file whose description does not contain a LINAGE clause, the repositioning is to the first line of the next logical page.

If PAGE is specified for a device to which it has no meaning, advancing is provided which is equivalent to ADVANCING 1 LINE.

- If *mnemonic-name* is specified, the file receiving the record must not contain a LINAGE clause in its description, and must be a line printer device file. *Mnemonic-name* can be equivalent to TOP, in which case, it is equivalent to specifying PAGE for a file whose description does not contain a LINAGE clause.

Mnemonic-name may also be equivalent to one of C01 to C12, or NOSPACE CONTROL.

If it is NOSPACE CONTROL, this is equivalent to ADVANCING 1 LINE.

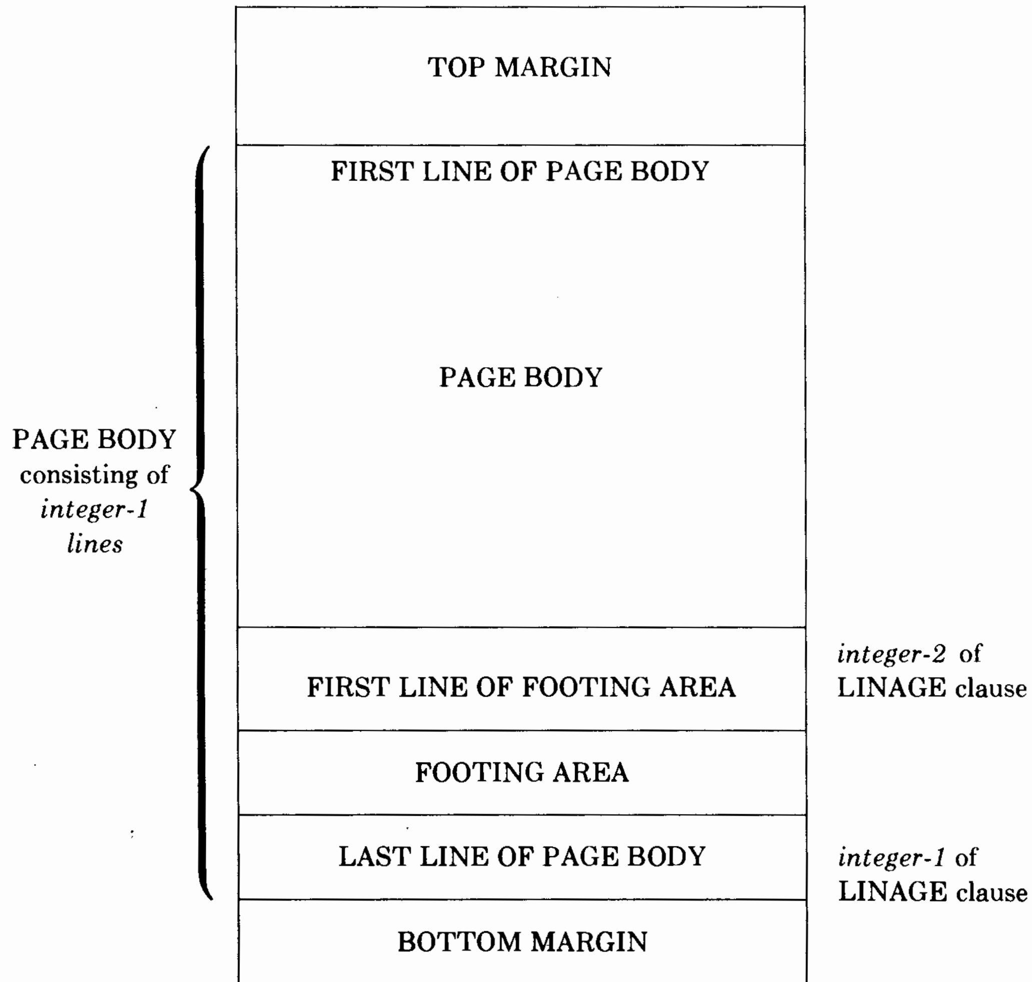
The C01 through C12 options are related to the VFU (Vertical Form Unit) holes punched into a paper tape of a line printer. For an HP lineprinter, the meanings of C01 through C12 are related to the FWRITE intrinsic, and have the following meanings:

C01: Page eject (skip to top of next page).
C02: Skip to the bottom of the form.
C03: Single spacing with automatic page eject.
C04: Single space on the next odd-numbered line with automatic page eject.
C05: Triple space with automatic page eject.
C06: Space a half page with automatic page eject.
C07: Space one quarter of a page with automatic page eject.
C08: Space one sixth of a page with automatic page eject.
C09: Space to the bottom of the form.
C10: User option.
C11: User option.
C12: User option.

END-OF PAGE Phrase

The END-OF-PAGE phrase can be used only in conjunction with the LINAGE clause of a sequential file description entry.

To clarify the following discussion the concept of a logical page is illustrated below:



Two conditions may occur which cause the execution of the END-OF-PAGE phrase.

The first occurs when a footing area has been defined using *integer-2* or *data-name-2* of the LINAGE CLAUSE. In this case, when a WRITE statement using the END-OF-PAGE phrase is executed, and this execution causes printing or spacing within the footing area, an end-of-page condition occurs. This is controlled by the value of the LINAGE-COUNTER for the associated file. Thus, when LINAGE-COUNTER equals or exceeds the value of *integer-2* or the data item specified by *data-name-2*, an end-of-page condition occurs, the data is written into the footing area, and the *imperative-statement* of the END-OF-PAGE phrase is executed. Note that an end-of-page condition does not automatically cause the next line of data to be written on the next logical page; it is your responsibility to control this using the LINAGE-COUNTER of the file, the ADVANCING phrase, or the automatic page overflow condition.

The second condition which causes the execution of an END-OF-PAGE phrase is an automatic page overflow. An automatic page overflow occurs when the LINAGE-COUNTER for the file associated with the WRITE statement exceeds *integer-1* or the data item referenced by *data-name-1*. In this case, if an ADVANCING phrase using the BEFORE keyword is present in the WRITE statement, the record is written the specified number of lines below the end of the page body, and the device used to contain the logical pages is repositioned to the first line that can be written on the next logical page. If an ADVANCING phrase is specified (implicitly or explicitly) which uses the AFTER keyword, the device used to contain the logical pages is repositioned to the first line that can be written on the next logical page, and the record is written.

No matter whether ADVANCING BEFORE or ADVANCING AFTER is specified, when the record has been written, control is transferred to the imperative-statement of the END-OF-PAGE clause.

Note that if *integer-2* or the data item referenced by *data-name-2* of the LINAGE clause is not specified (thus, no footing area has been defined), or if *integer-2* or the data item referenced by *data-name-2* is equal to *integer-1*or the data item referenced by *data-name-1* of the LINAGE clause, no end-of-page condition distinct from a page overflow is detected. Thus, an end-of-page condition is, in this case, equivalent to a page overflow condition.

BOUNDS OVERFLOW

When an attempt is made to write beyond the boundaries of a sequential file, an exception condition exists.

The contents of the record area specified by *record-name* are unaffected by such a condition, and the following actions take place:

a. The FILE STATUS data item, if any, of the associated file is set to indicate a boundary violation.

b. If a USE AFTER STANDARD EXCEPTION declarative is specified (explicitly or implicitly) for the file, the associated procedure is executed. If no USE statement is specified for the file, the program aborts, supplying a file error message.

MULTIPLE REEL/UNIT FILES

After an end-of-reel condition has been recognized for a multiple reel labeled tapes, the WRITE statement performs the standard ending reel or unit procedure, requests a reel or unit swap, and then performs the standard reel or unit label procedure. The record is then written according to the specifications of the WRITE statement.

Format 2 WRITE Statement

A format 2 WRITE statement can be used for random-access, relative, or indexed files.

In a format 2 WRITE statement, if no applicable USE statement has been issued for the referenced file, the INVALID KEY phrase of the WRITE statement must be used.

Also, the INVALID KEY phrase must always be used when a format 2 WRITE statement is issued for a random-access file.

RANDOM-ACCESS FILES

When a format 2 WRITE statement is issued for a random-access file, the contents of the ACTUAL KEY data item associated with the file are used in an implicit SEEK statement to find the record into which the data specified by *record-name* is to be written. If the address specified by the ACTUAL KEY data item is invalid, an invalid key condition exists, no data is written, the data in the record area is unaffected, and the *imperative statement* in the INVALID KEY phrase is executed.

The address can be invalid for one of three reasons, as follows:

A. It contains a negative value.

B. It is greater than the highest possible relative record number in the file.

C. The value moved to the ACTUAL KEY data item contains more than nine digits.

RELATIVE FILES

When a format 2 WRITE statement is issued for a relative file, and the file is open for output in sequential mode, the first execution of a WRITE statement for the file releases a record to that file, assigning a relative record number of 1 to it. Subsequent WRITE statements assign relative record numbers of 2, 3, 4, and so on as records are released to the file. If the RELATIVE KEY data item has been specified in the file control entry for the associated file, it is updated during each execution of the WRITE statement to indicate the relative record number of the record being written.

If a relative file is open in random or dynamic access mode, regardless of whether it is open for output only or input-output operations, your program must set the value of the relative key data item to specify where the record is to be placed in the file.

Provided that the relative key data item is a valid relative key, the data in the record area is released to the file.

INVALID KEY CONDITIONS FOR A RELATIVE FILE

An INVALID KEY condition exists for a relative file when an attempt is made to write beyond the externally defined boundaries of the file, or when the file is in dynamic or random access mode, and the RELATIVE KEY data item specifies a record which already exists in the file.

When this condition occurs, execution of the WRITE statement is unsuccessful, the contents of the record area are unaffected, and the FILE STATUS data item, if any, is updated to indicate the cause of the condition.

If the INVALID KEY phrase was specified in the WRITE statement, control passes to the *imperative-statement* appearing in the phrase. If no INVALID KEY phrase was specified, then a USE procedure must have been specified (either implicitly or explicitly), and is executed.

Note that if both a USE procedure and an INVALID KEY phrase are specified, the USE procedure is ignored.

INDEXED FILES

A format 2 WRITE statement for an indexed file uses the primary record key data item to write records to the file; therefore, the value of the primary record key must be unique within the records of the file unless the DUPLICATES phrase has been used in the RECORD KEY clause of the Environment Division.

If alternate record keys have been specified, they must also be unique within the file unless the DUPLICATES phrase has been used in the ALTERNATE RECORD KEY clause of the Environment Division.

If the DUPLICATES phrase is used, then after records containing duplicate keys have been written to the file, if they are later accessed sequentially, they are retrieved in the same order as they were written.

A WRITE statement for an indexed file open in sequential access mode writes records to the file in ascending order of prime record key values.

A WRITE statement for an indexed file in random or dynamic access mode writes a record to the file in whatever order your program specifies.

When a WRITE statement is successfully executed for an indexed file, all keys of the record are used in such a way that subsequent access of the record may be made based upon any of the specified record keys.

INVALID KEY CONDITIONS FOR INDEXED FILES

There are three conditions under which an invalid key condition can occur for an indexed file WRITE statement:

A. When the file is open for output in sequential access mode, and the value of the prime record key is not greater than the value of the prime record key of a previous record.

B. When the file is open for output or input-output operations in any access mode, and the value of the primary or an alternate record key for which duplicates are not allowed equals the value of the corresponding record key of a record already existing in the file.

C. When an attempt is made to write beyond the externally defined boundaries of the file.

In any case, when an invalid key condition occurs, execution of the WRITE statement is unsuccessful, the record area is unaffected, and the FILE STATUS data item, if any, associated with the file is set to a value which indicates the cause of the condition.

If an INVALID KEY phrase was specified in the WRITE statement, control is transferred to the imperative-statement appearing in the phrase, and any USE procedure specified for the file is ignored.

If no INVALID KEY phrase was specified, then a USE procedure must have been specified, implicitly or explicitly, and that procedure is executed.

INTERPROGRAM COMMUNICATION

SECTION XII

There are two types of program modules. The first type, segments, is discussed in Section X.

The second type consists of separately compiled, but logically coordinated programs, which, at execution time, are subdivisions of a single process. This approach to programming lends itself to making a large problem more easily programmed and debugged, by breaking a problem into logical modules and coding each module separately.

In COBOL terminology, a program is either a source or an object program. The distinction between the two is that a *source program* is simply a syntactically correct set of COBOL statements, whereas an *object program* is the set of instructions, constants, and other data resulting from the compilation of a source program.

A run unit is defined as being the total machine language necessary to solve a given generated data processing problem.

One run unit may contain several object programs, some of which may not have been generated by the COBOL compiler.

A run unit, then, is a combination of one main program with, optionally, one or more subprograms. Each subprogram may itself use one or more subprograms.

In COBOL II/3000, a program has the ability to transfer control to one or more subprograms, whether or not the names of the subprograms are known at compile time. Also, it is possible for the compiler to determine the availability of object time memory for a subprogram.

When a run unit contains more than one object program, there must be communication between them. Interprogram communication takes two forms:

- Transfer of control from one object program to another;
- Reference to common data.

Transfer Of Control

The CALL statement is the means used in COBOL programs to pass control from one object program to another, and there are no restrictions on a called program itself calling another object program. Caution should be used, however, to avoid calling a program that preceded, in the calling chain, the program currently having control. Otherwise, results of the run unit are unpredictable.

When control is passed to a called program, execution begins either at the first Procedure Division statement, or at a secondary entry point of the Procedure Division. Secondary entry points are described under the ENTRY statement, an HP extention to ANSI COBOL'74, later in this section. Program execution begins at the point of entry to the called program in the normal, line-by-line sequence, following the same conventions as for COBOL main programs. Termination takes place in a COBOL subprogram under two possible conditions.

The first is when an EXIT PROGRAM or GOBACK statement is encountered. When this occurs, control reverts to the calling program, which begins execution at the line immediately following the CALL statement that originally passed control.

The second condition causing termination in a COBOL program is when a STOP RUN statement is encountered. in this case, the entire run unit is terminated.

In short, the EXIT PROGRAM and GOBACK statements terminate only the program in which they appear, while the STOP RUN statement terminates the entire run unit.

An exception to this is when a GOBACK statement appears in a main program. In this case, it is equivalent to issuing a STOP RUN statement.

Of course, if the called program is not a COBOL program, termination of the run unit, or the return of control to the calling program must be peformed in accordance with the rules of the language in which the called program is written.

Reference To Common Data

Since a called program often accesses data which is also used by the calling program, both programs must have access to the same data items if you wish to pass data to, or return data from, the called program.

In the calling program, it does not matter which section in the Data Division is used to describe the common data.

In the called COBOL program, however, all common data must be described in the Linkage Section of the Data Division under 01 or 77 level description entries. Unlike the data in the calling program, no storage is allocated for Linkage Section items when the calling program is compiled.

Communication between the called COBOL program and the common data items stored in the calling program is provided by the USING clauses in both the calling and the called program.

The USING clause in the calling program is part of the CALL statement. It lists the names of common data items described in the Data Division.

In the called COBOL program, the USING clause appears as part of the Procedure Division header, or of the ENTRY statement. The common data items, which must be described in the Linkage Section, are listed by this clause.

Despite how the data items are described in the calling program, they are processed according to how they are described in the Linkage Section of the called program.

The only restriction is that descriptions of common data items must define an equal number of character positions.

Common data items are related to each other in the calling and called COBOL programs by their positions in the USING clauses, and not by their names.

This implies that you may use entirely different names in the called program to represent common data items of the calling program.

For example, if EMP-INFO is the fourth name in the USING clause of a calling program, and FOURTH-PASSED is the fourth name in the USING clause of the called program, then any reference to FOURTH- PASSED is treated as if it were a reference to the corresponding data item, EMP-INFO, in the calling program.

Also, although you may not use the same name twice within the USING phrase of a called program, you may do so in the USING phrase of the calling program. This allows you to have a data item in the calling program related to more than one data name in the called program.

Types Of Subprograms

In COBOL II/3000, there are two kinds of subprograms. The first type, non-dynamic subprograms, have their data storage declared as OWN (DB-relative). This is the COBOL'74 ANSI standard type of subprograms.

Dynamic subprograms are the second type of COBOL subprograms. These subprograms are an implementation of Hewlett-Packard, and have local (Q-relative) data storage. This type of data storage provides you with the ability to segment your data stack. A dynamic subprogram is always in its initial state when called. This implies that any files opened in dynamic programs should be closed before exiting. Otherwise, an error will occur.

To specify whether a subprogram is to be dynamic or non-dynamic, you can use the DYNAMIC and SUBPROGRAM parameters, respectively, of the $CONTROL subsystem command. This command is discussed in Appendix A.

If you do not use such a $CONTROL command, and the source program contains a LINKAGE section, it is compiled as a non-dynamic subprogram.

Note that a program containing no LINKAGE SECTION is considered a main program unless a SUBPROGRAM or DYNAMIC parameter is used in the $CONTROL command for that program.

When you exit a called non-dynamic program, its state is maintained. Thus, data items not common to the calling program or a program that called the calling program retain values associated with them when the program in which they are used is exited. Dynamic programs are always in their initial states when they are called.

If you wish to re-initialize a non-dynamic subprogram, you may use the CANCEL statement of COBOL. This statement is issued in the calling program after the EXIT PROGRAM or GOBACK statement has been executed in the called program.

CALL STATEMENT

In ANSI COBOL'74 the CALL statement can be used to transfer control from one object program to another within the same run unit. COBOL II/3000 adds the ability to invoke MPE intrinsics from within a given object program.

CALL Statement Format (ANSI COBOL '74)

```
CALL { identifier-1 } [ USING identifier-2 [ , identifier-3 ] ... ]
     { literal-1    }

[ ; ON OVERFLOW imperative-statement ]
```

To this, COBOL II/3000 adds the optional key word, INTRINSIC to specify that an intrinsic, and not a program, is being called. Also added are the characters,\\ , and @. These are used to pass a data item by value, or as a byte-pointer, and can only be used when calling non-COBOL programs.

The last HP extension to the CALL statement is the GIVING identifier-4 phrase. This phrase allows you to name an identifier which will hold the results of a call to a typed procedure.

Note that you can use the COBOL II/3000 CALL statement to lock and unlock files in the same manner as is done by using the EXCLUSIVE and UN-EXCLUSIVE statements. This is done by using the special procedures, COBOLLOCK and COBOLUNLOCK. These procedures are a carry-over from COBOL/3000. They are described in Appendix I of this manual.

CALL Statement Format (COBOL II/3000)

```
                                            [        { \\               } ]
                                            [        { @identifier-2    } ]
CALL [ INTRINSIC ] { identifier-1 }         [ USING  { identifier-2     } ]
                   { literal-1    }         [        { literal-2        } ]
                                            [        { \ identifier-2 \ } ]
                                            [        { \ literal-2 \    } ]

    [ [   { \\               } ]     ]
    [ [   { @identifier-3    } ]     ]
    [ [ , { identifier-3     } ] ... ]  [ GIVING identifier-4 ]
    [ [   { literal-3        } ]     ]
    [ [   { \ identifier-3 \ } ]     ]
    [ [   { \ literal-3 \    } ]     ]

    [ ; ON OVERFLOW imperative-statement ]
```

Where

identifier-1	names an alphanumeric data item whose value is a program name. It cannot be used if the INTRINSIC key word is specified, nor can it be used if the program being called is not a dynamic subprogram residing in a Segmented Library.
literal-1	is a nonnumeric literal which either names an MPE intrinsic, or names a program. If an intrinsic is named by *literal-1*, the key word, INTRINSIC, must precede it.
\\	represents a null value passed as a parameter to an intrinsic or to an SPL program which includes the OPTION VARIABLE command.
@ ***identifier-2,*** @ ***identifier-3,*** and so forth	indicates that the byte addresses of the data items represented by *identifier-2*, and so forth are to be passed as parameters. These types of parameters may only be passed to non-COBOL programs.
identifier-2, identifier-3, and so forth	are the names of any data items in the calling program, or are files named in an FD level entry in the File Section of your program. If these parameters represent data items, they must each start on a word boundary. They must be described in the File, Working-Storage, or Linkage Section of your program. When these values are passed to another program, they are passed by reference. If these parameters name files, the called program must not be a COBOL program.
literal-2, literal-3, and so forth	are all either integer numeric literals of less than 10 digits, or are octal numeric literals.
\ ***identifier-2*** \ , \ ***identifier-3*** \ , \ ***literal-2*** \ , and \ ***literal-3*** \	are used to indicate that the literals or data items represented by identifiers enclosed by the back slashes are to be passed by value to the called program or intrinsic. This may only be used when the called program is not a COBOL program. If an identifier is used in this way, it must represent a numeric data item of less than 10 digits.
identifier-4	is the name of a numeric or numeric edited data item in the calling program. It is used only in calls to non-COBOL programs.
imperative-statement	is one or more imperative statements.

A CALL statement may appear anywhere within a segmented program. The compiler provides all controls necessary to insure that the proper logic flow is maintained. Thus, when a CALL statement to a subprogram appears in a section with a segment-number greater than or equal to 50, that segment is in its last used state when control is returned to the calling program.

Calling Intrinsics

The INTRINSIC phrase is used to indicate that the CALL statement in which it appears is calling an MPE intrinsic rather than a subprogram. However, if you are calling an intrinsic which neither has parameters, nor returns a value, you need not specify the INTRINSIC phrase. For example, the call to the TERMINATE intrinsic, shown below, is valid.

```
IF ERR-IN-READ = 1 THEN CALL ''TERMINATE''.
```

When the INTRINSIC phrase is used, literal-1 must be used, and must name an MPE intrinsic. The USING phrase is then used to specify the values of various parameters to be passed to the intrinsic.

As with subprograms, the parameters passed to intrinsics are specified by position. If a parameter of an intrinsic is optional, and you do not wish to pass a value for that parameter, you must specify two consecutive back slashes (\ \ ) in the position within the USING phrase of the CALL statement that corresponds to that parameter's position within the intrinsic.

If you wish to pass a literal or a data item represented by an identifier, you need only specify the literal or identifier.

Unlike subprograms, if an intrinsic expects a parameter to be passed by value rather than by reference, it is unnecessary to enclose the literal or identifier with backslashes. The intrinsic automatically assumes that the parameter is being passed by value.

The special relation operator,

```
                              { > }
mnemonic-name   [ NOT ]       { = }    0
                              { < }
```

can be used after a call to an intrinsic to check the condition code returned by an intrinsic. This relation condition is discussed in Section X, under the heading RELATION CONDITIONS.

For information on the condition codes and the types, number, and position of parameters required to call a specific intrinsic, refer to the MPE INTRINSIC REFERENCE Manual, part number 30000- 90010.

As an example,

```
                  .
                  .
                  .
    SPECIAL NAMES.
       CONDITION-CODE IS C-C.
                  .
                  .
                  .
    WORKING-STORAGE SECTION.
    01 SHOWME.
        02 MPE-COMMAND     PIC X(07)  VALUE "SHOWJOB".
        02 CARRIAGE-RETURN PIC X      VALUE %15.
    01 ERR       PIC S9999  USAGE IS COMP VALUE ZERO.
    01 CATCHPARM PIC S9999  USAGE IS COMP VALUE ZERO.
    01 REPLY     PIC X(03)  VALUE SPACES.
                  .
                  .
                  .
    PROCEDURE DIVISION.
                  .
                  .
                  .
      CALL INTRINSIC "COMMAND", USING SHOWME, ERR, CATCHPARM.
      IF C-C   NOT = 0 THEN
            DISPLAY "COMMAND FAILED"
            STOP RUN.
         MOVE SPACES TO REPLY.
         DISPLAY "DO YOU WISH TO COMMUNICATE WITH ANYONE?"
         ACCEPT REPLY.
         IF REPLY = "YES" PERFORM BREAK-FOR-COMM.
                  .
                  .
                  .
    BREAK-FOR-COMM.
         DISPLAY "TYPE RESUME WHEN READY TO RESUME PROGRAM".
         CALL "CAUSEBREAK".
                  .
                  .
                  .
```

The first intrinsic call above uses the COMMAND intrinsic to issue the MPE command, SHOWJOB. This allows you to see who is currently logged on to your system. The second call to an intrinsic is the programatic equivalent to pressing the BREAK key, thus suspending your program.

The intrinsics used in the above manner could allow you to see who is using the system, and to ask them, for example, to release a file from their group to allow your program to access it.

When CALL INTRINSIC is used for intrinsics in the system file SPLINTR,

1. The special symbols "@" and "\" are optional.

2. Data conversions for parameters passed by value are performed automatically.

Calling Programs

When the CALL statement is used to call a subprogram, several rules apply.

If you wish to call a dynamic or ANSISUB subprogram, you can use either *literal-1* or *identifier-1*. If you use *identifier-1*, the subprogram must be either dynamic or ANSISUB, and it must reside in a Segmented Library (SL). To call a non-dynamic subprogram (or an intrinsic), you must use *literal-1*.

As stated earlier in this section, a dynamic subprogram is always in its initial state when it is called by another program. A non-dynamic subprogram is in its initial state only when it is called for the first time within a run unit. An ANSISUB subprogram is in its initial state only when it is called for the first time within a run unit or when it is called for the first time following the execution of a CANCEL statement which specifies that particular ANSISUB subprogram.

On all other entries into the called non-dynamic subprogram, the state of the program is maintained in its state when last exited. This includes all data fields, the status and positioning of all files, and all alterable switch settings.

Using Phrase

The USING phrase specified in a CALL statement to another program makes data and files of the calling program available to the called program.

If a CALL statement containing the USING phrase calls a COBOL program, then the called COBOL program must contain a USING phrase in its Procedure Division header (or in the ENTRY statement, if a secondary entry point name is used by the CALL statement). The USING phrase of the called program must contain as many operands as does the USING phrase of the calling program. The USING option makes available to a called subprogram up to 57 data items (or FD file names) defined in the calling program. However, file-names cannot be passed to a COBOL subprogram.

The order of appearance of names of data items in the USING phrase is very important, for the reasons discussed under the heading, REFERENCE TO COMMON DATA, at the beginning of this section. To reiterate briefly, data is passed from the calling program to the called program on a positional basis, rather than by name. Hence, the third name in a USING phrase of a calling program corresponds to the third name in the USING phrase of the called COBOL program.

This positional correspondence extends to non-COBOL called programs. Thus, for example, if the called program is an SPL program, then the names in the parameter list of the procedure declaration in that program are identified with those data items whose names appear in the corresponding position of the USING phrase in the calling program.

As was stated in the description of *identifier-2, identifier-3*, and so forth, these identifiers may name files to be passed to a called program only if it is not a COBOL program. Furthermore, although you can enclose such a file identifier in back slashes (which are ignored), preceding it with the commercial "at" sign results in an error.

Note that index-names in the calling and called programs always refer to different indices. To pass an index value from a calling program to a called program you must first move the index value associated with an index-name to a data item which appears in the USING phrase of the calling program. The corresponding data item in the called program can then be used to set an equivalent index-name to this value.

To pass a data item by value to a non-COBOL program, you must enclose the associated identifier in back slashes. If the value passed is a literal, the back slashes are optional. Note that passing a data item by value leaves the data item in the calling program unchanged following execution of the called program.

To pass a data item as a byte pointer (to a non-COBOL program), you must precede its identifier by the commercial "at" sign. (In this case, since the pointer is an address of data in the calling program, the data in the calling program can be altered by the called program.

If an identifier is not passed by value, or as a byte pointer (that is, is not enclosed in back slashes or preceded by a commercial at sign), it is passed as a word pointer (that is, by reference). Thus, the data in the calling program can be altered by the called program if it is passed in this manner. In calls to COBOL programs, this is the standard method of referencing common data.

Two consecutive back slashes (\\ ) may be specified in a USING phrase of a CALL statement if the called program is an SPL procedure whose OPTION command specifies the VARIABLE parameter. When two consecutive back slashes are used, this indicates that a parameter is not being sent, and should not be expected. In this case, the last parameter passed must be an unsigned numeric data item two characters long. This last parameter is called a "bit mask." The bit mask must be passed by value, and indicates to the called program those parameters which are not being passed.

To illustrate,

```
CALL "SPLPROC" USING \TESTER\, \\, @RESULT, \5\.
```

The bit mask in this case is 5, which, in a binary word (16 bits), is 0000000000000101. This indicates that the second parameter in the list is excluded, and that the SPL procedure need not expect it.

The appearance of ones in the bit mask indicate that the first and third parameters are present, and may be accessed. The bit mask is generated automatically by the compiler if you specify the INTRINSIC option.

GIVING PHRASE

The GIVING phrase allows calls to non-COBOL, typed procedures.

A typed procedure is, for example, an SPL procedure whose procedure declaration uses one of the key words LOGICAL, REAL, INTEGER, LONG, or DOUBLE. Such a procedure must always return a value on top of your data stack when it completes execution. In COBOL II/3000, this is assumed to be a one or two word binary value of up to nine digits.

The purpose of the GIVING phrase is to provide for this returned value. Since *identifier-4* is used to hold the value returned by a typed procedure, its length must be sufficient to accommodate the value returned.

Overflow Conditions

When you use the *identifier-1* form in the CALL statement (calling a dynamic or ANSISUB subprogram in an SL), the MPE intrinsic, LOADPROC, is used to attempt to load the called subprogram. If the execution of the LOADPROC intrinsic is unsuccessful, an overflow condition occurs. When this condition occurs, the attempt to load the called program fails. Additionally, if the ON OVERFLOW phrase is specified in the CALL statement, control is passed to the *imperative-statement* of that phrase.

If no ON OVERFLOW phrase is specified, and the CALL statement causes an overflow condition, the entire run unit is aborted.

CANCEL STATEMENT

The CANCEL statement is used in ANS COBOL to release the memory areas occupied by a specified subprogram.

Cancel Statement Format

CANCEL { *identifier-1* | *literal-1* } [, *identifier-2* | , *literal-2*]

Where

identifier-1 and ***identifier-2*** are each defined as an alphanumeric data item whose contents name a subprogram compiled by the COBOL II/3000 compiler.

literal-1 and ***literal-2*** are each non-numeric literals which name a COBOL subprogram compiled by the COBOL II/3000 compiler.

When a CANCEL statement is issued for a COBOL program, the program specified in the statement must have already executed a GOBACK or EXIT PROGRAM statement.

For both dynamic and non-dynamic subsprograms in COBOL II/3000, the CANCEL statement has no effect. This occurs because automatic release of the memory areas accompanies the exiting of the dynamic subprogram, and, for non-dynamic subprograms, the data area is permanently assigned and the code segments are automatically released.

For ANSISUB subprograms, a CALL statement which is executed after a CANCEL statement naming the same program will cause that program to be brought into memory in its initial state.

If a CANCEL statement specifies a program which has not been called, or has been called but is presently cancelled, the CANCEL statement is ignored, and control passes to the next executable statement.

ENTRY STATEMENT

The ENTRY statement establishes a secondary entry point in an HP COBOL subprogram. The ENTRY statement is an HP extension of ANSI COBOL'74.

Entry Statement Format

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

Where

literal-1 is a non-numeric literal. It must be formed according to the rules for program names, but must not be the name of the called program in which it appears. Furthermore, it must not be the name of any other entry point or program name in the run unit.

identifier-1, identifier-2, and so forth are as described and used in the USING phrase of the Procedure Division header. See Section X, THE PROCEDURE DIVISION, for details.

The link between the calling program and a secondary entry point of a called program is supplied by *literal-1*. That is, *literal-1* must be used in a CALL statement of the calling program, and must appear in an ENTRY statement in the called program.

When the called program is invoked using such CALL and ENTRY statements, the called program is entered at the ENTRY statement which specifies *literal-1*.

The USING option has the same format and meaning as in the USING phrase of the Procedure Divison header. See Section X for details.

The ENTRY statement is illustrated below.

```
IDENTIFICATION DIVISION.
PROGRAM ID.  MAINPROG.
       .
       .
       .
DATA DIVISION.
FILE SECTION.
FD INV-FILE.
01 INV-REC.
   02 PART-NUM                 PIC X(10).
   02 PART-NAME                PIC X(30).
   02 BEGIN-QTY                PIC 9(6).
   02 PRICE-WHSL               PIC 9(3)V99.
   02 PRICE-RETAIL             PIC 9(4)V99.
FD SALES-FILE.
01 SALES-REC.
   02 SOLD-PT-NO               PIC X(10).
   02 SOLD-PART                PIC X(30).
   02 SOLD-QTY                 PIC 9(6).
       .
       .
       .
PROCEDURE DIVISION.
       .
       .
       .
      IF SOLD-QTY IS NOT EQUAL TO ZERO
         CALL "SUBPRO1-ENTRY1" USING INV-REC, SALES-REC.
       .
       .
       .
      STOP RUN.
IDENTIFICATION DIVISION.
PROGRAM-ID.   SUBPRO1.
       .
       .
       .
DATA DIVISION.
FILE SECTION.
FD PRINT-FILE.
01 P-REC                       PIC X(132).
       .
       .
       .
LINKAGE SECTION.
01 ORIGINAL.                   PIC X(10).
   02 PT-NUM                   PIC X(30).
   02 PT-NAME                  PIC 9(6).
   02 START-QTY                PIC 9(3)V99.
   02 OUR-PRICE                PIC 9(4)V99.
   02 THEIR-PRICE
01 SALES.
   02 SOLD-NUM                 PIC X(10).
   02 SOLD-NAME                PIC X(30).
   02 QTY-SOLD                 PIC 9(6).
```

```
WORKING-STORAGE SECTION.
01 HEADER.
      .
      .
      .
01 WRITE-SALES.
   02 FILLER                PIC X(15)        VALUE SPACES.
   02 NAME                  PIC X(30)        VALUE SPACES.
   02 FILLER                PIC X(5)         VALUE SPACES.
   02 NUM-1                 PIC X(10)        VALUE SPACES.
   02 FILLER                PIC X(5)         VALUE SPACES.
   02 QUANTITY              PIC Z(3)9(3)     VALUE ZERO.
   02 FILLER                PIC X(5)         VALUE SPACES.
   02 GROSS-SALES           PIC $Z(10).99    VALUE ZEROS.
   02 FILLER                PIC X(5)         VALUE SPACES.
   02 GROSS-PROFIT          PIC $X(10).99.
      .
      .
      .
PROCEDURE DIVISION.
      .
      .
      .
ENTRY "SUBPRO1-ENTRY1" USING ORIGINAL, SALES.
     MULTIPLY QTY-SOLD BY THEIR-PRICE GIVING GROSS-SALES.
     SUBTRACT START-QTY FROM QTY-SOLD GIVING QUANTITY.
     COMPUTE GROSS-PROFIT =
       (THEIR-PRICE - OUR-PRICE) * QTY-SOLD.
     MOVE PART-NUM TO NUM-1.
     MOVE PART-NAME TO NAME.
     WRITE P-REC FROM HEADER AFTER ADVANCING 1 LINES.
     WRITE P-REC FROM WRITE-SALES AFTER ADVANCING 3 LINES.
     GOBACK.
      .
      .
      .
```

The CALL statement in MAINPROG specifies that SUBPRO1 is to be executed, starting at the ENTRY statement rather than at the first line following the Procedure Division header. Also, the data areas of INV-FILE and SALES-FILE are to be used in both programs.

EXIT PROGRAM STATEMENT

The EXIT PROGRAM statement marks the logical end of a logical program.

Exit Program Statement Format

EXIT PROGRAM

If you use an EXIT PROGRAM statement in a program (called or otherwise), it must appear as the only statement in a sentence, and the sentence must be the only sentence in a paragraph.

When encountered in a called program, the EXIT PROGRAM statement causes control to return to the statement of the calling program which immediately follows the CALL statement used to pass control to the called program.

If the EXIT PROGRAM statement is used in a main program, it is treated as an EXIT statement. Thus, in a main program, it serves only as a way of terminating a procedure.

GOBACK STATEMENT

The GOBACK statement is an extention to COBOL II. It marks the logical end of a program.

GOBACK statement Format

GOBACK

It must be the only statement in a sentence, or if used in a series of imperative statements, it must be the last statement in the series.

If a GOBACK statement is encountered in a called program, control returns to the statement in the calling program immediately following the CALL statement which initiated the called program. Thus, in a subprogram, the GOBACK statement acts in the same way as an EXIT PROGRAM statement.

If a GOBACK statement is used in a main program, it is equivalent to a STOP RUN statement. Thus, it indicates the logical end of the run-unit, and control is passed to MPE.

The chart below shows the relationship between the EXIT PROGRAM, 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 sub-program and the calling program(s)—return control to MPE
GOBACK	Logical end of run—return control to MPE	Return to calling program

SORT-MERGE OPERATIONS

SECTION XIII

The sort-merge capabilities of COBOL II allow you to sort one or more files of records, or to combine two or more identically ordered files of records one or more times within a given execution of a COBOL program.

Additionally, you are given the capability to apply some special processing to each of the individual records by input or output procedures which are part of the sort or merge process. For a sort operation, this special processing may be applied before as well as after the records have been sorted. For a merge operation, this processing may be applied after the records have been combined. The RELEASE and RETURN statements are used in these input and output procedures to release or return records which are to be, or have been, respectively, sorted or merged.

MERGE STATEMENT

The MERGE statement combines two or more identically sequenced files on a set of specified keys, and, as part of the merge operation, makes records available in their merged order to an output procedure or an output file.

The records are made available following the actual merging of the files; the output procedure or the moving of records to an output file is considered part of the merge process.

MERGE statements may apear anywhere within the Procedure Division except in the Declaratives portion, or in an input or output procedure associated with a SORT or MERGE statement.

MERGE Statement Format

```
                          { ASCENDING  }
MERGE file-name-1 ON      { DESCENDING }  KEY data-name-1 [ ,data-name-2 ] ...

                   [      { ASCENDING  }                                    ]
                   [ ON   { DESCENDING }  KEY data-name-3 [ ,data-name-4 ] ... ] ...

   [ COLLATING SEQUENCE IS alphabet-name ]

    USING file-name-2 , file-name-3 , [ , file-name-4 ] ...

   { OUTPUT PROCEDURE IS section-name-1 [ { THROUGH } section-name-2 ] }
   {                                    [ { THRU    }                ] }
   {                                                                   }
   { GIVING file-name-5                                                }
```

Where

file-name-1 is a sort/merge file, and is described in a sort/merge file description (SD level) entry in the Data Division.

data-name-1, data-name-2, and so forth — are data items described in records associated with *file-name-1*. Each may be qualified, and cannot be variable in length. None of these data names can be described by an entry either containing an OCCURS clause, or subordinate to an entry containing such a clause. If *file-name-1* has more than one record description, then the data items represented by these data names need be described in only one of the record descriptions.

alphabet-name is either EBCDIC, STANDARD-1, NATIVE, or an alphabet name as defined by you in the SPECIAL-NAMES paragraph of the Environment Division.

file-name-2, file-name-3, and ***file-name-4*** — are the files whose records are to be merged. These files must not be open at the time the MERGE statement is executed. Each must be a sequential file described in an FD level file description in the Data Division. No more than one such file name can name a file from a multiple file reel. Any given file name can be used only once in any given MERGE statement.

The actual size of the logical record or records described for these files must be equal to the actual size of the logical record or records described for *file-name-1*. If the data descriptions of the elementary items that make up these records are not identical, it is your responsibility to describe the corresponding records in such a way as to cause an equal number of character positions to be allocated for the corresponding records.

section-name-1 is the name of the first section in an output procedure.

section-name-2 is the name of the last section in an output procedure.

file-name-5 is the name of an output file. This file is subject to the same restictions as *file-name-2, file-name-3,* and *file-name-4*.

The words, THROUGH and THRU, are equivalent, and can be used interchangably.

The MERGE statement merges all records contained in the files named in the USING phrase. The files to be merged are automatically opened and closed by the merge operation with all implicit functions performed, such as the execution of any associated USE procedures. The files described by *file-name-2* through *file-name-5* must not be open when the MERGE verb is executed, and may not be opened or closed during an output procedure if one is specified.

Following the actual merging of the files, but before they have been closed, the merged records are released in the order in which they were merged, to either the specified output procedure, or the specified output file.

The results of a merge operation are predictable only when the records in the files to be merged are ordered as described in the ASCENDING or DESCENDING KEY phrase associated with the MERGE statement.

The data-names following the word KEY are listed from left to right in order of decreasing significance. This decreasing significance is maintained from KEY phrase to KEY phrase. Thus, in the format shown, *data-name-1* is the most significant, *data- name-2* is the next most significant, *data-name-3* is less significant than *data-name-2*, and *data-name-4* is the least significant.

When the MERGE statement is executed, the records of *file-name-2, file-name-3,* and *file-name-4* are merged in the specified order (ASCENDING or DESCENDING) using the most significant key data item. Within the records having the same value for the most significant key data item, the records are merged according to the next most significant key data item; this kind of merging continues until all key data items named in the MERGE statement have been used.

When the ASCENDING phrase is used, the merged records are in a sequence starting from the lowest value of the key data items, and ending with the highest value.

When the DESCENDING phrase is used, the merged records are in a sequence from the highest value of the key data item to the lowest.

Merging takes place using the rules for comparison of operands of a relation condition. If all corresponding key data items of records to be merged are equal, the records are written to *file- name-5*, or returned to the output procedure, in the order that the input files are specified in the MERGE statement.

COLLATING SEQUENCE Phrase

The COLLATING SEQUENCE phrase allows you to specify what collating sequence to use in the merging operation. This phrase is optional. The program collating sequence will be used if none is specified in the MERGE statement.

See the alphabet clause of the SPECIAL-NAMES paragraph for information on defining and using collating sequences.

GIVING and OUTPUT PROCEDURE Phrases

You must specify either the GIVING or OUTPUT PROCEDURE phrase in a MERGE statement.

If you specify the GIVING statement, all merged records are automatically written to *file-name-5* as the implied output procedure for the MERGE statement.

If you use the OUTPUT PROCEDURE phrase, there are several rules you must follow in writing the procedure.

The procedure must consist of one or more sections that appear contiguously in the source program, and that are not part of any other procedure.

Since the RETURN statement is the means of making sorted or merged records available for processing, at least one such statement must appear in the procedure. The procedure may consist of any procedures needed to select, modify, or copy records. The records are returned one at a time in merged order from *file- name-1.*

Control must not be passed to a sort/merge output procedure except during the execution of a SORT or MERGE statement. The procedure itself can contain no SORT or MERGE statements, nor can it contain any explicit transfers of control to points outside its bounds; ALTER, GO TO, and PERFORM statements in the output procedure must refer to procedure names within the bounds of the output procedure. Note that implied transfers of control to declarative procedures are permitted. Thus, for example, a WRITE statement without an AT END phrase is permitted, and will transfer control to some associated declarative procedure if an AT END condition occurs.

The remainder of the Procedure Division must not contain any transfers of control to points inside sort/merge output procedures. No ALTER, GO TO, or PERFORM statements outside an output procedure can refer to procedure names within the output procedure.

When an output procedure is specified in a MERGE statement, the output procedure is executed as part of the merge operation. The procedure is used after the records have been merged.

At compile time, the compiler inserts a return mechanism at the end of the last section in the output procedure. When this return mechanism is reached, the merge operation is terminated, and control passes to the next executable statement following the MERGE statement.

Segmentation Considerations

The following restrictions apply to the MERGE statement when it is used in a segmented program.

If the MERGE statement appears in a section whose segment number is less than 50, then any output procedure named in the MERGE statement either must be totally contained within a segment (or segments) whose segment number (or numbers) is less than 50, or must be wholly contained in a single segment whose segment number is greater than 49.

If the MERGE statement appears in a segment whose segment number is greater than 49, then any output procedure referenced by the MERGE statement must either be entirely contained within the same segment, or be entirely contained within segments whose segment numbers are less than 50.

RELEASE STATEMENT

The RELEASE statement can be used in an input procedure of a SORT statement to transfer records from your program to the initial phase of the sort operation.

RELEASE Statement Format

RELEASE ***record-name*** [**FROM** ***identifier***]

Where

record-name is the name of a logical record in the sort/merge file description entry of the file referenced in the associated SORT statement. It may be qualified. *Record-name* must not refer to the same storage area as *identifier*.

identifier is the name of a data item described in your program. It must not refer to the same storage area as *record-name*.

A RELEASE statement may only be used within the range of an input procedure associated with a SORT statement for a file whose sort/merge file description entry contains *record-name*.

When the RELEASE statement is executed, a single record is made available, from *record-name*, to the initial phase of the sort operation. After the execution of the RELEASE statement, the logical record is no longer available in the record area named by *record-name* unless the associated sort/merge file is named in a SAME RECORD AREA clause. If the sort/merge file is named in the clause, the logical record is available to the program as a record of the other files named in the SAME AREA clause, as well as to the sort/merge file.

If the FROM phrase is used in a RELEASE statement, the contents of *identifier* are moved to *record-name*, then the contents of *record-name* are released to the sort file.

Although, as stated in the preceding paragraph, the logical record named by *record-name* might no longer be available, the data in *identifier* remains available following execution of the RELEASE statement.

When control passes from the input procedure, the sort file consists of all those records which were placed in it by the execution of RELEASE statements.

RETURN STATEMENT

The RETURN statement can be used in an output procedure of a SORT or MERGE statement. It cannot be used in any other type of procedure.

When used, the RETURN statement obtains either sorted records from the final phase of a sort operation, or merged records during a merge operation. Each record is obtained by a RETURN statement in the order specified by the keys listed in the SORT or MERGE statement. These records are made available for processing in the record area associated with the sort or merge file, and, optionally, to another data area.

RETURN Statement Format

RETURN ***file-name*** **RECORD [INTO** ***identifier*****] ; AT END** ***imperative-statement***

Where

file-name is the name of the file used as the sort or merge file in the SORT or MERGE statment associated with the output procedure in which the RETURN statement appears. It must be described in a sort/merge file description entry (SD level) in the Data Division.

identifier is the name of a data item in your program. The storage area referenced by *identifier* must not be the same as the record area associated with *file- name.*

imperative-statement is one or more imperative statements.

When logical records of a sort/merge file are described with more than one record description, these records automatically share the same storage area; this is equivalent to an implicit redefinition of the area. The contents of any data items which lie beyond the range of the current data record are undefined at the completion of the execution of the RETURN statement.

INTO Phrase

The INTO phrase, if specified in the RETURN statement, moves the current record into the record area associated with *file-name*, and then uses an implicit MOVE statement (without the CORRESPONDING phrase) to move a copy of the data from the record area to the storage area referenced by *identifier*. Thus, the data obtained from the SORT or MERGE statement is available in the data area associated with *identifier* as well as to the input record area.

Any subscripting or indexing associated with *identifier* is evaluated after the record has been returned to the file, and immediately before it is moved to the storage area referenced by identifier.

The INTO phrase must not be used when the input file contains logical records of various sizes as indicated by their record descriptions.

AT END Phrase

If no next logical record exits for the file at the time a RETURN statement executes, an AT END condition occurs. The contents of the record areas associated with the file when the AT END condition occurs are undefined; however, if the INTO phrase was used, the contents of *identifier* are the data moved into it by the preceding execution of the RETURN statement.

When the AT END condition occurs, the *imperative-statement* in the AT END phrase is executed. Following execution of *imperative- statement*, no RETURN statement may be executed as a part of the current output procedure.

SORT STATEMENT

The SORT statement creates a sort file either by executing an input procedure, or by transferring records from another file. The records of the sort file are then sorted using a specified set of keys, and are made available in sorted order to either an output procedure or an output file.

SORT Statement Format

```
SORT file-name-1 ON { ASCENDING  } KEY data-name-1 [ , data-name-2 ] ...
                    { DESCENDING }

     [ ON { ASCENDING  } KEY data-name-3 [ , data-name-4 ] ... ] ...
     [    { DESCENDING }                                       ]

  [ COLLATING SEQUENCE IS alphabet-name ]

  { INPUT PROCEDURE IS section-name-1 [ { THROUGH } section-name-2 ] }
  {                                   [ { THRU    }                ] }
  { USING file-name-2 [ , file-name-3 ] ...                          }

  { OUTPUT PROCEDURE IS section-name-3 [ { THROUGH } section-name-4 ] }
  {                                    [ { THRU    }                ] }
  { GIVING file-name-4                                                }
```

Where

file-name-1 is a sort/merge file, and is described in a sort/merge file description entry in the Data Division.

data-name-1,
data-name-2,
and so forth

are data items described in records associated with *file-name-1*. Each may be qualified, and cannot be variable in length. None of these data names can be described by an entry which either contains an OCCURS clause, or is subordinate to an entry containing such a clause. If *file-name-1* has more than one record description, then the data items represented by these names need be described in only one of the record descriptions.

alphabet-name is a name defined by you in the SPECIAL-NAMES paragraph of the Environment Division.

file-name-2,
and
file-name-3,

are the files whose records are to be sorted. These files must not be open at the time the SORT statement is executed. Each must be a sequential file described in an FD level file description entry in the Data Division. No more than one of these file-names may name a file on a multiple file reel. Any given file name can be used only once in a given SORT statement.

The actual size of the logical record or records described by these files must be equal to the actual size of the logical record or records described by *file-name-1*. If the data descriptions of the elementary items that make up these records are not identical, it is your responsibility to describe the corresponding records in such a way as to cause an equal number of character positions to be allocated for the corresponding records.

section-name-1 is the name of the first section in an input procedure.

section-name-2 is the name of the last section in an input procedure.

section-name-3 is the name of the first section in an output procedure.

section-name-4 is the name of the last section in an output procedure.

file-name-4 is the name of an output file. This file is subject to the same restrictions and rules as *filename-2* and *file- name 3*, above.

The words, THROUGH and THRU, are equivalent, and may be used interchangably.

The Procedure Division may contain more than one SORT statement appearing anywhere except in a declarative procedure, or in the input and output procedures associated with a SORT or MERGE statement. The files named by *file-name-2* through *file-name-5* must not be open when the SORT verb is executed, and may not be opened or closed during the execution of an input or output procedure if such procedures are specified.

The data-names following the word KEY are listed from left to right in order of decreasing significance. This decreasing significance is maintained from KEY phrase to KEY phrase. Thus, in the format shown, *data-name-1* is the most significant, and *data-name-4* is the least significant.

When the SORT statement is executed, the records are first sorted in the specified order (ASCENDING or DESCENDING) using the most significant key data item. Next, within the groups of records having the same value for the most significant key data item, the records are sorted using the next most significant key data item, again, in ASCENDING or DESCENDING order as specified for that key. Sorting continues in this fashion until all key data items have been used.

For example, assume that the records to be sorted use the first three key data items, and that the unsorted records appear as shown below.

1	D	U	.....
0	A	N	.....
1	N	S	.....
1	F	O	.....
1	D	R	.....
0	X	T	.....
0	A	E	.....
0	B	D	.....
1	N	R	.....
0	X	E	.....
1	F	C	.....
1	C	S	.....

If the SORT statement uses the first character position as the most significant, and the third as the least significant, and the records are to be sorted in ascending order for the first two keys, and in descending order for the last key, then the results of each pass of the sort, as well as the SORT statement, are shown below.

```
SORT TESTFILE ON ASCENDING KEY FIRSTCHAR, SECONDCHAR
                  ON DESCENDING KEY THIRDCHAR
                   USING INFILE
                    GIVING OUTFILE.
```

Where TESTFILE is, in part, described as

```
SD TESTFILE.
01 FIRSTCHAR      PIC 9.
01 SECONDCHAR     PIC X.
01 THIRDCHAR      PIC X.
```

and INFILE is described in part as

```
FD INFILE.
01 FIRST          PIC 9.
01 SECOND         PIC XX.
```

```
0 A N .....      0   A  N  .....      0   A  N .....
0 X T .....      0   A  E  .....      0   A  E .....
0 A E .....      0   B  D  .....      0   B  D .....
0 B D .....      0   X  T  .....      0   X  T .....
0 X E .....      0   X  E  .....      0   X  E .....

1 D U .....      1   C  S  .....      1   C  S .....
1 N S .....      1   D  U             1   D  U .....
1 F O .....      1   D  R  .....      1   D  R .....
1 D R            1   F  O  .....      1   F  O .....
1 N R .....      1   F  C  .....      1   F  C .....
1 F C .....      1   N  S  .....      1   N  S .....
1 C S .....      1   N  R  .....      1   N  R .....
First Pass       Second Pass          Third Pass
```

Note that the third pass of the sort left the records unchanged from their order in the result of the second pass. The records were, by chance, arranged in their proper sequences. The SORT statement would not actually go through this third pass, as it recognizes the records as already being sorted. This saves execution time.

ASCENDING and DESCENDING Phrases

When the ASCENDING phrase is used, the sorted records are in a sequence starting from the lowest value of the key data items, and continuing to the highest value.

When the DESCENDING phrase is used, the sorted records are in a sequence from the highest value of the key data items to the lowest value.

Sorting takes place according to the rules for comparison of operands of a relation condition.

COLLATING SEQUENCE Phrase

The COLLATING SEQUENCE phrase allows you to specify what collating sequence to use in the sorting operation. This phrase is optional. The program collating sequence will be used if none is specified in the SORT statement. If you do not specify an alphabet-name in the PROGRAM COLLATING SEQUENCE clause of the Environment Division, the default is the ASCII collating sequence. See the alphabet name clause of the SPECIAL-NAMES paragraph in Section VII for information on defining and using collating sequences.

USING and INPUT PROCEDURE Phrases

You must specify either the USING or the INPUT PROCEDURE phrase in a SORT statement.

If you specify the USING phrase, all records from *file-name-2* and *file-name-3* are automatically transferred to *file-name-1*. The files named by *file-name-2* and *file-name-3* must not be open when the SORT statement is executed.

The SORT process automatically opens these files, transfers their records, and then closes them. If USE procedures are specified for the files, they are executed at the appropriate times during these implicit operations. The files are closed as though a CLOSE statement, without any optional phrases, had been issued for them.

The records are then sorted in the file named by *file-name-1*, and are released to *file-name-4* or the specified output procedure during the final phase of the sort operation.

If you do not specify a USING phrase, then you must specify an INPUT PROCEDURE phrase.

If you do so, then *section-name-1* through *section-name-2* must define an input procedure.

Control is passed to this procedure before *file-name-1* is sorted. The compiler inserts a return mechanism at the end of the last statement in the section named by *section-name-2*, and when control passes to the last statement in the input procedure, the records that have been released to *file-name-1* are sorted.

The input procedure must consist of one or more contiguous sections in the source program. The input procedure can include any statements needed to select, create, or modify records, and must include at least one RELEASE statement, since this is the statement which releases records to *file-name-1*, the sort file.

Input Procedure Restrictions

1. The input procedure must not form part of an output procedure.
2. Control must not pass to the input procedure except when a related SORT statement is being executed.
3. The input procedure must not contain a SORT or MERGE statement.
4. There must be no explicit transfers of control to points outside the input procedure. Thus, ALTER, GO TO, and PERFORM statements in an input procedure must only refer to procedure names within the input procedure.

   Implicit transfers of control, such as transfers to declarative procedures, are allowed.
5. The remainder of the Procedure Division must not contain any transfers of control to points inside the input procedure.

GIVING and OUTPUT PROCEDURE Phrases

You must specify either the GIVING or the OUTPUT PROCEDURE phrase in a SORT statement.

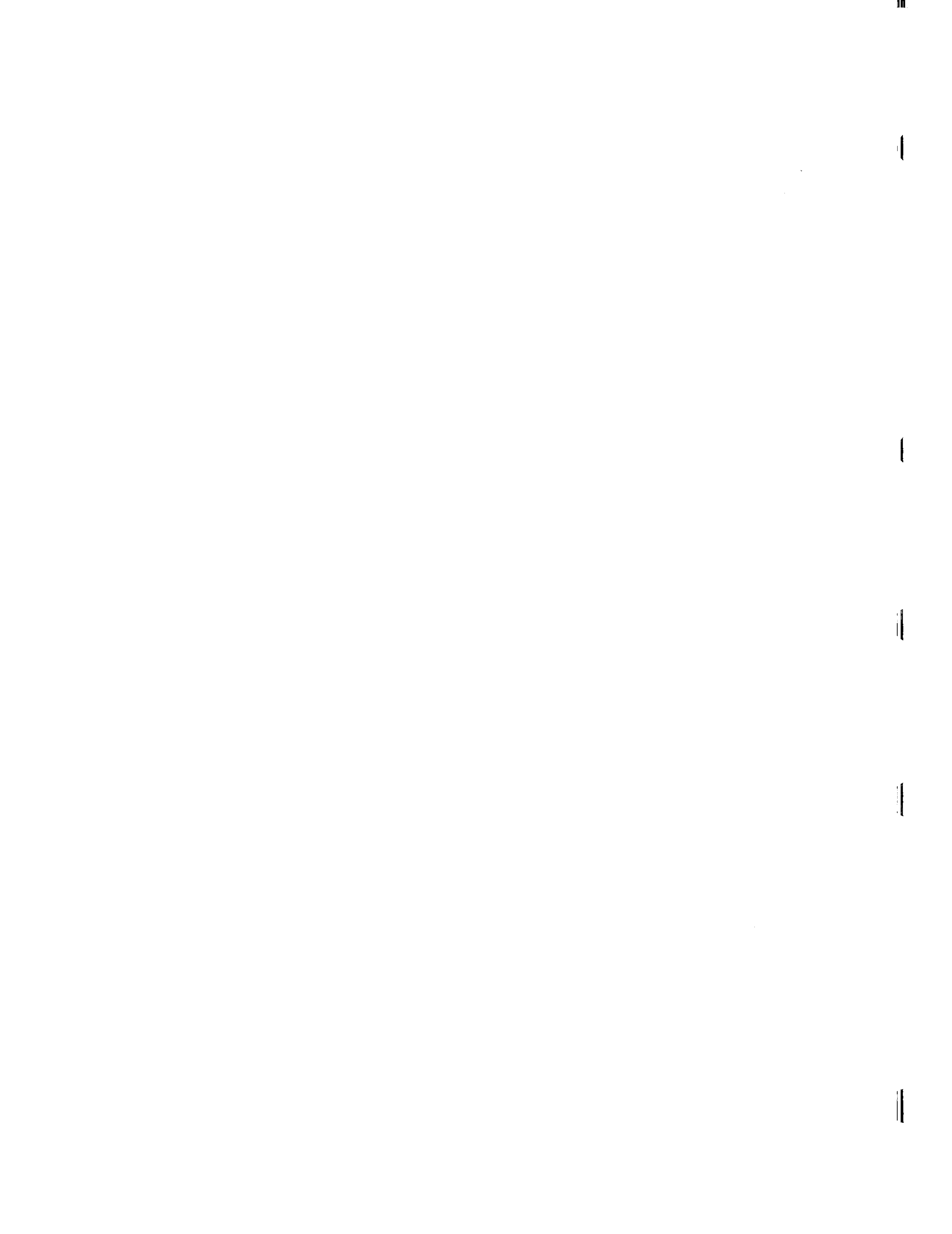
If you specify the GIVING phrase, then, as a last step in the sort process, the sorted records in *file-name-1* are automatically written to *file-name-4*.

File-name-4 must not be open when the sort process is executed.

The sort process automatically opens, writes records to, and closes *file-name-4*. If there are any USE procedures specified (whether implicitly or explicitly) for *file-name-4*, they are executed if and when appropriate, as part of the implicit function of the SORT statement. When all records have been written to *file-name-4*, the file is closed in a manner equivalent to issuing a CLOSE statement, with no optional phrases, for the file.

If you specify the OUTPUT PROCEDURE phrase rather than the GIVING phrase, the output procedure starting with the first statement in *section-name-3*, and ending with the last statement of *section-name-4*, is executed after all records have been sorted.

The rules and restrictions for an output procedure specified in a SORT statement are the same as for an output procedure specified in a MERGE statement. See the MERGE statement for details.



COBOL LIBRARIES AND THE COBEDIT PROGRAM

SECTION XIV

COBOL libraries are KSAM files containing modules of text which can be compiled and copied into your object program at compile time. The COPY statement, discussed at the end of this section, is the COBOL verb available to you to perform this function.

You can have more than one library, and by using multiple COPY statements, you can copy text from several libraries.

Copy Libraries

On an HP computer system, copy libraries are KSAM files of ASCII records which make up one or more modules. These modules are the sets of text which can be copied into your program.

The COBEDIT program in the PUB group of the SYS account is the mechanism provided to create, modify, and look at COBOL II/3000 copy libraries.

A module within a copy library is distinguished from other modules by having a string of alphanumeric characters in columns 73 through 80 of each record in that module. Since this is the only way of distinguishing one module from another, the string of characters must be unique with respect to other modules within the same library.

The example below shows three modules within a library named MYLIB. Note the use of the COBEDIT program. It is discussed on following pages.

```
:RUN COBEDIT.PUB.SYS

HP COPYLIB EDITOR  MON, MAY 21, 1980, 11:03 AM
(C) HEWLETT-PACKARD CO. 1980
     TYPE ''HELP'' FOR A LIST OF COMMANDS.
>LIBRARY MYLIB
>LIST ALL
001000$CONTROL SUBPROGRAM                                   MODULE1
001100 IDENTIFICATION DIVISION.                             MODULE1
001200 PROGRAM-ID.  DUMMY-SUB.                              MODULE1
005100 WORKING-STORAGE SECTION.                             MODULE2
005200 01  UNIV-TOTALER    PIC 9(8) COMP-3.                 MODULE2
005300 01  UNIV-ACCUM      PIC 9(8) COMP-3.                 MODULE2
008000       PERFORM TEST-IT.                               MODULE3
008100       IF RESULTANT IS LESS THAN 2                    MODULE3
008200        PERFORM TEST-FAILED;                          MODULE3
008300       ELSE NEXT SENTENCE.                            MODULE3
```

The three modules are MODULE1, MODULE2, and MODULE3. In general, there is no restriction on a module name; the names used above were chosen simply to help clarify the example.

COBEDIT Program

As was stated earlier, the COBEDIT program resides in the PUB group of the SYS account.

This program allows you to create, modify, and update a COBOL library file. Additionally, as part of the process of creating a library file, you are given the option to copy any ASCII file into the newly created library. Of course, since the records of the file will eventually be used in a COBOL program, they should be valid COBOL statements or preprocessor commands. Also, because of the restriction placed on modules by the COPY command, no COPY statement should be a part of the records in a library.

The commands available in the COBEDIT program are HELP, BUILD, COPY, LIST, EDIT, SHOW, PURGE, KEEP, and LIBRARY. You may issue any of these commands whenever the COBEDIT prompt, a greater-than sign (>), appears. In addition, you may issue several MPE commands from within the COBEDIT program by typing a colon (:) followed by the command you wish to have executed. The list of MPE commands which can be issued from within COBEDIT is shown below.

ALTSEC	LISTF	RESET	SHOWJCW
ALTVSET	LISTVS	RESETDUMP	SHOWJOB
BUILD	NEWVSET	RESTORE	SHOWME
COMMENT	PTAPE	SAVE	SHOWOUT
DEBUG	PURGE	SECURE	SHOWTIME
DSTAT	PURGEVSET	SETDUMP	SPEED
DSLINE	RELEASE	SETJCW	STORE
FILE	REMOTE HELLO	SETMSG	STREAM
GETRIN	RENAME	SHOWDEV	TELL
HELP	REPORT	SHOWIN	TELLOP

To enter the COBEDIT program, you issue the MPE RUN command as shown below.

```
:RUN COBEDIT.PUB.SYS
```

When this command is executed, the COBEDIT program displays a header, including the current date and time, and a "HELP" message followed by the greater-than prompt. To illustrate:

```
HP COPYLIB EDITOR  MON, MAY 21, 1980, 11:03 AM
(C) HEWLETT-PACKARD CO. 1980

TYPE "HELP" FOR A LIST OF COMMANDS.
>
```

There are nine commands in the COBEDIT program. Each is listed in table 14-1 and discussed on the following pages.

Table 14-1

COMMAND	MEANING
BUILD	Build a COPYLIB file.
COPY	Copy modules into library, as in BUILD command.
EDIT	Create or edit a module to be added to a COPYLIB file.
EXIT	Leave the COBEDIT program.
HELP	List all COBEDIT commands.
KEEP	Add a module to the currently active COPYLIB file.
LIBRARY	Activate an already existing COPYLIB file.
LIST	List text-names, or one or more modules of the currently active COPYLIB file.
PURGE	Purge a module of the currently active library, or purge the library itself.
SHOW	Show the name of the current library, its key file, and the latest module to be accessed.

BUILD COMMAND

The BUILD command allows you to build a new KSAM file to be used as a library file. Once this library file is built, it remains open and available for use until you either exit the COBEDIT program, or specify a new library by issuing another BUILD command, or a LIBRARY command.

BUILD Command Format

BUILD [*file-name*] [, *maxrecs*]

Where

file-name — is any name you wish to give your new library file, subject to the naming conventions for any MPE file. That is, *file-name* may be from one to eight alphanumeric characters, the first of which must be alphabetic.

maxrecs — if specified, must be greater than 0. It specifies the maximum number of records that may be placed in the file being built. If no value is specified for *maxrecs*, the default is 2500.

If you do not specify a file name, COBEDIT prompts you for one. After you are prompted for a name, if you simply press RETURN, you are given a second chance to provide a name. If you again press RETURN, the BUILD command is terminated, and no library file is created.

If you name a file in the BUILD command, or if you specify a name when COBEDIT prompts you for one, you are next prompted for a name to be used as the key file for the library file being created.

The restrictions on the key file name are the same as for *file-name*.

If you simply press RETURN, an MPE file system error message is listed, followed by an error message from COBEDIT, and the BUILD command is terminated.

Once a library file and an associated key file have been named, the COBEDIT program attempts to create a KSAM file using the specified names. If this attempt fails, an MPE error message is generated.

Otherwise, you are given the opportunity to copy a file into your newly created library file. When the file name prompt is given, if you respond with a carriage return, the BUILD command is terminated. To copy a file into the library file, you must type the name of the file in response to the prompt. This name can be fully qualified. That is, it can be specified in the form, *filename.group.account.*

If the file resides in your group, all you need to type is the name of the file. If it resides in a group other than your log-on group, you must type the name of the file followed by a period and the name of the group. If the file resides in an entirely different account from your log-on account, you must use the fully qualified form of the file name.

You must have the capability to access files in a group or account other than your own. The most common manner in which this is accomplished is by using the MPE RELEASE command. See the MPE Commands Reference Manual, part number 30000-90009 for details. Also, see the System Manager/System Supervisor Reference Manual, part number 30000-90014, for details on file security and access.

If you do not have access to the specified file, the message,

```
SECURITY VIOLATION  (FSERR 93)
BUILD TERMINATED
```

is returned.

After the file has been specified, you are asked if the file is in COPYLIB format. This is equivalent to asking you if the file to be copied has text-names in columns 73 through 80.

If you respond Y or YES, COBEDIT attempts to copy the requested file. Note that if the text-name is blank, the COBEDIT program will copy the records into your library, but you will be unable to access them later. Thus, if no text-name is present, you should respond to the question with anything other than Y or YES.

If you do supply a negative response, COBEDIT asks you for a text-name to be used for the copied records. This text-name must be from one to eight characters long.

After a file has been copied into your library file, you are asked if there are more files to be copied. A negative response terminates the BUILD command. A positive response causes the COBEDIT program to repeat the questions and actions described in the preceding three paragraphs.

NOTE

> If the file to be copied is in copylib format and has duplicate copies of one or more modules, COBEDIT will create a module with multiple copies of those modules. To avoid this problem, edit the file before copying it.

To illustrate the BUILD command:

```
>BUILD
What is the name of your library file?  MYLIB
Name a key file to be used with MYLIB:  KMYLIB
To copy a file into MYLIB now, enter the file name.
File name? COBCOPY
Is the file in copylib format? NO
Text-name for library module? MODULE1
5 records copied to library file.
Do you wish to copy more files?
Respond YES or NO:  NO
Library file created;  requested file(s) copied.
>
>BUILD MYLIB
Name a key file to be used with MYLIB: MYLIBKEY
Unable to create KSAM file
DUPLICATE PERMANENT FILE NAME  (FSERR 100)
>
>BUILD FRED
Name a key file to be used with FRED:  KEYFRED
To copy a file into FRED now, enter the file name.
File name?
Library file created.
>
```

Note that if you name a file to be copied into your library file, and the library file does not have a sufficient amount of free space to contain the records of the file being copied, no records are copied, and the BUILD command is terminated. To illustrate, we create a library file which is too small to contain the records from a specified file.

```
>BUILD ATLAS, 3
Name a key file to be used with ATLAS: KEYATLAS
To copy a file into ATLAS now, enter the file name.
File name? COBCOPY
NOT ENOUGH SPACE FOR FILE COBCOPY
0 records copied to library file.
BUILD TERMINATED
>
```

COPY COMMAND

The COPY command allows one to copy additional modules into a library which was created previously using the BUILD command. To use COPY, the library must either be the current library or it must be activated by using the LIBRARY command. COPY prompts and executes in a way similar to the BUILD command.

COPY Command Format

COPY

To illustrate use of the COPY command:

```
:RUN COBEDIT.PUB.SYS

HP32233A.00.00 COPYLIB EDITOR - COBEDIT TUE, FEB 12, 1980, 3:18 PM
(C) HEWLETT-PACKARD CO. 1980

TYPE ``HELP'' FOR A LIST OF COMMANDS.
>COPY
No library is open.
>LIB MYLIB
>COPY
To copy a file into MYLIB now, enter the file name.
File name?  COBCOPY
Is the file in copylib format? NO
Text-name for Library module?  MOD2
13 records copied to library file.
Do you wish to copy more files?
Respond YES or NO:  NO
Requested file(s) copied.
>EXIT

END OF PROGRAM
```

EDIT COMMAND

The EDIT command calls the EDIT/3000 subsystem, and optionally allows you to name a module from the currently active library to be edited.

EDIT Command Format

<u>**EDIT**</u> [***text-name***]

Where *text-name* is the name of a module in the currently active library.

EDTXT is the name of the temporary text file used as the interface between COBEDIT and EDIT/3000. If you name a module to be edited, a copy of the module, excluding the text-name in columns 73 through 80, is moved into EDTXT.

If you do not name a module, a single blank record with a record number of .001 is moved into EDTXT. This blank record is placed in EDTXT in order to place the EDIT/3000 work file in COBOL format. If you do not wish to use the blank record, you can delete it.

Once you have entered the EDIT/3000 subsystem, you can use any of its features, except two, to perform any of your editing tasks.

The two features which may not be used are the TEXT and KEEP commands.

The TEXT command cannot be used since EDTXT is automatically used as the TEXT file when you enter the EDIT command. You may use the JOIN command to append ASCII files to EDTXT.

The KEEP command may not be used for the same reason. That is, an automatic KEEP is issued, naming EDTXT as the KEEP file.

An example of the EDIT command is shown on the following page.

```
>EDIT

HP32201A.7.05 EDIT/3000  MON, MAY 21, 1980,  3:28 PM
(C) HEWLETT-PACKARD CO. 1980
WARNING: FORMAT=COBOL VALUES SET FOR LENGTH,RIGHT,FROM,DELTA,FRON
/L ALL
 .001
/D
 .001
*** WARNING *** WORK FILE IS EMPTY.
/A
      1       $CONTROL SUBPROGRAM
      1.1      PROGRAM-ID.  FRESHTEST.
      1.2      AUTHOR.  JAMES FISH.
      1.3     //
:..
/KEEP MINE
INVALID COMMAND
/E
EDTXT ALREADY EXISTS - RESPOND YES TO PURGE OLD AND KEEP NEW
PURGE OLD?Y
>
```

To keep the data entered in EDTXT, you can use the COBEDIT KEEP command. This command is discussed later.

Another example of the EDIT command:

```
>EDIT MODULE1
Previous Edit text was not saved.
OK to clear?  (Y/N)  Y
HP32201A.7.05 EDIT/3000  MON, MAY 21, 1980,  3:57 PM
(C) HEWLETT-PACKARD CO. 1980
WARNING: FORMAT=COBOL VALUES SET FOR LENGTH,RIGHT,FROM,DELTA,FRON
/L ALL
      1       SORT-PARA.
      1.1           SORT SORTFL ON ASCENDING KEY FIRST-KEY
      1.2            INPUT PROCEDURE IS INP-SECTION
      1.3            OUTPUT PROCEDURE IS OUTP-SECTION
      1.4                   THROUGH OUTP-END-SECTION.
```

```
/ADD
     1.5      INP-SECTION.
     1.6            OPEN INPUT FILE-IN.
     1.7            IF IN-REC IS ALPHABETIC
     1.8               THEN RELEASE IN-REC
     1.9            ELSE NEXT SENTENCE.
     2.0            CLOSE FILE-IN.
     2.1      OUTP-SECTION.
     2.2            OPEN OUTPUT FILE-OUT.
     2.3            IF SORT-REC IS NOT NUMERIC
     2.4               THEN RETURN SORTFL RECORD INTO FOR-WRITE
     2.5               WRITE REC-OUT FROM FOR-WRITE;
     2.6            ELSE NEXT SENTENCE.
     2.7      OUTP-END-SECTION.
     2.8            CLOSE FILE-OUT.
     2.9            //
     ....
/E EDTXT ALREADY EXISTS - RESPOND YES TO PURGE OLD AND KEEP NEW
PURGE OLD?Y
>
```

In the above example, the module named MODULE1 was specified when the EDIT command was issued. Note the message immediately following the EDIT command above. This message was issued because the data stored in EDTXT was not kept to the library file before the EDIT command was issued. Thus, since the response to the CLEAR question was Y (yes), EDTXT was cleared, and the records of MODULE1 were copied into it.

Also, although records were copied from MODULE1, the records of MODULE1 are still in the library file. Thus, you may retain them in the library file by issuing a KEEP command for the records in EDTXT, using a different text-name.

As a final example of using the EDIT command, a file created outside of the COBEDIT program is joined to the work space associated with EDTXT.

```
>EDIT
Previous Edit text was not saved.
OK to clear? (Y/N) N
>
>KEEP MODULE4
>EDIT

HP32201A.7.05 EDIT/3000  MON, MAY 21, 1980,  4:05 PM
(C) HEWLETT-PACKARD CO. 1980
WARNING: FORMAT=COBOL VALUES SET FOR LENGTH,RIGHT,FROM,DELTA,FRON
/L ALL.001 /M
MODIFY.001
     R* THIS MODULE IS CREATED BY JOINING THE FILE, FROMEDIT,
* THIS MODULE IS CREATED BY JOINING THE FILE, FROMEDIT,

/A
       .101* TO THE CURRENT WORK FILE.
       .201//
 ...
/JOINQ FROMEDIT
NUMBER OF LINES JOINED =2
/L ALL
       .001* THIS MODULE IS CREATED BY JOINING THE FILE, FROMEDIT,
       .101* TO THE CURRENT WORKFILE.
       .201* THIS LINE AND THE FOLLOWING WERE JOINED TO THE WORK
       .301* FILE FROM THE FILE, FROMEDIT.
/E
EDTXT ALREADY EXITS -  RESPOND YES TO PURGE OLD AND KEEP NEW
PURGE OLD? Y
>
```

EXIT COMMAND

The EXIT command is used to exit the COBEDIT program.

Exit Command Format

E [XIT]

That is, to exit COBEDIT, you must type either EXIT or just an E.

If you have used the EDIT command, and no KEEP command was issued before the EXIT command is executed, the message,

```
Edit text not empty.  OK to clear?
```

is displayed. If you respond with anything except Y or YES, the EXIT command terminates, and COBEDIT remains active. A Y or YES response causes COBEDIT to clear EDTXT, close the currently active library, and cease execution.

If a KEEP command has been performed for the current contents of EDTXT, or if the EDIT command was not used during the current execution of the COBEDIT program, then when the EXIT command is executed, COBEDIT ceases execution with no warning message.

To illustrate the EXIT command:

```
>EDIT

HP32201A.7.05 EDIT/3000 MON MAY 21, 1980,  4:51 PM
(C) HEWLETT-PACKARD CO. 1980
WARNING: FORMAT=COBOL VALUES SET FOR LENGTH,RIGHT,FROM,DELTA,FRONT
/L ALL
       .001
/E
EDTXT ALREADY EXISTS - RESPOND YES TO PURGE OLD AND KEEP NEW
PURGE OLD? Y
>EXIT
Edit text not empty.  OK to clear? Y

END OF PROGRAM
:
```

```
:RUN COBEDIT.PUB.SYS

HP COPYLIB EDITOR  MON, MAY 21, 1980, 5:01 PM
(C) HEWLETT-PACKARD CO. 1980

TYPE ''HELP'' FOR A LIST OF COMMANDS.
>E

END OF PROGRAM
:
:RUN COBEDIT.PUB.SYS

HP COPYLIB EDITOR MON, MAY 21, 1980, 5:02 PM
(C) HEWLETT-PACKARD CO. 1980

TYPE ''HELP'' FOR A LIST OF COMMANDS.
>LIB MYLIB
>LIST ALL
001000$CONTROL SUBPROGRAM                                        MODULE1
001100 IDENTIFICATION DIVISION.                                  MODULE1
001200 PROGRAM-ID.  DUMMY-SUB.                                   MODULE1
005100 WORKING-STORAGE SECTION.                                  MODULE2
005200 01   UNIV-TOTALER    PIC 9(8) COMP-3.                     MODULE2
005300 01   UNIV-ACCUM      PIC 9(8) COMP-3.                     MODULE2
008000        PERFORM TEST-IT.                                   MODULE3
008100        IF RESULTANT IS LESS THAN 2                        MODULE3
008200         PERFORM TEST-FAILED;                              MODULE3
008300        ELSE NEXT SENTENCE.                                MODULE3
>EDIT
HP32201A.7.05 EDIT/3000  MON, MAY 21, 1980,  3:57 PM
(C) HEWLETT-PACKARD CO. 1980
WARNING: FORMAT=COBOL VALUES SET FOR LENGTH,RIGHT,FROM,DELTA,FRON
/L ALL
      .001
/A
      .101* THIS IS TO SHOW WHAT HAPPENS WHEN A KEEP COMMAND
      .201* IS ISSUED BEFORE THE EXIT COMMAND IS USED.
      .301//
...
/E
EDTXT ALREADY EXISTS - RESPOND YES TO PURGE OLD AND KEEP NEW
PURGE OLD? Y
>KEEP MOD4
>E

END OF PROGRAM
:
```

HELP COMMAND

The HELP command lists all commands available in the COBEDIT program, and a brief description of each.

HELP Command Format

HELP

To illustrate the HELP command:

```
>HELP

the following is a list of COBEDIT commands:

BUILD library-name  [ , filesize ]
Create a new library file with name ``library-name''.

COPY
Copy modules into library, as in Build command.


EDIT  [ text-name ]
Activate EDIT/3000 and text in that module of the current
library which contains ``text-name'' in the id field.

EXIT Exit the COBEDIT program.

HELP
Display a list of COBEDIT commands.

KEEP  [ text-name ]
Insert an (edited) module in the current library.

LIBRARY library-name
Designate ``library-name'' as the current library.

LIST  [ text-name ]
      [    ALL    ]
Display all or part of the current library on $STDLIST.
With no parameter, will list the text-names of the current library.

PURGE  { text-name }
       {    ALL    }
Delete a module from the current library.  The ALL
option will purge the entire library.

SHOW
Display an information block for the current library.

:{ MPE Command }
Certain MPE commands may be executed from COBEDIT.
```

KEEP COMMAND

The KEEP command is the means by which you add a module to the currently active library, or replace an already existing module.

KEEP Command Format

<u>KEEP</u> [***text-name***]

Where *text-name* is the name to be used for the module being kept.

If the module to be kept is one which already exists on the file, and you named that module in a previous EDIT command, you need not specify a text-name in the KEEP command. In this case, you are asked if you want to replace the module in the library.

To illustrate:

```
>LIB MYLIB
>LIST ALL
001000$CONTROL SUBPROGRAM                                      MODULE1
001100 IDENTIFICATION DIVISION.                                MODULE1
001200 PROGRAM-ID.  DUMMY-SUB.                                 MODULE1
005100 WORKING-STORAGE SECTION.                                MODULE2
005200 01   UNIV-TOTALER     PIC 9(8) COMP-3.                  MODULE2
005300 01   UNIV-ACCUM       PIC 9(8) COMP-3.                  MODULE2
008000        PERFORM TEST-IT.                                 MODULE3
008100        IF RESULTANT IS LESS THAN 2                      MODULE3
008200         PERFORM TEST-FAILED;                            MODULE3
008300        ELSE NEXT SENTENCE.                              MODULE3
000101* THIS IS TO SHOW WHAT HAPPENS WHEN A KEEP COMMAND       MOD4
000201* IS ISSUED BEFORE THE EXIT COMMAND IS USED.             MOD4
>EDIT MODULE1

HP32201A.7.05 EDIT/3000  MON, MAY 21, 1980,  3:28 PM
(C) HEWLETT-PACKARD CO. 1980
WARNING: FORMAT=COBOL VALUES SET FOR LENGTH,RIGHT,FROM,DELTA,FRON
/L ALL
   1    $CONTROL SUBPROGRAM
   1.1   IDENTIFICATION DIVISION.
   1.2   PROGRAM-ID.  DUMMY-SUB.
/M 1.2
MODIFY   1.2
PROGRAM-ID.  DUMMY-SUB.
             RTEST-KEEP.
PROGRAM-ID.  TEST-KEEP.

/A
   1.3  AUTHOR. MYSELF.
   1.4//
... /E
EDTXT ALREADY EXISTS - RESPOND YES TO PURGE OLD AND KEEP NEW
PURGE OLD? Y
>KEEP
''MODULE1 '' already exists on Library MYLIB.
OK to clear? Y
```

```
>LIST ALL
001000$CONTROL SUBPROGRAM                                  MODULE1
001100 IDENTIFICATION DIVISION.                            MODULE1
001200 PROGRAM-ID.  TEST-KEEP.                             MODULE1
001300 AUTHOR. MYSELF.                                     MODULE1
005100 WORKING-STORAGE SECTION.                            MODULE2
005200 01  UNIV-TOTALER    PIC 9(8) COMP-3.                MODULE2
005300 01  UNIV-ACCUM      PIC 9(8) COMP-3.                MODULE2
008000       PERFORM TEST-IT.                              MODULE3
008100       IF RESULTANT IS LESS THAN 2                   MODULE3
008200        PERFORM TEST-FAILED;                         MODULE3
008300       ELSE NEXT SENTENCE.                           MODULE3
000101* THIS IS TO SHOW WHAT HAPPENS WHEN A KEEP COMMAND   MOD4
000201* IS ISSUED BEFORE THE EXIT COMMAND IS USED.         MOD4
```

Another example:

```
>EDIT

HP32201A.7.05 EDIT/3000  MON, MAY 21, 1980,  3:28 PM
(C) HEWLETT-PACKARD CO. 1980
WARNING: FORMAT=COBOL VALUES SET FOR LENGTH,RIGHT,FROM,DELTA,FRON
/MODIFY
MODIFY    .001

I*THIS MODULE WILL BE ADDED TO MYLIB BY
*THIS MODULE WILL BE ADDED TO MYLIB BY

/A
        .002*USING A TEXT NAME IN THE KEEP COMMAND
        .003//
... /E
EDTXT ALREADY EXISTS - RESPOND YES TO PURGE OLD AND KEEP NEW
PURGE OLD?Y
>KEEP MOD5
>LIST ALL
001000$CONTROL SUBPROGRAM                                  MODULE1
001100 IDENTIFICATION DIVISION.                            MODULE1
001200 PROGRAM-ID.  TEST-KEEP.                             MODULE1
001300 AUTHOR. MYSELF.                                     MODULE1
005100 WORKING-STORAGE SECTION.                            MODULE2
005200 01  UNIV-TOTALER    PIC 9(8) COMP-3.                MODULE2
005300 01  UNIV-ACCUM      PIC 9(8) COMP-3.                MODULE2
008000       PERFORM TEST-IT.                              MODULE3
008100       IF RESULTANT IS LESS THAN 2                   MODULE3
008200        PERFORM TEST-FAILED;                         MODULE3
008300       ELSE NEXT SENTENCE.                           MODULE3
000101* THIS IS TO SHOW WHAT HAPPENS WHEN A KEEP COMMAND   MOD4
000201* IS ISSUED BEFORE THE EXIT COMMAND IS USED.         MOD4
000101*THIS MODULE WILL BE ADDED TO MYLIB BY               MOD5
000201*USING A TEXT NAME IN THE KEEP COMMAND               MOD5
```

Note that if you use the KEEP command without a text-name, and the data in EDTXT was not entered by using text from an already existing module, the message, "Invalid text-name.", is returned. Also, if you have issued a KEEP command, and you have issued no EDIT command since, then when you issue another KEEP command, the message, "Edit file is empty.", is returned.

LIBRARY COMMAND

The LIBRARY command allows you to select the library that you wish to access. When you issue this command, the currently active library is closed, and the specified library is opened and made available.

LIBRARY Command Format

LIBRARY ***library-name***

Where *library-name* is the name of the library file you wish to access. It can be in any group and account. However, if it is in a group or account other than your log-on group and account, it must be released (see the MPE RELEASE command) before you issue the LIBRARY command.

The fully qualified form of a library name is the same as for all MPE files. That is, it is of the form, *filename.group.account,* where *filename* is *library-name, group* is the name of the group in which the file resides, and *account* is the name of the account in which the group resides.

Note that you can specify the name of the currently active library, even though it is already open. This has no effect upon the COBEDIT program.

If you specify no library name in the LIBRARY command, COBEDIT prompts you for one.

When the LIBRARY command executes, it checks to make sure that the file named is a valid library file. If it is not, an appropriate error message is generated by the MPE file system, and in two cases, a COBEDIT error message follows.

The two cases are when there exists no file of the specified name, and when an error occurs while trying to open the file. To illustrate the LIBRARY command:

```
>LIB
Library name?  MYLIB
>SHOW
************************************************************
Library file: MYLIB.MANAGERS.USERS
Text-name: Key file: KMYLIB
************************************************************
>LIB COPYLIB
>SHOW
************************************************************
Library file: COPYLIB.MANAGERS.USERS
Text-name:
Key file: KCOPYLIB
************************************************************
>LIB CLIB.PUB.USERS
SECURITY VIOLATION  (FSERR 93)
FILE CLIB.PUB.USERS NOT OPENED.
>:TELL WENDY.USERS; PLEASE RELEASE FILE CLIB FOR UPDATE
FROM /S21 WENDY.USERS/ IT'S RELEASED NOW
>:TELL WENDY.USERS; THANKS
>LIB CLIB.PUB.USERS
>SHOW
************************************************************
Library file: CLIB.PUB.USERS
Text-name:
Key file: CLIBKEY.KING.USERS
************************************************************
```

In the above examples, the LIBRARY command was used to obtain access to three different files. The first use of the command specified no library. Thus, COBEDIT prompted for one.

The third attempt to use the LIBRARY command failed, since the desired library, CLIB, resides in a group other than the log-on group, MANAGERS. The availability of MPE commands in COBEDIT made it easy to request that the file be released. Once the file was released, there was no problem in obtaining access to it.

The SHOW command was used to show which library file was currently open and available.

As a final example of the LIBRARY command, we attempt to open a non-existent file.

```
>LIBRARY FROTH
NONEXISTENT PERMANENT FILE  (FSERR 52)
FILE FROTH NOT OPENED.
```

LIST COMMAND

The LIST command allows you to list information about your currently active library.

The information available is a list of all module names within the library, or a list of either all or one of the modules in the library. A control Y will terminate the listing.

If you have no library open (that is, you have not built one, or used the LIBRARY command to name one, or you have purged the latest one, and have not opened another), the response to execution of a LIST command with or without parameters is the message, "No library file is open."

LIST Command Format

```
LIST [ text-name ]
     [ ALL       ]
```

Where

text-name is the name of a module in the currently active library.

ALL indicates that all modules in the library are to be listed, beginning with the first module on the file, and proceeding to the last.

If neither *text-name* nor the word ALL is used in the LIST command, only the names of the modules in the library are returned.

To illustrate the LIST command:

```
>LIBRARY MYLIB
>LIST
Text-names of modules in MYLIB:
    MODULE1
    MODULE2
    MODULE3
    MOD4
    MOD5
>LIST MODULE2
Text-name MODULE2
005100 WORKING-STORAGE SECTION.
005200 01  UNIV-TOTALER   PIC 9(8) COMP-3.
005300 01  UNIV-ACCUM     PIC 9(8) COMP-3.
```

```
>LIST ALL
001000$CONTROL SUBPROGRAM                                    MODULE1
001100 IDENTIFICATION DIVISION.                              MODULE1
001200 PROGRAM-ID.  TEST-KEEP.                               MODULE1
001300 AUTHOR. MYSELF.                                       MODULE1
005100 WORKING-STORAGE SECTION.                              MODULE2
005200 01  UNIV-TOTALER    PIC 9(8) COMP-3.                  MODULE2
005300 01  UNIV-ACCUM      PIC 9(8) COMP-3.                  MODULE2
008000       PERFORM TEST-IT.                                MODULE3
008100       IF RESULTANT IS LESS THAN 2                     MODULE3
008200        PERFORM TEST-FAILED;                           MODULE3
008300       ELSE NEXT SENTENCE.                             MODULE3
000101* THIS IS TO SHOW WHAT HAPPENS WHEN A KEEP COMMAND     MOD4
000201* IS ISSUED BEFORE THE EXIT COMMAND IS USED.           MOD4
000101*THIS MODULE WILL BE ADDED TO MYLIB BY                 MOD5
000201*USING A TEXT NAME IN THE KEEP COMMAND                 MOD5
```

PURGE COMMAND

The PURGE command allows you to purge either a single module from your currently active library, or the entire library.

If you choose to purge the entire library, it no longer exists after successful execution of the PURGE command.

PURGE Command Format

```
PURGE { text-name }
      { ALL       }
```

Where

text-name is the name of a module to be purged from the currently active library. This is the module to be purged.

ALL indicates that you want the entire library, including its key file, to be purged.

If you specify ALL in the PURGE command, COBEDIT double checks to be sure that you want the entire library file purged. It does so by asking the question,

```
Is it OK to purge library library-name?
```

Where *library-name* is the name of your currently active library.

If you respond with anything other than Y or YES, no purging takes place, and the current library file remains active. If you do respond with an affirmative response, the message,

```
COBOL Library file library-name purged.
```

is returned.

To illustrate the PURGE command, we use a file called MESSEDUP. This library contains only two modules. The first is a module copied into it at the time MESSEDUP was created. This module has no text-name associated with it. Thus, it is not accessable.

```
>LIB MESSEDUP
>LIST ALL
000101*THESE RECORDS WERE COPIED INTO MESSEDUP FROM AN ASCII
000201*FILE, BUT SINCE THE COBEDIT PROGRAM THOUGHT IT WAS IN
000301*COPYLIB FORMAT, IT DID NOT ASSIGN A TEXT-NAME.
001000$CONTROL USLINIT                                          M1
002000 IDENTIFICATION DIVISION.                                 M1
003000 DATA DIVISON.                                            M1
004000 PROCEDURE DIVISION.                                      M1
>PURGE M1
>LIST ALL
000101*THESE RECORDS WERE COPIED INTO MESSEDUP FROM AN ASCII
000201*FILE, BUT SINCE THE COBEDIT PROGRAM THOUGHT IT WAS IN
000301*COPYLIB FORMAT, IT DID NOT ASSIGN A TEXT-NAME.
>PURGE ALL
Is it OK to purge library MESSEDUP?  YES
COBOL Library file MESSEDUP purged.
>LIST ALL
No library file is open.
```

SHOW COMMAND

The SHOW command can be used to find out the name of the currently active library, its key file, and the name of the module which was most recently accessed by COBEDIT.

If no library is open, the result of execution of the SHOW command is the message, "No library is open."

SHOW Command Format

SHOW

To illustrate the SHOW command:

```
 :RUN COBEDIT.PUB.SYS

HP COPYLIB EDITOR  WED, MAY 23, 1980, 5:42 PM
(C) HEWLETT-PACKARD CO. 1980

TYPE ''HELP'' FOR A LIST OF COMMANDS.
>SHOW
No library is open.
>LIB MYLIB
>SHOW
************************************************************
Library name:  MYLIB.USERS.MANAGERS
Text-name:
Key file:  KMYLIB
************************************************************
>EDIT MODS
HP32201A.7.05 EDIT/3000 MON MAY 21, 1980,  4:51 PM
(C) HEWLETT-PACKARD CO. 1980
WARNING: FORMAT=COBOL VALUES SET FOR LENGTH,RIGHT,FROM,DELTA,FROM
/L ALL
  .101*THIS MODULE WILL BE ADDED TO MYLIB BY
  .201*USING A TEXT NAME IN THE KEEP COMMAND
/A
  .301*THIS LINE IS ADDED TO SHOW THE EFFECT OF USING
  .401*THE SHOW COMMAND WHEN A MODULE HAS BEEN ACCESSED.
  .501//
  ... /E
EDTXT ALREADY EXISTS - RESPOND YES TO PURGE OLD AND KEEP NEW
PURGE OLD? Y
>SHOW
************************************************************
Library name:  MYLIB.USERS.MANAGERS
Text-name:  MODS
Key file:  KMYLIB
************************************************************
>KEEP MOD6
>EXIT

END OF PROGRAM :
```

COPY STATEMENT

The COPY statement is the method by which source records in a COBOL library are copied into your source program.

This statement may appear anywhere in your source program, from the Identification Division to the end of the Procedure Division. Aside from allowing you to copy modules into your source file, it also allows you to replace occurrences of a string of text, an identifier, literal, or word appearing in the module being copied.

COPY Statement Format

```
COPY text-name [ { OF } library-name ]  [ NOLIST ]
                 { IN }

[            {   { ==pseudo-text-1== }        { ==pseudo-text-2== } }     ]
[ REPLACING  { , { identifier-1      }   BY   { identifier-2      } } ... ]
[            {   { literal-1         }        { literal-2         } }     ]
[            {   { word-1            }        { word-2            } }     ]
```

Where

text-name is the name of the module to be copied into your source program.

library-name is a name of from one to eight alphanumeric characters, the first of which must be alphabetic. This name is used to specify the library in which the module to be copied resides. If the library is in a group or account other than your log-on group and account, you must use an MPE FILE command to equate the *library-name* used in the COPY statement to the library file. Also, if the name used in the COPY statement is different from the name of the library file you wish to use, a FILE command can be used to equate the *library-name* of the COPY statement to the name of the library file.

Library-name must be used when you have more than one COBOL library in your log-on group. If *library-name* is not used, the COBOL compiler assumes that *library-name* is COPYLIB. You may use the MPE :FILE command to equate a library file to COPYLIB if you wish to use the library file in the COPY statement, but do not wish to specify it by name.

NOLIST if used, indicates that you do not want the text of the module named by *text-name* to be included in the list file created by the compilation process.

==pseudo-text-1== is a sequence of character strings and/or separators delimited on either end by double equal signs. It may consist of any text you wish, except that it must not consist of null text (that is, ====), all spaces, or all comment lines.

==pseudo-text-2== is a sequence of character strings and/or separators delimited on either end by double equal signs. It may be any text you wish, including null text; that is, it may be of the form, ====.

Note that character strings within *pseudo-text-1* and *pseudo-text-2* may be continued. However, both equal signs forming the delimiters must be on the same line.

literal-1 and ***literal-2*** may each be any COBOL literal.

identifier-1 and ***identifier-2*** may each be any COBOL identifier, and may be qualified.

word-1 and ***word-2*** may each be any single COBOL word.

A COPY statement may appear in a source program anywhere a character string or a separator may occur, except that a COPY statement may not appear within another COPY statement. When a COPY statement is used, it must be preceded by a space, and terminated by a period.

COPY statements are executed before source lines associated with them are sent to the compiler. Thus, only the lines of the copied module, including any replaced words, identifiers, and so forth, are sent to the compiler. The COPY statement itself is not.

Note that although the COPY statement is not sent to the compiler, it appears in the listing sent to the list file, along with the records of the copied module (unless NOLIST was specified).

If the REPLACING phrase is not used, the module is copied exactly as it appears in the library, including its sequence field, and text-name, which appears in columns 73 through 80 of each record in the module.

REPLACING Phrase

To facilitate the following discussion, the REPLACING phrase is rewritten as shown below.

```
COPY text-name [ {OF} library-name ] [ NOLIST ]
                 {IN}
    [REPLACING text-to-replace BY replace-text]
```

Before the comparison to determine which text, if any, is to be replaced in the copied module, spaces, commas, and semicolons to the left of the first word in the records of the module are moved into the source program. The first word of the record in a module is the first part of the module to be compared.

Starting with the left most word in the module being copied, and the first text-to-replace specifed in the REPLACING phrase, text-to-replace is compared to an equivalent number of contiguous words in the module.

Text-to-replace matches the text in the module if and only if each character in text-to-replace equals the character in the corresponding position of the text in the module.

For purposes of matching, each occurrence of a separator comma, or semicolon in text-to-replace or in the text of the module is considered to be a single space except when text-to-replace consists solely of either a separator comma or semicolon, in which case it participates in the match as a character. Each sequence of one or more spaces is considered to be a single space.

If no match occurs, the comparison is repeated with each next successive text-to-replace, if any, in the REPLACING phrase until either a match is found or there is no next successive text-to-replace.

When all occurrences of text-to-replace have been compared to the text in the module and no match has occurred, the left most word in the text in the module is copied into your source program.

The next word of the text in the module is then considered as the left most word of the text in the module, and the comparison cycle is repeated, starting with the first text-to-replace in the REPLACING phrase.

Whenever a match occurs between text-to-replace and text in the module, replace-text is placed into the source program. The word immediately to the right of the text in the module which participated in the match is then considered as the left most word of the text in the module, and comparison begins again, starting with the first text-to-replace in the REPLACING phrase.

The comparison operation continues until the rightmost word in the last line of the module has either participated in a match or been considered as the left most word of text in the module and has participated in a complete comparison cycle.

A comment line in text-to-replace or in the module is interpreted, for purposes of matching, as a single space. Comment lines appearing in replace-text and in the module are copied into the source program unchanged.

The syntactic correctness of the lines in a module cannot be independently determined. Nor can the syntactic correctness of the entire COBOL source program until all COPY statements have been completely processed.

To illustrate the COPY statement, we use a module named RITESTUF in a library called UTIL. UTIL resides in the MANAGERS group of the USERS account.

Assuming that the COBOL program is being compiled in an account called COBTEAM, and assuming that the UTIL library has been released, the COPY statement, the module, RITESTUF, the necessary file equation, and the results of the COPY statement are shown on the following page.

```
:RUN COBEDIT.PUB.SYS

HP COPYLIB EDITOR  THUR, MAY 24, 1980, 2:07 PM
(C) HEWLETT-PACKARD CO. 1980

TYPE ''HELP'' FOR A LIST OF COMMANDS.
>LIBRARY UTIL.MANAGERS.USERS
>LIST RITESTUF
001000  WRITE-ROUTINE.                                       RITESTUF
001100        OPEN OUTPUT FLUNKY.                            RITESTUF
002000        WRITE DUMMY  BEF-AFT ADVANCING X LINES;        RITESTUF
003000        AT EOP IMP-STAT.                               RITESTUF
>E

END OF PROGRAM :
```

The COPY statement appears as follows:

```
100000 PROCEDURE DIVISION.
                   .
                   .
                   .
102100       COPY RITESTUF OF FINDIT
102200            REPLACING "FLUNKY" BY FILEOUT,
102300            ==DUMMY== BY ==RECOUT==,
102310            ==X== BY ==1==,
102400            ==BEF-AFT== BY ==BEFORE==,
102500            ==IMP-STAT== BY ==PERFORM PAGER==.
102600        CLOSE FILEOUT.
                   .
                   .
                   .
```

```
:FILE FINDIT=UTIL.MANAGERS.USERS
:COBOL TESTPROG

PAGE 0001    HEWLETT-PACKARD 32233A.00.00  COBOL II/3000

00001   001000 IDENTIFICATION DIVISION.
00002   002000 PROGRAM-ID.  TESTPROG
                    .
                    .
                    .
00503   100000 PROCEDURE DIVISION.
                    .
                    .
                    .
00579   102100      COPY RITESTUF OF FINDIT
00580   102200           REPLACING ``FLUNKY'' BY FILEOUT,
00581   102300           ==DUMMY== BY ==RECOUT==,
00582   102310           ==X== BY ==1==,
00583   102400           ==BEF-AFT== BY ==BEFORE==
00584   102500           ==IMP-STAT== BY ==PERFORM PAGER==.
00585   001000  WRITE-ROUTINE.                                  RITESTUF
00586   001100       OPEN OUTPUT FILEOUT.                       RITESTUF
00587   002000       WRITE RECOUT  BEFORE ADVANCING 1 LINES;    RITESTUF
00588   003000       AT EOP PERFORM PAGER.                      RITESTUF
00589   102600       CLOSE FILEOUT.
                    .
                    .
                    .
```

An equivalent way of performing the same copying operation is shown below, using the MPE :FILE command to equate UTIL to COPYLIB, and changing the COPY statement.

:FILE COPYLIB=UTIL

The COPY statement could then written as follows:

```
COPY RITESTUF
          REPLACING "FLUNKY" BY FILEOUT,
          ==DUMMY== BY ==RECOUT==,
          ==X== BY ==1==,
          ==BEF-AFT== BY ==BEFORE==
          ==IMP-STAT== BY ==PERFORM PAGER==.
```

Since no library-name is specified in the COPY statement, the COBOL compiler assumes that the file COPYLIB is to be used. However, the :FILE command above equates COPYLIB to UTIL. Thus, UTIL is the file from which RITESTUF is copied.

PREPROCESSOR AND MODIFICATION COMMANDS

APPENDIX

A

Hewlett-Packard COBOL II/3000 contains special language independent compile-time features designed to simplify the compilation process. These features allow you to modify input text, set certain compilation options, or control the contents of the list output.

All these options are implemented by a simple compile-time language, consisting of commands, statements, variables or identifiers, and macro calls for text substitution.

Types Of Processes

The compile-time processes are divided into three types. These types are edit processing, source input modification, and list and compilation options.

The first two processes mentioned in the above paragraph are independent of the compilation process. In effect, they constitute a "first pass" on your source program. The output of these preprocessor functions is released to the COBOL compiler. This output is referred to as the expanded source program.

The editing process merges lines from an optional text file into your master file, and provides other functions such as deletion and replacement of lines of the master file. The end product of the editing process is a single input stream.

The MPE COBOL command, as well as the $EDIT and $INCLUDE preprocessor commands, are used to perform these editing operations.

The source input modification process uses macros to modify your source code. This is done by using the $DEFINE preprocessor command to define the macros, and then placing the macro name in your source code. When the source code is compiled, the definition of the macro replaces the macro name.

List and compilation options are provided to allow you to obtain compiler listings, source listings of the expanded source records, a symbol table map, the object code, a cross reference listing of symbols and labels used in the source program, and a listing of verbs appearing in the Procedure Division (as well as the line numbers in which they appear in the source code and their PB relative addresses). These options are available through use of the $CONTROL preprocessor command.

Compilation options include warnings of possible error conditions, limiting the number of compilation errors before halting compilation, initilizing the USL file, identifying the compiled code as being a subprogram (dynamic or non-dynamic), enabling the MPE debug facility, setting bounds checking on table indices and subscripts, and flagging of non-ANSI COBOL '74 HP features in the source program. These options are also available through use of the $CONTROL preprocessor command.

Preprocessor Programming Language

The preprocessor programming language is a simple language consisting of variables and commands. All of the processes are performed at compile-time; none are performed at run time.

All preprocessor commands are listed below.

COMMAND	DISCUSSION
$COMMENT	below
$DEFINE $PREPROCESSOR	MACRO definition and use
$IF $SET	conditional compilation
$INCLUDE $EDIT	file insertion, merging and editing operations
$PAGE $TITLE $CONTROL	compiler dependent options

The $DEFINE and $PREPROCESSOR commands are discussed under the heading, MACRO DEFINITION AND USE; $IF and $SET are discussed under CONDITIONAL COMPILATION; $INCLUDE and $EDIT are discussed under FILE INSERTION, AND MERGING AND EDITING OPERATIONS, and the remainder of the preprocessor commands are discussed under COMPILER DEPENDENT OPTIONS.

Since the $COMMENT preprocessor command can be used anywhere in your source program, and does not belong to any of the specific groups mentioned above, it is discussed now.

$COMMENT Preprocessor Command

Format

$COMMENT [*comment-text*]

Where *comment-text* is a string containing anything you wish to enter. Comment-text requires no delimiters; it ends at the end of the line in which the $COMMENT command is issued unless a continuation character is used.

To illustrate:

```
000100$COMMENT THIS LINE IS AN EXAMPLE OF THE $COMMENT PRE- &
000200$ PROCESSOR COMMAND.
```

General Format - Preprocessor Commands

Format

$[$] ***commandname*** **[** ***parameterlist*** **]**

Where

commandname is one of the command names shown in the list above.

parameterlist is a list of parameters for a given preprocessor command. The specific parameters (if any) allowed for a given preprocessor command are listed later in this section where the command is discussed. A list of parameters in a command must be separated from the command by one or more spaces, and each parameter specified must be separated from any succeeding parameter by a comma optionally followed or preceded by one or more spaces.

The required dollar sign is used to identify a preprocessor command. This symbol must appear in the first column of a line of source code, immediately following the sequence number field.

The second, optional, dollar sign is used to indicate that the preprocessor command of which it is a part is to be executed, but is not to be copied (in a merging process) to the new file.

A preprocessor command can be continued to the next line by using a continuation character as the last nonblank character in the line in which the command is written. The default continuation character is the ampersand (&).

A continuation character may be used anywhere in a preprocessor command in which a blank can be used and will not change the effect of the command.

For example,

```
001000$CONTROL USLINIT,&
002000$MAP,LIST
```

is a permissable use of the continuation character.

When you continue a preprocessor command to another line, the new line must contain a single dollar sign in the first column following the sequence number field of that line (see the example above).

The effect of using a continuation character in a preprocessor command is equivalent to replacing the continuation character and subsequent dollar sign with a blank, and concatenating the two lines.

Thus,

```
001000$CONTROL USLINIT,&
002000$MAP,LIST
```

is equivalent to

```
001000$CONTROL USLINIT,  MAP,LIST
```

Note that continuation lines of a preprocessor command are not copied to a new file during a merging operation if the preprocessor command begins with two dollar signs.

Thus, for example, the preprocessor command,

```
003000$$EDIT VOID=005000, &
004000$ SEQNUM=001000
```

is executed but neither line 003000 nor line 004000 are copied to the new file created by the merging process.

MACRO DEFINITION AND USE

One of the most powerful facilities of the preprocessor is the macro processor. This processor associates a macro name with a string of text. When the macro name is used by itself in a COBOL sentence, the preprocessor sends the definition of the macro to the compiler.

If a macro is defined in a COBOL source program, but the macro is never called (that is, is not named anywhere else in the program except in its definition), it has no effect upon the source program.

Macros may have either no parameters at all, or up to nine formal parameters. If formal parameters are used in the definition, actual parameters are supplied to replace them when the macro is called in the source program.

$DEFINE Command

The $DEFINE command is used to define a macro. By "define a macro," we mean to associate a macro name with a string of text.

The $DEFINE command may redefine a previously defined macro. However, all macro definitions are global. Thus, if you use the $DEFINE command to redefine a macro, the old definition is lost, and can only be recovered by issuing another $DEFINE command in which the old definition is repeated.

$DEFINE Preprocessor Command Format

$DEFINE ***macro-name*=[*string-text*]#**

Where

macro-name is the name of the macro being defined, and consists of an initial non-alphanumeric character (default is the percent sign, %), followed by an alphabetic character, followed by zero or more alphanumeric characters.

The macro name may be any number of characters long, but only the first fifteen are recognized by the preprocessor. Note that care must be taken to assure uniqueness of such names.

string-text can be any text you choose. However, since this text is sent to the compiler, it must be a valid COBOL statement or sentence, with one exception. This exception is the use of formal parameters in the *string-text*. Formal parameters are discussed below.

Note that *string-text* is delimited by an equal sign and a pound sign. The pound sign is the delimiter of the entire definition. This is a default delimiter, and can be changed by the $PREPROCESSOR command.

Nested $DEFINE commands and recursive macros can be used; however, you must take care in using recursive macros, as there is no specific method for terminating the macro call sequence when used in this manner.

If a continuation character is used in a macro definition, it is assumed to be a part of the macro definition, and not a continuation character.

To illustrate:

```
000100 $DEFINE %INCA=ADD 1 TO ALPHA-COUNTER;
000200       ADD ALPHA-COUNTER TO BETA-COUNTER.
000300*      INCREMENT THE VALUES OF COUNTERS ALPHA-COUNTER &
000400*      BETA-COUNTER.#
```

In the above example, the entire definition of the macro, %INCA, is the set of sentences,

```
000100                 ADD 1 TO ALPHA-COUNTER;
000200       ADD ALPHA-COUNTER TO BETA-COUNTER.
000300*      INCREMENT THE VALUES OF COUNTERS ALPHA-COUNTER &
000400*      BETA-COUNTER.
```

Note that the dollar symbol is not needed in continuation lines of a macro definition. The only time that a dollar sign would appear in the first column after the sequence number field is when the macro definition contains preprocessor commands.

When the $DEFINE command is processed, the *string-text* which makes up the definition of the macro is stored exactly as it appears in the command. All end-of-line markers and sequence numbers are saved. The only exception to this rule is that if all characters between the equal sign in the macro definition and the end of the text line in which the definition begins are blank, then only the end-of-line marker is saved, and the macro definition begins in the first column of the next line. This allows you to control the initial column of the macro definition without worrying about the column position of the macro name when it is called. To illustrate this,

Macro definition:

```
000300$DEFINE %ADDIT=MOVE SPACES TO DISPLY-ITM.#
000400                 .
                       .
                       .
```

Macro call:

```
001200       DISPLAY DISPLY-ITM; %ADDIT
                       .
                       .
                       .
```

Expanded source code:

```
001200       DISPLAY DISPLY-ITM; MOVE SPACES TO DISPLY-ITM.
                       .
                       .
                       .
```

In the above example, since the macro definition starts on the same line as the macro name, the definition of the macro replaces the macro call starting in exactly the same column as the macro call.

Another example:

```
000100$DEFINE %CHECKIF=
000200       IF STAT-ITEM EQUAL "10"
000300          THEN PERFORM STATUS-REPORT
000400       ELSE NEXT SENTENCE.#
```

In this case, since the characters between the equal sign and the end of line 000100 are blanks, the macro definition begins on line 000200.

Thus, when the macro call to %CHECKIF is made, the expanded source appears as shown below.

Macro call:

```
002400       WRITE FILE-OUT. %CHECKIF
002500            .
                  .
                  .
```

Replacement:

```
002400       WRITE FILE-OUT.
000200       IF STAT-ITEM EQUAL "10"
000300          THEN PERFORM STATUS-REPORT
000400       ELSE NEXT SENTENCE.
```

Formal Parameters

A macro definition may contain up to nine formal parameters. A formal parameter is designated by an exclamation point followed immediately by an integer from the range 1 to 9.

Formal parameters in a macro definition are replaced by values you assign when you call the macro in your source program. To illustrate:

Source program:

```
000100$DEFINE %PERFORM=
000200        PERFORM !1
000300           VARYING !2 FROM !3 BY !4
000400           UNTIL !5.#
                .
                .
                .
001200    %PERFORM(CHEK-PARA#,CTROL#,INIT#,OFFSET#,A = B#)
                .
                .
                .
```

Expanded source program:

```
                .
                .
                .
000200        PERFORM CHEK-PARA
000300           VARYING CTROL FROM INIT BY OFFSET
000400           UNTIL A = B.
                .
                .
                .
```

MACRO Calls

There are two different forms of a macro call:

macro-name

and

macro-name(p1#, p2#,**p3#,*****..., pn#)***

Where

macro-name	is the name of a macro which has been previously defined in the source program, using a $DEFINE command.
p1, p2, and so forth	are the actual parameters. Each of *p1, p2,* and so on may be either a null character, or any combination of characters and numerals, including spaces. Each actual parameter begins with the first character after a preceding comma (except *p1,* which begins after the left parenthesis), and ends with the pound sign. If no characters are specified for an actual parameter (that is, if an actual parameter is specified by ",#"), then a null string replaces its corresponding formal parameter in the macro definition.

Note from the above format that there can be no intervening spaces between the end of the macro name and the left parenthesis of the actual parameter list.

The first method of calling a macro is used when the macro definition has no formal parameters.

The second method must be used when formal parameters are specified in the definition of the macro.

When a macro name is encountered in a source program, it is deleted, and the associated macro definition is sent to the compiler in place of the name. Any formal parameters are replaced by actual parameters listed in the macro call.

With two exceptions, macro names are replaced wherever they occur in the source program, including quoted strings. Macro names are not expanded 1) in a comment, unless the comment itself is found in a macro, and 2) in list and compilation preprocessor commands (such as $CONTROL), where they are not recognized.

RELATIONSHIP OF FORMAL PARAMETERS TO ACTUAL PARAMETERS

The numeric value of a formal parameter determines which actual parameter in the macro call is to replace it.

That is, for a formal parameter, !*n* (where *n* is 1 through 9), the nth actual parameter from the left in the macro call replaces !*n*.

To illustrate:

Macro definition:

```
$DEFINE %OPENSTATE=
        OPEN INPUT !1.
        DISPLAY !2,!3.#
```

Macro call:

```
%OPENSTATE(FILE-IN#,"FILE STATISTICS"#,   OPEN-STATS#)
```

Replacement:

```
        OPEN INPUT FILE-IN.
        DISPLAY "FILE STATISTICS",   OPEN-STATS.
```

Macro definition:

```
090000$DEFINE %COMPUTESUM=     COMPUTE !3 + !2 = !1.#
```

Macro call:

```
091000 %COMPUTESUM(INCREMEN#,OFFSETTER#,7#)
```

Replacement:

```
091000      COMPUTE 7 + OFFSETTER = INCREMEN.
```

In the second example above, !3 is replaced by the third parameter in the macro call, which is 7; !2 is replaced by the second parameter, OFFSETTER, and !1 is replaced by the first parameter in the call, INCREMEN.

For a given macro definition, if there is a formal parameter, !*n*, and if there are less than *n* actual parameters in the macro call, then any formal parameter whose numeric value is greater than the number of parameters in the macro call is ignored.

To illustrate:

Macro definition:

```
001100$DEFINE %SHOWIT=      DISPLAY !1,!2,!3,!5.#
```

Macro call:

```
002500 %SHOWIT("A"#," WORD "#,"IS "#,"MISSING"#)
```

Replacement:

```
002500        DISPLAY "A"," WORD ","IS ", .
```

When you specify a formal parameter in a macro definition, you may choose not to use it in the macro call. This is accomplished by entering only a comma and a pound sign in the appropriate position within the macro call.

For example:

Macro definition:

```
000100$DEFINE %INIT'STUFF=
000200 IDENTIFICATION DIVISION.
000300 PROGRAM-ID. !1
000400 AUTHOR. !2
000500 DATE-COMPILED. !3
```

Macro call:

```
001000 %INIT'STUFF(MACRO-TEST.#,#,#)
```

Replacement:

```
000200 IDENTIFICATION DIVISION.
000300 PROGRAM-ID. MACRO-TEST.
000400 AUTHOR.
000500 DATE-COMPILED.
```

The last two actual parameters were specified by ",#", thus, when the replacement code was sent to the compiler, no author name or compile date was supplied.

Note that the format for a macro definition assures that the initial column of each line in the macro definition will map onto the same column when the macro definition is inserted into the source program at macro call time. This could cause a wraparound of the replacement text when actual parameters are substituted in the definition. However, to insure that this does not happen, blanks are removed from the executable text field to make it the correct size. If there are no trailing blanks to remove, the overflow portion of the executable text field is used to create a new record on the next line. The sequence number field is left blank for this new record.

$PREPROCESSOR Command

The $PREPROCESSOR command allows you to change the default characters used in macro definitions and names. The default characters are:

- # which is used to delimit string text in a macro definition;
- % which is used as the initial character in a macro name;
- ! which is used as the first character of a formal parameter in macro definitions.

To specify a different character to be used in place of one of these, you must use the $PREPROCESSOR command in the following format:

$PREPROCESSOR ***parameter=subparameter*** [, ***parameter=subparameter***]

[, ***parameter=subparameter***]

Where

subparameter is the character to be used in replacing the currently used initial character or delimiter.

parameter is one of the keywords shown below. Each may be used only once in a given $PREPROCESSOR command.

KEYCHAR specifies that the initial character of all macro names is to be subparameter.

DELIMITER specifies that the delimiting character in a macro string text is to be subparameter.

PARMCHAR specifies that the initial character of all formal parameters in macro definitions is to be subparameter.

To illustrate the $PREPROCESSOR command:

```
000100$PREPROCESSOR KEYCHAR=~,DELIMITER=^,PARMCHAR=?
000200$DEFINE ~MOVEIT=
000300       MOVE ?1 TO ?2.^
```

Note that care must be taken when you redefine the initial characters and the string text delimiter, since there may be cases when you use one of the newly defined characters in your string text as part of the string text itself, and not as a delimiter or an initial character.

CONDITIONAL COMPILATION

Usually, when you compile a source file, you want the entire program compiled. However, there may be occassions when you want only part of the program compiled. Conditional compilation, that is, compilation of source code contingent upon whether a switch is on or off, is accomplished using the $SET and $IF preprocessor commands.

$SET Command

The $SET preprocessor command may be used to turn ten software switches on or off. The ten software switches are of the form, Xn, where n is an integer from the range 0 through 9.

$SET Command Format

```
$SET [ Xn= { ON  } [ , Xr= { ON  } ] ]
           { OFF }          { OFF }
```

Where X*n* and X*r* are software switches as described above.

Initially, all software switches are set to OFF.

A $SET command may appear anywhere in the source text. If used without parameters, that is, in the form $SET, it sets all switches to OFF.

$IF Command

THE $IF command interrogates any of the ten software switches. If the condition specified in the $IF command is true, source records are sent to the compiler, beginning with the first one following the $IF command, and continuing until another $IF command is encountered which is false.

If the condition specified by the $IF command is false, no source records are sent to the compiler until a $IF command which is true is encountered.

$IF Command Format

```
$IF [ Xn= { OFF } ]
          { ON  }
```

Where X*n* is a software switch as described under the $SET command in the preceding paragraphs.

The $IF command may 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 without a parameter, it has the same effect as a $IF command whose condition is true. That is, the text following the command is compiled, and any previous $IF command is canceled.

Note that the merging of a text and master source file, and copying of this merged file to a new file is unaffected by $IF commands. Also, the $EDIT, $PAGE, and $TITLE commands are executed even when they appear in a portion of source code that is not to be sent to the compiler. However, all other preprocessor commands are ignored in such a portion of source code.

If you wish to list source records which are not compiled, you must use the CONTROL preprocessor command, specifying the MIXED parameter.

An example of using the $SET and $IF preprocessor commands is shown below.

```
$SET X1=ON, X3=ON

                   .
                   .
                   .

$IF X1=ON
$COMMENT SINCE X1 IS ON, CONTINUE SENDING RECORDS TO THE &
$        COMPILER.

                   .
                   .
                   .

$IF X3=OFF
$COMMENT THIS $IF COMMAND CANCELS THE PRECEDING ONE.  SINCE &
$        X3 IS SET TO "ON", DO NOT SEND THE FOLLOWING RECORDS &
$        TO THE COMPILER.

                   .
                   .
                   .

$IF
$COMMENT  PREVIOUS $IF CONDITION IS TERMINATED, AND COMPILATION &
$         RESUMES.
```

FILE INSERTION, AND MERGING AND EDITING OPERATIONS

There are essentially two types of file merging functions available to you through the preprocessor. The first uses the preprocessor command, $INCLUDE. The second merging function is done using any of the MPE COBOL commands (:COBOL, :COBOLGO, and :COBOLPREP), and optionally, the $EDIT preprocessor command.

$INCLUDE Command

The $INCLUDE command allows you to specify an entire file to be sent, line by line, to the compiler as part of your source file.

$INCLUDE Command Format

$INCLUDE ***filename***

Where *filename* is the name of the file whose records are to be sent to the compiler.

$INCLUDE commands may be nested. That is, the file which is being included may itself have a $IN-CLUDE command in it. This nesting may go to a depth of ten.

When the $INCLUDE command is encountered in a source file, the following actions take place:

1. The file named in the $INCLUDE command is opened.
2. This file then becomes the input file. Each line of the file is processed and the results are sent to the compiler. If sequence numbers exist for the lines (records) of the file, they are preserved.
3. When the end of file is reached for the included file, it is closed.
4. The next line in the original source code is processed and sent to the compiler.

Note that the $INCLUDE command has no effect upon the text, master, or new files named in the COBOL, COBOLGO, or COBOLPREP commands. However, the $INCLUDE command is not sent to the list file. Instead, the included file will be sent to the list file resulting from the compilation process.

To illustrate the $INCLUDE command:

```
000100$INCLUDE INITFILE
001000 ENVIRONMENT DIVISION.
001100 CONFIGURATION SECTION.
                .
                .
                .
```

If INITFILE contains the records,

```
000200$CONTROL SUBPROGRAM
000300 IDENTIFICATION DIVISION.
000400 PROGRAM-ID. SUBDUMMY.
000500 AUTHOR. JPW.
000600 INSTALLATION. GSD.
000700 DATE-WRITTEN. DUMMY-WRITTEN-05/08/80.
000800 DATE-COMPILED. WHEN-USED.
000900 SECURITY. NONE-ON-DUMMY.
```

Then when the $INCLUDE statement above is executed, the results are as follows.

```
000200$CONTROL SUBPROGRAM
000300 IDENTIFICATION DIVISION.
000400 PROGRAM-ID. SUBDUMMY.
000500 AUTHOR. JPW.
000600 INSTALLATION. GSD.
000700 DATE-WRITTEN. DUMMY-WRITTEN-05/08/80.
000800 DATE-COMPILED. WHEN-USED.
000900 SECURITY. NONE-ON-DUMMY.
001000 ENVIRONMENT DIVISION.
001100 CONFIGURATION SECTION.
                .
                .
                .
```

Merging Files And The $EDIT Command

Merging and editing operations are done prior to other preprocessor functions. The editing and merging operations available to you at compile-time are as follows:

1. Merge corrections or additional source text in the textfile with an existing source program (master file) to produce a new source program and commands.
2. Omit sections of the old source program during merging.
3. Check source-record sequence numbers for ascending order.

Merging Files

Merging is done at compile time by using the textfile, masterfile, and newfile parameters of the MPE COBOL commands.

These commands are defined and discussed in detail in the MPE Commands Reference Manual, part number 30000-90009; however, they are shown below to help in clarifying the following discussion.

:COBOL [***textfile***] [, [***uslfile***] [, [***listfile***] [, [***masterfile***] [, ***newfile***]]]]

:COBOLGO [***textfile***] [, [***listfile***] [, [***masterfile***] [, ***newfile***]]]

:COBOLPREP [***textfile***] [, [***progfile***] [, [***listfile***] [, [***masterfile***]

, [***newfile***]]]]

There are two ways in which you may specify a text, master, and new file in one of the above commands.

The first is simply to use the names of the text, master, and new file in the commands. For example, if MFILE is your master file, TFILE is your text file, and NFILE is your new file, then the command shown below causes TFILE to be merged with MFILE, and places the merged files into NFILE.

```
:COBOL TFILE,,,MFILE,NFILE
```

The second way is to equate the special formal file designators COBTEXT, COBMAST, and COBNEW to TFILE, MFILE, and NFILE before issuing the command. This allows you to avoid specifying the files by name in the command you issue. To illustrate:

```
:FILE COBTEXT=TFILE
:FILE COBMAST=MFILE
:FILE COBNEW=NFILE

:RUN COBOL.PUB.SYS;PARM=25
```

The second method shown above has exactly the same effect as the first. However, if you wish to use these same files again in another COBOL command (during the same session) the second method proves useful since the FILE commands stay in effect until you either equate the formal file designators to different files, or you end your current session.

Prior to merging, the records in both textfile and masterfile must be arranged in ascending order according to the values in their sequence fields, or sequence fields may be blank.

The order of sequencing is based on the ASCII collating sequence.

The merging operation is based on nonblank ascending sequence fields. During merging, nonblank sequence fields of 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 your list file (if you do not specify a list file in the command which initiates merging, the default is your terminal or the line printer, depending upon whether you issued the command in a session or a job, respectively).

Blank sequence fields are never considered out of sequence; they are assumed to have the same sequence value as the last preceding record which contains a nonblank sequence value. The sequence fields of some or all of the records in either file may be blank, and such records may appear anywhere in the files.

During each comparison in the merging process, one record is read from each file, and these records are compared, with one of three results, depending upon whether the value of the field in the record of the text file is equal to, greater than, or less than the value of the sequence field in the record of the master file.

If the values of the sequence fields are equal, the text file record is sent to the compiler, and the record of the master file is ignored.

If the value of the sequence field in the text file is greater than the value of the sequence field in the master file, the record of the master file is sent to the compiler, and the record of the text file is retained for comparison with the next record of the master file.

If the value of the sequence field in the text file is less than the value of the sequence field in the master file, the record of the text file is sent to the compiler, and the master file record is retained for comparison with the next text file record.

Records with a blank sequence field (from either file) are assumed to have the same sequence field value as that of the last record with a non-blank sequence field read from the same file. If no record with a non-blank sequence field has yet been encountered, the blank records are assumed to have a null sequence field.

Note that this implies that records from the master file with blank sequence fields will always be compiled. This is because the records of the master file with blank sequence fields will either eventually be less than a sequence field for a text file record, or the entire text file will have been used.

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). At this point, the subsequent records on the remaining file are checked for proper sequence, and are then compiled (except for master file records within the range of a VOID parameter in a $EDIT command).

Note that masterfile records which were replaced in the merging process are not listed in your listfile during compilation. Also, files sent to the compiler by a $INCLUDE command have no sequence field checking performed on them.

SEQUENCE FIELD CHECKING

Sequence fields are checked for proper order during the merging process provided that both a text and a master file are present.

If you do not have a text file, or if you have no master file, and you still wish to have sequence fields checked, you can equate the missing file to $NULL. For example, assume that the text file, TFILE is to be compiled. Then to check for proper sequence order, the following commands may be used.

```
COBOL TFILE, , , $NULL
```

If the sequence fields are out of order, a warning message is generated and sent to the list file. However, improperly arranged sequence fields will not cause the compilation of the specified file to fail.

$EDIT Command

The $EDIT preprocessor command can be used to bypass all records of the master file whose sequence fields contain a certain value, and renumber the numeric sequence fields of records in the new file created by merging a text and a master file.

$EDIT Command Format

$EDIT [*parameter=subparameter*] [, *parameter=subparameter*] [...]

Where

subparameter is either a sequence value, a sequence number, or an increment number. Which one is used is dependent upon parameter.

parameter is either VOID, SEQNUM, NOSEQ, or INC.

VOID PARAMETER

The VOID parameter is the only parameter of the $EDIT command which has any effect upon the compilation process. The other parameters have an effect only upon the new file created by the merge process. If no new file is specified in the MPE command which initiates the compilation process, all $EDIT commands other than $EDIT VOID=sequence value are ignored.

If VOID is specified, then the subparameter must be a sequence value. This parameter indicates that all records on the master file whose sequence values are less than or equal to the specified sequence value, and any subsequent records with blank sequence fields, are to be ignored. Thus, such records are not sent to the compiler.

This voiding of master file records continues until a record of the master file with a sequence value higher than that specified in the $EDIT VOID command is encountered. When this occurs, the merging process continues as described on earlier pages.

The sequence value of the VOID parameter can be either a sequence number or a character string. If the sequence value is less than the length of the sequence field for the master file, it is left-filled with zeroes (if a number), or it is left-filled with blanks (if a character string).

Note that if the sequence fields for records in the master file contain only numerals, and the sequence value of the VOID parameter is a character string, all of the master file is voided.

SEQNUM PARAMETER

SEQNUM allows you to renumber sequence fields in the new file. If SEQNUM is used as a parameter in the $EDIT command, it has no effect upon the master or text file.

Initially, renumbering of records in a new file is disabled.

SEQNUM requires a sequence number as its subparameter. This sequence number is the value used for the first record when resequencing begins. Subsequent records are renumbered using an increment of 100, unless the INC parameter of $EDIT is used to specify a different one. The records to be renumbered start with the first record to be sent to the new file following the $EDIT SEQNUM command. This renumbering continues until a $EDIT NOSEQ command is encountered in the file in which the $EDIT SEQNUM command appears.

If the sequence number subparameter of the SEQNUM command is of insufficient length to fill the sequence field of the new file, sufficient zeroes are appended to the left of the sequence number to fill the sequence field of the records in the new file.

NOSEQ PARAMETER

The NOSEQ parameter of the $EDIT command is used to terminate the resequencing of a new file initiated by a $EDIT SEQNUM command. If a $EDIT NOSEQ command is issued after a $EDIT SEQNUM command, resequencing of new file records is terminated, and remaining records sent to the new file retain their old sequence values until a new $EDIT SEQNUM command is encountered.

If a $EDIT NOSEQ command is issued when no resequencing has been specified, it is ignored. Thus, since no resequencing takes place unless you issue a $EDIT SEQNUM command, the $EDIT NOSEQ command is the default. In this case, sequence fields are retained and passed to the new file as records are sent to the compiler.

INC PARAMETER

If INC is used as a parameter of the $EDIT command, it must have an increment value associated with it. This increment value is used when a $EDIT SEQNUM command is issued to renumber lines of the new file. The increment value may range from 1 through 999999. Note, however, that a very large increment is of limited value, since it may cause the sequence number to be long for the sequence field.

If no $EDIT INC command is issued, the default value is 100. This default value stays in effect until an INC parameter specifying a new increment value is encountered.

In general, if you wish to provide for a different increment during a resequencing operation, the $EDIT INC command must be specified before the $EDIT SEQNUM command is executed. As with the default value, the increment value specified in the $EDIT INC command stays in effect until a new increment value is specified.

$EDIT commands are normally part of the text file. You may use them in your master file, but it is not recommended since any $EDIT command using the VOID parameter in your master file could void its own continuation records. $EDIT commands themselves are never sent to the new file.

While sequence fields are allowed, and indeed 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 master file records with blank sequence fields are never replaced by lines from the text file. They can only be deleted by using the VOID parameter of $EDIT. This is accomplished by using a sequence value subparameter at least as great as the contents of the last non-blank sequence field preceding the group of records with blank sequence fields.

Since voided records are never passed to the compiler or new file, their sequence fields are never checked for proper sequence, and they never generate an out-of-sequence diagnostic message.

Any master file record replaced by a text file 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 text file record that replaces it produces an out-of-sequence diagnostic message.

In general, whenever a record sent to the new file has a nonblank sequence field lower in value than that of the last record with a non-blank sequence field, a diagnostic message is printed.

COMPILER DEPENDENT OPTIONS

There are three preprocessor commands which are compiler dependent. By compiler dependent, we mean that they are processed by the COBOL compiler.

The compiler dependent preprocessor commands are $CONTROL, $PAGE, and $TITLE.

$PAGE Command

The $PAGE command allows you to replace the first line of the title portion of the standard page heading in a list file, and to advance to the next logical page of the list file.

PAGE Command Format

$PAGE [***string*** [, ***string***] ...]

Where *string* is the data to be used in replacing the first line of the title. The characters of *string* must be preceded and followed by a quotation mark. The total number of characters used in the strings is limited to 97. This includes any blanks appearing in strings, but does not include the quotation marks used to delimit the strings.

The title line resulting from execution of the $PAGE command is a concatenation of all characters in all strings used in the command, in the order in which the strings are specified.

If no string is specified, $PAGE does not change the first line of the title, but, if $CONTROL LIST is in effect, causes the list file to be advanced to the next logical page.

If the $CONTROL LIST command is in effect when the $PAGE command is encountered, the list file is advanced to the next logical page (physical page if the list file is a line printer), and the standard page heading, including the new title, is printed, followed by one or two blank lines, depending upon whether the title is one or two lines long.

If the $CONTROL NOLIST command is in effect when the $PAGE command is encountered, the first line of the title is replaced with the specified string (or strings), but no page is advanced, and no listing of title or text occurs.

Note that $PAGE is never listed in the list file.

$TITLE Command

The $TITLE command is similar to $PAGE in that it can be used to replace the first line of a title in the list file. However, it can also be used to replace the second line of the title as well as the first, or only the second line; unlike $PAGE, it does no page advancement on the logical page of the list file.

$TITLE Command Format

$TITLE [(*n*)] [*string* [, *string*] ...]

Where

string has the same format, restrictions and use as in the $PAGE command.

n specifies which line of the title is to be replaced. Thus, n can be either 1 or 2. The default is 1.

Note that if the $TITLE command is used with no parameters, it is equivalent to replacing the first line of the title with blanks.

$Control Command

The $CONTROL command controls compilation and list options; it has the following format:

$CONTROL *parameter* [, *parameterlist*]

Where

parameterlist is one or more valid parameters (see below), each separated from a preceding parameter by a comma and zero or more optional spaces.

parameter is a valid parameter for the $CONTROL command. There are many such parameters; each is shown and discussed below.

The $CONTROL parameters are listed below:

ANSISUB	LIST	NOSOURCE
BOUNDS	NOLIST	STDWARN =*level*
CHECKSYNTAX	LOCKING	NOSTDWARN
CODE	LOCOFF	SUBPROGRAM
NOCODE	LOCON	USLINIT
CROSSREF	MAP	VERBS
NOCROSSREF	NOMAP	NOVERBS
DEBUG	MIXED	WARN
DYNAMIC	NOMIXED	NOWARN
ERRORS =*number*	QUOTE=	
LINES=*pagenum*	SOURCE	

ANSISUB

The ANSISUB parameter can be seen as a combination of the DYNAMIC and SUBPROGRAM parameters. It determines that the expanded source code currently being compiled is to be a subprogram which strictly conforms to the ANSI standard in the following ways:

1. It can be called with the CALL *identifier-1* form of the CALL statement (as with DYNAMIC).
2. It maintains values of data items from one call to the next (as with SUBPROGRAM).
3. You may use the CANCEL statement to cause subsequent calls to the subprogram to reset data items to their initial values.

Although the data area is dynamically allocated on each call, the data values are saved in a file between calls. For this reason, ANSISUB is the least efficient of the three types of subprograms.

If you are going to call an ANSISUB subprogram with the CALL *identifier-1* form of the CALL statement, you must first place it in a segmented library (SL).

The defaults assumed by the $CONTROL command produce a condition equivalent to the one resulting from the $CONTROL command shown below.

```
$CONTROL NOCODE, NOCROSSREF, LINES = 60, LIST, LOCON, NOMAP,&
$  MIXED, SOURCE, NOSTDWARN, NOVERBS, WARN
```

BOUNDS

The BOUNDS parameter requests the compiler to generate code for the validation of indices and subscripts used in tables of the expanded source file being compiled. Initially, no bounds checking is enabled. This parameter, if used, must appear in a $CONTROL command before the Procedure Division is encountered.

CHECKSYNTAX

Checks the syntax of the program without producing an object program.

CODE

The CODE parameter requests a copy of the object code to be included in your list file. This object code is an octal listing of the machine code generated by the compilation of your expanded source code.

NOCODE

The NOCODE parameter requests that no object code be included in your source listing. This is the default; thus, the purpose of the NOCODE parameter is to negate a previously issued $CONTROL CODE command.

CROSSREF

The CROSSREF parameter of $CONTROL requests a cross reference of symbols and labels used in the expanded source file. This cross reference is sent to the list file.

NOCROSSREF

The NOCROSSREF parameter of $CONTROL requests that no cross reference of symbols and labels used in the expanded source file be listed in the list file.

This is the default; thus, its use is to cancel a previously issued $CONTROL CROSSREF command.

DEBUG

The DEBUG parameter requests that COBOL II/3000 generate a call in the initialization segment of your main program to the XCONTRAP intrinsic. This MPE intrinsic allows you to transfer control to MPE Debug at any time during execution of the prepared code. To transfer control to Debug, you must press the control (CNTL) key, and while holding it down, press the Y key.

DYNAMIC

The DYNAMIC parameter indicates that the expanded source code currently being compiled is to be a subprogram using Q-relative addressing. This or the SUBPROGRAM parameter must be used if you are compiling a subprogram; otherwise, the compiler assumes that the expanded source file is a main program.

ERRORS= ***number***

The ERRORS parameter uses an integer subparameter, *number,* to specify the maximum number of errors to be allowed before compilation of the expanded source file is terminated. Thus, for example, if *number* is set to 10, then when compilation of a source file begins, it will terminate either after all records have been compiled with less than ten errors, or after the tenth compilation error has been found and listed to the list file.

LINES= ***pagenum***

The LINES parameter of the $CONTROL command allows you to define the number of lines to be written on the logical page of your list file. For example, if the line printer is your list file, then the command, $CONTROL LINES=30, causes thirty lines to be listed on each page of the hard copy produced by the line printer.

LIST

The LIST parameter of the $CONTROL command enables the listing of all source text, as well as error and warning messages, subsystem initiation and completion messages, and all other listings requested by the $CONTROL command (as for example, $CONTROL CODE).

Initally, this listing option is in effect by default. You might wish to use it to cancel a previously issued $CONTROL NOLIST, or $CONTROL LOCOFF command.

NOLIST

The NOLIST parameter of $CONTROL disables the listing of source text, and all other listings requested by previous $CONTROL commands. It does not, however, disable the listing of erroneous source records, error and warning messages, and subsystem initiation and completion messages.

LOCKING

This parameter is not required if the EXCLUSIVE and UN-EXCLUSIVE statements are used. The LOCKING parameter of the $CONTROL parameter enables dynamic locking of files opened for shared access during the execution of your program. Note that a $CONTROL LOCKING command does not lock your file; it simply makes it possible for you to do so from within your program by using the COBOLLOCK and COBOLUNLOCK parameters of the CALL statement.

LOCOFF

The LOCOFF parameter of $CONTROL has the same effect as the NOLIST parameter. Thus, only erroneous source records, error and warning messages, and subsystem initiation and completion messages are listed.

A $CONTROL LOCOFF command remains in effect until either a $CONTROL LOCON or a $CONTROL LIST parameter is encountered.

A $CONTROL LOCOFF command may be nested.

LOCON

The LOCON parameter of the $CONTROL command negates the effect of any $CONTROL LOCOFF command issued previously; thus, if a $CONTROL LIST command has been issued before $CONTROL LOCOFF, then a following $CONTROL LOCON restores listing of output to the list file. If a $CONTROL NOLIST command was issued before a $CONTROL LOCOFF, then a following $CONTROL LOCON has no effect upon the list file.

$CONTROL LOCON and $CONTROL LOCOFF commands may be nested. Thus, if you wish, for example, to use $INCLUDE to copy a file into your source file, but you do not wish to have it listed, using $CONTROL LOCOFF as the first command, and $CONTROL LOCON as the last command in the file to be copied suppresses the listing of the copied file and restores the listing option in effect before the $INCLUDE command was encountered.

MAP

The MAP parameter of $CONTROL requests a symbol table map to be included in your list file.

NOMAP

The NOMAP parameter requests that no symbol table map be included in your list file. This is the default; thus, the purpose of the NOMAP parameter is to negate a previously issued $CONTROL MAP command.

MIXED

The MIXED parameter of the $CONTROL command requests that the list file include all preprocessor commands used in the expanded source file. Note that the $PAGE and $TITLE preprocessor commands are not listed even if this option is in effect.

This option is the default; thus, if you do not wish to have preprocessor commands included as part of the list file, you should use the NOMIXED parameter of the $CONTROL command.

NOMIXED

The NOMIXED parameter of $CONTROL requests that no preprocessor commands be listed in the list file.

QUOTE = { " ' }

The QUOTE parameter in COBOL II/3000 is usually not necessary. It is provided only for the purpose of defining the figurative constant QUOTE and simplifying conversion from COBOL/3000 source programs (based on ANSI COBOL '68) to COBOL II/3000 source programs.

SOURCE

The SOURCE parameter of $CONTROL command requests the listing of the expanded source file to the list file.

Initially, this listing option is in effect by default; however, if the input file is your terminal, this option is disabled unless you explicitly request it. Note that if you do request the SOURCE option while using your terminal as the input device, the result is an immediate echoing of each line entered.

NOSOURCE

The NOSOURCE parameter of $CONTROL disables listing of the expanded source file generated by the preprocessor.

STDWARN

```
          ┌   ┌ HIGH ┐ ┐
STDWARN   │ = │ H-I  │ │
          │   │ L-I  │ │
          └   └ LOW  ┘ ┘
```

The STDWARN parameter requests that the compiler flag features in your source file which are part of COBOL II/3000, but are not part of one of four levels of Federal Standard COBOL (FIPS). The four levels are low, low intermeditate, high intermediate, and high. The default, high, will flag COBOL II/3000 extensions. These flagged features are listed in your list file. STDWARN is useful in converting your COBOL II source program to conform to one of the four Federal Standard levels of COBOL.

The table below summarizes which of the two implementation levels of the 12 standard COBOL modules are contained in each of the four federal levels of the language.

To illustrate how to interpret the table, examine the Relative I-O module entry.

When you request that your program be compared to the low level federal standard, any program construct using Relative I-O will be flagged, since the low level federal standard implementation has no implementation of Relative I-O.

When compared to the low-intermediate federal standard, which has a level 1 implementation of Relative I-O, only constructs using the level 2 implementation would be flagged as not being part of the low-intermediate federal standard.

When your program is compared to the high-intermediate or to the high level federal standards for COBOL, no construct utilizing Relative I-O will be flagged, since both standards contain the level 2 implementation of Relative I-O.

	Low Level	Low Intermediate Level	High Intermediate Level	High Level
Nucleus	1	1	2	2
FPM's				
Table Handling	1	1	2	2
Sequential I–O	1	1	2	2
Relative I–O	—	1	2	2
Indexed I–O	—	—	—	2
Sort-Merge	—	—	1	2
Report Writer	—	—	—	—
Segmentation	—	1	1	2
Library	—	1	1	2
Debug	—	1	2	2
Inter-Program Communication	—	1	2	2
Communication	—	—	2	2

NOSTDWARN

The NOSTDWARN parameter requests that the compiler not flag non-standard features of COBOL II/3000. This is the initial condition of the compiler. Thus, if you wish to have features which are not part of ANSI COBOL'74 noted, you should use the STDWARN parameter of $CONTROL.

SUBPROGRAM

The SUBPROGRAM parameter indicates that the expanded source code currently being compiled is to be a subprogram using DB-relative addressing. This parameter, or the DYNAMIC parameter, must be used when you are compiling a subprogram, since the compiler otherwise assumes the expanded source code to be the source for a main program.

USLINIT

The USLINIT parameter initializes the usl file into which the compiled code is stored. This gives you a completely clear file. If you are compiling a source file into a usl file containing the object code of a main or subprogram to be used with the object code currently being compiled, you should not use this parameter.

VERBS

The VERBS parameter of the $CONTROL command requests that each verb used in the Procedure Division, as well as the verb's line number and PB relative address, be listed in the list file.

NOVERBS

The NOVERBS parameter of the $CONTROL command requests that the listing of verbs as described under VERBS above not be done.

This is the default; thus, the NOVERBS parameter is used to cancel a previously issued $CONTROL VERBS command.

WARN

The WARN parameter enables the flagging of possible, but not clearly, erroneous conditions within your expanded source file. These flagged conditions are listed in your list file, along with an appropriate warning message.

The WARN option is a default condition. Thus, you need use it only to negate the use of a previously issued $CONTROL NOWARN command.

NOWARN

The NOWARN parameter disables the flagging and listing of possible erroneous conditions and their associated warning messages.

MPE COMMANDS AND FILES

APPENDIX B

To the Multiprogramming Executive Operating System (MPE), the COBOL II/3000 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 segmenter subsystem. The preparation step formats the object program so that it can be manipulated efficiently by MPE when the program is executed. After preparation, the program is ready for execution.

Any one of the three MPE commands summarized below may be used to invoke the COBOL II/3000 compiler. These commands may be entered as part of the input stream in the 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 there are 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 its 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 preprocessor subsystem commands. Processor commands and their formats are fully explained in Appendix A of this manual.

After a user initiates an MPE batch job or session, he can compile, prepare, and execute his COBOL II/3000 programs through various MPE 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, Chapter VI.

Compiling/Preparing/Executing Programs

User source programs written in COBOL II/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, you 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 II/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 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 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 Using COBOL II Manual (32233-90003—available in mid-1980) discusses program development, compilation and preparation on the HP 3000.

Manipulating USL Files with the MPE Segmenter

The MPE Segmenter subsystem can be used to arrange Relocatable Binary Modules in a User Subprogram Library for presentation at program preparation time. The segmenter allows you to look at the contents of a User Subprogram Library or Relocatable Library and to add, delete, activate or deactivate, and rearrange the object code modules that reside in them. For more discussion on the use of the MPE Segmenter, consult Using COBOL II Manual (32233-90003) and MPE Segmenter Reference Manual (3000-90011).

NOTE

> The following method of compiling, preparing, and executing a COBOL program may be used if the COBOL II/3000 compiler has the name COBOLII.PUB.SYS.

Call the COBOL II compiler using the MPE :RUN command. Before using the :RUN command, you must use file equations for the files normally specified on the :COBOL command. The formal file designators are:

COBTEXT	(*textfile*)
COBLIST	(*listfile*)
COBUSL	(*uslfile*)
COBMAST	(*masterfile*)
COBNEW	(*newfile*)

Thus, to compile from the file MYSOURCE and send the listing to the line printer, you would use

```
:FILE COBTEXT=MYSOURCE
:FILE COBLIST;DEV=LP
```

before using the :RUN command.

Additionally, you must specify a PARM=*parameternum* parameter on the :RUN command to indicate which files are present, that is, which default values for formal file designators you have overridden through issuing :FILE commands. The value is between 0 and 31 as shown in the table on page B-5.

For example, to invoke the compiler with the *textfile* and *listfile* present, you would use the command:

```
RUN COBOLII.PUB.SYS;PARM=3
```

PARM VALUES

PARAMETERNUM	FILES REDEFINED BY FILE EQUATIONS
0	None
1	*textfile*
2	*listfile*
3	*listfile,textfile*
4	*uslfile*
5	*uslfile, textfile*
6	*uslfile, listfile*
7	*uslfile, listfile, textfile*
8	*masterfile*
9	*masterfile, textfile*
10	*masterfile, listfile*
11	*masterfile, listfile, textfile*
12	*masterfile, uslfile*
13	*masterfile, uslfile, textfile*
14	*masterfile, uslfile, listfile*
15	*masterfile, uslfile, listfile, textfile*
16	*newfile*
17	*newfile, textfile*
18	*newfile, listfile*
19	*newfile, listfile, textfile*
20	*newfile, uslfile*
21	*newfile, uslfile, textfile*
22	*newfile, uslfile, listfile*
23	*newfile, uslfile, listfile, textfile*
24	*newfile, masterfile*
25	*newfile, masterfile, textfile*
26	*newfile, masterfile, listfile*
27	*newfile, masterfile, listfile, textfile*
28	*newfile, masterfile, uslfile*
29	*newfile, masterfile, uslfile, textfile*
30	*newfile, masterfile, uslfile, listfile*
31	*newfile, masterfile, uslfile, listfile, textfile*

COBOL II EXAMPLE PROGRAM

APPENDIX C

This section contains the output listing resulting from the compile, prepare and run of a sample COBOL II program. The $CONTROL options MAP, CODE, VERBS and CROSSREF result in the symbol table map, the octal code, the verb map and the identifier cross-reference which follow the compiler listing. The following MPE commands were issued at a terminal:

```
:FILE COBLIST=$STDLIST
:FILE COBTEXT=EXAMPLE2
:RUN COBOLII.PUB.SYS;PARM=3
```

The following was output at the line printer:

```
PAGE 0001   HEWLETT-PACKARD 32233A.00.00  COBOL II/3000 TUE, NOV 27, 1979, 10:23 AM (C)
HEWLETT-PACKARD CO. 1979
:
00001            001000$CONTROL USLINIT,MAP,CODE,VERBS,CROSSREF
00003            001100 IDENTIFICATION DIVISION.
00004            001200 PROGRAM-ID.                       EXAMPLE.
00005            001300 AUTHOR.                           HEWLETT-PACKARD.
00006            001400 DATE-WRITTEN.                     JULY 1979.
00007            001500 DATE-COMPILED.  TUE, NOV 27, 1979, 10:23 AM
00008            001600*************************************************************
00009            001700*   BRIEF PROGRAM DESCRIPTION
00010            001800*
00011            001900*     THIS IS AN EXAMPLE OF THE USE OF COBOL II.  THIS IS A
00012            002000*   SEQUENTIAL UPDATE PROGRAM USING A STRUCTURED PROGRAMMING
00013            002100*   TECHNIQUE.  THE TRANSACTION FILE USED BY THE UPDATE PRO-
00014            002200*   GRAM HAS ALREADY BEEN EDITED AND SORTED INTO THE PROPER
00015            002300*   SEQUENCE FOR UPDATE PROCESSING.
00016            002400*************************************************************
00017            002500*   FILE REQUIREMENTS
00018            002600*
00019            002700*   COPY FILES: NONE REQUIRED IN THIS PROGRAM
00020            002800*
00021            002900*   DATA FILES INPUT FILES:  OLD INVENTORY MASTER
00022            003000*                             UPDATE TRANSACTIONS
00023            003100*              OUTPUT FILES: NEW INVENTORY MASTER
00024            003200*                             TRANSACTION ERROR FILE
00025            003300*                             PRINTED REPORT OF UPDATE
00026            003400*              I-O    FILES: NONE
00027            003500*
00028            003600*************************************************************
00029            003700 ENVIRONMENT DIVISION.
00030            003800 CONFIGURATION SECTION.
00031            003900 SOURCE-COMPUTER.                            HP-3000.
00032            004000 OBJECT-COMPUTER.                            HP-3000.
00033            004100 INPUT-OUTPUT SECTION.
00034            004200 FILE-CONTROL.
00035            004300     SELECT OLD-INV-MAST
00036            004400         ASSIGN TO ''OLDMAST''.
00037            004500     SELECT NEW-INV-MAST
00038            004600         ASSIGN TO ''NEWMAST''.
00039            004700     SELECT TRAN-FILE
00040            004800         ASSIGN TO ''TRANFILE''.
00041            004900     SELECT ERROR-FILE
00042            005000         ASSIGN TO ''TRANERR''.
00043            005100     SELECT PRINT-FILE
00044            005200         ASSIGN TO ''PRINT,UR''.
```

```
00045          005300/ ****** D A T A   D I V I S I O N ******
00046          005400 DATA DIVISION.
00047          005500 FILE SECTION.
00048          005600*
00049          005700 FD  OLD-INV-MAST
00050          005800     RECORD CONTAINS 40 CHARACTERS
00051          005900     DATA RECORDS ARE OLD-INV-REC.
00052          006000*
00053          006100 01  OLD-INV-MAST-REC.
00054          006200     03  OM-PART-NBR         PIC  X(05).
00055          006300     03  FILLER              PIC  X(35).
00056          006400*
00057          006500 FD  NEW-INV-MAST
00058          006600     RECORD CONTAINS 40 CHARACTERS
00059          006700     DATA RECORDS ARE NEW-INV-MAST-REC.
00060          006800*
00061          006900 01  NEW-INV-MAST-REC        PIC  X(40).
00062          007000*
00063          007100 FD  TRAN-FILE
00064          007200     RECORD CONTAINS 60 CHARACTERS
00065          007300     DATA RECORDS ARE TRAN-REC.
00066          007400 01  TRAN-REC.
00067          007500     03  TR-UPDATE-CODE          PIC  X(01).
00068          007600         88  TR-ADD-CODE                     VALUE ''A''.
00069          007700         88  TR-CHANGE-CODE                  VALUE ''C''.
00070          007800         88  TR-DELETE-CODE                  VALUE ''D''.
00071          007900     03  TR-PART-NBR             PIC  X(05).
00072          008000     03  TR-DESCRIPTION          PIC  X(25).
00073          008100     03  TR-PART-COST-FLD.
00074          008200         05  TR-PART-COST        PIC  9(07)V99.
00075          008300     03  TR-PART-PRICE-FLD.
00076          008400         05  TR-PART-PRICE       PIC  9(05)V99.
00077          008500     03  TR-PART-QUANTITY-FLD.
00078          008600         05  TR-PART-QUANTITY    PIC  9(04).
00079          008700     03  FILLER                  PIC  X(09).
00080          008800*
00081          008900 FD  ERROR-FILE
00082          009000     RECORD CONTAINS 60 CHARACTERS
00083          009100     DATA RECORDS ARE ERROR-REC.
00084          009200 01  ERROR-REC               PIC  X(60).
00085          009300*
00086          009400 FD  PRINT-FILE
00087          009500     RECORD CONTAINS 133 CHARACTERS
00088          009600     DATA RECORDS ARE PRINT-REC.
00089          009700 01  PRINT-REC               PIC  X(133).
```

```
00090          009800/ ******  W O R K I N G   S T O R A G E  ******
00091          009900 WORKING-STORAGE SECTION.
00092          010000*
00093          010100*
00094          010200 01  WS-PRINT-CONTROL.
00095          010300     03  WS-LINE-CTR          PIC S9(03) COMP VALUE +999.
00096          010400     03  WS-PAGE-CTR          PIC S9(03) COMP VALUE +0.
00097          010500     03  WS-SPACING           PIC S9(01) COMP VALUE +1.
00098          010600     03  WS-LINE-LMT          PIC S9(03) COMP VALUE +45.
00099          010700*
00100          010800 01  WS-ACCUMULATORS.
00101          010900     03  WS-CHANGES-CTR       PIC S9(05) COMP VALUE +0.
00102          011000     03  WS-ADDITIONS-CTR     PIC S9(05) COMP VALUE +0.
00103          011100     03  WS-DELETES-CTR       PIC S9(05) COMP VALUE +0.
00104          011200     03  WS-TOTAL-CTR         PIC S9(05) COMP VALUE +0.
00105          011300     03  WS-ERRORS            PIC S9(05) COMP VALUE +0.
00106          011400     03  WS-TRANS-READ        PIC S9(05) COMP VALUE +0.
00107          011500*
00108          011600 01  WS-UPDT-MESSAGES.
00109          011700     03  WS-CHANGE-MSG        PIC  X(10) VALUE ''CHANGED   ''.
00110          011800     03  WS-ADDITION-MSG      PIC  X(10) VALUE ''ADDED     ''.
00111          011900     03  WS-DELETE-MSG        PIC  X(10) VALUE ''DELETED   ''.
00112          012000*
00113          012100 01  WS-MASTER-REC.
00114          012200     03  WS-MR-PART-NBR       PIC  X(05).
00115          012300     03  WS-MR-DESCRIPTION    PIC  X(25).
00116          012400     03  WS-MR-PART-COST      PIC S9(07)V99 COMP SYNC.
00117          012500     03  WS-MR-PART-PRICE     PIC S9(05)V99 COMP SYNC.
00118          012600     03  WS-MR-PART-QUANTITY PIC S9(04)     COMP SYNC.
00119          012700*
00120          012800 01  HDG-1.
00121          012900     03  HDG1-DATE           PIC  X(08).
00122          013000     03  FILLER              PIC  X(22) VALUE SPACES.
00123          013100     03  HDG1-REPORT-NAME    PIC  X(24)
00124          013200         VALUE ''INVENTORY UPDATE LISTING''.
00125          013300     03  FILLER              PIC  X(20) VALUE SPACES.
00126          013400     03  FILLER              PIC  X(06) VALUE ''PAGE  ''.
00127          013500     03  HDG1-PAGE-NBR       PIC  ZZ9.
00128          013600*
00129          013700 01  HDG-2.
00130          013800     03  FILLER              PIC  X(20)
00131          013900         VALUE ''PART      PART      ''.
00132          014000     03  FILLER              PIC  X(20)
00133          014100         VALUE ''                    ''.
00134          014200     03  FILLER              PIC  X(20)
00135          014300         VALUE ''      PART         PA''.
00136          014400     03  FILLER              PIC  X(20)
00137          014500         VALUE ''RT        PART    UPD''.
00138          014600     03  FILLER              PIC  X(20)
00139          014700         VALUE ''ATE                 ''.
00140          014800     03  FILLER              PIC  X(20)
00141          014900         VALUE ''                    ''.
00142          015000     03  FILLER              PIC  X(13)
00143          015100         VALUE ''             ''.
00144          015200*
00145          015300 01  HDG-3.
00146          015400     03  FILLER              PIC  X(20)
```

```
00147           015500          VALUE ''NUMBER   DESCRIPTION''.
00148           015600      03  FILLER              PIC  X(20)
00149           015700          VALUE ''                  ''.
00150           015800      03  FILLER              PIC  X(20)
00151           015900          VALUE ''      COST        PR''.
00152           016000      03  FILLER              PIC  X(20)
00153           016100          VALUE ''ICE   QUANTITY   MES''.
00154           016200      03  FILLER              PIC  X(20)
00155           016300          VALUE ''SAGE              ''.
00156           016400      03  FILLER              PIC  X(20)
00157           016500          VALUE ''                  ''.
00158           016600      03  FILLER              PIC  X(13)
00159           016700          VALUE ''           ''.
00160           016800*
00161           016900 01  TOTALS-HDG-1.
00162           017000      03  FILLER              PIC  X(20)
00163           017100          VALUE ''TOTALS FOR INVENTORY''.
00164           017200      03  FILLER              PIC  X(20)
00165           017300          VALUE '' UPDATE RUN OF  -   ''.
00166           017400      03  TOT1-HDG-DATE       PIC  X(08).
00167           017500*
00168           017600 01  TOTALS-HDG-2.
00169           017700      03  FILLER              PIC  X(20)
00170           017800          VALUE ''CHANGES           ''.
00171           017900      03  TOT2-CHANGES        PIC  ZZ,ZZ9.
00172           018000*
00173           018100 01  TOTALS-HDG-3.
00174           018200      03  FILLER              PIC  X(20)
00175           018300          VALUE ''ADDITIONS         ''.
00176           018400      03  TOT3-ADDITIONS      PIC  ZZ,ZZ9.
00177           018500*
00178           018600 01  TOTALS-HDG-4.
00179           018700      03  FILLER              PIC  X(20)
00180           018800          VALUE ''DELETIONS         ''.
00181           018900      03  TOT4-DELETIONS      PIC  ZZ,ZZ9.
00182           019000*
00183           019100 01  TOTALS-HDG-5.
00184           019200      03  FILLER              PIC  X(20)
00185           019300          VALUE ''TOTAL UPDATES     ''.
00186           019400      03  TOT5-UPDATES        PIC  ZZ,ZZ9.
00187           019500*
00188           019600 01  TOTALS-HDG-6.
00189           019700      03  FILLER              PIC  X(20)
00190           019800          VALUE ''ERRORS            ''.
00191           019900      03  TOT6-ERRORS         PIC  ZZ,ZZ9.
00192           020000*
00193           020100 01  TOTALS-HDG-7.
00194           020200      03  FILLER              PIC  X(20)
00195           020300          VALUE ''TOTAL TRANSACTIONS  ''.
00196           020400      03  TOT7-TRANS-READ     PIC ZZ,ZZ9.
00197           020500*
00198           020600 01  WS-UPDATE-LINE.
00199           020700      03  WS-UP-PART-NBR        PIC  X(05).
00200           020800      03  FILLER                PIC  X(04) VALUE SPACES.
00201           020900      03  WS-UP-DESCRIPTION     PIC  X(25).
00202           021000      03  FILLER                PIC  X(04) VALUE SPACES.
00203           021100      03  WS-UP-PART-COST       PIC  Z,ZZZ,ZZZ.99-.
```

```
00204           021200     03  FILLER              PIC  X(03) VALUE SPACES.
00205           021300     03  WS-UP-PART-PRICE    PIC  ZZ,ZZZ.99-.
00206           021400     03  FILLER              PIC  X(05) VALUE SPACES.
00207           021500     03  WS-UP-PART-QUANTITY PIC  ZZZ9.
00208           021600     03  FILLER              PIC  X(04) VALUE SPACES.
00209           021700     03  WS-UP-UPDT-MESSAGE  PIC  X(10).
00210           021800*
```

```
00211          021900/  ******  P R O C E D U R E   D I V I S I O N  ******
00212          022000 PROCEDURE DIVISION.
00213          022100 100-START-OF-PROGRAM.
00214          022200     OPEN INPUT  OLD-INV-MAST
00215          022300                 TRAN-FILE
00216          022400          OUTPUT NEW-INV-MAST
00217          022500                 ERROR-FILE
00218          022600                 PRINT-FILE.
00219          022700*
00220          022800     MOVE SPACES       TO PRINT-REC.
00221          022900     MOVE CURRENT-DATE TO HDG1-DATE
00222          023000                          TOT1-HDG-DATE.
00223          023100*
00224          023200*  GET FIRST TRANSACTION RECORD TO PROCESS
00225          023300*
00226          023400     PERFORM 300-GET-TRANSACTION.
00227          023500*
00228          023600*  GET FIRST MASTER FILE RECORD
00229          023700*
00230          023800     PERFORM 310-GET-OLD-MASTER.
00231          023900*
00232          024000*  M A I N   P R O G R A M   D R I V E R
00233          024100*
00234          024200     PERFORM 200-MATCH-MAST-VS-TRAN
00235          024300         UNTIL WS-MR-PART-NBR EQUAL ALL ``9'' AND
00236          024400               TR-PART-NBR    EQUAL ALL ``9''.
00237          024500     PERFORM 420-PRINT-TOTALS.
00238          024600*
00239          024700     CLOSE OLD-INV-MAST
00240          024800           NEW-INV-MAST
00241          024900           TRAN-FILE
00242          025000           ERROR-FILE
00243          025100           PRINT-FILE.
00244          025200     STOP RUN.
00245          025300*
00246          025400 200-MATCH-MAST-VS-TRAN.
00247          025500*
00248          025600     IF  WS-MR-PART-NBR GREATER TR-PART-NBR
00249          025700         PERFORM 210-MASTER-COMPARED-HIGH
00250          025800     ELSE
00251          025900     IF  WS-MR-PART-NBR LESS    TR-PART-NBR
00252          026000         PERFORM 240-MASTER-COMPARED-LOW
00253          026100     ELSE
00254          026200         PERFORM 250-MASTER-AND-TRAN-EQUAL.
00255          026300 210-MASTER-COMPARED-HIGH.
00256          026400*
00257          026500     IF  TR-ADD-CODE
00258          026600         PERFORM 220-ADD-TO-MASTER
00259          026700     ELSE
00260          026800         PERFORM 230-TRAN-IN-ERROR.
00261          026900*
00262          027000 220-ADD-TO-MASTER.
00263          027100     MOVE TR-PART-NBR       TO WS-MR-PART-NBR.
00264          027200     MOVE TR-DESCRIPTION    TO WS-MR-DESCRIPTION.
00265          027300     MOVE TR-PART-COST      TO WS-MR-PART-COST.
00266          027400     MOVE TR-PART-PRICE     TO WS-MR-PART-PRICE.
00267          027500     MOVE TR-PART-QUANTITY TO WS-MR-PART-QUANTITY.
```

```
00268          027600     MOVE WS-MASTER-REC    TO NEW-INV-MAST-REC.
00269          027700     MOVE TR-PART-NBR      TO WS-UP-PART-NBR.
00270          027800     MOVE TR-DESCRIPTION   TO WS-UP-DESCRIPTION.
00271          027900     MOVE TR-PART-COST     TO WS-UP-PART-COST.
00272          028000     MOVE TR-PART-PRICE    TO WS-UP-PART-PRICE.
00273          028100     MOVE TR-PART-QUANTITY TO WS-UP-PART-QUANTITY.
00274          028200     MOVE WS-ADDITION-MSG  TO WS-UP-UPDT-MESSAGE.
00275          028300     PERFORM 300-GET-TRANSACTION.
00276          028400     PERFORM 330-WRITE-NEW-MASTER.
00277          028500     PERFORM 320-PRINT-UPDATE.
00278          028600     MOVE OLD-INV-MAST-REC TO WS-MASTER-REC.
00279          028700     ADD 1                 TO WS-ADDITIONS-CTR.
00280          028800*
00281          028900 230-TRAN-IN-ERROR.
00282          029000     MOVE TRAN-REC TO ERROR-REC.
00283          029100     WRITE ERROR-REC.
00284          029200     PERFORM 300-GET-TRANSACTION.
00285          029300     ADD 1 TO WS-ERRORS.
00286          029400*
00287          029500 240-MASTER-COMPARED-LOW.
00288          029600     MOVE WS-MASTER-REC TO NEW-INV-MAST-REC.
00289          029700     PERFORM 330-WRITE-NEW-MASTER.
00290          029800     PERFORM 310-GET-OLD-MASTER.
00291          029900*
00292          030000 250-MASTER-AND-TRAN-EQUAL.
00293          030100     IF  TR-DELETE-CODE
00294          030200         PERFORM 260-DELETE-MASTER
00295          030300     ELSE
00296          030400     IF  TR-CHANGE-CODE
00297          030500         PERFORM 270-CHANGE-MASTER
00298          030600     ELSE
00299          030700         PERFORM 230-TRAN-IN-ERROR.
00300          030800*
00301          030900 260-DELETE-MASTER.
00302          031000     MOVE WS-MR-PART-NBR       TO WS-UP-PART-NBR.
00303          031100     MOVE WS-MR-DESCRIPTION    TO WS-UP-DESCRIPTION.
00304          031200     MOVE WS-MR-PART-COST      TO WS-UP-PART-COST.
00305          031300     MOVE WS-MR-PART-PRICE     TO WS-UP-PART-PRICE.
00306          031400     MOVE WS-MR-PART-QUANTITY TO WS-UP-PART-QUANTITY.
00307          031500     MOVE WS-DELETE-MSG        TO WS-UP-UPDT-MESSAGE.
00308          031600     PERFORM 320-PRINT-UPDATE.
00309          031700     PERFORM 310-GET-OLD-MASTER.
00310          031800     PERFORM 300-GET-TRANSACTION.
00311          031900     ADD 1                     TO WS-DELETES-CTR.
00312          032000*
00313          032100 270-CHANGE-MASTER.
00314          032200*
00315          032300     IF  TR-DESCRIPTION NOT EQUAL SPACES
00316          032400         MOVE TR-DESCRIPTION   TO WS-MR-DESCRIPTION.
00317          032500     IF  TR-PART-COST-FLD NOT EQUAL SPACES
00318          032600         MOVE TR-PART-COST     TO WS-MR-PART-COST.
00319          032700     IF  TR-PART-PRICE-FLD NOT EQUAL SPACES
00320          032800         MOVE TR-PART-PRICE    TO WS-MR-PART-PRICE.
00321          032900     IF  TR-PART-QUANTITY-FLD NOT EQUAL SPACES
00322          033000         MOVE TR-PART-QUANTITY TO WS-MR-PART-QUANTITY.
00323          033100*
00324          033200     MOVE WS-MR-PART-NBR      TO WS-UP-PART-NBR.
```

```
00325          033300     MOVE WS-MR-DESCRIPTION   TO WS-UP-DESCRIPTION.
00326          033400     MOVE WS-MR-PART-COST     TO WS-UP-PART-COST.
00327          033500     MOVE WS-MR-PART-PRICE    TO WS-UP-PART-PRICE.
00328          033600     MOVE WS-MR-PART-QUANTITY TO WS-UP-PART-QUANTITY.
00329          033700     MOVE WS-CHANGE-MSG       TO WS-UP-UPDT-MESSAGE.
00330          033800     PERFORM 320-PRINT-UPDATE.
00331          033900     PERFORM 300-GET-TRANSACTION.
00332          034000     ADD 1                    TO WS-CHANGES-CTR.
00333          034100*
00334          034200 300-GET-TRANSACTION.
00335          034300     READ TRAN-FILE
00336          034400         AT END
00337          034500         MOVE ALL ''9'' TO TR-PART-NBR.
00338          034600*
00339          034700     IF  TR-PART-NBR NOT EQUAL ALL ''9''
00340          034800         ADD 1 TO WS-TRANS-READ.
00341          034900*
00342          035000 310-GET-OLD-MASTER.
00343          035100     READ OLD-INV-MAST
00344          035200         AT END
00345          035300         MOVE ALL ''9'' TO OM-PART-NBR.
00346          035400*
00347          035500     MOVE OLD-INV-MAST-REC TO WS-MASTER-REC.
00348          035600*
00349          035700 320-PRINT-UPDATE.
00350          035800*
00351          035900     IF  WS-LINE-CTR GREATER WS-LINE-LMT
00352          036000         PERFORM 410-PRINT-HEADING.
00353          036100*
00354          036200     MOVE WS-UPDATE-LINE TO PRINT-REC.
00355          036300     PERFORM 400-WRITE-PRINT-LINE.
00356          036400     ADD 1 TO WS-LINE-CTR.
00357          036500*
00358          036600 330-WRITE-NEW-MASTER.
00359          036700     WRITE NEW-INV-MAST-REC.
00360          036800*
00361          036900 400-WRITE-PRINT-LINE.
00362          037000     WRITE PRINT-REC BEFORE ADVANCING WS-SPACING.
00363          037100     MOVE SPACES    TO PRINT-REC.
00364          037200     ADD WS-SPACING TO WS-LINE-CTR.
00365          037300*
00366          037400 410-PRINT-HEADING.
00367          037500     WRITE PRINT-REC BEFORE ADVANCING PAGE.
00368          037600     MOVE ZEROES       TO WS-LINE-CTR.
00369          037700     ADD 1             TO WS-PAGE-CTR.
00370          037800     MOVE WS-PAGE-CTR TO HDG1-PAGE-NBR.
00371          037900     MOVE 2            TO WS-SPACING.
00372          038000     MOVE HDG-1        TO PRINT-REC.
00373          038100     PERFORM 400-WRITE-PRINT-LINE.
00374          038200     MOVE 1            TO WS-SPACING.
00375          038300     MOVE HDG-2        TO PRINT-REC.
00376          038400     PERFORM 400-WRITE-PRINT-LINE.
00377          038500     MOVE 2            TO WS-SPACING.
00378          038600     MOVE HDG-3        TO PRINT-REC.
00379          038700     PERFORM 400-WRITE-PRINT-LINE.
00380          038800*
00381          038900 420-PRINT-TOTALS.
```

```
PAGE 0009/COBTEXT  EXAMPLE
00382          039000     ADD WS-CHANGES-CTR
00383          039100         WS-ADDITIONS-CTR
00384          039200         WS-DELETES-CTR GIVING WS-TOTAL-CTR.
00385          039300*
00386          039400     MOVE 1                  TO WS-SPACING.
00387          039500     MOVE WS-CHANGES-CTR     TO TOT2-CHANGES.
00388          039600     MOVE WS-ADDITIONS-CTR   TO TOT3-ADDITIONS.
00389          039700     MOVE WS-DELETES-CTR     TO TOT4-DELETIONS.
00390          039800     MOVE WS-TOTAL-CTR       TO TOT5-UPDATES.
00391          039900     MOVE WS-ERRORS          TO TOT6-ERRORS.
00392          040000     MOVE WS-TRANS-READ      TO TOT7-TRANS-READ.
00393          040100*
00394          040200     MOVE TOTALS-HDG-1 TO PRINT-REC.
00395          040300     PERFORM 400-WRITE-PRINT-LINE.
00396          040400     MOVE TOTALS-HDG-2 TO PRINT-REC.
00397          040500     PERFORM 400-WRITE-PRINT-LINE.
00398          040600     MOVE TOTALS-HDG-3 TO PRINT-REC.
00399          040700     PERFORM 400-WRITE-PRINT-LINE.
00400          040800     MOVE TOTALS-HDG-4 TO PRINT-REC.
00401          040900     PERFORM 400-WRITE-PRINT-LINE.
00402          041000     MOVE TOTALS-HDG-5 TO PRINT-REC.
00403          041100     PERFORM 400-WRITE-PRINT-LINE.
00404          041200     MOVE TOTALS-HDG-6 TO PRINT-REC.
00405          041300     PERFORM 400-WRITE-PRINT-LINE.
00406          041400     MOVE TOTALS-HDG-7 TO PRINT-REC.
00407          041500     PERFORM 400-WRITE-PRINT-LINE.
00408          041600*
```

```
00000  170400 051604 031400 035022 171721 170002 140005 047514   00010  042115 040523 052040 021004 020003 041401 022442 171722
00020  000606 025001 004500 025001 004506 021001 021001 040214   00030  000000 035422 035022 171721 170002 140006 052122 040516
00040  043111 046105 020040 021005 020003 041401 022550 171722   00050  000606 025001 004500 025001 004506 021001 021001 040165
00060  000030 035422 035022 171721 170002 140005 047105 053515   00070  040523 052040 021004 020003 041401 022505 171722 000606
00100  025001 004500 025001 004506 021002 021001 040137 000027   00110  035422 035022 171721 170002 140005 052122 040516 042522
00120  051040 021004 020003 041401 022613 171722 000606 025001   00130  004500 025001 004506 021002 021001 040111 000027 035422
00140  035022 171721 170002 140004 050122 044516 052040 021003   00150  020003 041401 022656 171722 000606 025001 004500 025001
00160  004506 021002 021001 040064 000026 035422 041402 070061   00170  021040 162701 004573 021205 020063 041405 000000 041401
00200  070051 041405 010301 021004 020023 041401 070044 041405   00210  010301 021004 020023 041402 022444 020477 000043 021011
00220  020475 041402 022450 020477 000043 021012 020475 041402   00230  070023 170003 010201 140004 034471 034471 034400 021005
00240  020243 141214 140026 003003 003003 003003 003003 003003   00250  000646 000655 001161 001462 001151 041402 070402 170003
00260  010201 140004 034471 034471 034400 021005 020243 141210   00270  041402 022404 020477 000043 021001 020475 140447 041402
00300  022474 020477 000043 021017 020475 041401 022442 000600   00310  021001 000000 041401 022505 000600 021001 000005 041401
00320  022550 000600 021001 000005 041401 022613 000600 021001   00330  000005 041401 022656 000600 021001 000005 000000 041402
00340  070016 021005 041402 070014 021005 020473 141614 041402   00350  022410 020477 000043 021002 020475 140033 001462 001151
00360  001462 001151 041402 070403 021005 041402 070405 021005   00370  020473 141310 041402 022424 020477 000043 021005 020475
00400  140010 041402 022430 020477 000043 021006 020475 140001   00410  021001 020476 130013 156402 022101 141511 041402 022414
00420  020477 000043 021003 020475 140011 001150 041402 022420   00430  020477 000043 021004 020475 140001 021002 020476 041402
00440  070002 140035 001462 001151 001467 001156 000650 001207   00450  000652 001220 000654 001227 000522 000631 002606 001151
00460  002617 001156 001207 002654 001220 002674 001227 002713   00470  002723 001436 000631 000546 000600 000600 041402 070434
00500  021005 020063 041402 070437 041402 070440 021031 020063   00510  041401 070443 041405 021011 041402 070446 021011 020602
00520  020625 041401 070452 041405 021007 041402 070455 021007   00530  020602 020625 041401 070461 041405 021004 041402 070464
00540  021004 020602 020625 041401 070470 041401 070471 021024   00550  020023 041402 070474 041402 070475 021005 020063 041402
00560  070500 041402 070501 021031 020063 041405 041402 070505   00570  021011 020477 000027 170003 010201 140007 030734 031734
00600  031716 021341 020055 170000 041402 070522 041405 021011   00610  020462 000600 020470 041405 041402 070531 021007 020477
00620  000027 170003 010201 140006 031334 031716 021341 020055   00630  170000 041402 070545 041405 021007 020462 000600 020470
00640  041405 041402 070554 021004 020477 000027 170003 010201   00650  140003 031441 170000 041402 070565 041405 021004 020462
00660  000600 020470 041402 070573 041402 070574 021012 020063   00670  041402 022444 020477 000043 021011 020475 041402 022460
00700  020477 000043 021014 020475 041402 022454 020477 000043   00710  021013 020475 041401 070621 041401 070622 021024 020023
```

PAGE 0010/COBTEXT EXAMPLE

```
00720  130624 177401 004300 045000 045001 000733 001100 130632    00730  177401 004300 055001 055000 021003 020476 041401 070002
00740  140006 000426 000464 001054 000606 000606 041401 070405    00750  021036 020023 000700 041401 022613 041402 070413 000600
00760  021074 000000 000200 041402 022444 020477 000043 021011    00770  020475 130425 177401 004300 045000 045001 000733 001100
01000  130433 177401 004300 055001 055000 021004 020476 041401    01010  070002 140003 000522 000631 041401 070402 021024 020023
01020  041402 022460 020477 000043 021014 020475 041402 022450    01030  020477 000043 021012 020475 021005 020476 130013 156402
01040  022104 141512 041402 022434 020477 000043 021007 020475    01050  140025 001150 001150 130401 156402 022103 141510 041402
01060  022440 020477 000043 021010 020475 140010 041402 022420    01070  020477 000043 021004 020475 140001 021006 020476 041402
01100  070002 140017 002606 001462 002617 001467 000650 002654    01110  000652 002674 000654 002713 002723 001450 000602 000602
01120  041402 070416 021005 020063 041402 070421 041402 070422    01130  021031 020063 041405 021011 021022 071405 021011 041401
01140  070432 021002 020604 004000 020623 170747 010201 041402    01150  070441 041405 021011 020462 000600 020470 041405 021007
01160  021022 071405 021007 041401 070454 021002 020604 004000    01170  020623 170745 010201 041402 070463 041405 021007 020462
01200  000600 020470 041405 021004 021022 071405 021004 041401    01210  070476 021001 020604 004000 020623 170744 010201 041402
01220  070505 041405 021004 020462 000600 020470 041402 070513    01230  041402 070514 021012 020063 041402 022454 020477 000043
01240  021013 020475 041402 022450 020477 000043 021012 020475    01250  041402 022444 020477 000043 021011 020475 130540 177401
01260  004300 045000 045001 000733 001100 130546 177401 004300    01270  055001 055000 021007 020476 000700 041402 070002 140036
01300  001156 001467 001156 001207 000650 001207 001220 000652    01310  001220 001227 000654 001227 002606 001462 002617 001467
01320  000650 177255 002654 000652 177300 002674 000654 177322    01330  002713 002723 001424 000576 000576 021031 020472 141207
01340  041402 070440 041402 070441 021031 020063 000700 041402    01350  070445 021011 020472 141212 041401 070451 041405 021011
01360  041402 070454 021011 020602 020625 000700 041402 070461    01370  021007 020472 141212 041401 070465 041405 021007 041402
01400  070470 021007 020602 020625 000700 041402 070475 021004    01410  020472 141212 041401 070501 041405 021004 041402 070504
01420  021004 020602 020625 041402 070510 041402 070511 021005    01430  020063 041402 070514 041402 070515 021031 020063 041405
01440  021011 021022 071405 021011 041401 070525 021002 020604    01450  004000 020623 172531 010201 041402 070533 041405 021011
01460  020462 000600 020470 041405 021007 021022 071405 021007    01470  041401 070546 021002 020604 004000 020623 172552 010201
01500  041402 070554 041405 021007 020462 000600 020470 041405    01510  021004 021022 071405 021004 041401 070567 021001 020604
01520  004000 020623 172573 010201 041402 070575 041405 021004    01530  020462 000600 020470 041402 070603 041402 070604 021012
01540  020063 041402 022454 020477 000043 021013 020475 041402    01550  022444 020477 000043 021011 020475 130622 177401 004300
01560  045000 045001 000733 001100 130630 177401 004300 055001    01570  055000 021010 020476 000700 041401 022550 041402 070002
01600  140006 001150 001151 001151 000610 000610 021001 000000    01610  000200 141313 041402 070411 170003 010201 140004 034471
01620  034471 034400 021005 020043 041402 070422 170003 010201    01630  140004 034471 034471 034400 021005 020243 141215 130433
01640  177401 004300 045000 045001 000733 001100 130441 177401    01650  004300 055001 055000 021011 020476 000700 041401 022442
01660  041402 070002 140005 001314 001314 000631 000546 021001    01670  000061 000200 141313 041402 070410 170003 010201 140004
01700  034471 034471 034400 021005 020043 041401 070421 041401    01710  070422 021024 020023 021012 020476 130002 140007 000572
01720  000575 000646 002606 000572 000572 047401 130406 047401    01730  001700 141307 041402 022470 020477 000043 021016 020475
01740  041402 070420 041402 070421 021127 020062 021040 162701    01750  004573 021055 020063 041402 022464 020477 000043 021015
01760  020475 130436 047401 003300 130440 057401 021013 020476    01770  000700 041401 022505 041402 070002 140002 001244 000600
02000  021050 001020 000200 021014 020476 041401 022656 041402    02010  070002 140007 000646 000574 000646 000572 000574 000572
02020  000600 021205 130407 047401 013400 000600 000000 041402    02030  070414 021040 162701 004573 021205 020063 130421 047401
02040  130422 077401 130423 057401 021015 020476 041401 022656    02050  041402 070002 140020 000646 000572 000573 000573 000573
02060  001652 000574 000646 001532 000574 000646 001656 000574    02070  000646 002064 000600 021205 021061 000600 000050 000600
02100  130424 057401 130425 047401 003300 130427 057401 041405    02110  021003 021022 071405 021003 041401 070436 021001 020604
02120  004000 020623 170003 010201 140003 031041 170000 041402    02130  070450 041405 021003 020462 000600 020470 021002 130456
02140  057401 041402 070460 041402 070461 021123 020062 021040    02150  162701 004573 021061 020063 041402 022464 020477 000043
02160  021015 020475 021001 130477 057401 041402 070501 041402    02170  070502 021205 020063 041402 022464 020477 000043 021015
02200  020475 021002 130513 057401 041402 070515 041402 070516    02210  021205 020063 041402 022464 020477 000043 021015 020475
02220  021016 020476 130002 140034 000576 000600 000602 000604    02230  000574 000576 002376 000600 002430 000602 002462 000604
02240  002514 000606 002546 000610 002600 000646 002272 000646    02250  002352 000646 002404 000646 002436 000646 002470 177401
02260  004300 045000 045001 130436 177401 004300 045000 045001    02270  001100 130443 177401 004300 045000 045001 001100 130450
02300  177401 004300 055001 055000 021001 130455 057401 041405    02310  021005 021022 071405 021005 041401 070464 021002 020604
02320  004000 020623 170003 010201 140004 031334 031041 170000    02330  041402 070477 041405 021005 020462 000600 020470 041405
02340  021005 021022 071405 021005 041401 070512 021002 020604    02350  004000 020623 170425 010201 041402 070521 041405 021005
02360  020462 000600 020470 041405 021005 021022 071405 021005    02370  041401 070534 021002 020604 004000 020623 170451 010201
02400  041402 070543 041405 021005 020462 000600 020470 041405    02410  021005 021022 071405 021005 041401 070556 021002 020604
02420  004000 020623 170475 010201 041402 070565 041405 021005    02430  020462 000600 020470 041405 021005 021022 071405 021005
02440  041401 070600 021002 020604 004000 020623 170521 010201    02450  041402 070607 041405 021005 020462 000600 020470 041405
02460  021005 021022 071405 021005 041401 070622 021002 020604    02470  004000 020623 170545 010201 041402 070631 041405 021005
02500  020462 000600 020470 041402 070637 041402 070640 021060    02510  020062 021040 162701 004573 021124 020063 041402 022464
02520  020477 000043 021015 020475 041402 070656 041402 070657    02530  021032 020062 021040 162701 004573 021152 020063 041402
```

```
PAGE 0011/COBTEXT  EXAMPLE
 02540  022464 020477 000043 021015 020475 041402 070675 041402    02550  070676 021032 020062 021040 162701 004573 021152 020063
 02560  041402 022464 020477 000043 021015 020475 041402 070714    02570  041402 070715 021032 020062 021040 162701 004573 021152
 02600  020063 041402 022464 020477 000043 021015 020475 041402    02610  070733 041402 070734 021032 020062 021040 162701 004573
 02620  021152 020063 041402 022464 020477 000043 021015 020475    02630  041402 070002 140005 000646 002522 000646 002554 041402
 02640  070404 021032 020062 021040 162701 004573 021152 020063    02650  041402 022464 020477 000043 021015 020475 041402 070422
 02660  041402 070423 021032 020062 021040 162701 004573 021152    02670  020063 041402 022464 020477 000043 021015 020475 021017
 02700  020476 000600 000000

 00000  041604 025404 055000 171400 000500 055000 140011 000000    00010  027054 002766 001415 003042 001362 001357 000000 000000
 00020  040411 004500 010201 035013 041401 004300 045000 045001    00030  100000 004641 031000 051403 027210 055001 071403 055000
 00040  032402 045000 045001 100013 027210 055001 071403 055000    00050  032402 004400 021040 071401 005700 141154 040446 051403
 00060  021044 051404 041402 070452 051405 041401 070454 051406    00070  041402 070456 051407 041401 051411 041401 022440 051412
 00100  021402 041401 070466 041401 070467 004500 061702 141610    00110  041402 077701 057701 043700 026665 002000 140411 000200
 00120  021005 000000 025001 041402 020477 000043 020474
```

```
PAGE 0012/COBTEXT  EXAMPLE        SYMBOL TABLE MAP
LINE# LVL SOURCE NAME                  BASE DISPL   SIZE   USAGE      CATEGORY R O J BZ

FILE SECTION

00036 FD  OLD-INV-MAST                 Q+2: 000104  000106 SEQUENTIAL
00053 01   OLD-INV-MAST-REC            Q+2: 001314  000050 DISP       AN
00054 03   OM-PART-NBR                 Q+2: 001314  000005 DISP       AN
00055 03   FILLER                      Q+2: 001321  000043 DISP       AN
00038 FD  NEW-INV-MAST                 Q+2: 000212  000106 SEQUENTIAL
00061 01   NEW-INV-MAST-REC            Q+2: 001244  000050 DISP       AN
00040 FD  TRAN-FILE                    Q+2: 000320  000106 SEQUENTIAL
00066 01   TRAN-REC                    Q+2: 001150  000074 DISP       AN
00067 03   TR-UPDATE-CODE              Q+2: 001150  000001 DISP       AN
00068 88   TR-ADD-CODE
00069 88   TR-CHANGE-CODE
00070 88   TR-DELETE-CODE
00071 03   TR-PART-NBR                 Q+2: 001151  000005 DISP       AN
00072 03   TR-DESCRIPTION              Q+2: 001156  000031 DISP       AN
00073 03   TR-PART-COST-FLD            Q+2: 001207  000011 DISP       AN
00074 05   TR-PART-COST                Q+2: 001207  000011 DISP       N
00075 03   TR-PART-PRICE-FLD           Q+2: 001220  000007 DISP       AN
00076 05   TR-PART-PRICE               Q+2: 001220  000007 DISP       N
00077 03   TR-PART-QUANTITY-FLD        Q+2: 001227  000004 DISP       AN
00078 05   TR-PART-QUANTITY            Q+2: 001227  000004 DISP       N
00079 03   FILLER                      Q+2: 001233  000011 DISP       AN
00042 FD  ERROR-FILE                   Q+2: 000426  000106 SEQUENTIAL
00084 01   ERROR-REC                   Q+2: 001054  000074 DISP       AN
00044 FD  PRINT-FILE                   Q+2: 000534  000106 SEQUENTIAL
00089 01   PRINT-REC                   Q+2: 000646  000205 DISP       AN

WORKING-STORAGE SECTION

00094 01  WS-PRINT-CONTROL             Q+2: 001364  000010 DISP       AN
00095 03   WS-LINE-CTR                 Q+2: 001364  000002 COMP       NS
00096 03   WS-PAGE-CTR                 Q+2: 001366  000002 COMP       NS
00097 03   WS-SPACING                  Q+2: 001370  000002 COMP       NS
00098 03   WS-LINE-LMT                 Q+2: 001372  000002 COMP       NS
00100 01  WS-ACCUMULATORS              Q+2: 001374  000030 DISP       AN
00101 03   WS-CHANGES-CTR              Q+2: 001374  000004 COMP       NS
00102 03   WS-ADDITIONS-CTR            Q+2: 001400  000004 COMP       NS
00103 03   WS-DELETES-CTR              Q+2: 001404  000004 COMP       NS
00104 03   WS-TOTAL-CTR                Q+2: 001410  000004 COMP       NS
00105 03   WS-ERRORS                   Q+2: 001414  000004 COMP       NS
00106 03   WS-TRANS-READ               Q+2: 001420  000004 COMP       NS
00108 01  WS-UPDT-MESSAGES             Q+2: 001424  000036 DISP       AN
00109 03   WS-CHANGE-MSG               Q+2: 001424  000012 DISP       AN
00110 03   WS-ADDITION-MSG             Q+2: 001436  000012 DISP       AN
00111 03   WS-DELETE-MSG               Q+2: 001450  000012 DISP       AN
00113 01  WS-MASTER-REC                Q+2: 001462  000050 DISP       AN
00114 03   WS-MR-PART-NBR              Q+2: 001462  000005 DISP       AN
00115 03   WS-MR-DESCRIPTION           Q+2: 001467  000031 DISP       AN
00116 03   WS-MR-PART-COST             Q+2: 001520  000004 COMP-SYNC  NS
00117 03   WS-MR-PART-PRICE            Q+2: 001524  000004 COMP-SYNC  NS
00118 03   WS-MR-PART-QUANTITY         Q+2: 001530  000002 COMP-SYNC  NS
00120 01  HDG-1                        Q+2: 001532  000123 DISP       AN
00121 03   HDG1-DATE                   Q+2: 001532  000010 DISP       AN
00122 03   FILLER                      Q+2: 001542  000026 DISP       AN
```

PAGE 0013/COBTEXT EXAMPLE SYMBOL TABLE MAP

LINE#	LVL	SOURCE NAME	BASE	DISPL	SIZE	USAGE	CATEGORY	R	O	J	BZ
00123	03	HDG1-REPORT-NAME	Q+2:	001570	000030	DISP	AN				
00125	03	FILLER	Q+2:	001620	000024	DISP	AN				
00126	03	FILLER	Q+2:	001644	000006	DISP	AN				
00127	03	HDG1-PAGE-NBR	Q+2:	001652	000003	DISP	NE				
00129	01	HDG-2	Q+2:	001656	000205	DISP	AN				
00130	03	FILLER	Q+2:	001656	000024	DISP	AN				
00132	03	FILLER	Q+2:	001702	000024	DISP	AN				
00134	03	FILLER	Q+2:	001726	000024	DISP	AN				
00136	03	FILLER	Q+2:	001752	000024	DISP	AN				
00138	03	FILLER	Q+2:	001776	000024	DISP	AN				
00140	03	FILLER	Q+2:	002022	000024	DISP	AN				
00142	03	FILLER	Q+2:	002046	000015	DISP	AN				
00145	01	HDG-3	Q+2:	002064	000205	DISP	AN				
00146	03	FILLER	Q+2:	002064	000024	DISP	AN				
00148	03	FILLER	Q+2:	002110	000024	DISP	AN				
00150	03	FILLER	Q+2:	002134	000024	DISP	AN				
00152	03	FILLER	Q+2:	002160	000024	DISP	AN				
00154	03	FILLER	Q+2:	002204	000024	DISP	AN				
00156	03	FILLER	Q+2:	002230	000024	DISP	AN				
00158	03	FILLER	Q+2:	002254	000015	DISP	AN				
00161	01	TOTALS-HDG-1	Q+2:	002272	000060	DISP	AN				
00162	03	FILLER	Q+2:	002272	000024	DISP	AN				
00164	03	FILLER	Q+2:	002316	000024	DISP	AN				
00166	03	TOT1-HDG-DATE	Q+2:	002342	000010	DISP	AN				
00168	01	TOTALS-HDG-2	Q+2:	002352	000032	DISP	AN				
00169	03	FILLER	Q+2:	002352	000024	DISP	AN				
00171	03	TOT2-CHANGES	Q+2:	002376	000006	DISP	NE				
00173	01	TOTALS-HDG-3	Q+2:	002404	000032	DISP	AN				
00174	03	FILLER	Q+2:	002404	000024	DISP	AN				
00176	03	TOT3-ADDITIONS	Q+2:	002430	000006	DISP	NE				
00178	01	TOTALS-HDG-4	Q+2:	002436	000032	DISP	AN				
00179	03	FILLER	Q+2:	002436	000024	DISP	AN				
00181	03	TOT4-DELETIONS	Q+2:	002462	000006	DISP	NE				
00183	01	TOTALS-HDG-5	Q+2:	002470	000032	DISP	AN				
00184	03	FILLER	Q+2:	002470	000024	DISP	AN				
00186	03	TOT5-UPDATES	Q+2:	002514	000006	DISP	NE				
00188	01	TOTALS-HDG-6	Q+2:	002522	000032	DISP	AN				
00189	03	FILLER	Q+2:	002522	000024	DISP	AN				
00191	03	TOT6-ERRORS	Q+2:	002546	000006	DISP	NE				
00193	01	TOTALS-HDG-7	Q+2:	002554	000032	DISP	AN				
00194	03	FILLER	Q+2:	002554	000024	DISP	AN				
00196	03	TOT7-TRANS-READ	Q+2:	002600	000006	DISP	NE				
00198	01	WS-UPDATE-LINE	Q+2:	002606	000127	DISP	AN				
00199	03	WS-UP-PART-NBR	Q+2:	002606	000005	DISP	AN				
00200	03	FILLER	Q+2:	002613	000004	DISP	AN				
00201	03	WS-UP-DESCRIPTION	Q+2:	002617	000031	DISP	AN				
00202	03	FILLER	Q+2:	002650	000004	DISP	AN				
00203	03	WS-UP-PART-COST	Q+2:	002654	000015	DISP	NE				
00204	03	FILLER	Q+2:	002671	000003	DISP	AN				
00205	03	WS-UP-PART-PRICE	Q+2:	002674	000012	DISP	NE				
00206	03	FILLER	Q+2:	002706	000005	DISP	AN				
00207	03	WS-UP-PART-QUANTITY	Q+2:	002713	000004	DISP	NE				
00208	03	FILLER	Q+2:	002717	000004	DISP	AN				
00209	03	WS-UP-UPDT-MESSAGE	Q+2:	002723	000012	DISP	AN				

```
COBTEXT  EXAMPLE         SYMBOL TABLE MAP
 LINE# LVL SOURCE NAME                    BASE DISPL   SIZE   USAGE     CATEGORY R O J BZ

 STORAGE LAYOUT              (#ENTRYS)    (VALUES IN WORDS)

          START TABLE           (16)     Q+1: 000000  000040
          USER LABEL POINTER             Q+1: 000040  000002
          FILE TABLE            (5)      Q+1: 000042  000257
          TALLY                          Q+1: 000321  000002
          USER STORAGE                   Q+1: 000323  001036
          RUNNING PICTURES               Q+1: 001357  000003
          FIXUP AREA            (1)      Q+1: 001362  000011
          9 WORD TEMP CELLS     (2)      Q+1: 001373  000022
          1 WORD TEMP CELLS     (4)      Q+1: 001415  000004

 POINTER AREA

          DB-5  CURRENT VALUE OF Q FOR STORAGE AREA
          DB-4  'PARM=' WORD - SWITCHES
          Q+1   WORD ADDRESS OF STORAGE AREA
          Q+2   BYTE ADDRESS OF STORAGE AREA
          Q+3   DECIMAL POINT & COMMA
          Q+4   # PARMS AND CURRENCY SIGN
          Q+5   BYTE ADDRESS OF 9 WORD TEMPCELLS
          Q+6   WORD ADDRESS OF 1 WORD TEMPCELLS
          Q+7   BYTE ADDRESS OF LITERAL POOL
          Q+10  PLABEL OF SORT OR MERGE OUTPUT
          Q+11  WORD ADDRESS OF START TABLE
          Q+12  WORD ADDRESS OF USER LABEL POINTER
          Q+13  PREVIOUS VALUE OF DB-5
          Q+14  RESERVED
```

```
 LINE #    PB-LOC         PROCEDURE NAME/VERB                    INTERNAL NAME
-
 00213     000003          100-START-OF-PROGRAM                  100STARTOFPR00'
 00218     000003              OPEN
 00220     000166              MOVE
 00222     000175              MOVE
 00226     000213              PERFORM
 00230     000221              PERFORM
 00235     000227              PERFORM
 00236     000255              PERFORM
 00237     000277              PERFORM
 00243     000305              CLOSE
 00244     000336              STOP
 00246     000337          200-MATCH-MAST-VS-TRAN
 00248     000337              IF
 00250     000347              PERFORM
 00251     000362              IF
 00253     000372              PERFORM
 00254     000401              PERFORM
 00255     000412          210-MASTER-COMPARED-HIGH
 00257     000412              IF
 00259     000416              PERFORM
 00260     000426              PERFORM
 00262     000437          220-ADD-TO-MASTER
 00263     000437              MOVE
 00264     000502              MOVE
 00265     000510              MOVE
 00266     000521              MOVE
 00267     000532              MOVE
 00268     000543              MOVE
 00269     000551              MOVE
 00270     000557              MOVE
 00271     000565              MOVE
 00272     000613              MOVE
 00273     000640              MOVE
 00274     000662              MOVE
 00275     000670              PERFORM
 00276     000676              PERFORM
 00277     000704              PERFORM
 00278     000712              MOVE
 00279     000720              ADD
 00281     000736          230-TRAN-IN-ERROR
 00282     000736              MOVE
 00283     000752              WRITE
 00284     000762              PERFORM
 00285     000771              ADD
 00287     001007          240-MASTER-COMPARED-LOW
 00288     001007              MOVE
 00289     001020              PERFORM
 00290     001026              PERFORM
 00292     001036          250-MASTER-AND-TRAN-EQUAL
 00293     001036              IF
 00295     001042              PERFORM
 00296     001053              IF
 00298     001057              PERFORM
 00299     001066              PERFORM
 00301     001077          260-DELETE-MASTER
```

```
 LINE #    PB-LOC        PROCEDURE NAME/VERB                INTERNAL NAME
-
 00302     001077              MOVE
 00303     001124              MOVE
 00304     001132              MOVE
 00305     001156              MOVE
 00306     001202              MOVE
 00307     001226              MOVE
 00308     001234              PERFORM
 00309     001242              PERFORM
 00310     001250              PERFORM
 00311     001256              ADD
 00313     001274          270-CHANGE-MASTER
 00315     001274              IF
 00316     001340              MOVE
 00317     001346              IF
 00318     001354              MOVE
 00319     001365              IF
 00320     001373              MOVE
 00321     001404              IF
 00322     001412              MOVE
 00324     001423              MOVE
 00325     001431              MOVE
 00326     001437              MOVE
 00327     001463              MOVE
 00328     001507              MOVE
 00329     001533              MOVE
 00330     001541              PERFORM
 00331     001547              PERFORM
 00332     001555              ADD
 00334     001573          300-GET-TRANSACTION
 00337     001573              READ
 00337     001612              MOVE
 00339     001624              IF
 00340     001637              ADD
 00342     001655          310-GET-OLD-MASTER
 00345     001655              READ
 00345     001673              MOVE
 00347     001705              MOVE
 00349     001715          320-PRINT-UPDATE
 00351     001715              IF
 00352     001732              PERFORM
 00354     001740              MOVE
 00355     001753              PERFORM
 00356     001761              ADD
 00358     001770          330-WRITE-NEW-MASTER
 00359     001770              WRITE
 00361     002005          400-WRITE-PRINT-LINE
 00362     002005              WRITE
 00363     002027              MOVE
 00364     002036              ADD
 00366     002046          410-PRINT-HEADING
 00367     002046              WRITE
 00368     002077              MOVE
 00369     002102              ADD
 00370     002107              MOVE
 00371     002136              MOVE
```

```
PAGE 0016/COBTEXT  EXAMPLE         SYMBOL TABLE MAP
 LINE #    PB-LOC        PROCEDURE NAME/VERB                    INTERNAL NAME
-
 00372     002141              MOVE
 00373     002154              PERFORM
 00374     002162              MOVE
 00375     002165              MOVE
 00376     002173              PERFORM
 00377     002201              MOVE
 00378     002204              MOVE
 00379     002212              PERFORM
 00381     002222          420-PRINT-TOTALS
 00383     002222              ADD
 00386     002304              MOVE
 00387     002307              MOVE
 00388     002337              MOVE
 00389     002363              MOVE
 00390     002407              MOVE
 00391     002433              MOVE
 00392     002457              MOVE
 00394     002503              MOVE
 00395     002516              PERFORM
 00396     002524              MOVE
 00397     002537              PERFORM
 00398     002545              MOVE
 00399     002560              PERFORM
 00400     002566              MOVE
 00401     002601              PERFORM
 00402     002607              MOVE
 00403     002622              PERFORM
 00404     002630              MOVE
 00405     002650              PERFORM
 00406     002656              MOVE
 00407     002671              PERFORM
```

```
COBOL CROSS REFERENCE LISTING
:

-
100-START-OF-PROGRAM
00213
-
200-MATCH-MAST-VS-TRAN
00246   00236
-
210-MASTER-COMPARED-HIGH
00255   00250
-
220-ADD-TO-MASTER
00262   00259
-
230-TRAN-IN-ERROR
00281   00260   00299   00299
-
240-MASTER-COMPARED-LOW
00287   00253
-
250-MASTER-AND-TRAN-EQUAL
00292   00254
-
260-DELETE-MASTER
00301   00295
-
270-CHANGE-MASTER
00313   00298
-
300-GET-TRANSACTION
00334   00226   00275   00284   00310   00331
-
310-GET-OLD-MASTER
00342   00230   00290   00309
-
320-PRINT-UPDATE
00349   00277   00308   00330
-
330-WRITE-NEW-MASTER
00358   00276   00289
-
400-WRITE-PRINT-LINE

00361   00355   00373   00373   00376   00376   00379   00379   00395   00395   00397   00397
00399   00399   00401   00401   00403   00403   00405   00405   00407   00407
-
410-PRINT-HEADING
00366   00352
-
420-PRINT-TOTALS
00381   00237
-
ERROR-FILE
00042   00082   00218   00243
-
```

```
ERROR-REC
00084    00083    00282    00283
-
HDG-1
00120    00372
-
HDG-2
00129    00375
-
HDG-3
00145    00378
-
HDG1-DATE
00121    00222
-
HDG1-PAGE-NBR
00127    00370
-
HDG1-REPORT-NAME
00123
-
NEW-INV-MAST
00038    00058    00217    00241
-
NEW-INV-MAST-REC
00061    00059    00268    00288    00359
-
OLD-INV-MAST
00036    00050    00215    00240    00344
-
OLD-INV-MAST-REC
00053    00278    00347
-
OM-PART-NBR
00054    00345
-
PRINT-FILE
00044    00087    00218    00243
-
PRINT-REC

00089    00088    00220    00354    00362    00363    00367    00372    00375    00378    00394    00396
00398    00400    00402    00404    00406
-
TOT1-HDG-DATE
00166    00222
-
TOT2-CHANGES
00171    00387
-
TOT3-ADDITIONS
00176    00388
-
TOT4-DELETIONS
00181    00389
-
TOT5-UPDATES
00186    00390
```

```
TOT6-ERRORS
00191    00391
-
TOT7-TRANS-READ
00196    00392
-
TOTALS-HDG-1
00161    00394
-
TOTALS-HDG-2
00168    00396
-
TOTALS-HDG-3
00173    00398
-
TOTALS-HDG-4
00178    00400
-
TOTALS-HDG-5
00183    00402
-
TOTALS-HDG-6
00188    00404
-
TOTALS-HDG-7
00193    00406
-
TR-ADD-CODE
00068    00258
-
TR-CHANGE-CODE
00069    00297
-
TR-DELETE-CODE
00070    00294
-
TR-DESCRIPTION
00072    00264    00270    00315    00316
-
TR-PART-COST
00074    00265    00271    00318
-
TR-PART-COST-FLD
00073    00317
-
TR-PART-NBR
00071    00236    00249    00252    00263    00269    00337    00339
-
TR-PART-PRICE
00076    00266    00272    00320
-
TR-PART-PRICE-FLD
00075    00319
-
TR-PART-QUANTITY
00078    00267    00273    00322
-
```

```
TR-PART-QUANTITY-FLD
00077   00321
-
TR-UPDATE-CODE
00067
-
TRAN-FILE
00040   00064   00216   00242   00336
-
TRAN-REC
00066   00065   00282
-
WS-ACCUMULATORS
00100
-
WS-ADDITION-MSG
00110   00274
-
WS-ADDITIONS-CTR
00102   00279   00384   00388
-
WS-CHANGE-MSG
00109   00329
-
WS-CHANGES-CTR
00101   00332   00383   00387
-
WS-DELETE-MSG
00111   00307
-
WS-DELETES-CTR
00103   00311   00384   00389
-
WS-ERRORS
00105   00285   00391
-
WS-LINE-CTR
00095   00351   00356   00364   00368
-
WS-LINE-LMT
00098   00352
-
WS-MASTER-REC
00113   00268   00278   00288   00347
-
WS-MR-DESCRIPTION
00115   00264   00303   00316   00325
-
WS-MR-PART-COST
00116   00265   00304   00318   00326
-
WS-MR-PART-NBR
00114   00235   00248   00251   00263   00302   00324
-
WS-MR-PART-PRICE
00117   00266   00305   00320   00327
-
```

```
WS-MR-PART-QUANTITY
00118   00267   00306   00322   00328
-
WS-PAGE-CTR
00096   00369   00370
-
WS-PRINT-CONTROL
00094
-
WS-SPACING
00097   00362   00364   00371   00374   00377   00386
-
WS-TOTAL-CTR
00104   00384   00390
-
WS-TRANS-READ
00106   00340   00392
-
WS-UP-DESCRIPTION
00201   00270   00303   00325
-
WS-UP-PART-COST
00203   00271   00304   00326
-
WS-UP-PART-NBR
00199   00269   00302   00324
-
WS-UP-PART-PRICE
00205   00272   00305   00327
-
WS-UP-PART-QUANTITY
00207   00273   00306   00328
-
WS-UP-UPDT-MESSAGE
00209   00274   00307   00329
-
WS-UPDATE-LINE
00198   00354
-
WS-UPDT-MESSAGES
00108
:
0 ERRORS, 0 QUESTIONABLE,  0 WARNINGS
-
    DATA AREA IS %001421 WORDS.
    CPU TIME = 0:00:41.  WALL TIME = 0:01:50.

END OF PROGRAM
```

COBOL II/3000 ERROR MESSAGES

APPENDIX D

Compile-Time Error Messages

Note: All requests to "contact H-P" in error messages should be interpreted to mean "Please submit a service request (SR)".

DETERMINING THE SEVERITY OF COMPILER ERROR MESSAGES

Compile time error messages have been divided into 6 categories, as outlined in the table below.

Message No.	Classification
1-99	WARNINGS
100-399	QUESTIONABLE CONSTRUCTS
400-449	SERIOUS ERRORS
450-499	DISASTEROUS ERRORS
500-899	NONSTANDARD WARNINGS
900-999	INFORMATIONAL MESSAGES

These classifications have the following meaning:

WARNING — Something is incorrect in the code but the compiler can probably fix it to produce what the user intended.

QUESTIONABLE CONSTRUCT — An error has occurred and it will be fixed, but probably not produce what the user intended.

SERIOUS — An error has been detected which is either too difficult or impossible to fix. No code is generated.

DISASTEROUS — Something has occurred that makes further processing risky or impossible. All files are closed and processing is stopped immediately.

NONSTANDARD — The program uses a construct that is not part of the given level of the FIPS COBOL '74 standard.

JOB CONTROL WORD BIT SETTINGS

Serious and disasterous errors will cause the setting of the fatal bits of the JCW.

Warnings and questionable constructs will cause the setting of the warning bits of the JCW.

ERROR NUMBER	ERROR MESSAGE	DISCUSSION
001	ILLEGAL CHARACTER IN COLUMN 7.	Only space, *, –, $, or / allowed.
002	DEBUGGING LINE ILLEGAL BEFORE SOURCE-COMPUTER PARAGRAPH.	
003	TOO MANY CHARACTERS IN SYMBOL.	Symbol or picture string is limited to 30 characters.
004	MISSING SPACE.	Separator space is needed here.
005	CONTINUATION RECORD NOT ALLOWED HERE.	
006	MISSING QUOTE.	Quote is needed in nonnumeric literal.
007	INTEGER BEFORE SYSTEM FILE NAME NOT IMPLEMENTED.	
008	$LIST SHOULD BE REPLACED BY $CONTROL	
009	FILES IN MULTIPLE FILE TAPE CLAUSE MUST BE SEQUENTIAL.	
010	MULTIPLE SYSTEM FILE NAMES NOT IMPLEMENTED.	
011	RERUN NOT IMPLEMENTED.	
012	ADVANCING OPTION REQUIRES NON-NEGATIVE NUMBER OF LINES.	A "WRITE ADVANCING" specified a negative number of lines to advance. Code for the write will not be generated.
013	ERROR CONVERTING THE WRITE ADVANCING COUNT TO AN INTEGER.	A "WRITE ADVANCING" specified a number of lines to advance that cause an error when converting to integer. A value of 1 is used in this case.
015	DELETE VALID ONLY WITH RELATIVE OR INDEXED.	The "DELETE" verb can only be used with RELATIVE or INDEXED I/O.
016	START VALID ONLY WITH RELATIVE, RANDOM OR INDEXED.	The "START" verb can only be used with RELATIVE or INDEXED or RANDOM I/O.
017	FILE NAME IS UNDEFINED OR IS NOT UNIQUE.	The file name in this statement is not found in a SELECT or FD statement, or there exists a data name by the same name.
018	SEEK VALID ONLY WITH RELATIVE OR RANDOM.	The "SEEK" verb can only be used with RANDOM or RELATIVE files.
019	INVALID RECORD NAME IN WRITE STATEMENT.	The record name in this write statement is not an "01" level item.

ERROR NUMBER	ERROR MESSAGE	DISCUSSION
020	INVALID DATA TYPE FOR KEY.	Sort or Merge keys may only be of the following types: ALPHABETIC, ALPHANUMERIC, NUMERIC, OR DISPLAY.
022	INVALID ALPHABET NAME.	The name given in the "COLLATING SEQUENCE" clause is not a valid "ALPHABET NAME".
023	START STATEMENT MUST REFERENCE A VALID KEY.	The "START STATEMENT" must reference a data name which can be a valid INDEXED I/O key.
024	A SECTION NAME IS REQUIRED.	The "INPUT PROCEDURE" and/or "OUTPUT PROCEDURE" specifies a name which is not a "SECTION" name.
025	COLLATING SEQUENCE HAS NOT BEEN DEFINED.	Alphabet name for collating sequence is not specified
026	HYPHEN NOT ALLOWED AT END OF WORD.	
027	SPACE NOT ALLOWED IN THIS POSITION.	Embedded space in numeric literal.
028	HIGH-VALUE/LOW-VALUE HAS NOT BEEN DEFINED.	HIGH-VALUE/LOW-VALUE not defined because of undefined collating sequence.
029	OPEN REVERSED NOT SUPPORTED.	Open is generated but "REVERSED" is ignored.
030	NON-88 LEVEL ITEM IN FILE SECTION HAS VALUE CLAUSE.	The value clause is accepted by this compiler.
031	VALUE CLAUSE ON NON-88 LEVEL ITEM IN LINKAGE SECTION IGNORED.	
032	OCCURS CLAUSE USED ON 01 LEVEL ITEM.	
033	VALUE CLAUSE IN AN ITEM SUBORDINATE TO AN OCCURS TABLE.	The VALUE clause is not permitted within an entry subordinate to an OCCURS.
034	MISSING INVALID KEY PHRASE.	An INVALID KEY phrase is required since no applicable USE procedure is specified.
035	ILLEGAL INVALID KEY PHRASE.	An INVALID KEY phrase is not legal with the access mode specified for the file.
040	BLANK LINE WITH CONTINUATION CHARACTER WAS NOT PROCESSED.	
041	MISSING PERIOD IN IDENTIFICATION DIVISION.	

ERROR NUMBER	ERROR MESSAGE	DISCUSSION
042	INVALID COMMENT ENTRY.	Comment entry is not allowed here.
043	ILLEGAL IDENTIFICATION DIVISION PARAGRAPH.	Check allowed IDENTIFICATION DIVISION paragraphs.
044	LIBRARY TEXT NAME LESS THAN EIGHT CHARACTERS.	Message is never used.
045	UNPROCESSED SOURCE ON SUBSYSTEM COMMAND LINE.	Unprocessed source is ignored.
046	SUBSYSTEM COMMAND CHARACTER STRING WAS TRUNCATED.	String exceeds maximum allowed length.
047	SEQUENCING ERROR.	Sequence numbers are out of order.
048	MISSING "BY" IN COPY STATEMENT.	COPY statement is incomplete.
049	INVALID SUBSYSTEM COMMAND DELIMITER; EXPECTED !.	
050	ARITHMETIC OVERFLOW MAY OCCUR.	An intermediate or final result may have more than 28 digits when max. Operands and intermediate results are assumed in the arithmetic statement.
051	REDEFINING ITEM IS SMALLER THAN REDEFINED ITEM.	Except for level 01, a redefining item must be the same size as the item it redefines.
052	REDEFINING ITEM IS BIGGER THAN REDEFINED ITEM.	Except for level 01, a redefining item must be the same size as the item it redefines.
053	REDEFINING ITEM DOES NOT IMMEDIATELY FOLLOW REDEFINED ITEM.	Between an item containing a redefines clause and the item it redefines there must not be any entries which define new character positions.
054	CODE-SET CLAUSE SPECIFIED FOR A MASS-STORAGE FILE.	The code-set clause may not be specified for mass-storage files.
055	LEFT TRUNCATION MAY OCCUR.	This warning is generated whenever significant digits will be truncated in a numeric move.
056	VALUE-OF CLAUSE NOT APPLICABLE TO NON-SEQUENTIAL FILES--IGNORED.	The value-of clause if meaningful only for sequential files. If it occurs in any other type of file it is ignored.
100	ILLEGAL INTEGER.	A non-integer literal or an integer which is too large has occurred in a context where the compiler expects an integer.

ERROR NUMBER	ERROR MESSAGE	DISCUSSION
101	ALPHABET-NAME IN CODE-SET CLAUSE MAY NOT SPECIFY LITERAL PHRASE.	An alphabet-name which was defined using the literal phrase has been used in a code-set clause.
102	DATA-NAME IN CODE-SET CLAUSE IS NOT AN ALPHABET-NAME.	The data-name used in a code-set clause must be declared to be an alphabet-name.
103	DATA-NAME IN CODE-SET CLAUSE IS NOT DECLARED.	The data-name used in a code-set clause must be declared to be an alphabet-name.
109	BAD RECORDING-MODE SPECIFICATION.	The recording-mode specification must be "F", "V", "U", or "S".
112	NEGATIVE NUMBER OF OCCURRENCES SPECIFIED.	A table may not contain a negative number of occurrences.
113	MINIMUM NUMBER OF OCCURRENCES IS GREATER THAN MAXIMUM NUMBER OF OCCURENCES.	The minimum number of occurrences in a table must not be greater than the maximum number of occurrences of the table.
114	INPUT CLAUSE SPECIFIED MORE THAN ONCE.	The same clause has been used more than once in one input communication-description-entry.
115	OUTPUT CLAUSE SPECIFIED MORE THAN ONCE.	The same clause has been used more than once in one output communication-description-entry.
122	MORE THAN 18 DIGITS IN A NUMERIC PICTURE.	Numeric data items can have at most 18 digits.
123	MULTIPLE OCCURRENCES OF BLANK WHEN ZERO CLAUSE IN DATA ITEM.	Only one blank when zero clause may be specified for a single data item.
124	MULTIPLE OCCURRENCES OF JUSTIFIED CLAUSE IN DATA ITEM.	Only one justified clause may be specified for a single data item.
126	MULTIPLE OCCURRENCES OF OCCURS CLAUSE IN DATA ITEM.	Only one occurs clause may be specified for a single data item.
127	MULTIPLE OCCURRENCES OF PICTURE CLAUSE IN DATA ITEM.	Only one picture clause may be specified for a single data item.
128	MULTIPLE OCCURRENCES OF SYNCHRONIZED CLAUSE IN DATA ITEM.	Only one synchronized clause may be specified for a single data item.
129	MULTIPLE OCCURRENCES OF USAGE CLAUSE IN DATA ITEM.	Only one usage clause may be specified for a single data item.
130	MULTIPLE OCCURRENCES OF SIGN CLAUSE IN DATA ITEM.	Only one sign claue may be specified for a single data item.

ERROR NUMBER	ERROR MESSAGE	DISCUSSION
131	MULTIPLE OCCURRENCES OF VALUE CLAUSE IN DATA ITEM.	Only one value clause may be specified for a single data item.
133	BLANK WHEN ZERO CLAUSE MAY NOT BE SPECIFIED IN 88 LEVEL ENTRIES.	A blank when zero clause has been specified for a condition-name.
134	JUSTIFIED CLAUSE MAY NOT BE SPECIFIED IN 88 LEVEL ENTRIES.	A justified clause has been specified for a condition-name.
135	OCCURS CLAUSE MAY NOT BE SPECIFIED IN 88 LEVEL ENTRIES.	An occurs clause has been specified for a condition-name.
136	PICTURE CLAUSE MAY NOT BE SPECIFIED IN 88 LEVEL ENTRIES.	A picture clause has been specified for a condition-name.
137	SYNCHRONIZED CLAUSE MAY NOT BE SPECIFIED IN 88 LEVEL ENTRIES.	A synchronized clause has been specified for a condition-name.
138	USAGE CLAUSE MAY NOT BE SPECIFIED IN 88 LEVEL ENTRIES.	A usage clause has been specified for a condition-name.
139	SIGN CLAUSE MAY NOT BE SPECIFIED IN 88 LEVEL ENTRIES.	A sign clause has been specified for a condition-name.
140	MULTIPLE SIGN DESIGNATORS IN PICTURE.	"+", "–", "S", "CR", and "DB" are mutually exclusive in a picture string.
141	C NOT FOLLOWED BY R IN PICTURE.	"C" must be immediately followed by a "R" in a picture string.
142	D NOT FOLLOWED BY B IN PICTURE.	"D" must be immediately followed by a "B" in a picture string.
143	MULTIPLE POINT CHARACTERS IN PICTURE.	The decimal point location may be specified at most once in a picture string.
144	MULTIPLE FLOAT CHARACTERS IN PICTURE.	Either "*" or "Z" has been encountered in a picture string and some other character has already been determined to be the floating insertion character ("*", "Z", "+" used as floating insertion character, "–" used as floating insertion character, and currency sign used as floating insertion character are mutually exclusive in a picture string).
145	MULTIPLE REPETITION FACTORS IN PICTURE.	"(" immediately follows ")" in a picture string.

ERROR NUMBER	ERROR MESSAGE	DISCUSSION
146	MISSING ")" IN PICTURE REPETITION FACTOR.	"(" has occurred in a picture string without a following ")".
147	ILLEGAL CHARACTER IN PICTURE REPETITION FACTOR.	Only numeric digits ("0" - "9") may occur between "(" and ")" in a picture string.
148	REPETITION FACTOR WITH NO CHARACTER TOREPEAT IN PICTURE.	A picture string may not begin with the character "(".
149	REPETITION FACTOR IN PICTURE MAY NOT BE ZERO.	The number between a "(" and a ")" in a picture string may not be zero.
150	ILLEGAL REPETITION FACTOR IN PICTURE.	A repetition factor greater than one follows a character which may not occur more than once in a picture string.
151	ILLEGAL CHARACTER IN PICTURE.	Only the characters "B", "0", "/", ",", ".", "+", "–", "C", "R"(following a "C"), "D", "B"(following a "D"), the currency sign character, "Z", "*", "9", "A", "X", "S", "V", "P", "(", numeric digits following a "(", and ")" following the numeric digits following a "(" may occur in a picture string.
152	MULTIPLE NON-FLOATING CURRENCY SIGNS IN PICTURE.	More than one currency sign character has occurred in a picture string and some other character has been determined to be the float character.
153	NO DIGIT OR CHARACTER POSITIONS IN PICTURE.	At least one of the characters "A", "X", "Z", "9", or "*" or at least two of the characters "+", "–", or the currency symbol must occur in a picture string.
154	ILLEGAL COMBINATION OF PICTURE CHARACTERS.	The only clause which may be specified for a 66-level entry is the renames clause. Certain combinations of characters may not occur in a picture string.
155	ILLEGAL SEQUENCE OF PICTURE CHARACTERS.	Certain sequences of characters may not occur in a picture string.
156	66-LEVEL ENTRY HAS NO RENAMES CLAUSE.	66-level data items must have a renames clause.
157	88-LEVEL ENTRY HAS NO VALUE CLAUSE.	A value clause must be specified for every condition-name.
158	ELEMENTARY ITEM HAS NO PICTURE.	A 77-level or 49-level item is not usage index and has no picture clause.

ERROR NUMBER	ERROR MESSAGE	DISCUSSION
159	IMPROPER LEVEL NUMBER.	Either: 1) a level number is not between 01 and 49 or =66 or =77 or =88 2) a level number is immediately subordinate to a level whose subordinates are not equal to it, e.g.: 03 ..... 05 .... 05 .... 04 the level 04 is improper 3) a level 77 is in the file section 4) a level 88 is subordinate to an index item 5) the first level number in a section is not 01 or 77 6) the first level number subordinate to an FD or SD is not 01 7) a level 88 is subordinate to a 66 level item 8) a level 66 is subordinate to a 77 level item
160	ILLEGAL CLAUSE FOR 66-LEVEL ENTRY.	
161	ILLEGAL REDEFINES CLAUSE.	The item being redefined is not declared immediately subordinate to the item containing the redefines clause or, the item being redefined is a table or variable size item or, the item being redefined is a 66-level or 88-level item.
162	PICTURE CLAUSE IS ILLEGAL IN 66 AND 88 LEVEL ENTRIES.	A picture clause has occurred in a 66-level or 88-level entry.
163	USAGE CLAUSE CONFLICTS WITH GROUP USAGE CLAUSE.	The usage clause of an item must specify the same usage as any usage clause in any group containing it.
164	SIGN CLAUSE CONFLICTS WITH GROUP SIGN CLAUSE.	The sign clause may not be specified for any item whose group contains a sign clause.

ERROR NUMBER	ERROR MESSAGE	DISCUSSION
165	SIGN CLAUSE CONFLICTS WITH USAGE.	The sign clause may only be specified for items whose usage is display.
166	JUSTIFIED CLAUSE IS ILLEGAL IN INDEX ITEMS.	The justified clause may not be specified for items whose usage is index.
167	PICTURE CLAUSE IS ILLEGAL IN INDEX ITEMS.	The picture clause may not be specified for items whose usage is index.
168	VALUE CLAUSE IS ILLEGAL IN INDEX ITEMS.	The value clause may not be specified for items whose usage is index.
169	BLANK-WHEN-ZERO CLAUSE IS ILLEGAL IN INDEX ITEMS.	The blank when zero clause may not be specified for items whose usage is index.
170	ILLEGAL RENAMES CLAUSE.	A renames clause may only be specified in a 66-level item. The item(s) it renames must be defined in the immediately preceeding record and must not be special level (66,77,88) items, table items, table elements, or variable size items. If the "thru" phrase is used, the names must specify different items, the beginning of the second item may not be before the beginning of the first item, and the end of the second item must be after the end of the first item.
172	JUSTIFIED CLAUSE IS ILLEGAL IN DATA ITEMS WHICH ARE NOT EITHER ALPHABETIC OR ALPHANUMERIC.	The justified clause may only be specified for alphabetic and alphanumeric items.
173	BLANK-WHEN-ZERO CLAUSE IS ILLEGAL FOR THIS ITEM.	The blank when zero clause is legal only for items whose picture is numeric or numeric edited and does not contain the character "*".
174	BLANK-WHEN-ZERO CLAUSE IS REDUNDANT FOR THIS ITEM.	If all of the numeric character positions of a numeric edited item are represented by "Z" then the item is implicitly blank when zero.
175	EDIT PROGRAM FOR THIS PICTURE IS TOO BIG.	If a picture is excessively complicated, it can generate an edit program which will be too long to fit into a data table entry.
176	OCCURS CLAUSE IS ILLEGAL IN 77-LEVEL ITEMS.	The occurs clause may not be specified in a 77-level item.
177	ILLEGAL PICTURE FOR NON-DISPLAY USAGE.	If the usage of an item is comp or comp-3 then the picture must be numeric.
178	BLANK CLAUSE IS ILLEGAL IN GROUP ITEMS.	The blank when zero clause may only be specified for elementary items.

ERROR NUMBER	ERROR MESSAGE	DISCUSSION
179	JUSTIFIED CLAUSE IS ILLEGAL IN GROUP ITEMS.	The justified clause may only be specified for elementary items.
180	SYNCHRONIZED CLAUSE IS ILLEGAL IN GROUP ITEMS.	The synchronized clause may only be specified for elementary items.
183	ILLEGAL SIGN IN LITERAL.	A numeric literal in a value clause may not contain a sign if the corresponding data item is an unsigned numeric data item.
184	ILLEGAL LITERAL.	A literal has occurred in a value clause and either the corresponding data item is an index item, the corresponding data item is numeric and the literal is non-numeric, or the corresponding data item is non-numeric and the literal is numeric.
185	MULTIPLE INITIAL VALUES FOR A DATA ITEM.	When the value clause is used to specify an initial value for a data item it may only specify one value.
186	! ! FOR !.	This error message is for illegal or missing forward references. The insertions are for the forward reference type, the name of the forward reference, and the name of the file or table containing the forward reference. The forward reference types are table keys, alternate keys, depending on variables, file status identifiers, volume identifiers, labels identifiers, seq identifiers, exdate identifiers, linage identifiers, footing identifiers, top identifiers, and bottom identifiers.
188	LITERAL REQUIRES TRUNCATION OF NON-ZERO DIGITS.	A numeric literal in a value clause specified a value outside the range of values possible for the associated data item.
189	LITERAL TOO LONG--TRUNCATED.	A non-numeric literal in a value clause is longer than the associated data item.
190	MULTIPLE OCCURRENCES OF BLOCK CLAUSE IN FILE DESCRIPTION.	Only one block clause may be specified in a file description.
191	MULTIPLE OCCURRENCES OF DATA-RECORDS CLAUSE IN FILE DESCRIPTION.	Only one data records clause may be specified in a file description.
192	MULTIPLE OCCURRENCES OF LABEL-RECORDS CLAUSE IN FILE DESCRIPTION.	Only one label records clause may be specified in a file description.

ERROR NUMBER	ERROR MESSAGE	DISCUSSION
193	MULTIPLE OCCURRENCES OF RECORD-CONTAINS CLAUSE IN FILE DESCRIPTION.	Only one record contains clause may be specified in a file description.
194	MULTIPLE OCCURRENCES OF RECMODE CLAUSE IN FILE DESCRIPTION.	Only one recording mode clause may be specified in a file description.
195	MULTIPLE OCCURRENCES OF REPORT CLAUSE IN FILE DESCRIPTION.	Only one report clause may be specified in a file description.
196	MULTIPLE OCCURRENCES OF VALUE-OF CLAUSE IN FILE DESCRIPTION.	Only one value-of clause may be specified in a file description.
197	MULTIPLE OCCURRENCES OF LINAGE CLAUSE IN FILE DESCRIPTION.	Only one linage clause may be specified in a file description.
198	MULTIPLE OCCURRENCES OF CODE-SET CLAUSE IN FILE DESCRIPTION.	Only one code set clause may be specified in a file description.
199	FILE NAME DOES NOT APPEAR IN A SELECT CLAUSE.	A name that appears in an FD or SD entry in the data division did not appear in a select clause in the environment division.
200	MULTIPLE OCCURRENCES OF FD-SD ENTRIES FOR FILE NAME.	The same name has been used in more than one FD or SD entry.
201	AREA A MUST BE BLANK IN A CONTINUATION RECORD.	
202	ILLEGAL COBOL CHARACTER IGNORED.	Check list of legal COBOL characters.
203	ILLEGAL COBOL ELEMENT.	Message is never used.
204	CONTINUATION RECORD EXPECTED. ! IGNORED.	Message is never used.
205	RESERVED WORD ! NOT LEGAL IN THIS DIVISION.	The specified word is a Reserved word used in another division.
206	PICTURE CHARACTER-STRING TOO LONG.	Message is never used.
207	ILLEGAL OCTAL DIGIT.	
208	LITERAL TOO LONG.	Nonnumeric literal must not be longer than 132 characters.
209	OCTAL LITERAL GREATER THAN (OCTAL) 37777777777.	
210	ILLEGAL DUPLICATION OF CLAUSES IN SPECIAL NAMES PARAGRAPH.	
211	ALPHABET NAME HAS ALREADY BEEN USED.	Alphabet name has already been used in program COLLATING SEQUENCE clause.
212	ILLEGAL MNEMONIC NAME.	Illegal function name is used in MNEMONIC NAME clause.

ERROR NUMBER	ERROR MESSAGE	DISCUSSION
213	MNEMONIC NAME HAS ALREADY BEEN USED.	
214	ON-OFF CONDITION DOES NOT REFER TO A SWITCH.	
215	COLLATING SEQUENCE NOT SPECIFIED IN OBJECT-COMPUTER PARAGRAPH.	Message is never used.
216	NONNUMERIC LITERAL IN "THRU" CLAUSE HAS MORE THAN ONE CHARACTER.	
217	NONNUMERIC LITERAL IN "ALSO" CLAUSE HAS MORE THAN ONE CHARACTER.	
218	CURRENCY SIGN HAS MORE THAN ONE CHARACTER.	
219	ILLEGAL SUBSTITUTE CURRENCY SIGN.	
220	ILLEGAL COMBINATION OF FILE ORGANIZATION AND ACCESS METHODS.	
221	MISSING FILE POSITION NUMBER(S).	Missing file position number(s) in MULTIPLE FILE clause.
222	DUPLICATE FILE NAME IN MULTIPLE FILE TAPE CLAUSE.	
223	DUPLICATE FILE POSITION IN MULTIPLE FILETAPE CLAUSE.	
224	FILE NAME NOT DEFINED IN SELECT CLAUSE.	
225	FILE POSITION HAS MORE THAN 4 DIGITS.	
226	FILE POSITION NUMBERS MUST START WITH 1.	
227	FILE NAME IN MORE THAN ONE SAME RECORD AREA.	
228	FILE NAME IN MORE THAN ONE SAME SORT/MERGE AREA.	
229	FILE NAME IN MORE THAN ONE SAME AREA.	
230	FILE NAME HAS ALREADY BEEN USED.	
231	"ALL" CONSTRUCT NOT ALLOWED IN ALPHABET CLAUSE.	
232	DUPLICATE CHARACTER IN ALPHABET DEFINITION.	
233	NUMERIC LITERAL IN ALPHABET DEFINITION HAS SIGN--SIGN DROPPED.	

ERROR NUMBER	ERROR MESSAGE	DISCUSSION
234	NUMERIC LITERAL IN ALPHABET DEFINITION IS GREATER THAN HIGH-VALUE.	
235	NUMERIC LITERAL IN ALPHABET CLAUSE LESS THAN LOW-VALUE.	
236	NUMERIC LITERAL IN ALPHABET NAME CLAUSE MAY NOT EQUAL 0.	
237	LITERAL-2 MUST BE GREATER THAN LITERAL-1 IN THRU CLAUSE.	Literal-2 must be greater than literal-1 in ALPHABET NAME definition.
238	MISSING COMMA IN SYSTEM FILE NAME.	See FILE-INFO.
239	ILLEGAL FORMAL FILE DESIGNATOR IN SYSTEM FILE NAME.	See FILE-INFO.
240	1ST CHARACTER OF FORMAL FILE DESIGNATOR MUST BE '$' OR ALPHABETIC.	
241	FORMAL FILE DESIGNATOR HAS MORE THAN 8 CHARACTERS.	
242	ILLEGAL CHARACTER IN FORMAL FILE DESIGNATOR.	See MPE restrictions for file name.
243	ILLEGAL DEVICE CLASS.	
244	ILLEGAL RECORDING MODE.	
245	ILLEGAL DEVICE NAME.	
246	DEVICE CODE MUST CONTAIN 3 DIGITS.	
247	FILE SIZE HAS MORE THAN 9 DIGITS.	
248	ILLEGAL FILE SIZE.	Message is never used.
249	FORMS MESSAGE HAS MORE THAN 49 CHARACTERS.	
250	FORMS MESSAGE MUST END WITH A PERIOD.	
251	ILLEGAL LOCKING PARAMETER.	"L" is locking parameter (see page 7-33).
252	SYSTEM FILE NAME CONTAINS TOO MANY FIELDS.	See FILE-INFO.
253	ILLEGAL COMBINATION OF SAME AREA AND SAME RECORD AREA FILES.	Inconsistent combination of these clauses.
254	ILLEGAL COMBINATION OF SAME AREA AND SAME SORT AREA FILES.	Inconsistent combination of these clauses.
255	ILLEGAL CHARACTER.	Message is never used.
256	ILLEGAL WORD.	Message is never used.
258	MULTIPLE REEL/UNIT CLAUSE NOT IMPLEMENTED.	

ERROR NUMBER	ERROR MESSAGE	DISCUSSION
259	DEFAULT FILE NAME IS TEMPORARY NAMELESS FILE.	
260	FILE NAME NOT SPECIFIED IN ENVIRONMENT DIVISION.	Message 260 is never used.
261	COMPILER ERROR: INVALID VALUE FOR USAGE. PLEASE CONTACT H-P.	In a SORT or MERGE key, the only valid USAGE for a data name used as a key is "DISPLAY", "COMP", "COMPSYNC", or "COMP3".
262	KEY WORD "END" IGNORED.	
263	KEY WORD "REEL" IGNORED.	The "CLOSE" will not be performed.
264	KEY WORD "UNIT" IGNORED.	The "CLOSE" will not be performed.
265	KEY WORD "BEFORE" IGNORED.	
266	NAME OF "SD" ENTRY FOUND, EXPECTED "FD" ENTRY.	I/O statements should reference an "FD" name.
267	NAME OF "FD" ENTRY FOUND,EXPECTED "SD" ENTRY.	SORT and MERGE statements should reference an "SD" name.
268	LINAGE MUST BE SPECIFIED TO USE "WRITE AT END OF PAGE".	A "WRITE AT END-OF-PAGE" was found for a file without a linage clause specified.
269	LINAGE MUST BE SPECIFIED IN ORDER TO USE "LINAGE-COUNTER".	"LINAGE-COUNTER" was referenced for a file which does not have a linage clause.
272	INTRINSIC RETURN VALUE MISMATCH.	
273	TOO MANY PARAMETERS SPECIFIED FOR INTRINSIC CALL.	
281	INVALID MACRONAME; EXPECTED ! AS FIRST CHARACTER.	
282	INVALID SUBSYSTEM COMMAND PARAMETER.	Invalid or missing parameter.
283	INVALID SUBSYSTEM COMMAND.	
284	TOO MANY PARAMETERS IN MACRO CALL.	Check macro definition.
285	INVALID FILENAME IN $INCLUDE COMMAND.	
286	INVALID CHARACTER IN $PREPROCESSOR COMMAND.	
287	PSEUDO-TEXT-1 CAN'T BE NULL OR BLANK.	Pseudo-text-1 can't be null or blank in COPY REPLACING statement.
288	INVALID IDENTIFIER USED AS QUALIFIER.	The qualifier in COPY REPLACING identifier.
289	INVALID TOKEN IN COPY...REPLACING CLAUSE.	

ERROR NUMBER	ERROR MESSAGE	DISCUSSION
290	ILLEGAL TERMINATION OF NONNUMERIC LITERAL !.	
301	REPORT WRITER MODULE IS NOT IMPLEMENTED.	A language feature which is part of the report writer module has been encountered.
302	VOL SPECIFIED TWICE.	The vol phrase may be specified only once in a VALUE OF clause.
303	LABELS SPECIFIED TWICE.	The labels phrase may be specified only once in a value of clause
304	SEQ SPECIFIED TWICE.	the seq phrase may be specified only once in a value of clause
305	EXDATE SPECIFIED TWICE.	the exdate phrase may be specified only once in a value of clause
306	VOL MUST BE A NON-NUMERIC LITERAL ≤ 6 CHARACTERS LONG.	When the vol phrase of the value of clause specifies a literal. the literal must be non-numeric and no more than six characters long.
307	LABELS MUST BE NON-NUMERIC LITERAL = "IBM" OR "ANS".	When the labels phrase of the value of clause specifies a literal the literal must be "IBM" or "ANS".
308	SEQ MUST BE AN UNSIGNED NUMERIC LITERAL.	When the seq phrase of the value of clause specifies a literal the literal must be numeric and unsigned.
309	EXDATE MUST BE NON-NUMERIC LITERAL OF THE FORM "MM/DD/YY".	When the exdate phrase of the value of clause specifies a literal the literal must be non-numeric and of the form "MM/DD/YY".
310	LINAGE MUST BE AT LEAST 1.	When the lines phrase of the linage clause specifies an integer it must be ≤1.
311	FOOTING MAY NOT BE LESS THAN 1.	When the the footing phrase of the linage clause specifies an integer it is necessary that 1 ≤ footing integer ≤ linage integer(if specified).
312	FOOTING MAY NOT BE GREATER THAN LINAGE.	When the footing phrase of the linage clause specifies an integer it is necessary that 1 ≤ footing integer ≤ linage integer(if specified)
313	FOOTING SPECIFIED TWICE.	The footing phrase may be specified only once in a linage clause
314	TOP MAY NOT BE LESS THAN 0.	When the top phrase of the linage clause specifies an integer it must be ≥0
315	TOP SPECIFIED TWICE.	the top phrase may be specified only once in a linage clause

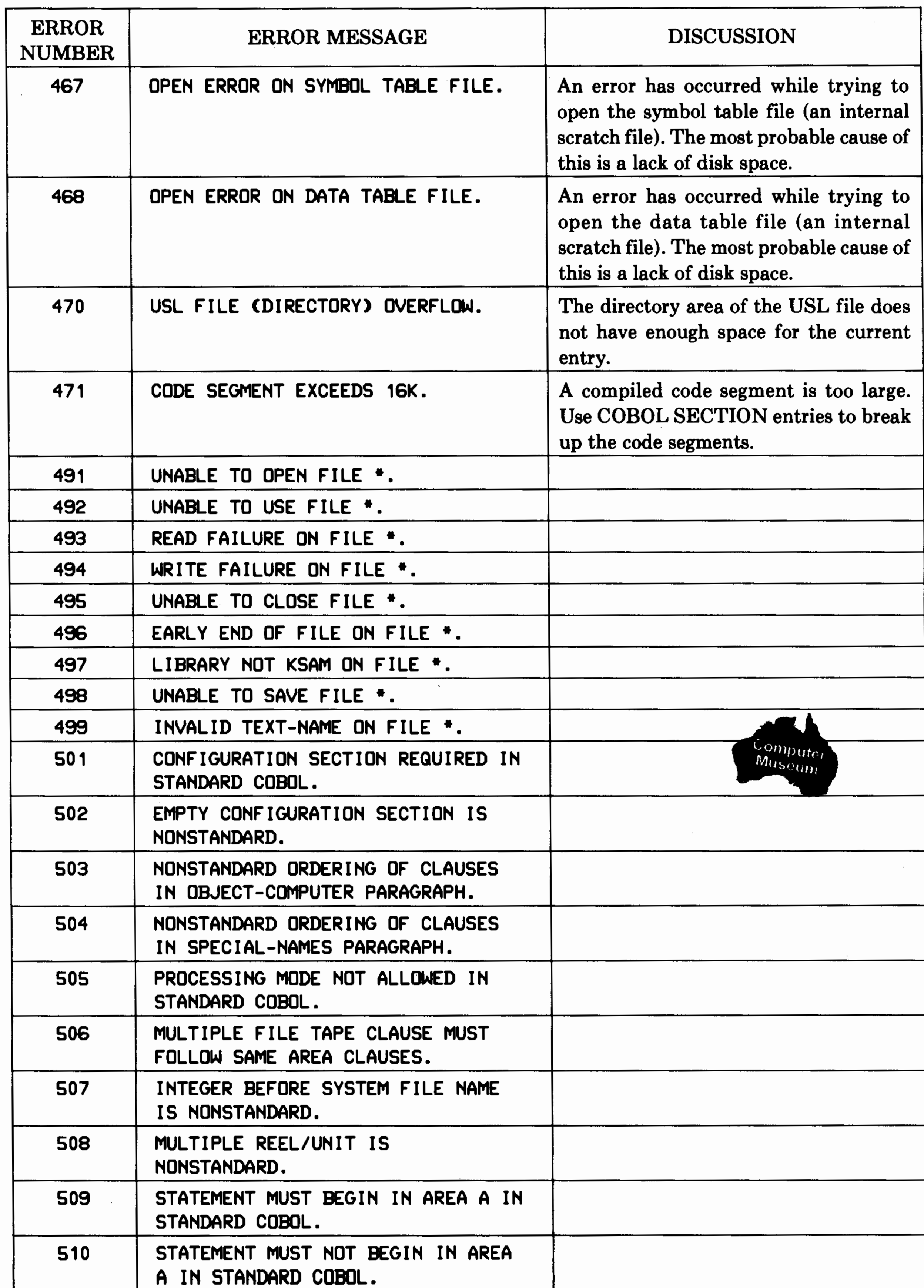
ERROR NUMBER	ERROR MESSAGE	DISCUSSION
316	BOTTOM MAY NOT BE LESS THAN 0.	When the bottom phrase of the linage clause specifies an integer it must be ≥ 0
317	BOTTOM SPECIFIED TWICE.	The bottom phrase may be specified only once in a linage clause
318	FILE RECORD SMALLER THAN MINIMUM SIZE IN RECORD-CONTAINS CLAUSE.	When the record contains clause is specified in an FD or SD every record subordinate to that file description must have a size that is within the specified bounds
319	FILE RECORD LARGER THAN MAXIMUM SIZE IN RECORD-CONTAINS CLAUSE.	When the record contains clause is specified in an FD or SD every record subordinate to that file description must have a size that is within the specified bounds
320	FILE RECORD SIZE IS ZERO.	When the largest record for a file is 0 characters and there is a block contains clause in the file description which contains the characters phrase, it is impossible to compute the block factor
350	EMPTY CORRESPONDING: NO CORRESPONDING ITEMS FOUND.	A CORRESPONDING statement has no CORRESPONDING pairs.
351	REFERENCE TO ! IS NOT UNIQUE.	Reference needs additional qualification(s) to make it unique.
352	ILLEGAL CORRESPONDING OPERAND OR EMPTY CORRESPONDING.	
353	SELF QUALIFICATION.	A reference has duplicate qualification.
354	ERROR IN COMPILER: INVALID SYMBOL/DATA TABLE REFERENCE.	An invalid symbol or data table reference has occurred internal to compiler. Please submit service request (SR).
355	ILLEGAL PROGRAM CONSTRUCT.	The program has an improper construct, e.g. it has a section appearing in a PROCEDURE DIVISION starting with a paragraph.
356	UNDEFINED DATA NAME ! .	The referenced data name is undefined.
357	DUPLICATE PARAGRAPH OR SECTION NAME.	A duplicate paragraph or section name appears.
358	ILLEGAL PARAGRAPH OR SECTION NAME.	A paragraph or section name is illegal, e.g. a signed numeric literal.
359	UNDEFINED OR IMPROPER PROCEDURE NAME.	The referenced procedure name is undefined or improper, e.g. data name or signed numeric literal. No code is generated for statement.

ERROR NUMBER	ERROR MESSAGE	DISCUSSION
360	ILLEGAL GO TO STATEMENT.	The GO TO statement is incorrect, e.g. GO TO appears, but it is not the first statement in a paragraph.
361	ILLEGAL ALTER STATEMENT.	The ALTER statement references a non-existing paragraph name or it does not reference an alterable GO TO.
362	ILLEGAL PERFORM-TIMES COUNT.	The count must be an unsigned, positive integer.
363	RECURSIVE PERFORM.	The PERFORM statement is recursive through itself or through its father.
364	TOO MANY PARAMETERS.	A CALL statement or USING option contains too many parameters. The maximum number of parameters is 60.
365	ILLEGAL RELATIONAL COMPARE.	e.g. comparing condition-code against non-zero literal.
366	COMPOSITE OF OPERANDS TOO BIG.	The composite of operands for the statement is too large.
367	ILLEGAL NUMERIC OPERAND.	The operand is not an allowed numeric operand. The reference is replaced with a reference to TALLY.
368	REFERENCE TO ! BY ! IS NOT UNIQUE.	The reference to the data item named requires further qualification to be unique. Reference is used as a key or DEPENDING ON variable.
369	ILLEGAL FUNCTION NAME.	Mnemonic-name in ACCEPT statement may only be SYSIN or CONSOLE. The function name is changed to SYSIN.
370	ILLEGAL STATEMENT FORMAT.	e.g. PERFORM-VARYING statement with more than 3 levels.
371	INTRINSIC NOT FOUND IN INTRINSIC FILE.	
372	INVALID RECORD NAME.	The record name in this WRITE/REWRITE/RELEASE statement is not an "01" level item. No code is generated for this statement.
380	INVALID USE OF A SEPARATOR CHARACTER.	
381	MISSING PERIOD IN COPY STATEMENT.	
382	ERROR OR MISSING PROGRAM-ID PARAGRAPH.	

ERROR NUMBER	ERROR MESSAGE	DISCUSSION
383	INCORRECT TABLE REFERENCE.	e.g. illegal subscript is used, or table referenced as simple variable, or index-name does not belong to table-name.
384	COMMUNICATIONS MODULE IS NOT IMPLEMENTED.	A language feature which is part of the communications module has been encountered.
385	DEBUG MODULE IS NOT IMPLEMENTED.	
386	ENTER STATEMENT IS NOT IMPLEMENTED.	
387	ILLEGAL MOVE.	A move statement (or an implied move statement, eg. a write from) has an illegal combination of "from" and "to" operands.
388	ILLEGAL COMPARE.	The subject and object of a relational operator are incompatible
389	PARAMETER ! IS NOT ON A WORD BOUNDARY.	A data item is being passed by reference as a word address but does not start on a word boundary.
390	ERROR IN COMPILER: UNIMPLEMENTED CASE ! IN PROCEDURE !.	The compiler was not prepared for a given situation. Please submit a service request (SR).
391	ENTRY STATEMENT NOT ALLOWED IN DECLARATIVES SECTION.	An ENTRY statement has been found in the Declaratives Section of the program. The compiler ignores the ENTRY statement.
401	ASSIGN CLAUSE REQUIRED WITH SELECT STATEMENT.	
402	MISSING RELATIVE KEY CLAUSE.	
403	MISSING RECORD KEY CLAUSE.	
404	MISSING ACTUAL KEY CLAUSE.	
405	TOO MANY KEY CLAUSES.	More than one RELATIVE/RECORD/ ACTUAL KEY clause appears.
406	! ! FOR !.	This error message is for illegal or missing forward references. The insertions are for the forward reference type, the name of the forward reference, and the name of the file or table containing the forward reference. The forward reference types are relative keys, record keys, and actual keys.

ERROR NUMBER	ERROR MESSAGE	DISCUSSION
407	FILE NAME IS UNDEFINED OR IS NOT UNIQUE.	The file name in this statement is not found in a "SELECT", or "FD" statement, or there exists a data name by the same name.
408	ILLEGAL OPERAND IN SEARCH STATEMENT.	
409	ILLEGAL OPERAND IN SET STATEMENT.	
410	SYNTAX ERROR--EXPECTING ONE OF THE FOLLOWING: !!	An item appears in some context that the compiler cannot recognize. The items which are allowed at this point are listed. The compiler attempts to recover from this error.
411	COMPILER ERROR: ILLEGAL INTERNAL (IDS) FORMAT.	Please submit a service request (SR).
412	PARAMETER MUST BE 01 OR 77 LEVEL ITEM IN LINKAGE SECTION.	An illegal parameter was used in a PROCEDURE DIVISION USING statement.
413	FILE NAME DOES NOT APPEAR IN A SELECT CLAUSE.	A name that appears in an FD or SD entry in the data division did not appear in a select clause in the environment division.
415	INVALID LIBRARY OR TEXT NAME.	
416	COPY LIB OR $INCLUDE FILES NESTED TOO DEEP.	
417	MACRO DEFINITION MAXIMUM LENGTH EXCEEDED.	
418	PSEUDO-TEXT-BUFFER OVERFLOW.	
421	OPERAND HAS ILLEGAL FORMAT FOR STATEMENT.	This type of operand may not be used in the statement.
422	SIZE OF DATA SEGMENT GREATER THAN 65K BYTES.	The upper limit on the size of the data area has been reached.
450	USL FILE OVERFLOW.	The USL file may overflow under the following conditions: a $CONTROL USLINIT command may be missing, the default size of 1023 records may be too small, or there may not be enough records left in the usl file.
451	PARSE STACK OVERFLOW.	The parse stack in compiler overflowed. Please contact H-P.
452	EARLY END OF FILE ON COBOL SOURCE.	
453	BAD INTRINSIC FILE.	The system intrinsic file (SPLINTR.PUB.SYS) is not in the proper format.

ERROR NUMBER	ERROR MESSAGE	DISCUSSION
454	READ ERROR ON IDS FILE.	An error has occurred while trying to do a read on the IDS file (an internal scratch file). The most probable cause is a serious compiler problem
455	WRITE ERROR ON IDS FILE.	an error has occurred while trying to do a write to the IDS file (an internal scratch file) this can be caused by a serious compiler problem, an excessively large source file, or a lack of disk space.
456	OPEN ERROR ON IDS FILE.	An error has occurred while trying to open the IDS file (an internal scratch file). The most probable cause of this is a lack of disk space.
457	COMPILER ERROR: OUT OF IDS FILE BUFFERS.	This can only be caused by a compiler problem.
458	COMPILER ERROR: INVALID INTERNAL LABEL	The compiler has generated code referencing an invalid internal label number. Contact H-P.
459	TOO MANY VALUE CLAUSES.	USL entry overflows max. size. Reduce number of VALUE clauses, e.g., by combining at group level.
460	MISSING IDENTIFICATION DIVISION, COMPILATION TERMINATED.	
461	DYNAMIC ARRAY ERROR, OUT OF STACK SPACE.	Please contact H-P.
462	AVAILABLE MEMORY INSUFFICIENT FOR COMPILATION.	This can only be caused by a compiler problem. Please contact H-P.
463	READ ERROR ON SYMBOL TABLE FILE.	An error has occurred while trying to read from the symbol table file (an internal scratch file). The most probable cause of this is a serious compiler error
464	READ ERROR ON DATA TABLE FILE.	An error has occurred while trying to read from the data table file (an internal scratch file). The most probable cause of this is a serious compiler error
465	WRITE ERROR ON SYMBOL TABLE FILE.	An error has occurred while trying to write to the symbol table file (an internal scratch file). This can be caused by a compiler error or by a lack of disk space.
466	WRITE ERROR ON DATA TABLE FILE.	An error has occurred while trying to write to the data table file (an internal scratch file). This can be caused by a compiler error or by a lack of disk space.



ERROR NUMBER	ERROR MESSAGE	DISCUSSION
467	OPEN ERROR ON SYMBOL TABLE FILE.	An error has occurred while trying to open the symbol table file (an internal scratch file). The most probable cause of this is a lack of disk space.
468	OPEN ERROR ON DATA TABLE FILE.	An error has occurred while trying to open the data table file (an internal scratch file). The most probable cause of this is a lack of disk space.
470	USL FILE (DIRECTORY) OVERFLOW.	The directory area of the USL file does not have enough space for the current entry.
471	CODE SEGMENT EXCEEDS 16K.	A compiled code segment is too large. Use COBOL SECTION entries to break up the code segments.
491	UNABLE TO OPEN FILE *.	
492	UNABLE TO USE FILE *.	
493	READ FAILURE ON FILE *.	
494	WRITE FAILURE ON FILE *.	
495	UNABLE TO CLOSE FILE *.	
496	EARLY END OF FILE ON FILE *.	
497	LIBRARY NOT KSAM ON FILE *.	
498	UNABLE TO SAVE FILE *.	
499	INVALID TEXT-NAME ON FILE *.	
501	CONFIGURATION SECTION REQUIRED IN STANDARD COBOL.	
502	EMPTY CONFIGURATION SECTION IS NONSTANDARD.	
503	NONSTANDARD ORDERING OF CLAUSES IN OBJECT-COMPUTER PARAGRAPH.	
504	NONSTANDARD ORDERING OF CLAUSES IN SPECIAL-NAMES PARAGRAPH.	
505	PROCESSING MODE NOT ALLOWED IN STANDARD COBOL.	
506	MULTIPLE FILE TAPE CLAUSE MUST FOLLOW SAME AREA CLAUSES.	
507	INTEGER BEFORE SYSTEM FILE NAME IS NONSTANDARD.	
508	MULTIPLE REEL/UNIT IS NONSTANDARD.	
509	STATEMENT MUST BEGIN IN AREA A IN STANDARD COBOL.	
510	STATEMENT MUST NOT BEGIN IN AREA A IN STANDARD COBOL.	

Computer Museum

ERROR NUMBER	ERROR MESSAGE	DISCUSSION
511	RANDOM I/O IS NO LONGER PART OF STANDARD COBOL.	
512	FILE LIMITS CLAUSE IS NOT STANDARD.	
513	FILE-CONTROL PARAGRAPH IS REQUIRED IN STANDARD COBOL.	
514	A SPACE IS REQUIRED AFTER COMMA (,) OR SEMICOLON (;).	This is illegal at all levels.
515	! IS IN THE LOW INTERMEDIATE FIPS LEVEL.	
516	! IS IN THE HIGH INTERMEDIATE FIPS LEVEL.	
517	! IS IN THE HIGH FIPS LEVEL.	
518	! IS A HEWLETT-PACKARD COBOL II EXTENSION.	
520	ITEM REDEFINES AN ITEM CONTAINING A REDEFINES CLAUSE.	An item may not redefine an item which contains a redefines clause.
521	REDEFINE OF FILE RECORD IGNORED.	The redefines clause is illegal at the 01-level in the file section
522	! ! FOR !.	This error message is for non-standard forward references. The insertions are for the forward reference type, the name of the forward reference, and the name of the file or table containing the forward reference. The forward reference types are signed relative keys, non-alphanumeric record keys, and non-alphanumeric alternate record keys.
530	COMMENTS AND BLANK LINES ARE NONSTANDARD BEFORE THE IDENTIFICATION & DIVISION HEADING.	
980	ATTEMPTING TO RECOVER FROM SYNTAX ERROR.	The indicated item is where the compiler is attempting to recover from the previous syntax error. The items between the two messages were ignored by the compiler.
990	POSSIBLE COMPILER ERROR: INVALID FORWARD REFERENCE TYPE.	Something totally inexplicable has happened. Please contact H-P.
991	MISSING ERROR MESSAGE !. PLEASE CONTACT H-P.	Message is never used.
992	POSSIBLE COMPILER ERROR: ERROR MESSAGE OVERFLOWS BUFFER. PLEASE CONTACT H-P.	

ERROR NUMBER	ERROR MESSAGE	DISCUSSION
993	POSSIBLE COMPILER ERROR: FAILURE IN INTERNAL CALL TO INTRINSIC GEN MESSAGE. PLEASE CONTACT H-P.	
994	TOO MANY ERRORS FOR ERROR FILE. SOME ERRORS MAY NOT BE LISTED.	
995	POSSIBLE COMPILER ERROR: INVALID SEVERITY CODE. PLEASE CONTACT H-P.	
996	POSSIBLE COMPILER ERROR: INVALID COLUMN PASSED TO COMPILER ERROR PROCEDURE. PLEASE CONTACT H-P.	
997	MISSING ERROR MESSAGE !. PLEASE CONTACT H-P..	
998	TOO MANY ERRORS.	
999	UNABLE TO CONTINUE. COMPILATION TERMINATED.	

Run Time Error Messages

Unless otherwise indicated, the following action will be taken for a run-time I/O error (610-751).

1. The error message will be printed.
2. If the error is file related (most are) then the MPE error number is printed and a PRINTFILEINFO is executed.
3. If a USE procedure, an AT END clause, an INVALID KEY clause, or a FILE STATUS key is specified (i.e. the user has some programmatic way to detect the error), then execution is allowed to continue. Otherwise a QUIT is issued.

ERROR NUMBER	ERROR MESSAGE	DISCUSSION
520	TRYING TO CLOSE AN UNOPENED FILE.	Correct error in program and recompile.
540	TRYING OPEN, FILE ALREADY OPEN.	Remove the redundant OPEN statement(s) or insert necessary CLOSE statement(s) and recompile program.
550	TRY TO READ CLOSED FILE, OR FILE NOT OPEN.	
551	READ SERVICE NOT GRANTED.	
610	TRYING WRITE FILE NOT OPEN YET	A WRITE statement was executed for a file which was not open.
611	WRITE SERVICE NOT GRANTED	A WRITE statement was unsuccessful. Refer to the MPE file system error.
630	ERROR DURING WRITE OF USER LABEL	An error occured while writing a label. Refer to the MPE file system error.
631	USER LABEL SPACE UNALLOCATED OR ATTEMPT TO WRITE BEYOND LABEL LIMIT	An error occured while writing a label. Refer to the MPE file system error.
632	POSSIBLE COMPILER ERROR: BAD MPE FILE NUMBER	An invalid MPE file number found. Internal file table may be invalid.
633	FILE NOT CLOSED; POSSIBLE DUPLICATE NAME	Error while closing file. Refer to the MPE file system error.
634	ATTEMPT TO CLOSE FILE THAT IS NOT OPEN	A CLOSE statement was executed for a file which was already closed.
635	ATTEMPT TO DELETE FILE THAT IS NOT OPEN	A DELETE statement was executed for a file which was not open.
636	RECORD NOT FOUND	An I/O statement was executed which addressed a nonexistant record. Refer to the MPE file system error.
637	DUPLICATE KEY	A WRITE statement was executed which incorrectly addressed an existing record. Refer to the MPE file system error.

ERROR NUMBER	ERROR MESSAGE	DISCUSSION
638	FILE NOT OPENED WITH INPUT/OUTPUT MODE	A REWRITE statement was executed for a file which was not opened in INPUT/OUTPUT mode.
639	ATTEMPT TO ISSUE EXCLUSIVE FOR FILE THAT IS NOT OPEN	An EXCLUSIVE statement was executed for a file which was not open.
640	MULTIPLE RIN CAPABILITY NEEDED	To use the EXCLUSIVE statement you must have MR capability.
641	FILE IN USE BY ANOTHER PROCESS	An EXCLUSIVE statement was executed for a file for which some other process has already executed an EXCLUSIVE statement.
642	FILE IS LOCKED BY A CLOSE	An OPEN statement was executed for a file which was locked by a previous close.
643	FILE IS ALREADY OPEN	An OPEN statement was executed for a file which was already open.
644	FILE IS IN USE OR SPACE REQUESTED IS NOT AVAILABLE	An OPEN statement was executed for a file which was in use by some other process or there was insufficient disc space for the file being created. Refer to the MPE file system error.
645	I/O ERROR ON OPEN	An I/O error occured while opening the file. Refer to the MPE file system error.
646	TOP TOO LARGE IN LINAGE CLAUSE	The value of TOP was larger than 1000—Zero assumed.
647	FILE NOT FOUND	An OPEN statement was executed for a file which could not be found.
648	FILE DIFFERS FROM ORGANIZATION CLAUSE.	Compare Environment Division description of file and actual organization of file.
649	RECORDS MUST BE IN ASCENDING ORDER BY KEY	A WRITE statement was executed for an INDEXED file which contained a key which was less than the previous key written.
650	END OF FILE ENCOUNTERED UPON READ	A READ statement was executed and no record was found.
651	ATTEMPT TO REWRITE FILE THAT IS NOT OPEN	A REWRITE statement was executed for a file which was not open.
652	KEY NOT FOUND	A START or SEEK statement was executed which specified a record which could not be found.

ERROR NUMBER	ERROR MESSAGE	DISCUSSION
653	FILE ALREADY OPEN WITH SAME AREA OR IS A MULTIPLE FILE	An OPEN statement was executed for a file which was specified in an area that was in use or for a file which was listed in a MULTIPLE FILE clause which specified a file which was already open.
654	ATTEMPT TO SEEK ON FILE THAT IS NOT OPEN	A SEEK statement was executed for a file which was not open.
655	SEQ ERROR ON LABELED TAPE	An OPEN statement resulted in a SEQ error for a labeled tape. Refer to the MPE file system error.
656	END OF FILE ENCOUNTERED UPON SEEK	Attempt to seek past the end of file. Refer to the MPE file system error.
657	ATTEMPT TO ISSUE START FOR FILE THAT IS NOT OPEN	A START statement was executed for a file which was not open.
658	I/O ERROR ON START	An I/O error occured while executing a START statement. Refer to the MPE file system error.
659	END OF FILE ENCOUNTERED UPON START	An end of file occured while executing a START statement. Refer to the MPE file system error.
660	ATTEMPT TO ISSUE UN-EXCLUSIVE FOR FILE THAT IS NOT OPEN	An UN-EXCLUSIVE statement was executed for a file which was not open.
661	ATTEMPT TO ISSUE UN-EXCLUSIVE FOR FILE FOR WHICH NO EXCLUSIVE WAS ISSUED	An UN-EXCLUSIVE statement was executed for a file for which no previous EXCLUSIVE statement was executed.
662	ATTEMPT TO WRITE TO FILE THAT IS NOT OPEN	A WRITE statement was executed for a file which was not open.
663	NO ROOM LEFT IN FILE	A WRITE statement was executed for a file which was full. Refer to the MPE file system error.
664	I/O ERROR	An I/O error occured. Refer to the MPE file system error.
665	END OF FILE ENCOUNTERED UPON WRITE	An attempt was made to write beyond the end of file. Refer to the MPE file system error.
666	ERROR WHILE READING LABEL	An error occured while reading the label. Refer to the MPE file system error.
667	READ/WRITE BEYOND NUMBER OF LABELS	Refer to the MPE file system error.

ERROR NUMBER	ERROR MESSAGE	DISCUSSION
668	LINAGE TOP BOTTOM FOOTING GREATER THAN 32767 NOT ALLOWED--RESET TO 32767	Check the LINAGE, TOP, BOTTOM, and FOOTING clauses.
671	RECORDS LARGER THAN LARGEST FD DESCRIPTION--WILL BE TRUNCATED	The record length found when the file was opened was larger than the amount of space specified in the largest FD description for this file. Any records which are too long will be truncated. Refer to the MPE file system error.
672	CCTL MUST BE SET TO USE LINAGE OPTIONS	Use of LINAGE requires CCTL to be set.
700	NONSTANDARD ERROR IN PROCEDURE DIVISION.	
701	NON-STANDARD LEVEL NUMBER.	
706	UNDERFLOW IN EXPONENTIATE	
707	OVERFLOW IN EXPONENTIATE	
708	UNDEFINED RESULT FROM EXPONENTIATE	
709	ILLEGAL DIGIT IN NUMERIC DATA ITEM TREATED AS 0	There was a bad digit in an item input with ACCEPT FREE. This digit was changed to zero.
710	ILLEGAL DECIMAL DIGIT	
711	ILLEGAL ASCII DIGIT	
750	DEPENDING-ON IDENTIFIER OUT OF BOUNDS	
751	SUBSCRIPT/INDEX OUT OF BOUNDS	

EXTENSIONS TO ANS COBOL '74

APPENDIX E

The following table lists all extensions to ANSI COBOL 74 which are incorporated in COBOL II/3000. Also included in the table is a page reference. This reference indicates where to find a description of the particular extension within the reference manual.

EXTENSION	PAGE
CALL statement:	
intrinsic calls (INTRINSIC phrase)	12-7
non-COBOL programs (USING phrase)	12-9
typed procedures (GIVING phrase)	12-10
OPTION VARIABLE procedures	12-10
Compiler options	A-26
COMPUTATIONAL-3 data-items	9-60
DYNAMIC and RANDOM (equivalent keywords)	7-34
Dynamic subprograms	12-4
File insertion and merging	A-17
Locking and Unlocking	
Locking files (COBOLLOCK procedure)	I-2
Unlocking files (COBOLUNLOCK procedure)	I-1
Macro definition and use	A-5
Octal literals	3-15
Random-access files	7-23
READ [NEXT] (for random-access files)	11-79
Special Names:	
TOP	7-10
NOSPACE CONTROL	7-10
SYSIN	7-10
SYSOUT	7-10
CONSOLE	7-10
SW0 through SW9	7-10
CONDITION-CODE	7-10
C01 through C12	7-10

Special registers:

TALLY	3-5
CURRENT-DATE	3-5
TIME-OF-DAY	3-5
WHEN-COMPILED	3-5

Statements:

ACCEPT FREE statement	11-1
ENTRY statement	12-13
EXAMINE statement	11-31
GOBACK statement	12-17
PERFORM statement (with common exits)	11-63
UN-EXCLUSIVE statement	11-114

Use of apostrophe (') as quotation mark 3-19

COBOL II CODING FORM

HEWLETT-PACKARD **COBOL** CODING FORM

PROGRAMMER	DATE	PROGRAM	PAGE OF

SEQUENCE (PAGE) 1–3	SEQUENCE (SER.) 4–6	CONT. 7	A 8	B 12	COBOL STATEMENT 16 20 24 28 32 36 40 44 48 52 56 60 64 68 72	IDENTIFICATION 73–80

0 ZERO   O ALPHA O   1 ONE   I ALPHA I   FOR 029 KEYPUNCHERS: ASCII [029 ¢   ASCII] 029 !
2 TWO   Z ALPHA Z   ASCII \ 029 0-8-2   ASCII ^ 029 ¬
ASCII ! 029 |

5951-4465

ASCII AND EBCDIC CHARACTER SETS

APPENDIX G

HOW TO USE THIS TABLE

- The table is sorted by character code, each code being represented by its decimal, octal, and hexadecimal equivalent.
- Each row of the table gives the ASCII and EBCDIC meaning of the character code, the ASCII↔EBCDIC conversion code, and the Hollerith representation (punched card code) for the ASCII character.

The following examples describe several ways of using the table:

Example 1: Suppose you want to determine the ASCII code for the $ character. Scan down the ASCII graphic column until you locate $, then look left on that row to find the character code – 36 (dec), 044 (oct), and 24 (hex). This is the code used by an ASCII device (terminal, printer, computer, etc.) to represent the $ character. Its Hollerith punched card code is 11-3-8.

Example 2: The character code 5B (hex) is the EBCDIC code for what character? Also, when 5B is converted to ASCII (for example, by FCOPY with the EBCDICIN option), what is the octal character code? First, locate 5B in the hex character code column and move right on that row to the EBCDIC graphic which is $. The next column to the right gives the conversion to ASCII, 044. As a check, find 044 (oct) in the character code column, look right to the ASCII graphic column and note that $ converted to EBCDIC is 133 (oct) which equals 5B (hex).

CHAR CODE			ASCII			EBCDIC	
Dec	Oct	Hex	Cntl/ Gph	to EBCDIC (Oct)	Hollerith	Cntl/ Gph	to ASCII (Oct)
0	000	00	NUL	000	12-0-1-8-9	NUL	000
1	001	01	SOH	001	12-1-9	SOH	001
2	002	02	STX	002	12-2-9	STX	002
3	003	03	ETX	003	12-3-9	ETX	003
4	004	04	EOT	067	7-9	PF	234
5	005	05	ENQ	055	0-5-8-9	HT	011
6	006	06	ACK	056	0-6-8-9	LC	206
7	007	07	BEL	057	0-7-8-9	DEL	177
8	010	08	BS	026	11-6-9		227
9	011	09	HT	005	12-5-9		215
10	012	0A	LF	045	0-5-9	SMM	216
11	013	0B	VT	013	12-3-8-9	VT	013
12	014	0C	FF	014	12-4-8-9	FF	014
13	015	0D	CR	015	12-5-8-9	CR	015
14	016	0E	SO	016	12-6-8-9	SO	016
15	017	0F	SI	017	12-7-8-9	SI	017
16	020	10	DLE	020	12-11-1-8-9	DLE	020
17	021	11	DC1	021	11-1-9	DC1	021
18	022	12	DC2	022	11-2-9	DC2	022
19	023	13	DC3	023	11-3-9	TM	023
20	024	14	DC4	074	4-8-9	RES	235
21	025	15	NAK	075	5-8-9	NL	205
22	026	16	SYN	062	2-9	BS	010
23	027	17	ET8	046	0-6-9	IL	207
24	030	18	CAN	030	11-8-9	CAN	030
25	031	19	EM	031	11-1-8-9	EM	031
26	032	1A	SUB	077	7-8-9	CC	222
27	033	1B	ESC	047	0-7-9	CU1	217
28	034	1C	FS	034	11-4-8-9	IFS	034
29	035	1D	GS	035	11-5-8-9	IGS	035
30	036	1E	RS	036	11-6-8-9	IRS	036
31	037	1F	US	037	11-7-8-9	IUS	037
32	040	20	SP	100	Blank	DS	200
33	041	21	!	117	12-7-8	SOS	201
34	042	22	"	177	7-8	FS	202
35	043	23	=	173	3-8		203
36	044	24	S	133	11-3-8	8YP	204
37	045	25	%	154	0-4-8	LF	012
38	046	26	&	120	12	ETB	027
39	047	27	'	175	5-8•	ESC	033
40	050	2B	(	115	12-5-8		210
41	051	29	)	135	11-5-8		211
42	052	2A	*	134	11-4-8	SM	212
43	053	28	+	116	12-6-8	CU2	213
44	054	2C	,	153	0-3-8		214
45	055	2D		140	11	ENQ	005
46	056	2E	.	113	12-3-8	ACK	006
47	057	2F	/	141	0-1	8EL	007

CHAR CODE			ASCII			EBCDIC	
Dec	Oct	Hex	Cntl/ Gph	to EBCDIC (Oct)	Hollerith	Cntl/ Gph	to ASCII (Oct)
48	060	30	0	360	0		220
49	061	31	1	361	1		221
50	062	32	2	362	2	SYN	026
51	063	33	3	363	3		223
52	064	34	4	364	4	PN	224
53	065	35	5	365	5	RS	225
54	066	36	6	366	6	UC	226
55	067	37	7	367	7	EOT	004
56	070	38	8	370	8		230
57	071	39	9	371	9		231
58	072	3A	:	172	2-8		232
59	073	38	;	136	11-6-8	CU3	233
60	074	3C	<	114	12-4-8	DC4	024
61	075	3D		176	6-8	NAK	025
62	076	3E	>	156	0-6-8		236
63	077	3F	?	157	0-7-8	SUB	032
64	100	40	@	174	4-8	SP	040
65	101	41	A	301	12-1		240
66	102	42	B	302	12-2		241
67	103	43	C	303	12-3		242
68	104	44	D	304	12-4		243
69	105	45	E	305	12-5		244
70	106	46	F	306	12-6		245
71	107	47	G	307	12-7		246
72	110	48	H	310	12-8		247
73	111	49	I	311	12-9		250
74	112	4A	J	321	11-1	¢	133
75	113	4B	K	322	11-2	.	056
76	114	4C	L	323	11-3	<	074
77	115	4D	M	324	11-4	(	050
78	116	4E	N	325	11-5	+	053
79	117	4F	O	326	11-6	|	041
80	120	50	P	327	11-7	&	046
81	121	51	Q	330	11-8		251
82	122	52	R	331	11-9		252
83	123	53	S	342	0-2		253
84	124	54	T	343	0-3		254
85	125	55	U	344	0-4		255
86	126	56	V	345	0-5		256
87	127	57	W	346	0-6		257
88	130	58	X	347	0-7		260
89	131	59	Y	350	0-8		261
90	132	5A	Z	351	0-9	!	135
91	133	5B	[	112	12-2-8	$	044
92	134	5C	\	340	0-2-8	*	052
93	135	5D	]	132	11-2-8	)	051
94	136	5E	^	137	11-7-8	;	073
95	137	5F	_	155	0-5-8	¬	136

CHAR CODE			ASCII			EBCDIC	
Dec	Oct	Hex	Cntl/ Gph	to EBCDIC (Oct)	Hollerith	Cntl/ Gph	to ASCII (Oct)
96	140	60	`	171	1-8	-	055
97	141	61	a	201	12-0-1	/	057
98	142	62	b	202	12-0-2		262
99	143	63	c	203	12-0-3		263
100	144	64	d	204	12-0-4		264
101	145	65	e	205	12-0-5		265
102	146	66	f	206	12-0-6		266
103	147	67	g	207	12-0-7		267
104	150	68	h	210	12-0-8		270
105	151	69	i	211	12-0-9		271
106	152	6A	j	221	12-11-1	¦	174
107	153	68	k	222	12-11-2	,	054
108	154	6C	l	223	12-11-3	%	045
109	155	6D	m	224	12-11-4	_	137
110	156	6E	n	225	12-11-5	>	076
111	157	6F	o	226	12-11-6	?	077
112	160	70	p	227	12-11-7		272
113	161	71	q	230	12-11-8		273
114	162	72	r	231	12-11-9		274
115	163	73	s	242	11-0-2		275
116	164	74	t	243	11-0-3		276
117	165	75	u	244	11-0-4		277
11B	166	76	v	245	11-0-5		300
119	167	77	w	246	11-0-6		301
120	170	78	x	247	11-0-7		302
121	171	79	y	250	11-0-8	`	140
122	172	7A	z	251	11-0-9	:	072
123	173	7B	{	300	12-0	#	043
124	174	7C	¦	152	12-11	@	100
125	175	7D	}	320	11-0	'	047
126	176	7E	~	241	11-0-1	=	075
127	177	7F	DEL	007	12-7-9	"	042
128	200	80		040	11-0-1-8-9		303
129	201	81		041	0-1-9	a	141
130	202	B2		042	0-2-9	b	142
131	203	83		043	0-3-9	c	143
132	204	84		044	0-4-9	d	144
133	205	B5		025	11-5-9	e	145
134	206	86		006	12-6-9	f	146
135	207	87		027	11-7-9	g	147
136	210	B8		050	0-8-9	h	150
137	211	89		051	0-1-8-9	i	151
138	212	8A		052	0-2-8-9		304
139	213	88		053	0-3-8-9		305
140	214	8C		054	0-4-8-9		306
141	215	8D		011	12-1-8-9		307
142	216	BE		012	12-2-8-9		310
143	217	8F		033	11-3-8-9		311
144	220	90		060	12-11-0-1-8-9		312
145	221	91		061	1-9	j	152
146	222	92		032	11-2-8-9	k	153
147	223	93		063	3-9	l	154
148	224	94		064	4-9	m	155
149	225	95		065	5-9	n	156
150	226	96		066	6-9	o	157
151	227	97		010	12-8-9	p	160
152	230	98		070	8-9	q	161
153	231	99		071	1-8-9	r	162
154	232	9A		072	2-8-9		313
155	233	9B		073	3-8-9		314
156	234	9C		004	12-4-9		315
157	235	9D		024	11-4-9		316
158	236	9E		076	6-8-9		317
159	237	9F		341	11-0-1-9		320
160	240	A0		101	12-0-1-9		321
161	241	A1		102	12-0-2-9	~	176
162	242	A2		103	12-0-3-9	s	163
163	243	A3		104	12-0-4-9	t	164
164	244	A4		105	12-0-5-9	u	165
165	245	A5		106	12-0-6-9	v	166
166	246	A6		107	12-0-7-9	w	167
167	247	A7		110	12-0-8-9	x	170
168	250	A8		111	12-1-8	y	171
169	251	A9		121	12-11[illegible]-9	z	172
170	252	AA		122	12-11-2-9		322
171	253	AB		123	12-11-3-9		323
172	254	AC		124	12-11-4-9		324
173	255	AD		125	12-11-5-9		325
174	256	AE		126	12-11-6-9		326
175	257	AF		127	12-11-7-9		327

CHAR CODE			ASCII			EBCDIC	
Dec	Oct	Hex	Cntl/ Gph	to EBCDIC (Oct)	Hollerith	Cntl/ Gph	to ASCII (Oct)
176	260	B0		130	12-11-8-9		330
177	261	B1		131	11-1-8		331
178	262	B2		142	11-0-2-9		332
179	263	B3		143	11-0-3-9		333
180	264	B4		144	11-0-4-9		334
181	265	B5		145	11-0-5-9		335
182	266	B6		146	11-0-6-9		336
183	267	B7		147	11-0-7-9		337
184	270	B8		150	11-0-8-9		340
185	271	B9		151	0-1-8		341
186	272	BA		160	12-11-0		342
187	273	BB		161	12-11-0-1-9		343
188	274	BC		162	12-11-0-2-9		344
189	275	BD		163	12-11-0-3-9		345
190	276	BE		164	12-11-0-4-9		346
191	277	BF		165	12-11-0-5-9		347
192	300	C0		166	12-11-0-6-9	{	173
193	301	C1		167	12-11-0-7-9	A	101
194	302	C2		170	12-11-0-8-9	B	102
195	303	C3		200	12-0-1-8	C	103
196	304	C4		212	12-0-2-8	D	104
197	305	C5		213	12-0-3-8	E	105
198	306	C6		214	12-0-4-8	F	106
199	307	C7		215	12-0-5-8	G	107
200	310	C8		216	12-0-6-8	H	110
201	311	C9		217	12-0-7-8	I	111
202	312	CA		220	12-11-1-8		350
203	313	C8		232	12-11-2-8		351
204	314	CC		233	12-11-3-8	⑀	352
205	315	CD		234	12-11-4-8		353
206	316	CE		235	12-11-5-8	⑁	354
207	317	CF		236	12-11-6-8		355
208	320	D0		237	12-11-7-8	}	175
209	321	D1		240	11-0-1-8	J	112
210	322	D2		252	11-0-2-8	K	113
211	323	D3		253	11-0-3-8	L	114
212	324	D4		254	11-0-4-8	M	115
213	325	D5		255	11-0-5-8	N	116
214	326	D6		256	11-0-6-8	O	117
215	327	D7		257	11-0-7-8	P	120
216	330	D8		260	12-11-0-1-8	Q	121
217	331	D9		261	12-11-0-1	R	122
218	332	DA		262	12-11-0-2		356
219	333	DB		263	12-11-0-3		357
220	334	DC		264	12-11-0-4		360
221	335	DD		265	12-11-0-5		361
222	336	DE		266	12-11-0-6		362
223	337	DF		267	12-11-0-7		363
224	340	E0		270	12-11-0-8	\	134
225	341	E1		271	12-11-0-9		237
226	342	E2		272	12-11-0-2-8	S	123
227	343	E3		273	12-11-0-3-8	T	124
228	344	E4		274	12-11-0-4-8	U	126
229	345	E5		275	12-11-0-5-8	V	126
230	346	E6		276	12-11-0-6-8	W	127
231	347	E7		277	12-11-0-7-8	X	130
232	350	E8		312	12-0-2-8-9	Y	131
233	351	E9		313	12-0-3-8-9	Z	132
234	352	EA		314	12-0-4-8-9		364
235	353	EB		315	12-0-5-8-9		365
236	354	EC		316	12-0-6-8-9	⑂	366
237	355	ED		317	12-0-7-8-9		367
238	356	EE		332	12-11-2-8-9		370
239	357	EF		333	12-11-3-8-9		371
240	360	F0		334	12-11-4-8-9	0	060
241	361	F1		335	12-11-5-8-9	1	061
242	362	F2		336	12-11-6-8-9	2	062
243	363	F3		337	12-11-7-8-9	3	063
244	364	F4		352	11-0-2-8-9	4	064
245	365	F5		353	11-0-3-8-9	5	065
246	366	F6		354	11-0-4-8-9	6	066
247	367	F7		355	11-0-5-8-9	7	067
248	370	F8		358	11-0-6-8-9	8	070
249	371	F9		357	11-0-7-8-9	9	071
250	372	FA		372	12-11-0-2-8-9	|	372
251	373	FB		373	12-11-0-3-8-9		373
252	374	FC		374	12-11-0-4-8-9		374
253	375	FD		375	12-11-0-5-8-9		375
254	376	FE		376	12-11-0-6-8-9		376
255	377	FF.		377	12-11-0-7-8-9	EO	377

COBOL GLOSSARY

APPENDIX H

INTRODUCTION

The terms in this appendix are defined in accordance with their meaning as used in this document describing COBOL and may not have the same meaning for other languages.

These definitions are also intended to be either reference material or introductory material to be reviewed prior to reading the detailed language specifications. For this reason, these definitions are, in most instances, brief and do not include detailed syntactical rules.

DEFINITIONS

Abbreviated Combined Relation Condition. The combined condition that results from the explicit omission of a common subject or a common subject and common relational operator in a consecutive sequence of relation conditions.

Access Mode . The manner in which records are to be operated upon within a file.

Actual Decimal Point. The physical representation, using either of the decimal point characters period (.) or comma (,), of the decimal point position in a data item.

Alphabet-Name. A user-defined word, in the SPECIAL-NAMES paragraph of the Environment Division, that assigns a name to a specific character set and/or collating sequence.

Alphabetic Character. A character that belongs to the following set of letters: A,B,C,D,E,F,G,H,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z, and the space.

Alphanumeric Character. Any character in the computer's character set.

Alternate record Key. A key, other than the prime record key, whose contents identify a record within an indexed file.

Arithmetic Expression. An arithmetic expression can be an identifier or a numeric elementary item, a numeric literal, such identifiers and literals separated by arithmetic operators, two arithmetic expressions separated by an arithmetic operator, or a arithmetic expression enclosed in parentheses.

Arithmetic Operator. A single character, or a fixed two-character combination, that belongs to the following set:

Character	Meaning
+	addition
-	subtraction
*	multiplication
/	division
**	exponentiation

Ascending Key. A key upon the values of which data is ordered starting with the lowest value of key up to the highest value of key in accordance with the rules for comparing data items.

Assumed Decimal Point. A decimal point position which does not involve the existence of an actual character in a data item. The assumed decimal point has logical meaning but no physical representation.

At End Condition. A condition caused:

1. During the execution of a READ statement for a sequentially accessed file.

2. During the execution of a RETURN statement, when no next logical record exists for the associated sort or merge file.

3. During the execution of a SEARCH statement, when the search operation terminates without satisfying the condition specified in any of the associated WHEN phrases.

Block. A physical unit of data that is normally composed of one or more logical records. For mass storage files, a block may contain a portion of a logical record. The size of a block has no direct relationship to the size of the file within which the block is contained or to the size of the logical record (s) that are either continued within the block or that overlap the block. The term is synonymous with physical record.

Body Group. Generic name for a report group of TYPE DETAIL, CONTROL HEADING or CONTROL FOOTING. Has no meaning in COBOL II/3000.

Called Program. A program which is the object of a CALL statement combined at object time with the calling program to produce a run unit.

Calling Program. A program which executes a CALL to another program.

Cd-Name. A user-defined word that names an MCS interface area described in a communication description entry within the Communication Section of the Data Division. Has no meaning in COBOL II/3000.

Character. The basic indivisible unit of the language.

Character Position. A character position is the amount of physical storage required to store a single standard data format character described as usage is DISPLAY. Further characteristics of the physical storage are defined by the implementor.

Character-String. A sequence of contiguous characters which form a COBOL word, a literal, a PICTURE character-string, or a comment-entry.

CICS COBOL I/O CONTROL SYSTEM.

CLASS CONDITION. The proposition, for which a truth value can be determined, that the content of an item is wholly alphabetic or is wholly numeric.

Clause. A clause is an ordered set of consecutive COBOL character-strings whose purpose is to specify an attribute of an entry.

COBOL Character Set.. The complete COBOL character set consists of the 51 characters listed below:

Character	Meaning
0,1,...,	digit
A,B,...,	letter
	space (blank)
+	plus sign
-	minus sign (hyphen)
*	asterisk
/	stroke (virgule, slash)
=	equal sign
$	currency sign
,	comma (decimal point)
;	semicolon
.	period (decimal point)
"	quotation mark
(	left parenthesis
)	right parenthesis
>	greater than symbol
<	less than symbol

COBOL Word. (See Word)

Collating Sequence. The sequence in which the characters that are acceptable in a computer are ordered for purposes of sorting, merging, and comparing.

column. A character position within a print line. The colums are numbered from 1, by 1, starting at the leftmost character position of the print line and extending to the rightmost position of the print line.

Combined Condition. A condition that is the result of connecting two or more conditions with the 'AND' or the 'OR' logical operator.

Comment-Entry. An entry in the Identification Division that may be any combination of characters from the computer character set.

Comment Line. A source program line represented by an asterisk in the indicator area of the line and any characters from the computer's character set in area A and area B of that line. The comment line serves only for documentation in a program A special form of comment line represented by a stroke (/) in the indicator area of the line and any characters from the computer's character set in area A and area B of that line causes page ejection prior to printing the comment.

Communication Description Entry. An entry in the Communication Section of the Data Division that is composed of the level indicator CD, followed by a cd-name, and then followed by a set of clauses as required. It describes the interface between the Message Control (MCS) and the COBOL program. Has no meaning in COBOL II/3000.

Communication Device. A mechanism (hardware of hardware/ software) capable of sending data to a queue and/or receiving data from a queue. This mechanism may be a computer or a peripheral device. One or more programs containing communication description entries and residing within the same computer define one or more of these mechanism. Has no meaning in COBOL II/3000.

Connective. A reserved word that is used to:

1. Associate a data-name, paragraph-name, condition-name, or text-name with its qualifier.
2. Link two or more operands written in a series.
3. Form conditions (logical connectives). (See Logical Operator)

Contiguous Items. Items that are described by consecutive entries in the Data Division, and that bear a definite hierarchic relationship to each other.

Control Break. A change in the value of a data item that is referenced in the CONTROL clause. More generally, a change in the value of a data item that is used to control the hierarchical structure of a report. Not used in COBOL II/3000.

Control Break Level. The relative position within a control hierarchy at which the most major control break occurred. Not used in COBOL II/3000.

Control Data Item. A data item, a change in whose contents may produce a control break. Not used in COBOL II/3000.

Control Data-Name. A data-name that appears in a CONTROL clause and refers to a control data item. Not used in COBOL II/3000.

Control Footing. A report group that is presented at the end of the control group of which it is a member. Not used in COBOL II/3000.

Control Group. A set of body groups that is presented for a given value of a control data item or of FINAL. Each control group may begin with a CONTROL HEADING, end with a CONTROL FOOTING, and contain DETAIL report groups. Not used in COBOL II/3000.

Control Heading. A report group that is presented at the begining of the control group of which it is a member. Not used in COBOL II/3000.

Control Hierarchy. A designated sequence of report subdivisions defined by the positional order of FINAL and the data-names within a CONTROL clause. Not used in COBOL II/3000.

Counter. A data item used for storing numbers or number representations in a manner that permits these numbers to be increased or decreased by the value of another number, or to be changed or reset to zero or to an arbitary positive or negative value.

Currency Sign. The character '$' of the COBOL character set.

Currency Symbol. The character defined by the CURRENCY SIGN clause in the SPECIAL-NAMES paragraph. If no CURRENCY SIGN clause is present in a COBOL source program, the currency symbol is identical to the currency sign.

Current Record. The record which is available in the record area associated with the file.

Current Record Pointer. A conceptual entity that is used in the selection of the next record.

Data Clause. A clause that appears in a data description entry in the Data Division and provides information describing a particular attribute of a data item.

Data Description Entry. An entry in the Data Division that is composed of a level-number followed by a data-name, if required, and then followed by a set of data clauses, as required.

Data Item. A character or a set of contiguous characters (excluding in either case literals) defined as unit of data by the COBOL program.

Data-Name. A user defined word that names a data item described in a data description entry in the Data Division. When used in the general formats, 'data-name' represents a word which can neither be subscripted, indexed, nor qualified unless specificaly permitted by the rules for that format.

Debugging Line. A debugging line is any line with 'D' in the indicator area of the line. Not used in first release of COBOL II/3000.

Debugging Section. A debugging section is a section that contains a USE FOR DEBUGGING statement. Not used in first release of COBOL II/3000.

Declaratives. A set of one or more special purpose sections, written at the beginning of the Procedure Division, the first of which is preceded by the key word DECLARATIVES and the last of which is followed by the key words END DECLARATIVES. A declarative is composed of a section header, followed by a USE compiler directing sentence, followed by a set of zero, one or more associated paragraphs.

Declarative-Sentence. A compiler-directing sentence consisting of a single USE statement terminated by the separator period.

Delimiter. A character or a sequence of contiguous characters that identify the end of a string of characters and separates that string of characters from the following string of characters and separates that string of characters from the following string of characters. A delimiter is not part of the string of characters that it delimits.

Descending Key. A key upon the values of which data is ordered starting with the highest value of key down to the lowest value of key, in accordance with the rules for comparing data items.

Destination. The symbolic identification of the receiver of a transmission from a queue.

Digit Position. A digit position is the amount of physical storage required to store a single digit. This amount may vary depending on the usage of the data item describing the digit position. Further characteristics of the physical storage are defined by the implementor.

Division. A set of zero, one or more sections of paragraphs, called the division body, that are formed and combined in accordance with a specific set of rules. There are four (4) divisions in a COBOL program: Identification, Environment, Data, and Procedure.

Division Header. A combination of words followed by a period and a space that indicates that beginning of a division. The division headers are:

IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
DATA DIVISION.
PROCEDURE DIVISION [USING data-name-1 [data-name-2] ...] .

Dynamic Access. An access mode in which specific logical records can be obtained from or placed into a mass storage file in a non-sequential manner (see Random Access) and obtained from a file in a sequential manner (see Sequential Access), during the scope of the same OPEN statement.

Editing Character. A single character or a fixed two-character combination belonging to the following set:

Character	**Meaning**
B	space
0	zero
+	plus
-	minus
CR	credit
DB	debit
z	zero suppress
*	check protect
$	currency sign
,	comma (decimal point)
.	period (decimal point)
/	stroke (virgule, slash)

Elementary. A data item that is described as not being further logically subdivided.

End of Procedure Division. The physical position in a COBOL source program after which no further procedures appear.

Entry. Any desceiptive set of consecutive clauses terminated by a period and written in the Identification Division, Environment Division, or Data Division of a COBOL source program.

Environment Clause. A clause that appears as part of an Environment Division entry.

Execution Time. (See Object Time)

Extend Mode. The state of a file after execution of an OPEN statement, with the EXTEND phrase specified, for that file and before the execution of a CLOSE statement for that file.

Figurative Constant. A compiler generated value referenced through the use of certain reserved words.

File. A collection of records.

File Clause. A clause that appears as part of any of the following Data Division entries:

File description (FD)
Sort-merge file Description (SD)
Communication description (CD) (Not used in COBOL II/3000).

FILE-CONTROL. The name of an Environment Division paragraph in which the data files for a given source program are declared.

File Description Entry. An entry in the File Section of the Data Division that is composed of the level indicator FD, followed by a file-name, and then followed by a set of file clauses as required.

File-Name. A user-defined word that names a file description entry or a sort-merge file description entry within the File Section of the Data Division.

File Organization. The permanent logical file structure established at the time that a file is created.

File Section. The section of the Data Division that contains file description entries and sort-merge file desceiption entries together with their associated record descriptions.

Format. A specific arrangement of a set of data.

Group Item. A named contiguous set of elementary or group items.

High Order End. The leftmost character of a string of characters.

I-O-CONTROL. The name of an Environment Division paragraph in which object program requirements for specific input-output techniques, rerun points, sharing of same areas by several data files,and multiple file storage on a single input-output device are specified.

I-O Mode. The state of a file after execution of an OPEN statement, with the I-O phrase specified, for that file and before the execution of a CLOSE statement for that file.

Identifier. A data-name, followed as required, by the syntactically correct combination of qualifiers, subscripts, and indices necessary to make unique reference to a data item.

Imperative Statement. A statement that begins with an imperative verb and specifies an unconditional action to be taken. An imperative statement may consist of a sequence of imperative statements.

Index. A computer storage position or register, the contents of which represent the identification of a particular element in a table.

Index Data Item. A data item in which the value associated with an index-name can be stored in a form specified by the implementor.

Index-Name. A user-defined word that names an index associated with a specific table.

Indexed Data-Name. An identifier that is composed of a data-name, followed by one or more index-names enclosed in parentheses.

Indexed File. A file with indexed organization.

Indexed Organization. The permanent logical file structure in which each record is identified by the value of one or more keys within that record.

Input File. A file that is opened in the imput mode.

Input Mode. The state of a file after execution of an OPEN statement, with the INPUT phrase specified, for that file and before the execution of a CLOSE statement for that file.

Input-Output File. A file that is opened in the I-O mode.

Input-Output Section. The section of the Environment Division that names the files and the external media required by an object program and which provides information required for transmission and handling of data during execution of the object program.

Input Procedure. A set of statements that is executed each time a record is released to the sort file.

Integer. A numeric literal or a numeric data item that does not include any character positions to the right of the assumed decimal point. Where the term 'integer' appears in general formats, integer must not be a numeric data item, and must not be signed, nor zero unless explicitly allowed by the rules of the format.

Invalid Key Condition. A condition, at object time, caused when a specific value of the kay associated with an indexed or or relative file is determined to be invalid.

Key. A data item which identifies the location of a record, or a set of data items which serve to identify the ordering of data.

Key of Reference. The key, either prime or alternate, currently being used to access records within an indexed file.

Key Word. A reserved word whose presence is required when the format in which the word appears is used in a source program.

77-Level-Description-Entry. A data description entry that describes a noncontiguous data item with the level-number 77.

Level Indicator. Two alphabetic characters that identify a specific file type or a position in hierarchy.

Level-Number. A user-defined word which indicates the position a data item in the hierarchical structure of a logical record or which indicates special properties of a data description entry. A level-number is expressed as a one or two digit number. Level-numbers in the range 1 through 49 indicate the position of a data item in the hierachical structure of a logical record. Level- numbers in the range 1 through 9 may be written either as a single digit or as a zero followed by a significant digit. Level-numbers 66, 77, and 88 identify special properties of a data description entry.

Library-Name. A user-defined word that names a COBOL library that is to be used by the compiler for a given source program compilation.

Library Text. A sequence of character-strings and/or separators in a COBOL library.

Line. (See Report Line) Not used in COBOL II/3000.

Line Number. An integer that denotes the vertical position of a report line on a page. Not used for COBOL II/3000.

Linkage Section. The section in the Data Division of the called program that describes data items available from the calling program. These data items may be referred to by both the calling and called program.

Literal. A character-string whose value is implied by the ordered set of characters comprising the string.

Logical Operator. One of the reserved words AND, OR, or NOT. In the formation of a condition, both or either of AND and OR can be used as logical connectives. NOT can be used for logical negation.

Logical Record. The most inclusive data item. The level- number for a record is 0l. (See Report Writer Logical Record)

Low Order End. The rightmost character of a string of characters.

Mass Storage. A storage medium on which data may be organized and maintained in both a sequential and nonsequential manner.

Mass Storage Control System (MSCS). An input-output control system that directs, or controls, the processing of mass storage files.

Mass Storage File. A collection of records that is assigned to a mass storage medium.

MCS. (See Message Control System) (Not used in COBOL II/3000).

Merge File. A collection of records to be merged by a MERGE statement. The merge file is created and can be used only by the merge function.

Message. Data associated with an end of message indicator or an end of group indicator. (See Message Indicators) (Not used in COBOL II/3000).

Message Control System (MCS). A communication control system that supports the processing of messages. Not used in COBOL II/3000.

Message Count. The count of the number of complete messages that exist in the designated queue of messages. Not used in COBOL II/3000.

Message Indicators. EGI (end of group indicator), EMI (end of message indicator), and ESI (end of segment indicator) are conceptual indications that serve to notify the MCS that a specific condition exists (end of group, end of message, end of segment).

Within the hierarchy of EGI, EMI, and ESI, an EGI is conceptually equivalent to an ESI, EMI, and EGI. An EMI is conceptually equivalent to an ESI and EMI. Thus, a segment may be terminated by an ESI, EMI, or EGI. A message may be terminated by an EMI or EGI. Not used in COBOL II/3000.

Message Segment. Data that forms a logical subdivision of a message normally associated with an end of segment indicator. (See Message Indicators) Not used in COBOL II/3000.

Mnemonic-Name. A user-defined word that is associated in the Environment Division with a specified implementor-name.

MSCS. (See Mass Storage Control System)

Native Character Set. The ASCII character set associated with the computer specified in the OBJECT-COMPUTER paragraph.

Native Collating Sequence. The ASCII collating sequence associated with the computer specified in the OBJECT-COMPUTER paragraph.

Negated Combined Condition. The 'NOT' logical operator immediately followed by a parenthesized combined condition.

Negated Simple Condition. The 'NOT' logical operator immediately followed by a simple condition.

Next Executable Sentence. The next sentence to which control will be transferred after execution of the current statement is complete.

Next Executable Statement. The next statement to which control will be transferred after execution of the current statement is complete.

Next Record. The record which logically follows the current record of a file.

Noncontiguous Items. Elementary data items, in the Working- Storage and Linkage Sections, which bear no hierarchic relationship to other data items.

Nonnumeric Item. A data item whose description permits its contents to be composed of any conbination of characters taken from the computer's character set. Certain categories of nonnumeric items may be formed from more restricted character sets.

Nonnumeric Literal. A character-string bounded by quotation marks. The string of characters may include any character in the computer's character set. To represent a single quotation mark character within a nonnumeric literal, two contiguous quotation marks must be used.

Numeric Character. A character that belongs to the following set of digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

Numeric Item. A data item whose description restricts its contents to a value represented by characters chosen from the digits '0' through '9'; if signed, the item may also contain a '+', '-', or an operational sign.

Numeric Literal. A literal composed of one or more numeric characters that also may contain either a decimal point, or an algebraic sign, or both. The decimal point must not be the rightmost character. The algebraic sign, if present, must be the leftmost character.

OBJECT-COMPUTER. The name of an Environment Division paragraph in which the computer environment, within which the object program is executed, is described.

Object of Entry. A set of operands and reserved words, within a Data Division entry, that immediately follows the subject of the entry.

Object Program. A set or group of executable machine language instructions and other material designed to interact with data to provide problem solutions. In this context, an object program is generally the machine language result of the operation of a COBOL compiler on a source program. Where there is no danger of ambiguity, the word 'program' alone may be used in place of the phrase 'object program'.

Octal Literal. A literal composed of an "%" followed by between 1 and 11 octal digits with a maximum value of 37777777777. No algebraic sign nor decimal point may appear in the literal. Octal Literals may appear as numeric literals and in VALUE clauses.

Open Mode. The state of a file after execution of an OPEN statement for that file and before the execution of a CLOSE statement for that file. The particular open mode is specified in the OPEN statement as either INPUT, OUTPUT, I-O or EXTEND.

OPERAND. Whereas the general definition of operand is 'that component which is operated upon', for the purposes of this publication, any lowercase word (or words) that appears in a statement or entry format may be considered to be an operand and, as such, is an implied reference to the data indicated by the operand.

Operational Sign. An algebraic sign, associated with a numeric data item or a numeric literal, to indicate whether its value is positive or negative.

Optional Word. A reserved word that is included in a specific format only to improve the readability of the language and whose presence is optional to the user when the format in which the word appears is used in a source program.

Output File. A file that is opened in either the output mode or extend mode.

Output Mode. The state of a file after execution of an OPEN statement, with the OUTPUT or EXTEND phrase specified, for that file and before the execution of a CLOSE statement for that file.

Output Procedure. A set of statements to which control is given during execution of a SORT statement after the function is completed, or during execution of a MERGE statement after the merge function has selected the next record in merged order.

Page. A vertical division of a report representing a physical separation of report data, the separation being based on internal reporting requirements and/or external characteristics of the reporting medium.

Page Body. That part of the logical page in which lines can be written and/or spaced.

Page Footing. A report group that is presented at the end of a report page as determined by the Report Writer Control System.

Page Heading. A report group that is presented at the beginning of a report page and determined by the Report Writer Control System.

Paragraph. In the Procedure Division, a paragraph-name followed by a period and a space and by zero, one, or more sentences. In the Identification and Environment Divisions, a paragraph header followed by zero, one, or more entries.

Paragraph Header. A reserved word, followed by a period and a space that indicates the beginning of a paragraph in the Identification and Environment Divisions. The permissible paragraph headers are:

In the Identification Division:

PROGRAM-ID
AUTHOR.
INSTALLATION.
DATE-WRITTEN.
DATE-COMPILED.
SECURITY.

In the Environment Division:

SOURCE-COMPUTER.
OBJECT-COMPUTER.
SPECIAL-NAMES.
FILE-CONTROL.
I-O-CONTROL.

Paragraph-Name. A user-defined word that identifies and begins a paragraph in the Procedure Division.

Phrase. A phrase is an ordered set of one or more consecutive COBOL character-strings that form a portion of a COBOL procedural statement or of a COBOL clause.

Physical Record. (See Block)

Prime Record Key. A key whose contents uniquely identify a record within an indexed file.

Printable Group. A report group that contains at least one print line. Not used in COBOL II/3000.

Printable Item. A data item, the extent and contents of which are specified by a elementary report entry. This elementary report entry contains a COLUMN NUMBER clause, a PICTURE clause, and a SOURCE, SUM or VALUE clause.

Procedure. A paragraph or group of logicaly successive paragraphs, or a section or group of logically successive sections, within the Procedure Division.

Procedure-Name. A user-defined word which is used to name a paragraph or section in the Procedure Division. It consists of a paragraph-name (which may be qualified), or a section-name.

Program-Name. A user-defined word that identifies a COBOL source program.

Pseudo-Text. A sequence of character-strings and/or separators bounded by, but not including, pseudo-text delimiters.

Pseudo-Text Delimiter. Two contiguous equal sign (=) characters used to delimit pseudo-text.

Punctuation Character. A character that belongs to the following set:

Character	Meaning
,	comma
;	semicolon
.	period
"	quotation mark
(	left parenthesis
)	right parenthesis
	space
=	equal sign

Qualified Data-Name. An identifier that is composed of a data-name followed by one or more sets of either of the connectives OF an IN followed by a data-name qualifier.

Qualifier. 1. A data-name which is used in a reference together with another data name at a lower level in the same hierarchy.

2. A section-name which is used in a reference together with a paragraph-name specified in that section.

3. A library-name which is used in a reference together with a text-name associated with that library.

Queue. A logical collection of messages awaiting transmission or processing. Not used in COBOL II/3000.

Queue Name. A symbolic name that indicates to the MCS the logical path by which a message or a portion of a completed message may be accessible in a queue. Not used in COBOL II/3000.

RANDOM ACCESS. An access mode in which the program-specified value of a key data item identifies the logical record that is obtained from, deleted from or placed into a relative or indexed file.

Record. (See Logical Record)

Record Area. A storage area allocated for the purpose of processing the record described in a record description entry in the File Section.

Record Description. (See Record Description Entry)

Record Description Entry. The total set of data description entries associated with a particular record.

Record Key. A key, either the prime record key or an alternate record key, whose contents identify a record within an indexed file.

Record-Name. A user-defined word that names a record described in a record description entry in the Data Division.

Reference Format. A format that provides a standard method for describing COBOL source programs.

Relation. (See Relational Operator)

Relation Character. A character that belongs to the following set:

Character	Meaning
>	greater than
<	less than
=	equal to

Relation Condition. The proposition, for which a truth value can be determined, that the value of an arithmetic expression or data item has a specific relationship to the value of another arithmetic expression or data item. (See Relational Operator)

Relational Operator. A reserved word, a relation character, a group of consecutive reserved words, or a group of consecutive reserved words and relation characters used in the construction of a relation condition. The permissible operators their meaning are:

Relational Operator	Meaning
Is (NOT) GREATER THAN IS (NOT) >	Greater than or not greater than
IS (NOT) LESS THAN IS (NOT) <	Less than or not less than
IS (NOT) EQUAL TO IS (NOT) =	Equal to or not equal to

Relative File. A file with relative organization.

Relative Key. A key whose contents identify a logical record in a relative file.

Relative Organization. The permanent logical file structure in which each record is uniquely identified by an integer value greater than zero, which specifies the record's logical ordinal position in the file.

Report Clause. A clause, in the Report Section of the Data Division, that appears in a report description entry or a report group description entry. Not used in COBOL II/3000.

Report Description Entry. An entry in Report Section of the Data Division that is composed of the level indicator RD, followed by a report name, followed by a set of report clauses as required. Not used in COBOL II/3000.

Report File. An output file whose file description entry contains a REPORT clause. The contents of a report file consist of records that are written under control of the Report Writer Control System. Not used in COBOL II/3000.

Report Footing. A report group that is presented only at the end of a report. Not used in COBOL II/3000.

Report Group. In the Report Section of the Data Division, an 01 level-number entry and its subordinate entries. Not used in COBOL II/3000.

Report Group Description Entry. An entry in the Report Section of the Data Division that is composed of the level-number 01, the optional data-name, a TYPE clause, and an optional set of report clauses. Not used in COBOL II/3000.

Report Heading. A report group that is presented only at the beginning of a report. Not used in COBOL II/3000.

Report line. A division of a page representing one row of horizontal character positions. Each character position of a report line is aligned vertically beneath the corresponding character position of the report line above it. Report lines are numbered from 1, by 1, starting at the top of the page. Not used in COBOL II/3000.

Report-Name. A user-defined word that names a report described in a report description entry within the Report Section of the Data Division. Not used in COBOL II/3000.

Report Section. The section of the Data Division that contains one or more report description entries and their associated report group description entries. Not used in COBOL II/3000.

Report Writer Control System (RWCS). An object time control system, provided by the implementor, that accomplishes the construction of reports. Not used in COBOL II/3000.

Report Writer Logical Record. A record that consists of the Report Writer print line and associated control information necessary for its selection and vertical positioning. Not used in COBOL II/3000.

Reserved Word. A COBOL word specified in the list of words which may be used in COBOL source programs, but which must not appear in the programs as user-defined words or system-names.

Routine-Name. A user-defined word that identifies a procedure written in a language other than COBOL.

Run Unit. A set of one or more object program which function, at object time, as a unit to provide problem solutions.

RWCS. (See Report Writer Control System) Not used in COBOL II/3000.

Section. A set of zero, one, or more paragraphs or entries, called a section body, the first of which is preceded by a section header. Each section consists of the section header and the related section body.

Section Header. A combination of words followed by a period and a space that indicates the begining of a section in the Environment, Data and Procedure Division.

In the Environment and Data Divisions, a section header is composed of reserved words followed by a period and a space. The permissible section headers are:

In the Environment Division:

CONFIGURATION SECTION.
INPUT-OUTPUT SECTION.

In the data division:

FILE SECTION.
WORKING-STORAGE SECTION.
LINKAGE SECTION.
COMMUNICATION SECTION.
REPORT SECTION.

In the procedure Division, a section header is composed of a section-name, followed by reserved word SECTION, followed by a segment-number (optional). followed by a period and a space.

Section-Name. A user-defined word which names a section in the Procedure Division.

Segment-Number. A user-defined word which classifies sections in the Procedure Division for purposes of segmentation. Segment-numbers may contain only the characters '0', '1', ..., '9'. A segment-number may be expressed either as a one or two digit number.

Sentence. A sequence of one or more statements, the last of which is terminated by a period followed by a space.

Separator. A punctuation character used to delimit character-strings.

Sequential Access. An access mode in which logical records are obtained from or placed into a file in a consecutive predecessor-to-successor logical record sequence determined by the order of records in the file.

Sequential File. A file with sequential organization.

Sequential Organization. The permanent logical file structure in which a record is identified by a predecessor-successor relationship established when the record is placed into the file.

Sign Condition. The proposition, for which a truth value can be determined, that the algebraic value of a data item or an arithmetic expression is either less than, greater than, or equal to zero.

Simple Condition. Any single condition chosen from the set:

relation condition
class condition
condition-name condition
switch-status condition
sign condition
(simple-condition)

Sort File. A collection of records to be sorted by a SORT statement. The sort file is created and can be used by the sort function only.

Sort-Merge File Description Entry. An entry in the File Section of the Data Division that is composed of the level indicator SD, followed by a file-name, and then followed by a set of file clauses as required.

Source. The symbolic identification of the originator of a transmission to a queue.

SOURCE-COMPUTER. The name of an Environment Division paragraph in which the computer environment, within which the source program is compiled, is described.

Source Item. An identifier designated by a SOURCE clause that provides the value of a printable item.

Source Program. Although it is recognized that a source program may be represented by other forms and symbols, in this document it always refers to a syntactically correct set of COBOL statements beginning with an Identification Division and ending with the end of the Procedure Division. In contexts where there is no danger of ambiguity, the word 'program' alone may be used in place of the phrase 'source program'.

Special Character. A character that belongs to the following set:

Character	Meaning
+	plus sign
-	minus sign
*	asterisk
/	stroke (virgule, slash)
=	equal sign
$	currency sign
,	comma (decimal point)
;	semicolon
.	period (decimal point)
"	quotation mark
(	left parenthesis
)	right parenthesis
>	greater than symbol
<	less than symbol

Special-Character Word. A reserved word which is an arithmetic operator or a relation character.

SPECIAL-NAMES. The name of an Environment Division paragraph in which implementor-names are related to user specified mnemonic-names.

Special Registers. Compiler generated storage areas whose primary use is to store information produced in conjunction with the use of specific COBOL features.

Standard Data Format. The concept used in describing the characteristics of data in a COBOL Data division under which the characteristics or properties of the data are expressed in a form oriented to the appearance of the data on a printed page of infinite length and breadth, rather than a form oriented to the manner in which the data is stored internally in the computer, or on a particular external medium.

Statement. A syntactically valid combination of words and symbols written in the Procedure Division beginning with a verb.

Sub-Queue. A logical hierarchical division of a queue. Not used in COBOL II/3000.

Subject of Entry. An operand or reserved word that appears immediately following the level indicator or the level-number in a Data division entry.

Subprogram. (See Called Program)

Subscript. An integer whose value identifies a particular element in a table.

Subscripted Data-Name. An identifier that is composed of a data-name followed by one or more subscrips enclosed in parentheses.

Sum Counter. A signed numeric data item established by a SUM clause in the Report Section of the Data Division. The sum counter is used by the Report Writer Control System to contain the result of designated summing operations that take place during production of a report. Not used in COBOL II/3000.

Switch-Status Condition. The proposition, for which a truth value can be dtermined, that an implementor-defined switch, capable of being set to an 'on' or 'off' status, has been set to a specific status.

System-Name. A COBOL word which is used to communicate with the operating environment.

Table. A set of logically consecutive items of data that are defined in the Data Division by means of the OCCURS clause.

Table Element. A data item that belongs to the set of repeated items comprising a table.

Terminal. The originator of a transmission to a queue, or the receiver of a transmission from a queue. Not used in COBOL II/3000.

Text-Name. A user-defined word which identifies library text.

Text-Word. Any character-string or separator, except space, in a COBOL library or in pseudo-text.

Truth Value. The representation of the result of the evaluation of a condition in terms of one of two values:

true
false

Unary Operator. A plus (+) or a minus (-) sign, which precedes a variable or a left parenthesis in an arithmetic expression and which has the effect of multiplying the expression by +1 or -1 respectively.

Unit. A module of mass storage the dimensions of which determined by each implementor.

User-Defined Word. A COBOL word that must be supplied by the user to satisfy the format of a clause or statement.

Variable. A data item whose value may be changed by execution of the object program. A variable used in an arithmetic expression must be a numeric elementary item.

Verb. A word that expresses an action to be taken by a COBOL compiler or object program.

Word. A character-string of not more than 30 characters which forms a user-defined word, a system-name, or a reserved word.

Working-Storage Section. The section of the Data Division that describes working storage data items, composed of noncontiguous items or of working storage records or of both.

COBOLLOCK AND COBOLUNLOCK

APPENDIX I

File locking and unlocking can best be accomplished in COBOL II/3000 through the use of the EXCLUSIVE and UN-EXCLUSIVE statements. See Section XI for information on the use of these statements.

In order to simplify the conversion process from COBOL/3000 programs to COBOL II/3000 programs, an alternative method of file locking and unlocking is included in COBOL II/3000. This appendix describes this alternative method.

- 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.)

Note that locking a file does not necessarily exclude other users form accessing it. All that locking does is provide a method by which users can tell if a file is currently being accessed. Thus, if a file is to be shared, it is advisable that all users of the file lock it before they access it, and be sure to unlock the file when they are finished with it.

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. As with COBOLLOCK, it is necessary to use the SHR parameter in the MPE file equation which is issued for the file in the SELECT statement.

- The cond-code parameter returns a condition code indicating the results of the request to unlock the file. As with COBOLLOCK, the receiving field must be defined with a PIC S9(4) USAGE COMP. The settings for this code are:

-1 (CCL)	Request denied because the file was not opened with dynamic locking or the file is not opened.
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.
```

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. Note that it is necessary to use the SHR (share) parameter in the MPE file equation which is issued for the file in the SELECT statement.
- 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 receiving field must be defined with PIC S9(4) USAGE COMP. 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.)

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.
```

Using the File Description entry:

```
FILE SECTION.

FD UPDATE-FILE.
   01 DATA     PIC X(80).
```

and using the data pictures:

```
WORKING-STORAGE SECTION.

77 LOCK-CONDU PIC S9(4) USAGE COMP VALUE 0 OR 1.
77 COND-CODEU PIC S9(4) USAGE COMP.
```

COMPOSITE LANGUAGE SKELETON

APPENDIX J

This appendix contains the composite language skeleton of COBOL II. It is intended to display complete and syntactically correct formats.

The following is a summary of the formats shown in this appendix.

Identification Division general format	Page J-2
Environment Division general format	Page J-3
The three formats of the file control entry	Page J-6
Data Division general format	Page J-8
The three formats for a data description entry	Page J-10
Procedure Division general format	Page J-12
General format of verbs listed in alphabetical order	Page J-13
General format for conditions	Page J-29
Formats for qualification, subscripting, and identifiers	Page J-31
General format for COPY statement	Page J-33

GENERAL FORMAT FOR IDENTIFICATION DIVISION

$\left\{\begin{array}{l}\underline{\textbf{ID}}\\ \underline{\textbf{IDENTIFICATION}}\end{array}\right\}$ **<u>DIVISION.</u>**

PROGRAM-ID. ***program-name.***

[**<u>AUTHOR.</u>** [***comment-entry***] ...]

[**<u>INSTALLATION.</u>** [***comment-entry***] ...]

[**<u>DATE-WRITTEN.</u>** [***comment-entry***] ...]

[**<u>DATE-COMPILED.</u>** [***comment-entry***] ...]

[**<u>SECURITY.</u>** [***comment-entry***] ...]

[**<u>REMARKS.</u>** [***comment-entry***] ...]

GENERAL FORMAT FOR ENVIRONMENT DIVISION

```
ENVIRONMENT DIVISION.
┌ CONFIGURATION SECTION.                                        ┐
│ SOURCE-COMPUTER. source-computer-entry                        │
│ OBJECT-COMPUTER. object-computer-entry                        │
└ [ SPECIAL-NAMES. special-names-entry ]                        ┘
┌ INPUT-OUTPUT SECTION.                                         ┐
│ FILE-CONTROL. {file-control-entry} ...                        │
└ [ I-O-CONTROL. input-output-control-entry ]                   ┘
```

```
CONFIGURATION SECTION.

SOURCE-COMPUTER. computer-name [ WITH DEBUGGING MODE ] .

OBJECT-COMPUTER. computer-name

  [                           { WORDS      } ]
  [ , MEMORY SIZE integer     { CHARACTERS } ]
  [                           { MODULES    } ]

  [, PROGRAM COLLATING SEQUENCE IS alphabet-name ]

  [, SEGMENT-LIMIT IS segment-number ] .

[ SPECIAL-NAMES.
[
[   [ , function-name
[   [  { IS mnemonic-name [  , ON STATUS IS condition-name-1   ]
[   [  {                  [ [, OFF STATUS IS condition-name-2 ] ]
[   [  {
[   [  { IS mnemonic-name [  , OFF STATUS IS condition-name-2  ]
[   [  {                  [ [, ON STATUS IS condition-name-1 ]  ] } ...
[   [  {
[   [  { ON STATUS IS condition-name-1 [, OFF STATUS IS condition-name-2 ]
[   [  {
[   [  { OFF STATUS IS condition-name-2 [, ON STATUS IS condition-name-1 ]
[
[   [                       { STANDARD-1
[   [                       { NATIVE
[   [ , alphabet-name IS    { EBCDIC
[   [                       {            [ { { THROUGH } literal-2     } ]
[   [                       { literal-1  [ { { THRU    }               } ]
[   [                       {            [ { ALSO literal-3            } ]
[   [                       {            [ { [, ALSO literal-4 ] ...   } ]        ...
[   [                       {
[   [                       {   [            [ { { THROUGH } literal-6    } ] ]
[   [                       {   [ literal-5  [ { { THRU    }              } ] ] ...
[   [                       {   [            [ { ALSO literal-7           } ] ]
[   [                       {   [            [ { [, ALSO literal-8 ] ...  } ] ]
[
[   [, CURRENCY SIGN IS literal-9 ]
[
[   [, DECIMAL-POINT IS COMMA ] .
```

```
INPUT-OUTPUT SECTION.

FILE-CONTROL.

    { file-control-entry } ...

[ I-O-CONTROL.
    [ ; SAME [ RECORD      ] AREA FOR file-name-1 {, file-name-2 } ... ] ...
             [ SORT        ]
             [ SORT-MERGE  ]

    [ ; MULTIPLE FILE TAPE CONTAINS file-name-3 [ POSITION integer-1 ]
                 [ , file-name-4 [ POSITION integer-2 ] ] ... ] ... .   ]
```

GENERAL FORMAT FOR FILE CONTROL ENTRY

Format 1 - For Sequential Files

```
SELECT [ OPTIONAL ] file-name
   ASSIGN TO "file-info-1" [, "file-info-2" ] ...

  ┌           ┌ integer-1 ┐ ┌ AREA  ┐┐ ┌                  ┌ REEL ┐┐
  │ ; RESERVE ┤           ├ │       ││ │ FOR MULTIPLE ┤      ├│
  └           └ NO        ┘ └ AREAS ┘┘ └                  └ UNIT ┘┘

  [; ORGANIZATION IS SEQUENTIAL ]

  [; ACCESS MODE IS SEQUENTIAL ]

  [; FILE STATUS IS stat-item ].
```

Format 2 - For Relative Files

```
SELECT file-name
   ASSIGN TO "file-info-1" [, "file-info-2" ] ...

  ┌                         ┌ AREA  ┐┐
  │ ; RESERVE integer-1     │       ││
  └                         └ AREAS ┘┘

  ; ORGANIZATION IS RELATIVE

  ┌                    ┌ SEQUENTIAL [, RELATIVE KEY IS data-name-1 ]  ┐┐
  │ ; ACCESS MODE IS   ┤ ┌ RANDOM  ┐                                   ││
  │                    │ │         ├ , RELATIVE KEY IS data-name-1     ├│
  └                    └ └ DYNAMIC ┘                                   ┘┘

  [; FILE STATUS IS stat-item ] .
```

```
SELECT file-name
   ASSIGN TO [ integer-1 ] "file-info-1" [, "file-info-2" ] ...

 [           { integer-1 }  [ AREA  ] ]
 [ ; RESERVE { NO        }  [ AREAS ] ]

  ; ACCESS MODE IS RANDOM

  [; PROCESSING MODE IS SEQUENTIAL ]

  ; ACTUAL KEY IS data-name-1

 [; FILE STATUS IS stat-item ] .

 [ ;{ FILE-LIMIT IS    } { data-name-1 } THRU { data-name-2 }
 [  { FILE-LIMITS ARE  } { literal-1   }      { literal-2   }

     [ , { data-name-3 } THRU { data-name-4 } ] ... ]
     [   { literal-3   }      { literal-4   } ]
```

Format 4 - For Indexed Files

```
SELECT file-name
   ASSIGN TO "file-info-1" [, "file-info-2" ] ...

 [                        [ AREA  ] ]
 [ ; RESERVE integer-1    [ AREAS ] ]

  ; ORGANIZATION IS INDEXED

 [                    { SEQUENTIAL } ]
 [ ; ACCESS MODE IS   { RANDOM     } ]
 [                    { DYNAMIC    } ]

  ; RECORD KEY IS data-name-1 [ WITH DUPLICATES ]

 [; ALTERNATE RECORD KEY IS data-name-2 [ WITH DUPLICATES ]]

 [ ; FILE STATUS IS stat-item ].
```

Format 5 - For Sort-Merge Files

```
SELECT file-name

   ASSIGN TO "file-info-1" [, "file-info-2"] ...
```

GENERAL FORMAT FOR DATA DIVISION

```
[ FILE SECTION.
  [ FD file-name
    [ , BLOCK CONTAINS [ integer-1 TO ] integer-2 { RECORDS    } ]
                                                   { CHARACTERS }

                               { F }
    [ ; RECORDING MODE IS      { V } ]
                               { U }
                               { S }

    [ ; RECORD CONTAINS [ integer-3 TO ] integer-4 CHARACTERS ]
    data-name-9 [ , data-name-10 ] ...
      ; LABEL { RECORD IS    } { STANDARD }
              { RECORDS ARE  } { OMITTED  }
    [ ; VALUE OF label-info-1 IS { data-name-1 }
                                 { literal-1   }
        [ , label-info-2 IS { data-name-2 } ] ... ]
                            { literal-2   }
    [ ; DATA { RECORD IS   } data-name-3 [ , data-name-4 ] ... ]
             { RECORDS ARE }
    [ ; LINAGE IS { data-name-5 } LINES [ , WITH        { data-name-6 } ]
                  { integer-5   }         FOOTING AT    { integer-6   }
        [ , LINES AT TOP { data-name-7 } ] [ , LINES     { data-name-8 } ] ]
                         { integer-7   }     AT BOTTOM   { integer-8   }
    [ ; CODE-SET IS alphabet-name ]
  { record-description-entry } ... ]
  [ SD file-name
    [ ; RECORD CONTAINS [ integer-1 TO ] integer-2 CHARACTERS ]
    [ ; DATA { RECORD IS   } data-name-1 [ , data-name-2 ] ... ] .
             { RECORDS ARE }
  { record-description-entry } ... ]
]
```

WORKING-STORAGE SECTION.

[[*77-level-description-entry* / *record-description-entry*] ....]

LINKAGE SECTION.

[[*77-level-description-entry* / *record-description-entry*] ....]

GENERAL FORMAT FOR DATA DESCRIPTION ENTRY

Format 1

```
level-number { data-name-1 }
             { FILLER      }

  [ ; REDEFINES data-name-2 ]

  [ ; { PICTURE } IS character-string ]
      { PIC     }

  [ ; [ USAGE IS ] { COMPUTATIONAL-3 } ]
                   { COMP-3          }
                   { COMPUTATIONAL   }
                   { COMP            }
                   { DISPLAY         }
                   { INDEX           }

  [ ; [ SIGN IS ] { LEADING  } [ SEPARATE CHARACTER ] ]
                  { TRAILING }

  [ ; 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 ] ... ]               ]

  [ ; { SYNCHRONIZED } [ LEFT  ] ]
      { SYNC         } [ RIGHT ]

  [ ; { JUSTIFIED } RIGHT ]
      { JUST      }

  [ ; BLANK WHEN ZERO ]

  [ ; VALUE IS literal ] .
```

Format 2

66 *data-name-1* ; **RENAMES** *data-name-2* [{ **THROUGH** / **THRU** } *data-name-3*] .

Format 3

88 *condition-name* ; { **VALUE IS** / **VALUES ARE** } *literal-1* [{ **THROUGH** / **THRU** } *literal-2*]

[, *literal-3* [{ **THROUGH** / **THRU** } *literal-4*]] ... .

GENERAL FORMAT FOR PROCEDURE DIVISION

Format 1

{ *paragraph-name.* [*sentence*] ... } ...

Format 2

[
DECLARATIVES.

{ *section-name* **SECTION** [*segment-number*]. *declarative-sentence*
[*paragraph-name.* [*sentence*] ...] ... } ...

END DECLARATIVES.
]

{ *section-name* **SECTION** [*segment-number*].
[*paragraph-name.* [*sentence*] ...] ... } ...

GENERAL FORMAT FOR VERBS

Format 1

```
ACCEPT identifier [ FREE ] [ FROM { SYSIN          } ]
                                  { CONSOLE        }
                                  { mnemonic-name  }
```

Format 2

```
ACCEPT identifier FREE [ FROM { SYSIN          } ]
                              { CONSOLE        }
                              { mnemonic-name  }

      ; ON INPUT ERROR imperative-statement
```

Format 3

```
ACCEPT identifier [ FREE] FROM { DATE }
                               { DAY  }
                               { TIME }
```

Format 1

```
ADD { identifier-1 } [ , identifier-2 ] ... TO identifier-m
    { literal-1    } [ , literal-2    ]

    [ ROUNDED ] [, 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 ]
           [, identifier-n [ ROUNDED ] ] ...
    [; ON SIZE ERROR imperative-statement ]
```

Format 3

```
ADD { CORRESPONDING }  identifier-1 TO identifier-2
    { CORR          }

    [ ROUNDED ] [; ON SIZE ERROR imperative-statement ]
```

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

CALL Statement Format (ANSI COBOL '74)

```
CALL { identifier-1 } [ USING identifier-2 [ , identifier-3 ] ... . ]
     { literal-1    }

[ ; ON OVERFLOW imperative-statement ]
```

CALL Statement Format (COBOL II/3000)

```
                                               [         { \\             } ]
CALL [ INTRINSIC ] { identifier-1 }            [         { @identifier-2  } ]
                   { literal-1    }            [  USING  { identifier-2   } ]
                                               [         { \ identifier-2 \ } ]
                                               [         { \ literal-2 \  } ]

    [   [   { \\             } ]     ]
    [   [   { @identifier-3  } ]     ]
    [   [ , { identifier-3   } ] ... ]  [ GIVING identifier-4 ]
    [   [   { literal-3      } ]     ]
    [   [   { \ identifier-3 \ } ]   ]
    [   [   { \ literal-3 \  } ]     ]

    [ ; ON OVERFLOW imperative-statement ]
```

Cancel Statement Format

```
CANCEL { identifier-1 }  [ , identifier-2 ]
       { literal-1    }  [ , literal-2    ]
```

Format 1 - Sequential Files

```
                  ┌ { REEL } ┌ WITH NO REWIND ┐ ┐
CLOSE file-name-1 │ { UNIT } └ FOR REMOVAL    ┘ │
                  │                             │
                  └ WITH { NO REWIND }          ┘
                         { LOCK      }

    ┌               ┌ { REEL } ┌ WITH NO REWIND ┐ ┐ ┐
    │ , file-name-2 │ { UNIT } └ FOR REMOVAL    ┘ │ │ ...
    │               │                             │ │
    └               └ WITH { NO REWIND }          ┘ ┘
                           { LOCK      }
```

Format 2 - Random, relative, or indexed files

```
CLOSE file-1 [ WITH LOCK ] [ , file-2 [ WITH LOCK ] ] ...
```

```
COMPUTE identifier-1 [ ROUNDED ] [ , identifier-2 [ ROUNDED ] ] ...

  = arithmetic-expression
       [ ; ON SIZE ERROR imperative-statement]
```

```
DELETE file-name RECORD
       [ ; INVALID KEY imperative-statement ]
```

```
                                                        { SYSOUT        }
DISPLAY { identifier-1 } , [ identifier-2 ] ... [ UPON  { CONSOLE       } ]
        { literal-1    } , [ literal-2    ]             { mnemonic-name }
```

Format 1

```
DIVIDE { identifier-1 }  INTO  identifier-2  [ ROUNDED ]
       { literal-1    }

   [ , identifier-3  [ ROUNDED ] ] ... [ ; ON SIZE ERROR imperative-statement ]
```

Format 2

```
DIVIDE { identifier-1 }  INTO  { identifier-2 }  GIVING  identifier-3  [ ROUNDED ]
       { literal-1    }        { literal-2    }

   [ , identifier-4  [ ROUNDED ] ] ... [ ; ON SIZE ERROR imperative-statement ]
```

Format 3

```
DIVIDE { identifier-1 }  BY  { identifier-2 }  GIVING  identifier-3  [ ROUNDED ]
       { literal-1    }      { literal-2    }

   [ , identifier-4  [ ROUNDED ] ] ... [ ; ON SIZE ERROR imperative-statement ]
```

Format 4

```
DIVIDE { identifier-1 }  INTO  { identifier-2 }  GIVING  identifier-3  [ ROUNDED ]
       { literal-1    }        { literal-2    }

   REMAINDER identifier-4  [; ON SIZE ERROR imperative-statement ]
```

Format 5

```
DIVIDE { identifier-1 }  BY  { identifier-2 }  GIVING  identifier-3  [ ROUNDED ]
       { literal-1    }      { literal-2    }

   REMAINDER identifier-4  [ ; ON SIZE ERROR imperative-statement ]
```

ENTER *language-name* [*routine-name*].

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

EXAMINE *identifier*

{ **TALLYING** { **UNTIL FIRST** / **ALL** / **LEADING** } *literal-1* [**REPLACING** **BY** *literal-2*]
  REPLACING { **ALL** / **LEADING** / [**UNTIL**] **FIRST** } *literal-3* **BY** *literal-4* }

EXCLUSIVE *file-name* [**CONDITIONALLY**]

EXIT **PROGRAM**

paragraph-name.
EXIT.
paragraph/section-name.

GOBACK

Format 1

GO TO [*procedure-name-1*]

Format 2

GO TO *procedure-name-1* [, *procedure-name-2*] ...
, *procedure-name-n* **DEPENDING ON** *identifier*

IF *condition* ; THEN { *statement-1* / **NEXT SENTENCE** } ; { **ELSE** **NEXT** **SENTENCE** / **ELSE** *statement-2* }

Format 1

```
INSPECT identifier-1 TALLYING

{ , identifier-2 FOR { , { { ALL     } { identifier-3 } }
                         { { LEADING } { literal-1    } }
                         { CHARACTERS                   }

      [ { BEFORE } INITIAL { identifier-4 } ] } }
      [ { AFTER  }         { literal-2    } ] } }
```

Format 2

```
INSPECT identifier-1 REPLACING

{ CHARACTERS BY { identifier-6 } [ { BEFORE } INITIAL { identifier-7 } ]                 }
{               { literal-4    } [ { AFTER  }         { literal-5    } ]                 }
{                                                                                        }
{ { , { ALL     } { , { identifier-5 } BY { identifier-6 }                               }
{ {   { LEADING } {   { literal-3    }    { literal-4    }                               }
{ {   { FIRST   } {                                                                      }
{                                                                                        }
{                  [ { BEFORE } INITIAL { identifier-7 } ] } }                           }
{                  [ { AFTER  }         { literal-5    } ] } }                           }
```

Format 3

```
INSPECT identifier-1 TALLYING

{ , identifier-2 FOR { , { { ALL     } { identifier-3 } } 
                            { LEADING } { literal-1    } }
                          { CHARACTERS                   }

    [ { BEFORE } INITIAL { identifier-4 } ] } }
      { AFTER  }         { literal-2    }


INSPECT identifier-1 REPLACING

{ CHARACTERS BY { identifier-6 } [ { BEFORE } INITIAL { identifier-7 } ]
                { literal-4    }   { AFTER  }         { literal-5    }

  { , { ALL     } { , { identifier-5 } BY { identifier-6 }
      { LEADING }     { literal-3    }    { literal-4    }
      { FIRST   }

              [ { BEFORE } INITIAL { identifier-7 } ] } }
                { AFTER  }         { literal-5    }      }


MERGE file-name-1 ON { ASCENDING  } KEY data-name-1 [ , data-name-2 ] ...
                     { DESCENDING }

             [ ON { ASCENDING  } KEY data-name-3 [ , data-name-4 ] ... ] ...
                  { DESCENDING }

   [ COLLATING SEQUENCE IS alphabet-name ]

     USING file-name-2 , file-name-3 , [ , file-name-4 ] ...

   { OUTPUT PROCEDURE IS section-name-1 [ { THROUGH } section-name-2 ] }
                                          { THRU    }
   { GIVING file-name-5                                               }
```

Format 1

```
        ⎧ identifier-1 ⎫
MOVE    ⎨ spec-reg     ⎬  TO identifier-2 [ , identifier-3 ] ...
        ⎩ literal      ⎭
```

Format 2

```
        ⎧ CORRESPONDING ⎫
MOVE    ⎩ CORR          ⎭  identifier-1 TO identifier-2
```

Format 1

```
            ⎧ identifier-1 ⎫
MULTIPLY    ⎩ literal-1    ⎭  BY identifier-2 [ ROUNDED ]

      [ , identifier-3 [ ROUNDED ] ] ...
                   [ ; ON SIZE ERROR imperative-statement ]
```

Format 2

```
            ⎧ identifier-1 ⎫        ⎧ identifier-2 ⎫
MULTIPLY    ⎩ literal-1    ⎭   BY   ⎩ literal-2    ⎭

    GIVING identifier-3 [ ROUNDED ] [ , identifier-4 [ ROUNDED ] ] ...
      [ ; ON SIZE ERROR imperative-statement ]
```

OPEN { INPUT *file-name-1* [REVERSED | WITH NO REWIND] [, *file-name-2* [REVERSED | WITH NO REWIND]] ...
OUTPUT *file-name-3* [WITH NO REWIND]
[, *file-name-4* [WITH NO REWIND]] ...
I-O *file-name-5* [, *file-name-6*] ...
EXTEND *file-name-7* [, *file-name-8*] ... } ...

Format 1

PERFORM *procedure-name-1* [{ **THROUGH** | **THRU** } *procedure-name-2*]

Format 2

PERFORM *procedure-name-1* [{ **THROUGH** | **THRU** } *procedure-name-2*] { *identifier-1* | *integer-1* } **TIMES**

Format 3

PERFORM *procedure-name-1* [{ **THROUGH** | **THRU** } *procedure-name-2*] **UNTIL** *condition-1*

Format 4

PERFORM *procedure-name-1* [{ **THROUGH** | **THRU** } *procedure-name-2*]

VARYING { *identifier-2* | *index-name-1* } **FROM** { *identifier-3* | *index-name-2* | *literal-1* }

```
        BY    { identifier-4 }   UNTIL  condition-1
              { literal-3    }
[                                          { identifier-6   } ]
[ AFTER  { identifier-5   }   FROM    { index-name-4   } ]
[        { index-name-3   }           { literal-3      } ]
[                                                         ]
[       BY    { identifier-7 }   UNTIL  condition-2       ]
[             { literal-4    }                            ]
[ [                                     { identifier-9   } ]
[ [AFTER  { identifier-8   }   FROM    { index-name-6   } ]
[ [       { index-name-5   }           { literal-5      } ]
[                                                         ]
[      BY    { identifier-10 }   UNTIL  condition-3 ]     ]
[            { literal-6     }                      ]     ]
```

Format 1 - Sequential files

READ *file-name* RECORD [INTO *identifier*]
[; AT END *imperative-statement*]

Format 2 - Relative, random, and indexed files

READ *file-name*[NEXT] RECORD [INTO *identifier*]
[; AT END *imperative-statement*]

Format 3 - Relative and random files

READ *file-name* RECORD [INTO *identifier*]
[; INVALID KEY *imperative-statement*]

Format 4 - Indexed files

READ *file-name* RECORD [INTO *identifier*] [KEY IS data-name]
[; INVALID KEY *imperative-statement*]

RELEASE *record-name* [**FROM** *identifier*]

RETURN *file-name* RECORD [**INTO** *identifier*]
; AT **END** *imperative-statement*

Format 1 - Sequential Files

REWRITE *record-name* [**FROM** *identifier*]

Format 2 - Relative, Random, and Indexed Files

REWRITE *record-name* [**FROM** *identifier*]
[; **INVALID** KEY *imperative-statement*]

Format 1

SEARCH *identifier-1* [**VARYING** { *identifier-2* / *index-name-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 ]

                        { IS EQUAL TO }   { identifier-3              }
 ; WHEN { data-name-1   { IS =        }   { literal-1                 } }
        {                                 { arithmetic-expression-1   } }
        { condition-name-1                                              }

    [        { data-name-2 { IS EQUAL TO } { identifier-4             } } ]
    [ AND    {             { IS =        } { literal-2                } } ] ...
    [        {                             { arithmetic-expression-2  } } ]
    [        { condition-name-2                                         } ]

    { imperative-statement-2 }
    { NEXT SENTENCE          }
```

```
SEEK file-name RECORD
```

Format 1

```
SET { identifier-1  [ , identifier-2 ] ...  } TO { identifier-3  }
    { index-name-1  [ , index-name-2 ] ...  }    { index-name-3  }
                                                 { integer-1     }
```

Format 2

```
SET index-name-4 [ , index-name-5 ] ... { UP BY   } { identifier-4 }
                                        { DOWN BY } { integer-2    }
```

```
SORT file-name-1 ON {ASCENDING } KEY data-name-1 [ ,data-name-2 ] ...
                    {DESCENDING}

        [ON {ASCENDING } KEY data-name-3 [ ,data-name-4 ] ... ] ...
        [   {DESCENDING}                                      ]

  [ COLLATING SEQUENCE IS alphabet-name ]

  {INPUT PROCEDURE IS section-name-1 [{THROUGH} section-name-2]}
  {                                  [{THRU   }               ]}
  {USING file-name-2 [ , file-name-3 ] ...                     }

  {OUTPUT PROCEDURE IS section-name-3 [{THROUGH} section-name-4]}
  {                                   [{THRU   }               ]}
  {GIVING file-name-4                                           }


                      [     {IS EQUAL TO        }           ]
                      [     {IS =               }           ]
START file-name       [ KEY {IS GREATER THAN    } data-name ]
                      [     {IS >               }           ]
                      [     {IS NOT LESS THAN   }           ]
                      [     {IS NOT <           }           ]

  [ ; INVALID KEY imperative-statement ]


STOP {RUN    }
     {literal}
```

```
                                                       { identifier-3 }
STRING { identifier-1 } [ , identifier-2 ] ... DELIMITED BY { literal-3    }
       { literal-1    } [ , literal-2    ]                  { SIZE         }

     [                                                      { identifier-6 }   ]
     [ , { identifier-4 } [ , identifier-5 ] ... DELIMITED BY { literal-6    }   ] ...
     [   { literal-4    } [ , literal-5    ]                  { SIZE         }   ]

     INTO identifier-7 [ WITH POINTER identifier-8 ]

     [ ; ON OVERFLOW imperative-statement ]
```

Format 1

```
SUBTRACT { identifier-1 } [ , identifier-2 ] ... FROM identifier-m [ ROUNDED ]
         { literal-1    } [ , literal-2    ]

    , identifier-n [ ROUNDED ]   ...  [ ; ON SIZE ERROR imperative-statement ]
```

Format 2

```
SUBTRACT { identifier-1 } [ , identifier-2 ] ... FROM { identifier-m }
         { literal-1    } [ , literal-2    ]          { literal-m    }

    GIVING identifier-n [ ROUNDED ] [ , identifier-o [ ROUNDED ] ] ...

   [ ; ON SIZE ERROR imperative-statement ]
```

Format 3

```
SUBTRACT { CORRESPONDING } identifier-1  FROM identifier-2 [ ROUNDED ]
         { CORR          }

   [ ; ON SIZE ERROR imperative-statement ]
```

```
UNSTRING identifier-1

  [ DELIMITED BY [ ALL ] { identifier-2 } [ , OR [ ALL ] { identifier-3 } ] ... ]
                         { literal-1    }                 { literal-2    }

   INTO identifier-4 [ , DELIMITER IN identifier-5 ] [ , COUNT IN identifier-6 ]

     [ , identifier-7 [ , DELIMITER IN identifier-8 ]
       [ , COUNT IN identifier-9 ] ] ...

  [ WITH POINTER identifier-10 ] [ TALLYING IN identifier-11 ]

  [ ; ON OVERFLOW imperative-statement ]
```

Format 1 - Error Handling Procedures

```
USE AFTER STANDARD { EXCEPTION } PROCEDURE
                   { ERROR     }

ON     { file-name-1 [ , file-name-2, ... ] }
       { INPUT                              }
       { OUTPUT                             } .
       { I-O                                }
       { EXTEND                             }
```

Format 2 - User Label Procedures

```
USE AFTER STANDARD BEGINNING [ FILE ]

                              { file-name-1 [ , file-name-2 ] ... }
                              { INPUT                             }
    LABEL PROCEDURE ON        { OUTPUT                            } .
                              { I-O                               }
                              { EXTEND                            }
```

Format 1 Sequential Files

```
WRITE record-name [ FROM identifier-1 ]

[                            { { { identifier-2 }  [ LINE  ] } } ]
[ { BEFORE }  ADVANCING      { { { integer      }  [ LINES ] } } ]
[ { AFTER  }                 {                               } ]
[                            {   { mnemonic-name }           } ]
[                            {   { PAGE          }           } ]

[        END-OF-PAGE                              ]
[ ; AT                    imperative-statement    ]
[        EOP                                      ]
```

Format 2 Relative, Indexed, or Random-Access Files

```
WRITE record-name [ FROM identifier-1 ]

                   [ ; INVALID KEY imperative-statement ]
```

GENERAL FORMAT FOR CONDITIONS

Relation Condition Format

identifier-1
literal-1
arithmetic-expression-1

IS [NOT] GREATER THAN
IS [NOT] LESS THAN
IS [NOT] EQUAL TO
IS [NOT] >
IS [NOT] <
IS [NOT] =

identifier-2
literal-2
arithmetic-expression-2

Class Condition Format

identifier IS [NOT]

NUMERIC
ALPHABETIC

Sign Condition Format

arithmetic-expression IS [NOT]

POSITIVE
NEGATIVE
ZERO

Condition-Name Condition Format

condition-name

Switch-Status Condition Format

condition-name

Negated Simple Condition Format

NOT *simple-condition*

General Format for Conditions

Combined Condition Format

{ *condition* { **AND** / **OR** } *condition* } ...

(**AND** and **OR** are underlined.)

Abbreviated Combined Relation Condition Format

Relation-condition { **AND** / **OR** } [**NOT**] [*relational-operator*] *object* ...

(**AND**, **OR** and **NOT** are underlined.)

MISCELLANEOUS FORMATS

Qualification

```
{ data-name-1    }  [ { OF }  data-name-2 ] ...
{ condition-name }  [ { IN }              ]
```

Format 2

```
paragraph-name  [ { OF }  section-name ]
                [ { IN }               ]
```

Format 3

```
text-name  [ { OF }  library-name ]
           [ { IN }               ]
```

Subscripting

```
{ data-name      }  ( subscript-1 [ , subscript-2  [ , subscript-3 ] ] )
{ condition-name }
```

Indexing

```
{ data-name      }  ( { index-name-1  [ { ± } literal-2 ] }
{ condition-name }    { literal-1                         }

     [ , { index-name-2  [ { ± } literal-4 ] }  [ , { index-name-3  [ { ± } literal-6 ] } ] ] )
     [   { literal-3                         }  [   { literal-5                         } ] ]
```

Identifier

Format 1

```
data-name-1 [ { OF | IN } data-name-2 ] ... [ ( subscript-1 [ , subscript-2

    [ , subscript-3 ] ] ) ]
```

Format 2

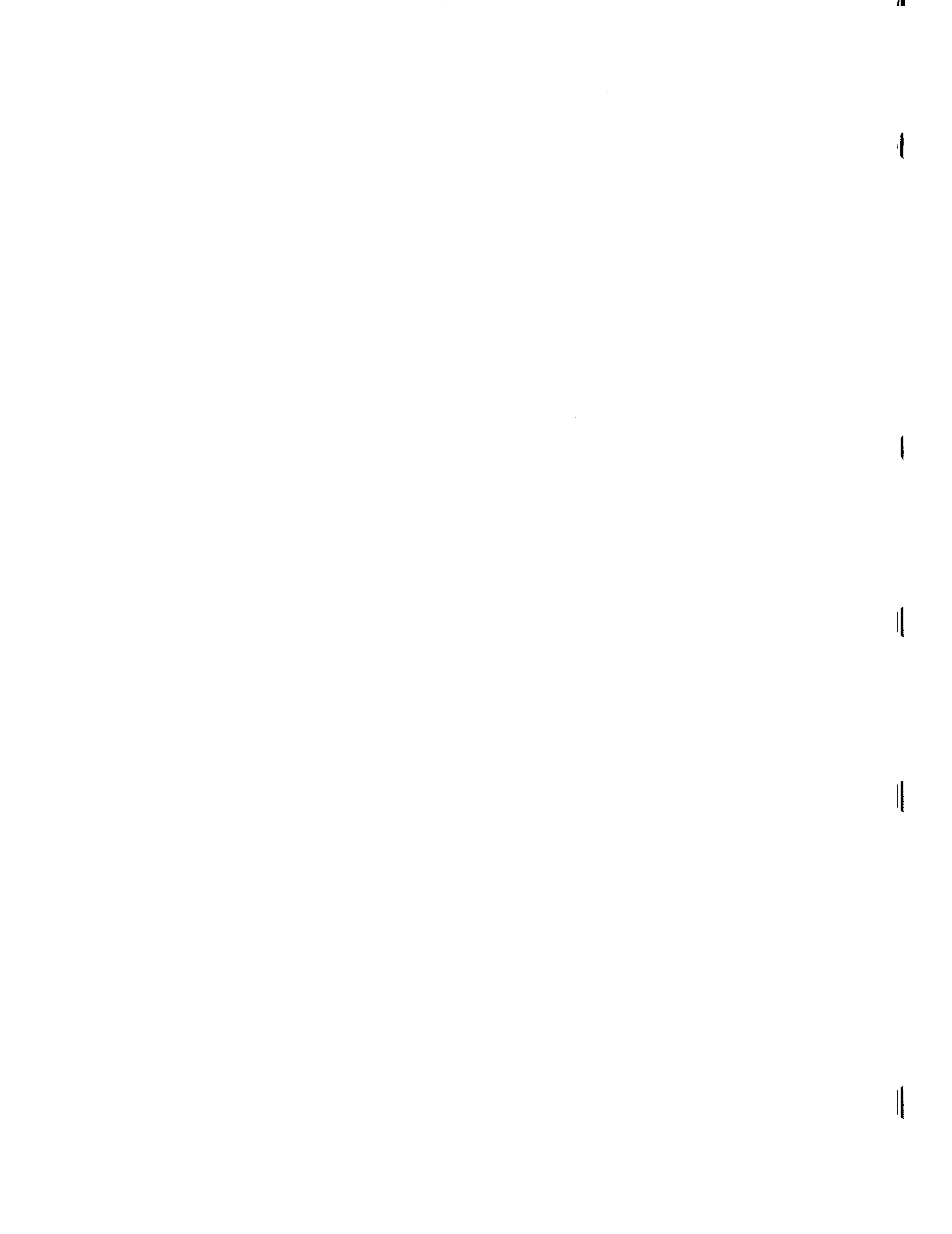
```
data-name-1 [ { OF | IN } data-name-2 ] ... [ ( { index-name-1 [ { + | - } literal-2 ] | literal-1 }

    [ , { index-name-2 [ { + | - } literal-4 ] | literal-3 } [ , { index-name-3 [ { + | - } literal-6 ] | literal-5 } ] ] ) ]
```

General Format for Copy Statement

$$\underline{\text{COPY}}\ \textit{text-name}\ \left[\begin{Bmatrix}\underline{\text{OF}}\\ \underline{\text{IN}}\end{Bmatrix}\textit{library-name}\right]\ [\ \text{NOLIST}\ ]$$

$$\left[\ \underline{\text{REPLACING}}\ \left\{,\begin{Bmatrix}==\textit{pseudo-text-1}==\\ \textit{identifier-1}\\ \textit{literal-1}\\ \textit{word-1}\end{Bmatrix}\ \underline{\text{BY}}\ \begin{Bmatrix}==\textit{pseudo-text-2}==\\ \textit{identifier-2}\\ \textit{literal-2}\\ \textit{word-2}\end{Bmatrix}\right\}\ ...\ \right]$$



COBOL RESERVED WORD LIST

APPENDIX K

ACCEPT
ACCESS
ACTUAL
ADD
ADVANCING
AFTER
ALL
ALPHABETIC
ALSO
ALTER
ALTERNATE
AND
ARE
AREA
AREAS
ASCENDING
ASSIGN
AT
AUTHOR

BEFORE
BEGINNING
BLANK
BLOCK
BOTTOM
BY

C01
C02
C03
C04
C05
C06
C07
C08
C09
C10
C11
C12
CALL
CANCEL
CD
CF
CH
CHARACTER
CHARACTERS
CLOCK-UNITS
CLOSE
COBOL
CODE
CODE-SET
COLLATING
COLUMN
COMMA
COMMUNICATION
COMP
COMP3
COMPUTATION
COMPUTATIONAL-3
COMPUTE
CONDITIONAL
CONDITION-CODE
CONFIGURATION
CONSOLE
CONTAINS
CONTROL
CONTROLS
COPY
CORR
CORRESPONDING
COUNT
CURRENCY
CURRENT-DATE

DATA
DATE
DATE-COMPILED
DATE-WRITTEN
DAY
DE
DEBUG-CONTENTS
DEBUG-ITEM
DEBUG-LINE
DEBUG-NAME
DEBUG-SUB-1
DEBUG-SUB-2
DEBUG-SUB-3
DEBUGGING
DECIMAL-POINT
DECLARATIVES
DELETE
DELIMITED
DELIMITER
DEPENDING
DESCENDING
DESTINATION
DETAIL
DISABLE
DISPLAY
DIVIDE
DIVISION
DOWN
DUPLICATES
DYNAMIC

EBCDIC
EGI
ELSE
EMI
ENABLE
END
END-OF-PAGE
ENDING
ENTER
ENTRY
ENVIRONMENT
EOP
EQUAL
ERROR
ESI
EVERY
EXAMINE
EXCEPTION
EXCLUSIVE
EXDATE
EXIT
EXTEND

FD
FILE
FILE-CONTROL
FILE-LIMIT
FILE-LIMITS
FILLER
FINAL
FIRST
FOOTING
FOR
FREE
FROM

GENERATE
GIVING
GO
GOBACK
GREATER
GROUP

HEADING
HIGH-VALUE
HIGH-VALUES

I-O
I-O-CONTROL
IDENTIFICATION
IF
IN
INDEX
INDEXED
INDICATE
INITIAL
ITITIATE
INPUT
INPUT-OUTPUT
INSPECT
INSTALLATION
INTO
INTRINSIC
INVALID
IS

JUST
JUSTIFIED

KEY

LABEL
LABELS
LAST
LEADING
LEFT
LENGTH
LESS
LINES
LIMIT

LIMITS
LINAGE
LINAGE-COUNTER
LINE
LINE-COUNTER
LINES
LINKAGE
LOCK
LOW-VALUE
LOW-VALUES

MEMORY
MERGE
MESSAGE
MODE
MODULES
MORE-LABELS
MOVE
MULTIPLE
MULTIPLY

NATIVE
NEGATIVE
NEXT
NO
NOLIST
NOT
NUMBER
NUMERIC

OBJECT-COMPUTER
OCCURS
OF
OFF
OMITTED
ON
OPEN
OPTIONAL
OR
ORGANIZATION
OUTPUT
OVERFLOW

PAGE
PAGE-COUNTER
PERFORM
PF
PH
PIC
PICTURE
PLUS
POINTER
POSITION
POSITIVE
PRINTING
PROCEDURE
PROCEDURES
PROCEED
PROCESSING
PROGRAM
PROGRAM-ID

QUEUE
QUOTE
QUOTES

RANDOM
RD
READ
RECEIVE
RECORD
RECORDING
RECORDS
REDEFINES
REEL
REFERENCES
RELATIVE
RELEASE
REMAINDER
REMOVAL
RENAMES
REPLACING
REPORT
REPORTING
REPORTS
RERUN
RESERVE
RESET
RETURN
REVERSED
REWIND
RF
RH
RIGHT
ROUNDED
RUN

SAME
SD
SEARCH
SECTION
SECURITY
SEEK
SEGMENT
SEGMENT-LIMIT
SELECT
SEND
SENTENCE
SEPARATE
SEQ
SEQUENCE
SEQUENTIAL
SET
SIGN
SIZE
SORT
SORT-MERGE
SOURCE
SOURCE-COMPUTER
SPACE
SPACES
SPECIAL-NAMES
STANDARD
STANDARD-1
START
STATUS
STOP
STRING
SUB-QUEUE-1
SUB-QUEUE-2
SUB-QUEUE-3
SUBTRACT
SUM
SUPPRESS
SW0
SW1
SW3
SW4
SW5
SW6
SW7
SW8
SW9
SYMBOLIC
SYNC
SYNCHRONIZED
SYSIN
SYSOUT

TABLE
TALLY
TALLYING
TAPE
TERMINAL
TERMINATE
TEXT
THAN
THEN
THROUGH
THRU
TIME
TIME-OF-DAY
TIMES
TO
TOP
TRAILING
TYPE

UN-EXCLUSIVE
UNIT
UNSTRING
UNTIL
UP
UPON
USAGE
USE
USING

VALUE
VALUES
VARYING
VOL

WHEN
WHEN-COMPILED
WITH
WORDS
WORKING-STORAGE
WRITE

ZERO
ZEROES
ZEROS
+
-
*
/
**
>
<
=

INDEX

A symbol, in PICTURE, 9-35
abbreviated combined relation condition, 10-32
ACCEPT statement, 11-1
ACCESS MODE clause, 7-34
ACCESS MODE clause, indexed files, 7-34
ACCESS MODE clause, random-access files, 7-34
ACCESS MODE clause, relative files, 7-34
ACCESS MODE clause, sequential files, 7-34
access modes, dynamic, 7-35
access modes, random, 7-35
ACTUAL KEY clause, 7-36
ACTUAL KEY clause, use in random-access files, 7-23
ACTUAL KEY data-item, updated by READ, 11-82
ADD statement, 11-8
additional (non-COBOL II) characters, 3-21
ADVANCING phrase, with LINAGE clause, 9-13, 14
ADVANCING phrase, WRITE statement, 11-129
AFTER phrase, INSPECT statement, 11-48
algebraic signs, 4-6
alignment of data, 4-7
alignment, of data-items, 9-57
ALL literal, 3-6
ALL phrase, INSPECT statement, 11-49
ALL, in DELIMITED phrase, 11-116
ALPHABET-NAME clause, 7-14
alphabet-name, 3-10
alphabet-name, in CODE-SET clause, 9-8
alphabetic data category, 4-5
alphabetic data class, 4-5
alphanumeric comparison, index data item, 9-66
alphanumeric data categories, 4-5
alphanumeric data category, 4-5
alphanumeric data class, 4-5
alphanumeric data, in PICTURE, 9-37
alphanumeric edited data category, 4-5
alphanumeric edited data in PICTURE, 9-38
ALTER statement, 11-10
ALTER statement, with GO TO statement, 11-10
alternate collating sequences, definition of, 7-14
ALTERNATE KEY clause, DUPLICATES phrase, 7-37
ALTERNATE RECORD KEY clause, 7-37
ALTERNATE RECORD KEY clause, with indexed files, 7-25
ANSI COBOL, 1-2, 1-3
apostrophes, 3-19
applications of COBOL, 1-2
areas, file buffers, 7-47
arithmetic expressions, 10-12
arithmetic expressions, valid combinations, 10-15
arithmetic operator, 3-8
arithmetic operators, 10-13
arithmetic statements, common features, 10-37
ASCENDING phrase, MERGE statement, 13-4
ASCENDING phrase, SORT statement, 13-13
ASCII character set, G-1ff
ASCII characters, storage as, 9-60, 1
ASCII collating sequence, 3-21
ASCII collating sequence, operations, 7-6
ASCII collating sequence, rearrangement of, 7-16
ASCII collating sequence, representation of, 7-15
ASCII collating sequence, use of, 11-100
ASCII, character codes in data storage, 9-8
asterisk symbol, in PICTURE, 9-41
AT END condition, RETURN statement, 13-8
AT END condition, with READ statement, 11-81
AT END phrase, RETURN statement, 13-8

B symbol, in PICTURE, 9-35
BCD (Binary Coded Decimal), 3-21
BEFORE phrase, INSPECT statement, 11-48
binary operators, 10-13
bit mask, in passing data-items, 12-10
BLANK WHEN ZERO clause, 9-27
BLOCK CONTAINS clause, 9-5
blocking factor, 9-5
blocking factor, examples, 9-6
bounds checking, A-26
bounds overflow, WRITE statement, 11-132
BOUNDS parameter, of $CONTROL, A-26
buffers, input/output for files, 7-47
buffers, sharing of with files, 7-51
BUILD command, COBEDIT, 14-4

C01,...,C12(condition codes), 7-10
C01-C12 options, with WRITE statement, 11-129
CALL statement, 12-2
CALL statement, use in transfer of control, 12-5
CALL statement, with LINKAGE SECTION, 8-7
calling intrinsics, 12-5, 7
calling non-COBOL programs, 12-5
calling non-COBOL procedures, 12-10
calling programs, 12-9
CANCEL statement, 12-12
carriage-control option, 7-33
CCTL, carriage-control option, 7-33
character set, COBOL, 3-20
character string, 3-1
character strings, PICTURE, 3-18
CHARACTERS phrase, INSPECT statement, 11-49
characters, lower and upper case, 3-21
CKERROR intrinsic, example, 7-50
CKERROR intrinsic, files, 7-41ff
class condition, 10-18
class parameter, of a device, 7-33
class, of a device, 7-33
clauses, 2-9
CLOSE statement, 11-13
CLOSE statement, general rules, 11-14
CLOSE statement, needed for successful READ, 11-81
CLOSE statement, NO REWIND phrase, 11-16
CLOSE statement, REEL/UNIT phrase, 11-15
CLOSE statement, REMOVAL phrase, 11-15
CLOSE statement, sequential files, 11-13
closing files, 11-13
COBEDIT program, 14-2
COBEDIT program, commands, 14-3
COBOL character set, 3-20
COBOL command, B-1
COBOL II coding form, 5-4, F-1
COBOL libraries, 14-1
COBOL mode, EDIT/3000, 5-1
COBOL modules, 1-4
COBOLGO command, B-1
COBOLLOCK, 12-5, I-2
COBOLPREP command, B-1

COBOLUNLOCK, 12-5, I-1
CODE parameter, of $CONTROL, A-26
Code, 3-21
CODE-SET clause, 9-7
CODE-SET clause, with SIGN CLAUSE, 9-56
coding conventions, 5-5
coding rules, 5-4
coding source programs, 1-6, 5-1
COLLATING SEQUENCE clause, 7-6
COLLATING SEQUENCE phrase, MERGE statement, 13-4
COLLATING SEQUENCE phrase, SORT statement, 13-13
collating sequence variations, 3-21
collating sequence, definition of, 7-14, 7-16
combined conditions, 10-29
comma symbol, in PICTURE, 9-40
comma, ix, 3-19
comment entries, 3-18
comment lines, 3-19
$COMMENT command, A-2
comment lines, with page eject, 3-19
common data-items, 12-3
common label item, 11-123
common phrases, 10-34
communication module, not implemented, 1-2
COMP, 9-60, 63
COMP-3, 9-60
comparison operation, INSPECT statement, 11-45ff
compiler dependent options, preprocessor commands, A-24
compiler directing statements, 10-8
compiling source program, 1-6
complex conditions, 10-27
composites of operands, 10-12, 11-111
COMPUTATIONAL, 9-60, 63
COMPUTATIONAL-3, 9-60
COMPUTATIONAL-3, sign configuration, 9-64
COMPUTE statement, 11-18
COMPUTE statement, arithmetic expressions, 10-12
computer-name, 3-14
concatenation of data-items, 11-104
condition codes, test of, 10-25
condition names, 9-71
CONDITION-CODE function, 7-11
CONDITION-CODE, 7-9
condition-name conditions, 10-26
condition-names, 3-10, 4-14
condition-names, qualification, 4-10
condition-names, value of, 9-67
conditional compilation, A-14
conditional expressions, 10-16
conditional locking, 11-33
conditional statements, 10-7
conditional variable, 3-10, 9-71
conditions, abbreviated combined relation, 10-32
conditions, class, 10-18
conditions, combined, 10-29
conditions, complex, 10-27
conditions, condition-name, 10-26
conditions, evaluation rules, 10-30
conditions, intrinsic relation, 10-25
conditions, negated simple, 10-28
conditions, relation, 10-21
conditions, sign, 10-17
conditions, simple, 10-16
conditions, switch status, 10-20
CONFIGURATION SECTION, paragraphs, 7-2
connectives, 3-3
CONSOLE, 7-9
constants, value of, 9-67
continuation line, 5-5
conventions, coding, 5-5
conventions, indentation, 5-5
conversion aids, COBOL to COBOL II, 1-5
$CONTROL command, 5-2, A-26
COPY command, COBEDIT, 14-7
copy libraries, 14-1
COPY statement, 14-24
copying, of modules in COBEDIT, 14-7
CORRESPONDING phrase, PROCEDURE DIVISION, 10-36
COUNT IN phrase, UNSTRING statement, 11-117
CR editing character, 9-46
CR symbol, in PICTURE, 9-41
cross referencing option, A-27
CROSSREF parameter, of $CONTROL, A-27
CURRENCY SIGN IS clause, 7-20
currency symbol, fixed insertion editing, 9-46
currency symbol, in PICTURE, 9-40
CURRENT-DATE, 3-5

DA, mass-storage device class parameter, 7-33
data alignment rules, 4-7
data alignment, 4-7
data categories, 4-5
data categories, alphabetic, 4-5
data categories, alphanumeric edited, 4-5
data categories, alphanumeric, 4-5
data categories, alphanumeric, 4-5
data categories, numeric edited, 4-5
data categories, numeric, 4-5
data class, alphabetic, 4-5
data class, alphanumeric, 4-5
data class, numeric, 4-5
data classes, 4-5
data description entries, 9-24
data description entries, general rules, 9-24
data description entries, RENAMES clause, 9-69
data description entries, summary, 9-22
data description entry, 4-2
data description entry, condition names, 9-71
data description entry, format 3, 9-71
DATA DIVISION, 8-1
data items, repeated, 9-29
data movement, 4-7
data names, uniqueness, 9-23
data record names, 9-9
DATA RECORDS clause, 9-9
data, alphabetic in PICTURE, 9-35
data, incompatible, 10-38
data, low volume transfer, 11-22
data, transfer of, 11-50ff
data-items, alignment, 9-57
data-items, assigning initial values, 8-6
data-items, concatenation of, 11-104
data-items, division of, 11-113
data-items, elementary, 4-3
data-items, noncontiguous, 9-23
data-items, receiving, 11-52
data-items, sort/merge file, 13-8
data-items, specification and reference, 4-5

data-items, storage, 9-60
data-items, subscripted, 4-12
data-items, superimposition of, 11-111
data-items, unrelated, 9-23
data-items, USAGE IS DISPLAY, 9-60
data-items, variable size, 9-67
data-items, Working Storage, 9-67
data-name or FILLER clause, 9-26
data-name parameters, 9-19, 20
data-name, 3-10, 4-2
data-names, identifiers, 4-14
data-names, qualification, 4-10, J-1
data-names, use of in LINAGE clause, 9-13
DATE-COMPILED paragraph, 6-2
DB editing character, 9-46
DB symbol, in PICTURE, 9-41
DEBUG ITEM, 3-4
debug module, not implemented, 1-2
debug option, (MPE), A-27
DEBUG parameter, of $CONTROL, A-27
DEBUGGING MODE CLAUSE, 7-4
decimal fields, 9-65
decimal point, in PICTURE, 9-36, 9-40
decimal point, literals, 3-15
decimal point, treatment in storage, 4-7
DECIMAL-POINT IS COMMA clause, 7-21
declarative sections, 10-3
DELETE statement, 11-19
DELETE statement, indexed files, 7-26
DELETE statements, with relative files, 7-25
DELIMITED, 11-106
DELIMITER IN phrase, 11-115
DESCENDING phrase, MERGE statement, 13-4
$DEFINE command, A-5
DESCENDING phrase, SORT statement, 13-13
device, class of, 7-33
device, file residence, 7-33
device, private volume, 7-33
direct access discs, 7-23
direct indexing, 4-12
DISPLAY statement, 11-22
DISPLAY statement, with literals, 3-16
DISPLAY, 9-60, 9-61
DIVIDE statement, 11-24
divisions, 2-2
dollar control commands, A-2
dollar sign, as default value, 7-20
dollar symbol, in PICTURE, 9-41
DUPLICATES phrase, 7-37
DUPLICATES phrase, with REWRITE
statement, 11-86
dynamic access mode, 7-35
dynamic access DELETE statement,
indexed files, 7-26
dynamic access REWRITE statement,
indexed files, 7-26
dynamic access WRITE statement,
indexed files, 7-26
dynamic access, indexed files, 7-26
dynamic access, of relative files, 7-25
DYNAMIC parameter, of $CONTROL, A-27
dynamic subprograms, 12-4
dynamic subprograms, overflowing, 12-11

EBCDIC (Extended Binary Coded
Decimal Information
Interchange), G-1
EBCDIC phrase, 7-15
EBCDIC, character codes in data storage, 9-8
EBCDIC, in SPECIAL-NAMES paragraph, 7-7
EDIT command, COBEDIT, 14-7
EDIT/3000, 5-1
EDIT/3000, call of, 14-8
edited data in PICTURE, alphanumeric, 9-38
edited data, numeric, 9-40
editing characters, VALUE clause, 9-68
editing signs, 4-6
$EDIT command, A-21
editing, allowable types, 9-43
editing, preprocessor commands, A-16
editing, rules, 9-43
editing, simple insertion, 9-44
editing, special insertion, 9-45
EDITOR, 5-1
efficiency, input/output, 7-47
elementary data items, 4-3
elementary data-item, editing requirements, 9-34
elementary data-item, general description, 9-34
elementary data-items, RENAMES clause, 9-69
elementary data-items, size, 9-42
elementary data-items, specification of, 9-26
elementary data-items, USAGE IS INDEX, 9-66
elementary moves, 11-51
END-OF-PAGE phrase, conditions for, 11-131
END-OF-PAGE phrase, WRITE statement, 11-128
ENTER statement, 11-29
entering source program, 1-6
entry points, subprograms, 12-13
ENTRY statement, 12-13
ENVIRONMENT DIVISION, 7-1
error checking, example, 7-50
error handling, USE statement, 11-122
ERRORS = parameter, of $CONTROL, A-27
EXAMINE statement, 11-31
EXCLUSIVE statement, 7-33, 11-33
EXDATE label parameter, 9-19, 20
executing source program, 1-7, B-1
EXIT command, COBEDIT, 14-12
EXIT PROGRAM statement, 12-2, 12-16
EXIT statement, 11-34
EXTEND phrase, sequential files, 11-60
extended capabilities, COBOL II, 1-5

F recording mode, 9-17
FCOPY, 9-8
FD file description, 8-4
FD level indicator, 9-4
features, COBOL, 1-2
figurative constant words, 3-6
figurative constants, VALUE clause, 9-68
file access and use, 7-23
file access, random, 7-23
file access, relative, 7-24
file access, sequential, 7-23
FILE CONTROL clause 7-27
FILE CONTROL paragraph, 7-23
file description clauses, 9-2
FILE LIMIT clause, 7-38
FILE SECTION, 8-2
File Section, ACTUAL KEY clause, 7-36
FILE STATUS clause, 7-39
file status data-item, 7-27

FILE STATUS data-item, updated by START statement, 11-102
FILE STATUS data-item, updating by READ statement, 11-80
FILE STATUS data-item, with OPEN statement, 11-62
FILE STATUS data-item, with UN-EXCLUSIVE statement, 11-114
FILE-CONTROL paragraph, 7-23
FILE-CONTROL paragraph, example, 7-49
file-name, 3-11
files, ACCESS MODE, 7-34
files, access of on labeled tapes, 9-19
files, blocking factor, 9-5
files, CLOSE statement 11-13
files, description, 8-4
files, devices, 7-33
files, example, 7-49
files, indexed, 7-25
files, input/output buffers, 7-47
files, level indicators, 8-4, 9-4
files, library, 14-1
files, location on tape, 7-51
files, locking with EXCLUSIVE statement, 11-33
files, locking, 7-33, 12-5
files, lockwords, 7-32
files, multiple on one reel, 7-54
files, NO REWIND phrase, 11-16
files, ORGANIZATION clause
files, permissible statements, 11-61
files, positioning, 11-100
files, proper condition for READ, 11-80
files, recording mode, 7-33
files, relative, 7-24
files, SEEK statement, 11-97
files, sharing of memory, 7-51
files, size of records, 9-16
files, size parameter, 7-33
files, sort-merge, 7-27
files, specification of record length, 9-17
files, specifying location on multi-file tape, 7-51
files, status keys, 7-39ff
files, transfer of control on opening with USE, 11-123
files, user labels, 11-123
file size, 7-33
filler character, slack byte, 9-59
FILLER, 9-26
FIRST phrase, INSPECT statement, 11-49
fixed insertion editing, 9-46
floating insertion editing, 9-47
FLOCK intrinsic, 11-33
FOOTING AREA, 9-12
FOOTING phrase, 9-12
forms-message, 7-33
FREE phrase, 11-2
free-field, 11-7
FROM phrase, REWRITE statement, 11-87
FROM phrase, WRITE statement, 11-128
FUNCTION-NAME clause, 7-9
function-names, control of line printer, 7-11
FUNLOCK intrinsic, reflection in FILE STATUS data-item, 11-114
FWRITE intrinsic, 11-129

GIVING phrase, 12-5, 12-10
GIVING phrase, SORT statement, 13-15
GIVING phrase, with SUBTRACT statement, 11-111
GIVING phrase, automatic writing of merged records, 13-5
GIVING phrase, with MERGE statement, 13-5
GO TO statement, 11-38
GO TO statement, with ALTER statement, 11-10
GOBACK statement, 12-2, 12-17
group items, 4-3
group level entries, VALUE clause, 9-68
groups, 4-3

header entries, divisions, 2-3
header entries, paragraphs, 2-7
header entries, sections, 2-6
HELP commmand, COBEDIT, 14-14
hierarchy, of data-items in record structure, 4-3
hierarchy, of operations, 10-13
hierarchy, structural, 2-2, 2-4
HIGH VALUE(S), 3-6
hyphen, use of, 5-5

I-O CONTROL paragraph, 7-51
I-O mode, READ statement, 11-80
identification code, 5-5
IDENTIFICATION DIVISION, 6-1
identifiers, 4-14
IF statement, 11-40
IF/ELSE pairing, diagram, 11-41
IMAGE/3000, 1-2
imperative statements, 10-8
INC parameter, of $EDIT, A-21
index data-item, 9-66
index data-item, prohibited from MOVE statement, 11-50
index data-items, comparison, 10-23
index names, between programs, 12-10
$IF command, A-14
index referencing, 4-13
index-name, 3-11
index-names, comparison, 10-23
index-names, in LINKAGE SECTION, 8-7
index-names, in OCCURS clause, 9-31
$INCLUDE command, A-16
index-names, passing between programs, 8-7
INDEXED BY phrase, 9-31
INDEXED BY phrase, with SEARCH statement, 11-92
indexed files, 7-25
indexed files, ALTERNATE RECORD KEY clause, 7-36
indexed files, CLOSE statement, 11-16
indexed files, DUPLICATES phrase
indexed files, dynamic access, 7-26
indexed files, example, 7-49
indexed files, INVALID KEY conditions, 11-134
indexed files, ORGANIZATION clause, 7-45
indexed files, positioning, 11-100
indexed files, prime record key, 7-46
indexed files, RECORD KEY clause, 7-46
indexed files, record length, 9-6
indexed files, restrictions with REWRITE statement, 11-86
indexed files, REWRITE statement, 11-87
indexed files, sequential access, 7-26
indexes, 4-12
indexing of identifiers, UNSTRING statement, 11-118

indexing, direct, 4-12
indexing, relative, 4-12
indices, application of, 9-29
initial values, assignment of, 8-6
INPUT ERROR phrase, 11-2
INPUT mode, READ statement, 11-80
INPUT PROCEDURE phrase, SORT statement, 13-14
INPUT-OUTPUT section, 7-22
input/output buffers, files, 7-47
input/output operations, FILE STATUS clause, 7-39
input/output, efficiency, 7-47
INSPECT statement, 11-42
INSPECT statement, overlapping operands, 10-38
internal storage of data-items, 9-60
interprogram communication, 12-1
INTO phrase, READ statement, 11-80
INTO phrase, RETURN statement, 13-8
intrinsic relation condition, 10-25
INTRINSIC, 12-5
intrinsics, calling, 12-5, 7
intrinsics, CKERROR, 7-41ff
intrinsics, FLOCK, 11-33
intrinsics, FUNLOCK, 11-114
intrinsics, FWRITE, 11-129
intrinsics, LOADPROC, 12-11
intrinsics, PRINTFILEINFO, 7-41ff
INVALID KEY condition, relative files, 11-133
INVALID KEY phrase, restrictions with REWRITE statement, 11-86
INVALID KEY phrase, WRITE statement, 11-127

JUST (JUSTIFIED), 9-28
JUSTIFIED clause, 9-28

KEEP command, COBEDIT, 14-15
KEY IS phrase, 9-31
key words, 3-3
key, prime record, 7-25
KSAM files, COBOL libraries, 14-1
KSAM/3000, 1-2

L, locking parameter, 7-33
label info fields, 9-19, 20
label info parameters, 9-19, 20
LABEL RECORD clause, 9-10
LABEL RECORDS clause, file description entry, 11-60
LABELS label parameter, 9-19, 20
labels, tapes, 9-19
language-name, 3-14
LEADING phrase, INSPECT statement, 11-48
level entry 66, RENAMES clause, 9-69
level entry 66, restrictions, 9-70
level indicators, FD, 9-4
level indicators, FILE SECTION, 8-4
level indicators, SD, 9-4
level number 66, 9-23
level number 77, 9-23
level number 88, 9-23
level number 88, with condition names, 9-71
level numbers, in data description entries, 9-22
level numbers, special, 4-4
level-number, 3-11, 4-2
level-numbers, in record description, 8-4
libraries, COBOL, 14-1
LIBRARY command, COBEDIT, 14-17
library-name, 3-11

LINAGE clause, 9-11
LINAGE clause, with END-OF-PAGE phrase, 11-129
LINAGE COUNTER, 3-4
LINAGE-COUNTER, 9-14
line printer, control of with function-names, 7-11
LINE-COUNTER, 3-4
line printer, options with WRITE statement, 11-129
LINES AT BOTTOM phrase, 9-12
LINES AT TOP phrase, 9-12
LINES parameter, of $CONTROL, A-27
LINKAGE SECTION, 8-7, 12-3
LIST command, COBEDIT, 14-19
LIST parameter, $CONTROL command, A-27
LITERAL phrase, 7-16
literals, 3-15
literals, non-numeric, 3-16
literals, numeric, 3-15
literals, octal numeric, 3-15
literals, VALUE clauses, 9-68
LOADPROC intrinsic, 12-11
$LIST command, (see $CONTROL LIST)
LOCK phrase, 11-16
locking files, 12-5
locking files, EXCLUSIVE statement, 11-33
LOCKING parameter, $CONTROL command, A-28
locking, conditional 11-33
locking, of files, 7-33
locking, with CLOSE statement, 11-16
LOCON parameter, of $CONTROL, A-28
logical operators, complex conditions, 10-27
logical page, format description, 9-11
logical page, illustration, 9-15
logical positioning in files, 11-100
logical records, of sort/merge file, 13-8
logical records, specification of size, 9-17
logical records, WRITE statement, 11-125
logical removal, records, 11-19
LOW VALUE(S), 3-6
lowercase characters, 3-21

macros, A-5
main memory, conserving space, 7-52
MAP parameter, of $CONTROL, A-28
memory sharing, REDEFINES clause, 9-53
memory, CANCEL statement, 12-12
memory, conserving space, 7-52
memory, packed decimal fields, 9-65
MEMORY-SIZE Clause, 7-5
merge facility, 7-27
MERGE statement, 13-2
MERGE statement, GIVING phrase, 13-5
MERGE statement, OUTPUT PROCEDURE phrase, 13-5
MERGE statement, segmentation considerations, 13-5
merged records, ASCENDING phrase, 13-4
merged records, DESCENDING phrase, 13-4
merging files, with preprocessor commands, A-18
merging, preprocessor commands, A-16
minus symbol, in PICTURE, 9-41
MIXED parameter, of $CONTROL, A-28
mnemonic names in SPECIAL-NAMES paragraph, 7-7
mnemonics, class parameter, 7-33
mnemonic-name, 3-11
modules, COBOL, 1-4
modules, copying of with COBEDIT, 14-7
monetary symbols, 7-20

MOVE statement, 11-50
MOVE statement, example, 11-54
MOVE statement, overlapping operands, 10-38
MOVE statement, with READ statement, 11-80
moves, elementary, 11-51
moves, permissible, 11-53
moving data, rules, 11-51
MPE :EDITOR command, 5-1
multiblock option, 9-17
MULTIPLE FILE clause, 7-54
MULTIPLE REEL phrase, 7-47
multiple reel/unit files, WRITE statement, 11-132
MULTIPLY statement, 11-56
MULTIRECORD option, 9-17

NATIVE phrase, 7-15
negated simple conditions, 10-28
nested PERFORM statements, 11-65
NO REWIND phrase, CLOSE statement, 11-16
NO REWIND phrase, sequential files, 11-60
non-dynamic subprograms, 12-4
non-numeric literals, 3-16
noncontiguous data-items, 9-23
nonnumeric comparison, example, 10-24
nonnumeric operands, comparison, 10-23
NOSEQ parameter, of $EDIT, A-21
NOSPACE CONTROL, 7-9
number of times option, PERFORM statement, 11-78
numeric data categories, 4-5
numeric data class, 4-5
numeric data in PICTURE, 9-36
numeric data, SIGN clause, 9-56
numeric data, storage, 9-61
numeric data, USAGE IS COMPUTATIONAL, 9-63
numeric edited data category, 4-5
numeric edited data, in PICTURE, 9-40
numeric literals, 3-15
numeric literals, octal, 3-15
numeric literals, permitted range in VALUE
clause, 9-37
numeric literals, VALUE clause, 9-68
numeric operands, comparison, 10-22

object code listing, A-26
object program, 12-1
OBJECT-COMPUTER paragraph, 7-5
occurence number, in table, 9-31
OCCURS clause, 9-29
OCCURS clause, general rules, 9-30
OCCURS clause, restriction, 9-67
OCCURS clause, with REDEFINES clause, 9-53
octal literals, with PARM= parameter, 7-11
octal numeric literals, 3-15
ON INPUT ERROR phrase, 11-2
on-line creation of program, 5-1
OPEN statement, 11-58
OPEN statement, needed for sucessful READ, 11-81
operand combinations, SET statement, 11-99
operands, overlapping, 10-38
operational signs, 4-6
operator, arithmetic, 3-8
operator, relational, 3-8
OPTIONAL phrase, 7-31
optional words, 3-3
ORGANIZATION clause, 7-45
organization, ANS COBOL, 1-4
output files, carriage-control option, 7-33
OUTPUT PROCEDURE phrase,
MERGE statement, 13-5
OUTPUT PROCEDURE phrase,
SORT statement, 13-15
overflow conditions, loading subprograms, 12-11
overflow conditions, UNSTRING statement, 11-118

P symbol, in PICTURE, 9-36
packed decimal fields, illustration, 9-65
packed decimal format storage, 9-60
page eject in comment lines, 3-19
PAGE-COUNTER, 3-4
paragraph-name, 3-11
paragraphs, 2-7
parentheses, 3-19, 10-13
PARM parameter, RUN command, B-4
PARM= parameter, with software switches, 7-11
percent sign, use of, 3-15
PERFORM statement, 11-63
PERFORM statement, illustration, 11-66ff
PERFORM statement, nested, 11-65
PERFORM statement, number of times option, 11-78
PERFORM statement, rules, 11-64
PERFORM statement, using UNTIL option, 11-78
PERFORM statement, VARYING option, 11-78
period symbol, in PICTURE, 9-40
period, ix
period, use in editing, 9-45
period, use in PICTURE, 9-36
permanent segments, 7-6
permissible data moves, 11-53
permissible statements, files, 11-61
phrases, 2-9
PICTURE character precedence chart, 9-52
PICTURE character strings, 3-18
PICTURE clause, 3-18, 9-34
PICTURE, alphabetic data, 9-35
PICTURE, alphanumeric data, 9-37
PICTURE, alphanumeric edited data, 9-38
PICTURE, numeric data, 9-36
PICTURE, numeric edited data, 9-40
plus symbol, in PICTURE, 9-41
POINTER phrase, STRING statement, 11-107
POINTER phrase, UNSTRING statement, 11-116
positioning of lines, vertical, 11-128
precedence chart, PICTURE characters, 9-52
precedence rules, editing of character strings in
PICTURE clause, 9-51
preparing source program for execution, 1-7, B-1
preprocessor commands, A-2
preprocessor programming language, A-2
prime record key, DUPLICATES phrase, 7-46
prime record key, indexed files, 7-25
PRINTFILEINFO intrinsic, with files, 7-40
private volume device type, 7-33
PROCEDURE DIVISION, 10-1
PROCEDURE DIVISION, general format, 10-3
PROCEDURE DIVISION, SORT statements, 13-11
$PREPROCESSOR command, A-14
procedures, 10-5
PROCESSING MODE clause, 7-48
PROGRAM COLLATING SEQUENCE clause, 7-6
program structure, 2-5
PROGRAM-ID paragraph, 6-2
program-name, 3-11

programs, calling, 12-9
programs, non-COBOL, 12-5
pseudo-text, 14-24
pseudo-text, 3-19
punch cards, signed decimal fields, 9-62
punctuation characters, 3-19
PURGE command, COBEDIT, 14-21

qualification, example, 4-11
qualifiers, 4-8, J-1
QUERY/3000, 1-2
quotation marks, 3-15, 3-16, 17
QUOTE parameter, of $CONTROL, A-29
QUOTE(S), 3-6

random access files, PROCESSING MODE
clause, 7-48
random access mode, 7-35
random files, CLOSE statement, 11-16
random files, record length, 9-6
random-access files, 7-23
random-access files, ACTUAL KEY clause, 7-36
random-access files, ASSIGN clause, 7-31
random-access files, FILE LIMIT clause, 7-38
random-access files, PROCESSING MODE
clause, 7-48
random-access files, WRITE statement, 11-132
random-access input/output statements, 7-25
READ statement, 11-79
READ statement, common rules, 11-80
READ statement, indexed files, 7-26
READ statement, proper condition of files, 11-80
READ statements, with relative files, 7-25
receiving data-item, 4-7
receiving items, alphabetic, 11-52
receiving items, alphanumeric or alphanumeric
edited, 11-52
receiving items, numeric or numeric edited, 11-52
RECORD AREA clause, with REWRITE, 11-87
RECORD CONTAINS clause, 9-16
record description entries, discussion, 9-23
record description entry, 4-4
record descriptions, level-numbers, 8-4
RECORD KEY clause, 7-46
RECORD KEY clause, with indexed files, 7-25
record labels, file description entry, 11-60
record length, random files, 9-6
record length, relative files, 9-6
record length, sequential or indexed file, 9-6
record-name, 3-12
RECORDING MODE clause, 9-17
recording mode, files, 7-33
records, deletion of, 11-19
records, determination of size, 9-16
records, labels, 9-10
records, merged, automatic writing of, 13-5
records, names of, 9-9
records, specification of size, 9-16, 17
REDEFINES clause, 9-53
REDEFINES clause, restriction, 9-67
REDEFINES clause, with SYNCHRONIZED
clause, 9-59
redefinitions, of character positions, 9-53
REEL/UNIT phrase, CLOSE statement, 11-15
reference, uniqueness of, 4-8
referencing of data, 4-2
referencing, of indexes, 4-13
relation conditions, 10-21
relational operator, 3-8
relative files, 7-24
relative files, CLOSE statement, 11-16
relative files, INVALID KEY condition, 11-133
relative files, ORGANIZATION clause, 7-45
relative files, positioning, 11-100
relative files, record length, 9-6
relative indexing, 4-12
RELATIVE KEY clause, relative files, 7-24
RELATIVE KEY data-item, updated by READ, 11-82
RELEASE statement, with sort operation, 13-6
remarks, 3-18
REMOVAL phrase, CLOSE statement, 11-15
RENAMES clause, 9-69
repeated data items, 9-29
repetition factors, 9-35, 9-42
replacement, of zeroes with blanks, 9-27
REPLACING phrase, 14-25
REPLACING phrase, EXAMINE statement, 11-32
report writer module, not implemented, 1-2
report-name, 3-9, 3-12
RESERVE clause, 7-47
RESERVE clause, input/output buffers, 7-47
reserved words, 3-2, K-1ff
resuming execution of program, 11-103
RETURN statement, 13-7
REVERSE phrase, sequential files, 11-60
REWRITE statement, 11-85
REWRITE statement, indexed files, 7-26
REWRITE statement, sequential, relative,
indexed files, 11-85
REWRITE statements, with relative files, 7-25
ROUNDED phrase, PROCEDURE DIVISION, 10-34
routine-name, 3-9, 3-12
rules, data alignment, 4-7
RUN command, B-4
run unit, 12-1

S recording mode, 9-17
S symbol, in PICTURE, 9-36
SAME clause, 7-52
SAME RECORD AREA clause, 7-52
SAME SORT AREA clause, 7-53
SAME SORT-MERGE AREA clause, 7-53
SD file description, 8-4
SD level indicator, 9-4
SEARCH statement, 11-90
SEARCH statement, diagram, 11-93
SEARCH statement, example, 11-95
SEARCH statement, index data-item, 9-66
secondary entry points, 12-13
section-name, 3-12
sections, 2-6, 10-5
security restrictions, 6-1
SEEK statement, 11-97
SEEK statement, improved performance through
READ, 11-97
segment number, 2-7, 3-12, 10-5
SEGMENT-LIMIT clause, 7-6
segmentation considerations, ALTER statement, 11-10
segmentation considerations, MERGE statement, 13-5
segmentation, of object program, 10-3, 5
segments, 2-6
segments, permanent, 7-6

segments, states of, 10-6
SELECT clause, 7-31
SELECT clause, all other files, 7-31
SELECT clause, for sort-merge files, 7-30
SELECT clause, sequential files, 7-31
SELECT statement, OPTIONAL phrase, 7-31
semicolons, ix, 3-19
sending data-item, 4-7
sentences, 2-9
sentences, PROCEDURE DIVISION, 10-7
SEPARATE CHARACTER phrase, 9-42
separation of data-items, 11-113
separators, 3-1, 3-19
SEQ label parameter, 9-19, 20
SEQNUM parameter, of $EDIT, A-21
sequence field checking, A-20
sequence number, 5-4ff
sequential access DELETE statement, indexed files, 7-26
sequential access READ statement, indexed files, 7-26
sequential access REWRITE statement, indexed files, 7-26
sequential access WRITE statement, indexed files, 7-26
sequential access, indexed files, 7-26
sequential access, relative files, 7-24
sequential DELETE statements, with relative files, 7-25
sequential file access, 7-23
sequential files, example, 7-49
sequential files, EXTEND, REVERSE, NO REWIND phrases, 11-60
sequential files, in CLOSE statement, 11-13
sequential files, logical page, 9-11
sequential files, multiple on one reel, 7-54
sequential files, OPTIONAL phrase, 7-31
sequential files, ORGANIZATION clause, 7-45
sequential files, record length, 9-6
sequential files, RESERVE clause, 7-47
sequential files, status-key settings, 7-40ff
sequential files, vertical positioning of lines, 11-128
sequential READ statements, with relative files, 7-25
sequential REWRITE statements, with relative files, 7-25
sequential WRITE statements, with relative files, 7-24
serial search, 11-92
SET statement, 11-98
SET statement, index data-item, 9-66
SET statement, overlapping operands, 10-38
SET statement, use in passing index-names between programs, 8-7
sharing of memory, files, 7-51
SHOW command, COBEDIT, 14-23
SIGN clause, 9-56
sign condition, 10-17
sign configuration, COMPUTATIONAL-3, 9-64
$SET command, A-14
sign control symbols, 9-46
SIGN IS SEPARATE phrase, 9-42, 9-56
sign, of sending data item, 9-46
signed data-items, 9-61
signed decimal fields, 9-62
signed numeric data-item, 9-63
signed numeric literal, VALUE clause, 9-68
signs, algebraic, 4-6
signs, editing, 4-6
signs, operational, 4-6
simple condition, 10-16
simple insertion characters, in floating insertion editing, 9-47
simple insertion editing, 9-44
SIZE ERROR phrase, PROCEDURE DIVISION, 10-34
size, elementary data-items, 9-42
slack bytes, 9-59
sort facility, 7-27
sort operations, RELEASE statement, 13-6
SORT statement, 13-9
SORT statement, ASCENDING phrase, 13-13
SORT statement, COLLATING SEQUENCE phrase, 13-13
SORT statement, DESCENDING phrase, 13-13
SORT statement, GIVING phrase, 13-15
SORT statement, INPUT PROCEDURE phrase, 13-14
SORT statement, OUTPUT PROCEDURE phrase, 13-15
SORT statement, USING phrase, 13-14
SORT statement, with RELEASE statement, 13-6
SORT statements, in PROCEDURE DIVISION, 13-11
sort-merge files, 7-27
sort-merge operations, 13-1
SORT-MERGE/3000, 1-2
sort/merge file, logical records of, 13-8
sorting, order, 13-11
SOURCE parameter, of $CONTROL, A-29
source program, 12-1
source program, coding, 1-6
source program, compiling, 1-6, B-3
source program, entering, 1-6
source program, preparing for execution, 1-7, B-3
SOURCE-COMPUTER paragraph, 7-4
SPACE(S), 3-6
spaces, 3-6, 3-19
special character words, 3-8
special forms request message, 7-33
special insertion editing, 9-45
special level numbers, 4-4
SPECIAL NAMES paragraph, condition codes, 10-25
SPECIAL NAMES paragraph, with COLLATING SEQUENCE, 7-6
special register words, 3-4
SPECIAL-NAMES paragraph, 7-7
standard COBOL, 1-3
STANDARD-1 phrase, 7-15
START statement, 11-100
START statement, relative files, 7-24
statements, categorized, 10-10
statements, PROCEDURE DIVISION, 10-7
statements, 2-9
status keys, 7-39ff
STDWARN parameter, of $CONTROL, A-29
STOP RUN statement, 11-103
STOP statement, 11-103
storage areas, redefinition, 9-53
string search and replacement, 11-42
STRING statement, 11-104
STRING statement, example, 11-107
STRING statement, execution of, 11-107
STRING statement, overlapping operands, 10-38
stroke editing symbol, in PICTURE, 9-38
subprogram compilation, A-27
SUBPROGRAM parameter, of $CONTROL, A-29
subprograms, dynamic, 12-3

subprograms, entry points, 12-13
subprograms, nondynamic, 12-3
subprograms, release of memory, 12-12
subscripted data-items, 4-12
subscripting of identifiers,
UNSTRING statement, 11-118
subscripts, 4-11
subscripts, application of, 9-29
SUBTRACT statement, 11-110
superimposition of data-items, 11-111
suspending of object program, 11-103
SW0,...,SW9(software switches), 7-11
switch-status condition, 10-20
switches, software, 7-11
symbol table map option, of $CONTROL, A-28
SYNC, equivalent to SYNCHRONIZED, 9-57
SYNCHRONIZED clause, 9-57
SYNCHRONIZED clause, slack bytes, 9-59
SYSIN, 7-9
SYSOUT, 7-9
system names, 3-14

table subscripts, 4-11
tables, 10-39
tables, assigning initial values, 8-6
tables, definition of, 10-39
tables, fixed length, 9-31
tables, one to three dimensional, 9-29
tables, referencing by index, 10-41
tables, referencing by subscript, 10-40
tables, SEARCH statement, 11-90
tables, SET statement, 11-99
tables, storage of index data-items, 9-66
tables, variable length, 9-32
TALLY, 3-5
TALLYING phrase, EXAMINE statement, 11-32
TALLYING phrase, UNSTRING statement, 11-116
tape labels, 9-19
terminating period, 2-3
terminators, divisions, 2-3
terminators, paragraphs, 2-7
terminators, sections, 2-6
text-name, 3-12
text-names, qualification, 4-10
TIME-OF-DAY, 3-5
$TITLE command, A-25
TOP, 7-9
twos complement binary integer format storage, 9-60

U recording mode, 9-17
UN-EXCLUSIVE statement, 7-33, 11-113
unary operators, 10-13
underscore, ix
uniqueness of reference, 4-8
unlocking files, UN-EXCLUSIVE statement, 11-113
unlocking, of files, 7-33

unrelated data items, 9-23
UNSTRING statement, 11-114
UNSTRING statement, example, 11-119
UNSTRING statement, execution of, 11-117
UNSTRING statement, overlapping operands, 10-38
UNTIL option, PERFORM statement, 11-78
uppercase characters, 3-21
UR, unit-record device class parameter, 7-33
USAGE clause, 9-60
USAGE IS COMPUTATIONAL, 9-63
USAGE IS COMPUTATIONAL,
SYNCHRONIZED clause, 9-57
USAGE IS COMPUTATIONAL-3, 9-64
USAGE IS DISPLAY, 9-60, 9-61
USAGE IS DISPLAY, SIGN clause, 9-56
USAGE IS INDEX, 9-66
USAGE IS INDEX, SYNCHRONIZED clause, 9-57
USE procedure, needed for file being unlocked, 11-114
USE procedure, needed with READ statement, 11-81
USE statement, 11-121
USE statement, error handling procedures, 11-122
USE statement, general rules, 11-122
USE statement, OPTIONAL phrase, 7-31
USE statement, user labels on files, 11-123
user labels, USE statement, 11-123
user-defined words, 3-9
USING clause, 10-2
USING clause, 3-3
USING clauses, in interprogram communication, 12-3
using COBOL II/3000, 1-5
USING phrase, between programs, 12-9
USING phrase, CALL statement, index data-item, 9-66
USING phrase, SORT statement, 13-14
USL, User Subprogram Library, A-29
USLINIT parameter, of $CONTROL, A-29
UT, utility device class parameter, 7-33

V recording mode, 9-17
V symbol, in PICTURE, 9-36
VALUE clause, 9-67
VALUE clause, literals, 9-68
VALUE clause, restrictions with
LINKAGE SECTION, 8-7
VALUE clause, restrictions, 9-67
VALUE OF clause, 9-19
VALUE OF clause, example, 9-21
variation of single identifier,
PERFORM statement, 11-71
variation of three identifiers,
PERFORM statement, 11-75
variation of two identifiers,
PERFORM statement, 11-73
variations between collating sequences, 3-21
VARYING option, PERFORM statement, 11-78
VARYING phrase, SEARCH statement, 11-92
VERBS parameter, of $CONTROL, A-30
verbs, categorized, 10-10
verbs, imperative, 10-9
VFU(Vertical Form Unit), 11-129
V/3000, 1-2
VOID parameter, of $EDIT, A-21
VOL label parameter, 9-19, 20

WARN parameter, of $CONTROL, A-30
WHEN clause, SEARCH statement, 11-94
WHEN-COMPILED, 3-5
WITH LOCK phrase, CLOSE statement, 11-16
word boundaries, data-items, 9-59
words, 3-2
words, figurative constant, 3-6
words, key, 3-3
words, optional, 3-3
words, reserved, 3-2, K-1ff
words, special character, 3-8
words, special register, 3-4

words, user-defined, 3-9
WORKING STORAGE SECTION, 8-6
Working Storage Section, ACTUAL KEY clause, 7-36
Working Storage, data-items, 9-67
WRITE statement, 11-125
WRITE statement, indexed files, 7-26
WRITE statement, invalid address, 11-132
WRITE statements, with relative files, 7-24

X symbol, in PICTURE, 9-37

Z symbol, editing data, 9-49
Z symbol, in PICTURE, 9-40
zero editing symbol, in PICTURE, 9-38
zero suppression and replacement, 9-47, 9-49
zero suppression editing, 9-49
ZERO(S), 3-6
ZEROES, 3-6
zone-signed fields, 9-62

* symbol, in PICTURE, 9-41
+ symbol, in PICTURE, 9-41
, symbol, in PICTURE, 9-40
− symbol, in PICTURE, 9-41
. editing symbol, in PICTURE, 9-40
/ editing symbol, in PICTURE, 9-38
0 editing symbol, in PICTURE, 9-38
9 symbol, in PICTURE, 9-36
$ symbol, in PICTURE, 9-41

subprograms, entry points, 12-13
subprograms, nondynamic, 12-3
subprograms, release of memory, 12-12
subscripted data-items, 4-12
subscripting of identifiers,
UNSTRING statement, 11-118
subscripts, 4-11
subscripts, application of, 9-29
SUBTRACT statement, 11-110
superimposition of data-items, 11-111
suspending of object program, 11-103
SW0,...,SW9(software switches), 7-11
switch-status condition, 10-20
switches, software, 7-11
symbol table map option, of $CONTROL, A-28
SYNC, equivalent to SYNCHRONIZED, 9-57
SYNCHRONIZED clause, 9-57
SYNCHRONIZED clause, slack bytes, 9-59
SYSIN, 7-9
SYSOUT, 7-9
system names, 3-14

table subscripts, 4-11
tables, 10-39
tables, assigning initial values, 8-6
tables, definition of, 10-39
tables, fixed length, 9-31
tables, one to three dimensional, 9-29
tables, referencing by index, 10-41
tables, referencing by subscript, 10-40
tables, SEARCH statement, 11-90
tables, SET statement, 11-99
tables, storage of index data-items, 9-66
tables, variable length, 9-32
TALLY, 3-5
TALLYING phrase, EXAMINE statement, 11-32
TALLYING phrase, UNSTRING statement, 11-116
tape labels, 9-19
terminating period, 2-3
terminators, divisions, 2-3
terminators, paragraphs, 2-7
terminators, sections, 2-6
text-name, 3-12
text-names, qualification, 4-10
TIME-OF-DAY, 3-5
$TITLE command, A-25
TOP, 7-9
twos complement binary integer format storage, 9-60

U recording mode, 9-17
UN-EXCLUSIVE statement, 7-33, 11-114
unary operators, 10-13
underscore, ix
uniqueness of reference, 4-8
unlocking files, UN-EXCLUSIVE statement, 11-114
unlocking, of files, 7-33

unrelated data items, 9-23
UNSTRING statement, 11-113
UNSTRING statement, example, 11-119
UNSTRING statement, execution of, 11-117
UNSTRING statement, overlapping operands, 10-38
UNTIL option, PERFORM statement, 11-78
uppercase characters, 3-21
UR, unit-record device class parameter, 7-33
USAGE clause, 9-60
USAGE IS COMPUTATIONAL, 9-63
USAGE IS COMPUTATIONAL,
SYNCHRONIZED clause, 9-57
USAGE IS COMPUTATIONAL-3, 9-64
USAGE IS DISPLAY, 9-60, 9-61
USAGE IS DISPLAY, SIGN clause, 9-56
USAGE IS INDEX, 9-66
USAGE IS INDEX, SYNCHRONIZED clause, 9-57
USE procedure, needed for file being unlocked, 11-114
USE procedure, needed with READ statement, 11-81
USE statement, 11-121
USE statement, error handling procedures, 11-122
USE statement, general rules, 11-122
USE statement, OPTIONAL phrase, 7-31
USE statement, user labels on files, 11-123
user labels, USE statement, 11-123
user-defined words, 3-9
USING clause, 10-2
USING clause, 3-3
USING clauses, in interprogram communication, 12-3
using COBOL II/3000, 1-5
USING phrase, between programs, 12-9
USING phrase, CALL statement, index data-item, 9-66
USING phrase, SORT statement, 13-14
USL, User Subprogram Library, A-29
USLINIT parameter, of $CONTROL, A-29
UT, utility device class parameter, 7-33

V recording mode, 9-17
V symbol, in PICTURE, 9-36
VALUE clause, 9-67
VALUE clause, literals, 9-68
VALUE clause, restrictions with
LINKAGE SECTION, 8-7
VALUE clause, restrictions, 9-67
VALUE OF clause, 9-19
VALUE OF clause, example, 9-21
variation of single identifier,
PERFORM statement, 11-71
variation of three identifiers,
PERFORM statement, 11-75
variation of two identifiers,
PERFORM statement, 11-73
variations between collating sequences, 3-21
VARYING option, PERFORM statement, 11-78
VARYING phrase, SEARCH statement, 11-92
VERBS parameter, of $CONTROL, A-30
verbs, categorized, 10-10
verbs, imperative, 10-9
VFU(Vertical Form Unit), 11-129
VIEW/3000, 1-2
VOID parameter, of $EDIT, A-21
VOL label parameter, 9-19, 20

WARN parameter, of $CONTROL, A-30
WHEN clause, SEARCH statement, 11-94
WHEN-COMPILED, 3-5
WITH LOCK phrase, CLOSE statement, 11-16
word boundaries, data-items, 9-59
words, 3-2
words, figurative constant, 3-6
words, key, 3-3
words, optional, 3-3
words, reserved, 3-2, K-1ff
words, special character, 3-8
words, special register, 3-4

words, user-defined, 3-9
WORKING STORAGE SECTION, 8-6
Working Storage Section, ACTUAL KEY clause, 7-36
Working Storage, data-items, 9-67
WRITE statement, 11-125
WRITE statement, indexed files, 7-26
WRITE statement, invalid address, 11-132
WRITE statements, with relative files, 7-24

X symbol, in PICTURE, 9-37

Z symbol, editing data, 9-49
Z symbol, in PICTURE, 9-40
zero editing symbol, in PICTURE, 9-38
zero suppression and replacement, 9-47, 9-49
zero suppression editing, 9-49
ZERO(S), 3-6
ZEROES, 3-6
zone-signed fields, 9-62

* symbol, in PICTURE, 9-41
+ symbol, in PICTURE, 9-41
, symbol, in PICTURE, 9-40
– symbol, in PICTURE, 9-41
. editing symbol, in PICTURE, 9-40
/ editing symbol, in PICTURE, 9-38
0 editing symbol, in PICTURE, 9-38
9 symbol, in PICTURE, 9-36
$ symbol, in PICTURE, 9-41

Part No. 32233-90001
Printed in U.S.A. 12/79
3COBOLA.320.32233-90001

Sales and service from 172 offices in 65 countries.
5303 Stevens Creek Blvd., Santa Clara, California 95050