Data General

COBOL

Reference Manual

093-000180-02

COBOL

Reference Manual

093-000180-02

For the latest enhancements, cautions, documentation changes, and other information on this product, please see the Release Notice (085-series) supplied with the software.

Ordering No. 093-000180

Printed in the United States of America
Revision 02, August 1978

NOTICE

COBOL Reference Manual
093-000180

Revision History:

093-000180

Original Release - August 1975
First Revision - August 1976
Second Revision - August 1978

This document has been extensively revised from revision 01; therefore, change indicators have not been used.

Preface

We have written this manual for experienced COBOL programmers (or those with experience in a COBOL-like language) who need familiarity with Data General's ECLIPSE COBOL programming language. The manual describes all aspects of ECLIPSE COBOL and shows you how to code, compile, debug, and execute your COBOL programs. ECLIPSE COBOL runs under Data General's Real-Time Disk Operating System (RDOS) and Advanced Operating System (AOS). We present the information in this manual in a system independent manner. Wherever operating system differences occur in the language, we either refer you to Appendix B, "Language Differences Between RDOS and AOS," or we include a table to point out these differences. The text itself is always system independent.

We have organized this manual as follows:

Chapter 1 discusses the ANSI Standard conventions we have implemented (ANSI X3.23-1974), the general structure of a COBOL program, language extensions to the ANSI Standard, and special facilities provided by Data General's COBOL.

Chapter 2 describes COBOL language concepts and defines COBOL terms.

Chapter 3 describes the Identification Division.

Chapter 4 describes the Environment Division, presenting details of its two sections (the Configuration Section and the Input-Output Section), and discusses the organization, access, and declaration of COBOL files.

Chapter 5 describes the Data Division and discusses COBOL data types, telling you how to declare, define, and edit data in your program.

Chapter 6 describes the Procedure Division and discusses its features including procedure statement clauses, expressions, subprogramming, file formatting, indexed file record selection, and I/O exception conditions.

Chapter 7 gives a detailed description of all Procedure Division statements, arranged in alphabetic order.

Chapter 8 describes the COBOL COPY statement which allows you, at compile time, to include source code from another file as part of your program.

Chapter 9 tells you how to compile, bind, and execute your COBOL program under RDOS.

Chapter 10 tells you how to compile, bind, and execute your COBOL program under AOS.

Chapter 11 describes the COBOL source-level debugger and its commands.

Appendix A contains a list of COBOL reserved words.

Appendix B describes the syntactical differences between RDOS and AOS language features.

Appendix C tells you how to write COBOL-callable assembly language routines for both RDOS and AOS.

Appendix D lists compiler error and warning messages.

Appendix E contains a table of the ASCII character set.

Appendix F contains a table of the EBCDIC character set.

Appendix G describes the ECLIPSE Communications Access Manager (CAM).

Appendix H tells you how to create and read unlabeled magnetic tape files using COBOL in AOS.

Required Manuals

You should supplement your reading of this manual with the following: *Introduction to COBOL* (69-000017), *INFOS Storybook* (69-000019), *RDOS Reference Manual* (93-000075), *RDOS/INFOS System User's Manual* (93-000114), *Communications Access Manager Reference Manual,* (93-000183), *ECLIPSE-line Sort and Merge Utility Programs User's Manual* (93-000126), *AOS Programmer's Manual* (93-000120), *AOS/INFOS System Manual* (93-000152), and *AOS Sort/Merge User's Manual* (93-000155).

Reader Please Note

We use the following conventions for COBOL statement formats in this manual:

KEYWORD OPTIONWORD *required [optional]...*

Where	Means
KEYWORD	This is a COBOL reserved word which you must specify in the statement which contains it. You must spell it as shown or use one of the abbreviations or variant spellings, if any are given.
OPTIONWORD	This is an optional COBOL reserved word. You may include or omit it, but if you specify it you must spell it (or its abbreviation) exactly as shown.
required	This is a generic term which you *must* specify in the clause or option which contains it. (It may be a COBOL word, literal, picture string, comment entry, or complete syntactic entry.) Sometimes we use: *required-1* which means you may specify more than one of the generic terms indicated.
[]	Brackets enclose optional key words, generic terms, and clauses. Do not enter the brackets; they merely set off the option.
{}	Braces often enclose one or more arguments, clauses, phrases, or options. They indicate that you must specify one of these (in the case of options, specify one or none). Braces may also delimit a set of clauses. Do not enter the braces; they only set off the choices.
. . .	The ellipsis indicates that you may repeat the single element or set of choices which immediately precedes it.

Additionally, we use certain symbols in special ways:

Symbol	Means
)	Press the NEW LINE or RETURN key on your terminal's keyboard.
□	Be sure to put a space here. (We use this only when we must; normally, you can see where to put spaces.)

All numbers are decimal unless we indicate otherwise; e.g., 35_8.

Finally, in examples of system interactions, we use:

THIS TYPEFACE TO SHOW YOUR ENTRY)
THIS TYPEFACE FOR THE SYSTEM RESPONSE

We show the punctuation characters comma (,) and semicolon (;) in some formats for clarity. You may use them anywhere you use a space in your COBOL program, or you may omit them.

As required by the ANSI Standard COBOL language, periods (.) appear in the text after all section headers, paragraph names, Procedure Division sentences, and conditional statements and at the end of the last sentence or statement in a paragraph or section.

End of Preface

Table of Contents

Chapter 1 - Introduction

Chapter 2 - Language Concepts

Chapter 3 - The Identification Division

Chapter 4 - The Environment Division

Chapter 5 - The Data Division

Chapter 6 - The Procedure Division

Chapter 7 - Procedure Statements

Chapter 8 - The COPY Facility

Chapter 9 - The COBOL Interactive Debugger

Chapter 10 - How to Use COBOL Under RDOS

Chapter 11 - How To Use COBOL Under AOS

Appendix A - COBOL Reserved Words

Appendix B - Language Differences Between RDOS and AOS

Illustrations

Tables

Chapter 1
Introduction

General Description of ECLIPSE COBOL

Data General ECLIPSE COBOL is a complete COBOL language processing system. It includes the COBOL compiler and runtime library, full debugging support through an interactive COBOL-level debugging program, thorough English language error diagnostics, and a full range of program listings: source listing, cross-reference table, program map, generated code listing, and compilation statistics. It provides full access to Data General's INFOS file system and Sort/Merge facility.

ECLIPSE COBOL is based on the 1974 ANSI Standard ("American National Standard Programming Language COBOL," ANSI X3.23-1974). It implements the following modules of the ANSI Standard at the highest level:

Nucleus
Table Handling
Sequential I/O
Relative I/O
Indexed I/O
Sort/Merge
Segmentation
Library
Inter-Program Communication (Subprogramming)

COBOL Program Structure

A COBOL program is comprised of four divisions, named and ordered as follows:

```
IDENTIFICATION DIVISION .

     identification division body

ENVIRONMENT DIVISION .

     environment division body

DATA DIVISION .

     data division body

PROCEDURE DIVISION .

     procedure division body
```

Language Extensions

I/O Extensions

Data General COBOL includes significant extensions in the three input/output modules, particularly in the indexed I/O module where an extensive set of structured database capabilities is available. Data General COBOL provides the full standard indexed I/O capability including primary plus alternate indexes. In addition, we provide extremely powerful and efficient database management capabilities under INFOS via structured, multilevel indexing. You may dynamically define and create subindexes, each of which is linked to one or to many data records. The I/O verbs provide easy movement upward and downward through the index levels, as well as forward and backward within a single index level.

We provide many extra index key features, for example, we allow variable length keys, and we support partial records associated with keys. Also, you may specify approximate key references and generic key references. You may reference duplicate keys not only sequentially, but also randomly by occurrence number. And, because the keys are stored separately from the data records, access time is minimized.

You may delete indexed records logically (locally, globally, or both) as well as physically. Then you may restore the logically deleted records. Also indexed files are organized in such a way that they do not require frequent maintenance.

With sequential I/O, COBOL supports four record formats: fixed-length, variable-length, undefined-length, and data-sensitive. The code-set translation capability, which includes optional EBCDIC translation, allows you to select the fields you want translated so that you may process records containing packed decimal and binary data.

Data Extensions

ECLIPSE COBOL provides the following data types in addition to the standard alphabetic, alphanumeric, and decimal character data types:

variable length binary
packed decimal
floating point
external (character) floating point

COBOL uses ASCII code for internal character data and runs most efficiently with external character data in ASCII code as well. However, we provide code-set and collating-sequence conversion for other codes, particularly EBCDIC.

Communications

In place of the standard COBOL communications feature, which is large and costly to use on a small computer system, ECLIPSE COBOL provides a simpler communications facility which is very flexible and very efficient. Through simple file descriptors and CALL statements, you may employ all of the system control and asynchronous I/O features of the ECLIPSE Communications Access Manager (CAM). These calls let you send and receive messages, and check for I/O completion plus provide a number of other features. This facility is system dependent -- see Appendix G.

COBOL-Level Debugger

In place of the ANSI Standard debug module, which requires writing and compiling separate debug code, ECLIPSE COBOL provides an on-line, interactive COBOL-level debug module.

This module allows you to set and clear breakpoints at runtime. These breakpoints interrupt program execution; you may examine and/or modify data items at that point, then continue program execution. With these and other advanced features you can substantially reduce debugging time.

Sample Program

Figure 1-1 is a sample program that illustrates some of the features of ECLIPSE COBOL. The program calculates and displays a mortgage payment schedule. Simply enter it using a Data General text editor, compile it, bind it, and then execute it. (Chapters 10 and 11 contain complete operating procedures.) The messages displayed on the console will be:

ENTER PRINCIPAL:
INTEREST RATE (%):
YEARS TO PAY:
FUNCTION (0=SUMMARY, 1=FULL SCHEDULE):

After each colon, the system will await your response. When you have answered all questions, the system will calculate the requested schedule and then output it to the line printer.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. MORTPROG.
AUTHOR.  JOE SCHMOE.
DATE-WRITTEN.  6 OCT 1977.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.        ECLIPSE.
OBJECT-COMPUTER.        ECLIPSE.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
        SELECT OUTFILE, ASSIGN TO PRINTER.
DATA DIVISION.
FILE SECTION.
FD      OUTFILE, BLOCK CONTAINS 512 CHARACTERS.
01      OUTREC.
        02 OUT-PAYMT-NUM          PIC ZZZ9.
        02 FILLER                 PIC X(6).
        02 OUT-MON-INT            PIC $(4)9.99.
        02 FILLER                 PIC X(3).
        02 OUT-MON-PRIN           PIC $(6)9.99.
        02 FILLER                 PIC X(3).
        02 OUT-BALANCE            PIC $(6)9.99.
        02 FILLER                 PIC X(8).
        02 OUT-INT-TO-DATE        PIC $(6)9.99.
        02 FILLER                 PIC X(10).

WORKING-STORAGE SECTION.
01      CRT-INPUTS.
        02 PRINCIPAL              PIC 9(6)V99.
        02 PERCENT                PIC 99V99.
        02 YEARS                  PIC 99.
        02 FUNCTION               PIC 9.
        02 REPEAT-FLAG            PIC 9.

01      TEMPS.
        02 MONTHLY-INT-RATE       USAGE COMP-1.
        02 MONTHS                 PIC 9(4).
        02 MONTHS-LEFT            PIC 9(4).
        02 MONTHLY-PAYMT          PIC 9(4)V99.
        02 LOAN-BAL               PIC 9(6)V99.
        02 INT-TO-DATE            PIC 9(6)V99.
        02 PAYMT-NUM              PIC 9(4), USAGE COMP.
        02 INT-PAYMT              PIC 9(4)V99.
        02 PRIN-PAYMT             PIC 9(4)V99.
```

Figure 1-1. COBOL Sample Program

```
01      SUMMARY-LINE1.
        02 FILLER        PIC X(16), VALUE "PRINCIPAL =         ".
        02 SUMMARY-PRIN, PIC $(6)9.99.
        02 FILLER        PIC X(50), VALUE SPACES.
01      SUMMARY-LINE2.
        02 FILLER        PIC X(20), VALUE "INTEREST RATE =        ".
        02 SUMMARY-RATE, PIC 9.9(4).
        02 FILLER        PIC X(50), VALUE SPACES.
01      SUMMARY-LINE3.
        02 FILLER        PIC X(18), VALUE "LOAN LIFE =           ".
        02 SUMMARY-YEARS, PIC Z9.
        02 FILLER        PIC X(6), VALUE " YEARS".
        02 FILLER        PIC X(50), VALUE SPACES.
01      SUMMARY-LINE4.
        02 FILLER        PIC X(18), VALUE "MONTHLY PAYMENT = ".
        02 SUMMARY-PAYMT, PIC $(4)9.99.


        02 FILLER        PIC X(50), VALUE SPACES.

01      HEADLINE        PIC X(80),
            VALUE " NUM        INTEREST     PRIN. PAY  PRIN.
-                 " BAL       INTEREST PAID TO DATE".

PROCEDURE DIVISION.
INIT.   OPEN OUTPUT OUTFILE.
OPERATOR.
        DISPLAY "ENTER PRINCIPAL:   $" WITH NO ADVANCING.
        ACCEPT PRINCIPAL.
        DISPLAY "INTEREST RATE (%): " WITH NO ADVANCING.
        ACCEPT PERCENT.
        COMPUTE MONTHLY-INT-RATE = PERCENT / 100 / 12.
        DISPLAY "YEARS TO PAY:      " WITH NO ADVANCING.
        ACCEPT YEARS.
        COMPUTE MONTHS = YEARS * 12.
        DISPLAY "FUNCTION (0=SUMMARY, 1=FULL SCHEDULE): "
                                WITH NO ADVANCING.
        ACCEPT FUNCTION.
        COMPUTE MONTHLY-PAYMT ROUNDED =
                PRINCIPAL * MONTHLY-INT-RATE *
                        (1 + MONTHLY-INT-RATE) ** MONTHS /
*               ------------------------------------------
                ((1 + MONTHLY-INT-RATE) ** MONTHS - 1).
        PERFORM SUMMARY-OUTPUT.
        IF FUNCTION NOT = 0,
                PERFORM DETAIL-OUTPUT.
        DISPLAY "TYPE 1 TO REPEAT, 0 TO STOP: " WITH NO ADVANCING.
        ACCEPT REPEAT-FLAG.
        IF REPEAT-FLAG NOT = 0, GO TO OPERATOR.
        CLOSE OUTFILE.
        STOP RUN.

SUMMARY-OUTPUT.
        MOVE PRINCIPAL TO SUMMARY-PRIN.
        WRITE OUTREC FROM SUMMARY-LINE1 BEFORE ADVANCING 1.
        COMPUTE SUMMARY-RATE = PERCENT / 100.
        WRITE OUTREC FROM SUMMARY-LINE2 BEFORE ADVANCING 1.
        MOVE YEARS TO SUMMARY-YEARS.
        WRITE OUTREC FROM SUMMARY-LINE3 BEFORE ADVANCING 2.
        MOVE MONTHLY-PAYMT TO SUMMARY-PAYMT.
        WRITE OUTREC FROM SUMMARY-LINE4 BEFORE ADVANCING 2.
```

Figure 1-1. COBOL Sample Program (continued)

```
DETAIL-OUTPUT.
        MOVE PRINCIPAL TO LOAN-BAL.
        MOVE MONTHS TO MONTHS-LEFT.
        MOVE 0 TO INT-TO-DATE.
        WRITE OUTREC FROM HEADLINE BEFORE ADVANCING 2.
        MOVE SPACES TO OUTREC.
        PERFORM DO-DETAIL-LINE
                VARYING PAYMT-NUM FROM 1 BY 1
                        UNTIL PAYMT-NUM > MONTHS.

DO-DETAIL-LINE.
        COMPUTE PRIN-PAYMT ROUNDED =
                LOAN-BAL * MONTHLY-INT-RATE /
*               ----------------------------------------
                ((1 + MONTHLY-INT-RATE) ** MONTHS-LEFT - 1).
        SUBTRACT 1 FROM MONTHS-LEFT.
        COMPUTE INT-PAYMT = MONTHLY-PAYMT - PRIN-PAYMT.
        SUBTRACT PRIN-PAYMT FROM LOAN-BAL.
        ADD INT-PAYMT TO INT-TO-DATE.
        MOVE PAYMT-NUM TO OUT-PAYMT-NUM.
        MOVE INT-PAYMT TO OUT-MON-INT.
        MOVE PRIN-PAYMT TO OUT-MON-PRIN.
        MOVE LOAN-BAL TO OUT-BALANCE.
        MOVE INT-TO-DATE TO OUT-INT-TO-DATE.
        WRITE OUTREC BEFORE ADVANCING 2.
*   END OF PROGRAM
```

Figure 1-1. COBOL Sample Program (continued)

End of Chapter

Chapter 2
Language Concepts

The Character Set

The COBOL character set consists of the characters listed in Table 2-1. Corresponding upper- and lowercase letters are equivalent. Comma and semicolon are equivalent to a space. In text format, the characters tab, NL (new line), and CR (carriage return) are also equivalent to a space. Wherever a COBOL program requires a space, you may replace it by any number of spaces.

Table 2-1. COBOL Character Set

Character	Name	Character	Name
0,1,...,9	digits	,	comma
A,B,...,Z	letters	;	semicolon
	space	.	period, decimal point
+	plus sign	"	quotation mark
-	minus sign or hyphen	'	apostrophe
*	asterisk	(	left parenthesis
/	slash	)	right parenthesis
=	equal sign	>	greater than symbol
$	dollar sign	<	less than symbol

You may use the entire ASCII character set (except for null, rubout, NL, and CR) in alphanumeric literals.

Separators

The elements of a COBOL source program (words, literals, etc.) are delimited by separators. The separators are space, comma, and semicolon, and a period followed by either a space or any space equivalent.

COBOL Words

A word is a string of up to 30 characters selected from the set of valid COBOL letters (A through Z), the set of valid COBOL digits (0 through 9), and the hyphen (-). A word may not begin or end with a hyphen.

Key, Optional, and Special Character Words

Key words are reserved words you must include in your COBOL program. (They are underlined and uppercase in the source formats.) Optional words may appear in your program at your discretion. (They are uppercase but not underlined in the source formats.) If you do include optional words in your program, you

must spell them *exactly* as given. Special character words are =, +, -, >, <, *, /, **, (, and). In order to avoid confusion with other symbols (e.g., ≥), we do not underline them in the source formats. However, these characters have the same status as key words and you must specify them. A list of all ECLIPSE COBOL key words appears in Appendix A.

Figurative Constants

Figurative constants are COBOL key words which represent specific constants. The singular and plural forms are equivalent and you may use them interchangeably. The COBOL figurative constants are:

ZERO ZEROS ZEROES	Represents the value 0 or one or more of the character 0, depending on the context.
SPACE SPACES	Represents one or more of the character space.
HIGH-VALUE HIGH-VALUES	Represents one or more of the character that is highest in the program collating sequence (rubout in ASCII).
LOW-VALUE LOW-VALUES	Represents one or more of the character that is lowest in the program collating sequence (null in ASCII).
QUOTE QUOTES	Represents one or more of the character ". You cannot use the word QUOTE or QUOTES in place of a quotation mark to delimit an alphanumeric literal.
CR	Represents one or more line-terminating characters.
ALL *literal*	Represents one or more of the string of characters comprising *literal.* The literal must be either an alphanumeric literal or a figurative constant other than the word ALL. When you use a figurative constant, the word ALL is redundant and merely enhances readability.

When a figurative constant represents a string of one or more characters, the context determines the length of the string according to the following rules. If the figurative constant is associated with a data item, the string of characters specified by the figurative constant is repeated character by character from left to right until the size of the resultant string is equal to the length of the associated data item. COBOL does this without interpreting any JUSTIFIED clause in the data description entry for the data item. If the figurative constant is not associated with a data item, the length of the string is one character.

You may use figurative constants interchangeably with literals in a source format unless the literal is numeric. If the literal is numeric, you can only use the figurative constant ZERO (ZEROS, ZEROES). The characters associated with the figurative constants HIGH-VALUE and LOW-VALUE depend on the program collating sequence you specify in the Object-Computer paragraph of the Environment Division.

Special Register

LINAGE-COUNTER is a special register associated with a print file. You establish the counter by specifying the LINAGE clause in the print file's FD entry in the Data Division. You may specify this register in the Procedure Division, but you may not modify it. COBOL automatically creates and modifies the LINAGE-COUNTER for each print file. The LINAGE-COUNTER contains the number of the current line on the current page.

User-Defined Words

User-defined words are names you declare to specify various entries in your program. The types of names that you may declare are listed in Table 2-2.

Table 2-2. User-defined Words

Type	Use it to name	Reference
Alphabet name	A code set or a collating sequence	Chapter 4
Channel name	An output device control channel	Chapter 4
Condition name	A switch status or a value of a data item	Chapters 4 and 5
Data name	A record or other data item	Chapter 5
Filename	A data file	Chapters 4 and 5
Mnemonic name	A Data General system name	Chapter 4
Paragraph name	A Procedure Division paragraph	Chapter 6
Program name	The source program	Chapter 3
Section name	A Procedure Division section	Chapter 6
Switch name	A system switch	Chapter 4

We refer to both paragraph names and section names as procedure names. All names except procedure names must contain at least one alphabetic character.

Literals

A literal is a constant whose value is determined by the characters which form it. Literals may be either numeric or alphanumeric.

Numeric Literals

You may specify numeric literals in any of the following forms:

Integer $\pm$ ddd.

In this form, ddd is a string of 1 to 18 digits. The decimal point is optional. If you do not specify a sign, positive is assumed.

Decimal ±ddd.ddd

You may specify no more than 18 digits for this type. The digit string to the left of the decimal point is optional. If you do not specify a sign, positive is assumed.

Floating Point ±ddd.dddE±dd

The left part of a floating point literal, ±ddd.ddd, is the mantissa, M, and may be either a decimal or an integer literal. It may have no more than 16 digits. The right part, E+dd, is the exponent, e; you must specify both digits and the sign. The value of the literal is calculated as:

$M \times 10^e$

Alphanumeric Literals

An alphanumeric literal is a string of up to 132 characters delimited on both ends by a special character. The beginning and ending delimiters must both be either apostrophes (') or quotes ("). If the delimiters are quotes, COBOL treats an apostrophe within the literal as an ordinary character. If the delimiters are apostrophes, COBOL treats a quotation mark within the literal as an ordinary character. If you want the delimiter character to appear within the literal, specify two, contiguous instances of the delimiter character.

The value of an alphanumeric literal within the object program is the string of characters itself, with the following exceptions:

1. COBOL does not interpret the delimiters.
2. Program compilation replaces each imbedded pair of delimiter characters by a single instance of the character.
3. You may not specify null, CR (carriage return), NL, or rubout within a literal.
4. If you specify the global /E compilation switch, you may include any character in the literal by inserting its octal ASCII code delimited by angle brackets (< and >). For example:

To include	Insert
carriage return	<015>
<	<074>
>	<076>
null	<000>
rubout	<177>
"	<042>
' (apostrophe)	<047>
A	<101>
line feed (new line)	<012>
form feed	<014>

If you enclose a nonoctal code within the angle brackets, COBOL signals an error at compile time. If you enclose an octal code outside the range 0 to 377_8 within angle brackets, COBOL ignores the number and the angle brackets and no error is signalled. (For a table of the ASCII character set, see Appendix E.) For example, if you specify the /E compilation switch, the literal

"AB'CD""EF<046>GH<401>IJ"

has the compiled value

AB'CD"EF&GHIJ

Source Formats

You always code source text for ECLIPSE COBOL in ASCII if you are using Data General's input devices. COBOL ignores the null and rubout characters. The COBOL compiler accepts two source text formats: terminal-oriented text format and industry-compatible card format.

Lines, Statements, and Sentences

In *text format,* a line consists of 0 to 255 nonterminator characters, followed by a line-terminating character. COBOL compiles the entire line as source text. An indicator character (one of the characters listed in Table 2-3) may be the first character of any line.

In *card format,* a line is also a line-terminating character preceded by 0 to 255 nonterminator characters. COBOL treats the first six characters of the line (the sequence number field) as a comment. COBOL also treats any characters after the 72nd character in a line as a comment. Column 7 is the indicator field. It must contain either a space or one of the characters listed in Table 2-3.

Table 2-3. Indicator Characters

Indicator Character	Meaning
D	Debug Line
* or /	Comment Line
Tab	A-Margin is Blank
	Continuation Line

A *statement* is a valid combination of words and symbols that you write in the Procedure Division of your program. It begins with a COBOL verb and optionally ends with a period.

A *sentence* is a sequence of one or more statements followed by a period.

Debug Lines

You indicate a debug line by inserting the character D in the first character position of the source line in text format, or in column 7 in card format. COBOL compiles debug lines if you specify the global /D switch in the compilation command line. Otherwise, it treats them as comment lines. In text format, a space or tab must immediately follow the D, or COBOL will treat the D as a source character.

Comment Lines

A COBOL comment line is a blank line or a line that contains *, /, or D (if you do not specify /D in the compilation command line) in the indicator field, and is followed by any combination of COBOL characters. The character / advances the source program listing to a new page before printing the comment. You must follow the character D by a space or a tab; this is not required for * or /. Comment lines will appear in your output listing, but the compiler will ignore them.

Continuation Lines

Ordinarily a line-terminating character is syntactically equivalent to a space (except for its special function of advancing to the next line). However, if the indicator character of the next source line is a hyphen (-), then COBOL ignores the line terminator and interprets the following line as a continuation of the previous line. The rules for continuation lines are:

1. If the preceding line ended with an incomplete alphanumeric literal (e.g., "ABCD), the first nonblank character of the continuation line must be the delimiter character (that is, ' or ", whichever you used in the preceding line). The characters immediately following must be the next characters in the continued literal, ending with the delimiter (e.g., "EFGH").

2. If the preceding line did not end with an incomplete alphanumeric literal, then the first nonblank character of the continuation line is the immediate successor of the last nonterminator character on the previous line.

Examples:

```
Line 1  □.....MO
Line 2  -□□VE.....
```

COBOL compiles the word MOVE.

```
Line 1  □....."AB
Line 2  -□□□"CD"....
```

COBOL compiles the literal "ABCD".

Clauses and Phrases

A *clause* is an ordered set of consecutive COBOL character strings. It specifies an attribute of an entry.

A *phrase* takes the same form as a clause, but it is either a portion of a COBOL Procedure Division statement or of a COBOL clause.

The A-Margin

The A-margin is the leftmost part of the source line. Paragraph and section names in the Procedure Division must begin in the A-margin. In text format, the A-margin is the first 4 characters of the source line excluding the indicator, if any. In card format, the A-margin is columns 8 through 11 of the complete line. A tab anywhere in the A-margin (in card format, a tab in column 7) causes COBOL to compile the remaining characters of the source line as if they were immediately to the right of the A-margin.

End of Chapter

Chapter 3
The Identification Division

The Identification Division identifies the entry point for your program. It also allows you, at your option, to include special documentary information. It takes the following format:

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  progname.
[AUTHOR.          [comment entry] ... ]
[INSTALLATION.    [comment entry] ... ]
[DATE-WRITTEN.    [comment entry] ... ]
[DATE-COMPILED.   [comment entry] ... ]
[SECURITY.        [comment entry] ... ]
```

All paragraphs in this division are optional except the PROGRAM-ID paragraph. *Progname* not only identifies an entry point in the COBOL program unit, but also identifies three debugger files *(progname.DB, progname.DL,* and *progname.DS).* The names of your source and object files need not be the same as *progname.* However, *progname* should not be the same as another program's source filename.

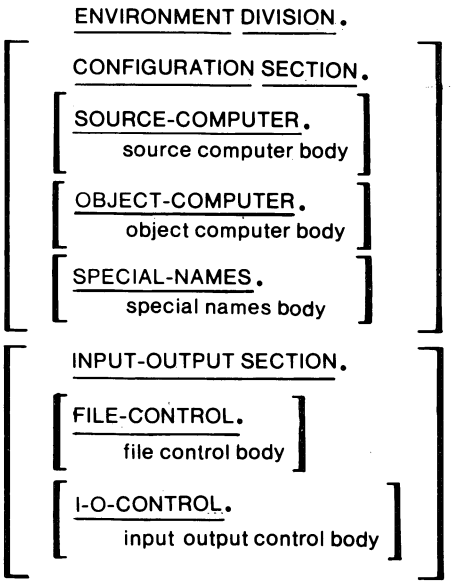
You may present the optional paragraphs in this division in any order. The compiler ignores the comment entry text strings. However, if you specify the DATE-COMPILED paragraph, the compiler will print the current date on the listing between DATE-COMPILED and the comment entry.

End of Chapter

Chapter 4
The Environment Division

Structure

The Environment Division supplies information about the physical characteristics of your computer system. It is comprised of two sections which have different functions. The first, the Configuration Section, deals with the characteristics of your source and object computer. The second, the Input-Output Section, contains information pertinent to the transmission and handling of data between external media and your object program. The Environment Division takes the following format:



Configuration Section

The Configuration Section consists of three paragraphs: the Source-Computer paragraph, the Object-Computer paragraph, and the Special-Names paragraph.

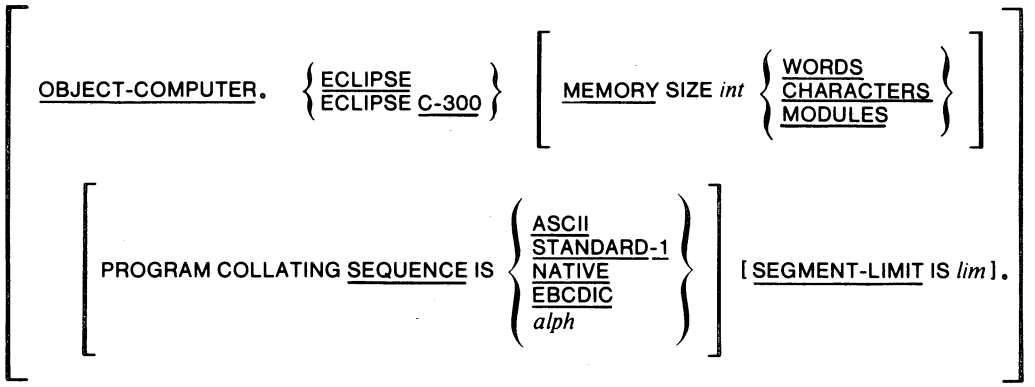
The Source-Computer Paragraph

The Source-Computer paragraph identifies the computer on which you will compile your program and has the format:

[SOURCE-COMPUTER. [*comment entry*] ...]

The Object-Computer Paragraph

The Object-Computer paragraph identifies and describes the computer on which you will execute your object program. It has the format:



where:

int is a positive integer literal that specifies the size of memory in WORDS, CHARACTERS, or MODULES, whichever you specify.

alpha is an alphabet name indicating that this program will use the collating sequence associated with the *alpha* you define in the Special-Names paragraph.

lim is an integer literal in the range 1 through 49, inclusive, that specifies a segment limit.

The ECLIPSE COBOL system does not require you to specify a MEMORY SIZE clause. If you do specify one, the value is ignored by the compiler.

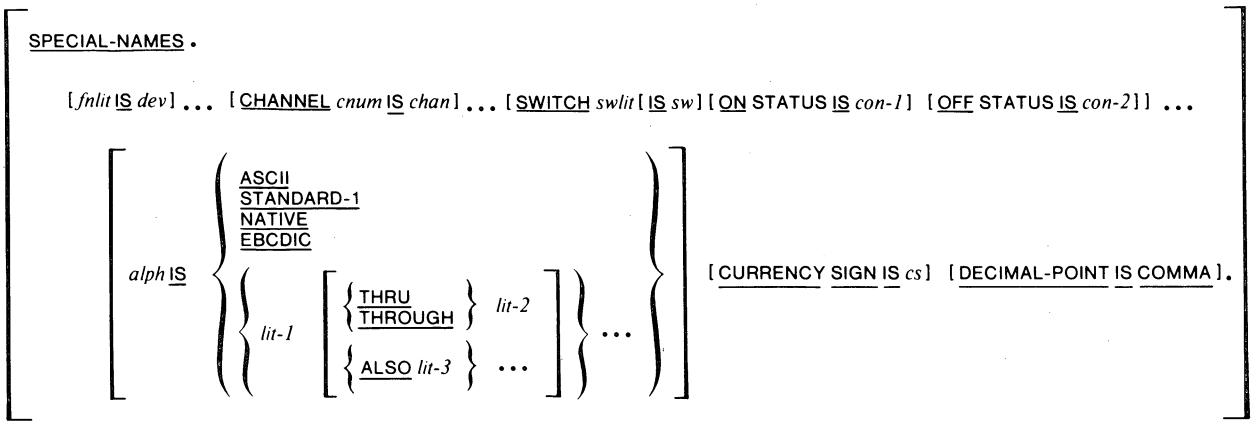
The *program collating sequence* is the relational order of the data characters which COBOL encounters internally during your program's execution. When COBOL compares character strings (alphanumeric data items or alphanumeric literals) in explicitly stated relational conditions and relational conditions implied by condition names, this order determines the outcome of those comparisons. It also applies to character string comparisons that are implicit in SORT and MERGE operations unless you specify another collating sequence for a particular SORT or MERGE. It does not apply to character string comparisons implicit in other statements, such as STRING and UNSTRING.

If you do not specify the PROGRAM COLLATING SEQUENCE clause or if you specify ASCII, STANDARD-1, or NATIVE, the program collating sequence is the standard ordering of the ASCII character set as specified in Appendix E. If you specify EBCDIC, the program collating sequence is that of the EBCDIC character set, as in Appendix F.

The SEGMENT-LIMIT clause indicates which segments of your program are resident. You assign a segment number in the section header for each segment in the Procedure Division. If you omit this clause, COBOL assumes that all segments to which you assigned segment numbers between 0 and 49 inclusive are resident; those with segment numbers between 50 and 99 inclusive are overlay segments. If you do specify this clause, those segments with numbers from 0 up to, but not including, *lim* are resident segments; those with segment numbers from *lim* through 99 inclusive are overlay segments.

The Special-Names Paragraph

The Special-Names paragraph specifies filenames and mnemonic names for special program elements and relates alphabet names to a specific character code set and/or collating sequence. It has the format:



where:

fnlit is an alphanumeric literal that specifies the name of a system I/O device or file.

dev is a mnemonic name that you use in your program to reference *fnlit.*

cnum is an integer literal that specifies a channel number with a value from 1 through 12.

chan is the channel name you use in your program to reference the channel associated with *cnum.*

swlit is a single alphabetic character literal that specifies the program execution switch.

sw is the internal switch name for *swlit.*

con is a condition name associated with the ON and OFF STATUS conditions.

alph is an alphabet name.

lit is a numeric or alphanumeric literal. If it is numeric, it must be an unsigned integer, with a value from 1 through 256. If it is alphanumeric, and you specify THROUGH or ALSO, it must be one character in length.

cs is a single-character alphanumeric literal that specifies the currency character.

You use the device clause (the items *fnlit* and *dev)* with ACCEPT and DISPLAY Procedure Division statements.

The CHANNEL clause declares a line printer control channel. You may use channel names in the ADVANCING clause of a WRITE statement to format data for printed forms. The actual interpretation of a given channel number depends on the channels specified to the VFU utility (if your printer uses a programmable data channel VFU), or the punches in the channel on the paper control tape mounted in the line printer.

The SWITCH clause declares the ON and/or OFF STATUS conditions for a program execution switch. You never reference the internal switch name *(sw)* in your COBOL program; if you specify it, it serves as documentation only. You may specify the ON and OFF clauses in any order, if you specify them both.

For example, if you declare switches in the Special Names paragraph of program PAYROLL as:

```
SWITCH "F" ON STATUS IS FIRST-OF-MONTH,
SWITCH "B" ON IS Y-TO-D, OFF IS NO-Y-TO-D
```

you can then give the following execution commands:

Command	Meaning
PAYROLL/F/B (or /B/F)	Both switches on
PAYROLL/B	B on, F off
PAYROLL/F	F on, B off
PAYROLL	Both off

The alphabet clause defines a specific character code set and/or collating sequence. When referenced by the PROGRAM COLLATING SEQUENCE clause in the Object Computer paragraph, or the COLLATING SEQUENCE clause of a SORT or MERGE statement, *alph* represents a collating sequence. When referenced by the CODE-SET clause of a file declaration, it represents a character code set. If you specify ASCII, STANDARD-1, or NATIVE, *alph* represents the ASCII code set/collating sequence in Appendix E. If you specify EBCDIC, *alph* represents the EBCDIC code set/collating sequence in Appendix F. If you specify THROUGH or ALSO, i.e., if you define *alph* by a series of contiguous literals, you may reference *alph* only as a collating sequence. You must not specify a single character or number more than once in the alphabet clause. THRU is equivalent to THROUGH.

You define the collating sequence identified by the literal phrase according to the following rules:

1. The order in which you specify the literals in the alphabet clause determines the ordinal position in ascending sequence of the characters within the collating sequence.

2. If the literal is numeric, its value represents the ASCII code of the character whose ordinal position corresponds to that value.

3. If the literal is a single alphanumeric character, it defines the ordinal position of that character within the ASCII character set. If you specify a multicharacter alphanumeric literal, COBOL assigns the next position (in ascending order) in the collating sequence to each character in the literal, starting with the leftmost.

4. Any character that you do not explicitly specify in the literal phrase assumes a position in the collating sequence greater than that of any explicitly specified character. The relative order within the set of these unspecified characters is their order in the ASCII sequence.

5. If you specify THROUGH, and if *lit-1* is less than *lit-2,* COBOL assigns successive ascending positions in the collating sequence to the contiguous characters in the ASCII set beginning with the character specified by *lit-1* and ending with the character specified by *lit-2.* If *lit-1* is greater than *lit-2* COBOL assigns successive ascending positions (reverse order) in the collating sequence to the contiguous characters in the ASCII set. Assignment begins with *lit-1* and continues backward through *lit-2.*

6. If you specify ALSO, COBOL assigns the same position in the collating sequence to the characters of the ASCII set within *lit-1* and *lit-3.*

Examples:

AL-1 IS "A" "1" "B" "2"

specifies the collating sequence:

A, 1, B, 2, NULL, ↑A, ↑B,...

(↑ before a letter means CONTROL.) NULL, ↑A, etc., follow 2 because they are the first characters of the ASCII code set.

AL-2 IS 65 49 66 50

specifies the same collating sequence as above, because these numbers are the decimal ASCII equivalents to the literals specified above.

AL-3 IS "A" THRU "Z", "9" THRU "0"

specifies the collating sequence:

A,B, ..., Y, Z, 9, 8, 7, ..., 0, NULL, ↑A, ↑B, ...

AL-4 IS "DdEeFf", "0123456", "GgHh"

specifies the collating sequence:

D, d, E, e, F, f, 0, 1, 2, 3, 4, 5, 6, G, g, H, h, NULL, ↑A, ↑B...

AL-5 is "R", "A" ALSO "Z", "M" THRU "Q", "B" ALSO "F", "E"

specifies the collating sequence:

$$R, \left\{\begin{matrix} A \\ Z \end{matrix}\right\}, M, N, O, P, Q, \left\{\begin{matrix} B \\ F \end{matrix}\right\}, E, NULL, \uparrow A. \uparrow B, \ldots$$

If you want to use a character other than $ in picture strings to signify the currency symbol, specify the CURRENCY SIGN clause. The character you specify will be the character COBOL stores during data editing. The currency symbol may be any of the letters, E,F,G,H,I,J,K,M,N,O,Q,T,U,W,Y, or any of the special characters: !, @, #, $, %, :, &, , and - (hyphen). If you do not specify the CURRENCY SIGN clause, CURRENCY SIGN IS $ is assumed.

The DECIMAL-POINT IS COMMA clause reverses the use of comma and period in a picture string and in numeric literals.

Processing Files in COBOL

Before we get into the Input/Output Section of the Environment Division, it is important for you to know about COBOL's file system. ECLIPSE COBOL uses the INFOS file management system. If you are unfamiliar with the concept of file management, the *INFOS Storybook* (69-019) tells you about the basics and describes the experiences of a company putting these basics to work. If you are running your COBOL system under RDOS, you should read the *RDOS/INFOS System User's Manual* (93-114) to become familiar with SAM (Sequential Access Method), RAM (Random Access Method), ISAM (Indexed Sequential Access Method), and DBAM (Data Base Access Method). If you are running under AOS, you should read the *AOS Programmer's Manual* (93-120) for information on sequential and relative file organizations, and sequential and random access methods; also read the *AOS/INFOS System Manual* (93-152) for information on ISAM and DBAM.

File Organizations

The structure of data on a physical file is called the file's organization. A file's organization remains fixed for the life of the file. The ECLIPSE COBOL system provides three types of file organization: sequential, relative, and indexed. It also provides one special kind of file called a sort/merge file.

Sequential Files

A *sequential file* is a string of logical records whose order you establish by the sequence in which you record them at the file's creation time. COBOL reads these records in the order that you wrote them. When you write additional records to a sequential file, COBOL appends them to the end of the existing file, again adding them in the order in which you write them. No reference keys exist for records in sequential files.

If you record your data on a sequential access device (e.g., magnetic tape), or if you want to output your data to a sequential device (e.g., line printer), you should use sequential file organization to create and process your data.

With sequentially organized files, you can choose any of four formats for your records: fixed-length, variable-length, undefined-length, or data-sensitive, whichever is appropriate to your application. (See Chapter 5 for more information on record formats.) You can create your sequential file on any Data General peripheral device, including labeled and unlabeled tape. Because COBOL anticipates your data requirements and automatically brings buffers in as they are needed to process your data, you can process sequentially organized files with great time efficiency.

With a sequentially organized file, you can use COBOL Procedure Division statements to open, close, and delete the file (OPEN, CLOSE, EXPUNGE); read, write, and rewrite records (READ, WRITE, REWRITE); position the record pointer in the file (START); terminate I/O operations on records from the current logical block (TRUNCATE); and define procedures for I/O error handling (USE).

You declare a sequentially organized file with the clause ORGANIZATION IS SEQUENTIAL in the file's SELECT clause.

Relative Files

A *relative file* consists of logical records that the COBOL program identifies by a relative record number, which you assign. You can think of this file as a string of areas, each area capable of holding one logical record. Each of these areas is identified by a relative record number, with the first record of the file numbered 1, the second numbered 2, etc. The sequence in which COBOL writes the records has no bearing on the sequence in which you retrieve them. Because you store and retrieve the records on the basis of their relative record numbers, you may access them nonsequentially if you want. Relative file organization is allowed only on disk devices.

Keep in mind that you should assign relatively low numbers to your records, because COBOL assigns a record to a physical area which corresponds to that record's relative number. For example, COBOL will write record number 500 into area number 500. If 500 is your lowest record number, you have just wasted areas 1 through 499.

When you build a relative file, you associate a relative record number with each record. These numbers are the keys by which COBOL references the records. When you want to access a relative file's record, you simply specify the record's relative number and COBOL goes directly to that record.

With a relative file, you can use COBOL Procedure Division statements to open, close, and delete the file (OPEN, CLOSE, EXPUNGE); read, write, and rewrite records (READ, WRITE, REWRITE); position the record pointer in the file (START); position the I/O system at a record (SEEK); and define procedures for I/O error handling (USE).

You declare a relative file with the clause ORGANIZATION IS RELATIVE in the file's SELECT clause.

Indexed Files

ECLIPSE COBOL provides four types of *indexed file* capabilities:

- simple indexing,
- indexing with alternate record keys,
- multiple indexing, and
- multilevel indexing.

The first two types are standard COBOL features. The last two are ECLIPSE COBOL extensions based on the INFOS file management system.

In general terms, an indexed file contains logical records which the COBOL program identifies by the value of a key, rather than by their physical or logical position. A *key* is a shorthand way of telling the system which record you want. It can be any piece of data within a record, or an element external to the record. When you create an indexed file, you associate a key with each record. You may then randomly reference these keys to locate or process your data, or you may reference your data sequentially in ascending order of the keys.

A key may consist of numbers, letters, or both, and can vary in length to give you storage efficiency. COBOL automatically maintains key/data association, allowing you fast access to your data and enabling your file to grow or shrink without restrictions.

A record description may include one or more keys, each associated with an index. This index provides a logical path to the database records based on the contents of the key for that index. So you may have several indexes for different orders (inversions) and/or subsets of the file. (We define inversion later in this section.) And you may build a hierarchy of subindexes within an index, identifying each level by a key.

COBOL stores an indexed file as several files. The number and organization of files depend on your operating system. (See Appendix B.) Among these files are those containing the file's indexes and those containing the file's database. A database file is a random access file. An index file is an ordered file containing keys that reference database records. As you write each database record, you supply a key for that record. Then the system automatically keeps track of the location of that key's database record. Remember, *you* supply the keys; they have no meaning to the system except for their association with your database records. When you want to retrieve a record, you supply the key and the system gives you the database record.

With an indexed file, you can use COBOL Procedure Division statements to open, close, and delete the file (OPEN, CLOSE, EXPUNGE); read, write, rewrite, and remove records (READ, WRITE, REWRITE, DELETE); restore previously deleted records (UNDELETE); position the record pointer in the file (START); create, delete, and provide shared subindexes (DEFINE SUB-INDEX, EXPUNGE SUB-INDEX, LINK SUB-INDEX); obtain information about a key (RETRIEVE); and define procedures for I/O error handling (USE).

You declare an indexed file by specifying the ORGANIZATION IS INDEXED clause in the file's SELECT clause. You can access an indexed file only on disk devices.

A *simple indexed file* provides a single index for the database file. There is no hierarchy of keys; a single key exists for each record in the database file. Figure 4-1 shows an example of simple indexing.

You may access a simple indexed file either sequentially, randomly, or dynamically.

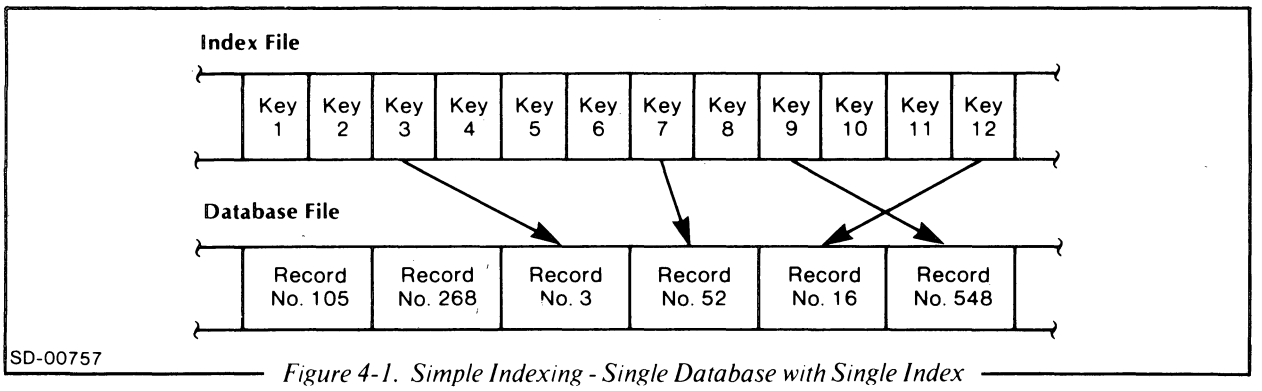


Figure 4-1. Simple Indexing - Single Database with Single Index

An *indexed file with alternate record keys* has as many subindexes for the database file as there are alternate keys plus one subindex for the prime key. There is one prime record key and one or more alternate record keys. COBOL maintains a one-to-one correspondence between the key entries of each index and each record of the data file, just as it does with the simple indexed files. You may read a record using any of the keys, but you may write a record only with the prime record key. When you write a record, COBOL writes a key entry for it in every subindex. You cannot combine this form of indexing with multiple or multilevel indexing.

Because COBOL constructs indexed files with alternate keys to meet the requirements of ANSI Standard COBOL, you must exercise special care if you plan to access such a file with programs written in languages other than COBOL. With alternate key files, when you enter a record using the prime key inversion, COBOL automatically creates corresponding entries in all the other inversions. ECLIPSE COBOL implements this by constructing an indexed file with alternate keys as a two-level indexed file with a separate subindex for each of the inversions.

For example, if you had a file where the main index refers to various regional offices, and you define keys A (the prime record key), B, C, and D (the alternate keys), then COBOL would construct the file shown in Figure 4-2. The entry contained in key 0 in the main level of the index would point to the subindex which represents the inversion associated with the prime key; the entry contained in key 1 in the main level would point to the subindex for the inversion associated with the first alternate key, and so on.

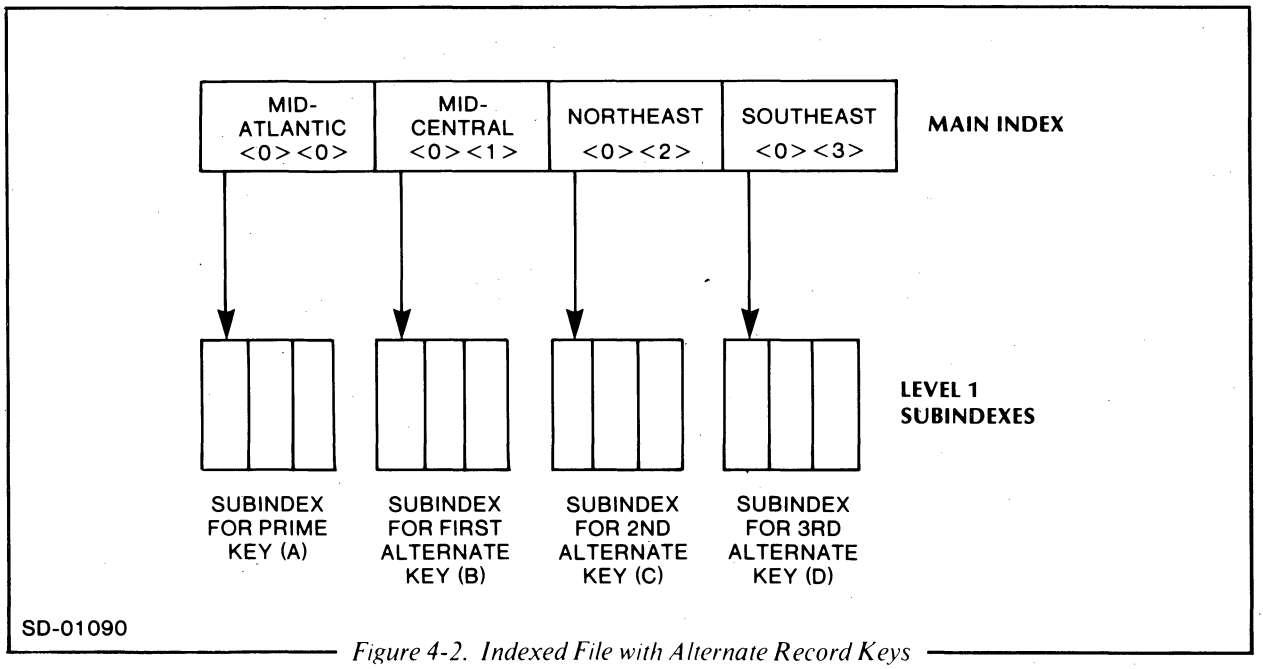


Figure 4-2. Indexed File with Alternate Record Keys

To the ordinary COBOL user the main-level index is completely transparent and subindexes seem to be simply inversions of the same database.

With *multiple indexing,* you can create several completely independent indexes for one data file, as shown in Figure 4-3. Each index may reflect all or any subset of the database, in any order. You can regard each index along with the database as a separate file called an *inversion.*

To invert a file, you have to access a database record via an already established key or set of keys. Then you do a WRITE INVERTED to your new index and the system will automatically create a new index entry in it. Thus, you will have linked two unique index entries, one in each index, to the same database record.

You may make an entry in any one of the inversions without entering it in any of the others; or you can write the entry in as many of the inversions as you want. Of course, the system enters the data record in the database file only once. You may also link the subindexes in an indexed file, which lets you share data without reproducing a subindex structure. Figure 4-4 shows a possible application of linked subindexing.

Multilevel indexing allows you to use structured, hierarchical database files. Entries in the main index may point to subordinate subindexes in place of or in addition to data records. Entries in a subordinate subindex may point to even lower level subindexes. There is no limit to the variety of structure which a multilevel indexed file may have.

You access records in such a file by multikey references, by relative positioning in the hierarchy (UP, DOWN, etc.), or by a combination of these. You can access a single data record through many sets of keys in one index, and through many sets of keys in an entirely different index.

You may combine multiple and multilevel indexing by creating several simple and/or complex indexes for the same database file. Figure 4-5 illustrates a multilevel indexed file.

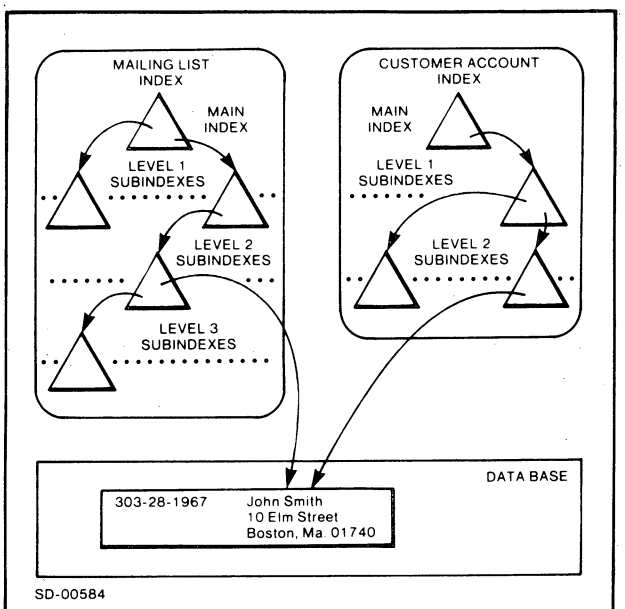


Figure 4-3. Multiple Indexing; Single Database with Two Indexes

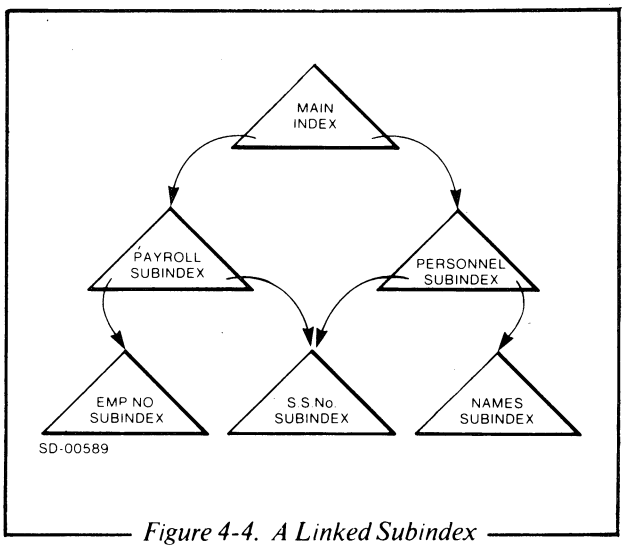


Figure 4-4. A Linked Subindex

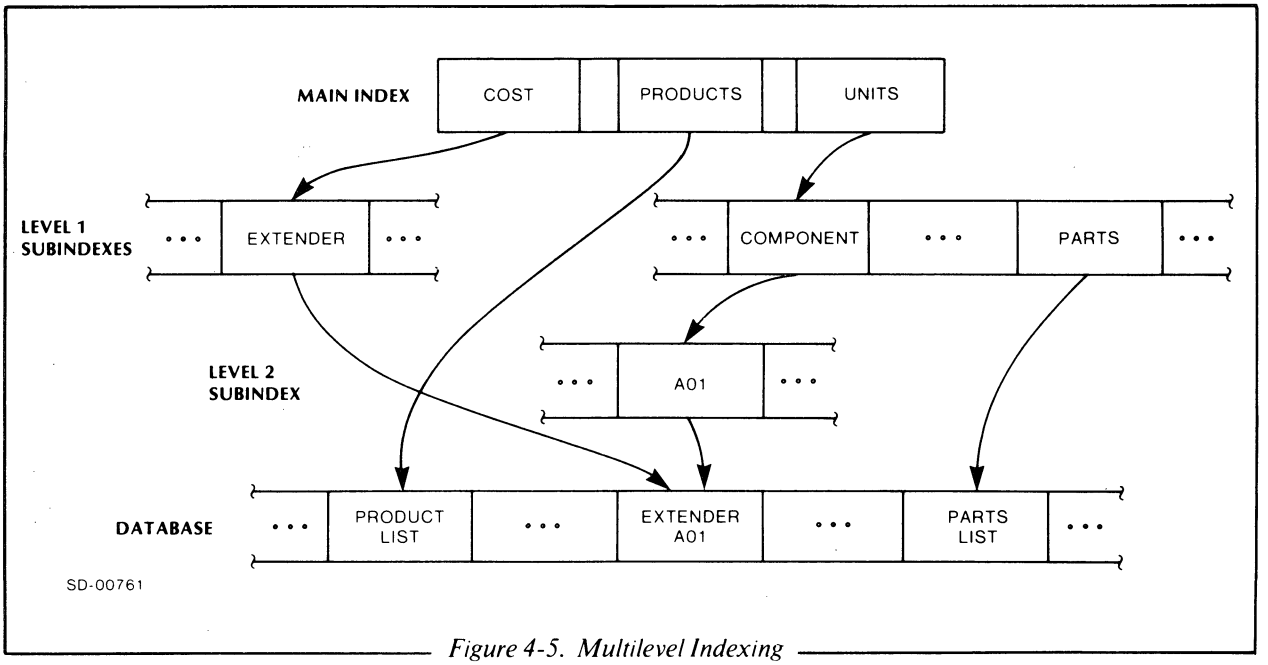


Figure 4-5. Multilevel Indexing

Sort/Merge Files

A *sort/merge file* can participate in a SORT and/or MERGE operation. You may not specify input/output operations for such a file. You declare the sort/merge file in an SD entry in the Data Division of your program. With a sort/merge file, you can use COBOL Procedure Division statements to sort one or more files (SORT), to combine two or more files (MERGE), to pass a record to the sort operation (RELEASE), and to retrieve a record from the sort or merge operation (RETURN).

File Access Modes

COBOL supplies you with three modes for getting at your data: sequential access, random access, and dynamic access. Table 4-1 illustrates the access modes available for the three file organizations, the operating systems which implement the access modes, and certain device considerations.

Sequential Access Mode

If you want to access data in the order in which you recorded it, you want to use the sequential access mode. The order of the file (which is the order you wrote the records when you first created the file) determines the order in which COBOL references the records.

You may specify the ACCESS IS SEQUENTIAL clause in the SELECT clause for files with sequential relative, or indexed organization.

Random Access Mode

The random access mode permits you to read and write any record in your file without accessing any other records. Because you read and write records according to relative record numbers or index key values, the sequence in which COBOL stores the records has nothing to do with the sequence in which you access them. Random access is the quickest and easiest access method.

You may specify the ACCESS IS RANDOM clause in the SELECT clause for a relative or indexed file.

Dynamic Access Mode

The dynamic access mode combines the sequential and random access modes and allows you to switch from one to the other by using the various forms of COBOL input/output statements.

You may specify the ACCESS IS DYNAMIC clause in the SELECT clause for a relative or indexed file.

Table 4-1. COBOL File Handling

	Access Mode			File Handling System				Device				
File Organization	**Sequential**	**Random**	**Dynamic**	**RDOS**	**RDOS INFOS**	**AOS**	**AOS INFOS**	**Disk**	**Tape**	**Line Printer**	**Inter-active Terminal**	**Card Reader**
Sequential	yes	no	no	no	yes	yes	no	yes	yes	yes	yes	yes
Relative	yes	yes	yes	no	yes	yes	no	yes	no	no	no	no
Indexed	yes	yes	yes	no	yes	no	yes	yes	no	no	no	no
Multilevel Indexed	no*	yes	no*	no	yes	no	yes	yes	no	no	no	no

* You may process a given index sequentially or dynamically, but your program must direct the movement between the indexes.

Input-Output Section

The Input-Output Section consists of two paragraphs which supply information needed to control the transmission and manipulation of data between external media and your object program. They are called the File-Control paragraph and the I-O-Control paragraph.

The File-Control Paragraph

In this paragraph you name the files you will use in your program and associate them with system devices or external files. The File-Control paragraph takes the format:

```
[ FILE-CONTROL.
      SELECT clauses ]
```

SELECT Clause

You must specify one SELECT clause for each of your files. In each clause, you name a file; you may also specify other file-related information. For each file you specify in this paragraph, you must include a file description entry in the Data Division (see Chapter 5).

The information you may provide in a SELECT clause depends on the organization of the file you are declaring. The SELECT clause takes one of four forms:

Format for a sequential file:

```
SELECT [ OPTIONAL ] fn   ASSIGN TO { { sysfn-1 [ VOLUME SIZE IS vsiz-1 [ CONTIGUOUS [[ NO ] INITIALIZATION ]]] }...
                                   { PRINTER [ sysfn-1, ... ]                                                    }

[ RESERVE buf { AREA  } ]   [ ORGANIZATION IS SEQUENTIAL ]   [ ACCESS MODE IS SEQUENTIAL ]
              { AREAS }

[ FILE STATUS IS fs ]   [ INFOS STATUS IS infs ]   [ PARITY IS { ODD  } ] .
                                                              { EVEN }
```

Format for a relative file:

```
SELECT fn   ASSIGN TO { sysfn-1 [ VOLUME SIZE IS vsiz-1 [ CONTIGUOUS [[ NO ] INITIALIZATION ]]] }...

[ RESERVE buf { AREA  } ]   ORGANIZATION IS RELATIVE
              { AREAS }

[ ACCESS MODE IS { SEQUENTIAL [ RELATIVE KEY IS relkey ]                  } ]   [ FILE STATUS IS fs ]   [ INFOS STATUS IS infs ].
                 { { RANDOM  }                                            }
                 { { DYNAMIC } RELATIVE KEY IS relkey                     }
```

Format for an indexed file:

```
SELECT fn   ASSIGN INDEX TO { sysfn-1 [ MERIT mer-1 ] [ VOLUME SIZE IS vsiz-1   [ CONTIGUOUS [[ NO ] INITIALIZATION ]]] } ...
     [ TEMPORARY ] [ SPACE MANAGEMENT ] [ ROOT MERIT IS rootmer ]  [ HIERARCHICAL | LRU ]
  [ ASSIGN DATA TO { sysfn-2 [ MERIT mer-2 ] [ VOLUME SIZE IS vsiz-2   [ CONTIGUOUS [[ NO ] INITIALIZATION ]]] } ...
     [ SPACE MANAGEMENT ]  [ RESERVE ind INDEX { AREA | AREAS } ] [ RESERVE buf DATA { AREA | AREAS } ]
  ORGANIZATION IS INDEXED  [ ACCESS MODE IS { SEQUENTIAL | RANDOM | DYNAMIC } ]
  [ ALTERNATE ] RECORD { KEY IS | KEYS ARE } { reckey-1 [ KEY LENGTH IS keylen-1 ]   [ WITH DUPLICATES [ OCCURRENCE IS occ-1 ]] } ...
  [ FILE STATUS IS fs ]   [ INFOS STATUS IS infs ]   [ ALLOW SUB-INDEX [ LEVELS IS lev ]]   [ KEY COMPRESSION ].
```

Format for a sort/merge file:

```
SELECT fn   ASSIGN TO sysfn-1,  ... .
```

Because the SELECT clause is so complex, we present it by discussing each of its clauses separately. You may specify all of the major clauses in any order, with one exception: the ASSIGN clause must occur where shown. Table 4-2 lists all of the major SELECT clauses, along with any subordinate clauses. It also tells which file organizations use the clause, what the default is if you omit the clause, and any restrictions on the clause's use. *Pay particular attention to the restrictions column because many of the clauses differ in use from operating system to operating system.* If a clause is specified as functioning under only one operating system, you may still specify it for the other, but it will have no effect.

You may not specify any of the SELECT clause data items in the File Section or Linkage Section of the Data Division.

In the SELECT clause, you may qualify any reference to a data item of any form, but you may not subscript it.

Table 4-2. SELECT Clause

Clauses and Options	Sequential Files	Relative Files	Indexed Files	Sort/Merge Files	Default	Restrictions
OPTIONAL	X					File must be open for input.
ASSIGN	X	X	X	X		
MERIT			X		Merit factor equals 0.	RDOS only.
VOLUME SIZE	X	X	X		COBOL will allocate space as needed (up to 65,535 blocks).	
CONTIGUOUS	X	X	X		COBOL will allocate blocks randomly.	
INITIALIZATION	X	X	X		Initialization.	RDOS only.
PRINTER	X					If you do not specify a disk file *(sysfn)*, you may only have as many PRINTER files as you have printers.
TEMPORARY			X		Index is permanent.	RDOS only.
SPACE MANAGEMENT			X		No space management.	
ROOT MERIT			X		COBOL assigns root node priority.	RDOS only.
HIERARCHICAL/LRU			X		Hierarchical.	RDOS only; not for database file.
RESERVE (DATA)	X	X	X		COBOL assigns 1 buffer.	RDOS only; you must specify at least 1 buffer.
RESERVE (INDEX)			X		COBOL assigns 2 buffers.	RDOS only; you must specify at least 2 buffers.
ORGANIZATION	X	X	X		Sequential.	

Table 4-2. SELECT Clause (continued)

Clauses and Options	Sequential Files	Relative Files	Indexed Files	Sort/Merge Files	Default	Restrictions
ACCESS	X	X	X		Sequential.	
RELATIVE KEY		X			Current record pointer determines current record for sequential access. This clause is not optional for dynamic or random access.	
RECORD KEY IS			X		Not optional.	
KEY LENGTH			X		COBOL assigns a maximum key length equal to the longest key specified for the file.	Not for a database file.
DUPLICATES			X		COBOL sends an error message if you attempt to write a key that already exists.	
OCCURRENCE			X		None	
ALTERNATE RECORD						
KEY IS			X		If your file has alternate record keys, this clause is mandatory.	You must specify all alternate record keys associated with your file. The key items must be stored in the records of your file.
KEY LENGTH			X		COBOL assigns a maximum key length equal to the longest key specified for the file.	Not for a database file.
DUPLICATES			X		COBOL sends an error message if it encounters a duplicate alternate key.	
OCCURRENCE			X		None.	

Table 4-2. SELECT Clause (continued)

Clauses and Options	Sequential Files	Relative Files	Indexed Files	Sort/Merge Files	Default	Restrictions
FILE STATUS	X	X	X		You cannot check the file status after I/O processing.	
INFOS STATUS	X	X	X		You cannot check for an INFOS/operating system error after I/O processing.	
PARITY	X				Parity is odd.	For RDOS magnetic tape files only.
ALLOW SUBINDEX			X		No subindexing allowed.	You cannot specify this option if you have alternate record keys.
LEVELS			X		Maximum number of levels is 1.	
KEY COMPRESSION			X		Redundant key information will be stored.	RDOS only.

OPTIONAL Clause

```
[ OPTIONAL ] fn
```

Where:

fn is a symbolic name that specifies the name you use to reference the file in your program.

You may specify the OPTIONAL clause only for sequential files OPENed for input. This clause is required for a file that may not necessarily be present each time you execute your object program. If you specify this clause and the file is not present at execution time, the first READ executed for the file signals an end-of-file condition (see the section "I/O Exception Conditions" in Chapter 6).

ASSIGN Clauses

```
ASSIGN INDEX TO { sysfn-1 [ MERIT mer-1 ] [ VOLUME SIZE IS vsiz-1 [ CONTIGUOUS [[ NO ] INITIALIZATION ]]] } ...
    [ TEMPORARY ] [ SPACE MANAGEMENT] [ ROOT MERIT IS rootmer ] [ HIERARCHICAL | LRU ]

[ ASSIGN DATA TO { sysfn-2 [ MERIT mer-2 ] [ VOLUME SIZE IS vsiz-2 [ CONTIGUOUS [[ NO ] INITIALIZATION ]]] } ...
    [ SPACE MANAGEMENT ] ]
```

Where:

sysfn is an alphanumeric literal or an alphanumeric or alphabetic data item whose value, when you OPEN the file, specifies the system file containing the data records and/or index entries of the logical COBOL file *fn*.

mer is a positive integer literal that specifies the priority of a volume.

vsiz is a positive integer literal that specifies a number of blocks.

rootmer is a positive integer literal that specifies which volume priority has the highest level root node.

The ASSIGN clause is composed of several subordinate clauses. Its purpose is to associate the file *fn* with a storage medium and say something about its physical makeup.

You must specify at least one symbolic name *(sysfn)* to identify your file. Because COBOL files are organized in logical volumes, you must specify a symbolic name for each volume of your file. The order in which you specify the volumes determines their logical order within the file. The name you specify for the first volume of the file becomes the file's symbolic name. If you do not specify more than one name, COBOL assumes the file has only one volume.

For sequential and relative files, you only need to specify the ASSIGN clause for volumes of the file's database. However, if you intend to open the file for output, that is, if you created the file as an entirely new indexed file or as a new inversion of an existing indexed file, you must also specify the ASSIGN DATA clause with the data record file system names.

You may indicate the maximum number of blocks you want a volume to attain by specifying the VOLUME SIZE clause. If you omit this clause, the operating system will allocate blocks to your file as the need arises. How volume size is determined depends on the operating system. (See Appendix B.) It is a good idea to specify CONTIGUOUS for the first (or only) volume of the file. If you do so, the operating system will allocate a group of contiguous blocks on disk (you must specify the number) for this volume, which will speed up processing. You can then default your choice on subsequent volumes and let the system allocate disk space randomly. If you specify INITIALIZATION, COBOL will clear (set to zero) all information from the blocks reserved for this file when you open it for output. If you specify NO INITIALIZATION, no action is taken and the information in those blocks remains. If you omit this option, INITIALIZATION is the default.

For sequential files, you may specify the PRINTER clause to indicate that the file is a print file. Whenever you write data using this file, the output goes to the line printer or the file associated with the *sysfn* you specify. For specific information on print file formatting, see Chapter 6.

For indexed files, you may specify the MERIT phrase for any volume you wish to give a particular processing priority. For example, a volume may reside on a fast storage device and you, therefore, will want to use it for those entries you reference most often. The larger the value of the merit factor item *(mer)* is, the higher the volume's priority will be. If you omit this clause, the default is 0. Several volumes may have the same priority.

If you want to use an index of an indexed file to store data for only the duration of your program's current run, you may specify TEMPORARY. This will tell the system not to increment the counters which keep track of the index entries referencing each data record every time you add a record to this temporary inversion. Thus, you can easily delete this inversion without adjusting any counters, which makes the deletion process much faster.

You may specify the SPACE MANAGEMENT clause for either the index or the database file (or both) of an indexed file. It simply tells INFOS to reuse the space left after you have deleted a record. This allows you to better use available space, keeps your files in better shape from a maintenance viewpoint, and allows your files to grow and shrink as necessary.

If your indexed file is made up of several volumes, each having a different processing priority, you may want to assign a priority to the root node of a particular volume. Generally the system cannot store all an index's nodes in memory when it processes an indexed file; INFOS must remove some of them from time to time. By specifying the ROOT MERIT clause and establishing a high priority node, you ensure that the node will not be randomly selected for removal (0 is the highest priority).

Because COBOL lets you request any number of buffers, it uses two techniques for buffer management. If you specify LRU (Least Recently Used) and all buffers are in use when you issue a read or write request, COBOL writes the contents of the buffer accessed least recently to the file, and places the current data in the buffer. If you specify HIERARCHICAL or omit this clause, the system uses the lowest node in the index structure in the same manner.

RESERVE Clause

Where:

ind is an integer literal that specifies the number of input/output buffers you want to allocate for your file's indexes.

buf is an integer literal that specifies the number of input/output buffers you want to allocate for your file's database.

The RESERVE clause allows you to indicate the number of input/output buffers you want to allocate for the database of your file. In addition, if your file is an indexed file, you may specify the number of input/output buffers you want allocated for the indexes of your file. (Remember, an INFOS indexed file is essentially two files: one for the indexes (keys) and one for the database.) The minimum number of input/output buffers you may specify for a database is 1 and the minimum for indexes is 2. If you omit this option, INFOS automatically assigns a number of input/output buffers, which could reduce data efficiency.

ORGANIZATION Clause

```
ORGANIZATION IS { SEQUENTIAL }
                { RELATIVE   }
                { INDEXED    }
```

The ORGANIZATION clause specifies the logical structure of your file. When you create a file, you establish its organization and cannot later change it. If you omit this option, the default is SEQUENTIAL.

ACCESS MODE Clause

```
[ ACCESS MODE IS { SEQUENTIAL [ RELATIVE KEY IS relkey ]                    } ]
                 { { RANDOM  } RELATIVE KEY IS relkey                       }
                 { { DYNAMIC }                                              }

[ ALTERNATE ] RECORD { KEY IS  } { reckey-1 [ KEY LENGTH IS keylen-1 ] [ WITH DUPLICATES [ OCCURRENCE IS occ-1 ]] } ...
                     { KEYS ARE }
```

Where:

relkey is an unsigned integer data item that specifies the relative record number of a key.

reckey is an alphabetic, alphanumeric, or unsigned numeric data item that specifies the name of a prime record key or an alternate record key in an indexed file.

keylen is a positive integer literal or an unsigned integer data item that specifies a key length. It serves several functions which we discuss later in this section.

occ is an unsigned integer data item that receives an occurrence number.

The ACCESS MODE clause determines the manner (sequential, random, or dynamic) in which COBOL will access records from your file. If you omit this option, the default is SEQUENTIAL.

When you specify SEQUENTIAL access, COBOL accesses the records in your file in the sequence determined by the file's organization. In a sequential file, you establish the record sequence by the order in which you write the records when you create or extend the file. In a relative file, the sequence is determined by the ascending relative record numbers of the existing records in your file. In an indexed file, the sequence is determined by the ascending record key values within a given index or subindex.

If you specify RANDOM access for a relative file, COBOL accesses the record with the value of the relative key data item *(relkey)*. When randomly accessing an indexed file, COBOL accesses the record with the same value as the record key data item *(reckey)*.

You only need to specify one record key for a simple indexed file. For a multilevel indexed file, you may specify as many record key items as required to make the various references in the program.

If you specify DYNAMIC access for a relative or indexed file, COBOL may access the file sequentially and/or randomly.

For example, if your program needs a reference of the form:

READ *fn* KEY *reckey-1, reckey-2*

then you must specify at least two record key items. However, if all the references in your program have the form:

READ *fn [position phrase]* KEY *reckey*

then you need to specify only one record key item, even though there may be many levels of indexing.

COBOL makes no association between the particular key items and the levels of the index. You may use any of the key items to specify a key in any level of the index.

If you want to provide an alternate access path to records in an indexed file (using an ANSI standard approach rather than the more general INFOS independent inversion technique), you must observe the following rules when specifying the ALTERNATE RECORD KEYS clause:

1. You may specify only *one* record key, called the prime record key. It need not be contained in the file's record area.

2. You must specify *all* alternate record keys associated with the file, whether your program uses them or not. The alternate key data items must be contained in the records of the file. You define the number and the location of the alternate keys within the records when you create the file; you cannot change this after file creation.

You may specify a KEY LENGTH clause for any record or alternate record key. If you do specify this clause, the key length items *(keylen)* have several functions depending on the context in which they appear:

1. When you OPEN a file for output, the value of *keylen* (for the first *reckey)* at that time is the maximum key length for the main level of that file's index.

2. When you specify a WRITE statement, the value of *keylen* at that time represents the number of characters (leftmost) in *reckey* which will be stored as the value of that record's index.

3. When you specify a READ statement whose KEY LENGTH clause contains a GENERIC clause, the value of *keylen* at that time represents the number of characters (leftmost) in *reckey* which must be matched in order to have access to a given record.

If you want to use one key to access a group of data records, you may specify the WITH DUPLICATES clause for the record key or alternate record key. If you specify this clause, you can execute a WRITE statement that references the associated key item *(reckey)* even though there may already be a record in the file with that same key. If you specify the OCCURRENCE clause, the system automatically assigns occurrence numbers to your keys. The manner in which COBOL makes the assignments depends on your operating system. See Appendix B for a discussion of the differences.

After executing a WRITE statement, COBOL updates the value of the occurrence number data item *(occ)* to contain the occurrence number of the last record written which had the specified key value. You can obtain the value of the occurrence number and key length data items by issuing a RETRIEVE KEY statement in the Procedure Division.

FILE STATUS and INFOS STATUS Clauses

[FILE STATUS IS *fs*] [INFOS STATUS *infs*]

Where:

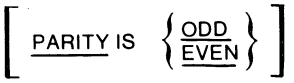
fs is a two-character alphanumeric data item that receives the COBOL file status item.

infs is an alphanumeric data item that receives the INFOS file status item which you must define in the Working Storage Section of the Data Division (the size of this data item is system dependent - see Appendix B.)

If you specify the FILE STATUS clause, you establish a data item *(fs)* which COBOL updates to indicate the completion status of each I/O statement's execution. The values returned are described in the section "Handling I/O Exception Conditions" in Chapter 6.

If you specify the INFOS STATUS clause, you also set up a data item *(infs)* which COBOL updates to contain the error code that INFOS returned on completion of each I/O statement's execution. INFOS clears the data item on a normal return.

PARITY Clause



You may specify the PARITY clause only for sequential files stored on magnetic tape. It identifies the parity of the data recorded in the file. Odd parity is the default; even parity is rarely used.

ALLOW SUB-INDEX and LEVELS Clauses

[ALLOW SUB-INDEX [LEVELS IS *lev*]]

Where:

lev is a positive integer literal that specifies the maximum number of index or subindex levels the file will have.

In a multilevel indexed file, you may want to break your indexed structure into more manageable, smaller units of records. You accomplish this by building subindexes within your index structure. Subindexing allows you to structure your file by establishing logical relationships between records.

You must specify ALLOW SUB-INDEX for any indexed file which already has subindexing or for which you are going to define subindexing. You cannot specify this statement if you specified alternate record keys.

You specify the LEVELS clause when you are creating an indexed file and you want to indicate the expected maximum number of index and subindex levels that the file will have. If you do not specify this clause when you create the file, the maximum number of levels is assumed to be 1.

KEY COMPRESSION Clause

[KEY COMPRESSION]

If you have several keys with identical leading characters, you can save room in your index by specifying the KEY COMPRESSION clause. Let us say you have several keys whose first seven characters are the same. If you specify KEY COMPRESSION, INFOS automatically stores only the unique part of your keys in the appropriate records; it stores only one entry of the identical, 7-character part. For a very large file, you can save a lot of space using this clause.

SELECT Clause Examples

Example 1:

```
FILE-CONTROL.
     SELECT FILE1
          ASSIGN TO "SET21"
          ORGANIZATION IS SEQUENTIAL
          ACCESS IS SEQUENTIAL.
```

Example 2:

```
FILE-CONTROL.
     SELECT FILE2
          ASSIGN TO PRINTER.
```

Example 3:

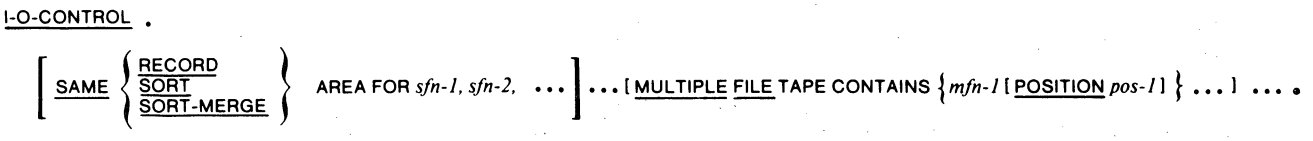
```
FILE-CONTROL.
     SELECT FILE3
          ASSIGN TO "SET25"
          ORGANIZATION IS RELATIVE,
          ACCESS IS RANDOM,
            KEY IS KEY05.
```

Example 4:

```
FILE-CONTROL.
     SELECT FILE4
          ASSIGN INDEX TO "MYINDEX"
          ASSIGN DATA TO "MYDATA"
          ORGANIZATION IS INDEXED
          ACCESS MODE IS DYNAMIC
          RECORD KEYS ARE
            RS-GRADE KEY LENGTH IS W1-LENG
            RS-WIDTH WITH DUPLICATES,
              OCCURRENCE IS W2-OCCR
          STATUS IS RS-STATUS
          INFOS STATUS IS RS-INFOS
          ALLOW SUB-INDEX, LEVELS IS 2.
```

The I-O-Control Paragraph

You use the I-O-CONTROL paragraph to specify certain relationships among your files. It takes the form:



Where:

sfn is a symbolic name that specifies a file.

mfn is a symbolic name that specifies a labeled tape file or a system file specifier that specifies an unlabeled tape file.

Note that a period is required at the end of this paragraph.

Only the SAME RECORD AREA clause is necessary in the ECLIPSE COBOL system, because the system itself automatically provides efficient memory management, including buffer reuse whenever possible. Other forms are redundant and COBOL ignores them.

The SAME RECORD AREA clause specifies that the record area declared for the files *sfn-2* to *sfn-n* begin at the same character position in memory as those declared for file *sfn-1*, thereby redefining the record area declared for file *sfn-1*.

Any or all of these files may be open at the same time. You must specify at least two files in each SAME RECORD AREA clause. However, they need not have the same organization or access.

The MULTIPLE FILE clause is never required in ECLIPSE COBOL and, when specified, is ignored. If more than one file is stored on a single reel of tape, and the tape is labeled, you identify the files by name in the ASSIGN clause of the SELECT clause. If the tape is unlabeled, you identify the files by system file specifiers in the ASSIGN clause.

End of Chapter

Chapter 5
The Data Division

Structure

The Data Division describes the data that your program will accept as input, and will create, manipulate, or produce as output. You may specify that data belongs to a file or direct COBOL to store it in working storage. You may also request that data be formatted for output. Or you may simply specify data as constants for COBOL to use in calculations.

The Data Division consists of three sections: the File Section, the Working Storage Section, and the Linkage Section.

The *File Section* defines the structure of each of your data files by a file description entry (an SD entry for sort/merge files, an FD entry for all other file types) and one or more record descriptions. Descriptions of any subordinate data items follow these.

The *Working Storage Section* describes the records and subordinate data items that COBOL will develop and process internally. These elements function as general, temporary, data storage items during the execution of your program. They are not part of external files.

You specify the *Linkage Section* if your program is a subprogram under the control of a CALL statement which contains a USING phrase. This section may contain one or more record and subordinate data item descriptions. However, no space is allocated in the program for data items referenced by data names in the program's Linkage Section. The system resolves this at runtime by equating the reference in the called program to the location used in the calling program. Linkage Section data items receive the parameters passed by the calling program.

The Data Division has the following format:

```
DATA DIVISION.

[ FILE SECTION.
      FD and SD entries with associated record descriptions ]

[ WORKING-STORAGE SECTION.
      working storage section body ]

[ LINKAGE SECTION.
      linkage section body ]
```

File Section

The File Section consists of FD and SD entries containing record descriptions which define the storage areas COBOL uses to transfer data to and from files. An FD or SD entry continues a file declaration that the SELECT clause began. You must specify one FD or SD entry for each file you declared in a SELECT clause. You specify an FD entry for files upon which you want COBOL to perform input/output operations. You specify an SD entry for sort/merge files, that is, files that are the object of a SORT or a MERGE statement.

One or more record descriptions (01-level data descriptions) must follow each FD or SD entry. These record descriptions define the record area for the file. If you specify more than one record description for a file, they all implicitly redefine the same physical record area. The length of the record area is the length of the longest record you defined for the file. We discuss record descriptions in detail later on in this chapter.

File Description Entry

A file description entry may take one of four forms.

Format for a sequential file:

```
FD fn [ BLOCK CONTAINS [ blk-1 TO ] blk-2 { RECORDS | CHARACTERS } ] [ RECORD CONTAINS [ rsiz-1 TO ] rsiz-2 CHARACTERS ]

[ RECORDING MODE IS { FIXED
                    | { VARIABLE | DATA-SENSITIVE [ DELIMITER IS del ] } [ RECORD LENGTH IS dlen ]
                    | UNDEFINED } ]

[ LABEL { RECORD IS | RECORDS ARE } { STANDARD [ lev ] | EBCDIC [ lev ] | OMITTED } ]

[ VALUE OF [ OWNER IS vown ]
           [ EXPIRATION DATE IS vexp ]
           [ SEQUENCE NUMBER IS vseq ]
           [ GENERATION NUMBER IS vgen ]
           [ ACCESSIBILITY IS vacc ]
           [ OFFSET IS voff ]
           [ VOLUME STATUS IS vstat ]
           [ USER VOLUME { LABEL IS | LABELS ARE } uvl-1, ... ]
           [ USER HEADER { LABEL IS | LABELS ARE } uhl-1, ... ]
           [ USER TRAILER { LABEL IS | LABELS ARE } utl-1, ... ] ]

[ DATA { RECORD IS | RECORDS ARE } rec-1, ... ] [ LINAGE IS body LINES [ WITH FOOTING AT foot ]
                                                  [ LINES AT TOP top ] [ LINES AT BOTTOM bot ] ]

[ CODE-SET IS { ASCII | STANDARD-1 | NATIVE | EBCDIC | alph }
              [ { FIELD IS | FIELDS ARE } fld-1, ... ] ] [ FEEDBACK IS fbk ] [ PAD CHARACTER IS pad ].

01 rec-1, ...
```

Format for a relative file:

```
FD fn [ BLOCK CONTAINS [ blk-1 TO ] blk-2  { RECORDS / CHARACTERS } ]

[ RECORD CONTAINS [ rsiz-1 TO ] rsiz-2 CHARACTERS ]  [ RECORDING MODE IS FIXED ]

[ LABEL { RECORD IS / RECORDS ARE } { STANDARD / OMITTED } ]  [ DATA { RECORD IS / RECORDS ARE } rec-1, ... ]

[ FEEDBACK IS fbk ]  [ PAD CHARACTER IS pad ].

01 rec-1, ...
```

Format for an indexed file:

```
FD fn [ INDEX BLOCK CONTAINS [ iblk-1 TO ] iblk-2 CHARACTERS ]  [ DATA BLOCK CONTAINS [ blk-1 TO ] blk-2 { RECORDS / CHARACTERS } ]

[ INDEX NODE SIZE IS nsiz CHARACTERS ]  [ RECORD CONTAINS [ rsiz-1 TO ] rsiz-2 CHARACTERS ]

[ RECORDING MODE IS VARIABLE [ RECORD LENGTH IS dlen ] ]

[ LABEL { RECORD IS / RECORDS ARE } { STANDARD / OMITTED } ]  [ DATA { RECORD IS / RECORDS ARE } rec-1, ... ]

[ FEEDBACK IS fbk ]  [ MERIT IS recmer ]  [ PARTIAL RECORD IS partrec ].

01 rec-1, ...
```

Format for a sort/merge file:

```
SD fn [ BLOCK CONTAINS blk-1 TO blk-2 { RECORDS / CHARACTERS } ]  [ RECORD CONTAINS [ rsiz-1 TO ] rsiz-2 CHARACTERS ]

[ DATA { RECORD IS / RECORDS ARE } rec-1, ... ].

01 rec-1, ...
```

In all formats, *fn* is a symbolic name that specifies the name of the internal file you specified in the file's SELECT clause; and *rec* is the name of an 01-level data record.

We will present the file description entry by separately discussing each of its clauses. You may specify all of the major clauses in any order you choose. Table 5-1 lists all of the major file description entry clauses along with any subordinate clauses. It also presents the file organizations which use the clause, the default if you omit the clause, and any restrictions on the clause's use. Pay particular attention to the restrictions column because many of the clauses differ in use from operating system to operating system. *If the table lists a clause as functioning under only one operating system, you may still specify it for the other, but it will have no effect.*

In the file description entry, you may qualify any reference to a data item of any form, but you may not subscript it.

Table 5-1. FD Entry Clauses

Clauses and Options	Sequential Files	Relative Files	Indexed Files	Sort/Merge Files	Default	Restrictions
BLOCK CONTAINS	X	X	X	X	Maximum size is one block.	
INDEX NODE SIZE			X		System calculates size.	
RECORD CONTAINS	X	X	X	X	None.	
RECORDING MODE	X	X	X		FIXED for sequential and relative files; VARIABLE for indexed files.	
RECORD LENGTH	X	X			Maximum length is that of the file's record area.	
DELIMITER	X				Delimiters are CR, NL, FF, and NULL.	
LABEL RECORDS	X	X	X		ANSI Standard.	For sequential magnetic tape files only.
VALUE OF						
OWNER	X				None.	For labeled magnetic tape files only.
EXPIRATION DATE	X					
GENERATION NUMBER	X					
ACCESSIBILITY	X					
OFFSET	X					
VOLUME STATUS	X					

Table 5-1. FD Entry Clauses (continued)

Clauses and Options	Sequential Files	Relative Files	Indexed Files	Sort/Merge Files	Default	Restrictions
USER VOLUME LABEL	X				None.	For magnetic tape files only.
USER HEADER LABEL	X					
USER TRAILER LABEL	X					
DATA RECORDS	X	X	X	X	None.	Clause is ignored.
LINAGE	X				No automatic formatting provided.	File must be a sequential PRINTER file.
FOOTING	X				No footing area.	
TOP	X				No top margin.	
BOTTOM	X				No bottom margin.	
CODE-SET	X				ASCII	
FIELD IS	X				All data is translated.	
FEEDBACK			X		Cannot access feedback information.	
MERIT			X		Merit factor is 0.	RDOS only.
PARTIAL			X		None	
PAD	X	X			Pad character is null.	RDOS only.

Block, Node, and Record Sizes

[INDEX BLOCK CONTAINS [*iblk-1* TO] *iblk-2* CHARACTERS] [DATA BLOCK CONTAINS [*blk-1* TO] *blk-2* { RECORDS / CHARACTERS }]

[INDEX NODE SIZE IS *nsiz* CHARACTERS] [RECORD CONTAINS [*rsiz-1* TO] *rsiz-2* CHARACTERS]

Where:

iblk is a positive integer literal that specifies the maximum number of characters which a logical block in an indexed file may contain.

blk is a positive integer literal that specifies the maximum number of characters or records which a logical block in a data file may contain.

nsiz is a positive integer literal that specifies the number of characters in an index node.

rsiz is a positive integer literal that specifies the number of characters in a data record.

The BLOCK CONTAINS clause specifies the maximum size, in characters, of a data file's logical block. COBOL always ignores *blk-1*. If you specify RECORDS, COBOL calculates the block size as *blk-2* times the maximum length of the file's record area. If you specify CHARACTERS, the block size is equal to the number of characters specified in *blk-2*. If you omit this option, the default size is one record per block. The function of the BLOCK CONTAINS clause differs from operating system to operating system. (See Appendix B.)

If the file is a PRINTER file, and you want to later do a CLI "transfer" of that file, you must specify BLOCK CONTAINS 512 CHARACTERS. This requirement is system dependent. (See Appendix B.)

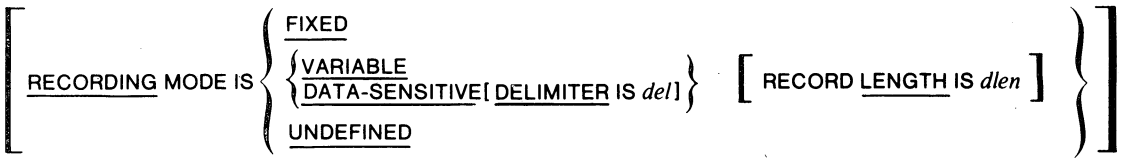
The INDEX BLOCK CONTAINS clause specifies the size, in characters, of your index file's logical block. COBOL always ignores *iblk-1*. The block size is equal to the number of characters specified in *iblk-2*. If you omit this option, the default size is one block.

For both of the above clauses, if you are processing disk files, the system will transfer your data in multiples of 512 characters. For other devices, if you don't specify a size, the system will use 80 characters. But for the line printer, the default is 132 characters.

The INDEX NODE SIZE clause specifies the size, in characters, of an index node. The node size must be large enough to hold three keys. If you omit this option, the system calculates the size according to the maximum key length, the partial record length, and whether or not you allow subindexing. You define subindex node sizes in the DEFINE SUB-INDEX statement (see Chapter 7).

Because you define the size of each data record within the record description entry, COBOL never requires the RECORD CONTAINS clause to specify the data record's size. However, you may use it if you want COBOL to check that the data records you define for this file fall within the range *rsiz-1* through *rsiz-2,* inclusive.

RECORDING MODE Clause



Where:

del is an alphanumeric literal which specifies a character that delimits the end of a record.

dlen is an integer data item that either specifies or receives a number of characters (see the following paragraphs).

The RECORDING MODE clause specifies the record format used in the file. There are four different record formats available through COBOL:

- fixed-length,
- variable-length,
- data-sensitive, and
- undefined-length.

If you specify FIXED, all records will have the same number of characters, the length of which is determined by the size of the file's record area.

If you specify VARIABLE, you must specify a maximum length for the records in the RECORD LENGTH clause. No two records in the file need to be the same length. However, they may not exceed the maximum length *(dlen)*, and they must never be less than 1. The system keeps track of both the maximum and the actual length of your record. VARIABLE is the only record format you may use with simple and multilevel indexed files.

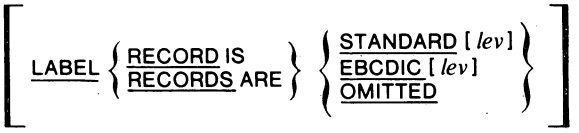
If you specify DATA-SENSITIVE, the length of a record is determined by the occurrence of some special character(s) *(del)*. If you do not specify a delimiter character, the default characters are carriage return, line feed, null, or new line. You may also set a maximum length for a data-sensitive record by specifying the RECORD LENGTH clause. You do not count the delimiter as part of the record.

If you specify UNDEFINED, you may treat your file as a sequence of bytes rather than a sequence of records with a specific length. In this way, you may append new data onto the end of the file or read any section within the file, regardless of the individual records' size. For each READ operation, COBOL reads one physical block and returns the number of characters read.

In all appropriate cases, if you omit the RECORD LENGTH clause, a record may not be more than the length of the file's record area. If you do specify this clause, COBOL will store the number of characters read on record input in *dlen*. On output in variable record format, *dlen* will specify the number of characters you want to write. On output in data-sensitive record format, *dlen* specifies the maximum number of characters you want to write and receives the number actually written.

If you omit the RECORDING MODE clause, the default is FIXED for sequential and relative files and VARIABLE for indexed files.

LABEL RECORDS Clause

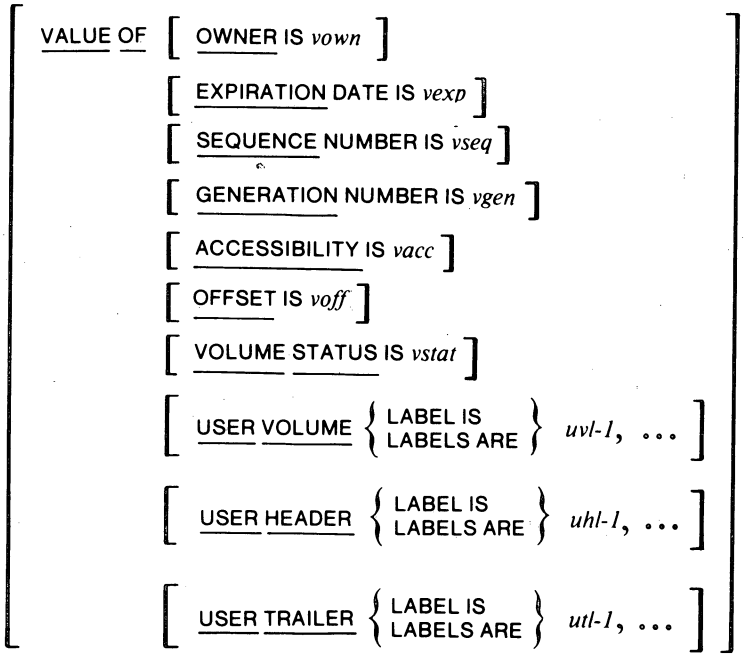


Where:

lev is a positive integer literal that indicates the level of the label you specify.

For magnetic tape files (which must be sequential), the LABEL RECORDS clause specifies the kind of labels you use for the file. COBOL supports ANSI Standard tape labels in levels 1 and 3, and IBM format tape labels in levels 1 and 2. STANDARD means ANSI Standard labels; EBCDIC means IBM format labels. The default levels *(lev)* are the highest for each type (3 for STANDARD, 2 for EBCDIC). Specifying the highest level allows you to read any level on input. If you specify OMITTED, all records in the file are database records. For all files except magnetic tape files, COBOL ignores the LABEL RECORDS clause. If you omit this clause for a magnetic tape file, the default is STANDARD.

VALUE OF Clause



Where:

vown is a 14-character alphanumeric data item that specifies the owner identification.

vexp is a 6-character data item that specifies the date after which the file is no longer valid. It takes the form BYYDDD, where B is a space, YY is the year-in-century, and DDD is the day of the year.

vseq is a 2-digit unsigned numeric DISPLAY data item that specifies the number of a volume in a volume set.

vgen is a 4-digit unsigned numeric DISPLAY data item that specifies a number to identify this file among successive generations of the file.

vacc is a 1-character alphanumeric data item that specifies the restrictions placed on access to the file.

voff is a 2-digit unsigned numeric DISPLAY data item that specifies the number of characters you want ignored at the beginning of each block.

vstat is a 4-character alphanumeric data item that receives the current status of volume and label processing.

uvl is a 76-character alphanumeric data item that specifies volume label information.

uhl is a 76-character alphanumeric data item that specifies file label header information.

utl is a 76-character alphanumeric data item that specifies file trailer label information.

You use the VALUE OF clause to retrieve or write some of the information stored in the labels of a labeled magnetic tape file. For all other files, COBOL ignores this option.

The following options are system dependent (see either the *RDOS INFOS System User's Manual* or the *AOS Programmer's Manual):*

OWNER
EXPIRATION DATE
SEQUENCE NUMBER
GENERATION NUMBER
ACCESSIBILITY
OFFSET
USER VOLUME
USER HEADER
USER TRAILER

Note that you specify volume and file identifiers for labeled magnetic tape files in the ASSIGN clause of the file's SELECT clause.

The VOLUME STATUS clause receives a 0 or a 1 in each character position of *vstat* following the execution of every READ or WRITE statement, depending on whether the following conditions are false or true, respectively:

Character 1: Volume change indicator. A transition from one volume to another has been made. The next record accessed will be in the new volume.

Character 2: The system has processed a user trailer label.

Character 3: The system has processed a user header label.

Character 4: The system has processed a user volume label.

Volume transition and user label processing require no program or operator intervention, but you may monitor the VOLUME STATUS item to give operator instructions or to modify the contents of the label items that will be written on each volume.

DATA RECORD Clause

COBOL always ignores this clause.

LINAGE Clause

```
[ LINAGE IS body LINES    [ WITH FOOTING AT foot ]    [ LINES AT TOP top ]    [ LINES AT BOTTOM bot ] ]
```

Where:

body is a positive integer literal or an unsigned integer data item that specifies the number of lines in the page body.

foot is a positive integer literal or an unsigned integer data item that specifies the line number in the page body where the footing area begins.

top is a positive integer literal or an unsigned integer data item that specifies the number of lines in the top margin of the logical page.

bot is a positive integer literal or an unsigned integer data item that specifies the number of lines in the bottom margin of the logical page.

The LINAGE clause defines the format of a logical page in the file *fn*. If you specify this clause, COBOL assumes that the file is a print file, even if you did not specify PRINTER in the file's SELECT clause in the Environment Division.

The LINAGE clause consists of four specifications:

- the body *(body)*,
- the footing *(foot)*,
- the top *(top)*, and
- the bottom *(bot)*.

COBOL calculates the total logical size of a page as *top* + *body* + *bot*. No correspondence is established between a logical page and the physical size of the forms you are printing.

The size you specify for the *body* includes the lines which COBOL may write or explicitly space per page by executing ADVANCING clauses (which appear in output procedure statements).

COBOL automatically spaces the top and bottom margins according to what you specify in *top* and *bot*. The value of an unspecified margin (either top or bottom) is zero.

The footing area of a page is generally the last few lines in the page (excluding the bottom margin). It is usually used for printing end-of-page information (e.g., page numbers, footnotes, etc.). The value you specify for *foot* must be greater than zero, but less than or equal to *body*. If you omit this clause, COBOL does not create a footing area for any page in the file.

For example, if you specify the clause

```
LINAGE 7, FOOTING 5, TOP 2, BOTTOM 2
```

COBOL outputs what is shown in Figure 5-1.

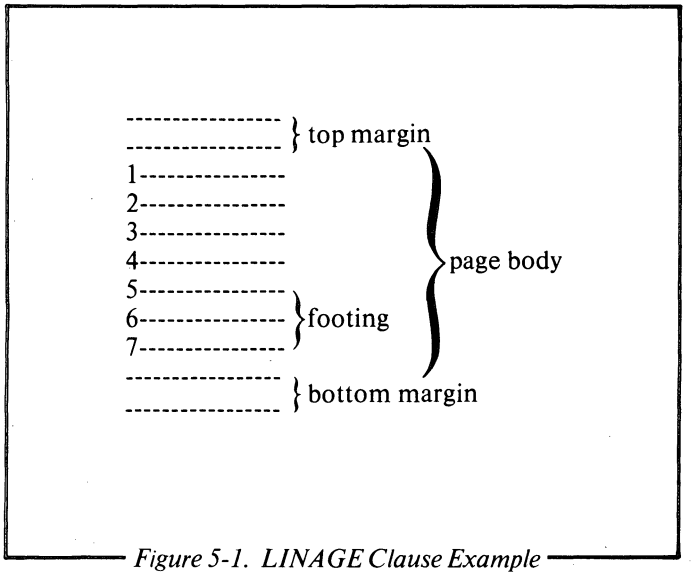
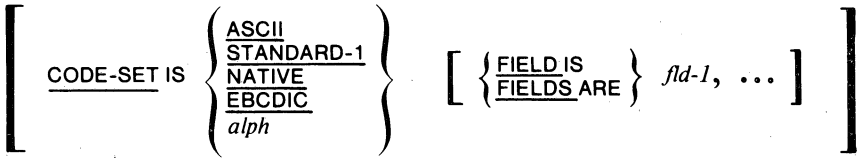


Figure 5-1. LINAGE Clause Example

If you omit the LINAGE clause, COBOL does not provide automatic formatting; it outputs characters as specified by the I/O operations. For more information on formatting, see the section "Print File Formatting" in Chapter 6.

CODE-SET Clause



Where:

alph is an alphabet name indicating that this file will use the collating sequence associated with the *alph* you defined in the Special-Names Paragraph of the Environment Division.

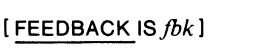
fld is an alphabetic, alphanumeric, unsigned numeric, or group data item that is contained in one of the record descriptions you declared for the file and that specifies the data fields you want translated.

The CODE-SET clause specifies the code set you want COBOL to use for a sequential file. If you omit this clause, COBOL will use the ASCII code (as specified in Appendix E). This is the code which the file system normally uses and which ECLIPSE COBOL programs use internally. ASCII, STANDARD-1, and NATIVE also indicate that the system will record the file in ASCII code. EBCDIC means that the system will record the file in EBCDIC code (see Appendix F). If you specify EBCDIC, COBOL will translate data to or from ASCII when the COBOL program reads or writes the file records.

If you specify the FIELDS clause, and you have selected EBCDIC to ASCII translation, COBOL will translate the data fields specified in this clause and will not translate any others. The fields you specify must be from only one record. If you omit the FIELDS clause, COBOL will translate all the data in the file.

For EBCDIC to ASCII translation, you need not declare a packed decimal field in the CODE-SET clause if the field's encoded sign follows the same format as Data General's packed decimal with USAGE COMP-3.

FEEDBACK Clause



Where:

fbk is a 6-byte data item that receives feedback information concerning the location of the file's records.

The FEEDBACK clause allows you to access specially formatted feedback information concerning the location of records in your file. If you save the information obtained from one access to a file, you can use it to speed up future file accesses. You need feedback when you are writing a record in an inversion of an indexed file (see the READ, REWRITE, and WRITE statements in Chapter 7).

For example, if you declared *fbk* with PIC X(6) for file FIL, then the following sequence of commands will insert the record in an inversion:

```
READ FIL.

(fbk now contains the 6-byte block of information.)

WRITE FIL-INV INVERTED.
```

The system uses the information contained in *fbk* to write the inversion. *Fbk* is unmodified.

You never need to explicitly reference the FEEDBACK item.

PAD Clause

[PAD CHARACTER IS *pad*]

Where:

pad is a single-character alphanumeric literal or data item that specifies a character to fill unused portions of a logical block.

If you specify the PAD clause, COBOL uses *pad* to fill unused portions of a logical block. The default pad character is null.

MERIT Clause

[MERIT IS *recmer*]

Where:

recmer is an integer data item that specifies the priority of this indexed file volume.

You use the MERIT clause when you have a multivolume indexed file and the volumes have different merit factors. You must store the merit factor number in *recmer* prior to writing or rewriting each record into the file. (Merit factors are discussed in the section "ASSIGN Clause" in Chapter 4.) If you omit this clause, the default is 0. Any number of volumes may have a merit factor of 0.

PARTIAL RECORD Clause

[PARTIAL RECORD IS *partrec*]

Where:

partrec is an alphanumeric data item whose length defines the length of partial record data. It receives the partial record on every operation that accesses a data record (unless the partial record is suppressed).

Through multilevel indexing, you can set up partial record index entries to hold frequently used data. This avoids wasting space because you can access just the information you need without accessing the database file.

The size of the data item you specify in *partrec* determines the length any partial record can have. This length cannot be larger than 255 characters. When COBOL accesses a data record for this file, it will return the partial record, if one exists, to *partrec*.

Working Storage Section

You use the Working Storage Section to describe records and noncontiguous data items which are not part of external files, but which are used for general temporary data storage during your program's execution. You define the noncontiguous (or elementary) data items in separate data description entries which begin with either the level number 01 or 77. You define those data items with a hierarchical relationship to one another (group data items) by grouping them into records, and then specifying a data description entry for each one. You begin a data description with any level number from 0 through 49, inclusive. The level numbers 66 and 88 have special functions, and are discussed later on in this chapter.

Linkage Section

You use the Linkage Section only if your program is a subprogram under the control of a CALL statement in another program, and if that CALL statement contains a USING phrase. The Linkage Section's structure is the same as that of the Working Storage Section. It contains data description entries for items that are referred to by both the calling and called programs. COBOL does not allocate space in the program for these data items, but resolves them at runtime by equating the reference in the called program to the location in the calling program. The CALL statement in Chapter 7 and the section "Subprogramming" in Chapter 6 describe the COBOL subprogramming environment.

Data Types

ECLIPSE COBOL provides five types of elementary data. (We describe data editing later on in this chapter.)

Alphabetic data consists of character strings that COBOL stores in 8-bit bytes of ASCII code. It may contain letters (A through Z) and spaces only.

Alphanumeric data consists of character strings, is stored like alphabetic data, and may contain any valid ASCII characters.

Alphanumeric edited data consists of ASCII character strings, is stored like alphabetic data, and provides a special mechanism to specify certain kinds of editing for data output.

Numeric data consists of character strings containing any combination of the numbers 0 through 9. The nine forms of numeric data are discussed below.

Numeric edited data consists of character strings containing any combination of the numbers 0 through 9, and provides a special mechanism to specify certain kinds of editing for data output.

Numeric Data

ECLIPSE COBOL provides nine types of numeric data:

An *unsigned decimal* datum is an unsigned string of not more than 18 ASCII digits. (COBOL handles 16 or less digits more efficiently than it handles 16 to 18 digits.)

A *decimal with trailing sign overpunch* datum is a string of not more than 18 ASCII digits. You may indicate that the datum is negative by changing the rightmost (low-order) digit to a letter according to Table 5-2. You may indicate a positive datum by either leaving the rightmost digit unchanged, or by changing it to a letter according to Table 5-2.

Table 5-2. Sign Overpunch Characters

Digit	Positive	Negative
1	A	J
2	B	K
3	C	L
4	D	M
5	E	N
6	F	O
7	G	P
8	H	Q
9	I	R
0	<173>	<175>

A *decimal with leading sign overpunch* datum is the same as a decimal with trailing sign overpunch datum except that you indicate the sign overpunch in the leftmost (high-order) digit.

A *decimal with trailing separate sign* datum is a string of not more than 18 digits, plus one additional character position immediately to the right of the rightmost digit. This additional position contains a + if the datum is positive and a - if the datum is negative.

A *decimal with leading separate sign* datum is similar to a decimal with trailing separate sign, but you specify the sign character immediately to the left of the leftmost digit.

In all of the numeric forms above, the space character in any digit position represents a 0 digit.

COBOL stores a *packed decimal* datum as a string of 4-bit half-bytes. Each half-byte, except the rightmost, contains a hexadecimal digit of 0 through 9; the remaining half-byte contains a hexadecimal C or F if the value of the datum is nonnegative, or D if the value of the datum is negative. Because the string is always aligned on an 8-bit byte boundary, COBOL will pad the string on the left with a hexadecimal 0 at storage allocation time, if this is necessary to make an even number of half-bytes.

A *byte-aligned binary* datum is a signed number that COBOL stores as a single, two's complement binary integer. COBOL uses as many 8-bit bytes as it needs to store the maximum numeric value of the datum. If a number is nonnegative, the leftmost bit is 0. If it is negative, COBOL stores the two's complement of the number. The number of bytes required to store binary numbers for various decimal digit lengths is shown in Table 5-3. COBOL automatically truncates binary items of 17 and 18 digits to 16 digits.

Table 5-3. Binary Number Storage

Number of Decimal Digits	Bytes Required
1 or 2	1
3 or 4	2
5 or 6	3
7, 8 or 9	4
10 or 11	5
12, 13 or 14	6
15 or 16	7

A *byte-aligned floating point* datum is a signed number in ECLIPSE hardware double-precision floating point format, except that COBOL may store it on any 8-bit byte boundary.

An *external floating point* datum is a signed number that COBOL stores as a string of 8-bit ASCII characters in the format

+9(i).9(f)E+99

where *i* and *f* are the number of integer and fractional digits, respectively. The number of integer and fractional digits must not exceed 16.

COBOL regards several contiguous data items as a single data item which we call *group data.* You may group these data items to form larger groups, thus forming a hierarchical structure of data items. When you reference a group name (unless otherwise noted) COBOL treats the items of the group as alphanumeric data when MOVEing it.

The Data Description Entry

You must declare every data item in your program in a data description entry. In each entry, you specify a level number, a name, and a series of optional clauses that fully describe the data item.

The data description entry has the format:

```
levnum { dn       } [ REDEFINES redef]
       { FILLER   }

[ OCCURS [ int1 TO ] int2 TIMES [ DEPENDING ON dep] [ { ASCENDING  } KEY IS ky-1, ... ] [ INDEXED BY ix-1, ... ] ]
                                                     { DESCENDING }

[ { PICTURE } IS picstr ]
  { PIC     }

[ USAGE IS { DISPLAY                } ]
           { COMPUTATIONAL          }
           { COMP                   }
           { COMPUTATIONAL-1        }
           { COMP-1                 }
           { COMPUTATIONAL-2        }
           { COMP-2                 }
           { COMPUTATIONAL-3        }
           { COMP-3                 }
           { INDEX                  }

[ [ SIGN IS ] { LEADING  } [ SEPARATE CHARACTER ] ]
              { TRAILING }

[ { SYNCHRONIZED } [ LEFT  ] ]
  { SYNC         }   { RIGHT }

[ { JUSTIFIED } RIGHT ]
  { JUST      }

[ BLANK WHEN ZERO ]

[ VALUE IS vallit ].
```

The data description entry consists of several optional clauses, which we will discuss separately. You *must* specify the level number, the data item name or the word FILLER, and either the PICTURE clause or the USAGE IS clause. Each data item name *(dn)* you specify must be unique. The reserved word FILLER specifies an elementary data item in a record. You may never reference a FILLER item explicitly in your program.

You may specify the clauses of a data description entry in any order with two exceptions: the data item name *(dn)* or FILLER must immediately follow the level number, and, if you specify the REDEFINES clause, it must immediately follow the data item name.

Level Numbers

```
levnum { dn       }
       { FILLER   }
```

Where:

levnum is an integer with the value 01 to 49, or 77. It specifies either the hierarchy of the item within a logical record, or that the item is an elementary data item not part of a record.

dn is a data name that specifies the item you are declaring.

You specify a level number to establish hierarchical relationships between data items. Increasing level numbers indicate decreasing positions in the hierarchy. You must state the level number as the first element in each data description.

Independent data items are called records and may be either group or elementary items. You use level number 01 to indicate a record. The maximum level number in a hierarchy is 49. The level numbers 01, 02, ...09 are equivalent to 1, 2, ...9. ECLIPSE COBOL requires the leading zero to support tradition.

When you declare a record and its data items, the subordinate items must immediately follow the group (higher level) item declaration. For example:

```
01 A
   02 B
   02 C
      03 D
            04 E
            04 F
      03 G
      03 H
            04 I
            04 J
   02 K
01 L
01 M
   05 N
      10 P
            15 Q
```

In this example, data items B, E, F, G, I, J, K, L, and Q are elementary items; none of them have subordinate items. All the others are group items. The elementary items E and F are subordinate to the group D; the groups D and H and the elementary item G are subordinate to the group C, which in turn is subordinate to A. The three items A, L, and M are independent items; they are not subordinate to anything else (except possibly an FD or SD entry) and they are not hierarchically related to each other.

COBOL assigns special level numbers to identify certain entries where there is no real concept of levels. Level number 77 identifies noncontiguous elementary data items in the Working Storage or Linkage Section, and is equivalent to an 01-level number. Level number 66 identifies a data item in a RENAMES entry. Level number 88 identifies a data item in a condition name entry. (Both level numbers 66 and 88 are discussed later on in this chapter.)

REDEFINES Clause

[REDEFINES *redef*]

Where:

redef is a data name specifying the data item whose storage area you want to redefine.

The REDEFINES clause allows more than one data name item to reference the same computer storage area even if the structure of each item is different. If you specify the REDEFINES clause, it must immediately follow *dn*. In addition, the data item whose storage area you are redefining *(redef)* must have the same level number as *dn*. You may specify several successive redefinitions of *redef*, each of which may reference either *redef* itself or one of the items that redefines *redef*. You may not declare any data item with the same level number as *redef* and *dn* between the declarations of *redef* and *dn* (except those items which are other redefinitions of *redef)*.

When you use REDEFINES at the record level, the number of character positions you declare does not have to be the same as the previous declaration of that storage area. The number of character positions allocated for the record is the number required by the largest record description given for that storage area. However, redefinitions of subordinate data items must specify the same number of character positions as the item you are redefining.

For successive record descriptions following an FD or SD entry in the File Section, REDEFINES is automatically assumed; an explicit REDEFINES clause is not allowed at the record level in the File Section. You may *not* specify a REDEFINES clause for an item that has an OCCURS clause in its description. However, you may specify a REDEFINES clause for an item subordinate to an item declared with an OCCURS clause. In the latter case, COBOL references *redef* in the REDEFINES clause without subscript references.

When you specify REDEFINES, neither the declaration of *dn* nor the declarations of any of its subordinate items may contain a VALUE clause. Also, you cannot declare an OCCURS DEPENDING clause for either *redef, dn,* or any of their subordinate items.

An example of record-level storage area redefinition is as follows:

```
01 A
   02 B
      03 C                    (5 characters)
      03 D                    (5 characters)
   02 E REDEFINES B
      03 F                    (2 characters)
      03 G
           04 H               (4 characters)
           04 I               (4 characters)
      03 J REDEFINES G
           04 K               (3 characters)
           04 L               (5 characters)
   02 M REDEFINES B
      03 N                    (10 characters)
```

In this example, COBOL allocates 25 character positions for this record, where H defines the first 15 positions, A and I the first 20, and D all. The first 10 character positions of record A are defined in three ways (B, E, and M); and the last 8 positions of E are defined in two ways (G and J).

The following is not permitted:

```
01 A
01 B
01 C REDEFINES A
```

because B declares intervening character positions.

OCCURS Clause

```
[ OCCURS [ int1 TO ] int2 TIMES [ DEPENDING ON dep ] [ { ASCENDING } KEY IS ky-1, ... ] [ INDEXED BY ix-1, ... ] ]
                                                       { DESCENDING }
```

Where:

int1 is a positive integer literal used by the DEPENDING clause.

int2 is a positive integer literal that specifies the number of occurrences of *dn* as an entry in an array.

dep is an unsigned integer data item that specifies a variable number of entries in an array. You may qualify it, but you cannot subscript it.

ky is the name of the entry that contains the OCCURS clause or of an entry subordinate to that entry. You may qualify it, but you cannot subscript it.

ix is an integer data item that receives information generated by COBOL. This name must be unique within your program. You must not declare it in the Working Storage Section; COBOL declares it automatically.

To declare an array, you must specify the OCCURS clause. If you want to declare an array with several dimensions or a subarray within an array, you may specify nested OCCURS clauses. The OCCURS clause indicates that the data item is repeated a number of times to form an array.

An *array* is composed of elements which may be elementary or group data items, or both. All entries in the array, including all items subordinate to the data item being declared, have the same form. There is no limit to the number of elements a COBOL array may have. You reference each of these elements by appending a subscript either to a data name that references an individual element in the array, or to a condition name associated with an array element. The value of the subscript must not be 0 and must not exceed the specified subscript limit *(int2)*. Subscripted references may appear in the Procedure Division of your program (see the section "Array Name Qualification" in Chapter 6). All references to elements of an array must be subscripted references.

If you specify the DEPENDING clause, the associated data item must be the last item of its level in the record. For example, no item at the same level as E in this array follows E in record A:

```
01 A.
   02 B PIC XXX.
   02 C.
      03 D PIC XX.
   02 E OCCURS 1 TO 50 TIMES DEPENDING ON J.
      03 E1 PIC X.
      03 E2 PIC XX.
```

COBOL treats any group data item containing an OCCURS DEPENDING array as if it were a variable-size item. When you reference the group item, your program references only that part of the array specified by the current value of *dep* (i.e., from the first through the *nth* entries, where *n* is the value of *dep)*. In the preceding example, if J contains 28 when a character string is moved into record A, only the first 28 E entries will receive data; the remaining 22 entries will be unchanged. A is (28*3) + 5 characters long.

If you specify the DEPENDING clause, you must specify *int1* and it must be less than *int2*. When you execute your compiled program, the value of *dep* must not be less than *int1* nor greater than *int2*. The value of *dep* (or of *int1)* does not affect subscripted references to items in an OCCURS DEPENDING array.

The data item declared with the OCCURS DEPENDING clause must not be subordinate to an item declared with an OCCURS clause. Furthermore, you may not describe any subordinate item within the OCCURS DEPENDING array as an array.

In ANSI Standard COBOL, you specify the KEY clause for arrays used by a SEARCH statement. However, the SEARCH statement in ECLIPSE COBOL places no restrictions on data items that you may reference in the conditionals of a WHEN clause. Therefore the KEY clause is never required. If you specify it, COBOL checks for syntax, but otherwise ignores it.

The INDEXED BY clause automatically generates the data items *ix-1,...ix-n* in the Working Storage Section as if you had explicitly declared them as:

```
01 IX-1 PIC 9(4) COMPUTATIONAL.
01 IX-2 PIC 9(4) COMPUTATIONAL.
```

Examples of Array Declarations

```
01 AA OCCURS 5 TIMES, PICTURE X.
```

A is an array of 5 single-character alphanumeric data items.

```
01 A
   02 B OCCURS 3 TIMES.
      03 C OCCURS 2 TIMES, PICTURE X.
```

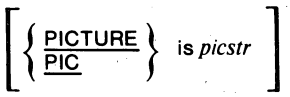
This two-dimensional array has the following structure:

```
A:  B(1):  C(1,1)
           C(1,2)

    B(2):  C(2,1)
           C(2,2)

    B(3)   C(3,1)
           C(3,2)
```

```
01 A.
   02 B OCCURS 5 TIMES.
      03 C PIC 99 COMP.
      03 D PIC X(12).
      03 E OCCURS 4 TIMES PIC 9.
      03 F PIC X.
      03 G OCCURS 10 TIMES.
           04 H OCCURS 8 TIMES.
              05 I PIC XX.
              05 J PIC X.
   02 K PIC X(24).
```

Item A includes a single item K plus an array B of 5 entries. Each of these entries includes a C item, a D item, an array of 4 E items, an F item, and an array of 10 G items. The G items are each composed of 8 I and J item pairs. B(1) is the first entry in the B array; J(5, 10, 8) is the last J item; E(5,1) is the first E item in the last B entry.

PICTURE Clause



Where:

picstr is a valid combination of characters in the COBOL set. It specifies the size and type of the elementary data item *dn*.

The PICTURE clause specifies the length and data type of the elementary data item *dn*. There are five categories of data for which you can specify the PICTURE clause: alphabetic, alphanumeric, alphanumeric edited, numeric, and numeric edited. Data editing is described later on.

You must specify the PICTURE clause for all elementary data items with USAGE DISPLAY COMPUTATIONAL, and COMPUTATIONAL-3. Use of this clause for elementary data items with USAGE COMP-1, COMP-2, and INDEX is prohibited. Use of the clause with group data items is also prohibited.

The maximum number of characters you may write in a PICTURE string is 30. PIC is an equivalent name for PICTURE.

Defining Alphabetic Items

The PICTURE clause of an alphabetic item may contain only the picture symbol A and the editing symbols B, /, and 0. It has the format:

```
A(n)
```

where n is an integer that specifies the length in characters of the item. You may specify the form as shown, for example A(5); or you may write it out as AAAAA; or you may use a combination such as A(3)AA.

Defining Alphanumeric Items

The PICTURE clause of an alphanumeric item may contain only the picture symbols A, X, and 9. It has the format:

```
X(n)
```

where n is an integer that specifies the length in characters of the item. The string must include at least one X. You may specify the form as shown, for example X(4); or you may write it out as XXXX; or you may use a combination such as XXX(2). You may also combine the symbols A, X, and 9, such as:

```
XA9X(2)A
```

In this context, COBOL interprets A's and 9's as X's, so the above example is equivalent to X(6).

Defining Alphanumeric Edited Items

The PICTURE clause of an alphanumeric edited item may contain only the picture symbols A, X, and 9 and the editing symbols B, 0 and /. It has the same format as an alphanumeric item, but must include at least one editing symbol.

Defining Numeric Items

The PICTURE clause of a numeric item may contain only the picture symbols 9, P, S, V, E, -, and +. Certain combinations of these symbols are allowed to create the various forms of this clause for numeric declarations. In all forms, the maximum length of the string is 30 characters, which may specify no more than 18 digits.

If you are declaring a numeric data item that is an *unsigned integer* (i.e., that has no fraction or exponent part), use the format:

9(n)

where n is an integer that specifies the length in digits of the item. You may specify the form as shown, for example 9(6); or you may write it out as 999999; or you may use a combination such as 9(4)99.

The form to use for a signed numeric data item that is a *signed integer* is:

S9(n)

COBOL allocates a character position for the symbol S, depending on the exact data type of the numeric data item (see the section "Numeric Data" earlier in this chapter). COBOL does not count the S in the size of the data item unless you specify the SIGN clause, described later on in this chapter.

The form to use for a numeric data item which is a *signed or unsigned real number,* that is, it has a fractional and (optionally) a whole part, is:

[S][9(n)]V9(m)

where n is an integer that specifies the number of digits to the left of an implicit decimal point, and m is an integer that specifies the number of digits to the right of an implicit decimal point.

The symbol V represents an implicit decimal point; that is, COBOL does not reserve space for it when storing the data item, but all calculations involving the item recognize the implied number of decimal places in the value. COBOL does not count the V in the size of the data item.

The sign symbol S is optional when you are specifying a real number and COBOL treats a sign as it does in a simple numeric form.

You may include *scale factoring* in the PICTURE clause of a signed/unsigned numeric data item by using the form:

[S]9(n)P(m) or [S]P(m)9(n)

where n is an integer that specifies the number of digits to the left of an implicit decimal point and m is an integer that specifies the number of digits to the right of an implicit decimal point; or where m+n is the number of digits to the right of an implicit decimal point.

The P symbol is an implicit decimal point that specifies leading or trailing digits (m) which are not present in the data item, but whose presence affects all calculations involving the data item. It scales the number by powers of 10.

The symbol P may occur to the right or the left of 9's but not on both sides. The implicit decimal point is to the right of the rightmost P if P's are to the right of the 9's; it is to the left of the leftmost P if P's are to the left of the 9's. The V is redundant when used with P's, but you may include it for readability. Thus, the picture strings

```
99PP, PP99, and VPP99
```

are valid, but

```
PP99V, V99P, and PP9P
```

are illegal. The string

```
999P(6)
```

represents a number in the millions, though only the three leftmost digits are stored.

```
P(3)9
```

(equivalent to VPPP9) represents a number in the range of 1 to 9 ten-thousandths, but COBOL stores only the single rightmost digit in the data item.

You may describe a numeric data item in *external floating point representation* by using the format:

```
+9(n)V9(m)E+99
```

where n is an integer that specifies the number of digits to the left of the implicit decimal point, and m is an integer that specifies the number of digits to the right of the implicit decimal point.

The + indicates the position in which you may specify a sign (+ or -). If you omit the sign, COBOL stores a +.

The E represents a character position occupied by the character E. You must specify the + and two digits following the E. They represent the sign and two digits of the number's exponent part.

Defining Numeric Edited Items

The PICTURE clause of a numeric edited item may contain only the picture symbols 9, P, S, V, E, -, +, and the editing symbols B, /, Z, 0, , (comma), . (period), *, CR (credit), DB and $. It has the same format as any of the numeric items, but must include at least one editing symbol.

COBOL stores the value with decimal-point alignment, and with zero fill or truncation on either end as required (just as it stores a value in ordinary numeric data items). COBOL bases the alignment on the picture and editing symbols that represent digit positions in the description. You may specify a maximum of 18 digits in any numeric edited item.

Examples

```
PICTURE IS A(10)
```

Item is alphabetic and 10 characters long.

```
PIC IS X(15)
```

Item is alphanumeric and 15 characters long.

```
PICTURE XA9
```

Item is alphanumeric and 3 characters long.

```
PIC 9999
```

Item is unsigned numeric and 4 digits long.

```
PIC IS S9(6)
```

Item is signed numeric and 6 digits long.

```
PICTURE IS 9(5)V9(2)
```

Item is unsigned numeric with 5 digits to the left of the implicit decimal point and two digits to the right.

```
PIC IS S9(4)P(3)
```

Item is signed numeric and 4 digits long, with scaling to a number in the millions.

```
PICTURE +9(6)V9(4)E+03
```

Item is external floating point and 10 digits long.

Data Editing

COBOL provides special editing features that you may include in the PICTURE clause of an alphabetic, alphanumeric edited, or numeric edited data item. All edited data items are character string items. The five basic editing operations are:

- simple insertion,
- special insertion,
- fixed insertion,
- floating insertion, and
- zero suppression.

Alphanumeric/Alphabetic Editing

COBOL allows simple insertion editing with alphabetic and alphanumeric data items. The three editing symbols COBOL provides are:

B Insert a space character.

/ Insert a slash character.

0 Insert a zero digit.

COBOL counts each of these editing symbols as occupying one character position in the data item.

When COBOL transfers a character string into an alphabetic or alphanumeric edited item, it stores the characters into the positions of the data item (represented by picture symbol X's) in turn from left to right. Positions of the data item pictured by insertion characters do not receive characters from the source string. Instead, COBOL stores a space, slash, or zero in the appropriate position(s). If you declared the edited item as purely alphabetic (all A's) except for the insertion characters, COBOL also checks the source characters to verify that they are all letters and spaces.

Examples:

Source Characters	Picture for Data Item	Data Item Result Value
"ABCDEFGH"	XXBXX/XX0XX	AB CD/EF0GH
"XYZ"	ABA	X Y

Numeric Editing

For all numeric edited items, COBOL provides the simple insertion characters B, /, 0, and , (comma) as well as the special insertion character . (decimal point), the fixed insertion characters +, -, CR (credit), DB, and $, the zero suppression characters Z and *, and the floating insertion characters +, -, and $.

Table 5-4 lists the editing symbols allowed in the PICTURE clause of a numeric edited data item. All the numeric editing characters count as character positions in the data item.

Table 5-4. PICTURE Editing Symbols

Picture Character	Symbol Definition	Editing Function
B	Letter B	Space insertion
/	Slash	Slash insertion
0	Zero	Zero insertion
.	Decimal point	Decimal point insertion
,	Comma	Comma insertion
+	Plus sign	+ insertion
-	Minus sign	Blank or - insertion
CR	Credit sign	CR insertion
DB	Debit sign	DB insertion
$	Dollar sign	Currency symbol insertion
Z	Letter Z	Zero suppression by space
*	Asterisk	Zero suppression by *

The simple insertion characters B, /, and 0 function just as they do in alphanumeric edited items. The insertion character , inserts a comma at its position in the data item. It must not be the last character in a picture string.

You may use the special insertion character . (explicit decimal point) in place of V (implicit decimal point) to represent the position of the decimal point in the number's value. COBOL will perform decimal alignment on the value it is storing (as with V), and will insert the . character itself in the data item at the character position it defines. You must not use the P character if you use the explicit decimal point. Also, the period must not be the last character in a picture string.

You may specify only one of the fixed insertion characters +, -, CR, or DB in a given picture string. If you specify the character + in the picture string, COBOL inserts a + or - character in the data item, depending on whether the value being stored is negative or nonnegative. A single minus character in the picture string functions the same as the plus character except that COBOL inserts a space if the value is nonnegative.

The + or -, if used, must represent either the leftmost or the rightmost character position in the data item. The CR or DB, if used, must represent the rightmost two character positions.

If the value being stored is negative, the character pair CR in the picture string inserts the letters CR at the two character positions which it defines; otherwise, it inserts two spaces. The character pair DB functions the same as CR except that COBOL inserts DB if the value is nonnegative.

If you specify the fixed insertion character $, it must represent the leftmost character position of the data item (except you may precede it by a single + or -). If you defined a different character as the program's currency symbol in the CURRENCY SIGN IS clause of the Special-Names Paragraph (the Environment Division), use that character as the currency symbol in the picture string.

Both Z and * represent digit positions in the data item. You may use suppression symbols to represent either all of the digit positions in the data item, or any of the leading digit positions to the left of the decimal point. A picture such as ZZ.Z9 is not permitted.

If the suppression symbols appear only to the left of the decimal point, COBOL replaces any leading zero in the data which appears in the position of a suppression symbol. Suppression terminates at the first nonzero digit in the data item or at the decimal point, whichever COBOL encounters first. COBOL also suppresses commas or simple insertion characters which appear between or immediately to the right of suppression symbols if no nonzero digit has been stored to the left of their position.

If you represent all numeric character positions in the picture string by suppression symbols and the value of the data item is not zero, the result is the same as if the suppression characters were only to the left of the decimal point. If the value of the data item is zero and the suppression symbol is Z, COBOL will represent the entire data item as spaces. If the value is zero and the suppression symbol is *, COBOL will represent the data item as all *s except for the decimal point. For examples of the use of suppression symbols, see Table 5-5.

Table 5-5. Suppression Symbol Examples

Source Characters	Picture for Data Item	Data Item Result Value
156	999VBOO	156 00
031475	99/99/99	03/14/75
-2153.88	$9,999.99CR	$2,153.88CR
-3452000	9,999PPP+	3,452-
-3452000	-$9,999PPP	-$3,452
26	$999DB	$026□□
8950	$ZZZ,ZZZ.99	$□□8,950.00
0.36	$***,***.99	$*******.36
0	$ZZ9.99	$□□0.00
52.5	+ZZZ.ZZ	+□52.50
0	ZZZ.ZZ	□□□□□□
0	***.**	***.**
456	ZZ//ZZ//99	□□□□□4//56
56	ZZ//ZZ//99	□□□□□□□□56

Floating insertion editing is a form of zero suppression in which one of the characters $, + or - "floats" across the suppressed zeros; COBOL inserts this character immediately before the first unsuppressed digit. For example, moving the values 2000, 200, and 20 into an item with picture $$$$$ gives the results $2000, □$200 and □□$20, respectively. You may use the following three kinds of floating insertion:

+(n) COBOL float-inserts a - or + character, depending on whether the value stored is negative or nonnegative.

-(n) COBOL float-inserts a - character if the value stored is negative; otherwise only zero suppression by spaces occurs.

$(n) COBOL float-inserts a $ character. If you defined a different character as the currency symbol (in the Special-Names Paragraph) you use that character in the picture string and COBOL inserts it in the data item instead of $.

The floating insertion characters must occur in strings of at least two characters. You may insert commas or simple insertion characters between floating insertion characters. COBOL counts commas or simple insertion characters which appear in or immediately to the right of the rightmost floating insertion character as part of the field through which the insertion character floats. Each position you fill with a floating insertion character represents a digit position, except for one --COBOL reserves one position for the floating insertion character itself.

There are two ways you can define a floating insertion character: it can represent either all the digit positions in a picture string, or only those digits to the left of the decimal point. If the floating insertion characters are only to the left of the decimal point in the picture string, COBOL will place a single floating insertion character in the character position immediately preceding either the decimal point or the first nonzero digit in the data item (whichever is the leftmost). COBOL inserts spaces in all character positions to the left of the floating insertion character, except for those positions occupied by a fixed insertion character (+, -, CR, or DB).

If floating insertion characters represent all numeric character positions in the picture string, the result depends upon the value of the datum. If the value is zero the entire data item will contain spaces. If the value is not zero, the result will be the same as if the floating insertion characters are to the left of the decimal point.

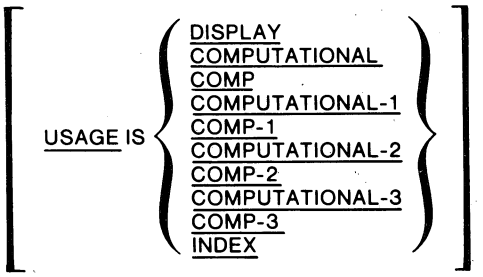
To avoid truncation, the minimum size of the picture string for the result data item must be the number of characters in the sending data item, plus the number of nonfloating insertion characters specified for that item, plus one for the floating insertion character.

You may not combine zero suppression editing with floating insertion editing. For examples of floating insertion editing, see Table 5-6.

Table 5-6. Floating Insertion Editing Examples

Source Characters	Picture for Data Item	Data Item Result Value
256	$$,$$$.99	$256.00
-3170.50	$$,$$$.$$CR	$3,170.50CR
-8.5	++++99.99	-08.50
23456	++++++	+23456
-123456	-(6)	-23456
345	+$(3)999	+□□$345

USAGE Clause



The USAGE clause specifies the representation of a numeric data item in computer storage. The terms used in this clause and their meanings are:

Usage	Data Type
DISPLAY	Decimal or external floating point
COMPUTATIONAL	Binary
COMPUTATIONAL-1 or COMPUTATIONAL-2	Internal floating point (no PICTURE clause permitted)
COMPUTATIONAL-3	Packed decimal
INDEX	Equivalent to PIC 9(4) USAGE COMPUTATIONAL

If you omit the USAGE clause, the default is USAGE IS DISPLAY. You may specify USAGE IS DISPLAY for alphabetic and alphanumeric data items, but the system will ignore it.

COMP, COMP-1, COMP-2, and COMP-3 are abbreviations for COMPUTATIONAL, COMPUTATIONAL-1, COMPUTATIONAL-2, and COMPUTATIONAL-3, respectively.

If you specify the USAGE clause for a group data item, COBOL treats all items subordinate to that group item as if you had declared them with the same USAGE. If you explicitly declare a USAGE clause for one of the subordinate items, it must agree with the USAGE you gave at the higher level.

If you specify USAGE IS INDEX for a group or elementary data item, you may not use the SYNCHRONIZED, JUSTIFIED, PICTURE, VALUE, or BLANK WHEN ZERO clause in the data item's description entry.

SIGN Clause

```
[ [ SIGN IS ] { LEADING  } [ SEPARATE CHARACTER ] ]
              { TRAILING }
```

You indicate the SIGN clause when you must explicitly state the manner of sign representation in numeric data items for which you specified the S picture symbol, and which have USAGE IS DISPLAY. You may not specify the SIGN clause for a packed decimal or binary numeric data item. The terms used in this clause and their meanings are:

Sign Clause	Sign Type
SIGN LEADING	Decimal with leading sign overpunch
SIGN TRAILING	Decimal with trailing sign overpunch
SIGN LEADING SEPARATE	Decimal with leading separate sign
SIGN TRAILING SEPARATE	Decimal with trailing separate sign

If you omit the SIGN clause, the default is SIGN TRAILING.

If sign overpunch data is called for, only the proper overpunch characters, A through R, <173>, and <175> may occur in the overpunch fields. (COBOL does not treat any of the digits 0 through 9 as an implied positive sign if they occur in the overpunch field.)

If you specify the SIGN clause for a group data item, then COBOL treats all items subordinate to that group item as if you had declared them with that same SIGN clause. If you explicitly declare a SIGN clause for one of the subordinate items, it must agree with the SIGN clause you gave for the higher level.

SYNCHRONIZED Clause

```
[ { SYNCHRONIZED } [ LEFT  ] ]
  { SYNC         } [ RIGHT ]
```

COBOL ignores the SYNCHRONIZED clause, but you may include it for compatibility with traditional COBOL implementations. SYNC is an equivalent name for SYNCHRONIZED.

JUSTIFIED Clause

```
[ {JUSTIFIED} RIGHT ]
  {JUST     }
```

The JUSTIFIED clause specifies nonstandard positioning for an elementary alphabetic or alphanumeric data item (no editing specification is allowed).

When you specify JUSTIFIED for a receiving data item that is smaller than the sending item, COBOL truncates the leftmost characters when MOVEing it. When you specify JUSTIFIED for a receiving data item that is larger than the sending data item, COBOL moves the data, aligning it with the rightmost character position, and space-fills the extra, leftmost character positions.

You may not specify the JUSTIFIED clause for a data item described with the USAGE IS INDEX clause.

BLANK WHEN ZERO Clause

```
[ BLANK WHEN ZERO ]
```

The BLANK WHEN ZERO clause substitutes spaces for a numeric or numeric edited data item whose value is zero.

You may not specify the BLANK WHEN ZERO clause for a data item described with the USAGE IS INDEX clause.

VALUE Clause

```
[ VALUE IS vallit ]
```

Where:

vallit is an alphanumeric literal or a figurative constant that specifies the initial value of *dn*.

The VALUE clause assigns an initial value to a data item. If the data item is an alphabetic, alphanumeric, alphanumeric edited, numeric edited, external floating point, or group data item, the VALUE clause must specify an alphanumeric literal. COBOL stores the initial value in the data item by performing a simple character string MOVE with no editing or right justification. Nonnumeric literals in a VALUE clause must not exceed the size of the item defined by the associated PICTURE clause.

If the data item is a numeric item in a form other than external floating point, the VALUE clause must specify a numeric literal. COBOL stores the initial value in the data item according to MOVE rules. The literal may be a floating point literal only if you specify USAGE IS COMP-1 or COMP-2. The value of a numeric literal in a VALUE clause must be within the range of values indicated by the associated PICTURE clause (including any sign), and must not have a value that would require truncation of nonzero digits.

You may not specify the VALUE clause with group or elementary data items described with the USAGE IS INDEX clause.

Examples of Data Description Entries

```
01 CUST-NAME, PIC X(16).
```

Some values that could be stored in CUST-NAME are "JOHN WILLIAMSON" and "VANDERBILT, FRED".

```
01 TOT-AMT, PIC S9(5)V99.
```

The USAGE of TOT-AMT is implicitly DISPLAY and the sign is implicitly trailing overpunch. It is a 7-character data item, with the possible values:

Real Value	Stored
256.38	002563H
-3200.85	032008N

```
01 TOT-AMT, PIC S9(5)V99, SIGN LEADING SEPARATE.
```

has the possible values:

Real Value	Stored
256.38	+0025638
-3200.85	-0320085

```
01 MILLIONS, PIC 9(3)P(6).
```

has the possible values:

Real Value	Stored
526,000,000	526

```
01 TEN-THOUSANDTHS, PIC SVPP99.
```

has the possible values:

Real Value	Stored
+0.0044	4D
-0.0033	3L

```
01 NUM-GROUP.
   02 A, PIC S9(8)V999,  USAGE COMP.
   02 B, PIC S9(12),     USAGE COMP-3.
   02 C,                 USAGE COMP-2.
```

In this example, A is a binary item requiring 5 bytes of storage. B is a packed decimal item filling 14 half-bytes. If B contains the value -123456789012, COBOL stores it as the hexadecimal string 01, 23, 45, 67, 89, 01, 2D. Item C is a floating point binary item.

```
01 EFP, PIC +99.99E+99.
```

This is an external floating point item; it may contain values such as:

Real Value	Stored
2.46×10^{8}	+02.46E+08
88.5×10^{-2}	+88.50E-02
-3.24×10^{-16}	-03.24E-16

The RENAMES Entry

The RENAMES entry permits alternate groupings of elementary data items by providing an alternate name. It has the format:

66 *dn* RENAMES *itm1* [{ THRU / THROUGH } *itm2*]

Where:

dn is a data name that specifies the item you are declaring.

itm1 is a data name that specifies a subordinate data item. It is part of the same record as *itm2.*

itm2 is a data name that specifies a subordinate data item. It is part of the same record as *itm1.*

The words THRU and THROUGH are equivalent.

The RENAMES entry is actually a combination of a data description entry with a level number of 66, and a RENAMES clause.

You may specify more than one RENAMES entry for a logical record. All RENAMES entries referring to data items within a given record must immediately follow the description of the last data item in that record.

If you specify *itm2,* the data item *dn* is the name of a group data item containing all character positions from the first position of *itm1* through the last position of *itm2.*

However, no data item may be subordinate to *dn* if you specify the RENAMES clause. If you omit *itm2, dn* is simply an alternate name for *itm1.*

The character positions you define for *itm2* (if you specify it) must be to the right of those defined for *itm1.* You must not have defined either item with an OCCURS clause or made either subordinate to an item with an OCCURS clause in its data description entry.

You may qualify references to *dn* (see the section "Data Name Qualification" in Chapter 6) only by the name of the record (01-level item) or file with which it is associated. You may not use *dn* itself as a qualifier.

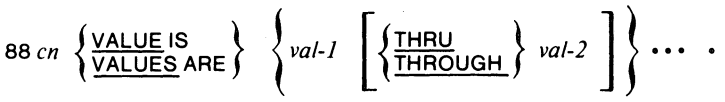
A 66-level entry cannot rename another 66-level entry, nor a 77-, 88-, or 01-level entry.

An example of data description entries which specify RENAMES entries is:

```
01 A.
   02 B PIC X.
   02 C.
      03 D PIC X.
      03 E PIC X.
   02 F PIC X.
   02 G.
      03 H PIC X.
      03 I PIC X.
66 Q RENAMES C.
66 R RENAMES E THRU H.
66 S RENAMES B THRU E.
```

The Condition Name Entry

A condition name entry provides a convenient name for a value, a set of values, or a range of values within a data item whose data description entry it follows. It has the format:



Where:

cn is a condition name that specifies a data item.

val is either an alphanumeric literal or a numeric literal that specifies a value for the data item associated with *dn*.

If the data item with which condition name *cn* is associated is an alphabetic, alphanumeric, alphanumeric edited, numeric edited, external floating point, or group item, *val-1* and *val-2* must be alphanumeric literals; otherwise, they must be numeric literals. If you specify the THRU clause, *val-1* must be less than *val-2*.

You must specify the level number 88. The words THRU and THROUGH are equivalent.

An example of condition name declarations of single data items is:

```
01 A
88 OK VALUE IS "MN".
   02 B PIC X.
   88 B1 VALUES ARE "A", "B".
   88 B2 VALUES ARE "C", "D", "E".
   02 C PIC X.
   88 C1 VALUES ARE "A" THRU "D",
                    "M" THRU "P", "Z".
```

An example of condition name declarations of array elements is:

```
01 D.
   02 E OCCURS 12 TIMES.
      03 F PIC X.
      88 FR VALUE IS "R".
      88 FZ VALUE IS "Z".
```

Note that you must subscript references to FR and FZ just as you would references to F. For example, you could use FR(9) to test whether the ninth entry in F contains R.

End of Chapter

Chapter 6
The Procedure Division

Structure and Concepts

The Procedure Division contains the algorithms that your program performs. It may contain both declaratives and nondeclarative procedures.

The declaratives section is a set of procedures that, if specified, must appear at the beginning of the Procedure Division, must be preceded by the key word DECLARATIVES (followed by a period), and must end with the key words END DECLARATIVES (followed by a period).

A nondeclarative procedure may consist of a paragraph, a group of successive paragraphs, a section, or a group of successive sections. We refer to both paragraph names and section names as procedure names.

The following is a list of all the elements that may appear in the Procedure Division, along with a definition of each:

A *section* consists of a section header with the format:

section name SECTION *segment number.*

followed by one or more paragraphs, or zero or more sentences. The *segment number* indicates whether the section is resident or overlayable. It must be an integer literal in the range 1 through 99 inclusive. All segments with numbers 0 through 49 are resident; all those with numbers 50 through 99 are overlay segments. If you don't specify a segment number, 0 is assumed. You can alter the ranges of numbers which indicate resident and overlay segments by specifying a SEGMENT-LIMIT in the Object-Computer paragraph of the Environment Division (see Chapter 4). Segmentation concepts are discussed later on in this chapter. A section ends immediately before the next section, at the end of the Procedure Division, or, if it resides in the declaratives portion, at the key words END DECLARATIVES.

A *paragraph* consists of a paragraph name, followed by a period, and zero or more statements or sentences. A paragraph ends immediately before the next paragraph name or section name, at the end of the Procedure Division, or, if it resides in the declaratives portion, at the key words END DECLARATIVES.

A *sentence* consists of one or more statements followed by a period.

A *statement* is a syntactically valid combination of COBOL words and symbols. It begins with a COBOL verb, and may (optionally) end with a period. It is the smallest executable unit of a COBOL program. If a statement begins with an imperative verb, one that unconditionally specifies an action for the system to take, we call it an *imperative statement.* All Procedure Division statements are described in full detail in Chapter 7.

A *phrase* is an ordered set of consecutive COBOL character strings that constitutes a portion of a COBOL Procedure Division statement.

Generally, you should specify a section header or paragraph name in the A-margin of your program. However, if you terminated the immediately preceding section or paragraph by a period, you may indent the section header or paragraph name. Do not write any other information (user-defined words, key words, etc.) in the A-margin of your program.

You may have either a simple or a sectioned Procedure Division in your program. A simple Procedure Division has the format:

```
PROCEDURE DIVISION [USING arg-1, ...].
paragraph-1
paragraph-2
...
paragraph-n
```

A sectioned Procedure Division has the format:

```
PROCEDURE DIVISION [ USING arg-1, ... ].
[ declaratives subdivision ]
section-1
section-2

...

section-n
```

Where:

arg is a 01- or 77-level data item which you define in the Linkage Section of the Data Division and which specifies an argument.

You specify the USING phrase only when the program functions as a subprogram with arguments, which is under the control of a CALL statement that passes parameters. We discuss subprogramming later on in this chapter.

The declaratives subdivision is a series of procedures that your program invokes whenever I/O exception conditions occur. We discuss this subdivision, along with other error-handling mechanisms, later on in this chapter.

Name Qualification

Each user-defined name you include in your COBOL program must reference one element uniquely, either by the spelling of its name, or by its position within a unique hierarchy of elements. You reference the higher level elements in a hierarchy to *qualify* a user-defined name that appears in your program more than once. You must qualify each user-defined name sufficiently to make it unique.

Procedure Name Qualification

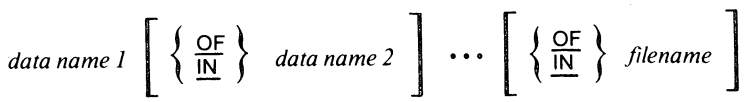
Paragraphs in the Procedure Division may have the same name if they appear in different sections. If you want to reference a paragraph that appears more than once in your program, you must qualify the paragraph name with the name of the section in which it appears. Use the format:

paragraph name [{ OF / IN } *section name*]

If, in any given section, you reference an unqualified paragraph name which appears more than once in your program, the system assumes you are referencing that paragraph in the current section.

Data Name Qualification

The names of records in the Working Storage and Linkage Sections and of index items defined in the INDEXED BY phrase of an OCCURS clause must be unique in a COBOL program. You may qualify the name of a subordinate data item in a hierarchy with the names of the higher level group items which contain it. You may qualify the name of a File Section record by the name of the file with which you associate it. The format for such qualification is:



You may specify a data name for more than one data item in a program if that name identifies data items at the same level. When you reference one of these data items, you must qualify it sufficiently to give it a unique identification. To do this, you specify a series of data names at successively higher levels. You need not include the names of data items at every level of the hierarchy, but must include enough to identify the data name uniquely.

You may use a filename qualifier for those data items listed in the File Section of your program's Data Division.

Given the following data description entries,

```
01 A
   02 B
      03 C
            04 D
      03 E
            04 D
   02 F
      03 C
```

some valid data name qualifications would be:

```
D OF E
D OF C OF B OF A
E OF A
```

some invalid data name qualifications would be:

```
C OF A
D OF B
```

Condition Name Qualification

The switch status condition names in your program must be unique. As with data names, you must identify duplicate condition names in a form which makes them unique. The format to do so is:

The data name may be the name of the data item with which the condition is associated, or the name of any group item to which that data item is subordinate. You may also qualify the data name itself according to the rules for data name qualification.

Array Name Qualification

You use subscripts to reference an individual element in an array, a condition name associated with an array element, or an element in a table. You must not assign an individual data name to these elements. The format to reference these items is:

$$\left\{\begin{matrix}\textit{data name}\\ \textit{condition name}\end{matrix}\right\} \ (\textit{subsc-1}, \ \ldots)$$

Where:

subsc is any arithmetic expression that evaluates to an integer not less than 1 nor more than the number of occurrences you specified in the OCCURS clause associated with this item.

You may qualify, but not subscript, the data name or condition name specifying the qualification before the left parenthesis of the subscript. For example, it is valid to write:

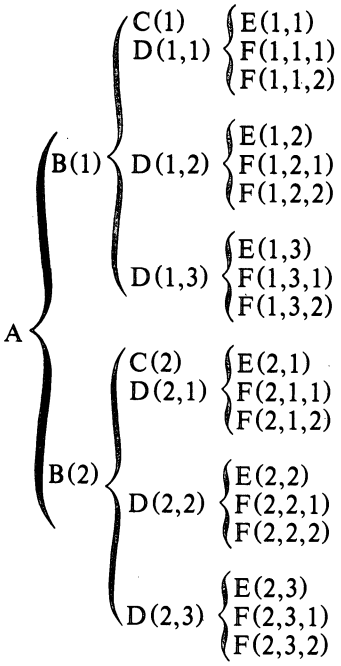
```
C OF B OF A (I,J).
```

If an OCCURS clause appears at only one level in a hierarchy, you specify one and only one subscript to reference elements or subordinate elements (or condition names associated with these) within that array. If two OCCURS clauses appear at nested levels within a hierarchy, then you must use two subscripts to reference items at and below the level of the second OCCURS clause. References to items at or below the first level of the OCCURS clause (but above the second) require one subscript. Similarly, with more than two OCCURS clauses at nested levels, COBOL associates the rightmost subscript with the innermost array; the next subscript to the left with the next higher level array, and so on.

For example, given the data description entries:

```
01 A
   02 B OCCURS 2 TIMES
      03 C
      03 D OCCURS 3 TIMES
           04 E
           04 F OCCURS 2 TIMES
```

The data structure is as follows:



Handling Arithmetics

You may use the ADD, COMPUTE, DIVIDE, MULTIPLY, and SUBTRACT statements to perform your arithmetic functions. COBOL automatically supplies you with several mechanisms to handle your data during calculations made in any of these arithmetic statements. They include the following:

1. If the data descriptions of the operands in an arithmetic operation are not the same, COBOL performs any necessary conversion and decimal point alignment.

2. During the execution of an arithmetic statement that involves more than one operation, such as

   ```
   COMPUTE A = (A + B) * (C + D)
   ```

   COBOL obtains intermediate results. The maximum number of digits of accuracy that COBOL can store for an intermediate result is 31. COBOL truncates any digits that exceed this number from the high-order end of the value.

   For example, let op1 have the form 9(i1).9(d1) and op2 the form 9(i2).9(d2). Then the number of decimal places in an intermediate result is determined by the following formulas:

Operation	Number of Decimal Places in Intermediate Result
op1 + op2	the maximum of d1 and d2.
op1 * op2	d1 + d2
op1/op2	d1 - d2 or the maximum number of decimal places in the result operands, whichever is greater.
op1 ** op2	d1 * op2 if op2 is an integer; if op2 is not an integer, the intermediate result is a floating point value with 16 digits of true accuracy.

3. The maximum size of any operand is 18 decimal digits. In an arithmetic operation the composite of operands (which is a hypothetical data item resulting from the superposition of specified operands in a statement aligned on their decimal points) must not be longer than 18 decimal digits.

4. In any COBOL arithmetic statement, you may specify more than one result item. If you do so, COBOL performs all calculations necessary to determine a result, and stores this result in a temporary location. COBOL then transfers and/or combines the value of this temporary location with each individual result item. COBOL treats the result items in a left-to-right sequence.

   For example, the result of the statement

   ```
   ADD A, B, C TO C, D(C), E
   ```

   is equivalent to the statements:

   ```
   ADD A, B, C GIVING TEMP
   ADD TEMP TO C
   ADD TEMP TO D(C)
   ADD TEMP TO E
   ```

   where TEMP is an intermediate result item.

5. When COBOL sends or receives items in an arithmetic, INSPECT, MOVE, SET, STRING, or UNSTRING statement, and these items share (overlap) part of their storage areas, the result of the statement's execution is undefined. An example of such a statement is:

```
01 A.
   02 B.
      03 C PIC X.
      03 D.
         04 E PIC X.
         04 F PIC X.

MOVE B TO D.
```

6. Except in the case of ALPHABETIC and NUMERIC class conditions (described later on), when you reference a data item in the Procedure Division and the contents of that data item are not compatible with the class specified in the item's data description entry, the result of the reference is undefined.

Common Arithmetic Phrases

There are three optional phrases which may appear in the arithmetic statements of your program. They are designed to give you more control over the results that COBOL generates, if the result's precision is of particular interest to you.

The ROUNDED Phrase

You may specify ROUNDED for any result of an arithmetic operation in the statements ADD, COMPUTE, DIVIDE, MULTIPLY, and SUBTRACT. COBOL calculates the result of an arithmetic operation and aligns the decimal point for storage in the result item. Then if the number of fractional places computed is greater than the number of fractional positions you allowed for, COBOL truncates the result to fit the result item. When you specify ROUNDED in an arithmetic statement, COBOL adds a 1 to the rightmost digit of the result if the most significant digit of the result's truncated portion is greater than or equal to 5.

COBOL ignores the ROUNDED option when processing internal floating point items (COMP-1 or COMP-2 in the PICTURE clause).

For example if you store 1.284 in an item with PICTURE 9V99 with or without rounding, the result is 1.28. If you store 1.285 in the same item with rounding, the result is 1.29; without rounding the result is 1.28.

If you store 2549 in an item with PICTURE 99PP with or without rounding, the result is 25. If you store 2550 with rounding, the result is 26; without rounding the result is 25.

The SIZE ERROR Phrase

You may specify the SIZE ERROR phrase in the arithmetic statements ADD, COMPUTE, DIVIDE, MULTIPLY, and SUBTRACT.

During the execution of an arithmetic statement, when the absolute value of any result, after decimal point alignment, is too large to fit in the number of decimal places allowed in the result item, a size error condition occurs. In this case, COBOL leaves the result item unchanged, but execution of the statement continues until COBOL has performed all operations and stored all satisfactory results. Then, if you specified the SIZE ERROR phrase, control will pass to the first statement in the SIZE ERROR phrase.

If a size error occurs and you did not specify a SIZE ERROR phrase, the value of the result item(s) affected is undefined.

If no size error condition occurs or if you omit the SIZE ERROR phrase, program control goes to the statement following the entire arithmetic sentence (including the statements in the SIZE ERROR phrase).

For example, given the data description entries:

```
01 A PIC S99, VALUE 60.
01 B PIC S99, VALUE 60.
01 C PIC S999, VALUE 260.
```

and the procedure statements

```
ADD A TO B, C;
   SIZE ERROR DISPLAY "SE1",
      STOP RUN.
```

program execution proceeds as follows:

1. A + B = 120;
2. Note size error condition;
3. B is left unchanged;
4. A + C = 320;
5. Store 320 in C;
6. Display "SE1";
7. Halt.

The CORRESPONDING Phrase

You may specify the CORRESPONDING phrase in the Procedure Division statements ADD, MOVE, and SUBTRACT. If you specify this phrase, COBOL will process all items subordinate to the two group items you specify, if the names of those items correspond. For example, given the two group items d1 and d2, a pair of their items correspond if:

1. The word FILLER does not designate a data item in d1 or a data item in d2.

2. Both data items have the same data name and the same qualifiers up to, but not including, d1 and d2.

3. In an ADD or SUBTRACT statement, both data items are elementary numeric data items. In a MOVE statement, at least one of the data items is an elementary data item.

4. No data items subordinate to d1 and d2 contain REDEFINES, RENAMES, or OCCURS clauses (however, d1 and d2 may contain REDEFINES or OCCURS clauses).

For example, given the data description entries,

```
01 A1.                          01 A2.
   02 C PIC X.                     02 K PIC X.
   02 D.                           02 D.
      03 E PIC X.                     03 E PIC X.
      03 F1.                          03 F1 PIC XX.
           04 G PIC X.      AND       03 F2.
           04 H PIC X.                     04 H PIC X.
      03 J PIC X.                          04 I PIC X.
   02 K PIC X.                        03 G PIC X.
                                   02 C PIC X.
```

the MOVE CORRESPONDING statement

```
MOVE CORR A1 TO A2.
```

is equivalent to

```
MOVE C OF A1 TO C OF A2;
MOVE E OF A1 TO E OF A2;
MOVE F1 OF A1 TO F1 OF A2.
```

Arithmetic Expressions

An arithmetic expression can be

- a numeric data item,
- a numeric literal,
- a combination of numeric data items and numeric literals separated by arithmetic operators,
- two arithmetic expressions separated by an arithmetic operator,
- an arithmetic expression preceded by a unary operator, or
- an arithmetic expression enclosed in parentheses.

Table 6-1 contains the COBOL arithmetic operators and their meanings.

Table 6-1. Arithmetic Operators

Operator	Meaning
+	addition or unary plus
-	subtraction or unary minus
*	multiplication
/	division
**	exponentiation

The unary plus operator has no effect on an operand. The unary minus negates the value of its operand. If you specify exponentiation and the number of fractional digits times the exponent exceeds 18, COBOL performs the operation in floating point, which results in approximations. If you specify ROUNDED on the result, you may improve the accuracy in items with DISPLAY usage.

If you enclose an arithmetic expression in one or more sets of parentheses, the expression must have a balanced set of left and right parenthesis pairs.

You must delimit all operators with separators.

When you specify a series of data items and literals with intermixed operators and parentheses, the priority for evaluating the expression is:

1. Elements within parentheses
2. Negation
3. Exponentiation in left to right order
4. Multiplication and division in left to right order
5. Addition and subtraction in left to right order

Examples of arithmetic expressions are:

```
A
(-A)
(A + 2) * 3
1 + 2 * 3 ** 2 (the value is 19)
```

Conditional Expressions

COBOL provides both simple and compound conditional expressions.

Simple Expressions

A *simple conditional expression* is an expression that COBOL evaluates to either a true or false value. The kinds of simple conditional expressions that COBOL provides are relation, class, condition name, and switch status conditions.

COBOL evaluates a *relation condition* as true or false depending on the operator you select and the result of the comparison. A relation condition has the format:

op-1 IS [NOT] { = / EQUAL TO / > / GREATER THAN / < / LESS THAN } *op-2*

Where:

op is an arithmetic expression, an alphanumeric literal, or an alphabetic or alphanumeric data item that specifies an operand.

If the operands are arithmetic expressions, the relational operators have their normal algebraic meanings. If the operands are alphanumeric items, COBOL compares the two items character by character, from left to right, until it either reaches the end of both items or finds a pair of different characters. In the former case, the expressions are equal; in the latter, they are not equal. If the characters of one item are exhausted before those of the other, COBOL uses spaces to continue the comparison. If the operands are mixed (numeric and alphanumeric), COBOL copies the numeric item to a temporary location and performs an alphanumeric comparison.

If COBOL determines that a pair of characters is unequal, it tests for the conditions *greater than* or *less than.*

Unless you specify otherwise, COBOL executes the entire comparison using the character values defined by the program collating sequence in the Object-Computer paragraph of the Environment Division.

If you specify NOT, COBOL inverts the evaluated result.

You specify a *condition name condition* simply by stating the condition name. Its value is true if the value of the data item you associated with the condition name in its data description entry is equal to one of the values listed for that condition name; otherwise it is false. (For more information, see the section "The Condition Name Entry" in Chapter 5.)

A condition name condition has the format:

[NOT] *condition name*

If you specify NOT, COBOL inverts the evaluated result.

A *class condition* determines whether an operand is alphabetic or numeric, or, if it is numeric, whether it is positive, negative, or zero. A class condition has the format:

dat IS [NOT] { ALPHABETIC / NUMERIC / POSITIVE / NEGATIVE / ZERO }

Where:

dat is an alphabetic, alphanumeric, or numeric data item.

If you specify ALPHABETIC, the condition is true if *dat* is either alphabetic or alphanumeric with every character a letter or space. Otherwise, the condition is false.

If you specify NUMERIC, the condition is true if *dat* is either numeric or alphanumeric with every character a number. Otherwise, the condition is false.

If you specify POSITIVE, NEGATIVE, or ZERO, and *dat* is numeric, the condition is true if *dat* has the stated algebraic value. Otherwise, the condition is false.

If you specify NOT, COBOL inverts the evaluated result.

A *switch condition* determines the status (on or off) of a program switch. You specify the switch by a condition name in the Special-Names paragraph of the Environment Division, where you associate it with the ON or OFF status. In the execution command line of your program, you may specify switches /A, /B, /C, .../Z. If you append a switch to your program name, that switch has the status ON. Any switch you omit has the status OFF. A switch condition evaluates to true if the command-line switch is in the state corresponding to the switch's condition name. Otherwise, the condition is false.

A switch condition has the format:

[NOT] *switch name*

If you specify NOT, COBOL inverts the evaluated result.

Compound Expressions

You may combine the preceding types of simple conditional expressions into compound conditional expressions. These include

- a conditional expression enclosed in parentheses (the parentheses must occur in balanced left-right pairs),
- a conditional expression preceded by the unary operator NOT, or
- two conditional expressions separated by one of the logical operators AND or OR.

In a series of conditional expressions with intermixed parentheses and logical operators, the order of precedence in the evaluation is:

1. Elements within parentheses
2. Negation (NOT)
3. Conjunction (AND)
4. Disjunction (OR)

The logical operators and their meanings are shown in Table 6-2.

Table 6-2. Logical Operators

Operator	Meaning
AND	Logical conjunction: the value is true if both of the conjoined conditions are true, and false if one or both of the conjoined conditions are false.
OR	Logical inclusive OR: the value is true if one or both of the included conditions is true, and false if both included conditions are false.
NOT	Logical negation: the value is true if the condition is false, and false if the condition is true.

COBOL evaluates operators in this order: arithmetic, then relational, then logical.

Given A=1, B=2, C=3, D=3, and COND-1=true, the following occurs:

Condition	Value
COND-1 AND A<B	true
COND-1 AND C<B	false
COND-1 OR C<B	true
NOT COND-1	false
NOT B>D	true
NOT (A=B OR C=D)	false
NOT (A=B OR B=C)	true

You may abbreviate a compound conditional expression only if it contains simple conditional operands and logical operators. If the first operand in each of the relational operations is the same, you may omit all but the first. Further, if all the relational operators are the same, you may also omit all but the first.

For example, the statement;

```
A=B AND A>C OR A NOT <D
```

can be abbreviated as:

```
A=B AND >C OR NOT <D
```

The statement:

```
A NOT=B AND A NOT=C
```

can be abbreviated as:

```
A NOT=B AND C
```

You may *not* use parentheses in abbreviated compound conditional expressions.

Subprogramming

The COBOL subprogramming facility allows you to pass parameters from a calling program to the called subprogram. You establish parameter correspondence between calling and called programs by specifying USING phrases in both the CALL statement and the subprogram's Procedure Division header.

The calling program's USING phrase contains the actual parameters that COBOL passes to the called subprogram. For example, given the calling program statement,

```
CALL "SUBPRO" USING P1, P2, P3.
```

P1, P2, and P3 are data items declared in the Working Storage Section of the calling program's Data Division. They specify the parameters you want to pass to the called subprogram.

The parameters in the called subprogram's USING phrase are dummy arguments that merely receive the parameters passed by the calling program. The names of the dummy arguments need not match those of the passed parameters. However, the number of parameters in the calling program must match the number of dummy arguments in the called subprogram. Parameter/argument correspondence is established on the basis of relative position within the USING phrases.

Take the calling program statement we have previously indicated and the following section of the called subprogram, SUBPRO.

```
LINKAGE SECTION.
01 A1 ...
01 A2 ...
01 A3 ...
PROCEDURE DIVISION USING A1, A2, A3.
```

A1, A2, and A3 are dummy data items declared in the Linkage Section of the called subprogram's Data Division. They must be 01- or 77-level items. They correspond to the parameters of the calling program in this manner: A1 to P1, A2 to P2, and A3 to P3.

The sizes of corresponding parameters and arguments must be the same. However, the descriptions of the argument items given in the subprogram's Linkage Section does not have to exactly match that of the passed parameters. For example, you might describe P1 as PIC X(50), but describe A1 as an array of 24 two-character entries with PIC 99 preceded by an independent PIC XX item.

COBOL does not allocate any storage for the Linkage Section items. These items merely describe the subprogram's interpretation of data that resides in another program. The parameters are not physically passed to the subprogram. Instead, COBOL makes an association between the addresses of the parameters in the calling program and the descriptions of the arguments in the subprogram. Any change the subprogram makes to the data will also alter that data for the calling program, since control returns to the calling program after execution of the subprogram's EXIT PROGRAM statement. This means that the subprogram may return data to the calling program as well as receive it from the calling program.

Segmentation

Segmentation allows you to reduce a program's memory requirements by overlaying various procedures. When you use segmentation, you assign each section a segment number between 0 and 99; you assign this number in the section header (described earlier in this chapter). The number you assign determines whether COBOL will handle the segment as resident or overlayable. You may also use the SEGMENT LIMIT Clause of the Environment Division (see Chapter 4) to specify which segments of your program will be resident. All sections with the same segment number are a single segment. Resident segments always reside in memory. Overlayable segments must share the same memory space, so COBOL brings one segment into memory at a time. COBOL brings overlay segments into memory as you reference them for execution.

Print File Formatting

Because COBOL must intersperse printer control characters in output records, you must identify those files in your program that you want to be print files. You do this either by specifying PRINTER in the ASSIGN clause of the file's SELECT clause (in the Environment Division) and/or by specifying the LINAGE clause in the file's FD entry (in the Data Division).

If you ASSIGN your print file to an internal disk or tape file rather than a device, you can display the contents of that file on a device by specifying the appropriate operating system transfer command. (Do not specify the operating system print command, because it does not properly handle the vertical formatting.) If you use this method of outputting your print file data, be sure to specify BLOCK CONTAINS 512 CHARACTERS in the FD entry for the print file. (See Appendix B for operating system differences.)

COBOL provides three format control features: the ADVANCING and AT END-OF-PAGE phrases in the WRITE statement for a sequential file, and the LINAGE clause in the FD entry for a sequential file.

The ADVANCING phrase specifies when and how much printer formatting should be done upon execution of a WRITE statement for the file. The file must be a print file. This phrase offers several options:

1. Advancing to a position *x* lines ahead of the current position;
2. Advancing to the next position on the line printer control channel;
3. Advancing to the next form, or to the next logical page;
4. Outputting the data before outputting the control information;
5. Outputting the data after outputting the control information.

For example, given the program:

```
MAIN PRO.
      OPEN OUTPUT REP.
      WRITE REC FROM PHEAD AFTER ADVANCING 0.
      PERFORM NEWGROUP.
      PERFORM REGULAR 2 TIMES.
      PERFORM SUBTOT.
      PERFORM NEWGROUP.
      PERFORM REGULAR.
      PERFORM SUBTOT.
      WRITE REC FROM PHEAD AFTER ADVANCING PAGE.
      PERFORM NEWGROUP.
      PERFORM REGULAR 3 TIMES.
      PERFORM SUBTOT.
      CLOSE REP.
      STOP RUN.
NEWGROUP.
      WRITE REC FROM ITEM AFTER ADVANCING 3.
REGULAR.
      WRITE REC FROM ITEM.
SUBTOT.
      WRITE REC FROM TOTALS AFTER ADVANCING 2.
```

COBOL outputs a report that looks like this:

```
PHEAD   (page 1)        PHEAD   (page 2)
-                       -
-                       -
ITEM                    ITEM
ITEM                    ITEM
ITEM                    ITEM
-                       ITEM
TOTALS                  -
-                       TOTALS
-
ITEM
ITEM
-
TOTALS
```

The particulars of the ADVANCING phrase are fully explained in the section "WRITE for a Sequentially Organized File" in Chapter 7.

If you specify the LINAGE clause in the print file's FD entry, COBOL automatically generates a special register called the LINAGE-COUNTER, which is defined as if you had declared it with the clause PICTURE 9(6) USAGE DISPLAY. COBOL updates it to contain the number of the line within a page body where the print file is currently positioned. You may reference this register by specifying LINAGE-COUNTER and by qualifying it with the associated print filename. The following actions affect the LINAGE-COUNTER register:

1. When you open a file, it is positioned at the first line of the logical page and LINAGE-COUNTER is set to the value 1.

2. Upon execution of a WRITE statement, if you specify AFTER ADVANCING, COBOL positions the file ahead the number of lines you indicate, outputs the record, and then updates LINAGE-COUNTER to the new value of the current line.

3. Upon execution of a WRITE statement, if you specify BEFORE ADVANCING, COBOL outputs the record, positions the file ahead the number of lines you indicated, and updates LINAGE-COUNTER to the new value of the current line.

4. Upon execution of a WRITE statement, if you specify AFTER/BEFORE ADVANCING PAGE or AFTER/BEFORE ADVANCING *x* LINES, where *x* is larger than the number of lines available on the page, COBOL positions the file at the first line of the next logical page, and resets LINAGE-COUNTER to 1.

We describe in detail how to specify the LINAGE clause in the section "SELECT Clause" in Chapter 4.

You may specify the AT END-OF-PAGE phrase only if you specify the LINAGE clause. You use this phrase if you want to execute special statements should an end-of-page condition occur during an output operation. The execution of a WRITE statement that contains this phrase passes control to the first statement in the AT END-OF-PAGE phrase under two conditions: if the WRITE operation left the file positioned at a line in the footing area of the logical page, or if a page overflow occurred (more lines were specified for the page than were available). Chapter 7 discusses the use of the AT END-OF-PAGE phrase in the section "WRITE Statement for a Sequentially Organized File".

Indexed File Record Selection

To reference records in an indexed file, you can use either keyed access, relative access, or a combination of both. COBOL provides three I/O statement options to perform both types of access for indexed file record selection: the POSITION phrase, the relative option phrase, and the KEY series phrase.

For example, you could use READ KEY A1 to retrieve a record in a simple indexed file, and READ NEXT to retrieve the next record whose key value is greater than that of the last record referenced. In the complex multilevel indexed file in Figure 6-1, the statement READ KEYS A1, B2 obtains the record with key B2. If the file's record pointer is already positioned at the record with key A1, the statement READ STATIC KEY B2 obtains the same record. The statement READ RETAIN POSITION, KEYS A2, B3 would get the record with key B3 without changing the file's record pointer from its setting at the record with key A1.

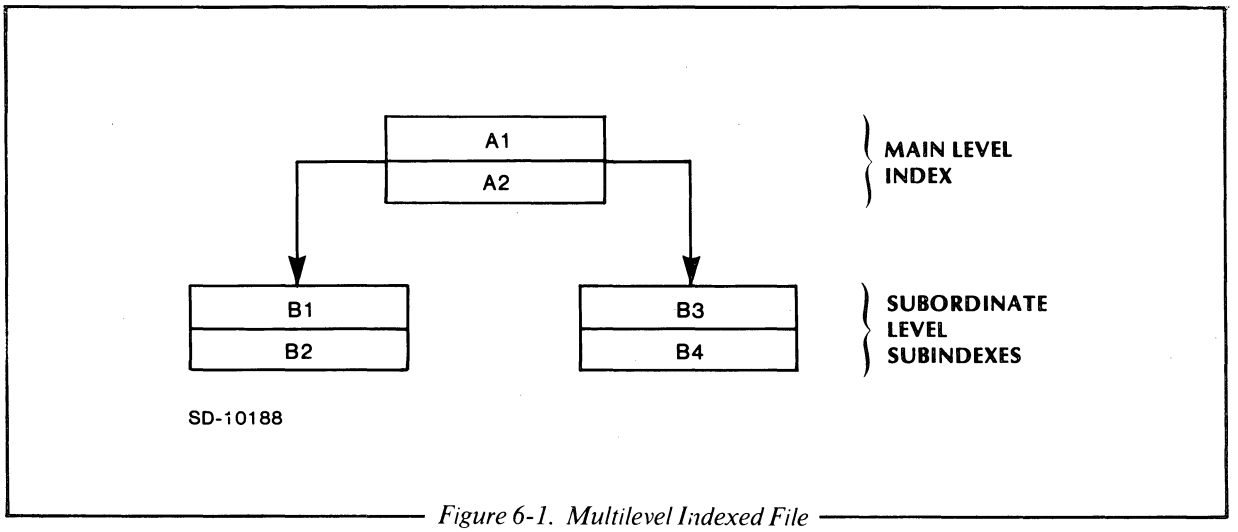


Figure 6-1. Multilevel Indexed File

The POSITION Phrase

The POSITION phrase allows you to override COBOL's automatic positioning of the record pointer to set a current position for a file. The POSITION phrase has the format:

If you specify FIX, COBOL sets the record pointer for the file to the record referenced by the statement. If you specify RETAIN, COBOL leaves the record pointer for the file unchanged (i.e., at the position where it was last fixed).

When COBOL executes a READ or RETRIEVE statement, it sets the file's record pointer to the record it accessed. You may override this by specifying RETAIN POSITION.

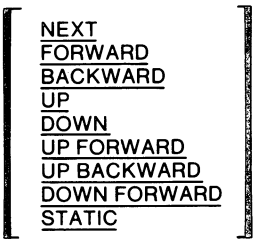
When COBOL executes a DEFINE SUB-INDEX, EXPUNGE SUB-INDEX, LINK SUB-INDEX, REWRITE, UNDELETE, or WRITE statement, the record pointer remains on the last record for which it was set. You may override this by specifying FIX POSITION. The effect of a DELETE statement on a file's record pointer depends on the operating system you are running under. See Appendix B. The START statement always sets the record pointer to the record that is referenced.

If you omit the POSITION phrase in a READ or RETRIEVE statement, the default is FIX; if you omit it in a DEFINE SUB-INDEX, EXPUNGE SUB-INDEX, LINK SUB-INDEX, REWRITE, or WRITE statement, the default is RETAIN. These default conditions permit programs using ANSI Standard COBOL features (without ECLIPSE COBOL I/O extensions) to run without modification.

The Relative Option Phrase

Each time you access a key (and its associated database record), you can set a new current position at that key. Thus, by setting your current position with each processing operation, you may read your file sequentially using relative motion. Since all positioning in your file thereafter will be relative to your current position, we call this *relative access* or *relative position processing.*

The relative option phrase references records in an indexed file, relative to the last setting of the file's record pointer. The format is:



NEXT and FORWARD are equivalent; they position the record pointer to the key with the next greater value in the current index. BACKWARD positions the record pointer to the next lower key in the current index. UP positions the record pointer up one index level to the entry which points to the current (subordinate) index. DOWN positions the record pointer down to the subordinate index pointed to by the current index entry. UP FORWARD, UP BACKWARD, and DOWN FORWARD are combinations of hierarchical motion followed by lateral positioning. STATIC retains the current position, letting you access the same key without changing position (see Table 6-3).

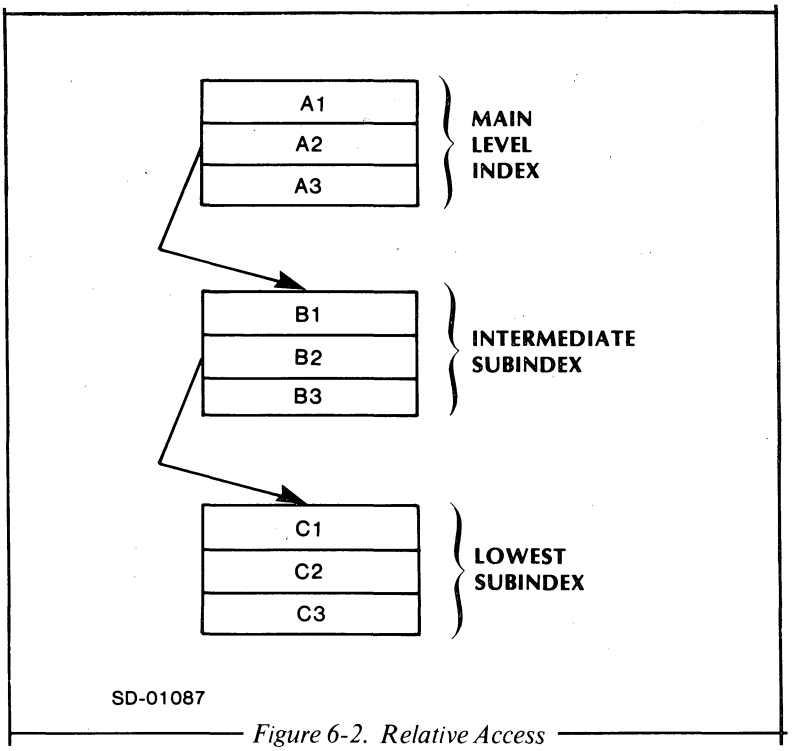


Figure 6-2. Relative Access

For example, in Figure 6-2 the record pointer is positioned at B2. Exercising each of the relative options on that index will yield the following:

Option	Position
FORWARD (NEXT)	B3
BACKWARD	B1
UP	A2
DOWN	Beginning of C index
UP FORWARD	A3
UP BACKWARD	A1
DOWN FORWARD	C3
STATIC	B2

The KEY Series Phrase

Keys explicitly reference a particular database record in an indexed file. The format of the KEY series phrase is:

```
[ { KEY IS  } { k-1 [ APPROXIMATE ] } ... ]
  { KEYS ARE }       [ GENERIC     ]
```

Where:

k is an alphabetic, alphanumeric, or unsigned numeric data item that specifies a record key associated with a file defined in a SELECT clause in the Environment Division.

The value of the key data item at the time COBOL executes the I/O statement specifies the record being referenced.

For a simple indexed file or a simple inversion of an indexed file, you may specify only one key in any I/O statement referencing the file. For an indexed file with ALTERNATE RECORD KEYS, you may specify only one key. The key data item indicates which inversion of the file you want COBOL to use. In either case, you may not use the relative option phrase.

For a multilevel indexed file, you may specify several keys in a single I/O statement. Assuming that you do not specify the relative option phrase, the value, at the time COBOL executes the statement, of the first key data item specified indicates the entry in the highest index level of the file. The value of the second key specified indicates the entry in the subordinate index pointed to by the main-level entry; the value of the third key specified is the entry in the next lower level of the index, and so on. The example at the beginning of this section (READ KEYS A1, B2) illustrates this.

If you specify APPROXIMATE for any key, the index entry accessed will be the one whose value exactly matches the value specified, if such an entry exists. Otherwise, it will be the entry whose value is the next larger. If you specify GENERIC for any key, the index entry accessed will be the first one in the index whose leading characters match the value of the specified key.

If you specify a KEY LENGTH item (in the file's SELECT clause) for a particular key data item, the value of the key length item determines how many of the leading characters of the key data item COBOL will use to determine a match in the key reference.

Table 6-3 indicates where you may use the various relative options, and what effect the KEY series will have, if specified.

Indexed File Record Options

You use the RECORD options phrase with indexed files to tell the system to suppress the input or output of either the data record information or the partial record information (stored with the index entry), or both. The RECORD option phrase has the format:

[SUPPRESS [PARTIAL RECORD] [DATA RECORD]]

If you specify SUPPRESS PARTIAL RECORD in a READ, REWRITE, or WRITE statement, execution will not input the partial record associated with the referenced index entry to nor output it from the file's partial record area. If you specify SUPPRESS DATA RECORD in a READ, REWRITE, or WRITE statement, execution will not input the data record associated with the referenced index entry to or output it from the file's record area.

If you suppress both partial record and data record information you may do so in either order. You can use a READ statement specifying both partial and data record suppression to position the record pointer. A WRITE statement specifying both will make an index entry with which no data need be associated, for example, an entry that references only a subindex. Thus, you can use record suppression to generate inversions for an indexed file.

Table 6-3 Relative Access

Option	Use Only with Multilevel Files?	Position Without KEY Series	Position With KEY Series
FORWARD (or NEXT)	no	Next entry in current index.	--
BACKWARD	no	Previous entry in current index.	--
UP	yes	Entry in higher level index that references current (subordinate) index.	Same as without key series, then apply key series.
DOWN	yes	Position before the first record in the subindex.	Subordinate index referenced by current index entry, then apply key series.
UP FORWARD	yes	UP, then next entry in that index.	--
UP BACKWARD	yes	UP, then previous entry in that index.	--
DOWN FORWARD	yes	First entry in subordinate index referenced by current index entry.	--
STATIC	yes	Remain at current entry in current index.	Apply key series in current index.

Handling I/O Exception Conditions

COBOL can detect three exception conditions in the I/O statements of your program's Procedure Division:

- reaching the end of a file during sequential processing,
- supplying an invalid key when making a keyed reference to a file
- encountering I/O error conditions, such as device errors and data errors.

There are two facilities available to handle these errors. You may specify the AT END and INVALID KEY options in the appropriate I/O processing statements for a file. You may also specify the Declaratives Section at the beginning of the Procedure Division to process all exception conditions for any files or sets of files.

The AT END Phrase

You may specify the AT END phrase in a READ, START, or RETURN statement that is sequentially processing the data in a file. If an end-of-file condition should occur, normal processing will terminate and control will pass to the first statement in the AT END phrase. In a SEARCH statement, the end-of-file condition occurs when processing reaches the end of a subordinate index in a multilevel index file.

For the format and position of this option, see the specific Procedure Division statement in Chapter 7.

The INVALID KEY Phrase

You may specify the INVALID KEY phrase with any statement making a relative or keyed access to a file. Normal statement processing will terminate:

- if a key does not exist upon execution of a READ,
- if the key value is not numeric for a relative file, or
- if a key already exists upon execution of a WRITE.

Control will then pass to the first statement in the INVALID KEY phrase.

Other invalid key conditions include referencing a relative position outside the bounds of a file and specifying an improperly formed key.

For the format and position of this option, see the specific Procedure Division statement in Chapter 7.

The Declaratives Section

COBOL invokes declarative procedures (specified as sections in a separate subdivision at the beginning of the Procedure Division) whenever I/O exception conditions occur, unless the statement that caused the error contains an AT END or INVALID KEY phrase. After execution of the Declaratives Section, control passes to the point in the program following the statement whose processing encountered the exception condition.

The format for the Declaratives Section is:

```
DECLARATIVES.

{ section header          } ...
{ USE statement           }
{ section body.           }

END DECLARATIVES .
```

The USE statement defines the conditions under which COBOL executes the associated declarative section(s). This statement is described, along with other Procedure Division statements, in Chapter 7.

I/O exception conditions that invoke declarative procedures are:

- when the file system detects an I/O error (e.g., an error occurring when OPENing or CLOSEing a file),
- when COBOL encounters an end-of-file condition while sequentially processing a file, or
- when you supply an invalid key in the reference to a record in a relative or indexed file (including a relative position outside the bounds of the file, a key not present in an indexed file, or an improperly formed key).

COBOL File Status Data Items

A file status data item is a two-character data item that you specify in a file's SELECT clause (in the Environment Division). If you specify a file status data item, then any time you execute an I/O statement for that file, COBOL will store a status indicator in this data item. The values stored are shown in Table 6-4.

Table 6-4. COBOL File Status Indicators

Value	Meaning
00	Successful completion.
02	Successful completion - duplicate key entry written.
10	End-of-file condition.
22	Invalid key condition - duplicate key not permitted.
23	Invalid key condition - selected record does not exist.
24	Invalid key condition - relative key value is too large.
30	I/O error (such as data check, parity error, or transmission error).
34	Disk overflow, or physical end of file (end of a reel of tape).
91	File does not exist (OPEN error).
92	On access, file not open or not open in correct mode; on OPEN, file locked by previous CLOSE with LOCK option.
93	WRITE verification error.
94	On access, record locked; on OPEN, file already opened EXCLUSIVE; on OPEN EXCLUSIVE, file already opened.
95	OPEN labeled tape error. On output, indicates volume specifier does not match tape; on input, indicates inconsistent label information.
96	On access, record accessed has been previously marked as logically deleted, either locally or globally.
97	REWRITE or DELETE attempted without executing previous READ for an indexed file with sequential access.
99	INFOS error has occurred for which there is no corresponding file status code. The INFOS error code is in the INFOS status item, if you specified one in the file's SELECT clause.

If a nonstandard error occurs (an error with the status code 99), and you specified a Declaratives Section, control passes to this section to execute its procedures, then returns to the point in the program following the statement that caused the exception. If you did not specify a Declaratives Section, the program terminates after displaying a fatal error message.

INFOS Status Data Items

An INFOS status data item is a data item you specify in a file's SELECT clause (in the Environment Division). If you specify one, then any time COBOL executes an I/O statement for the file, the INFOS status data item will contain the exception code that INFOS or the operating system returns. It will contain zero on a normal return.

End of Chapter

Chapter 7
Procedure Statements

This chapter includes a detailed description of all DGC COBOL procedure statements, arranged alphabetically by the COBOL verb that begins each statement. We present the purpose and format of each statement along with a discussion of its execution. References to several Procedure Division phrases and features appear in this chapter. We describe the basics of these features, but suggest you read Chapter 6 for more specific details.

The following is a summary of the procedure statements contained in this chapter, arranged functionally.

Arithmetic Operations

ADD	Sum two or more operands.
COMPUTE	Evaluate arithmetic expression.
DIVIDE	Divide one operand into others.
MULTIPLY	Multiply one operand by others.
SET UP/DOWN	Variant form of ADD and SUBTRACT.
SUBTRACT	Subtract sum of one operand set from another set.

Data Manipulation and Editing

INSPECT	Search and substitution within character string.
MOVE	Copy contents of data item with optional editing.
SEARCH	Search table for match with data item.
SET	Inverted form of MOVE.
STRING	Concatenate character strings.
UNSTRING	Parse a character string.

Transfer of Program Control

ALTER	Change the destination of a GO statement.
CALL	Transfer control to a subprogram.
CANCEL	Restore subprogram to its initial state.
EXIT	Document end of PERFORM-type subroutine.
EXIT PROGRAM	Return from called subprogram.
GO	Transfer of program control.
IF	Conditional transfer of program control.

PERFORM	Execute subroutine with optional iteration.
STOP	Terminate execution of a program's run.

Console Input/Output

ACCEPT	Low-volume data input.
DISPLAY	Low-volume data output.

File Handling

CLOSE	Terminate processing of files.
DEFINE SUB-INDEX	Create subindex in indexed file.
DELETE	Remove record from indexed file.
EXPUNGE	Delete a file.
EXPUNGE SUB-INDEX	Delete subindex from indexed file.
LINK SUB-INDEX	Provide shared subindex in indexed file.
OPEN	Initialize files for processing.
READ*	Input record from a file.
RETRIEVE	Obtain information about a key in an indexed file.
REWRITE*	Write over existing record in a file.
SEEK	Position I/O system at next record in a relative file.
START*	Position record pointer in a file.
TRUNCATE	Terminate record access in current sequential file block.
UNDELETE	Restore previously deleted record in indexed file.
USE	Define procedures for I/O error handling.
WRITE*	Output record to a file.

Sort/Merge

MERGE	Combine two or more files in sorted order.
RELEASE	Output sort record.
RETURN	Input next sort/merge record.
SORT	Sort one or more files in sorted order.

Miscellaneous

ACCEPT DAY/DATE/ TIME	Obtain calendar and time information.

* These statements have separate descriptions for sequentially organized files, relative files, and indexed files.

ACCEPT

Makes low-volume data (in particular, console input) available for input to a specified data item.

Format

ACCEPT *res* [FROM *dev*]

Where:

res is a numeric or alphanumeric data item that receives the input. If *res* is a numeric data item, COBOL interprets the input data as external format numeric data (external floating point, except that the leading sign and exponent are optional). If *res* is not numeric, COBOL interprets the input data as an alphanumeric string.

dev is an alphanumeric literal or a mnemonic name that specifies an operating system input device or file.

Statement Execution

The input data consists of a single line of characters, excluding the line terminator, or any beyond the 132nd character if you do not specify a line terminator. The ACCEPT statement transfers the input data to *res,* and converts it and edits it if necessary according to MOVE rules. (See the MOVE statement later on in this chapter.)

If you specify a mnemonic name as the input device, you must define that name in the Special-Names paragraph of the Environment Division. If you do not specify an input device, the default is the current input console.

If *res* is larger than the transferred data, COBOL left justifies the data when storing it. If *res* is smaller than the transferred data, COBOL stores only the leftmost characters of the data and ignores those characters that do not fit.

If *res* is numeric, and the input data has more digits than are specified for *res,* COBOL decimal aligns the number and truncates the extra digits on the left as in a MOVE operation. This is a Data General extension to ANSI COBOL.

Examples

Example 1

```
ACCEPT YOUR-NAME FROM READ-ER.
```

Upon execution of this statement, the program pauses. It waits for you to input a string of characters (possibly null) followed by a line terminator, from the device associated with READ-ER in the Special-Names paragraph of your program's Environment Division.

Example 2

```
ACCEPT IN-NAME.
```

If you input ME at the input console, and IN-NAME is defined as PICTURE X(4), COBOL stores ME□□ in IN-NAME.

Example 3

```
ACCEPT NUMBR.
```

If you input 012345 at the input console, and NUMBR is defined as PICTURE 9(4), COBOL stores 2345 in NUMBR.

ACCEPT DATE/DAY/TIME

Obtains calendar and time-of-day information current at the execution of the statement.

Format

ACCEPT *res* FROM { DATE / DAY / TIME }

Where:

res is a data item which receives the day, date, or time, and whose length depends on the function selected.

Statement Execution

DATE, DAY, and TIME are unsigned integer data items automatically supplied by the operating system. This statement transfers the specified data item to *res* according to MOVE rules. (See the MOVE statement later on in this chapter.)

DATE is a 6-digit number representing the current date in the form YYMMDD, where YY is the year, MM is the month, and DD is the day. For example, you would interpret 770901 as September 1, 1977.

DAY is a 5-digit number representing the current Julian date in the form YYDDD, where YY is the year and DDD is the day of the year. For example, you would interpret 77244 as September 1, 1977.

TIME is a 6-digit number representing the time of day, based on a 24-hour clock. The form is HHMMSS, where HH is the hour, MM is the minute, and SS is the second. For example, you would interpret 163500 as 4:35 P.M.

Example

```
ACCEPT TO-DAY FROM DATE.
```

If the current date is October 17, 1977, and you defined TO-DAY as PICTURE 9(6), the value stored in TO-DAY is 771017.

ADD

Sums two or more operands and stores the result.

Formats

ADD TO:

```
ADD addnd-1, ... TO {addndres-1 [ROUNDED]} ... [ON SIZE ERROR statement]
```

ADD GIVING:

```
ADD addnd-1, ... GIVING {res-1 [ROUNDED]} ... [ON SIZE ERROR statement]
```

ADD CORRESPONDING:

```
ADD {CORRESPONDING / CORR} graddnd TO graddndres [ROUNDED] [ON SIZE ERROR statement]
```

Where:

addnd is a numeric literal or a numeric data item that specifies an addend.

addndres is a numeric data item that specifies an addend and receives the result of an ADD TO operation.

statement is an imperative statement to which control passes if a size error condition occurs.

res is a numeric data item or a numeric edited data item that receives the result of an ADD GIVING operation.

graddnd is a group item containing addends that are numeric data items.

graddndres is a group item containing numeric data items corresponding to those in *graddnd*.

Statement Execution

An ADD TO statement sums the addends and then maintains their sum as a constant through the remainder of the operations. Execution proceeds by adding this constant to the current value of *addndres-1* and storing the result in *addndres-1* according to MOVE rules. (See the MOVE statement later on in this chapter.) This process repeats itself for each operand following the word TO.

An ADD GIVING statement sums the addends and stores the result, according to MOVE rules, in each *res* that follows the word GIVING.

An ADD CORRESPONDING statement adds data items in *graddnd* to corresponding data items in *graddndres*, storing the results in the *graddndres* data items according to MOVE rules. COBOL operates on each pair of data items as if you had specified an ADD TO for that pair.

Correspondence occurs according to the rules for the CORRESPONDING phrase (see the section "The CORRESPONDING Phrase" in Chapter 6). CORR is an abbreviation for CORRESPONDING.

If you specify ROUNDED, and COBOL truncates the result of this operation to fit the given result item, it performs the rounding as follows: COBOL adds a 1 to the rightmost digit in the result item if the most significant digit of the truncated portion is equal to or greater than 5.

If you specify the SIZE ERROR phrase and if the absolute value of any result, after decimal point alignment, is too large to fit in the number of decimal places you allowed in the result item, then the result item remains unchanged. The ADD statement completes any remaining operations, stores all satisfactory results, and then transfers control to *statement*. If you omit this phrase or if no size error condition occurs, control passes to the first executable statement following the ADD sentence.

Examples

Example 1

```
ADD A TO B.
```

If A = 4 and B = 3, the result is B = 7.

Example 2

```
ADD A1, A2, A3 TO B1, B2.
```

If A1 = 1, A2 = 2, A3 = 3, B1 = 4, and B2 = 5, the result is B1 = 10 and B2 = 11.

Example 3

```
ADD A1, A2, A3 GIVING B1, B2.
```

If A1 = 1, A2 = 2, A3 = 3, B1 = 4, and B2 = 5, the result is B1 = 6 and B2 = 6.

Example 4

```
ADD CORRESPONDING A TO B.
```

If

```
01 A.
   02 A1 PIC 99 VALUE 1.
   02 A2 PIC 99 VALUE 2.
   02 A3 PIC 99 VALUE 3.

01 B.
   02 A1 PIC 99 VALUE 4.
   02 B2 PIC 99 VALUE 5.
   02 A3 PIC 99 VALUE 6.
```

the result is A1 of B = 5, A3 of B = 9, and B2 of B is unchanged.

ALTER

Changes the destination of a GO TO statement.

Format

```
ALTER { para-1 TO [ PROCEED TO ] dest-1 } ...
```

Where:

para is the name of a paragraph in the Procedure Division that contains a single sentence consisting of a simple GO TO statement.

dest is the name of a paragraph or section in the Procedure Division.

Statement Execution

The ALTER statement modifies the GO TO statement in each *para* you specify so that subsequent executions of these paragraphs will transfer control to the corresponding *dest.*

CAUTION: We do not recommend using ALTER in new program development because this statement will likely be deleted from the next revision of the ANSI COBOL standard. It is also poor programming style.

CALL

Transfers control to a specified subprogram.

Format

CALL *"prog"* [USING *param-1*, ...] [ON OVERFLOW *statement*]

Where:

prog is an alphanumeric literal or an alphanumeric data item that specifies the program id of a subprogram you want to transfer control to.

param is an 01- or 77-level data item that is defined in the Working Storage section and that specifies a parameter you want to pass to the called subprogram.

statement is an imperative statement to which control passes if an overflow condition occurs.

Statement Execution

A called subprogram is in its initial state the first time you CALL it within a specific execution of your program and the first time you CALL it after a CANCEL statement is executed for that subprogram. (See the CANCEL statement later on in this chapter.) On all other entries during that execution, the state of the called program, (including all data items, the status and positioning of all files, and all alterable GO TO statements,) remains unchanged from its state when last exited. You can't call subprograms recursively.

If a CALL statement executes successfully, control passes to the called subprogram. Execution of the subprogram continues until COBOL encounters an EXIT PROGRAM statement. Then control returns to the first executable statement following the CALL sentence in the calling program.

If you want to pass parameters to a called subprogram, you must specify the USING phrase. In addition, you must specify dummy arguments in the Procedure Division header of the called subprogram in order to establish a correspondence between the calling and called programs. The USING phrase must contain the same number of parameters as the called subprogram contains of dummy arguments. Parameter/argument correspondence is based on relative position. For information on COBOL subprogramming, see the "Subprogramming" section in Chapter 6.

If you failed to load the specified subprogram, and you specify the OVERFLOW phrase, control passes to *statement.* If you omit this phrase, or if no overflow condition occurs, control passes to the first executable statement following the CALL sentence in the calling program.

CALL (continued)

Examples

Example 1

```
PROGRAM-ID.   A-PROG.
   .
   .
   .
FILE SECTION.
   .
   .
   .
01 IN-REC.
   02 IN-A PIC X(12).
   02 IN-B PIC X(63).
      .
      .
      .
WORKING-STORAGE SECTION.

01 NUM, PIC S9(5)99 COMP-3.
01 AGROUP.
   02 A-1 PIC X(10).
   02 TAB-ITEM OCCURS 150.
      03 T PIC X.
      03 S PIC 99.
         .
         .
         .

PROCEDURE DIVISION.
   .
   .
   .
   CALL 'SUBPROG' USING NUM, IN-REC, AGROUP.
   .
   .
   .
PROGRAM-ID.   SUBPROG.
   .
   .
   .
LINKAGE SECTION.
01 TABL.
   02 FILLER PIC X(11).
   02 TA OCCURS 100 TIMES.
      03 T PIC 99.
      03 FILLER PIC X.
   02 FILLER PIC X (149).
01 REC.
   02 A PIC X(12).
   02 B PIC X(3).
   02 C PIC 9(5).
   02 D PIC 9.
   02 E PIC X(54).
01 N PIC S9(5)V99 COMP-3.
   .
   .
   .
PROCEDURE DIVISION USING N, REC, TABL.
   .
   .
   .
```

Example 2

```
IDENTIFICATION DIVISION.
PROGRAM-ID. MAIN.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 A1 PIC X(5) VALUE "HELLO".
PROCEDURE DIVISION.
   CALL "SUB1" USING A1.
   STOP RUN.

IDENTIFICATION DIVISION.
PROGRAM-ID. SUB1.
ENVIRONMENT DIVISION.
DATA DIVISION.
LINKAGE SECTION.
01 S1 PIC X(5).
PROCEDURE DIVISION USING S1.
   DISPLAY S1 "FROM SUB1!".
   CALL "SUB2" USING S1.
   EXIT PROGRAM.

IDENTIFICATION DIVISION.
PROGRAM-ID. SUB2.
ENVIRONMENT DIVISION.
DATA DIVISION.
LINKAGE SECTION.
   01 S2 PIC X(5)
PROCEDURE DIVISION USING S2.
   DISPLAY S2 "FROM SUB2!".
   EXIT PROGRAM.
```

CANCEL

Restores a previously called subprogram to its initial state.

Format

CANCEL *"prog-1"*, ...

Where:

prog is an alphanumeric literal or alphanumeric data item that specifies the program id of a subprogram you want to cancel.

Statement Execution

This statement resets all data items, the status and positioning of all files, and all alterable GO TO statements to the values they had before execution of the subprogram.

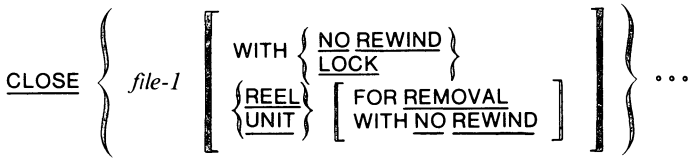
You cannot specify a CANCEL statement within the subprogram you want to CANCEL. You cannot CANCEL a subprogram that has not yet executed an EXIT PROGRAM statement. No action occurs if you try to CANCEL a subprogram that your program did not call in this run unit or one that you already cancelled. Control simply passes to the next executable statement.

NOTE: It is a better programming technique to initialize the data area of your program than to use this statement (see Chapter 5).

CLOSE

Terminates the processing of files or file volumes.

Format



Where:

file is a filename that specifies the file you want to close.

Statement Execution

If you specify more than one file in a CLOSE statement, the files do not need to have the same organization or access, but they cannot be sort/merge files.

The effect of the following options varies from operating system to operating system. See Appendix B for a discussion of these differences.

If you specify LOCK for a file, you may not re-OPEN that file during the current run unit.

If you CLOSE a file or file volume on magnetic tape, the CLOSE statement rewinds the tape unless you specify NO REWIND. If you specified LABELS STANDARD in the FD entry for a file or file volume on magnetic tape, the CLOSE statement writes an end-of-file or end-of-volume label.

UNIT is an equivalent name for REEL. Each time you specify REEL(UNIT) for a single-volume file, this statement closes the file and you cannot specify any input/output operations for that file until you re-OPEN it.

If you specify REEL(UNIT) for a multivolume file, this statement closes only the current volume of that file and opens the next volume (if one exists). You cannot specify any input/output operations for the closed volume until you close the entire file and then re-OPEN it. However, input/output will continue on the next volume.

If you specify REMOVAL for a file volume, the operating system logically releases the device on which you mounted that volume, freeing the device for other uses.

COMPUTE

Assigns the value of an arithmetic expression to one or more data items.

Format

```
COMPUTE { res-1 [ ROUNDED ] } ... = expression [ ON SIZE ERROR statement ]
```

Where:

res is a numeric or numeric edited data item.

expression is any legal arithmetic expression (see the section "Arithmetic Expressions" in Chapter 6).

statement is an imperative statement to which control passes if a size error condition occurs.

Statement Execution

A COMPUTE statement evaluates the *expression* and stores it in *res* according to MOVE rules. (See the MOVE statement later on in this chapter.) If you specify more than one data item, this statement successively stores the value of the *expression* as the new value of each data item.

If you specify ROUNDED, and COBOL truncates the result of this operation to fit the given result item, it performs the rounding as follows: COBOL adds a 1 to the rightmost digit in the result item if the most significant digit of the truncated portion is equal to or greater than 5.

You may combine any number of arithmetic operations in a COMPUTE statement. This differs from the ADD, SUBTRACT, MULTIPLY, and DIVIDE statements where you can specify only one operation.

If you specify the SIZE ERROR phrase, and if the absolute value of any result, after decimal point alignment, is too large to fit in the number of positions you allowed in the result item, then the result item remains unchanged. The COMPUTE statement completes all operations, stores all satisfactory results, and then transfers control to *statement.* If you omit this option or if no size error condition occurs, control passes to the first executable statement following the COMPUTE sentence.

Example

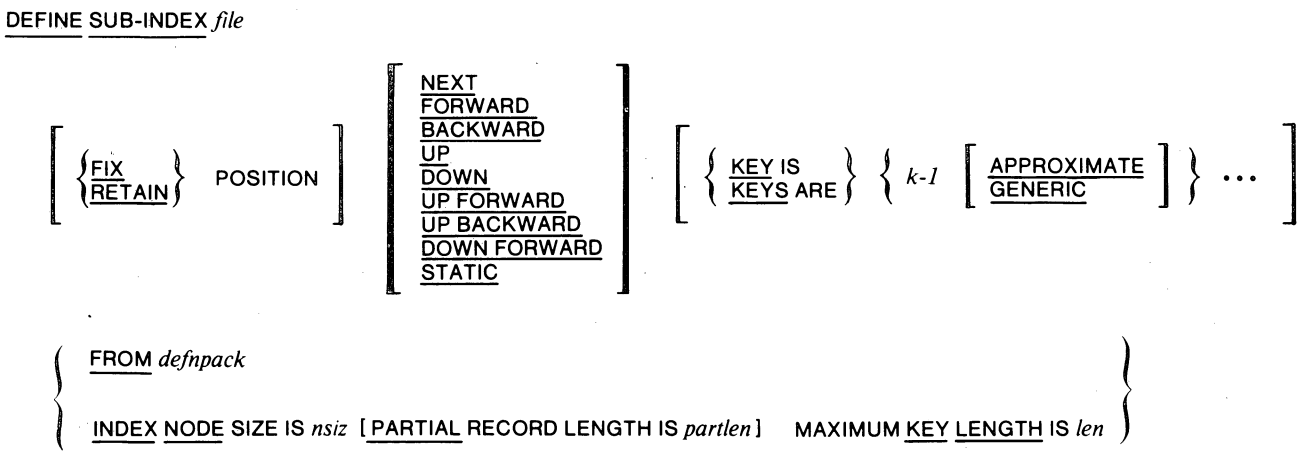
```
COMPUTE A, B ROUNDED = NUM1 + NUM2 * NUM3
```

If A = 1, B = 2 (and both are defined as whole numbers), NUM1 = 6.10, NUM2 = 10.70 and NUM3 = 2.50, then the result is A = 32 and B = 33.

DEFINE SUB-INDEX

Creates a subindex in an indexed file and associates it with a specified index entry in that file.

Format



Where:

file is a filename that specifies an indexed file OPENed for output or I/O and SELECTed for ALLOW SUB-INDEX.

k is an alphanumeric data item that specifies a record key associated with *file*.

defnpack is an alphanumeric data item which contains data in the form of an INFOS subindex definition packet and which is defined in working storage.

nsiz is an integer data item that specifies the size of a subindex node.

partlen is an alphanumeric data item that specifies the partial record length for the subindex.

len is an integer literal or an integer data item that specifies the maximum key length for a subindex.

statement is an imperative statement to which control passes if you specify invalid record selection indicators.

Statement Execution

COBOL determines the location of the index entry with which you want to associate the subindex according to what you specify (explicitly or implicitly) in the POSITION phrase, the relative options phrase (NEXT, FORWARD, etc.), and the KEY series phrase.

By specifying FIX POSITION, you set the record pointer to the record specified by this statement. If you specify RETAIN POSITION, you do not change the current position of the record pointer (i.e., it points to the record for which you last set it). If you omit this option, the default is RETAIN POSITION.

When you specify a relative option, you reference a record in the indexed file, relative to the current position of the file's record pointer. If you omit both this option and the KEY series option, the default is STATIC.

If you specify the KEY series phrase you must have declared each key *(k)* in the SELECT clause for this file.

For more information on these options, see the section "Indexed File Record Selection" in Chapter 6.

You may specify the subindex definition in one of two ways. If you specify the FROM phrase, the data contained in *defnpack* will define the subindex. (Note that the size of this packet is operating system dependent. See Appendix B for more information.) The second way to define the subindex is to specify the INDEX NODE phrase. Here you must define the node size *(nsiz)* and the maximum key length *(len)* for the subindex. If you want partial records stored in the subindex, you must specify the PARTIAL RECORD option. If you want key compression and/or subordinate subindexing for the specified subindex, you must specify KEY COMPRESSION and/or ALLOW SUB-INDEX respectively.

You must specify the ALLOW DUPLICATES phrase if you intend to specify duplicate keys for the new subindex you are defining. This option is system dependent (see Appendix B).

If you specify record selection indicators that reference a record already associated with a subindex or that result in a key positioning error, execution of the DEFINE SUB-INDEX statement terminates, and control passes to *statement*. If you omit this option, or if no invalid key condition occurs, control passes to the first executable statement following the DEFINE SUB-INDEX sentence.

Examples

Example 1

Given the definition packet:

```
01 SUB-INDX-PKT PIC X(16)
   VALUE <000><105><007><372>
   <000><012><000><000><000><000>
   <100><000><000><000><000><000>.
```

you might write the statement:

```
DEFINE SUB-INDEX CUSTMER-FIL FROM SUB-INDX-PKT.
```

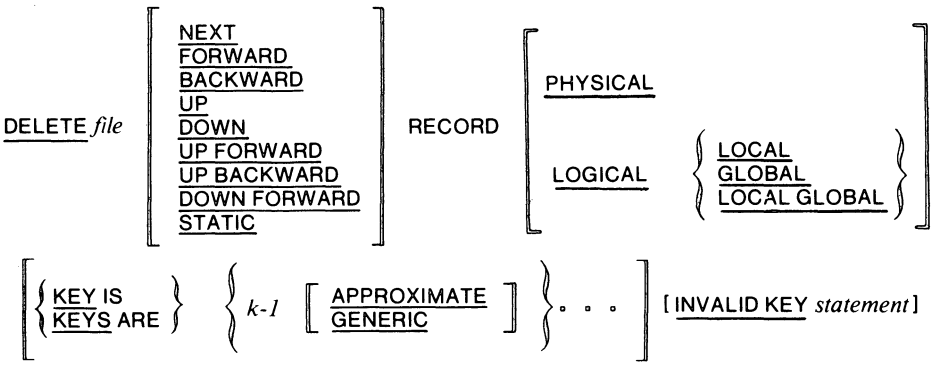
Example 2

```
DEFINE SUB-INDEX
   INDEX NODE SIZE IS 2042
   MAXIMUM KEY LENGTH IS 10.
```

DELETE

Removes the link between a key and its associated data record, either physically or logically.

Format



Where:

file is a filename specifying an indexed file that you OPENed for output or I/O.

k is an alphanumeric data item that specifies a record key associated with *file.*

statement is an imperative statement to which control passes if you specify invalid record selection indicators.

Statement Execution

COBOL determines which record and/or key the DELETE statement will act upon according to what you specify (explicitly or implicitly) in the relative options phrase (NEXT, FORWARD, etc.) and the KEY series phrase. When you specify a relative option, you reference a record in *file,* relative to the current position of the file's record pointer. If you omit this option, the default is STATIC.

If you specify the KEY series phrase, you must have declared each key *(k)* in this file's SELECT clause.

For more information on these options, see the section "Indexed File Record Selection" in Chapter 6.

There are four types of deletion. A PHYSICAL deletion deletes the key you specify and its associated data record, reducing the data record's use count by 1 (see the INFOS System Manual associated with your operating system). If you reduce the use count to zero, you delete the data record itself so that it is no longer in the file or inversion.

A LOGICAL LOCAL deletion marks the *key* as logically deleted. Then whenever you access a record or key through this index, you will receive a file status of 96 (see the section "File Status Registers" in Chapter 6). However, the record will still be available through other indexes. You can restore the key by using the UNDELETE statement described later on in this chapter.

A LOGICAL GLOBAL deletion marks the *data record* as logically deleted. Then whenever you access the data record through any index, you receive a file status of 96. However, you can still access the index entry, including partial record data and any subindex, without receiving a 96. You can restore the data record by using the UNDELETE statement.

A LOGICAL LOCAL GLOBAL deletion accomplishes both a LOGICAL LOCAL deletion and a LOGICAL GLOBAL deletion: it logically deletes *the key and the data record.* You can restore the key and data record by using the UNDELETE statement.

If you do not specify a deletion type, the default is PHYSICAL. If you want a LOGICAL deletion, you must specify LOCAL or GLOBAL or both.

If you want to know whether a record has been deleted, use the RETRIEVE statement (described later on in this chapter).

A DELETE statement does not change the current position of the record pointer unless the pointer's current position is at the deleted record. If this is the case, the result depends on the operating system you are running under (see Appendix B).

If you specify record selection indicators which reference a record that does not exist or one that results in a key positioning error, an invalid key condition occurs. If you specify the INVALID KEY option and an invalid key condition occurs, execution of the DELETE statement terminates and control passes to *statement.* If you omit this option, or if no invalid key condition occurs, control passes to the first executable statement following the DELETE sentence.

Example

```
DELETE MYFILE NEXT RECORD LOGICAL GLOBAL.
```

This statement marks as logically deleted the data record in MYFILE that immediately follows the record pointed to by the current record pointer.

DISPLAY

Makes low-volume data available for output to a specified hardware device or file.

Format

```
DISPLAY src-1, ... [UPON dev] [WITH NO ADVANCING]
```

Where:

src is any data item or literal that you want to display on a device or transfer to a file.

dev is an alphanumeric literal or a mnemonic name which is defined in the Special-Names paragraph of the Environment Division and which specifies an operating system output device or file.

Statement Execution

This statement transfers the source(s) *(src)* to the output device or file in the order you specify them. It transfers them as a continuous, left-justified string of characters with no separation between them. If you specify NO ADVANCING, COBOL does not output an end-of-line character following the last source.

If the total length of the sources is greater than the natural record length of the output device, COBOL sends the output as a series of as many successive records as is necessary to contain the sources without any special formatting by the system.

If a source is a figurative constant, it represents a single-character alphanumeric literal. If a DISPLAY statement contains a source (including the external floating point data type and all numeric literals) which has a DISPLAY usage, it outputs the source as a character string without modification. If a DISPLAY statement contains a source that has COMPUTATIONAL usage, it moves the source to a temporary location according to MOVE rules (see the MOVE statement later on in this chapter), then outputs the temporary value. For information on DISPLAY and COMPUTATIONAL usage see the section "USAGE Clause" in Chapter 5.

The types of temporary storage are:

Source Type	Temporary Type
COMP or COMP-3	DISPLAY, SIGN LEADING SEPARATE, same PICTURE
COMP-2	External floating point PICTURE +9(15)E+99

If you omit the UPON option, the default is the current output console.

Examples

Example 1

```
DISPLAY "PRINCIPAL□RATE□IS□", PRIN.
```

Example 2

```
DISPLAY "OVERFLOW□OCCURRED".
```

Example 3

```
DISPLAY PDQ UPON PRINT-ER NO ADVANCING.
```

DIVIDE

Divides one operand into another or others, and stores the quotient and (optionally) the remainder.

Format

Simple DIVIDE -

```
DIVIDE divsr INTO { divndquot-1 [ ROUNDED ] }... [ ON SIZE ERROR statement ]
```

DIVIDE GIVING -

```
DIVIDE { divsr INTO divnd } GIVING { quot-1 [ ROUNDED ] }... [ REMAINDER rem ] [ ON SIZE ERROR statement ]
       { divnd BY divsr   }
```

Where:

divsr is a numeric literal or a numeric data item that specifies a divisor.

divndquot is a numeric data item that specifies a dividend and receives the quotient of a simple DIVIDE.

statement is an imperative statement to which control passes if a size error condition occurs.

divnd is a numeric literal or a numeric data item that specifies a dividend.

quot is a numeric or numeric edited data item that receives a quotient.

rem is a numeric or numeric edited data item that receives a remainder.

Statement Execution

A simple DIVIDE statement divides a divisor into each dividend/quotient and stores the result of each division in the respective divident/quotient according to MOVE rules. (See the MOVE statement later on in this chapter.)

The DIVIDE GIVING statement divides the divisor into the dividend and stores the result according to MOVE rules in each quotient you specify.

If you specify ROUNDED, and COBOL truncates the result of this operation to fit the given result item, it rounds in the following manner: COBOL adds a 1 to the rightmost digit in the result item if the most significant digit of the truncated portion is equal to or greater than 5.

If you specify the SIZE ERROR phrase, and if the absolute value of any result, after decimal point alignment, is too large to fit in the number of decimal places you allowed in the result item, then the DIVIDE statement completes all operations and transfers control to *statement.* If you omit this phrase or if no size error condition occurs, control passes to the first executable statement following the DIVIDE sentence.

Examples

Example 1

```
DIVIDE A INTO B, C
```

If A = 2, B = 10, and C = 20,
the result is B = 5 and C = 10.

Example 2

```
DIVIDE A BY B GIVING C REMAINDER D.
```

If A = 12, B = 7, C = 0, and D = 0,
the result is C = 1 and D = 5.

EXIT

Documents the end of a PERFORM-type subroutine.

Format

EXIT

Statement Execution

The EXIT statement is a nonexecutable statement. Consequently, COBOL will not signal an error if the EXIT is not the only statement in a sentence or the only sentence in a paragraph.

The existence of an EXIT statement in a subroutine that is not under the control of a PERFORM statement has no effect on your program.

EXIT PROGRAM

Returns control to the point in the calling program immediately following the CALL statement.

Format

EXIT PROGRAM

Statement Execution

The EXIT PROGRAM statement does not have to be the only statement in a sentence or the only sentence in a paragraph.

Execution of an EXIT PROGRAM statement in a program which is not under the control of a CALL statement has no effect.

EXPUNGE
Deletes disk files.

Format

EXPUNGE *file-1, ...*

Where:

file is a filename that specifies an unopened disk file.

Statement Execution

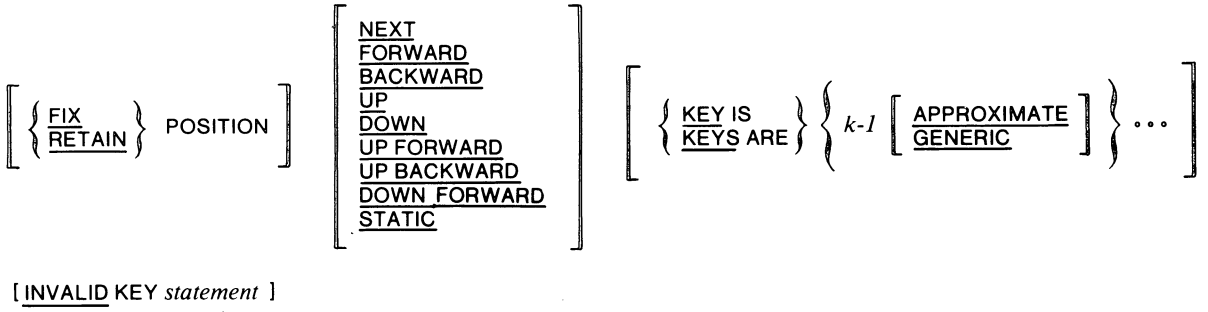
For each *file* you name, the operating system deletes all physical disk files named in the associated ASSIGN clause.

If *file* is an indexed file, you must specify the ASSIGN DATA clause in the SELECT clause for that file, or the operating system will not delete the associated data file.

EXPUNGE SUB-INDEX

Deletes a subindex from an indexed file.

Format



Where:

file is a filename that specifies an indexed file OPENed for output or I/O and SELECTed for ALLOW SUB-INDEX.

k is an alphanumeric data item that specifies a record key associated with *file.*

statement is an imperative statement to which control passes if you specify invalid record selection indicators.

Statement Execution

COBOL determines the location of the subindex you want to delete according to what you specify (explicitly or implicitly) in the POSITION phrase, the relative options phrase (NEXT, FORWARD, etc.), and/or the KEY series phrase.

By specifying FIX POSITION, you set the record pointer to the record specified by this statement. If you specify RETAIN POSITION, you do not change the current position of the record pointer (i.e., it points to the record for which you last set it). If you omit this option, the default is RETAIN POSITION.

When you specify a relative option, you reference a record in an indexed file, relative to the current position of the file's record pointer. If you omit both this option and the KEY series option, the default is STATIC.

If you specify the KEY series phrase, you must have declared each key *(k)* in this file's SELECT clause.

For more information on the options, see the section "Indexed File Record Selection" in Chapter 6.

If you specify record selection indicators which reference a key that does not exist, and if you specify the INVALID KEY option, then execution of the EXPUNGE SUB-INDEX statement terminates and control passes to *statement.* If you omit this option, or if no invalid key condition occurs, control passes to the first executable statement following the EXPUNGE SUB-INDEX sentence.

Example

```
EXPUNGE SUB-INDEX MYFILE.
```

This statement deletes the subindex in MYFILE which is pointed to by the file's current record pointer.

GO

Transfers control from one point in the Procedure Division of your program to another.

Format

Simple GO TO -

GO TO [*proc-1*]

GO TO DEPENDING -

GO TO *proc-1* , *proc-2* ... , *proc-n* DEPENDING ON *int*

Where:

proc is a paragraph name or a section name.

int is an integer data item that specifies which *proc* you want to transfer control to.

Statement Execution

A simple GO TO statement transfers control to *proc,* unless you previously modified this GO TO statement by an ALTER statement. If you do not specify *proc,* you must specify an ALTER statement referring to this GO TO statement. That ALTER statement's execution must occur prior to the GO TO statement's execution in order to affect the execution of the GO TO statement.

A GO TO DEPENDING statement transfers control to *proc-1, proc-2, ...,* or *proc-n* depending on whether the value of *int* is 1, 2, ..., or n. If the value of *int* is anything other than 1, 2, ..., or n, no transfer occurs and control passes to the first executable statement following the GO TO DEPENDING sentence.

Examples

Example 1

```
GO TO PARAGRAPH-9.
```

Control passes to PARAGRAPH-9, unless a previously executed ALTER statement changed the destination.

Example 2

```
GO TO PARA-9, PARA-10, PARA-11 DEPENDING ON TRANS.
GO TO PARA-ERR.
```

If TRANS is 1, control passes to PARA-9; if it is 2, control passes to PARA-10, etc. If TRANS is not equal to 1, 2, or 3, the next statement is executed and control passes to PARA-ERR, (unless a previously executed ALTER statement changed this statement's destination).

IF

Evaluates a condition, then transfers control to one of two statements depending on whether the value of the condition is true or false.

Format

```
IF condition THEN [ { statement-1 | NEXT SENTENCE } ] [ ELSE { statement-2 | NEXT SENTENCE } ]
```

Where:

condition is any legal conditional expression.

statement is any legal Procedure Division statement, including another IF statement.

Statement Execution

A true *condition* transfers control to *statement-1* (if it exists) and executes it. If you specify NEXT SENTENCE or omit the whole option, control passes to the first executable statement following the IF sentence.

A false *condition* transfers control to *statement-2* (if it exists) and executes it. If you specify NEXT SENTENCE or omit this whole option, control passes to the first executable statement following the IF sentence.

After the execution of either of the above conditions, control passes to the first executable statement following the IF sentence, unless a GO TO statement passes control out of the IF statement.

If the ELSE NEXT SENTENCE clause would be the last clause in the IF sentence, you need not specify it.

You may nest IF statements to the limit of memory available to the compiler. If you specify an ELSE in a nested IF statement, it pairs up with the immediately preceding IF that is not already paired with an ELSE (see the example in Figure 7-1). COBOL reports the error COMPILER STACK OVERFLOW if nesting depth is exceeded, and the compilation of your program aborts.

Example

```
IF A = B THEN
   MOVE X TO Y,
   IF C = D THEN
         MOVE O TO Z,
      ELSE
         MOVE X TO Z,
         IF E = F THEN
            NEXT SENTENCE,
         ELSE
            MOVE -1 TO Z.
ELSE
   MOVE O TO Y.
```

Figure 7-1 shows the functions that the above IF statement performs.

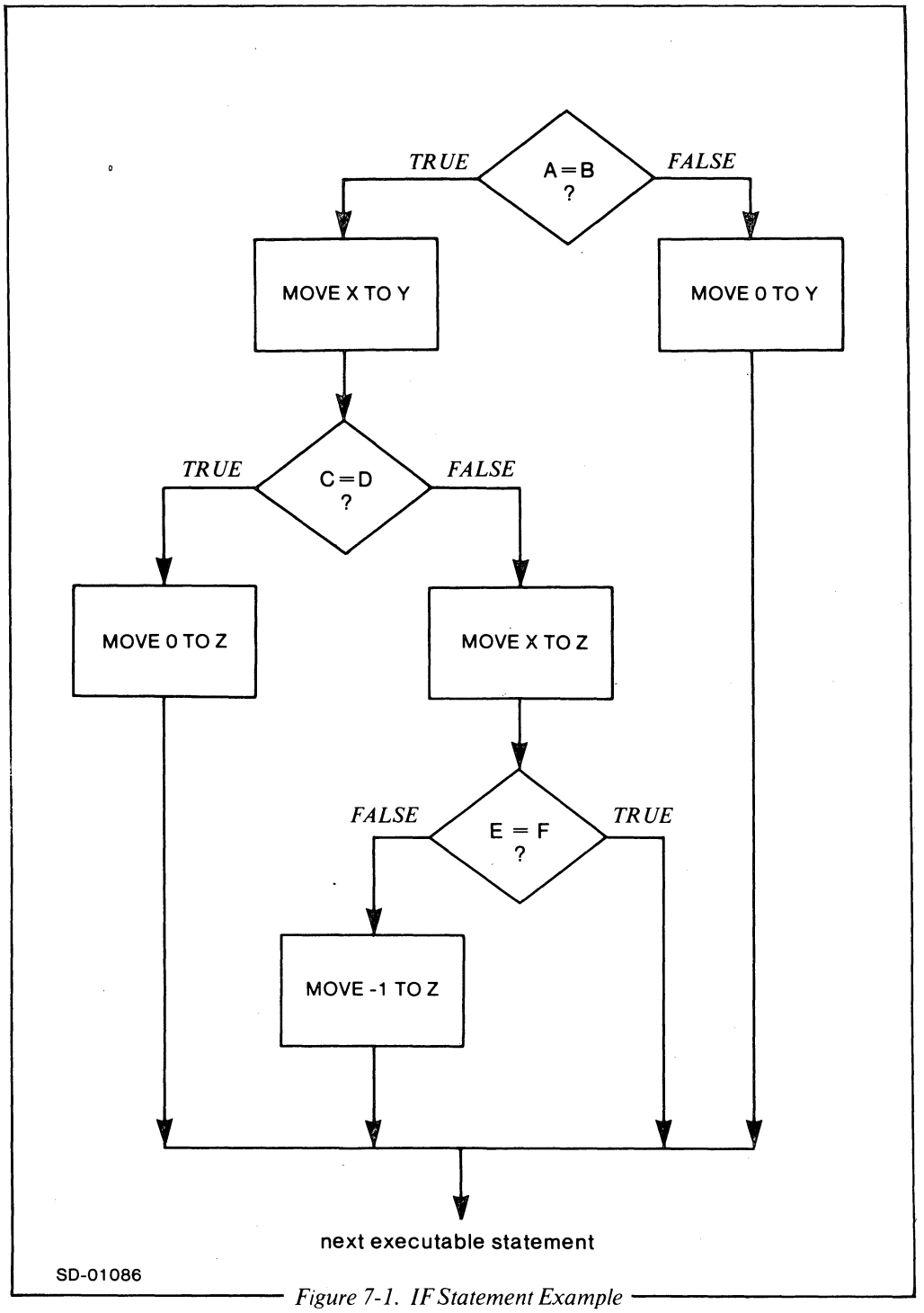


Figure 7-1. IF Statement Example

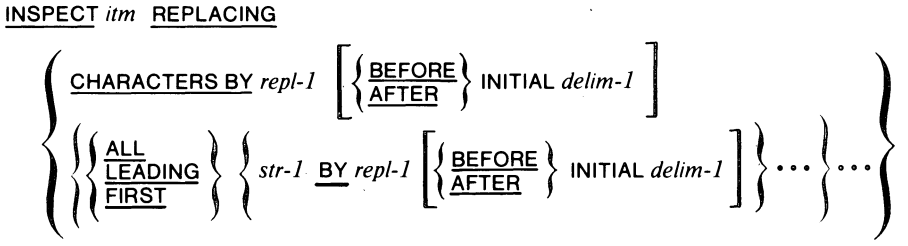
INSPECT

Counts, replaces, or counts and replaces occurrences of specified character strings in a data item.

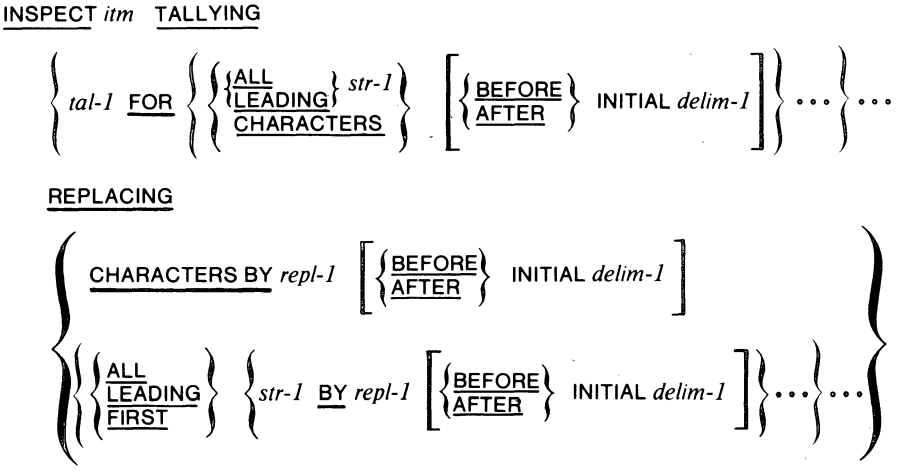
Format

INSPECT TALLYING -

INSPECT REPLACING -



INSPECT TALLYING and REPLACING -



Where:

itm is an alphanumeric literal or a numeric data item that specifies the item you want to inspect.

tal is a numeric data item that the INSPECT statement increments every time it finds a matching string.

str is an alphanumeric literal or a numeric data item that specifies the character string you want to match.

delim is an alphanumeric literal or a numeric data item that specifies an operation delimiter.

repl is an alphanumeric literal or a numeric data item that specifies a replacement string.

All *itm, str, delim,* and *repl* must have DISPLAY usage in their data descriptions. If you specify any of these as data items, COBOL interprets them as alphanumeric. If you specify any as figurative constants, COBOL interprets them as single-character, alphanumeric literals.

Statement Execution

An INSPECT statement searches a data item *(itm)* for a character string(s), which may occur between boundaries determined by a delimiter you may optionally specify *(delim)*. INSPECT compares this string to a string you specify *(str)* and, if there is an exact match within the boundaries, tallies or replaces the string, or both. This is a simple way to state the function of this complex COBOL statement. The following sections give you the details you need to efficiently use the INSPECT statement and all its options.

INSPECT TALLYING

An INSPECT TALLYING statement must set up boundaries before the comparison cycle (discussed below) can take place. The options you specify in an INSPECT TALLYING statement determine the boundaries of each operation that participates in the cycle. The beginning and ending boundary characters and the string of characters in between comprise the string that will be considered for tallying.

For the remainder of this discussion, we will use the following example to demonstrate the actions of an INSPECT TALLYING statement:

```
INSPECT IT TALLYING C-1 FOR ALL "AB",
   C-2 FOR LEADING "Z",
   C-3 FOR ALL "33" BEFORE INITIAL "Q",
   C-4 FOR ALL "6" AFTER INITIAL "B",
   C-5 FOR CHARACTERS.
```

The item to INSPECT is:

```
IT=ZZZABZ336APB33QAB633
   ↑                  ↑
position 1          position 20
```

Table 7-1 shows the table-like structure that COBOL sets up before it scans IT and tallies the appropriate strings.

Table 7-1. INSPECT Table Structure for Example

Operation Order	INSPECT IT TALLYING ...	Character Position Boundaries	Matches
1	C-1 FOR ALL "AB"	1,20	2
2	C-2 FOR LEADING "Z"	1,20	3
3	C-3 FOR ALL "33" BEFORE INITIAL "Q"	1,14	2
4	C-4 FOR ALL "6" AFTER INITIAL "B"	6,20	2
5	C-5 FOR CHARACTERS	1,20	7

As you can see, the INSPECT statement contains several options. We will discuss each of the five operations in the above example separately to point out the effect of these options. We can separate the INSPECT options into two groups. In the first group are the ALL *str*, LEADING *str*, and CHARACTERS options. You must include one of these options in each operation of an INSPECT statement. In the second group are the BEFORE and AFTER options. You may specify one of these with each operation of an INSPECT statement, but you need not specify either of them.

In the first operation of the preceeding example, C-1 FOR ALL "AB", the INSPECT process increments C-1 for *every* occurrence of the string AB in IT because of the ALL option. When you specify ALL, the string INSPECT considers in the comparison cycle is always the entire data item.

In the second operation, C-2 FOR LEADING "Z", the LEADING option indicates that INSPECT will increment C-2 if Z is the leftmost (first) character in IT, and then will increment for each *contiguous* Z thereafter. If Z was the first character of the string, but there were no contiguous Zs following, then INSPECT would increment the tally item once. However, in the above example, Z is the first character in IT, and it is followed by two contiguous Zs; INSPECT, therefore, increments C-2 three times.

In the third operation, C-3 FOR ALL "33" BEFORE INITIAL "Q", the BEFORE option indicates that INSPECT will increment C-3 for every match of 33 BEFORE it encounters the first Q in IT. The string that INSPECT considers in this operation is from the beginning of IT up to, but not including the first occurrence of Q. In this example, the string 33 appears in IT three times, but only two of those occurrences are before the first Q. So, INSPECT increments C-3 twice.

If you specify the BEFORE option with a delimiter that doesn't exist in the data item you are INSPECTing, the comparison cycle will take place as if you had not specified the BEFORE option (i.e., INSPECT will consider the entire string in the comparison cycle).

In the fourth operation, C-4 FOR ALL "6" AFTER INITIAL "B", the AFTER option indicates that INSPECT will increment C-4 for every match of 6 that occurs AFTER the first occurrence of B. The first B occurs at character position 5. There are two occurrences of 6 after that position, so INSPECT increments C-4 twice.

If you specify the AFTER option with a delimiter that does not exist, the operation is never eligible to participate in the comparison cycle.

In the final operation, C-5 FOR CHARACTERS, the CHARACTERS option indicates that INSPECT will increment C-5 for each single character that has not been tallied in a previous operation in this statement. Up to now in this example, we have tallied two occurrences of AB, three Zs, two 33s, and two 6s. The characters in IT that have not participated are (in order) Z, A, P, B, Q, 3, and 3. INSPECT, therefore, increments C-5 seven times.

The Comparison Cycle

The INSPECT statement scans *itm* from left to right. It considers the operations in the statement in the order you specify them. Each operation is elligible for consideration only within the boundaries that COBOL determines for it before the comparison cycle takes place (as shown in Table 7-1).

The comparison cycle begins by considering the leftmost character in *itm* for a match, then continues scanning character by character until it considers the rightmost character in *itm* for a match. This action completes the comparison cycle. *Note that the INSPECT statement does not initialize any of the tally items to zero before the comparison cycle takes place.*

Using the previous example and the data item IT will best illustrate the actions of a comparison cycle. Scanning begins at the leftmost character, Z. INSPECT then checks the operations in order.

The following is the type of question/answer procedure the comparison cycle generates:

Is this character within the boundaries of the first operation? Yes.

Is it a possible AB match? No.

Is it within the boundaries of the second operation? Yes.

Is it a leading Z? Yes.

Increment C-2.

INSPECT then considers the next character in IT and goes back to the first operation the INSPECT statement.

Is this character within the boundaries of the first operation? Yes.

Is it a possible AB match? No.

Is it a leading Z? No.

Is it a contiguous Z? Yes.

Increment C-2.

This cycle continues until it considers the final character, 3, in IT. For each character in IT, the INSPECT statement asks the above questions for each of its operations until it finds a match or until it has considered all operations.

INSPECT REPLACING

An INSPECT REPLACING statement first establishes boundaries for the comparison cycle in the same manner as the INSPECT TALLYING statement. However, the beginning and ending boundary characters, and the string of characters in between, comprise the string that will be considered for replacing, instead of tallying. The comparison cycle is the same as in an INSPECT TALLYING statement, except that it accomplishes replacement.

The ALL and LEADING options function the same as in INSPECT TALLYING but instead of tallying a match with *str*, INSPECT REPLACING replaces each match with the associated *rep1*. The lengths of each pair of *str* and *rep1* must be the same.

If you specify the FIRST option, INSPECT REPLACING replaces only the first occurrence of *str* in *itm*.

The CHARACTERS option functions the same as in INSPECT TALLYING except that it replaces the appropriate characters rather than tallying them.

INSPECT TALLYING and REPLACING

An INSPECT TALLYING and REPLACING statement functions as two separate statements, INSPECT TALLYING and INSPECT REPLACING, in that order.

Example

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01 CAT PIC X(20) VALUE IS
   "ZQXYJJMQRZJXYAZZZXYA".
01 DOG PIC X(4) VALUE IS "XYZA".
01 BIRD PIC X(4) VALUE IS "PPPA".

01 I1 99 COMP VALUE IS 0.
01 I2 99 COMP VALUE IS 0.
01 I3 99 COMP VALUE IS 80.
01 I4 99 COMP VALUE IS 0.
01 I5 99 COMP VALUE IS 0.
01 I6 99 COMP VALUE IS 0.

PROCEDURE DIVISION.
INSPECT CAT TALLYING I1 FOR ALL "XY",
   ALL "J" BEFORE "QR",
   LEADING "Z" AFTER "A",
   I2 FOR CHARACTERS.

INSPECT DOG TALLYING I3 FOR ALL "XY",
   I4 FOR ALL "YZ".

INSPECT BIRD TALLYING I5 FOR ALL "PA",
   I6 FOR LEADING "P".
```

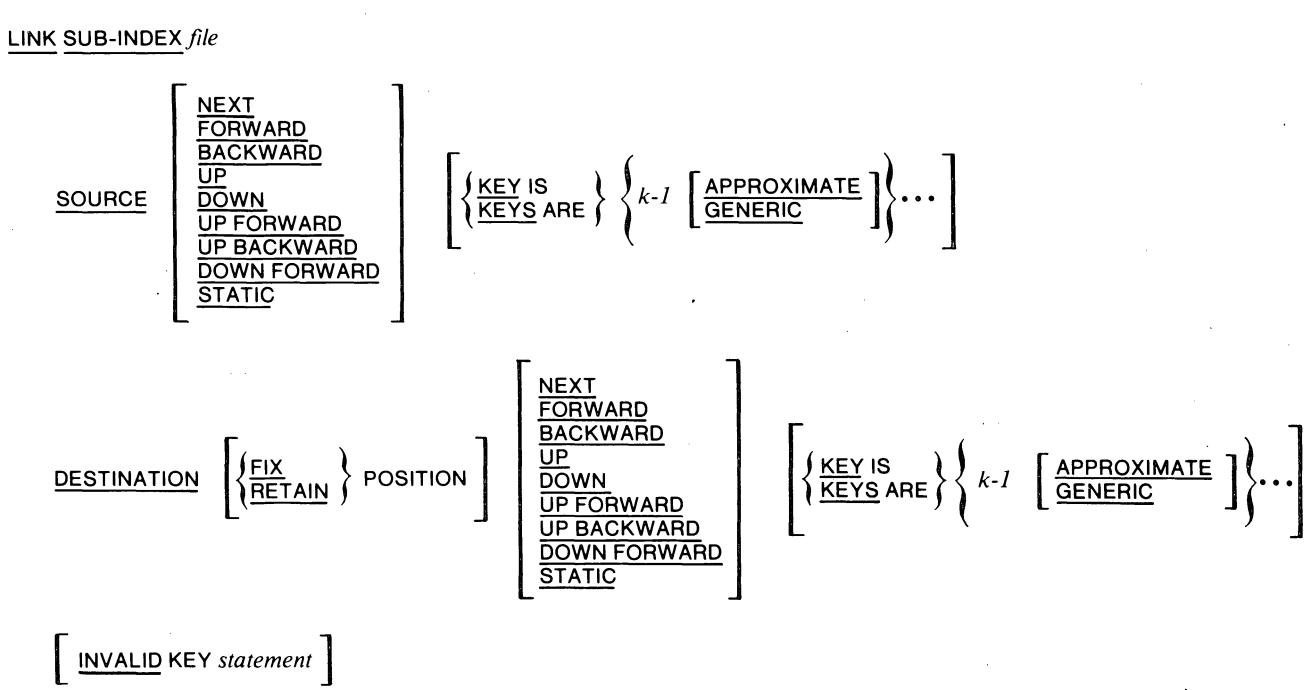
The results of this example are:

I1 = 8 I2 = 9 I3 = 81
I4 = 0 I5 = 1 I6 = 2

LINK SUB-INDEX

Links a file subindex to another index entry so that the subindex can be shared.

Format



Where:

file is a filename that specifies an indexed file OPENed for output or I/O and SELECTed for ALLOW SUB-INDEX.

k is an alphanumeric data item that specifies a record key associated with *file*.

statement is an imperative statement to which control passes if you specify invalid record selection indicators.

Statement Execution

The relative options phrase (NEXT, FORWARD, etc.) and KEY series phrase in the SOURCE phrase specify the location of the subindex that will be linked. COBOL then transfers the link information to the index entry specified by the POSITION phrase, the relative options phrase, and the KEY series phrase in the DESTINATION phrase. This must be the first subindex to which you give this DESTINATION entry.

By specifying FIX POSITION you set the record pointer to the record specified by this statement. (Note that you may specify the POSITION phrase only in the DESTINATION phrase). If you specify RETAIN POSITION, you do not change the current position of the record pointer (i.e., it points to the record for which you last set it). If you omit this option, the default is RETAIN POSITION.

When you specify a relative option, you reference a record in an indexed file, relative to the current position of the file's record pointer. If you omit both this option and the KEY series option, the default is STATIC.

LINK SUB-INDEX (continued)

If you specify the KEY series phrase, you must have declared each key *(k)* in this file's SELECT clause.

For more information on these options, see the section "Indexed File Record Selection" in Chapter 6.

If you specify record selection indicators which reference a key that does not exist, and if you specify the INVALID KEY option, then execution of the LINK SUB-INDEX statement terminates and control passes to *statement*. If you omit this option, or if no invalid key condition occurs, control passes to the first executable statement following the LINK SUB-INDEX sentence.

Example

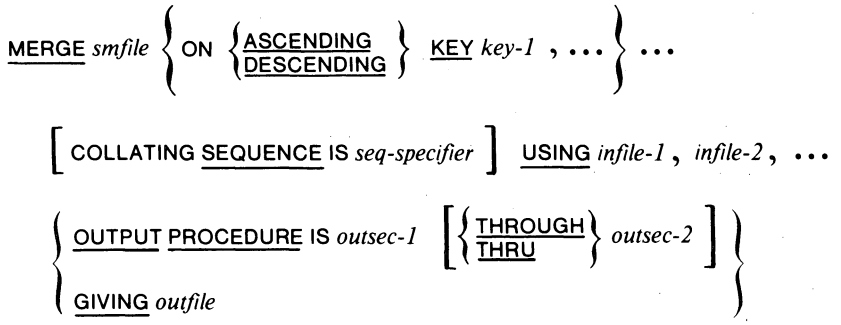
```
LINK SUB-INDEX FILE1 SOURCE
   DESTINATION DOWN FORWARD
   INVALID STOP RUN.
```

This statement transfers the link information from the subindex pointed to by the current record pointer in FILE1 to FILE1's subordinate index entry. If the key referenced does not exist, the program will terminate.

MERGE

Combines two or more sorted files in a sorted order according to a set of specified keys.

Format



Where

smfile is a filename that specifies a sort/merge file described in a sort/merge file description entry in the Data Division.

key is an alphanumeric data item that specifies a key described in records associated with *smfile.*

seq-specifier is one of the following and specifies the program collating sequence you want used when COBOL makes comparisons during the MERGE operation: ASCII, NATIVE, STANDARD-1, EBCDIC, and *alph.*

infile is a filename specifying a file that is not a sort/merge file, does not have subindexing or alternate record keys, is not open at the time of the MERGE operation's execution, and must have been sorted according to specified keys.

outsec is a Procedure Division section name that specifies an output procedure executed as part of the MERGE operation.

outfile is a filename that specifies either an existing but closed indexed file with no subindexing, or a nonexistent sequential file to be created by an implicit OPEN OUTPUT.

Statement Execution

Execution of a MERGE statement combines all records in the files specified in the USING phrase, sorts them according to the keys you specify, and then releases them to an output procedure or output file. All files must be closed prior to execution of the MERGE statement. The MERGE operation automatically opens and closes these files as necessary during its operation.

You must specify ASCENDING or DESCENDING to indicate the comparison you want performed by the MERGE operation when it arranges the records. When comparing the key values in two records, COBOL uses the rules for evaluating relational expressions in which there are three comparisons: greater than, less than, and equal to. If you specify ASCENDING, the MERGE operation arranges the records so that those with lesser key values will come first. If you specify DESCENDING, MERGE arranges the records so that those with greater key values come first. If the MERGE statement encounters two records with identical keys, it writes the records to *outfile* or returns them to the output procedure (whichever you specify), in the order in which you specified the input files in the USING phrase.

The *keys* you specify represent the portion of the data records used in the comparison process. The first *key* you specify is the most important key. It is the first key the MERGE operation uses when comparing two records. The second *key* you specify is the next most significant key; the third *key* is the third most significant key, and so on. The MERGE operation uses these keys only if the value of the first key in the records it is comparing is the same.

If you specify the COLLATING SEQUENCE phrase, the *seq-specifiers* have the same meanings as the specifiers used in the PROGRAM COLLATING SEQUENCE clause of the Environment Division (see the section "Object-Computer Paragraph" in Chapter 4), except that they apply only to comparisons made during the MERGE statement's execution. If you omit this option in the MERGE statement, the MERGE operation uses what you specified in the PROGRAM COLLATING SEQUENCE in the Environment Division. If you omit both options, the default is ASCII.

If you specify the OUTPUT PROCEDURE clause, it must consist of one or more sections that appear contiguously in your program and it must include at least one RETURN statement. It is the RETURN statement that requests the next record for processing. MERGE executes the sections of the output procedure according to the rules for a simple PERFORM (see the PERFORM statement earlier in this chapter). You must not pass control out of the output procedure, which may include any process necessary to select, modify, or copy records that the MERGE operation is returning from *smfile*. The words THROUGH and THRU are equivalent.

If you specify the GIVING clause, control transfers all merged records from *smfile* to *outfile*.

The logical records in the *infile(s)* and in *outfile* must be equal in size to the logical records in *smfile*.

MERGE statements may appear anywhere in your program's Procedure Division except in the declaratives portion or in an input or output procedure associated with a SORT or MERGE statement.

After completion of the MERGE operation, control passes to the first executable statement following the MERGE statement.

Example

```
MERGE CUST-FINAL ON ASCENDING KEY NAME, ADDR, PHONE
   USING CUST1, CUST2, CUST3
   OUTPUT PROCEDURE IS UPDATE.

.
.
.
UPDATE.
   MOVE CURR-DATE INTO CUST-DATE.
   ADD 1 TO FLAG.
   RETURN CUST-FINAL.
```

The above statements will combine the sorted files CUST1, CUST2, and CUST3 into the single file CUST-FINAL according to three keys: customer name, customer address, and customer phone number, in that order. Before each record is stored in CUST-FINAL, a date field will be updated and a special flag set to 1.

MOVE

Transfers data from one data item to one or more other data items, with format conversion and editing as required.

Format

Simple MOVE -

```
MOVE src TO dest-1, ...
```

MOVE CORRESPONDING -

```
MOVE { CORRESPONDING } grsrc TO grdest
     { CORR         }
```

Where:

src is any data item or literal that specifies a data source you want to move.

dest is any data item that receives a data source.

grsrc is a group data item that specifies data sources you want to move.

grdest is a group data item that receives a group of data sources.

Statement Execution

A simple MOVE statement moves the data source to the first (or only) destination item, then to the second, if you specified one, etc. This statement evaluates any subscripting or indexing associated with *dest* before moving the data.

If you specify a group data item as either the source or destination of a simple MOVE statement, the MOVE executes as a character string move with no editing.

A MOVE CORRESPONDING statement moves data items in *grsrc* to corresponding data items in *grdest.*

This operation is the same as if you specified a separate MOVE statement for each pair.

Correspondence occurs according to the rules for the CORRESPONDING phrase (see the section "CORRESPONDING Phrase" in Chapter 6). CORR is an abbreviation for CORRESPONDING.

Character String Move

A MOVE statement moves the characters of the source to the character positions of the destination. For left-justified destinations, COBOL moves the first source character to the first destination position; the second source character to the second destination position; etc. If the source is shorter than the destination, COBOL space-fills the extra (rightmost) positions of the destination. If the source is longer than the destination, COBOL does not move the extra (rightmost) source characters.

If you define the destination as JUSTIFIED RIGHT in its PICTURE clause, COBOL moves the last source character to the last destination position; the next-to-last source character to the next-to-last destination position; etc. If the source is shorter than the destination, COBOL space-fills the extra (leftmost) positions of the destination. If the source is longer, COBOL does not move the extra (leftmost) source characters.

If the source is numeric, COBOL moves an unsigned decimal equivalent of the source's value to the destination, according to the rules for a character string move. For example, when moving an item with PICTURE S9(4) USAGE DISPLAY or COMP VALUE-563 to an item with PICTURE X(3), the result is "056".

If the destination is an alphanumeric edited data item, COBOL moves the characters of the source to the character positions of the destination in a left to right manner as in a typical character string move, except that editing takes place according to the PICTURE of the destination (see the section "Data Editing" in Chapter 5).

Numeric Move

A MOVE statement containing numeric data items stores the algebraic value of the source in the destination item. The source and destination may be any of the nine numeric types; COBOL performs any necessary type conversion automatically. If the source is alphanumeric, it must be a simple digit string (e.g., "00590"). COBOL treats it as an unsigned decimal data item.

COBOL aligns the value stored in the destination on the destination decimal point. If the source has excess digits either to the left or right of the decimal point, COBOL truncates the extra (leftmost and/or rightmost) digits. If the source has too few places to the left or right of the decimal point, COBOL zero-fills the extra (leftmost and/or rightmost) positions of the destination.

If the destination is a numeric edited data item, the MOVE statement stores the algebraic value of the source in the destination, just as in a numeric move, except that editing takes place according to the PICTURE of the destination (see the section "Data Editing" in Chapter 5).

Table 7-2 shows the techniques used when moving elementary data types in a MOVE statement. The gray squares signify illegal moves.

Source Data Type \ Destination Data Type	Alphanumeric	Alphanumeric Edited	Alphabetic	Numeric (all types)	Numeric Edited
Alphanumeric	character string move	character string move with editing	character string move	numeric move	numeric move with editing
Alphanumeric Edited	character string move	character string move with editing	character string move		
Alphabetic	character string move	character string move with editing	character string move		
Numeric (all types)	character string move after conversion	edited character string move after conversion		numeric move	numeric move with editing
Numeric Edited	character string move	character string move with editing			

Table 7-2. MOVE Rules

Examples

Example 1

```
MOVE "STRING" TO FLD.
```

If FLD is defined as PIC X(5), the result is FLD=STRIN.
If FLD is defined as PIC X(7), the result is FLD=STRING□.
If FLD is defined as PIC X(5) JUSTIFIED RIGHT, the result is FLD=TRING.
If FLD is defined as PIC X(7) JUSTIFIED RIGHT, the result is FLD=□STRING.

Example 2

```
MOVE "59032" TO NBR-FLD.
```

If NBR-FLD is defined as PIC 9(3)V9(2), the result is NRB-FLD containing the value 590.32. (Remember that the decimal point is an implicit one.)
If NBR-FLD is defined as PIC 9(2)V9(1), the result is NBR-FLD containing the value 03.2.
If NBR-FLD is defined as PIC 9(4)V9(3), the result is NRB-FLD containing the value 0059.032.

MULTIPLY

Multiplies one operand by one or more others and stores the result.

Format

Simple MULTIPLY -

```
MULTIPLY mplcnd BY { mplrres-1 [ROUNDED] } ... [ON SIZE ERROR statement]
```

MULTIPLY GIVING -

```
MULTIPLY mplcnd BY mplr GIVING { res-1 [ROUNDED] } ... [ON SIZE ERROR statement]
```

Where:

mplcnd is a numeric literal or a numeric data item that specifies a multiplicand.

mplrres is a numeric data item that specifies the multiplier and receives the result of a simple MULTIPLY operation.

statement is an imperative statement to which control passes if a size error condition occurs.

mplr is a numeric literal or a numeric data item that specifies a multiplier.

res is a numeric or numeric edited data item that receives the result of a MULTIPLY GIVING operation.

Statement Execution

A simple MULTIPLY statement multiplies the multiplicand *(mplcnd)* by each multiplier/result *(mplrres)* and stores each result in the corresponding *mplrres* according to MOVE rules. (See the MOVE statement earlier in this chapter.)

A MULTIPLY GIVING statement multiplies the multiplicand by the multiplier and stores the product in each result *(res)* according to MOVE rules.

If you specify ROUNDED and COBOL truncates the result of this operation to fit the given result item, it rounds in the following manner. COBOL adds a 1 to the rightmost digit in the result item if the most significant digit of the truncated portion is greater than or equal to 5.

If you specify the SIZE ERROR phrase and if the absolute value of any result, after decimal point alignment, is too large to fit in the number of decimal places you allowed in the result item, then the MULTIPLY statement completes all operations and transfers control to *statement.* If you omit this phrase or if no size error condition occurs, control passes to the first executable statement following the MULTIPLY sentence.

Examples

Example 1

```
MULTIPLY FLD1 BY FLD2 ROUNDED, FLD3.
```

If FLD1 = 9.873, FLD2 = 1.1, and FLD3 = 5 (and FLD2 and FLD3 are defined as PIC 9(3)V9), the result is FLD2 = 10.9 and FLD3 = 49.3.

Example 2

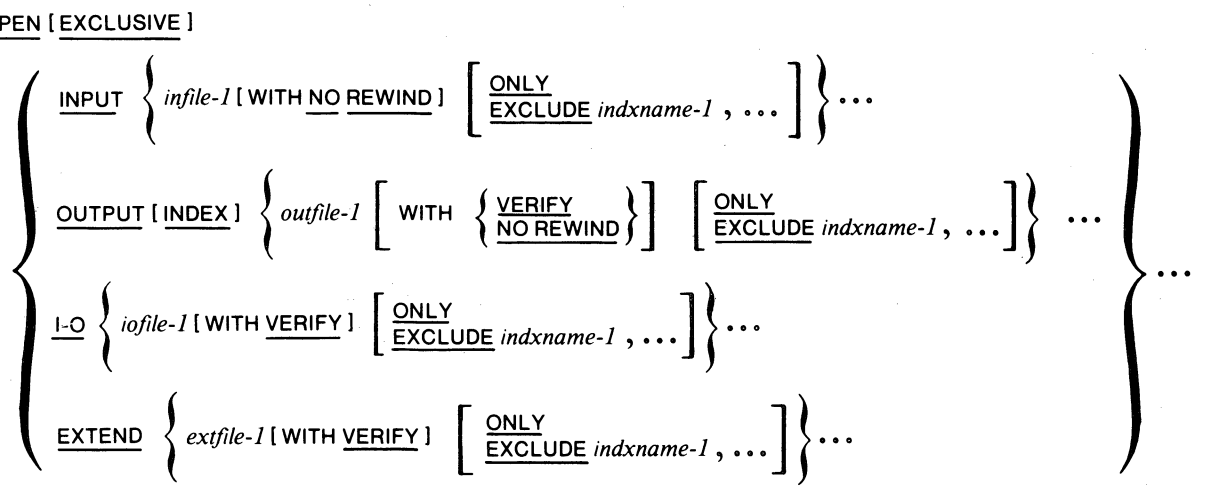
```
MULTIPLY A BY B GIVING C, D.
```

If A = 4, B = 5, C = 6, and D = 7, the result is C = 20 and D = 20.

OPEN

Initializes files for input and/or output operations.

Format



Where:

infile is a filename that specifies a file you want to open for input.

indxname is a filename that specifies an index you want to suppress buffer space for.

outfile is a filename that specifies a file you want to create for output.

iofile is a filename that specifies a file you want to open for input and output.

extfile is a filename that specifies a sequential file you want to open for extension.

Statement Execution

An OPEN statement puts a file in the open mode, where it remains until you issue a CLOSE statement for it. You may not request any input or output operations for a file unless it is open. You may not issue an OPEN statement for a file that your program has already opened.

You may specify the EXCLUSIVE option for sequential and relative files only. If you specify EXCLUSIVE and another program is using the specified file, COBOL cannot open that file. The file status field for that file will receive the error condition (see the section "File Status Registers" in Chapter 6). If you specify this option for a file that is not in use, statement execution opens the file and will not allow any other program to open it until you close it.

If you want only input operations performed on a file, specify the INPUT clause. Statement execution will set the current record pointer to the first record in the file.

If you want only output operations performed on a file, specify the OUTPUT clause. The file you specify must not already exist.

If you want to create an additional index (an inversion) for an indexed file, specify OUTPUT INDEX. In this case, the file specified must already exist.

If you want both input and output operations performed on a file, specify the I-O clause. The file you specify must exist. Statement execution will set the record pointer to the first record in the file.

If you want to write additional records to the end of a sequential file, specify the EXTEND clause. Statement execution will set the record pointer to the position immediately following the last logical record of that file.

The remaining options exist only in certain operating system environments. See Appendix B.

If you are opening a magnetic tape file and do not want the system to rewind the file, specify NO REWIND. If you omit this option, statement execution will position the file at its beginning.

When you open an indexed file and specify OUTPUT or I-O for that file, the system allocates buffer space for all the file's indexes. If you know that you will not be using all the file's indexes you can prevent the system from allocating extra buffer space by specifying ONLY. If you know you will be using some of the indexes, but not all, specify EXCLUDE with a list of the *indxnames* you do not need buffer space for.

If you are opening a disk file and want the system to read back every block that is output to the file and check it against the original to make certain they are identical, specify VERIFY.

Example

Example 1

```
OPEN I-O MAINTAIN.
```

This statement opens the file named MAINTAIN for input and output.

Example 2

```
OPEN EXCLUSIVE INPUT SAMFILE.
```

This statement opens the file named SAMFILE for input. If SAMFILE is OPENed by another program, this statement will not execute. SAMFILE must be an INFOS SAM or RAM file.

Example 3

```
OPEN OUTPUT INDX1 ONLY.
```

This statement opens the file named INDX1 for output and does not allocate extra buffer space for the file's indexes.

PERFORM

Transfers control explicitly to an internal subroutine, executes it simply or with looping, and returns control (implicitly) whenever execution of the specified procedure completes.

Format

```
PERFORM proc-1 [ { THRU    } proc-2 ]
                 { THROUGH }
   [ { ctr TIMES                                                                                      } ]
   [ { UNTIL condition                                                                                } ]
   [ { VARYING vctr-1 FROM vbase-1 BY vstep-1 UNTIL condition-1 [AFTER vctr-2 FROM vbase-2 BY vstep-2 UNTIL condition-2] ... } ]
```

Where:

proc is the name of a procedure paragraph or section you want to transfer control to.

ctr is an integer literal or an integer data item that specifies the number of times you want to execute the range of the PERFORM.

condition is any legal conditional expression (see the section "Conditional Expressions" in Chapter 6).

vctr is a numeric data item that specifies a counter.

vbase is a numeric literal or a numeric data item that specifies an initial value.

vstep is a nonzero numeric literal or numeric data item that specifies an increment.

Statement Execution

The range of a PERFORM statement is from the first statement in *proc-1* to the last statement in *proc-2,* (if you specify it). If you omit *proc-2,* the last statement in *proc-1* is the end of the range.

If you specify a simple PERFORM statement (one without the TIMES clause, UNTIL clause, or VARYING phrase), the range of the PERFORM executes once. Control then passes to the first executable statement following the PERFORM sentence.

If you specify a TIMES phrase, the PERFORM statement executes the range of the PERFORM the number of times specified by *ctr.* If *ctr* is zero or negative, COBOL never executes the range of the PERFORM and control passes to the first executable statement following the PERFORM sentence.

If you specify the UNTIL clause, the PERFORM statement executes the range of the PERFORM until the value of *condition* is true. When the condition is true, control passes to the first executable statement following the PERFORM sentence, even if control has just entered the PERFORM statement.

You specify the VARYING phrase to augment the values referenced by one or more data items in an orderly fashion during the execution of the PERFORM statement. If, in addition, you specify one or more AFTER phrases, the range of the PERFORM is within a nested loop. The innermost loop is the loop defined in the last AFTER phrase you specify.

As soon as the *condition* in the VARYING phrase is true, control passes to the first executable statement following the PERFORM sentence. If the condition is false, COBOL executes the range of the PERFORM unless you specify an AFTER phrase or phrases. In this case, COBOL evaluates the condition in the first AFTER phrase. If it is true, COBOL reevaluates the data items of the VARYING phrase and its condition. If the first AFTER phrase's condition is false, COBOL evaluates the condition in the next AFTER phrase, and so on. If all conditions are false the first time through a complete cycle, the innermost loop will vary most. See the flowchart in Figure 7-2 for a visual explanation.

You can nest PERFORM statements as long as the procedure(s) associated with the included PERFORM(s) is either totally included in or totally excluded from the logical sequence of the first PERFORM. A nested, active PERFORM statement may not pass control to the exit of another active PERFORM statement. No nested PERFORM statement may share a common exit with another PERFORM statement.

Example

```
PERFORM P VARYING C1 FROM B1 BY S1 UNTIL C1 > 50
   AFTER C2 FROM B2 BY S2 UNTIL C2 > 70
   AFTER C3 FROM B3 BY S3 UNTIL C3 > 90.
```

Figure 7-2 is a flowchart that shows the logical flow of the above example.

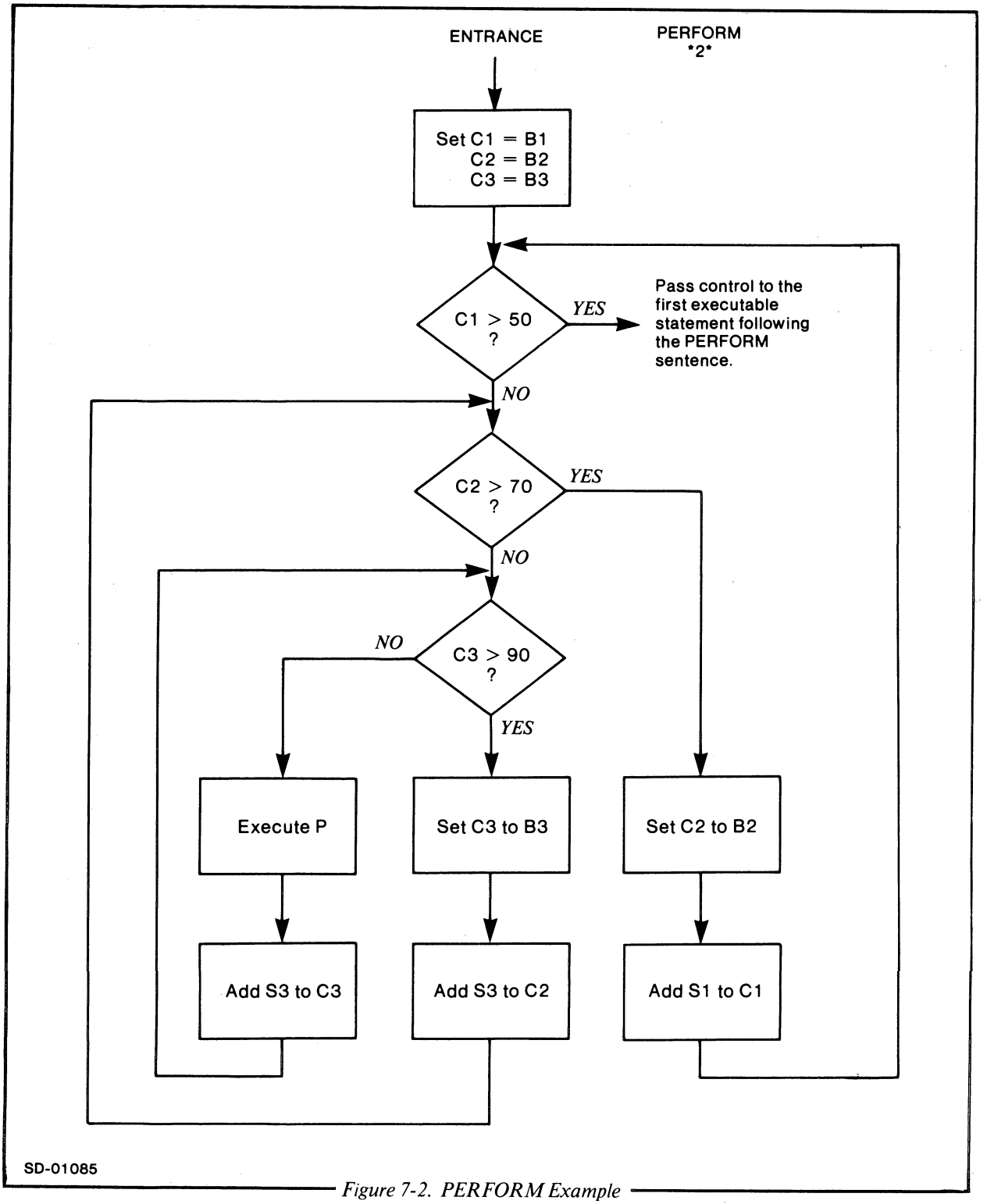


Figure 7-2. PERFORM Example

READ for a Sequential File

Makes a record available from a sequentially organized file.

Format

READ *file* NEXT RECORD [LOCK / UNLOCK] [INTO *dest*] [AT END *statement*]

Where:

file is a filename that specifies a sequential file OPENed for input or I/O.

dest is any data item specifying a destination that receives the file's record area.

statement is an imperative statement to which control passes if an end-of-file condition occurs.

Statement Execution

A READ statement for a sequential file reads into the specified file's record area the first record after the last one read. If you specify the INTO option, the READ will move the file's record area to *dest* according to MOVE rules. (See the MOVE statement earlier in this chapter.) If this is the first READ, it reads the first record of the file.

If you specify the LOCK option, no one can access the block containing the referenced record until you issue an I/O statement with UNLOCK for that same record, or until you CLOSE the file which automatically UNLOCKs the record. You must issue the corresponding LOCK and UNLOCK statements in the same program. LOCKing records varies from operating system to operating system (see Appendix B).

If you specify the AT END option and the READ operation reaches the end of the file, control passes to *statement*. If you omit this option, or if no end-of-file condition occurs, control passes to the first executable statement following the READ sentence.

Example

```
READ INVENTORY LOCK INTO WORKAREA.
```

This statement reads the next record in INVENTORY, and moves it into WORKAREA, locking the block that contains the record.

READ for a Relative File

Makes a record available from a relative file.

Format

```
READ file [NEXT] RECORD [LOCK | UNLOCK] [WAIT] [INTO dest] [{AT END | INVALID KEY} statement]
```

Where:

file is a filename that specifies a relative file OPENed for input or I/O.

dest is any data item specifying a destination that receives the file's record area.

statement is an imperative statement to which control passes if an end-of-file condition occurs or if you specify invalid record selection indicators (depending on which option you select).

Statement Execution

If you specify the NEXT option, a READ statement for a relative file reads into the specified file's record area the first record after the last one read. If this is the first READ, it reads the first record of the file.

If you do not specify the NEXT option, COBOL determines the record to read by an integer, *n*, specified in the file's relative key. The READ statement reads the *nth* record, counted from the beginning of the file, into the file's record area.

If you specify the INTO option, the READ will move the file's record area into *dest* according to MOVE rules. (See the MOVE statement earlier in this chapter.)

If you specify the LOCK option, no one can access the block containing the referenced record until you issue an I/O statement with UNLOCK for that same record, or until you CLOSE the file which automatically UNLOCKs the record. You must issue the corresponding LOCK and UNLOCK statements in the same program.

If you specify the WAIT option, program execution temporarily suspends if the READ statement attempts to access a locked record. Program execution resumes when the user of that record UNLOCKS it. If you do not specify WAIT, an exception condition occurs when the program attempts to read a locked record (see the section "Handling I/O Exception Conditions" in Chapter 6).

The LOCK and WAIT options vary from operating system to operating system (see Appendix B).

If you are reading the file in sequential order, you may specify the AT END phrase to capture control if a READ reaches the end of the file. If you specify this option and an end-of-file condition occurs, control passes to *statement.* If you omit this option, or if no end-of-file condition occurs, control passes to the first executable statement following the READ sentence. The AT END option varies from operating system to operating system (see Appendix B).

If you are reading the file in random order, you may specify the INVALID KEY phrase to capture control if the record selection indicators are invalid (i.e., if they reference a record or key that does not exist). If you specify this option and an invalid key condition occurs, execution of the READ statement terminates and control passes to *statement.* If you omit this option, or if no invalid key condition occurs, control passes to the first executable statement following the READ sentence.

Example

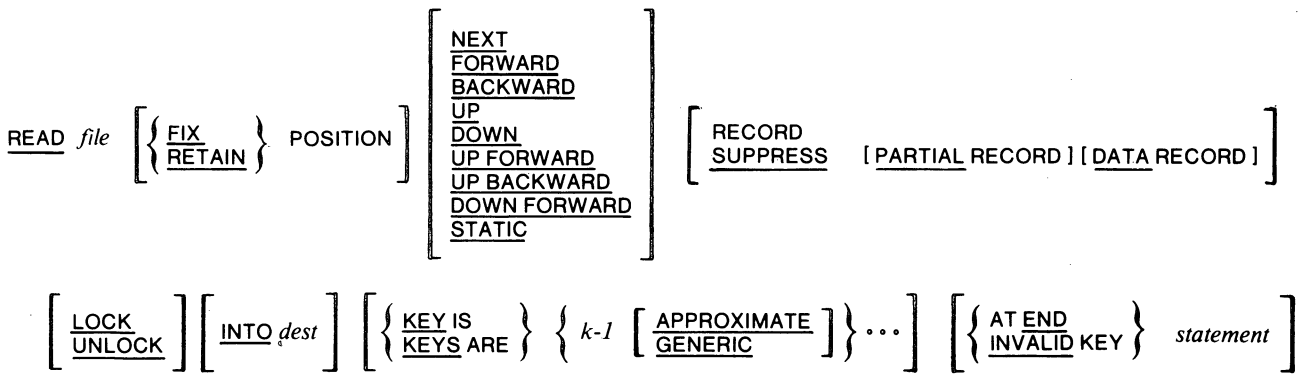
```
READ NAMES INTO LOC100
   INVALID KEY "KEY DOES NOT EXIST".
```

This statement reads the record in NAMES which is specified by the current value of the relative key, and moves the file's record area into LOC100. If the record or key referenced does not exist, the message KEY DOES NOT EXIST is displayed on the current console.

READ for an Indexed File

Makes a record available from an indexed file.

Format



Where:

file is a filename that specifies an indexed file OPENed for input or I/O.

dest is any data item specifying a destination that receives the file's record area.

k is an alphanumeric data item that specifies a key used to determine a record's location.

statement is an imperative statement to which control passes if an end-of-file condition occurs or if you specify invalid record selection indicators (depending on which option you select.)

Statement Execution

A READ statement for an indexed file reads a record into the file's record area. COBOL determines the record according to what you specify (explicitly or implicitly) in the POSITION phrase, the relative options phrase (NEXT, FORWARD, etc.), and the KEY series phrase. If you specify the INTO option, the READ will move the file's record area to *dest* according to MOVE rules. (See the MOVE statement earlier in this chapter.)

By specifying FIX POSITION, you set the record pointer to the record specified in the KEY series phrase or the relative options phrase (discussed below). If you specify RETAIN POSITION, you do not change the current position of the record pointer (i.e., it points to the record for which you last set it). If you omit this option, the default is FIX POSITION.

When you specify a relative options phrase, you reference a record in an indexed file, relative to the current position of the file's record pointer. If you omit both this option and the KEY series option, the default is STATIC.

If you specify the KEY series phrase, you must have declared each key *(k)* in this file's SELECT clause.

For more information on these options, see the section "Indexed File Record Selection" in Chapter 6.

If you specify the LOCK option, no one can access the block containing the referenced record until you issue an I/O statement with UNLOCK for that record, or until you CLOSE the file which automatically UNLOCKs the record. You must issue the corresponding LOCK and UNLOCK statements in the same program. LOCKing records varies from operating system to operating system (see Appendix B).

If you specify SUPPRESS PARTIAL RECORD, COBOL will not retrieve the partial record associated with the referenced index entry from the file's partial record area (which you defined in the file's FD entry in the Data Division). If you specify SUPPRESS DATA RECORD, COBOL will not input the data record associated with the referenced index entry to the file's record area. You may specify both of these options in any order. Specifying them both lets you position the record pointer, and updates the FEEDBACK item in the FD entry for this file; this is one way you can generate a new inversion in an indexed file.

If you are reading the file in sequential order, you may specify the AT END phrase to capture control if the READ reaches the end of the file or subindex. If you specify this option and an end-of-file condition occurs, control passes to *statement.* If you omit this option, or if no end-of-file condition occurs, control passes to the first executable statement following the READ sentence.

If you are reading the file in random order, you may specify the INVALID KEY phrase to capture control if the record selection indicators are invalid (i.e., if no record is found that has a key value equal to that of the key of reference). If you specify this option and an invalid key condition occurs, execution of the READ statement terminates and control passes to *statement.* If you omit this option, or if no invalid key condition occurs, control passes to the first executable statement following the READ sentence.

Example

```
READ FILE-09 FIX POSITION
   SUPPRESS DATA RECORD
   KEY IS REM02.
```

This statement reads the partial record of key REM02 into FILE-09's partial record area (declared in the SELECT statement) without reading the data record associated with that key. The record pointer is set at this key entry.

RELEASE

Passes records to the initial phase of a SORT operation.

Format

RELEASE *sortrec* [FROM *src*]

Where:

sortrec is an alphanumeric data item specifying a record from the sort/merge description entry which is given in the controlling SORT statement.

src is any data item or literal that specifies a source you want to pass.

Statement Execution

A RELEASE statement releases the contents of *sortrec* as an input record to the initial phase of the SORT operation. If you specify the FROM option, RELEASE moves the contents of *src* to *sortrec* according to MOVE rules prior to performing the RELEASE operation. (See the MOVE statement earlier in this chapter.)

You may issue a RELEASE statement only within the range of an input procedure associated with a SORT statement. *Sortrec* and *src* must not reference the same storage area.

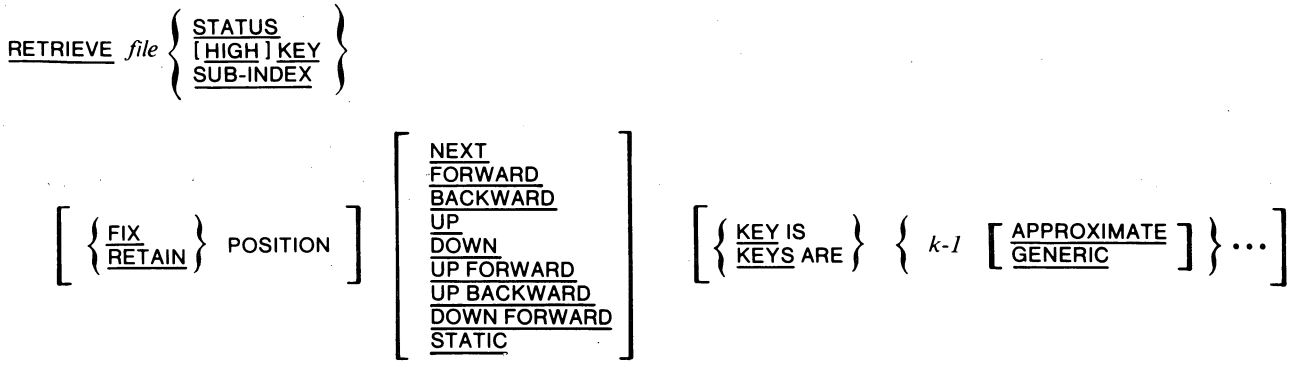
Example

```
RELEASE REC01.
```

RETRIEVE

Obtains information about an indexed file.

Format



Where:

file is a filename that specifies an open indexed file.

k is an alphanumeric data item that specifies a record key associated with *file.*

dest is any data item specifying a destination that receives record or key status information.

statement is an imperative statement to which control passes if you specify invalid record selection indicators.

Statement Execution

COBOL determines the record interpreted by a RETRIEVE statement according to what you specify (explicitly or implicitly) in the POSITION phrase, the relative options phrase (NEXT, FORWARD, etc.), and/or the KEY series phrase.

By specifying FIX POSITION, you set the record pointer to the record specified by this statement. If you specify RETAIN POSITION, you do not change the current position of the record pointer (i.e., it points to the record for which you last set it). If you omit this option, the default is FIX POSITION.

When you specify a relative option, you reference a record in *file,* relative to the current position of *file's* record pointer. If you omit both this option and the KEY series option, the default is STATIC.

If you specify the KEY series phrase, you must have declared each key *(k)* in the SELECT statement for *file.*

For more information on these options, see the section "Indexed File Record Selection" in Chapter 6.

The type of information you receive from the execution of a RETRIEVE statement depends on which of four types you select: STATUS, HIGH KEY, KEY, or SUB-INDEX. If you specify STATUS, COBOL interprets *dest* as a four-character data item and stores either a 1 (if the condition is true) or a 0 (if the condition is false) in each character position, to signify the following conditions:

RETRIEVED (continued)

1st character (leftmost) - a LOGICAL LOCAL DELETE was executed for this record;

2nd character - the key is a duplicate;

3rd character - the data record is LOCKed;

4th character (rightmost) - a LOGICAL GLOBAL DELETE was executed for this record.

Note that LOCKed records are handled differently from operating system to operating system. See Appendix B for more information.

If you specify HIGH KEY, RETRIEVE moves the value of the highest key in the subindex associated with the selected record into *dest* (which therefore, must be large enough to contain the largest key).

If you specify KEY, RETRIEVE moves the key value of the selected record into *dest.*

If you specify either HIGH KEY or KEY, completion of the RETRIEVE statement will update two variables in the KEY IS phrase of the SELECT statement associated with *file:*

1. The *reckey* data item that names either the last *k* you specified in the KEY IS option of the RETRIEVE statement or, if you omitted this option,

2. The first key *(reckey-1)* you specified in the KEY IS phrase of the SELECT statement.

   The variables that RETRIEVE updates are *keylen* which will contain the length of the RETRIEVEd record's key, and *occ* which will contain the occurrence number of the RETRIEVEd key.

If you specify SUB-INDEX, RETRIEVE moves into *dest* the subindex definition packet associated with the key you specified in the KEY series phrase. The size of this packet is operating system dependent (see Appendix B).

If you specify record selection indicators which reference a record that does not exist, and if you specify the INVALID KEY option, execution of the RETRIEVE statement terminates and control passes to *statement.* If you omit this option, or if no invalid key condition occurs, control passes to the first executable statement following the RETRIEVE sentence.

Examples

Example 1

```
MOVE 1 TO KEY01.
RETRIEVE IND02 KEY, KEY IS KEY01 INTO DEST.
```

This statement retrieves the key from file IND02 whose value is equal to one, and moves the contents of that key to DEST. You might use this implementation of a RETRIEVE statement to ensure that your current position in the file is at the record whose key is equal to one.

Example 2

```
RETRIEVE IND02 SUB-INDEX DOWN KEY IS KEY01 INTO SUB1.
```

The subindex definition packet for the subindex associated with KEY01 is returned in SUB1. (This example assumes that a subindex exists for KEY01.)

Example 3

```
RETRIEVE IND02 STATUS KEY IS KEY01 INTO STAT1.
```

If after execution of this statement STAT1 = 0100, KEY01 is a duplicate key.

RETURN

Obtains sorted records from the final phase of a SORT operation or merged records during a MERGE operation.

Format

RETURN *smfile* RECORD [INTO *dest*] [AT END *statement*]

Where:

smfile is a filename that specifies a sort/merge file given in the controlling SORT or MERGE statement.

dest is any data item specifying a destination that receives the file's record area.

statement is an imperative statement to which control passes if an end-of-file condition occurs.

Statement Execution

A RETURN statement transfers the next record from the final phase of the SORT or MERGE operation into the record area associated with *smfile.*

If you specify the INTO clause, RETURN moves the contents of the file's record area to *dest* according to MOVE rules, after successful execution of the RETURN statement. (See the MOVE statement earlier in this chapter.) However, this move will not occur if there is an end-of-file condition.

You may issue a RETURN statement only within the range of an output procedure associated with a SORT or MERGE statement. *Smfile* and *dest* must not reference the same storage area.

If no logical record exists for the file at the time of the RETURN statement's execution, and if you specify the AT END option, control passes to *statement* and the contents of the file's record area remain undefined. After execution of *statement,* you may not specify any RETURN statements in the current output procedure. If you omit this option, or if no end-of-file condition occurs, control passes to the first executable statement following the RETURN sentence.

Example

```
RETURN MRG-FILE RECORD.
```

This statement passes the next record to be processed to the record area for MRG-FILE, a name specified in the controlling SORT or MERGE statement.

REWRITE for a Sequential File

Writes a new version of a record already existing in a sequentially organized file.

Format

REWRITE *rec* [FROM *src*]

Where:

rec is a data name that specifies a logical record associated with a file declared in the File Section of the Data Division, and OPENed for I/O.

src is any data item or literal that specifies the source you want to write.

Statement Execution

A REWRITE statement for a sequential file rewrites the last record COBOL read from that file. Following the REWRITE, the current record is the record after the one just written. If you specify the FROM option, COBOL moves the contents of *src* to *rec* according to MOVE rules, prior to performing the REWRITE operation. (See the MOVE statement earlier in this chapter.) After execution, COBOL updates the value of the file status register, if you specified it in this file's SELECT statement. (See the sections on File Status Registers in Chapter 6.)

Src and *rec* must not reference the same storage area. The number of character positions in *rec* must be the same as the number of characters in the record you are rewriting.

Example

```
REWRITE INFO FROM NEW-INFO.
```

This statement moves the contents of the record NEW-INFO to the record INFO. COBOL then rewrites the record last read in the sequential file associated with the record INFO.

REWRITE for a Relative File

Writes a new version of a record associated with a key that already exists in a relative file.

Format

REWRITE *rec* [IMMEDIATE] [FROM *src*] [INVALID KEY *statement*]

Where:

rec is a data name that specifies a logical record associated with a file declared in the File Section of the Data Division and OPENed for I/O.

src is any data item or literal that specifies the source you want to write.

statement is an imperative statement to which control passes if the record selection indicators are invalid.

Statement Execution

A REWRITE statement for a relative file that you are accessing sequentially rewrites the record that COBOL last read. Following the REWRITE, the current record is the record after the one just written. If you specify the FROM option, COBOL moves the contents of *src* to *rec* according to MOVE rules, prior to performing the REWRITE operation. (See the MOVE statement earlier in this chapter.) After execution, COBOL updates the value of the file status register, if you specified it in this file's SELECT statement. (See the sections on File Status Registers in Chapter 6.)

Src and *rec* must not reference the same storage area. The number of character positions in *rec* must be the same as the number of characters in the record you are rewriting.

A REWRITE statement for a relative file that you are accessing randomly rewrites the record specified by the value of the relative key for this file.

If you specify IMMEDIATE, the REWRITE statement immediately writes the block containing the record out to the file, even though the next I/O operation may reference the same block. This option trades I/O efficiency for extra data security. The IMMEDIATE phrase is system dependent (see Appendix B).

An invalid key condition occurs if, in random or dynamic access mode, the record specified by the key does not exist in the file. If you specify the INVALID KEY option and if the above condition occurs, the REWRITE statement terminates and control passes to *statement.* If you omit this option or if no invalid key condition occurs, control passes to the first executable statement following the REWRITE sentence. Do not specify the INVALID KEY phrase for a relative file that you are accessing sequentially.

Example

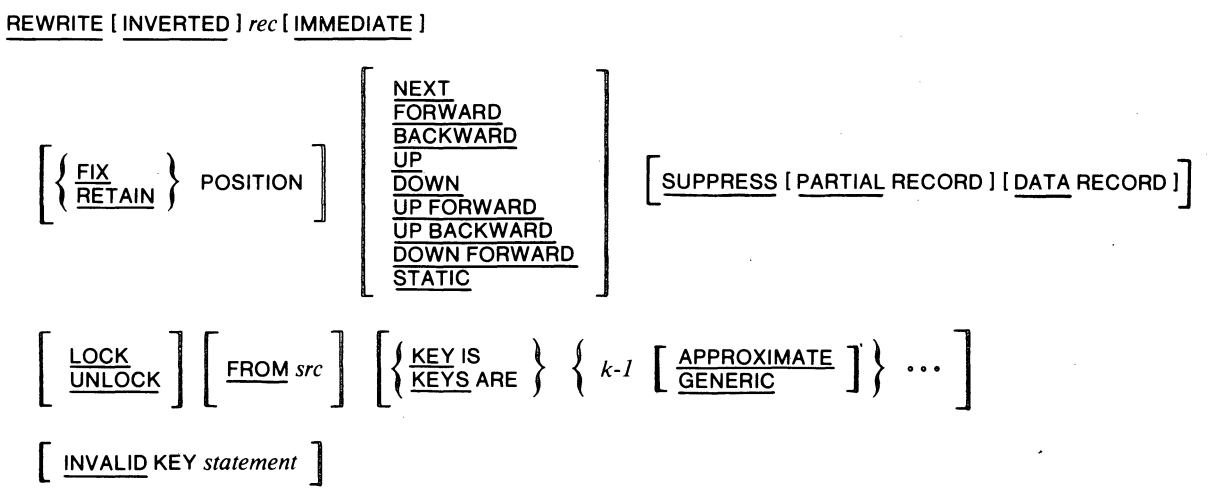
```
REWRITE COMP1 IMMEDIATE INVALID KEY DISPLAY "KEY ERROR".
```

Before processing any other I/O operations, COBOL rewrites the contents of the last record it read in the current relative file to contain COMP1. If the relative key that is referenced does not exist, the message "KEY ERROR" is displayed on the current console.

REWRITE for an Indexed File

Writes a new version of a record associated with a key that already exists in an indexed file.

Format



Where:

rec is a data name that specifies a logical record declared in the File Section of the Data Division and OPENed for I/O.

src is any data item or literal that specifies the source you want to write.

k is an alphanumeric data item that specifies a record key associated with the current indexed file.

statement is an imperative statement to which control passes if you specify invalid record selection indicators.

Statement Execution

If you specify INVERTED, the REWRITE statement does not write a data record. You use this feature to link an existing index entry with no data record to a currently existing data record. To use this option you must specify the FEEDBACK phrase in the FD entry for this file.

If you do not specify INVERTED, COBOL writes a record in a location determined by what you specify (explicitly or implicitly) in the POSITION phrase, the relative option phrase (NEXT, FORWARD, etc.), and the KEY series phrase. If you specify the FROM option, COBOL moves the contents of *src* to *rec* according to MOVE rules, prior to performing the REWRITE operation. (See the MOVE statement earlier in this chapter.) After execution, COBOL updates the value of the file status register, if you specified it in this file's SELECT statement. (See the section on File Status Registers in Chapter 6.)

Src and *rec* must not reference the same storage area. The number of character positions in *rec* must be the same as the number of characters in the record you are rewriting.

By specifying FIX POSITION, you set the record pointer to the record specified in the KEY series phrase (discussed below). If you specify RETAIN POSITION, you do not change the current position of the record pointer (i.e., it points to the record for which you last set it). If you omit this option, the default is RETAIN POSITION.

When you specify a relative option, you reference a record in an indexed file, relative to the current position of the file's record pointer. If you omit both this option and the KEY series option, the default is STATIC.

If you specify the KEY series phrase, you must have declared each key *(k)* in this file's SELECT clause. However, if the indexed file has alternate record keys, the system keys the REWRITE operation by the prime record key (defined in the RECORD KEY clause of this file's SELECT clause). You need not specify this key in the KEY series phrase.

For more information on these options, see the section "Indexed File Record Selection" in Chapter 6.

If you specify the LOCK option, no one can access the referenced record until you issue an I/O statement with UNLOCK for that record, or until you CLOSE the file which automatically UNLOCKs the record. You must issue the corresponding LOCK and UNLOCK statements in the same program. LOCKing of records varies from operating system to operating system (see Appendix B).

If you specify SUPPRESS PARTIAL RECORD, COBOL will not output the partial record associated with the referenced index entry to the file's partial record area (which you specified in the file's FD entry in the Data Division). If you specify SUPPRESS DATA RECORD, COBOL will not output the data record associated with the referenced index entry to the file's record area. The current value of the feedback item was set by a previous access to this file.

If you specify IMMEDIATE, the REWRITE statement immediately writes the block containing the record out to the file, even though the next I/O operation may reference the same block. This option trades I/O efficiency for extra data security.

An invalid key condition exists when, in sequential access mode, the value contained in the prime record key data item of the record to be rewritten is not equal to the value of the prime record key of the last record read from this file. It exists in random or dynamic access mode, when the value contained in the prime record key data item does not equal that of any record stored in the file. If you specify the INVALID KEY option and any of the above conditions occur, execution of the REWRITE statement terminates and control passes to *statement*. If you omit this option or if no invalid key condition occurs, control passes to the first executable statement following the REWRITE sentence.

Example

```
REWRITE CUST01 FIX SUPPRESS PARTIAL RECORD FROM NEW-CUST KEY IS KEY01.
```

This statement moves the contents of NEW-CUST to CUST01, sets the record pointer to the record referenced by KEY01, and rewrites the contents of that record with the contents of CUST01, suppressing partial records.

SEARCH

Locates an element in a table that satisfies specified conditions.

Format

```
SEARCH { [ALL] elemt                    } [AT END statement1]
       { elemt [VARYING ndx]            }

{ WHEN condition-1 { statement2      } } ...
                   { NEXT SENTENCE   }
```

Where:

elemt is an unsubscripted array name whose data description must contain an OCCURS clause specifying the INDEXED BY phrase.

ndx is an integer data item that specifies an index item.

statement1 is an imperative statement to which control passes if *elemt* contains a value greater than the highest possible occurrence number.

condition is any legal conditional expression (see the section "Conditional Expressions" in Chapter 6).

statement2 is an imperative statement to which control passes if all specified *conditions* are satisfied.

Statement Execution

If you reference a table element that is less than or equal to the maximum number of occurrences which you specified in the OCCURS clause of *elemt's* data definition, then the SEARCH statement will evaluate the *conditions* in the order in which you write them. If none of the conditions is true, COBOL increments the table element to reference the next occurrence. The SEARCH statement then repeats this whole process for the new table element, unless it is greater than the highest possible occurrence number. For more information on the OCCURS clause, see the section "OCCURS Clause" in Chapter 5.

If you specify or attain a table element that is greater than the highest possible occurrence number, the SEARCH operation terminates immediately. If you specify the AT END option, control passes to *statement1*. If you omit this option or if you do not go outside the legal range, control passes to the first executable statement following the SEARCH sentence.

If one of the *conditions* in a SEARCH operation is true, the search terminates immediately and control passes to the imperative statement associated with that condition. The table element remains set at the occurrence which satisfied the condition. Control then passes to the first executable statement following the SEARCH sentence, unless the imperative statement transfers control elsewhere.

If you specify the ALL option, COBOL initializes the index item *(i-1* in the INDEXED BY phrase of *elemt's* OCCURS clause) to 1.

If you specify VARYING *ndx* and if *ndx* is one of the index items listed in *elemt's* INDEXED BY phrase, then the SEARCH operation varies that index item instead of *i-1*. If *ndx* is not one of the index items for *elemt* or if you omit the VARYING phrase, then the SEARCH operation will use *i-1* as the index, and COBOL will increment *ndx* every time it increments *i-1*.

If you specify the NEXT SENTENCE phrase, it performs no operation, but passes control to the first executable statement following the SEARCH sentence.

Example

Given the data descriptions:

```
01 A.
   02 B OCCURS 200 TIMES INDEXED BY I,J.
      03 B1 PIC S99.
      03 B2 PIC X.
```

you might write the statements:

```
SEARCH B VARYING J
        AT END DISPLAY "NOT FOUND",
                       STOP RUN;
        WHEN B1 (J) > - 12
                       MOVE B(J) TO REC,
                       WRITE REC;
        WHEN B2 (J) = "A" AND B1(J) NOT = 0
                       SET A-CTR UP BY 1,
                       MOVE -1 TO B1(J).

SEARCH B VARYING K
        WHEN B2(I) = "C"
                       SET TAB (K) TO 1.
```

SEEK

Positions the I/O system at that record in the relative file which is indicated by the current value of the file's relative key.

Format

SEEK *file* RECORD [INVALID KEY *statement*]

Where:

file is a filename that specifies a relative file OPENed for input or I/O.

statement is an imperative statement to which control passes if record selection indicators are invalid.

Statement Execution

NOTE: Use of this statement depends on your operating system environment - see Appendix B.

The SEEK statement overlaps I/O activity when you are randomly processing a relative file.

If you specify the INVALID KEY option and the current value of the file's relative key references a record that does not exist, then execution of the SEEK statement terminates and control passes to *statement.* If you omit this option, or if no invalid key condition occurs, control passes to the first executable statement following the SEEK sentence.

SET

Is an inverted form of the MOVE statement. It sets one or more data items equal to another data item.

Format

SET *dest-1* , ... TO *src*

Where:

dest is an index, numeric, or alphanumeric data item that receives a data source.

src is an index, numeric, or alphanumeric data item that specifies a data source you want to move.

Statement Execution

A SET statement moves the data source to the first (or only) destination item, then to the second (if you specified one), etc. COBOL evaluates any subscripting or indexing associated with *dest* before moving the data.

If you specify a group data item as either the source or destination, COBOL executes the SET statement as a character string move with no editing.

See the MOVE statement for examples and a detailed explanation of character string and numeric moves.

SET UP/DOWN

Adds an operand to or subtracts an operand from one or more operands and stores the results.

Format

SET *opres-1* , ... { UP / DOWN } BY *op*

Where:

opres is a numeric data item that specifies an addend or minuend and that receives the result of the addition or subtraction.

op is a numeric literal or a numeric data item that specifies an addend or subtrahend.

Statement Execution

A SET UP statement adds the addend *(op)* to each addend/result *(opres),* and stores each result in the corresponding *opres* according to MOVE rules. (See the MOVE statement earlier in this chapter.) The SET UP statement is equivalent to the statement:

ADD *op* TO *opres-1*, ...

A SET DOWN statement subtracts the subtrahend *(op)* from the minuend/result *(opres)* and stores each result in the corresponding *opres* according to MOVE rules. The SET DOWN statement is equivalent to the statement:

SUBTRACT *op* FROM *opres-1*...

Examples

Example 1

```
SET RES1, RES2 UP BY A.
```

If RES1 = 4, RES2 = 11, and A = 3, the result is RES1 = 7 and RES2 = 14.

Example 2

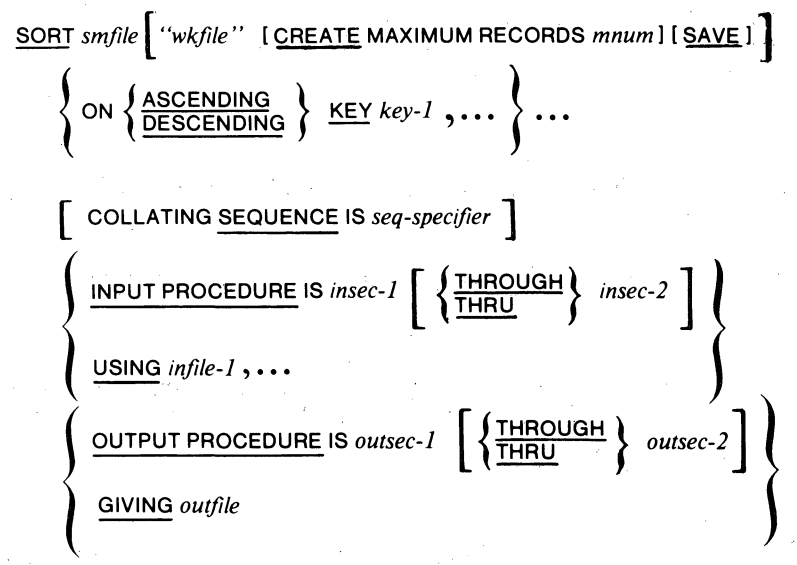
```
SET A, B, C DOWN BY D.
```

If A = 2, B = 4, C = 6, and D = 1, the result is A = 1, B = 3, and C = 5.

SORT

Arranges one or more files in a sorted order according to a set of specified keys.

Format



Where:

smfile is a filename that specifies a sort/merge file described in a sort/merge file description entry in the Data Division.

wkfile is an alphanumeric literal that specifies a work file which receives a sorted copy of *smfile.*

mnum is a numeric data item that specifies the number of records you want to allocate for *wkfile.*

key is an alphanumeric data item that specifies a key described in a record associated with *smfile.*

seq-specifier specifies the program collating sequence you want used when COBOL is making comparisons during the SORT operation and is one of the following: ASCII, NATIVE, STANDARD-1, EBCDIC, and *alph.*

insec is a Procedure Division section name specifying an input procedure which COBOL executes as the initial phase of the SORT operation.

infile is a filename specifying a file which is not a sort/merge file, does not have subindexing or alternate record keys, and is not open at the time of the SORT operation's execution.

outsec is a Procedure Division section name specifying an output procedure which COBOL executes as the final phase of the SORT operation.

outfile is a filename specifying a sequential file that either does not already exist, or exists but is a closed, empty inversion of a currently existing indexed file.

Statement Execution

A SORT statement creates a work file by executing input procedures which release records to the work file or by transferring records from another file to the work file. The SORT operation then arranges the records in order according to specified keys, and makes the sorted records available to an output procedure or passes them to an output file. Before the SORT operation begins, COBOL closes all open files. Completion of the SORT does not re-open these files.

If you do not specify *wkfile,* the action the SORT operation takes depends on the operating system you are running under. See Appendix B.

You must specify ASCENDING or DESCENDING to indicate the comparison you want the SORT operation to perform when it arranges the records. When comparing the key values in two records, COBOL uses the rules for evaluating relational expressions in which there are three comparisons: greater than, less than, and equal to. If you specify ASCENDING, the SORT operation arranges the records so that those with lower key values will come first. If you specify DESCENDING, SORT arranges the records so that those with greater key values will come first. The sorted order of records is undefined if the SORT operation compares two records where all keys are identical.

The *keys* you specify represent the portion of the data records COBOL uses in the comparison process. The first *key* you specify is the most important key. It is the first key the SORT operation uses when comparing two records. The second *key* you specify is the next most significant key, the third is the third most significant key, and so on. The SORT operation uses these keys only if the value of the first key in the records it is comparing is the same.

If you specify the COLLATING SEQUENCE phrase, the *seq-specifiers* have the same meanings as the specifiers used in the PROGRAM COLLATING SEQUENCE clause of the Environment Division (see the section "Object Computer Paragraph" in Chapter 4), except that they apply only to comparisons made during the SORT statement's execution. If you omit this option in the SORT statement, the SORT operation uses what you specified in the PROGRAM COLLATING SEQUENCE in the Environment Division. If you omit both options, the default is ASCII.

If you specify the INPUT PROCEDURE and OUTPUT PROCEDURE clauses, SORT executes the specified procedures according to the rules for a simple PERFORM (see the PERFORM statement earlier in this chapter). Each set of sections you specify for the input and the output procedures must appear contiguously in your program. You must not pass control out of either the input or output procedure. The words THRU and THROUGH are equivalent.

The SORT operation performs the input procedure *(insec* or *insec-1* THROUGH *insec-n),* obtains records by executing RELEASE statements within the input procedure, and transfers the records to *wkfile.* COBOL organizes the records in *wkfile* in the specified order. Then the SORT operation performs the output procedure *(outsec* or *outsec-1* THROUGH *outsec-n)* and obtains records from *wkfile* by executing RETURN statements within the output procedure. The output procedure may include any process necessary to select, modify, or copy records in *wkfile.*

If you specify the USING and GIVING clauses, SORT reads the records of the *infile(s)* and transfers them to *wkfile* in a single, sorted order. The SORT operation then writes these records, as they are made available, to *outfile.*

The logical records in the *infile(s)* and in *outfile* must be equal in size to the logical records of *smfile* and *wkfile.*

SORT statements may appear anywhere in the Procedure Division of your program, except in the declaratives portion or in an input or output procedure associated with a SORT or MERGE statement.

After completion of the SORT operation, COBOL copies the contents of *wkfile* to *smfile* and control passes to the first executable statement following the SORT statement. If you specify SAVE, COBOL does not delete the work file on completion of the SORT operation. If you omit this option, COBOL deletes it.

Example

```
PROCEDURE DIVISION.
I-1.
   SORT SORTFILE-1A
      ON ASCENDING KEY KEY1, KEY2
      ON DESCENDING KEY KEY3, KEY4
      INPUT PROCEDURE IS INSORT
      OUTPUT PROCEDURE IS OUTP1 THRU OUTP2.

I-2.
   STOP RUN.

INSORT SECTION.
IN-1.
   SUBTRACT 1 FROM QUANTITY.
   RELEASE S-RECORD.
   MOVE 9999 TO ORDER-FLD.
   RELEASE S-RECORD.

IN-2.
   PERFORM IN-3 2 TIMES.
   GO TO IN-EXIT.

IN-3.
   MOVE CURDATE TO UPDATE-FLD.
   IF UPDATE-FLD IS GREATER THAN
      LAST-DATE GO TO IN-EXIT.
   RELEASE S-RECORD.

IN-EXIT.
   EXIT.

OUTP1.
   IF DELETE-CNT IS EQUAL TO ZERO
      MOVE "NO" TO ERROR-TOTAL ELSE
      MOVE DELETE-CNT TO ERROR-TOTAL.
   MOVE IN-PART TO OUT-PART.
   MOVE IN-NUM TO OUT-NUM.
   MOVE INFO-LINE TO DUMMY-RECORD.
   WRITE DUMMY-RECORD AFTER
      ADVANCING 3 LINES.

OUTP2.
   IF ERROR-COUNTER IS EQUAL TO
      ZERO GO TO OUTP3.
   RETURN SORTFILE-1A.

OUTP3.
   CLOSE SORTFILE-1A.
   EXIT.
```

START for a Sequential File

Positions the record pointer in a sequentially organized file.

Format

START *file* BLOCK *blk* [CHARACTER *chr*] [AT END *statement*]

Where:

file is a filename that specifies a sequential file on a direct access device, OPENed for input or I/O.

blk is an integer literal or an integer data item that specifies a logical block number.

chr is an integer literal or an integer data item that specifies a character offset within *blk*.

statement is an imperative statement to which control passes if you specify a position that is not within the given file.

Statement Execution

A START statement for a sequential file positions the file so that the next READ issued for that file will begin at the position *blk*, offset by *chr*.

The first logical block of a sequential file is numbered 0. The first character in the block is numbered 0. If you do not specify *chr*, the default value is 0.

If you specify the AT END option, and the position you indicate is outside this file, then control passes to *statement*. The position of the record pointer remains undefined. If you omit this option, or if no AT END condition occurs, control passes to the first executable statement following the START sentence.

Example

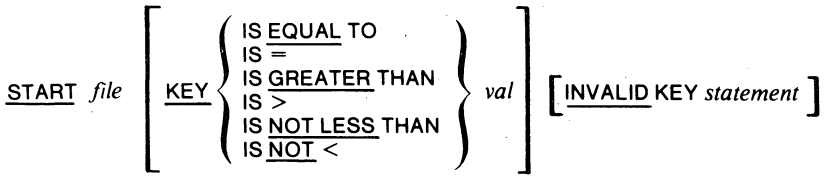
```
START MYFILE BLOCK 6 CHARACTER 62.
```

After execution of this statement, the next READ issued for MYFILE will begin at the 63rd character position in the seventh logical block.

START for a Relative file

Positions the record pointer in a relative file.

Format



Where:

file is a filename that specifies a relative file OPENed for input or I/O.

val is an integer data item specifying a key that appears in the RELATIVE KEY phrase of the file's SELECT clause.

statement is an imperative statement to which control passes if the record selection indicators are invalid.

NOTE: We do not underline the required relational characters >, <, and = in order to avoid confusion with other symbols such as ≥ (greater than or equal to).

Statement Execution

A START statement for a relative file positions the file to the beginning of the record indicated by the KEY phrase. If you specify EQUAL, START positions the file at the record indicated by *val,* if it exists. If you specify GREATER, START positions the file at the record whose key is the next key higher than the one indicated by *val* and of the same length as *val,* if it exists. If you specify NOT LESS, START positions the file to the record indicated by *val.* If you omit the KEY phrase, the default is EQUAL.

The START statement updates the value of the file status register, if you specified it in the SELECT statement for this file (see the sections on File Status Registers in Chapter 6).

If you specify the INVALID KEY option and no existing record satisfies the stated comparison, control passes to *statement.* The position of the record pointer remains undefined. If you omit this option, or if no invalid key condition occurs, control passes to the first executable statement following the START sentence.

Example

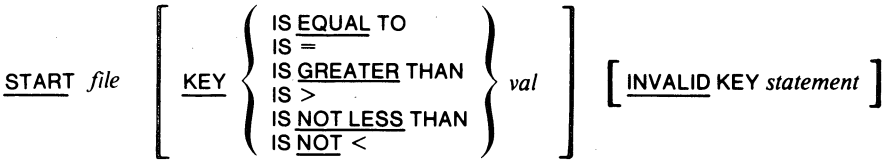
```
START INC99 KEY GREATER IT.
```

This statement positions the record pointer in INC99 to the record whose key is the next key greater than IT, and of the same length as IT.

START for an Indexed File

Positions the record pointer in an indexed file.

Format



Where:

file is a filename that specifies an indexed file, with sequential or dynamic access, and OPENed for input or I/O.

val is an alphanumeric data item that specifies either a record key associated with *file* or a subordinate data item whose leftmost character position corresponds to the leftmost character position of a record key data item.

statement is an imperative statement to which control passes if the record selection indicators are invalid.

NOTE: We do not underline the required relational characters >, <, and = in order to avoid confusion with other symbols such as ≥ (greater than or equal to).

Statement Execution

A START statement for an indexed file positions the file to the beginning of the record indicated by the KEY phrase. If you specify EQUAL, START positions the file at the record indicated by *val,* if it exists. If you specify GREATER, START positions the file at the record whose key is the next key higher than the one indicated by *val* and of the same length as *val,* if it exists. START also updates the value of *val* to the value of the new key. If you specify NOT LESS, START positions the file to the record indicated by *val,* or, if that does not exist, to the record whose key is the next key higher. START also updates the value of *val* to the value of the new key. If you omit the KEY phrase, the default is EQUAL and COBOL uses the data item you specified in the RELATIVE KEY clause associated in this file's SELECT clause for comparison.

If *val* and the record key it references are of unequal length, START truncates the longer one on the right so that its length is equal to the shorter one's and the relational comparison proceeds.

A START statement updates the value of the file status register, if you specified it in this file's SELECT statement (see the sections on File Status Registers in Chapter 6).

If you specify the INVALID KEY option and no existing record satisfies the stated comparison, control passes to *statement.* The position of the record pointer remains undefined. If you omit this option, or if no invalid key condition occurs, control passes to the first executable statement following the START sentence.

In a multilevel indexed file, the current position of the record pointer determines the current index.

Example

```
START MYFILE KEY IS GREATER THAN KEY01.
```

If the keys in this index are AA, AA0, AA1, BB, BB0, and BB1, and KEY01 is equal to AA, the record pointer will be positioned at the beginning of the record with the key BB. (AA0 and AA1 are not of the same length.)

STOP

Terminates or temporarily suspends execution of the currently executing program.

Format

$$\underline{\text{STOP}} \left\{ \begin{matrix} \underline{\text{RUN}} \\ msg \end{matrix} \right\}$$

Where:

msg is a numeric literal, alphanumeric literal, or figurative constant that specifies a message.

Statement Execution

A STOP RUN statement closes all open files and terminates the current run unit.

A STOP *msg* statement displays *msg* on the program console and suspends execution of the current run unit until you type any character on the program console. Execution resumes with the next executable statement.

If you specify a message that is a numeric literal, it must be an unsigned integer.

Example

```
STOP "ENTER ANY CHARACTER TO RESTART".
```

STRING

Concatenates the contents or part of the contents of one or more data items into a single data item.

Format

```
STRING { src-1, ... DELIMITED BY { del-1 } }, ... INTO dest [WITH POINTER ptr] [ON OVERFLOW statement]
       {                         { SIZE  }
```

Where:

src is an alphanumeric literal or a numeric or alphanumeric data item that specifies a source field, and that has DISPLAY usage stated or implied by its data definition.

del is an alphanumeric literal or a numeric or alphanumeric data item that specifies a string delimiter, and that has DISPLAY usage stated or implied by its data definition.

dest is an alphanumeric data item that specifies the destination field in which concatenation occurs, and that has DISPLAY usage stated or implied by its data definition.

ptr is an unsigned integer data item that specifies the character position in *dest* at which the STRING operation begins.

statement is an imperative statement to which control passes if an overflow condition occurs.

If you specify a *src* or *del* data item as a numeric data item, the STRING statement interprets it as alphanumeric.

If you specify a *src* or *del* data item as a figurative constant, it represents a single-character, alphanumeric literal. Do not use the optional word ALL when specifying a figurative constant.

Do not specify editing symbols or the JUSTIFIED clause in the data definition of a *dest* data item.

Statement Execution

A STRING statement transfers selected characters from specified source fields *(src)* to a destination field *(dest)*.

COBOL processes the source fields in the order in which you write them, and scans the characters in each field from left to right. When COBOL has selected the appropriate source characters, it transfers them to the destination field according to the MOVE rules governing alphanumeric to alphanumeric moves (see the MOVE statement earlier in this chapter). Even if the set of source characters transferred is smaller than the size of the destination field, the STRING statement does not provide space-filling. That portion of the destination field not referenced by the transfer of source characters will contain the characters that were present before execution of the STRING statement. For example, if the destination field contains ABCD and you transfer the source characters 12, the resulting destination field is 12CD.

The characters that the STRING statement selects to transfer depends on what you specify in the DELIMITED BY phrase. If you specify *del* in the DELIMITED BY phrase, COBOL transfers the characters in each source field from the leftmost character up to, but not including, the first occurrence of *del* (the delimiter). If you specify a delimiter that does not exist, COBOL transfers the entire contents of the appropriate source field(s). If you specify a delimiter that indicates the leftmost character(s) of a source field, no transfer occurs. Execution terminates after COBOL transfers the selected characters from all source fields or when it reaches the last character in the destination field.

If you specify SIZE in the DELIMITED BY phrase, COBOL transfers the entire contents of each source field to the destination field. Execution terminates after COBOL transfers all source field characters or when it reaches the last character in the destination field.

COBOL uses an internal value, called the destination index, to determine where in the destination field data transfer should begin. You can override this value by initializing a pointer *(ptr)*. If you do not specify the WITH POINTER phrase, the destination index default value is 1. After data transfer to the destination field is complete, COBOL updates the value of the destination index to the number of characters transferred plus 1. Then, if you specified it, COBOL transfers this value to *ptr,* according to MOVE rules. The *ptr* data item must be large enough to contain a value equal to the number of characters in *dest* plus 1.

An overflow condition occurs if, at any point in the execution of a STRING statement, the value of *ptr* or the destination index is less than 1 or greater than the length of the destination field plus 1. If you specify the ON OVERFLOW option and an overflow condition occurs, data transfers to the destination field terminate and control passes to *statement.* If you omit this option, or if no overflow condition occurs, control passes to the first executable statement following the STRING sentence.

Example

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01  DES PIC X(26)
    VALUE "ABCDEFGHIJKLMNOPQRSTUVWXYZ".
01  SOURCES.
    02 S1 PIC X(4) VALUE "0123".
    02 S2 PIC X(4) VALUE "4567".
    02 S3 PIC X(6) VALUE ALL "89".
    02 S4 PIC X(4) VALUE "#!&".
    02 S5 PIC X(5) VALUE SPACES.
    02 S6 PIC X(7) VALUE "@%$".
    02 S7 PIC X(6) VALUE "ABCABC".
    02 S8 PIC X(5) VALUE "AAAAA".
    02 S9 PIC X(3) VALUE "333".
    02 S10 PIC X(3) VALUE "EXD".
01  ONEC PIC X VALUE "C".
01  THREE PIC X VALUE "3".
01  EXEX PIC XX VALUE "XX".
01  DEL PIC XX VALUE "12".
01  P PIC 99 VALUE 1.

PROCEDURE DIVISON.

STRING S1,S2 DELIMITED BY DEL,
       S3 DELIMITED BY SIZE,
       S4,S5,S6 DELIMITED BY SPACES,
       S7 DELIMITED BY ONEC,
       S8,S9 DELIMITED BY THREE,
       S10 DELIMITED BY EXEX,
   INTO DES, POINTER P,
   ON OVERFLOW DISPLAY "OVERFLOW".
```

Execution of the above STRING statement produces the following results:

```
DES = 04567898989#!@%$ABAAAAAEXD
P = 27
No overflow occurred.
```

SUBTRACT

Subtracts one operand, or the sum of two or more operands, from one or more other operands and stores the result.

Formats

Simple SUBTRACT -

```
SUBTRACT sub-1, ... FROM { minres-1 [ ROUNDED ] } ... [ ON SIZE ERROR statement ]
```

SUBTRACT GIVING -

```
SUBTRACT sub-1, ... FROM min GIVING { res-1 [ ROUNDED ] } ... [ ON SIZE ERROR statement ]
```

SUBTRACT CORRESPONDING -

```
SUBTRACT { CORRESPONDING } grsub FROM grsubres [ ROUNDED ] [ ON SIZE ERROR statement ]
         { CORR          }
```

Where:

sub is a numeric literal or a numeric data item that specifies a subtrahend.

minres is a numeric data item that specifies a minuend and receives the result of a simple SUBTRACT operation.

statement is an imperative statement to which control passes if a size error condition occurs.

min is a numeric literal or a numeric data item that specifies a minuend.

res is a numeric or numeric edited data item that receives the result of a SUBTRACT GIVING operation.

grsub is a group data item that specifies subtrahends.

grsubres is a group data item that specifies minuends and that receives the results of a SUBTRACT CORRESPONDING operation.

Statement Execution

A simple SUBTRACT statement sums the subtrahends, if you specify more than one, and maintains this sum as a constant through the remainder of the operations. Its execution then proceeds by subtracting this sum from the current value of *minres-1,* and storing the result in *minres-1* according to MOVE rules (see the MOVE statement earlier in this chapter). This process repeats itself for each operand following the word FROM.

A SUBTRACT GIVING statement sums the subtrahends, and then maintains this sum as a constant through the remainder of the operations. If you specify more than one subtrahend, this statement subtracts their sum from the minuend. SUBTRACT then stores the result, according to MOVE rules, in each *res* following the word GIVING.

A SUBTRACT CORRESPONDING statement subtracts data items in *grsub* from corresponding data items in *grsubres,* storing the results in the *grsubres* data items according to MOVE rules. The operation on each pair of data items is the same as if you specified a simple SUBTRACT for that pair. Correspondence occurs according to the rules for the CORRESPONDING phrase (see the section "The CORRESPONDING Phrase" in Chapter 6). CORR is an abbreviation for CORRESPONDING.

If you specify ROUNDED, and COBOL truncates the result of this operation to fit the given result item, it performs the rounding as follows: COBOL adds a 1 to the rightmost digit in the result item if the most significant digit of the truncated portion is equal to or greater than 5.

If you specify the SIZE ERROR phrase and if the absolute value of any result, after decimal point alignment, is too large to fit in the number of decimal places you allowed in the result item, then the SUBTRACT statement completes all operations and transfers control to *statement.* If you omit this phrase, or if no size error condition occurs, control passes to the first executable statement following the SUBTRACT sentence.

Examples

Example 1

```
SUBTRACT A FROM B.
```

If A = 4 and B = 7, the result is B = 3.

Example 2

```
SUBTRACT A, B, C FROM D GIVING E.
```

If A = 5, B = 4, C = 3, D = 15, and E = 10, the result is E = 3.

Example 3

```
SUBTRACT CORR GRPA FROM GRPB.
```

If

```
01 GRPA.
   02 ITM1 PIC 99 VALUE 1.
   02 ITM2 PIC 99 VALUE 1.
   02 ITM3 PIC 99 VALUE 1.

01 GRPB.
   02 ITM1 PIC 99 VALUE 3.
   02 ITM2 PIC 99 VALUE 4.
   02 RAT PIC 99 VALUE 5.
```

the result is ITM1 of GRPB = 2, ITM2 of GRPB = 3, and RAT of GRPB is unchanged.

TRUNCATE

Terminates input/output operations on records from the current logical block of a sequential file, so that the next input/output operation executed on that file will start at the beginning of the next logical block.

Format

TRUNCATE *file* BLOCK

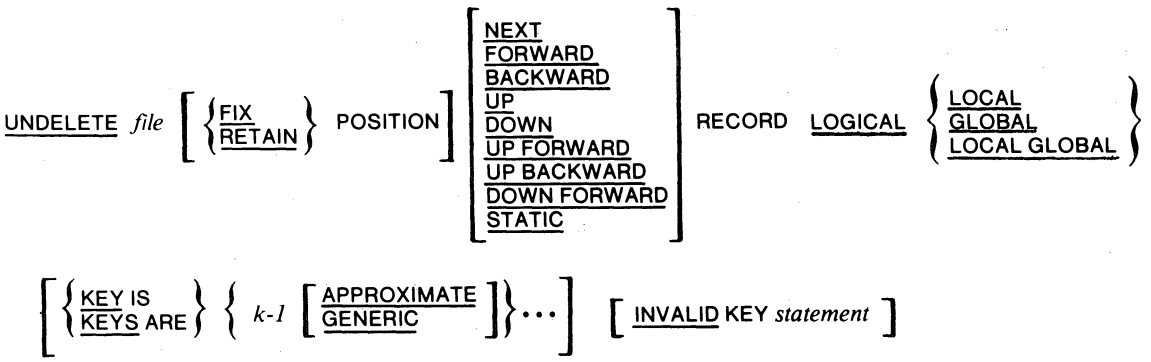
Where:

file is a filename that specifies an open, sequentially organized file.

UNDELETE

Restores a record that you logically deleted from an indexed file.

Format



Where:

file is a filename that specifies an open, indexed file.

k is an alphanumeric data item that specifies a record key associated with *file*.

statement is an imperative statement to which control passes if you specify invalid record selection indicators.

Statement Execution

COBOL determines which record the UNDELETE statement will restore according to what you specify in the POSITION phrase, the relative options phrase (NEXT, FORWARD, etc.), and/or the KEY series phrase.

By specifying FIX POSITION, you set the record pointer to the record specified in the relative options phrase and/or the KEY series phrase (discussed below). If you specify RETAIN POSITION, you do not change the current position of the record pointer (i.e., it points to the record for which you last set it). If you omit this option, the default is RETAIN POSITION.

When you specify a relative option, you reference a record in an indexed file, relative to the current setting of the file's record pointer. If you omit both this option and the KEY series option, the default is STATIC.

If you specify the KEY series phrase you must have declared each key *(k)* in the indexed file's SELECT clause.

For more information on these options, see the section "Indexed File Record Selection" in Chapter 6.

If you specify LOGICAL LOCAL, COBOL marks the key as logically restored, canceling the effect of any previously issued DELETE LOGICAL LOCAL.

If you specify LOGICAL GLOBAL, COBOL marks the data record as logically restored, canceling the effect of any previously issued DELETE LOGICAL GLOBAL.

If you specify LOGICAL LOCAL GLOBAL, COBOL marks the key and data record as logically restored, canceling the effect of any previously issued DELETE LOGICAL LOCAL GLOBAL.

If you specify record selection indicators which reference a record that does not exist, and if you specify the INVALID KEY option, then execution of the UNDELETE statement terminates and control passes to *statement.* If you omit this option, or if no invalid key condition occurs, control passes to the first executable statement following the UNDELETE sentence.

If you specify record selection indicators which reference a record that exists but is not marked for deletion, COBOL signals a runtime error.

Example

```
UNDELETE MYFILE RETAIN LOGICAL LOCAL.
```

This statement marks as logically restored the key pointed to by the current record pointer in MYFILE.

UNSTRING

Separates contiguous data located in a single source field and moves it to one or more destination fields.

Format

```
UNSTRING src [ DELIMITED BY [ ALL ] del-1 [ OR BY [ ALL ] del-2 ] ... ]

        INTO  dest-1 [ DELIMITER IN delstore-1 ] [ COUNT IN ctr-1 ]  [ WITH POINTER ptr ] [ TALLYING IN tal ]

        [ ON OVERFLOW statement ]
```

Where:

src is an alphanumeric data item that specifies a source field.

del is an alphanumeric data item or an alphanumeric literal that specifies a string delimiter.

dest is an alphanumeric, alphabetic, or numeric data item specifying a destination field that receives all or part of the source field, and that has DISPLAY usage stated or implied by its data definition.

delstore is an alphanumeric data item that receives a string delimiter.

ctr is an integer data item that receives the number of characters transferred in one iteration of the UNSTRING operation.

ptr is an integer data item that specifies the character position in *src* where the UNSTRING operation begins.

tal is an integer data item that receives the number of iterations the UNSTRING operation performs.

statement is an imperative statement to which control passes if an overflow condition occurs.

If you specify any data item in the UNSTRING statement as a figurative constant, it represents a single-character literal. Do not use the optional word ALL when specifying a figurative constant.

COBOL evaluates any subscripting for *src, ptr,* or *tal* only once, at the beginning of the UNSTRING operation. COBOL evaluates any subscripting for *del, dest, delstore,* or *ctr* immediately before transferring data to that item.

Statement Execution

An UNSTRING statement transfers selected characters from a source field *(src)* to one or more destination fields *(dest)*. The UNSTRING operation is an iterative process that performs one UNSTRING operation for each destination field. It scans the source field from left to right, and passes the selected characters to the destination fields in the order in which you specify them. If there are characters to pass, COBOL transfers them to the appropriate destination field according to the MOVE rules governing an alphanumeric to alphanumeric move (see the MOVE statement earlier in this chapter). If there are no characters to pass, COBOL space-fills or zero-fills the appropriate destination field, depending on whether the field is non-numeric or numeric, respectively.

UNSTRING (continued)

The characters that the UNSTRING statement selects to transfer to a particular destination field depend on whether or not you specify the DELIMITED BY phrase. If you do specify this phrase, UNSTRING transfers the characters in the source field from the leftmost character up to, but not including, the first occurrence of the delimiter. If you specify the DELIMITER IN phrase, and a delimiter match exists, COBOL interprets the delimiter as an alphanumeric data item and transfers it to *delstore.* If there was no delimiter match, COBOL space-fills *delstore.* If you specify the COUNT IN phrase, COBOL stores the number of characters transferred in *ctr,* according to MOVE rules. You may specify the DELIMITER IN and COUNT IN phrases only if you also specify the DELIMITED BY phrase. Execution terminates when COBOL has transferred characters to all the destination fields or when it has reached the last character in the source field.

COBOL uses an internal value, called the source scan base, to determine where in the source field scanning should start. You can override this value by initializing a pointer *(ptr).* If you do not specify the *WITH POINTER* phrase, the source scan base default value is 1. After scanning the source field and transferring selected characters to a destination field, COBOL updates the source scan base to the number of characters scanned plus one. This new value determines the point at which scanning will begin for the next iteration of the UNSTRING operation. If you specified the WITH POINTER phrase, COBOL transfers the new value of the source scan base to *ptr,* according to MOVE rules. The *ptr* data item must be large enough to contain a value equal to the number of characters in *dest* plus 1.

If you specify ALL in the DELIMITED BY phrase, the UNSTRING statement scans for not only the first delimiter match, but for contiguous delimiter matches. It interprets this contiguous "string" of delimiters as one delimiter. When this occurs, COBOL updates the source scan base to the character position to the right of the last delimiter found. It then moves the entire string of delimiters to *delstore* (if you specified it), and the number of contiguous delimiters to *ctr* (if you specified it).

If you do not specify a delimiter, the UNSTRING operation transfers the number of source characters needed to fill the appropriate destination field(s). Execution terminates when COBOL has filled all destination fields or when it has reached the last character in the source field.

If the UNSTRING statement terminates normally (if no overflow occurs), and if you specified the TALLYING IN phrase, COBOL adds the number of UNSTRING operations performed to the value in *tal.* If you specified the WITH POINTER phrase, COBOL stores the current value of the source scan base in *ptr,* according to MOVE rules.

An overflow condition occurs if, at any point in the execution of an UNSTRING statement, the value of *ptr* or the source scan base is less than 1 or greater than the length of the source field plus 1. If you specify the ON OVERFLOW option and the above condition occurs, data transfers to the destination fields terminate and control passes to *statement.* If you omit this option, or if no overflow condition occurs, control passes to the first executable statement following the UNSTRING sentence.

Example

Example 1

Given:

```
01 S PIC X(30),
     VALUE "ABC□□DEXYZYXFGHIJ□KLXMXYXYXYXOP".
01 DGROUPS.
     02 DG OCCURS 8.
          03 DES PIC X(4).
          03 DR PIC X(8).
          03 CNT PIC 9.
01 POINT PIC 99, VALUE 2.
01 TALLY PIC 99, VALUE 5.
```

Execute:

```
UNSTRING S DELIMITED BY SPACE
   OR ALL "XY" INTO
   DES(1), DELIMITER DR(1), COUNT CNT(1)
   DES(2), DELIMITER DR(2), COUNT CNT(2)
   DES(3), DELIMITER DR(3), COUNT CNT(3)
   DES(4), DELIMITER DR(4), COUNT CNT(4)
   DES(5), DELIMITER DR(5), COUNT CNT(5)
   DES(6), DELIMITER DR(6), COUNT CNT(6)
   DES(7), DELIMITER DR(7), COUNT CNT(7)
   DES(8), DELIMITER DR(8), COUNT CNT(8)
   POINTER POINT, TALLYING TALLY.
```

Results:

	DES	DR	CNT	Notes
1	BC□□	spaces	2	initial POINT = 2
2	spaces	spaces	0	
3	DE□□	XY□□□□□□	2	
4	ZXFG	spaces	7	HIJ truncated
5	KLXM	XYXYXY□□	4	
6	XOP□	spaces	3	
7	unchanged	unchanged	unchanged	
8	unchanged	unchanged	unchanged	

POINT = 31, TALLY = 11 (5+6).

Example 2

Given:

```
01  SR PIC X(24),
    VALUE "14/AB,22/RP,06/MX,42/AY".
01  PT PIC 99, VALUE 1.
01  GR.
    02 DSTAB OCCURS 50.
       03 DS PIC XXX.
01  I PIC 99.
```

Execute:

```
UNSTRING SR DELIMITED BY "/" OR ","
   INTO I, DS(I), I, DS(I), I,DS(I)
   POINTER PT;
   ON OVERFLOW DISPLAY "UNDERFLOW".
```

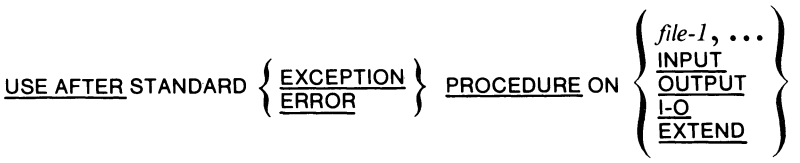
Results:

DS(6) = MX□
DS(14) = AB□
DS(22) = RP□
other DS(j) unchanged,
I = 06
PT = 19
"UNDERFLOW" is displayed

USE

Defines procedures for input/output error handling that are in addition to the standard procedures provided by the I/O control system.

Format



Where:

file is a filename that specifies the file you want to associate with the USE mechanism.

Statement Execution

A USE statement must immediately follow a section name in the declaratives section which is located at the beginning of your program's Procedure Division (see the section "The Declaratives Section" in Chapter 6). It may be followed by any number of procedural paragraphs defining the procedures to be used. A USE statement is not executable; it merely defines the conditions that will invoke the error-handling procedures.

The words EXCEPTION and ERROR have the same meaning; you may use them interchangeably.

Declarative procedures are invoked by the input/output system when the file system detects an I/O error (such as a device read error or a parity error), or when an end-of-file or invalid key condition occurs and the I/O statement involved does not contain an AT END or INVALID KEY phrase. Declarative procedure execution occurs when:

- You specify INPUT and an exception condition occurs while COBOL is processing a file that is open for input;
- You specify OUTPUT and an exception condition occurs while COBOL is processing a file that is open for output;
- You specify I-O and an exception condition occurs while COBOL is processing a file that is open for I/O;
- You specify EXTEND and an exception condition occurs while COBOL is processing a file that is open for extension;
- You specify one or more *files* and an exception condition occurs while COBOL is processing one of the specified files.

If you specify more than one declarative procedure, where one names a specific *file* and the other references one of the above opened modes, and if an exception condition occurs that satisfies both conditions, then COBOL executes the procedure naming the specific *file*.

After execution of a USE statement, control passes to the first executable statement following the statement whose processing invoked the declaratives procedure section.

Example

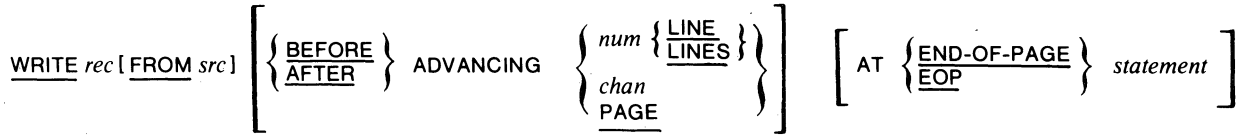
```
DECLARATIVES.
   SEC100 SECTION 2.
      USE AFTER ERROR PROCEDURE ON INPUT
            ADD 1 TO FLAG.
            MOVE COUNT TO ERR-SUM.
            .
            .
            .
END DECLARATIVES.
```

This section will be invoked by any statement that causes an error when processing a file which is open for input.

WRITE for a Sequential File

Outputs records to a sequential file and includes format control if the file is a print file.

Format



Where:

rec is a data name that specifies a logical record in a sequential file OPENed for output or extension.

src is any data item or literal that specifies the source you want to write.

num is a nonnegative integer literal or an unsigned integer data item that specifies the number of lines you want to advance.

chan is an alphanumeric literal that specifies the name of a line printer control channel that you specify in the Special-Names paragraph of the Environment Division.

statement is an imperative statement to which control passes if an end-of-page or page-overflow condition occurs.

Statement Execution

A WRITE statement for a sequential file writes a record which follows the record previously written in the file. If you specify the FROM option, WRITE moves the contents of *src* to *rec* according to MOVE rules, prior to performing the WRITE operation. (See the MOVE statement earlier in this chapter.) *Src* and *rec* must not reference the same storage area.

Both the ADVANCING and END-OF-PAGE phrases give you control over the vertical positioning of each line on a printed page. You may specify the ADVANCING option only for print files. If you omit this option, AFTER ADVANCING 1 LINE is the default. You may not specify *chan* if you specified the LINAGE phrase in this file's FD entry.

If you specify BEFORE ADVANCING, WRITE outputs the data record before processing the control information (described below). If you specify AFTER ADVANCING, WRITE processes the control information and then outputs the data record.

The control information offers three choices. You specify *num* to advance the current position to a position *num* lines ahead. You specify *chan* to advance the current position to the next position on the line printer control channel associated with the specified channel name. You specify PAGE to advance the current position to the next form or to the next logical page, if you specified the LINAGE clause in the file's FD entry.

For more information on the ADVANCING option, see the section "Print File Formatting" in Chapter 6.

WRITE for a Sequential File (continued)

You may specify the END-OF-PAGE phrase only for a file whose FD entry contains the LINAGE phrase. EOP is an abbreviation for END-OF-PAGE. There are two conditions which, when they occur, send control to this option. An end-of-page condition occurs when execution of a WRITE statement causes printing or spacing within the footing area of a page body. (You defined the size of the page's foot in the LINAGE clause.) A page overflow condition occurs when the current page body cannot accommodate the execution of a given WRITE statement. (You define the size of the page's body in the LINAGE clause.)

If you specify the END-OF-PAGE option and an end-of-page condition occurs, the WRITE statement completes its execution and control passes to *statement.* If you specify this option and a page overflow condition occurs, execution outputs the current record BEFORE or AFTER repositioning the device to the first line on the next logical page. Control then passes to *statement.*

If you omit this option, or if neither of the above conditions occur, control passes to the first executable statement following the WRITE sentence.

Example

```
WRITE CUST-INFO FROM NEW-REC AFTER ADVANCING 4 LINES AT EOP GO TO PARA-ERR.
```

This statement moves the contents of NEW-REC to CUST-INFO, advances the current position in the PRINTER file four lines, and then outputs the data contained in CUST-INFO. If an end-of-page condition occurs, control passes to the paragraph PARA-ERR.

WRITE for a Relative File

Outputs records to a relative file.

Format

```
WRITE rec [ IMMEDIATE ]  [ FROM src ]  [ INVALID KEY statement ]
```

Where:

rec is a data name that specifies a logical record in a relative file OPENed for output or I/O.

src is any data item or literal that specifies the source you want to write.

statement is an imperative statement to which control passes if the record selection indicators are invalid.

Statement Execution

A WRITE statement for a relative file that you are accessing sequentially writes a record which follows the record previously written in the file. If you specify the FROM option, WRITE moves the contents of *src* to *rec* according to MOVE rules, prior to performing the WRITE operation. (See the MOVE statement earlier in this chapter.) *Src* and *rec* must not reference the same storage area.

WRITEing to a relative file that you access sequentially updates the relative key to contain the relative record number of the record you just wrote. WRITEing to a relative file that you access randomly or dynamically writes the record at the relative record position indicated by the value of the relative key.

If you specify IMMEDIATE, the WRITE statement immediately writes the block containing the record out to the file, even though the next I/O operation may reference the same block. This option trades I/O efficiency for extra data security. The IMMEDIATE option is system dependent (see Appendix B).

An invalid key condition occurs if, in random or dynamic access mode, the record selection indicators specify a record that already exists, or if you attempt to write beyond the boundaries of a file. If you specify the INVALID KEY phrase and one of the above conditions occurs, execution of the WRITE statement terminates and control passes to *statement*. If you omit this option or if no invalid key condition occurs, control passes to the first executable statement following the WRITE sentence.

Example

```
WRITE DATAREC IMMEDIATE FROM INFO1.
```

This statement moves the contents of INFO1 to DATAREC and immediately writes DATAREC into the file, following the record previously written. It also updates the relative key to contain the relative record number of this newly written record.

WRITE for an Indexed File

Outputs records to an indexed file.

Format

```
WRITE [ INVERTED ] rec [ IMMEDIATE ]

[ { FIX    } POSITION ]  [ UP     ]
[ { RETAIN }          ]  [ DOWN   ]  [ SUPPRESS [ PARTIAL RECORD ] [ DATA RECORD ]]
                         [ STATIC ]

[ LOCK   ]  [ FROM src ] [ KEY IS key-1, ... ] [ INVALID KEY statement ]
[ UNLOCK ]
```

Where:

rec is a data name that specifies a logical record in an indexed file OPENed for output or extension.

src is any data item or literal that specifies the source you want to write.

key is an alphanumeric data item that specifies a record key associated with a *file*.

statement is an imperative statement to which control passes if you specify invalid record selection indicators.

Statement Execution

If you specify INVERTED, the WRITE statement does not write a data record. You can use this feature to write an inversion of an existing indexed file. To use this option, you must specify the FEEDBACK phrase in this file's FD entry.

If you do not specify INVERTED, WRITE writes a record into a location determined by what you specify (explicitly or implicitly) in the POSITION phrase, the relative options phrase (UP, DOWN, STATIC), and the KEY series phrase. If you specify the FROM option, COBOL moves the contents of *src* to *rec* according to MOVE rules, prior to performing the WRITE operation. (See the MOVE statement earlier in this chapter.) *Src* and rec must not reference the same storage area.

By specifying FIX POSITION, you set the record pointer to the record specified in the KEY series phrase or the relative options phrase (discussed below). If you specify RETAIN POSITION, you do not change the current position of the record pointer (i.e., it points to the record for which you last set it). If you omit this option, the default is RETAIN POSITION.

When you specify a relative option, you reference a record in an indexed file, relative to the current setting of the file's record pointer. If you omit both this option and the KEY series option, the default is STATIC.

If you specify the KEY series phrase, you must have declared each *key* in this file's SELECT clause. However, if the indexed file has alternate record keys, you must key the WRITE operation by the prime record key. You need not specify this key in the KEY series option.

For more information on these options, see the section "Indexed File Record Selection" in Chapter 6.

If you specify the LOCK option, no one can access the referenced record until you issue an I/O statement with UNLOCK for the same record, or until you CLOSE the file, which automatically UNLOCKs the record. You must issue the corresponding LOCK and UNLOCK statements in the same program. LOCKing of records varies from operating system to operating system (see Appendix B).

If you specify SUPPRESS PARTIAL RECORD, WRITE will not output the partial record associated with the referenced index entry to the file's partial record area (which you defined in the file's FD entry in the Data Division). If you specify SUPPRESS DATA RECORD, WRITE will not output the data record associated with the referenced index entry to the file's record area. You may specify both of these options in any order. Specifying both options lets you create an index entry with which no data need be associated.

If you specify IMMEDIATE, the WRITE statement immediately writes the block containing the record out to the file, even though the next I/O operation may reference the same block. This option trades I/O efficiency for extra data security.

An invalid key condition occurs when the record selection indicators specify a record that already exists, or when you attempt to write beyond the boundaries of the file. If you specify the INVALID KEY clause and any of the above conditions occurs, execution of the WRITE statement terminates and control passes to *statement*. If you omit this option or if no invalid key condition occurs, control passes to the first executable statement following the WRITE sentence.

Example

```
WRITE HD-PARTS FIX POSITION KEY IS KEY03.
```

This statement writes the record information in HD-PARTS and associates it with the key value specified by KEY03. It then sets the current position of the record pointer at this key entry.

End of Chapter

Chapter 8
The COPY Facility

The COBOL COPY statement directs the compiler to insert source code from another file into your compiled program. The COPY statement makes it possible for you to refer to frequently used program text by a name instead of writing its lines of code into your program.

Unlike a subprogram, the COPY file becomes part of the main program file during compilation. Moreover, you can collect separate text files into a directory called a *library*. During compilation, the COBOL compiler will use only those library texts that your program refers to.

A library may consist of library texts or a combination of library texts and files. For information on the operating systems' file structures, see the reference manual appropriate to your operating system.

The COBOL compiler also allows you to make textual substitutions as it copies. The COPY statement can specify up to ten substitutions of up to ten elements each. Using these replacements, you can tailor a single file to the needs of different data sets and programming strategies. You may also specify nested COPY statements, up to a depth of 10.

When you compile your COBOL program, you must include in the compilation command line the names of any files or directories your program calls in to copy. See either Chapter 10 or Chapter 11 for a discussion of compilation instructions for your operating system.

Structure

A COPY statement may occur at any point in a source program where a language element is permitted, except within another COPY statement. You must delimit it from the surrounding elements with separators, and terminate it with a period (which is considered part of the statement). A COPY statement has the following format:

Where:

text is a symbolic name that specifies the name of the file you want to copy. You may use any valid file specifier.

lib is a symbolic name specifying the name of the library directory (if one exists) that contains *text*.

rep1 is a single COBOL word, a numeric or alphanumeric literal, a character string, or a pseudo-text that specifies source code in the file you want to COPY.

rep2 is a single COBOL word, a numeric or alphanumeric literal, a character string, or a pseudo-text that specifies text you want to replace each occurrence of *rep1*.

Compilation of your program logically replaces any COPY statement with the source text of the files you specify.

For texts located in libraries, you can also give the filename as *lib:text* and omit the option OF/IN LIB.

If all the characters of *text* or *lib* are letters, digits, periods, or embedded hyphens, you need not enclose the name in quotation marks; otherwise, you must.

Some examples of simple COPY statements are as follows:

```
COPY "LIBR:TEXT1.CO".

COPY "TEXT1.CO" OF "LIBR".

COPY LIBR:TEXT1.CO.
```

The three statements above illustrate equivalent uses of COPY.

```
COPY DATA_TEXT OF LIB.
```

This statement is incorrect because DATA_TEXT must be enclosed in quotation marks.

Replacement Strings

In the COPY statement's optional REPLACING clause, you can specify up to ten replacements. The word REPLACING occurs once, followed by the pairs of replacement texts. For each pair, the compiler will replace every occurrence of *rep1* in your program's copy of the library file with *rep2*.

Each string can consist of ten elements: *rep1* may not be a null, but *rep2* may. Otherwise, *rep1* or *rep2* can consist of any of the following elements:

- A single COBOL reserved word, user-defined word, or undefined word;
- A single COBOL special character word from this list: +, -, /, **, =, < , >, (,);
- A single literal, either numeric or alphanumeric;
- A COBOL identifier (a text string) of the form:

  ```
  A IN/OF B IN/OF C ... (I, J, ...)
  ```

- A pseudo-text, that is, a string of COBOL elements delimited by double equal signs.

Some examples of COPY with REPLACING are:

```
COPY "LTEST" REPLACING
   ABC BY ==SET A TO BC==
   "LMN" BY MOVE
   X OF Y OF Z (3,A) BY "PQR"
   ==TRY TO FIT 26== BY 26
   13 BY LOC.
```

This example inserts a file, LTEST, into the source code and makes five kinds of replacement. Under these instructions, when the compiler encounters these lines in "LTEST":

```
"LMN" TRY TO
FIT 26 TO 13
IF X OF Y OF M (3,A) =
   X OF Y OF Z (3,A), ABC.
```

it will insert the following into the compiled source file:

```
MOVE 26 TO LOC
   IF X OF Y OF M (3,A) = "PQR", SET A TO BC.
```

Notice that the order of the replacement pairs need not match the order in which they occur in the library text. Moreover, the replacement will be made for each occurrence of *rep1*.

End of Chapter

Chapter 9
The COBOL Interactive Debugger

The ECLIPSE COBOL system provides a source-level debugging aid that allows you to debug your program in an interactive mode at the console. This debugger is able to respond to COBOL-like commands as you issue them from the console.

The debug program allows you to set breakpoints in your program which will halt execution, then transfer control to the debugger. While in debug mode, you can examine and modify data items in the object program, and then return control to the execution of your program.

After a debugging session, you will have to edit your program's source file, to include the necessary changes, and then recompile it.

Operating Instructions

To use the debugger, you must first compile your source file. In the compilation command line, you must append the global switch /D, which tells the compiler to include the debug program with your code. Next, load the debugger and your program by appending the global switch /D to the CBIND command.

When the system loads the debugger, it will increase the memory storage your program needs by about 500 words. This block of storage includes the code for debugger commands and a symbol table (to keep track of your program's code).

To use the debugger while executing your program, issue the CLI command appropriate to your operating system:

```
DEB MAINPROGRAM)   (RDOS)

DEB MAINPROGRAM)   (AOS)
USER)
```

When the debugger is loaded and ready, it will signal you with the prompt character *. You may then issue your first command. The debugger will issue the same prompt character each time it is ready to accept a new command. You must terminate each debugger command with an operating system line terminator. You may, however, specify more than one command on a line provided that you separate each by a period or a space (the only exception is the COPY command which must appear on a line by itself).

Comment Lines

Comments are useful for documenting debugger COPY files and for making notes in the audit file. To insert a *comment line,* begin the line with an *asterisk* and end it with an operating system line terminator.

Debug Lines

You indicate a debug line in your source program by specifying the character D as the first character of the source line in text format, or by inserting it in column 7 in card format. In text format, you must also immediately follow the D with a space or a tab, or it will be treated as any other source character. When you compile your program, COBOL will compile these debug lines only if you specify the global switch /D in the command line. Otherwise, COBOL treats them as comment lines.

Debugger Features

The COBOL debugger includes a set of twelve commands you can use in a debugging session. These commands cover four basic debugging features: using breakpoints, checking program status, controlling program execution, and using other programs and files.

Using Breakpoints

The SET command allows you to specify breakpoints - lines at which your program's execution will halt and control will pass to the debugger. The CLEAR command removes either all or specific breakpoints that you previously set.

Checking Program Status

When your program's execution halts, you can direct the debugger to display data items with the DISPLAY command. You can also reset a data item. The MOVE command sets a data item to a specified value and the COMPUTE command sets a data item to the value of an arithmetic expression. By changing data values, you can define new tests for your program, check for internal consistency in data handling, and change faulty values before continuing execution.

The WALKBACK command displays information about the program's execution status. It lists all active CALL and PERFORM statements at the console.

Controlling Program Execution

The CON command resumes execution of the program, which will continue until it encounters a break, error, or normal termination.

In a multiprogram run unit, you can use the ENV command to specify a single program to which the debugger commands will apply.

When you type STOP at the console, it terminates execution immediately. You will want to use this command to abort execution or to end a debugging session before the program has fully executed.

Using Other Programs and Files

The debugger offers a means by which you can use files outside of the debug program and object file. When the debugger has control, you can use the CLI command to temporarily enter the operating system's Command Line Interpreter and issue system commands. With the AUDIT command, you can save a record of the debugger/program interaction as it appeared at the console. You can enter comments in this file by beginning the string with an asterisk. Finally, during debugging you can execute a file containing a previously listed set of debug commands by using the command COPY. This command can save time and errors at the console, and allows you to define standard debugging procedures.

Debugger Commands

The remainder of this chapter presents the COBOL debugging commands, in alphabetical order, along with a detailed description of each command's implementation.

Please note that the COBOL interactive debugger does not process data names that begin with a numeric character.

AUDIT

Saves user, debugger, and object program interaction as it occurs at the console.

Format

AUDIT [*"afile"*]

Where:

afile is an alphanumeric literal that specifies the name of an operating system file (called the audit file).

Description

If you specify *afile,* the system opens the indicated file and outputs a copy of all succeeding console interaction between the user, the debugger, and the object program. If you omit *afile,* the system closes the current audit file, if one exists, and audit output terminates.

You may open only one audit file at a time.

CLEAR

Removes breakpoints that you previously set with a SET command.

Format

CLEAR [*break-1*, ...]

Where:

break is an alphanumeric literal that specifies the name of a breakpoint defined in a previous SET command.

Description

If you specify a *break* (or *breaks)* the system removes that particular breakpoint. If you omit *break* altogether, the system removes all previously set breakpoints.

CLI

Allows you to enter the operating system's Command Line Interpreter (CLI).

Format

CLI

Description

Before you enter the CLI, the system saves the current state of your program for reentry. You reenter the debugger by specifying the appropriate operating system reentry command. For more information on the CLI, see the Command Line Interpreter reference manual that is appropriate for your operating system.

COMPUTE

Sets one or more data items to the value of an arithmetic expression.

Format

COMPUTE { *res-1* [ROUNDED] } [, . . .] = *expr*

Where:

res is a numeric data item defined in the object program.

expr is any valid arithmetic expression.

Description

The system evaluates the arithmetic expression and stores the result in the result item(s) you specify, according to the rules for the COBOL COMPUTE statement (see Chapter 7).

You may qualify and/or subscript any *res*, as appropriate.

CON

Either initiates execution of the object program, or resumes program execution after encountering a breakpoint.

Format

CON [[*break*] *rep*]

Where:

break is an alphanumeric literal that specifies the name of a breakpoint defined in a previous SET command.

rep is a positive integer literal that specifies the number of times you want *break* disabled.

Description

The first time you specify the CON command in a given run of your program, the system will begin execution at the first statement in the main program. Thereafter, object program execution will resume after each breakpoint. (Remember: you only interrupt the flow of control in your program when using the debugger; you do not alter it.)

If you only specify *break,* and *break* is set at line *n* in the object program, then execution will resume and control will return to the debugger the next time line *n* is encountered.

If you specify *break* and *rep,* and *break* is set at line *n* in the object program, then execution will resume, but control will not return to the debugger until it has encountered line *n* the number of times specified by *rep.*

If you omit both *break* and *rep,* the default breakpoint is the most recent one and the default repeat count is 1.

This command only affects the specified trap or the implied one. Other traps are handled in the usual manner.

Examples

```
CON
```

Repeat count for current breakpoint is 1.

```
CON 5
```

Repeat count for current breakpoint is 5.

```
CON BUG 10
```

Repeat count for breakpoint BUG is 10.

COPY

Executes a series of debugger commands stored in a file that you set up.

Format

COPY *"dfile"*

Where:

dfile is an alphanumeric literal that specifies an operating system file containing a series of COBOL debugger commands.

Description

The COPY command takes the debugger commands from *dfile,* and executes them in the order in which they appear. When the file is exhausted, control returns to the console, and the debugger awaits a new command.

You may specify COPY statements within *dfile* to a depth of 5 nested COPY commands.

DISPLAY

Outputs the contents of object program data items to the current console.

Format

DISPLAY *item-1*[, ...]

Where:

item is any data item defined in the object program.

Description

Output occurs according to the rules for the COBOL DISPLAY statement (see Chapter 7) with the following exceptions:

- the system displays each data item on a new line;
- it displays items of more than 80 characters on multiple lines, 80 characters per line;
- and it directs all output to the current console.

ENV

Names the program, in a multiprogram environment, to which future debugger commands will apply.

Format

ENV [*prog*]

Where:

prog is a symbolic name that specifies the program ID of a program in the current execution.

Description

After you issue the ENV command, all further references to data items and line numbers in debugger commands will apply to the specified program.

If you omit *prog,* the default is the main program. This is what the debugger assumes when you begin execution of your program. Afterwards, if you do not issue an ENV command, the environment is determined by the breakpoint you set; i.e., the current environment is that of the last breakpoint encountered.

MOVE

Sets data items in the object program to specified values.

Format

MOVE *src* TO *dest-1* [,...]

Where:

src is a literal or data item defined in the object program.

dest is a data item defined in the object program.

Description

This command stores the value of *src* in the destination items according to the rules for the COBOL MOVE statement (see Chapter 7).

The debugger has a 500-byte area that it uses for executing MOVE operations. If the area needed for a MOVE exceeds this limit, you will receive an error message at the console.

SET

Defines and enables a breakpoint.

Format

SET [*break*, [*prog* +] *lin*]

Where:

break is an alphanumeric literal that specifies the breakpoint you want defined and enabled.

prog is a symbolic name that specifies the object program.

lin is an integer literal that specifies the number of the line at which you want to define the breakpoint.

Description

The line number, *lin*, does not refer to the card format sequence number. It refers to the absolute number of the line in the source file, as numbered by the COBOL compiler in the source listing.

If you specify *prog*, the system sets a breakpoint at the line number indicated by *lin* in *prog*. If you omit *prog*, the system sets a breakpoint at the line number indicated by *lin* in the program last referenced in an ENV command. If you have not issued an ENV command, *prog* is the current object program. (Note that this command does not change the current environment.)

When the system encounters the specified line during execution of the object program, control immediately passes to the debugger, (unless altered by a previously issued CON command). The object program statement at that line is not executed until you issue a CON command for that breakpoint.

You may have up to eight enabled breakpoints at a given time.

If you issue a call to CON without specifying any arguments, the system displays a list of the current breakpoints on the console.

Examples

```
SET
```

Lists all currently defined breakpoints.

```
SET BREAK5 50
```

Defines a breakpoint named BREAK5 at line 50 in the current environment.

```
SET XX PROGA + 150
```

Defines breakpoint XX at line 150 in the program with the program ID PROGA.

STOP

Ends execution of the current program.

Format

STOP RUN

Description

This command has the same effect as the COBOL STOP RUN statement (see Chapter 7).

WALKBACK

Describes the current program control situation by listing active subroutine CALLs and PERFORMs.

Format

WALKBACK [*num*]

Where:

num is an integer literal that specifies the number of CALLs and PERFORMs you want listed.

Description

If you specify *num,* the system lists the *num* most recent CALL statements and/or PERFORM statements executed in the current program environment. Otherwise, it lists them all.

For a PERFORM, the listing has the form:

```
PERFORM AT LINE # IN PROGRAM id;
REPEAT COUNT IS n
```

where *n* is the remaining number of times that the PERFORM will execute (in a PERFORM of the form PERFORM *n* TIMES).

For a CALL, the listing has the form:

```
PROGRAM id CALLED FROM LINE # IN
  PROGRAM id
```

End of Chapter

Chapter 10
How to Use COBOL Under RDOS

The ECLIPSE COBOL system includes a compiler, a runtime library, and a debugger program. The compiler generates relocatable, binary object files; the runtime library is a collection of modules that are necessary to execute system calls your program invokes; and the debugger program allows you to monitor your COBOL program, as well as make and test alterations to it interactively, while it executes. The debugger is discussed in detail in Chapter 9.

The operating environment required for COBOL is a commercial ECLIPSE RDOS system with an INFOS file management system.

Compiling, Loading, and Executing

There are four stages needed to bring your COBOL program to life. Using one of Data General's text editors, you create the file. The command, COBOL, compiles the source file you created, produces an object file, and reports any detected errors. The command, CBIND, names input object modules, directs the binder to build an executable program file with an optional overlay file, and scans the COBOL runtime library for modules the system will need to execute certain types of system calls associated with your program. Once you have compiled and bound your program, you are ready to execute it.

Using the Compiler

The COBOL compiler's main function is to produce an object file from your COBOL source program. It will also report any errors encountered during compilation. You may call on the compiler to produce a listing of your source file with true line numbers (as opposed to card sequence numbers) indicated in the left margin.

In addition, the compiler can produce listings that describe different aspects of the program and the compilation. These optional listings include:

- A listing of the generated code for the object file;
- An address map, showing the relative locations of Procedure Division lines;
- A map of data and procedures in the object file;
- A list of statistics describing the compilation (such as the number of lines compiled, the speed of the compilation);
- A source program cross-reference table.

You may request the listing file as well as any of the above compilation listings in the compilation command for your program.

Calling the Compiler

To compile a COBOL source file, issue the COBOL command to the RDOS Command Line Interpreter (CLI). The command has the format:

COBOL [*/sw/sw . . .*] *sourcefile* [*listingfile/L*] [*objectfile/R*] [*library/W*])

Where:

sw is a global switch to the compilation command that specifies an option you want to use in the compilation. See the list of global switches in the following section.

sourcefile is a filename that specifies the source program file you want to compile.

listingfile specifies the file or device to which you want the listing file output. It may be the console ($TTO), the line printer ($LPT), or a disk or tape file.

objectfile is a filename that specifies the name you want assigned to the object file the compiler produces. If you do not supply a name, the compiler uses *sourcefile* with the extension .RB.

library is the name of any directory that contains COPY text files.

You must supply the source filename. If you designate any other optional filenames, you must append the appropriate local switches (/L, /R, or /W). The filenames may appear in any order. Append any global switches you use to the command word and any local switches to the appropriate filename. These switches may also appear in any order.

Compiler Switches

Global switches give the compiler information about the nature of your source file, and instruct it about the kind of output you want it to produce. The following list contains all the options available for a COBOL compilation.

/A Produce an address map of the relative locations of the Procedure Division lines.

/C Source is in card format. If you omit this switch, the compiler assumes the source is in text format. (See Chapter 2 for details on format.)

/D Compile debug lines and load the interactive debugger. Use this switch if your file includes debug lines and if you want COBOL to load the code for the debugger along with your source file code. (See Chapter 2 for details on debug lines and Chapter 9 for details on the debugger.)

/E Compile language extensions. Use this switch if you want octal values produced for alphanumeric literals (this conflicts with ANSI Standard COBOL features.)

/G List the generated machine code. (This switch overrides the /A switch.)

/L List the source code at *listingfile*. If you did not name a listing file, $LPT is assumed.

/M List a map of data and procedure storage in the object file.

/P Do not generate an object file.

/Q Do not compile; simply scan the source file code and produce a cross-reference table.

/S List compilation statistics (the number of lines, the speed of compilation, etc.).

/V Compile virtual overlays. If you specify this switch, the compiler produces a .RB file from which you can load overlay segments as virtual overlays rather than disk overlays.

/W Suppress warning messages.

/X Include a cross-reference table in the listing.

The compiler normally produces its listing at either the line printer or the console. However, if you call for a listing of the source code, error messages, warning messages, generated code, map, compilation statistics, or cross-reference table, and you omit *listingfile* in the compilation command line, it will always be output to $LPT.

The local switches you may specify in the compilation command line are /L, /R, and /W as shown in the preceding list.

Example

The following command calls the COBOL compiler to compile a file named FILE1:

```
COBOL/L/X/W FILE1 FILE1.LS/L COLIB/W)
```

This command will compile the source file FILE1, and produce an object file named FILE1.RB (default name). The listing file FILE1.LS will contain the source listing (/L), a cross-reference table (/X), and error messages (automatically). The compiler will suppress warning messages (/W).

The library COLIB contains any copy text files specified in the source file.

Error Messages

The COBOL compiler always outputs a listing of all errors it detects in the source program during compilation. This error report appears either at the current console, or in a listing file if you specify one in the compilation command line. If more than fifty errors occur in any one phase of compilation, the compilation aborts.

For each error, the report identifies the true line number and the element in the line where the error occurred, followed by an English language diagnostic message describing the error. The compiler defines words, literals, special characters, and periods as elements of a line; commas, semicolons, and blanks do not count as elements.

In the following example, the input file contains an error in the declaration of a data item:

```
01 DAT-ITEM, PICTURE $$$999.99C, USAGE DISPLAY.
```

The compiler reports the error in this form:

LINE	ELEMENT	ERROR
0345	04	"R" MUST FOLLOW "C" IN PICTURE STRING

If you requested a listing of the source file, the output will have the number 0345 next to the line in question. The fourth element, $$$999.99C, contains the error.

Warning Messages

Normally the compiler generates warning messages in addition to error messages (unless you suppress them with the /W global switch). Warning messages have the same format as error messages.

The compiler signals a warning where it finds details in the program that are not incorrect syntactically, but are in some way inconsistent. For example, if you specify BLOCK CONTAINS *n* in a file's FD entry, and *n* is less than the record length specified for the file, you will receive a warning message. You may safely ignore warning messages if you are sure your program will not create an error condition; you may, in fact, have reasons for apparently inconsistent coding.

Compiler errors will prevent your program from executing; you must correct and recompile your source file, then load and execute it. Warnings will not prevent execution, but may produce runtime errors or faulty computation.

For a complete list of the COBOL compiler's error and warning messages, see Appendix D.

Loading Program Files

After your program has been successfully compiled, you must issue the CBIND command to build an executable program file from your object file(s). CBIND loads your main file and any subprograms or macro subprograms you specify, as a single program image.

Issue the CBIND command in the following format:

CBIND *[/sw/sw ...] main* /M *[subprog1 subprog2 ...] [filename/sw]*

Where:

sw is a global switch if appended to CBIND, and a local switch if appended to *filename*. It is a literal that gives the loader information about the input files and specifies the format of the save file you want to create.

main is a filename that specifies the main program. You must supply this argument and use the local switch /M. If you do not include a filename extension, CBIND assumes that it is .RB.

subprog is a filename that specifies a subprogram you want to include in the loaded program. If any of these subprograms has overlays, you must indicate it by specifying the local switch /P, which instructs the loader to generate an overlay file.

filename is a name you want to give to the loaded program file. You must append the local switch /S, which tells the binder to assign the name *filename.SV* to the save file, and the name *filename.OL* to the overlay file (if one exists).

If you do not supply a name for the loaded program file, the loader uses the name of the main file, and gives it either a .SV (save file) or .OL (overlay file) extension.

CBIND Switches

The following list contains all the global switches for the CBIND command word:

/A List the symbol table in alphabetical and numerical order.

/C Program contains CAM modules; link in CAM routines.

/D Link in the COBOL debugger program.

/E List error messages on the console, even though another listing file is specified.

/F Channel designations in Special-Names paragraph will correspond to paper tape VFU (Vertical Forms Unit) channels. By default, the system assumes that the line printer is a data channel VFU line printer.

/H Generate numerics in listing in hexadecimal format.

/L Produce a listing file.

/M Suppress load map and all console output.

/N Suppress SYS.LB search.

/O Configure the save file for a stand-alone system (the system will assume you have a DCU-assisted configuration in a mapped environment).

/Q Terminate loading process after command file is built.

/S Save the symbol table.

The following list contains all the local switches available with CBIND. You must append these to the appropriate *filename* or integer value.

n/C	Allocate n channels for I/O. By default, the loader allocates 10 channels (25 if you are using the debugger).
filename/E	Output error messages to file filename.
filename/I	Specifies the table built by the CGEN process as the local file filename. (If you select CAM, and do not specify a table, the CGEN default name TABLE.RB is used.)
n/K	Allow n tasks for CAM.
filename/L	Output listing to file filename.
filename/M	Identifies filename as the main program. You must specify this switch.
addr/N	Start loading next module at address addr.
subprog/P filename/V	The subprog may have overlays. Identifies filename as a virtual overlay for this program or subprogram. All filename's overlays become virtual overlays.

Link loading produces the following output: a listing of the file (unless suppressed by the global switch /M), a load map, a symbol table, and a list of any loading error messages. These will appear at the current output console unless you specify a listing file. You may also have the error messages printed at the console while the rest of the listing goes to a different file (/E switch).

For a complete list of load error messages, see the *Extended Relocatable Loader User's Manual,* (93-000080).

Splitting the Loading Process

The system loads your program in two stages. CBIND itself is a utility that creates a load command and stores it in the command file CLI.CM (FCLI.CM for foreground execution). CBIND then passes control to the RDOS Extended Relocatable Loader (RLDR), which actually loads the object code.

The CBIND global switch /Q allows you to split this two-step process. It instructs CBIND to stop after creating the command line, and before it passes control to RLDR. Stopping the load operation allows you to do special load processing; you can edit the command file.

After editing the command file, execute the load with the command @CLI.CM@ (or @FCLI.CM@). For a full description of RLDR and load processing, see the *Extended Relocatable Loader User's Manual,* (93-000080).

Example:

```
CBIND MYFILE/M UPDATSUB/P HACKSUB MYFILE.MP/L)
```

The binder will link-load MYFILE, the main program, and two subprograms, UPDATSUB (which is segmented) and HACKSUB. The binder output listing will go to MYFILE.MP. If the load is successful, MYFILE will be ready for execution.

Executing Your COBOL Program

To execute a loaded COBOL object program, type the name of the program save file at the console. Program execution will begin at the main program's entry point. You may, in addition, append execution switches to the program name. You declare these switches in the Environment Division of your program. See the section "The Special-Names Paragraph" in Chapter 4 for information on execution switches.

The following examples show two calls that will execute the file NAMEFIL. The first calls for simple execution; the second uses execution switches declared in the program:

```
NAMEFIL)
```

```
NAMEFIL/C/A)
```

Executing in the Debugger

The COBOL debugger is a program that allows you to check your program's effectiveness. Using the debug program, you can execute your program piecemeal, halt its execution to examine its performance, make changes to the object code if necessary, and resume execution.

To use the debugger, you must load it along with your program and subprograms by using the /D switch in the CBIND command line.

To begin program execution in the debugger, type DEB before the save filename:

```
DEB MYPROG)
```

```
DEB MYPROG/C/A)
```

When the debugger is ready to accept a command, it will signal you with the prompt character *. See Chapter 9 for a complete description of the COBOL interactive debugger.

Runtime Errors

A runtime error condition occurs when, during the execution of your program, the system cannot properly complete an instruction. Error conditions commonly involve inconsistencies in runtime calls, discrepancies between an instruction and the current state of memory or files, and conflicts which evolve from hardware and software limitations.

There are two classes of errors: nonfatal and fatal. If a fatal error occurs, your program's execution terminates. Program execution continues after a nonfatal error, but the results may be faulty. In a running COBOL program, five types of error conditions may occur.

I/O Exception Conditions: These error conditions are defined in Chapter 6. If your program declares an error-handling procedure, control goes to the handler and then execution resumes at the first executable statement following the statement that invoked the error handler.

Size Error, String/Unstring Overflow, Call Overflow, Search at End: The system does not output a message for these nonfatal errors. If your program declares an error-handling procedure, control goes to the handler and then execution resumes as it does for I/O exception conditions.

Nonfatal Program Errors: These error conditions are listed below. There is no error-handling option; execution simply continues. However, the system outputs COBOL trace information to the console (described later in this chapter).

Fatal Program Errors: These error conditions are listed below. There is no error-handling option; the program's execution terminates and control passes to the CLI. The system outputs a COBOL error message and trace information (described later in this chapter) for the current program and the calling program.

Fatal System Errors: These error conditions are defined in the *RDOS Reference Manual.* There is no error-handling option; the program's execution terminates and control passes to the CLI. The system outputs an RDOS error message and COBOL trace information (described later in this chapter) for the current program only.

Error Messages

The system displays error messages for COBOL program errors and system errors at the output console. For a list of all RDOS system error messages, see the *RDOS Reference Manual.* The following lists contain all COBOL runtime error messages for fatal and nonfatal program errors.

Nonfatal COBOL Program Errors

INTERMEDIATE OVERFLOW

An arithmetic operation gives an intermediate result with more than 31 digits. Processing continues with the high-order overflow truncated.

ILLEGAL ARGUMENT FOR EXP

The base of an exponentiation operation exceeds 16 x 10 (**63).

ILLEGAL EXPONENTIATION

Attempt to raise zero to the zero power, or raise a negative number to a nonzero fractional power.

INTEGER OVERFLOW ON CONVERSION

Loss of high-order significance when converting an external floating point number to internal floating point.

INVALID SIGN-NUMERIC
INVALID CHARACTER-NUMERIC

An invalid sign/invalid character has been detected in a numeric item.

Fatal COBOL Program Errors

STACK OVERFLOW

The system stack has overflowed its capacity.

PROGRAM ENTRY WITH INVALID ARGS

The arguments of a called program do not match the parameters passed by the calling program.

SUBSCRIPT OUT OF BOUNDS

A subscript item is outside the limits declared for the reference that contains it.

OCCURS-DEPENDING OUT OF BOUNDS

The depending data item contains a number exceeding the bounds of the occurrence item.

Trace Information

A COBOL error report consists of the error message, program name, and COBOL trace information. They appear in the following form:

(error message) : (name of program save file)
CALLED FROM *(program id)* SEGMENT *(number)*
RELATIVE LOCATION *(loc)*

Where:

program id is the filename you specified in the Identification Division to identify this program.

number is the number of the segment containing the instruction.

loc is the relative address of the instruction that caused the error.

Example

INTEGER OVERFLOW ON CONVERSION: A.SV
CALLED FROM B SEGMENT 35
RELATIVE LOCATION 563

The message gives the name of the save file, program *A.SV*, rather than the name of the subprogram, *B*, that actually called the subroutine. The address given in the trace information is the relative address in *A.SV* of the instruction that caused the error.

If you compile the program with the /A global switch (omitting the /L global switch), you will receive a table containing the line numbers of your Procedure Division statements beside a table containing each statement's relative address. You then reference relative location 563 in the table to determine the number of the line in your program which contains the instruction that caused the error. (If you specify both the /A and /L global switches, the system will output the relative address table on the line printer following your source file output.)

End of Chapter

Chapter 11
How to Use COBOL Under AOS

The ECLIPSE COBOL system includes a compiler, a runtime library, and a debug program. The compiler generates relocatable binary object files; the runtime library is a collection of modules that are necessary to execute system calls your program invokes; and the debugger program allows you to monitor your COBOL program, as well as make and test alterations to it interactively, while it executes. The debugger is discussed in detail in Chapter 9.

The operating environment required for COBOL is a commercial ECLIPSE AOS system with an INFOS file management system.

Compiling, Loading, and Executing

There are four stages needed to bring your COBOL program to life. Using one of Data General's text editors, you create the file. The command, COBOL, compiles the source file you created, produces an object file, and reports any detected errors. The command, CBIND, names input object modules, directs the binder to build an executable program file with an optional overlay fi e, and scans the COBOL runtime library for modules the system will need to execute certain types of system calls associated with your program. Once you have compiled and bound your program, you are ready to execute it.

Using the Compiler

The COBOL compiler's main function is to produce an object file from your COBOL source program. It will also report any errors encountered during compilation. You may call on the compiler to produce a listing of your source file with true line numbers (as opposed to card sequence numbers) indicated in the left margin.

In addition, the compiler can produce listings that describe different aspects of the program and the compilation. These optional listings include:

- A listing of the generated code for the object file;
- An address map, showing the relative locations of Procedure Division lines;
- A map of data and procedures in the object file;
- A list of statistics describing the compilation (such as the number of lines compiled, the speed of the compilation);
- A source program cross-reference table.

You may request the listing file as well as any of the above listings in the compilation command for your program.

Calling the Compiler

To compile a COBOL source file, issue the COBOL command to the AOS Command Line Interpreter (CLI). The command has the format:

COBOL [*/sw/sw ...*] *sourcefile* [*listingfile/L*] [*objectfile/R*])

Where:

sw is a global switch to the compiler command that specifies an option you want to use in the compilation. See the list of global switches in the following section.

sourcefile is a filename that specifies the source program file you want compiled.

listingfile specifies the file or device to which you want the listing file output. It may be the console (@OUTPUT), the line printer (@LIST), or a disk or tape file.

objectfile is a filename that specifies the name you want assigned to the object file the compiler produces. If you do not supply a name, the compiler uses *sourcefile* with the extension .OB.

You must supply the source filename. If you designate any other optional filenames, you must append the appropriate local switches (/L or /R). The filenames may appear in any order. Append any global switches you use to the command word and any local switches to the appropriate filename. These switches may also appear in any order.

Compiler Switches

Global switches give the compiler information about the nature of your source file, and instruct it about the kind of output you want it to produce. The following list contains all the options available for a COBOL compilation.

/A Produce an address map of the relative locations of the Procedure Division lines.

/C Source is in card format. If you omit this switch, the compiler assumes the source is in text format. (See Chapter 2 for details on format.)

/D Compile debug lines and load the interactive debugger. Use this switch if your file includes debug lines and if you want COBOL to load the code for the debugger along with your source file code. (See Chapter 2 for details on debug lines and Chapter 9 for details on the debugger.)

/E Compile language extensions. Use this switch if you want octal values produced for alphanumeric literals (this conflicts with ANSI Standard COBOL features).

/G List the generated machine code. (This switch overrides the /A switch.)

/L List the source code at *listingfile*. If you did not name a listing file, @LIST is assumed.

/M List a map of data and procedure storage in the object file.

/P Do not generate an object file.

/Q Do not compile; simply scan the source file code and produce a cross-reference table.

/S List compilation statistics (the number of lines, the speed of compilation, etc.)

/W Suppress warning messages.

/X Include a cross-reference table in the listing.

The compiler normally produces its listing at either the line printer or the console. However, if you call for a listing of the source code, error messages, warning messages, generated code, map, compilation statistics, or cross-reference table, and you omit *listingfile* in the compilation command line, it will always be output to @LIST.

The local switches you may specify in the compilation command line are /L and /R as shown in the preceding list.

Example

The following command calls the COBOL compiler to compile a file named FILE1:

```
COBOL/L/X/W FILE1 FILE1.LS/L)
```

This command will compile the source file FILE1, and produce an object file named FILE1.OB (default name). The listing file FILE1.LS will contain source listing (/L), a cross-reference table (/X), and error messages (automatically). The compiler will suppress warning messages (/W).

Error Messages

The COBOL compiler always outputs a listing of all errors it detects in the source program during compilation. This error report appears either at the current console, or in a listing file if you specify one in the compilation command line. If more than fifty errors occur in any one phase of compilation, the compilation aborts.

For each error, the report identifies the true line number and the element in the line where the error occurred, followed by an English language diagnostic message describing the error. The compiler defines words, literals, special characters, and periods as elements of a line; commas, semicolons, and blanks do not count as elements.

In the following example, the input file contains an error in the declaration of a data item:

```
01 DAT-ITEM, PICTURE $$$999.99C,
   USAGE DISPLAY.
```

The compiler reports the error in this form:

LINE ELEMENT ERROR

0345 04 "R" MUST FOLLOW "C" IN PICTURE STRING

If you requested a listing of the source file, the output will have the number 0345 next to the line in question. The fourth element, $$$999.99C, contains the error.

Warning Messages

Normally the compiler generates warning messages in addition to error messages (unless you suppress them with the /W global switch). Warning messages have the same format as error messages.

The compiler signals a warning when it finds details in the program that are not incorrect syntactically, but are in some way inconsistent. For example, if you specify BLOCK CONTAINS *n* in a file's FD entry and *n* is less than the record length specified for the file, you will receive a warning message. You may safely ignore warning messages if you are sure your program will not create an error condition; you may, in fact, have reasons for apparently inconsistent coding.

Compiler errors will prevent your program from executing; you must correct and recompile your source file, then load and execute it. Warnings will not prevent execution, but may produce runtime errors or faulty computation.

For a complete list of the COBOL compiler's error and warning messages, see Appendix D.

Loading Program Files

After your program has been successfully compiled, you must issue the CBIND command to build an executable program file from your object file(s). CBIND loads your main file and any subprograms you specify, as a single program image.

The system normally loads COBOL programs and runtime routines into shared memory partitions. Of course it always loads data into unshared memory partitions.

Issue the CBIND command in the following format:

CBIND [*/sw/sw ...*] *name* [*subprog1 subprog2 ...*]

Where:

sw is a global switch if appended to CBIND, and a local switch if appended to *name*. It is a literal that gives the loader information about the input files and specifies the format of the save file you want it to create.

name is a filename that specifies the main program. You must supply this argument first. If you do not include a filename extension, CBIND assumes that it is .OB.

subprog is a filename that specifies a subprogram you want included in the loaded program.

CBIND Switches

The following list contains all the global switches for the CBIND command word:

/B List the symbol table in alphabetical and numerical order.

/D Bind in the COBOL debugger program. Load the COBOL program modules as unshared code. (CBIND automatically supplies the /S switch on each object file or subprogram.)

/E Output the load map to the output file, even though another listing file is specified.

/H List all numbers in hexadecimal.

/I Build a nonexecutable program file, lacking a UST, TCBs, and all other system databases (does not scan URT.LB).

/K=n Allocate n TCBs for multitask use, regardless how many (if any) are specified in a .TSK statement.

/L Produce a listing file, using the currently specified CLI @LIST file.

/L=name Produce a listing file, using the file name.

/M=n Reserve n 2K-byte blocks of memory for shared library routines.

/N Do not scan the user runtime library, URT.LB.

/O Suppress error flags whenever bind overwrites occur. A bind overwrite occurs when one module places code in one or more locations and a succeeding module overwrites these locations.

/Q Terminate binding after creating the files name.CK and name.CM which contain the bind command. You may edit these files. (PROG.CK performs the bind.)

/S Produce a shared routine for a shared library.

/T=n Specify the highest address in the shared partition. If n is not a multiple of 2048 bytes, the binder rounds it down to the next lower 2048-byte multiple. If you do not use this switch, your shared code partition will be placed at the top of the 64K-byte context.

/Z=n Specify the size of the stack for the default task. If you omit this switch, the system allocates a 25-word stack.

The following list contains all the local switches available with CBIND. You must append these to the appropriate *filename* or integer value.

name/B Bind the externally referenced routines from name, a shared library, into the root context.

/C Specify the name of a command file (required when defining overlays using square brackets).

/D Load nonshared code in this module as nonshared data. This is currently the only way that you can create a nonshared data partition, if you are using the macroassembler.

/H Load nonshared code in this module as shared code. If you apply this switch to a nonshared library, modules extracted from the library will be bound into the shared code partition. Note that the standard way you place code into the shared code partition when using the macroassembler is to use the .NREL 1 pseudo-op in your code.

/O Allow overwrites in this module (see /O function switch).

/R Issue a warning if any code in this module is not position-independent.

/S Convert shared code modules to unshared code modules. For example, COBOL currently produces shared code only; this switch lets a COBOL program be unshared. Note that you can also prevent or restrict file sharing by using the CLI ACL command, which employs the system's Access Control List facility.

/U Load local symbols from this module into the symbol file. This switch will work only if you applied /U to this same module in the earlier macroassembler command.

name/V=number Create an accumulating symbol, name, with absolute relocation, and initialize it to the value number. You can create more than one accumulating symbol by using this switch repeatedly. If you name an accumulating symbol that is also defined within a module as an accumulating symbol, there is no conflict; any value the .ASYM pseudo-op specifies will simply be added to the current sum of the accumulating symbol.

name/X Do not bind the shared library routine, name, into the root context. This switch must immediately follow the /B switch. For example, this command

```
...LIB/B A/X B/X...
```

is valid; it would tell the binder not to bind routines A and B into the root. This command

```
...LIB/B A/X C/R B/X...
```

does not use the /X switch correctly; the binder would not exclude routine B from the root in this case.

n/Z Set the current ZREL base to n. If the current ZREL base exceeds n, then the current base remains and n is ignored.

Link loading produces the following output: a listing of the file a load map, a symbol table, and a list of any loading error messages. These will appear at the current output console unless you specify a listing file. You may also have the error messages printed at the console while the rest of the listing goes to a different file (/E switch).

For a complete list of load error messages, see the *AOS Binder User's Manual,* (93-000190).

Splitting the Loading Process

The system loads your program in two stages. CBIND itself is a utility that creates a load command and stores it in the command files name.CK and name.CM. CBIND then passes control to the AOS binder (BIND) which actually loads the object code.

The CBIND global switch /Q allows you to split this two-step process. It instructs CBIND to stop after creating the command line, and before it passes control to BIND. Stopping the load operation allows you to do special load processing; you can edit the command file.

After editing the command file, execute the load with the command [name.CK] name.CM. For a full description of BIND and load processing, see the *AOS Binder User's Manual,* (93-000190).

Example:

```
CBIND/L=MYFILE.MP UPDATSUB HACKSUB)
```

The binder will link-load MYFILE.PR, the main program, and two subprograms, UPDATSUB and HACKSUB. The binder output listing will go to MYFILE.MP. If the load is successful, MYFILE will be ready for execution.

Executing Your COBOL Program

To execute a loaded COBOL object program, type the name of the program save file at the console. Program execution will begin at the main program's entry point. You may, in addition, append execution switches to the program name. You declare these switches in the Environment Division of your program. See the section "The Special-Names Paragraph" in Chapter 4 for information on execution switches.

The following examples show two calls that will execute the file NAMEFIL. The first calls for simple execution; the second uses execution switches declared in the program:

```
XEQ NAMEFIL)

XEQ NAMEFIL/C/A)
```

Executing in the Debugger

The COBOL debugger is a program that allows you to check your program's effectiveness. Using the debug program, you can execute your program piecemeal, halt its execution to examine its performance, make changes to the object code if necessary, and resume execution.

To use the debugger, you must load it along with your program and subprograms by using the /D switch in the CBIND command line.

To begin program execution in the debugger, type DEB before the save filename:

```
DEB MYPROG)
USER)

DEB MYPROG/C/A)
USER)
```

When the debugger is ready to accept a command, it will signal you with the prompt character *. See Chapter 9 for a complete description of the COBOL interactive debugger.

Runtime Errors

A runtime error condition occurs when, during the execution of your program, the system cannot properly complete an instruction. Error conditions commonly involve inconsistencies in runtime calls, discrepancies between an instruction and the current state of memory or files, and conflicts which evolve from hardware and software limitations.

There are two classes of errors: nonfatal and fatal. If a fatal error occurs, your program's execution terminates. Execution continues after a nonfatal error, but the results may be faulty. In a running COBOL program, five types of error conditions may occur.

I/O Exception Conditions: These error conditions are defined in Chapter 6. If your program declares an error-handling procedure, control goes to the handler and then execution resumes at the first executable statement following the statement that invoked the error handler.

Size Error, String/Unstring Overflow, Call Overflow, Search at End: The system does not output a message for these nonfatal errors. If your program declares an error-handling procedure, control goes to the handler and then execution resumes as it does for I/O exception conditions.

Nonfatal Program Errors: These error conditions are listed below. There is no error-handler option; execution simply continues. However, the system outputs COBOL trace information to the console (described later on in this chapter).

Fatal Program Errors: These error conditions are listed below. There is no error-handler option; the program's execution terminates and control passes to the CLI. The system outputs a COBOL error message and trace information (described later in this chapter) for the current program and the calling program.

Fatal System Errors: These error conditions are defined in the *AOS Reference Manual.* There is no error-handler option; the program's execution terminates and control passes to the CLI. The system outputs an AOS error message, and COBOL trace information (described later in this chapter) for the current program only.

Error Messages

The system displays error messages for COBOL program errors and system errors at the output console. For a list of all AOS system error messages, see the *AOS Reference Manual.* The following lists contain all COBOL runtime error messages for fatal and nonfatal program errors.

Nonfatal COBOL Program Errors

INTERMEDIATE OVERFLOW

An arithmetic operation gives an intermediate result with more than 31 digits. Processing continues with the high-order overflow truncated.

ILLEGAL ARGUMENT FOR EXP

The base of an exponentiation operation exceeds 16 x 10 (**63).

ILLEGAL EXPONENTIATION

Attempt to raise zero to the zero power, or raise a negative number to a nonzero fractional power.

INTEGER OVERFLOW ON CONVERSION

Loss of high-order significance in conversion of an external floating point number to internal floating point.

INVALID SIGN-NUMERIC
INVALID CHARACTER-NUMERIC

An invalid sign/invalid character has been detected in a numeric item.

Fatal COBOL Program Errors

STACK OVERFLOW

The system stack has overflowed its capacity.

PROGRAM ENTRY WITH INVALID ARGS

The arguments of a called program do not match the parameters passed by the calling program.

SUBSCRIPT OUT OF BOUNDS

A subscript item is outside the limits declared for the reference that contains it.

OCCURS-DEPENDING OUT OF BOUNDS

The depending data item contains a number exceeding the bounds of the occurrence item.

Trace Information

A COBOL error report consists of the error message, program name, and COBOL trace information. They appear in the following form:

```
CALLED FROM ( program id)
SEGMENT ( number)
RELATIVE LOCATION ( loc)
```

Where:

program id is the filename you specified in the Identification Division to identify this program.

number is the number of the segment containing the instruction.

loc is the relative address of the instruction that caused the error.

Example

```
CALLED FROM B
SEGMENT 35
RELATIVE LOCATION 563
```

The message gives the name of the save file, program A.PR, rather than the name of the subprogram, B, that actually called the subroutine. The address given in the trace information is the relative address in A.PR of the instruction that caused the error.

If you compile the program with the /A global switch (omitting the /L global switch), you will receive a table containing the line numbers of your Procedure Division statements beside a table containing each statement's relative address. You then reference relative location 563 in the table to determine the number of the line in the program which contains the instruction that caused the error. (If you specify both the /A and /L global switches, the system will output the relative address table on the line printer following your source file output).

End of Chapter

Appendix A
COBOL Reserved Words

ACCEPT
ACCESS
ACCESSABILITY
ADD
ADVANCING
AFTER
ALL
ALLOW
ALPHABETIC
ALSO
ALTER
ALTERNATE
AND
APPROXIMATE
ARE
AREA
AREAS
ASCENDING
ASCII
ASSIGN
AT
AUTHOR
BACKWARD
BEFORE
BIT
BLANK
BLOCK
BOTTOM
BY
C-300
CALL
CAM
CANCEL
CCNL
CD
CDAC
CDIS
CF
CH
CHANNEL
CHARACTER
CHARACTERS
CINT
CIOC
CLOSE
CMOD
CODE
CODE-SET
COLLATING
COLUMN
COMMA
COMMUNICATION
COMP
COMPRESSION
COMPUTATIONAL
COMP-1
COMPUTATIONAL-1
COMP-2
COMPUTATIONAL-2
COMP-3
COMPUTATIONAL-3
COMPUTE
CONFIGURATION
CONTAINS
CONTIGUOUS
CONTROL
CONTROLS
COPY
CORR
CORRESPONDING
COUNT
CR
CRCV
CREATE
CSND
CURRENCY
DATA
DATA-SENSITIVE
DATE
DATE-COMPILED
DATE-WRITTEN
DAY
DE
DECIMAL-POINT
DECLARATIVES
DEFINE
DELETE
DELIMITED
DELIMITER
DEPENDING
DESCENDING
DESTINATION
DETAIL
DISABLE
DISPLAY
DIVIDE
DIVISION
DOWN
DUPLICATES
DYNAMIC
EBCDIC
ECLIPSE
EGI
ELSE
EMT
ENABLE
END
END-OF-PAGE
ENTER
ENVIRONMENT
EOP
EQUAL
ERROR
ESI
EVEN
EVERY
EXCEPTION
EXCLUDE
EXCLUSIVE
EXIT
EXPIRATION
EXPUNGE
EXTEND
FD
FEEDBACK
FIELD
FIELDS
FILE
FILE-CONTROL
FILLER
FINAL
FIRST
FIX
FIXED
FOOTING
FOR
FORWARD
FROM
GENERATE
GENERATION
GENERIC
GIVING
GLOBAL
GO
GREATER
GROUP
HEADER
HEADING
HIERARCHICAL
HIGH
HIGH-VALUE
HIGH-VALUES
I-O
I-O-CONTROL
ID
IDENTIFICATION
IF
IMMEDIATE
IN
INDEX
INDEXED
INDICATE
INFOS
INITIAL
INITIALIZATION
INITIATE
INPUT
INPUT-OUTPUT
INSPECT
INSTALLATION
INTO
INVALID
INVERTED
IS
JUST
JUSTIFIED
KEY
KEYS
LABEL
LABELS
LAST
LEADING
LEFT
LENGTH
LESS
LEVELS
LIMIT
LIMITS

RDOS only

LINAGE
LINAGE-COUNTER
LINE-COUNTER
LINE
LINES
LINK
LINKAGE
LOCAL
LOCK
LOGICAL
LOW-VALUE
LOW-VALUES
LRU
MANAGEMENT
MAXIMUM
MEMORY
MERGE
MERIT
MESSAGE
MODE
MODULES
MOVE
MULTIPLE
MULTIPLY
NATIVE
NEGATIVE
NEXT
NO
NODE
NOT
NUMBER
NUMERIC
OBJECT-COMPUTER
OCCURRENCE
OCCURS
ODD
OF
OFF
OFFSET
OMITTED
ON
ONLY
OPEN
OPTIONAL
OR
ORGANIZATION
OUT
OUTPUT
OVERFLOW
OWNER
PAD
PAGE
PAGE-COUNTER
PARITY
PARTIAL
PERFORM
PF
PH
PHYSICAL
PIC
PICTURE
PLUS
POINTER
POSITION
POSITIVE
PRINTER
PRINTING
PROCEDURE
PROCEED
PROGRAM
PROGRAM-ID
QUEUE
QUOTE
QUOTES
RANDOM
RD
READ
RECEIVE
RECORD
RECORDING
RECORDS
REDEFINES
REEL
RELATIVE
RELEASE
REMAINDER
REMOVAL
RENAMES
REPLACING
REPORT
REPORTING
REPORTS
RERUN
RESERVE
RESET
RETAIN
RETRIEVE
RETURN
REWIND
REWRITE
RF
RH
RIGHT
ROOT
ROUNDED
RUN
SAME
SAVE
SD
SEARCH
SECTION
SECURITY
SEEK
SEGMENT
SEGMENT-LIMIT
SELECT
SEND
SENTENCE
SEPARATE
SEQUENCE
SEQUENTIAL
SET
SIGN
SIZE
SORT
SORT-MERGE
SOURCE
SOURCE-COMPUTER
SPACE
SPACES
SPECIAL-NAMES
STANDARD
STANDARD-1
STANDARD-2
STANDARD-3
START
STATIC
STATUS
STOP
STRING
SUB-INDEX
SUB-QUEUE-1
SUB-QUEUE-2
SUB-QUEUE-3
SUBTRACT
SUM
SUPPRESS
SWITCH
SYMBOLIC
SYNC
SYNCHRONIZED
TABLE
TALLYING
TAPE
TEMPORARY
TERMINAL
TERMINATE
TEXT
THAN
THEN
THROUGH
THRU
TIME
TIMES
TO
TOP
TRAILER
TRAILING
TRUNCATE
TYPE
UNDEFINED
UNDELETE
UNIT
UNLOCK
UNSTRING
UNTIL
UP
UPON
USAGE
USE
USER
USING
VALUE
VALUES
VARIABLE
VARYING
VERIFY
VIRTUAL
VOLUME
WAIT
WHEN
WITH
WORDS
WORKING-STORAGE
WRITE
XECS
XMOD
XNMT
XPND
XTRN
ZERO
ZEROES
ZEROS

End of Appendix

RDOS only

Appendix B
Language Differences Between RDOS and AOS

Identification Division (Chapter 3)

None.

Environment Division (Chapter 4)

1. Under RDOS, sequential, random, indexed, and multilevel indexed files are handled by the RDOS INFOS file system. Under AOS, sequential and random files are handled by the AOS operating system, and indexed and multilevel indexed files are handled by the AOS INFOS file system.

2. RDOS supports multivolume sequential and random files, AOS does not.

3. In the SELECT clause:

   - RDOS supports merit factors assigned in the MERIT and ROOT MERIT clauses; you may specify them in AOS, but they have no meaning.

   - Under AOS, you must specify CONTIGUOUS, so that the VOLUME SIZE clause will have meaning for the operating system.

   - RDOS supports INITIALIZATION of sequential, random, and indexed files; you may specify it in AOS, but it has no meaning.

   - You may create TEMPORARY indexes in RDOS; you may specify them in AOS, but they will not be temporary.

   - You may request a specific buffer management technique by specifying the HIERARCHICAL/LRU clause in RDOS; you may specify it in AOS, but it has no meaning.

   - You may RESERVE I/O buffers in RDOS; you may specify this for AOS, but it has no meaning.

   - You may specify either odd or even PARITY in RDOS; parity is always odd in AOS.

   - In RDOS, if you specify the OCCURRENCE clause, the system automatically assigns the first key an occurrence number of zero. It assigns an occurrence number of 1 to the first duplicate key you write, an occurrence number of 2 to the second duplicate key, etc., assigning incremental numbers to the records in the order in which you write them. Then you simply specify a READ statement for the duplicate key with an occurrence number of 0, and you can read the duplicate key records sequentially.

     In AOS, the occurrence numbers do not denote the occurrence of a particular key; they establish uniqueness among all keys of a subindex. If you are searching in a subindex that allows duplicates, AOS COBOL will return the occurrence number only if you are positioned on a duplicate key.

   - You may save room in an index by using KEY COMPRESSION in RDOS; you may specify it in AOS, but it has no meaning.

   - The RDOS INFOS status register data item is three characters in length; the AOS INFOS status register data item is four characters in length.

Data Division (Chapter 5)

In the FD entry:

- RDOS supports merit factors assigned in the MERIT clause; you may specify them in AOS, but they have no meaning.
- You may specify your own PAD character in RDOS; you may specify it in AOS, but the null character is used.
- In RDOS, you must specify BLOCK CONTAINS 512 CHARACTERS in a PRINTER file's FD entry if you later want to do a CLI XFER of that file. This is not necessary under AOS.
- Under RDOS, the BLOCK CONTAINS clause in an FD entry specifies the maximum size of a data file's logical block. Under AOS (for disk files), this clause determines the number of physical disk blocks the operating system will transfer to or from its buffers on each I/O operation for a data file. Careful selection of the block size can result in improved performance. For example, if you want to access a file sequentially, you might specify BLOCK CONTAINS 1024 CHARACTERS in the file's FD entry. AOS would then access two blocks for each READ operation, which would cut down on disk access time.

Procedure Division (Chapters 6 and 7)

1. The following features function in an RDOS environment. You may specify them in AOS, but they have no meaning:
   - REWINDing magnetic tape volumes (OPEN and CLOSE statements); AOS never rewinds;
   - KEY COMPRESSION for indexed files (DEFINE SUB-INDEX statement);
   - Controlling buffer space allocation with the EXCLUDE/ONLY option (OPEN statement);
   - SEEK statement;
   - LOCKing file volumes (CLOSE statement);
   - REEL/UNIT specifications (CLOSE statement);
   - LOCKing sequential or relative file records (READ and WRITE statements);
   - REMOVAL of a file volume (CLOSE statement);
   - WAITing to see if a READ statement is attempting to access a locked record;
   - IMMEDIATEly writing a block out to a file (WRITE and REWRITE statements);
   - Setting up a work file in a SORT statement (default is WORK.WI); there is no work file in AOS.
2. The features that function under AOS, but not under RDOS are:
   - End of file detected on relative file records;
   - Compilation of programs in BATCH mode;
   - ALLOW DUPLICATES option in the DEFINE SUB-INDEX statement.

3. The features that exist in both RDOS and AOS, but behave differently are:

   - If you LOCK a record in RDOS, no one, including yourself, can access that record until it is UNLOCKed. In AOS, the process ID determines locked record access; no one can access the LOCKed record except the process ID that LOCKed it (CLOSE, READ, WRITE, and REWRITE statements).

   - RDOS INFOS subindex definition packet is 12 characters in length; the AOS INFOS subindex definition packet is 16 characters in length (DEFINE SUB-INDEX statement);

   - If the current position of a file's record pointer is at a record you delete, in RDOS the record pointer points to the record immediately following the deleted record. In AOS, the record pointer points to the record immediately before the deleted record (DELETE statement).

4. The Communications Access Manager (CAM) supplies eight statements supported by RDOS, but not AOS (the keywords do not exist in AOS): CCNL, CDAC, CDIS, CINT, CIOC, CMOD, CRCV, and CSND.

End of Appendix

Appendix C
Writing COBOL-Callable Assembly Language Routines

ECLIPSE COBOL programs can call routines written in assembly language, provided that these routines conform to COBOL runtime linkage conventions.

The COBOL statement:

```
CALL 'SUBX' USING ARG-1,ARG-2,...,ARG-N
```

generates the following code:

```
JSR     @.CAL
N+1
.EXTN   SUBX
SUBX            ;ADDRESS OF CALLED ROUTINE
                ;ENTRY POINT
P.1             ;ARG-1 POINTER
P.2             ;ARG-2 POINTER
.
.
.
P.N             ;ARG-N POINTER
```

Where each P.i is a word pointer to a byte pointer to the respective argument in the CALL statement.

The CALL runtime routine pointed to by .CAL does the following:

1. verifies that the called routine was loaded,
2. saves the state of the calling routine on the stack,
3. transfers control to the called routine.

In order to correctly interface to the COBOL runtime system, your called routine must conform to the following conventions:

1. The name used in the CALL statement must reference the routine as an entry point, in an .ENT statement. The entry point label must be five characters or less in length.

2. The entry point must be an EJMP instruction.

3. Your routine must declare .ENTR and .EXIT in an .EXTD statement.

4. Your routine must obtain argument pointers by calling ENTR (JSR @.ENTR).

5. Your routine must return to COBOL by calling EXIT (JSR @.EXIT).

6. If your called routine uses the stack for temporary storage, take care to ensure that it also restores the stack correctly. If it fails to do so, the program will fail during return linkage.

For example, the following code will call ENTR and copy pointers to the arguments passed by the COBOL program:

```
JSR     @.ENTR
N               ;# OF ARGUMENTS--MUST COUNT
                ;IN THE CALL STATEMENT
PLIST           ;POINTER TO PARAMETER LIST
```

The parameter list has the following form:

```
PLIST:  N
P1:     0       ;BYTE PTR TO ARG PUT BY
                ;ENTR
        0       ;USED BY COBOL
        0       ;SEQUENCE # OF ARG IN
                ;CALL STATEMENT
        0       ;USED BY COBOL

P2:     0
        0
        1
        0
P3:     .
        .
        .
PN:     0
        0
        N-1
        0
```

Following the call to ENTR, byte pointers to each passed argument will appear in the first word of the four-word parameter list entry for each argument.

Return to COBOL by calling EXIT:

```
JSR @.EXIT
```

AOS and RDOS examples follow in Figures C-1 and C-2.

```
02                      ;
03                      ;
04                      ;       THIS IS AN EXAMPLE OF A MODULE WRITTEN IN ASSEMBLER
05                      ;       LANGUAGE THAT IS CALLABLE FROM A COBOL PROGRAM.
06                      ;       ITS FUNCTION IS TO PASS THE CONTENTS OF THE COMMAND
07                      ;       LINE FILE -(F)COM.CM- TO THE COBOL PROGRAM THAT CALLS
08                      ;       IT. THE CALL FORMAT IS:
09                      ;
10                      ;               CALL "CMARG" USING CHAN FILE SWITCHES.
11                      ;
12                      ;               CHAN - A 2-BYTE DATA ITEM WHOSE INITIAL VALUE
13                      ;                       IS HIGH-VALUES. THIS FIELD IS USED BY
14                      ;                       THE CMARG ROUTINE TO SAVE AN RDOS CHANNEL
15                      ;                       NUMBER AND MUST NOT BE MODIFIED BY THE
16                      ;                       COBOL PROGRAM.
17                      ;
18                      ;               FILE - A DATA ITEM THAT WILL RECEIVE THE FILENAME
19                      ;                       FROM THE COMMAND LINE, LEFT JUSTIFIED.
20                      ;                       IF THE 1ST BYTE OF 'FILE' CONTAINS HIGH-
21                      ;                       VALUES WHEN CMARG IS CALLED, A NULL BYTE WILL
22                      ;                       BE INSERTED FOLLOWING THE FILENAME.'FILE' MUST
23                      ;                       BE LONG ENOUGH TO HOLD THE LARGEST FILENAME
24                      ;                       EXPECTED, AS NO LENGTH CHECKS ARE MADE.
25                      ;
26                      ;               SWITCHES - A 4-BYTE DATA ITEM THAT WILL RECEIVE THE
27                      ;                       SWITCHES ASSOCIATED WITH THE COMMAND LINE
28                      ;                       ARGUMENT RETURNED IN 'FILE'.
29                      ;
30                      ;       THE FIRST CALL TO CMARG WILL RETURN THE FIRST ARGUMENT
31                      ;       IN (F)COM.CM, THE SECOND WILL RETURN THE SECOND ARGUMENT
32                      ;       AND SO ON. END OF FILE IS INDICATED BY RETURNING HIGH-
33                      ;       VALUES IN 'CHAN'.
34                      ;
35                      ;
36                      ;       THROUGH THE COMMENTS, THE SYMBOL '=>' MEANS
37                      ;       'BYTE POINTER TO' AND '->' MEANS 'WORD POINTER TO '
38                      ;
39                              .ENT    CMARG
40                              .EXTD   .ENTR,.EXIT
41      000001                  .TXTM   1
42
43                              .NREL
44
45                      CMARG:
46 00000'102470                 EJMP    BGN             ;1ST INSTR MUST BE EJMP
47      000001
48
49 00002'006002$        BGN:    JSR     @.ENTR          ;COPY POINTERS
50 00003'000003                 3                       ;# OF ARGUMENTS
51 00004'000101'                PLIST                   ;PARAMETER LIST ADDR
52 00005'034475                 LDA     3,.CHN          ;=>'CHAN'
53 00006'175220                 MOVZR   3,3             ;->'CHAN'
54 00007'031400                 LDA     2,+0,3          ;LOAD CHANNEL
55 00010'150014                 COM#    2,2,SZR         ;IS IT -1?
56 00011'000417                 JMP     READ            ;NO
57
58                      ; 1ST TIME THROUGH (CHAN=-1) GET A CHANNEL
59                      ; FROM RDOS (.GCHN), SAVE IT IN CHAN, FIGURE OUT
60                      ; WHICH GROUND YUO'RE IN, AND OPEN THE PROPER
```

Figure C-1. RDOS Assembly Language Routine Example

```
01                              ; COMMAND FILE
02
03  00012'177110                       PSH     3,3             ;SAVE ADDR OF 'CHAN' ON STACK
04  00013'006017                       .SYST                   ;CALL RDOS
05  00014'021052                       .GCHN                   ;GET A FREE CHANNEL
06  00015'000451                       JMP     BAD             ;HARD CHEESE
07  00016'177210                       POP     3,3             ;RETREIVE ADDR OF 'CHAN'
08  00017'051400                       STA     2,+0,3          ;SAVE CHANNEL
09  00020'020451                       LDA     0,.FCOM         ;ASSUME FOREGROUND
10  00021'036012                       LDA     3,@USTP         ;LOAD 1ST WORD OF UST
11  00022'175015                       MOV#    3,3,SNR         ;0=BG, NOT 0=FG
12  00023'101400                       INC     0,0             ;BUMP=>FILENAME
13  00024'126400                       SUB     1,1             ;CLEAR AC1
14  00025'006017                       .SYST                   ;CALL RDOS
15  00026'014077                       .OPEN   CPU             ;OPEN COMMAND FILE
16  00027'000437                       JMP     BAD             ;POOP
17
18                              ;READ COMMAND FILE --
19                              ;
20                              ;1) CHECK 1ST BYTE OF 'FILE' FOR HIGH-VALUES
21                              ;2) DO A READ-LINE TO GET FILENAME ARGUMENT FROM
22                              ;       COMMAND FILE
23                              ;3) DO A READ-SEQUENTIAL (4 BYTES) TO GET SWITCHES
24                              ;       FROM COMMAND FILE
25                              ;4) PUT SPACE AFTER FILENAME UNLES NULL REQUESTED
26
27
28  00030'020456        READ:   LDA     0,.FNM          ;=> 'FILE'
29  00031'106710                LDB     0,1             ;GET 1ST BYTE OF 'FILE'
30  00032'127110                PSH     1,1             ;SAVE IT FOR LATER
31  00033'006017                .SYST                   ;CALL RDOS
32  00034'015477                .RDL    CPU             ;READ LINE
33  00035'000420                JMP     EOFCK           ;CHECK FOR EOF
34  00036'104110                SBI     1,1             ;SUBTRACT 1 FROM READ COUNT
35  00037'107000                ADD     0,1             ;=>1ST BYTE AFTER FILENAME
36  00040'127110                PSH     1,1             ;SAVE ON STACK
37  00041'020451                LDA     0,.SW           ;=> 'SWITCHES'
38  00042'166070                ELEF    1,4             ;READ 4 BYTES
39        000004
40  00044'006017                .SYST                   ;CALL RDOS
41  00045'015077                .RDS    CPU             ;READ SWITCHES
42  00046'000420                JMP     BAD             ;NO EOF HERE, MUST BE ERROR
43  00047'123210                POP     1,0             ;GET PTR AND FLAG FROM STACK
44  00050'030427                LDA     2,NFLG          ;LOAD FLAG FOR COMPARE
45  00051'034425                LDA     3,SPC           ;LOAD ASCII SPACE
46  00052'112404                SUB     0,2,SZR         ;NULL REQUESTED?
47  00053'137010                STB     1,3             ;NO, STORE SPACE
48
49                              ;RETURN TO COBOL
50
51  00054'006001$       BACK:   JSR     @.EXIT
52
53                              ;ERROR RETURN ON READ-LINE, CHECK FOR EOF
54
55  00055'103210        EOFCK:  POP     0,0             ;CLEAN UP STACK
56  00056'020422                LDA     0,EOFLG         ;GET EOF FLAG
57  00057'112414                SUB#    0,2,SZR         ;IS IT?
58  00060'000406                JMP     BAD             ;NO SUCH LUCK
59  00061'030421                LDA     2,.CHN          ;=> 'CHAN'
60  00062'151220                MOVZR   2,2             ;-> 'CHAN'
```

Figure C-1. RDOS Assembly Language Routine Example (continued)

```
01 00063'102000                     ADC     0,0             ;MAKE -1
02 00064'041000                     STA     0,+0,2  ;SHOW EOF CONDITION
03 00065'000767                     JMP     BACK            ;OFF YOU GO
04
05                          ;SYSTEM ERRCR
06
07 00066'006017             BAD:    .SYST                   ;CALL RDOS
08 00067'006400                     .ERTN                   ;FOR THE LAST TIME
09 00070'000400                     JMP
10
11                          ;ASSORTED STUFF
12
13 00071'000164"            .FCOM:  .+1*2                   ;=>COMMAND FILENAME
14 00072'043103                     .TXT    /FCOM.CM/
15       047515
16       027103
17       046400
18
19
20 00076'000040             SPC:    40
21 00077'000177             NFLG:   177
22 00100'000006             EOFLG:  6
23
24                          ;PARAMETER TABLE
25
26 00101'000003             PLIST:  3
27 00102'000000             .CHN:   0
28 00103'000000                     0
29 00104'000000                     0
30 00105'000000                     0
31 00106'000000             .FNM:   0
32 00107'000000                     0
33 00110'000001                     1
34 00111'000000                     0
35 00112'000000             .SW:    0
36 00113'000000                     0
37 00114'000002                     2
38 00115'000000                     0
39
40                                  .END

**00000 TOTAL ERRORS, 00000 PASS 1 ERRORS
```

Figure C-1. RDOS Assembly Language Routine Example (continued)

```
IDENTIFICATION DIVISION.
PROGRAM-ID. CTSWHO.
*               CALLS ASSEMBLY LANGUAGE ROUTINE WHO TO
*               GET PROCESS ID AND CONSOLE NAME.

ENVIRONMENT DIVISION.

DATA DIVISION.
WORKING-STORAGE SECTION.
01      PID     PIC 999.
01      CONSOLE PIC X(32).

PROCEDURE DIVISION.
        MOVE SPACES TO CONSOLE.
        CALL "WHO" USING PID,CONSOLE.
        DISPLAY "PROCESS ID: ",PID.
        DISPLAY "CONSOLE:    ",CONSOLE.
        STOP RUN.
        .TITLE  WHO2
        .ENT    WHO

; THIS IS AN ASSEMBLY LANGUAGE SUBPROGRAM CALLABLE FROM
; A COBOL PROGRAM.  IT RETURNS PROCESS ID AND CONSOLE OR
; STREAM NAME.
; CALLING SEQUENCE:
;  CALL "WHO" USING <PID-VARIABLE> <CONSOLE-VARIABLE>

        .EXTD   .ENTR .EXIT

        .NREL   0               ; UNSHARED CODE

; PARAMETER TABLE
PLIST:  2               ; 2 PARAMETERS

.PID:   0               ; BECOMES BYTE ADDRESS OF PID VARIABLE
.PIDA:  202             ; CIS ATTR (TYPE=UNSIGNED,LEN=3)
        0               ; SEQUENCE NUMBER
        0               ; USED BY COBOL

.CON:   0               ; BECOMES BYTE ADDRESS OF CONSOLE VARIABLE
.CONL:  32.             ; BYTE LENGTH OF CONSOLE VARIABLE
        1               ; SEQUENCE NUMBER
        0               ; USED BY COBOL

; AOS PACKET FOR ?EXEC CALL
APACK:
**      .DO ?XLTH
        0
**      .ENDC

; TEMPORARY BUFFER FOR CONSOLE NAME
.BUF:   BUF*2
BUF:    .BLK    16.


        .NREL   1               ; SHARED CODE

; 16 WORD TRANSLATION TABLE FOR CMT INSTRUCTION
; BIT FOR NULL IS ON, ALL OTHERS ARE OFF
TRAN:   100000
** .DO 15.
```

Figure C-2. AOS Assembly Language Routine Example

```
; SUBPROGRAM STARTS HERE
WHO:    EJMP    WHO2            ; MUST START WITH EJMP

; INITIALIZATION OF PARAMETER LIST
WHO2:   JSR     @.ENTR
        2                       ; # PARAMETERS
        PLIST                   ; ADDRESS OF PARAMETER LIST

; MAKE AOS CALL TO GET PROCESS ID
        SUB     0,0             ; AC0 GETS 0
        ADC     1,1             ; AC1 GETS -1
        ?PNAME
        JMP     .+1             ; ERROR NOT POSSIBLE

; PID IS IN AC1, FLOAT TO FPAC0
        FLAS    1,0

; STORE THE PID IN THE PID VARIABLE
        ELDA    1,.PIDA         ; CIS ATTRIBUTE OF PID
        ELDA    3,.PID          ; BYTE ADDRESS OF PID VARIABLE
        STI     0

; MAKE AOS CALL TO GET CONSOLE NAME TO BUF
        ELEF    2,APACK         ; ACS PACKET ADDRESS TO AC2
        LEF     0,?XFSTS        ; EXEC FUNCTION IS STATUS
        STA     0,?XRFNC,2
        ELDA    0,.BUF          ; TEMP BUFFER BYTE ADDR TO PACKET
        STA     0,?XFP2,2
        ?EXEC
        JMP     DONE            ; QUIT IF ANY ERROR

; MOVE CONSOLE NAME FROM BUF TO <CONSOLE> VARIABLE
; MOVE CHARACTERS UNTIL NULL ENCOUNTERED
        ELEF    0,TRAN          ; TRANSLATION TABLE WORD ADDR
        ELDA    1,.CONL         ; BYTE LENGTH OF CONSOLE VARIABLE
        ELDA    2,.CON          ; DESTINATION BYTE ADDR
        ELDA    3,.BUF          ; SOURCE BYTE ADDRESS
        CMT

; SUBPROGRAM IS DONE, RETURN TO CALLING PROGRAM
DONE:   JSR     @.EXIT


        .END
```

Figure C-2. AOS Assembly Language Routine Example (continued)

End of Appendix

Appendix D
Compiler Error and Warning Messages

```
'"IDENTIFICATION" EXPECTED',
'"DIVISION" EXPECTED',
'ENVIRONMENT DIVISION NOT SEEN',
'PERIOD EXPECTED',
'ILLEGAL PARAGRAPH HEADER',
'ILLEGAL PROGRAM-ID, IDENTIFIER EXPECTED',
'ILLEGAL CHARACTER INSIDE "<...>"',
'LITERAL LONGER THAN 132 CHARACTERS',
'ILLEGAL SPECIAL CHARACTER PAIR',
'ILLEGAL CHARACTER',
'ILLEGAL COPY TEXT-NAME, STRING LITERAL EXPECTED',
'ILLEGAL COPY LIBRARY NAME',
'ILLEGAL ELEMENT IN COPY STATEMENT',
'ILLEGAL ELEMENT IN COPY REPLACING STRING',
'MORE THAN 10 ELEMENTS IN COPY REPLACING STRING',
'MORE THAN 10 REPLACING PAIRS IN COPY STATEMENT',
'PROGRAM-ID DEFAULTED TO "MAIN"',
'INCOMPLETE COPY STATEMENT',
'INCOMPLETE COPY REPLACING CLAUSE',
'OMITTED SPACE OR ILLEGAL SEQUENCE OF CHARACTERS',
'ALPHABETIC CHARACTER IN NUMERIC LITERAL',
'FATAL COMPILE ERROR: TOO MANY USER DEFINED WORDS',
'FATAL COMPILE ERROR: SPELLING TABLE OVERFLOW',
'COPY FILE NOT FOUND',
'ILLEGAL PROGRAM-ID. MUST CONTAIN ALPHANUMERIC CHARACTERS ONLY',
'MISSING COPY FILES. COMPILE ABORTED.'


        0
** .ENDC
```

```
'DATA NAME NOT DECLARED'
'CURRENCY SIGN MUST BE ONE CHARACTER IN LENGTH'
'ILLEGAL CHARACTER FOR CURRENCY SIGN'
'NONNUMERIC CHARACTER IN PARENTHESIS OF PICTURE STRING'
'ILLEGAL CHARACTER IN PICTURE STRING'
'"R" MUST FOLLOW C IN PICTURE STRING'
'"B" MUST FOLLOW D IN PICTURE STRING'
'PICTURE STRING TOO LONG'
'INVALID CHARACTER COMBINATION IN PICTURE STRING'
'INVALID SIGN CHARACTER IN EXT. FLT. PT. PICTURE STRING'
'"V" AND "." CANNOT APPEAR IN SAME PICTURE STRING'
'MORE THEN ONE "E" IN PICTURE STRING'
'MISSING SIGN IN MANTISSA OF EXT. FLT. PT. PICTURE STRING'
'MISPLACED SIGN IN EXT. FLT. PT. PICTURE STRING'
'NO "9" IN MANTISSA OF EXT. FLT. PT. PICTURE STRING'
'MAXIMUM OF 16 DIGITS ALLOWED IN MANTISSA OF EXT. FLT. PT. PICTURE'
'INVALID EXPONENT OF EXT. FLT. PT. PICTURE'
'ONLY ONE "V" ALLOWED IN PICTURE STRING'
'ILLEGAL USE OF "S" IN PICTURE STRING'
'NO NUMERIC POSITIONS IN NUMERIC PICTURE STRING'
'NO MORE THEN 18 DIGIT POSITIONS ALLOWED IN NUMERIC DATA ITEM'
'ILLEGAL USE OF "V" AND "P" IN PICTURE STRING'
'"P" CAN ONLY BE ON EITHER THE LEFT OR RIGHT OF "9" IN PICTURE STRING'
'MULTIPLE FLOAT CHARACTERS NOT ALLOWED IN PICTURE STRING'
'FLOATING INSERTION CHARACTERS MUST BE RIGHTMOST IN PICTURE STRING'
'MUST HAVE A STRING OF AT LEAST 2 FLOATING INSERTION CHARACTERS'
'ONLY ONE FIXED INSERT "+","-","CR" OR "DB" CAN APPEAR IN THE PICTURE '
'"." AND "P" CANNOT APPEAR IN THE SAME PICTURE STRING'
'FIXED INSERTION CHARACTER CANNOT FOLLOW "P" BEFORE DECIMAL'
'ILLEGAL USE OF FLOATING INSERTION CHARACTER IN PICTURE STRING'
'FIXED INSERTION "+" OR "-" MUST BE LEFT OR RIGHT MOST CHARACTER'
'"CR" OR "DB" MUST BE LEFT OR RIGHT MOST CHARACTER'
'NONFLOAT CURRENCY SIGN MUST BE RIGHT MOST CHARACTER'
'COMPILER STACK OVERFLOW'
'  '
'  '
'DUPLICATE DEFINITION OF NAME'
'TOO MANY RECORDS, KEYS, OR LEVEL 88 DATA-ITEMS DECLARED'
'WORKING STORAGE OVERFLOW. DATA-ITEM EXCEEDS 32767 CHARACTERS'
'WORKING STORAGE OVERFLOW. DATA AREA EXCEEDS 65535 CHARACTERS'
'REDEFINED ITEM NOT DECLARED'
'"OCCURS DEPENDING" CLAUSE MUST BE THE LAST AREA DEFINED IN THE RECORD'
'"OCCURS DEPENDING" ITEM CANNOT BE SUBORDINATE TO ITEM WITH OCCURS'
'CANNOT REDEFINE ITEMS WITH DIFFERENT LEVEL NUMBERS'
'ITEM BEING REDEFINED CANNOT HAVE OCCURS CLAUSE'
'ITEM BEING REDEFINED CANNOT HAVE A SUBORDINATE "OCCURS DEPENDING"'
'SIZE OF AREA BEING REDEFINED NOT EQUAL'
'AREA BEING REDEFINED CANNOT HAVE "OCCURS DEPENDING"'
'ILLEGAL DATA TYPE BEING RENAMED'
'RENAMED ITEM MUST BE IN THE SAME RECORD'
'CANNOT RENAME A SUBSCRIPTED ITEM'
'SECOND AREA BEING RENAMED CANNOT OCCUR BEFORE THE FIRST'
'PICTURE CLAUSE MUST CONTAIN A SIGN CHARACTER'
'NON FLOATING POINT ITEMS CANNOT BE INITIALIZED WITH F.P. LITERALS'
'ELEMENTARY ITEM CANNOT BE FOLLOWED BY A HIGHER LEVEL NUMBERED ITEM'
'VALUE CLAUSE CANNOT BE USED WITH REDEFINES ITEM'
'INITIAL VALUE NOT ALLOWED WHEN GROUP ITEM HAS INITIAL VALUE'
'PICTURE CLAUSE NOT ALLOWED WITH USAGE COMP-1,COMP-2, OR INDEX'
'MISSING PICTURE CLAUSE ON ELEMENTARY ITEM'
'  '
```

```
'  '
'MISSING "FILE-CONTROL" SENTENCE'
'MISSING "ASSIGN CLAUSE" '
'ONLY 1 ALTERNATE RECORD KEY ALLOWED WITH MULTIPLE RECORD KEYS'
'THIS DATA ITEM MAY NOT BE DECLARED IN THE LINKAGE SECTION'
'  '
'  '
'  '
'DUPLICATE "ASSIGN DATA" CLAUSE'
'DUPLICATE "INDEX AREAS" CLAUSE'
'DUPLICATE "DATA AREAS" CLAUSE'
'DUPLICATE "FILE ORGANIZATION" CLAUSE'
'DUPLICATE "ACCESS MODE" CLAUSE'
''
'DUPLICATE "FILE STATUS" CLAUSE'
'DUPLICATE "SUBINDEX" CLAUSE'
'DUPLICATE "KEY COMPRESS" CLAUSE'
'  '
'  '
'  '
'  '
'OPTIONAL FILE ONLY ALLOWED WITH SEQUENTIAL ORGANIZATION'
'DECLARATIONS IN "ASSIGN" CLAUSE REQUIRES INDEXED ORGANIZATION'
'"ASSIGN DATA" ONLY ALLOWED WITH INDEXED ORGANIZATION'
'"INDEX AREAS" ONLY ALLOWED WITH INDEXED ORGANIZATION'
'"RESERVE DATA AREAS" ONLY ALLOWED WITH INDEXED ORGANIZATION'
'  '
'"ACCESS MODE" CLAUSE CONFLICTS WITH FILE ORGANIZATION'
'RECORD KEYS ONLY ALLOWED WITH INDEXED ORGANIZATION'
'  '
'SUBINDEX ONLY ALLOWED WITH INDEXED ORGANIZATION'
'"KEY COMPRESS" ONLY ALLOWED WITH INDEXED ORGANIZATION'
'"PARITY" CLAUSE ONLY ALLOWED WITH SEQUENTIAL ORGANIZATION'
'ALTERNATE RECORD KEYS ONLY ALLOWED WITH INDEXED ORGANIZATION'
'MERIT FACTOR ONLY ALLOWED WITH INDEXED ORGANIZATION'
'  '
'MISSING "FILE SECTION" HEADER'
'FILE NOT DECLARED IN ENVIRONMENT DIVISION'
'ALPHABET NAME NOT DECLARED IN "SPECIAL NAMES"'
'ALPHABET NAME CANNOT BE USER DEFINED IN FILE SECTION'
'BLOCK SIZE MUST BE AT LEAST AS LARGE AS ONE RECORD'
'ILLEGAL NAME IN DATA RECORDS CLAUSE'
'RECORD NAME IN DATA RECORDS CLAUSE NOT A RECORD OF THIS FILE'
'  '
'  '
'  '
'MULTIPLE "BLOCK CONTAINS" CLAUSE'
'  '
'MULTIPLE "INDEX BLOCK CONTAINS" CLAUSE'
'MULTIPLE "INDEX NODE SIZE" CLAUSE'
'MULTIPLE "RECORD CONTAINS" CLAUSE'
'MULTIPLE "LABEL RECORD" CLAUSE'
'MULTIPLE "DATA RECORD" CLAUSE'
'MULTIPLE "FEEDBACK" CLAUSE'
'MULTIPLE "PARTIAL RECORD" CLAUSE'
'MULTIPLE "PAD CHARACTER" CLAUSE'
'MULTIPLE "CODE SET" CLAUSE'
'MULTIPLE "VALUE OF" CLAUSE'
'MULTIPLE "LINAGE" CLAUSE'
'MULTIPLE "RECORDING MODE" CLAUSE'
```

```
'DUPLICATE "MERIT" CLAUSE'
'MULTIPLE "OWNER" CLAUSE IN "VALUE OF" STATEMENT'
'MULTIPLE "EXPIRATION DATE" CLAUSE IN "VALUE OF" STATEMENT'
'MULTIPLE "SEQUENCE NUMBER" CLAUSE IN "VALUE OF" STATEMENT'
'MULTIPLE "GENERATION NUMBER" CLAUSE IN "VALUE OF" STATEMENT'
'MULTIPLE "ACCESSABILITY" CLAUSE IN "VALUE OF" STATEMENT'
'MULTIPLE "DATA OFFSET" CLAUSE IN "VALUE OF" STATEMENT'
'MULTIPLE "USER VOLUME" CLAUSE IN "VALUE OF" STATEMENT'
'MULTIPLE "USER HEADER" CLAUSE IN "VALUE OF" STATEMENT'
'MULTIPLE "USER TRAILER" CLAUSE IN "VALUE OF" STATEMENT'
'MULTIPLE "VOLUME STATUS" CLAUSE '
'   '
'"BLOCK CONTAINS RECORDS" NOT ALLOWED WITH INDEXED FILES'
'"INDEXED BLOCK CONTAINS" ONLY ALLOWED WITH INDEXED FILES'
'"INDEX NODE SIZE" ONLY ALLOWED WITH INDEXED FILES'
'   '
'   '
'   '
'   '
'"PARTIAL RECORDS" ONLY ALLOWED WITH INDEXED FILES'
'"PAD CHARACTER" NOT ALLOWED WITH INDEXED FILES'
'   '
'"VALUE OF" ONLY ALLOWED WITH SEQUENTIAL FILES'
'"LINAGE" ONLY ALLOWED WITH SEQUENTIAL FILES'
'"RECORDING MODE" ONLY ALLOWED WITH SEQUENTIAL FILES'
'"RECORD MERIT" ONLY ALLOWED WITH INDEXED FILES'
'LITERAL PRECEDING "THROUGH" MUST BE ONE CHARACTER IN LENGTH'
'INTEGER CANNOT EXCEED 128'
'CANNOT REDEFINE A RECORD OF A DIFFERENT FILE'
'INVALID SWITCH NAME'
'"THRU <LITERAL>" OF COL. SEQ. MUST BE ON THE RIGHT OF FIRST <LITERAL>'
'INVALID "SEGMENT LIMIT" NUMBER'
'   '
'   '
'   '
'   '
'USAGE CONFLICTS WITH THAT ON GROUP LEVEL'
'RECORD NAME PREVIOUSLY DECLARED'
'"FILLER" ENTRY NOT ALLOWED ON 77 LEVEL'
'"VALUE" CLAUSE NOT ALLOWED IN "FILE" OR "LINKAGE" SECTION'
'FILLER BYTE ADDED'
'MISSING "PICTURE" CLAUSE ON 77 LEVEL ITEM'
'LEVEL NUMBER MUST BE LOWER THAN "OCCURS DEPENDING" LEVEL NUMBER'
'"OCCURS" CLAUSE ILLEGAL WITH 77 LEVEL'
'MAXIMUM NUMBER OF OCCURRENCES DOES NOT EXCEED MINIMUM NUMBER'
'MULTIPLE "OCCURS DEPENDING" CLAUSES NOT ALLOWED WITHIN RECORD'
'SUBSCRIPTED ITEMS CANNOT HAVE AN INITIAL VALUE CLAUSE'
'   '
'   '
'   '
'   '
'MULTIPLE "PICTURE" CLAUSES'
'MULTIPLE "USAGE" CLAUSES'
'MULTIPLE "SIGN" CLAUSES'
'MULTIPLE "SYNC" CLAUSES'
'MULTIPLE "JUSTIFY" CLAUSES'
'MULTIPLE "BLANK WHEN ZERO" CLAUSES'
'MULTIPLE "INITIAL VALUE" CLAUSES'
'MULTIPLE "OCCURS" CLAUSES'
'ILLEGAL "SIGN" CLAUSE, TYPE MUST BE NUMERIC'
```

```
'ILLEGAL "SIGN" CLAUSE, USAGE MUST BE DISPLAY'
'ILLEGAL "SYNC" CLAUSE, MUST BE ELEMENTARY ITEM'
'ILLEGAL "JUSTIFIED" CLAUSE, MUST BE ELEMENTARY ITEM'
'ILLEGAL "BLANK WHEN ZERO" CLAUSE, MUST BE ELEMENTARY ITEM'
'ILLEGAL "JUSTIFY" CLAUSE, MUST BE ALPHABETIC OR ALPHANUMERIC TYPE'
'ILLEGAL "BLANK WHEN ZERO" CLAUSE, FLOATING "*" NOT ALLOWED'
'ILLEGAL "BLANK WHEN ZERO" CLAUSE, MUST BE NUMERIC OR NUMERIC EDITED'
'"PROGRAM COLLATING SEQUENCE" NAME NOT DEFINED'
'SIGN ALREADY DECLARED ON GROUP LEVEL'
'INVALID USE OF OVER-PUNCH SIGN WITH "CODE SET" CLAUSE'
'COMPUTATIONAL USAGES NOT ALLOWED WITH "CODE SET" CLAUSE'
'ITEM MUST BE AN INTEGER OR AN ELMENTARY ITEM'
'ITEM MUST BE A POSITIVE INTEGER'
'IDENTIFIER IS NOT UNIQUELY QUALIFIED'
'ILLEGAL DATA TYPE. DATA NAME REQUIRED'
'ILLEGAL USAGE WITH "BLANK WHEN ZERO" CLAUSE'
'SORT FILES MUST HAVE A SEQUENTIAL FILE ORGANIZATION'
'ILLEGAL FILE ACCESS FOR A SORT FILE'
'SWITCH NAME MUST BE A SINGLE CHARACTER QUOTED ALPHABETIC STRING'
'ILLEGAL LABEL RECORD TYPE'
'ILLEGAL RECORDING MODE. PRINTER FILES MUST BE "DATA SENSITIVE" '
'MISSING "I-O-CONTROL" STATEMENT'
'INVALID LITERAL ON "INITIAL VALUE" CLAUSE.  STRING LITERAL REQUIRED'
'INVALID LITERAL ON "INITIAL VALUE" CLAUSE.  NUMERIC LITERAL REQUIRED'
'INVALID NONNUMERIC FIGURATIVE CONSTANT ON "INITIAL VALUE" CLAUSE.'
'ILLEGAL CHARACTER IN NUMERIC LITERAL IN "VALUE" CLAUSE'
'ILLEGAL NUMERIC LITERAL IN "VALUE" CLAUSE'
'OVERFLOW ON LOADING NUMERIC LITERAL IN "VALUE IS" CLAUSE'
'LITERAL AFTER "THROUGH" MUST BE ONE CHARACTER IN LENGTH'
'LITERAL PRECEDING "ALSO" MUST BE ONE CHARACTER IN LENGTH'
'LITERAL AFTER "ALSO" MUST BE ONE CHARACTER IN LENGTH'
'FD'S OF FILES IN SAME AREA CLAUSE NOT DECLARED TOGETHER'
'FILE APPEARS IN MORE THAN ONE "SAME AREA" CLAUSE'
'BINARY (COMP) ITEMS OF GREATER THAN 16 DIGITS HANDLED AS 9(16)'
'A CAM FILE CANNOT BE DECLARED "OPTIONAL"'
'"ASSIGN INDEX" INVALID FOR CAM FILES'
'DUPLICATE "ACTIVE LINES" CLAUSE IN "SELECT" STATEMENT'
'DUPLICATE "RESERVE INPUT AREAS" CLAUSE IN "SELECT" STATEMENT'
'DUPLICATE "RESERVE OUTPUT AREAS" CLAUSE IN "SELECT" STATEMENT'
'DUPLICATE "INPUT RECORD CONTAINS" CLAUSE IN FD'
'DUPLICATE "OUTPUT RECORD CONTAINS" CLAUSE IN FD'
'NON DISPLAY USAGE ONLY ALLOWED ON NUMERIC ITEMS'
'MANTISSA IN EXTERNAL FLOATING POINT LITERAL CANNOT EXCEED 16 DIGITS'
'INVALID EXPONENT IN EXTERNAL FLOATING POINT LITERAL'
'NUMERIC LITERAL CANNOT EXCEED 18 DIGITS'
'MISPLACED "ENVIRONMENT DIVISION" STATEMENT'
'MISPLACED "CONFIGURATION SECTION" STATEMENT'
'MISPLACED "INPUT-OUTPUT SECTION" STATEMENT'
'MISPLACED 'DATA DIVISION" STATEMENT'
'MISSING "DATA DIVISION" STATEMENT'
'MISPLACED "FILE SECTION" STATEMENT'
'MISPLACED "WORKING-STORAGE SECTION" STATEMENT'
'MISPLACED "LINKAGE SECTION" STATEMENT'
'MISSING "CONFIGURATION SECTION" STATEMENT'
'MISSING "INPUT-OUTPUT SECTION" STATEMENT'
'MISSING "FILE SECTION" STATEMENT'
'MISSING "WORKING-STORAGE SECTION" STATEMENT'
'MISSING "LINKAGE SECTION" STATEMENT'
'MISSING "FILE-CONTROL" STATEMENT'
'MISSING "I-O-CONTROL" STATEMENT'
```

```
'UNRECOGNIZED ENVIRONMENT OR DATA DIVISION CONSTRUCTION'
'DUPLICATE "CAM STATUS" CLAUSE IN FD'
'DUPLICATE "LENGTH" CLAUSE IN FD'
'DUPLICATE "XPND" CLAUSE IN FD'
'DUPLICATE "XTRN" CLAUSE IN FD'
'DUPLICATE "XMOD" CLAUSE IN FD'
'DUPLICATE "XECS" CLAUSE IN FD'
'DUPLICATE "XNMT" CLAUSE IN FD'
'DUPLICATE "TIME OUT" CLAUSE IN FD'
'"SAME RECORD AREA" STATEMENT CONTAINS A NAME THAT IS NOT A FILE'
'OVER-PUNCH SIGN NOT ALLOWED FOR FIELD OF "CODE SET" CLAUSE'
'COMPUTATIONAL USAGES NOT ALLOWED FOR FIELD OF CODE SET CLAUSE'
'"DIVISION" DOES NOT FOLLOW "ENVIRONMENT"'
'MISSING PERIOD AFTER "ENVIRONMENT DIVISION"'
'"SECTION" DOES NOT FOLLOW "CONFIGURATION"'
'PERIOD DOES NOT FOLLOW "CONFIGURATION SECTION"'
'"SECTION" DOES NOT FOLLOW "INPUT-OUTPUT"'
'MISSING PERIOD AFTER "INPUT-OUTPUT SECTION"'
'"DIVISION" DOES NOT FOLLOW "DATA"'
'PERIOD DOES NOT FOLLOW "DATA DIVISION"'
'"SECTION" DOES NOT FOLLOW "FILE" IN "FILE SECTION" HEADER'
'MISSING PERIOD FOLLOWING "FILE SECTION"'
'"SECTION" DOES NOT FOLLOW "WORKING-STORAGE" IN SECTION HEADER'
'MISSING PERIOD AFTER "WORKING-STORAGE SECTION".'
'"SECTION" DOES NOT FOLLOW "LINKAGE" IN SECTION HEADER'
'MISSING PERIOD AFTER "LINKAGE SECTION"'
'"DIVISION" DOES NOT FOLLOW "PROCEDURE"'
'MISSING PERIOD FOLLOWING "SOURCE-COMPUTER"'
'INVALID COMPUTER-NAME IN "SOURCE-COMPUTER" STATEMENT'
'MISSING PERIOD FOLLOWING SOURCE-COMPUTER NAME'
'MISSING PERIOD FOLLOWING "OBJECT COMPUTER"'
'MISSING COMPUTER-NAME IN "OBJECT COMPUTER" PARAGRAPH'
'INVALID CONSTRUCTION FOLLOWING OBJECT-COMPUTER NAME'
'INTEGER DOES NOT FOLLOW "MEMORY SIZE"'
'"SEQUENCE" DOES NOT FOLLOW "COLLATING"'
'MISSING ALPHABET NAME IN'
'MISSING INTEGER IN "SEGMENT-LIMIT" CLAUSE'
'"IS" DOES NOT FOLLOW "CURRENCY SIGN"'
'MISSING QUOTED CHARACTER IN "CURRENCY SIGN" CLAUSE'
'"COMMA" DOES NOT FOLLOW "DECIMAL-POINT"'
'CHANNEL NUMBER DOES NOT FOLLOW "CHANNEL" IN "SPECIAL-NAMES"'
'MISSING CHANNEL-NAME FOLLOWING A CHANNEL-NUMBER IN "SPECIAL-NAMES"'
'QUOTED CHARACTER DOES NOT FOLLOW "SWITCH"'
'MISSING MNEMONIC-NAME IN "SWITCH" DECLARATION'
'MISSING CONDITION-NAME IN "ON STATUS" CLAUSE'
'MISSING CONDITION-NAME IN "OFF STATUS" CLAUSE'
'MISSING "IS" IN "ON STATUS" OR "OFF STATUS" CLAUSE'
'MISSING MNEMONIC-NAME'
'INVALID ALPHABET-NAME DEFINITION'
'LITERAL DOES NOT FOLLOW "ALSO" IN DEFINITION OF ALPHABET-NAME'
'LITERAL DOES NOT FOLLOW "THROUGH" IN DEFINITION OF ALPHABET-NAME'
'UNRECOGNIZED CONSTRUCT IN "I-O-CONROL" PARAGRAPH'
'MISSING PERIOD AFTER "FILE-CONTROL"'
'MISSING PERIOD AFTER "I-O-CONTROL"'
'MISSING FILE-NAME IN "SELECT" STATEMENT'
'"ASSIGN" DOES NOT FOLLOW FILE-NAME IN SELECT STATEMENT'
'INVALID DEVICE-NAME IN "ASSIGN" CLAUSE OF THE "SELECT" STATEMENT'
'INVALID DEVICE-NAME IN "ASSIGN DATA" CLAUSE OF THE "SELECT" STATEMENT'
'"MANAGEMENT" DOES NOT FOLLOW "SPACE" IN THE "SELECT" STATEMENT'
'"MERIT" DOES NOT FOLLOW "ROOT" IN "SELECT" STATEMENT'
```

```
'MISSING INTEGER IN "ROOT MERIT" CLAUSE OF "SELECT" STATEMENT'
'INVALID FILE-CONTROL-ENTRY CLAUSE'
'AN INTEGER DOES NOT FOLLOW "MERIT" IN "ASSIGN" CLAUSE'
'"SIZE" DOES NOT FOLLOW "VOLUME" IN "ASSIGN" CLAUSE'
'MISSING INTEGER AFTER "VOLUME SIZE IS" IN "ASSIGN" CLAUSE'
'AN INTEGER DOES NOT FOLLOW "RESERVE" IN "SELECT" STATEMENT'
'INVALID FILE ORGANIZATION IN "SELECT" STATEMENT'
'INVALID "ACCESS MODE" IN SELECT STATEMENT'
'MISSING DATA-NAME IN "RELATIVE KEY" CLAUSE'
'"RECORD" DOES NOT FOLLOW "ALTERNATE" IN "SELECT" STATEMENT'
'MISSING DATA-NAME IN "ALTERNATE KEY" CLAUSE'
'"STATUS" DOES NOT FOLLOW "FILE" IN "SELECT" STATEMENT'
'MISSING DATA NAME IN "FILE STATUS" CLAUSE'
'MISSING DATA-NAME IN "INFOS STATUS" CLAUSE'
'MISSING DATA-NAME IN "RECORD KEY" CLAUSE'
'"SUBINDEX" DOES NOT FOLLOW "ALLOW" IN "ALLOW SUBINDEX" CLAUSE'
'MISSING INTEGER IN "LEVELS" PHRASE OF "ALLOW SUBINDEX" CLAUSE'
'"COMPRESSION" DOES NOT FOLLOW "KEY" IN "KEY COMPRESSION" CLAUSE'
'INVALID PARITY TYPE IN "PARITY IS" CLAUSE'
'"LENGTH" DOES NOT FOLLOW "KEY" IN "KEY LENGTH" CLAUSE'
'"DUPLICATES" DOES NOT FOLLOW "WITH" IN "KEY LENGTH" CLAUSE'
'DATA-NAME DOES NOT FOLLOW "OCCURRENCE IS" IN "KEY LENGTH" CLAUSE'
'MISSING DATA-NAME IN "RERUN" CLAUSE'
'MISSING DATA-NAME IN "RERUN" CLAUSE'
'MISSING "REEL" OR "UNIT" IN "RERUN" CLAUSE'
'MISSING "RECORDS" IN "RERUN" CLAUSE'
'MISSING FILE-NAME IN "SAME RECORD AREA" CLAUSE'
'MISSING FILE-NAME IN "SAME AREA" CLAUSE'
'"FILE" DOES NOT FOLLOW "MULTIPLE" IN "MULTIPLE FILE TAPE" CLAUSE'
'MISSING FILE-NAME IN "MULTIPLE FILE TAPE" CLAUSE'
'AN INTEGER DOES NOT FOLLOW "POSITION" IN "MULTIPLE FILE TAPE" CLAUSE'
'MISSING FILE-NAME AFTER "FD"'
'INVALID CONSTRUCT IN FILE DESCRIPTION ENTRY'
'MISSING RECORD DESCRIPTION ENTRY FOLLOWING FILE DESCRIPTION'
'"BLOCK" OR "NODE" DOES NOT FOLLOW "INDEX" IN "FD" STATEMENT'
'"BLOCK" OR "RECORD" DOES NOT FOLLOW "DATA" IN "FD" STATEMENT'
'FILE-NAME DOES NOT FOLLOW "SD"'
'MISSING PERIOD IN SORT DESCRIPTION STATEMENT'
'RECORD DESCRIPTION ENTRY DOES NOT FOLLOW SORT DESCRIPTION STATEMENT'
'RECORD DOES NOT FOLLOW "DATA" IN "SD" STATEMENT'
'MISSING INTEGER IN "BLOCK CONTAINS" CLAUSE'
'MISSING "RECORDS" OR "CHARACTERS" IN "BLOCK CONTAINS" CLAUSE'
'MISSING INTEGER IN "INDEX BLOCK CONTAINS" CLAUSE'
'MISSING INTEGER IN "NODE SIZE" CLAUSE'
'MISSING INTEGER IN "RECORD CONTAINS" CLAUSE'
'INVALID LABEL RECORD TYPE IN "LABEL RECORD" CLAUSE'
'MISSING LITERAL OR DATA-NAME IN "OWNER IS" CLAUSE'
'MISSING LITERAL OR DATA-NAME IN "EXPIRATION DATE" CLAUSE'
'MISSING LITERAL OR DATA-NAME IN "SEQUENCE NUMBER" CLAUSE'
'MISSING LITERAL OR DATA-NAME IN "GENERATION NUMBER" CLAUSE'
'MISSING LITERAL OR DATA-NAME IN "ACCESABILITY IS" CLAUSE'
'MISSING LITERAL OR DATA-NAME IN "OFFSET IS" CLAUSE'
'"STATUS" DOES NOT FOLLOW "VOLUME" IN "VOLUME STATUS CLAUSE'
'MISSING DATA-NAME IN "VOLUME STATUS" CLAUSE'
'MISSING DATA-NAME IN "USER VOLUME" CLAUSE'
'MISSING DATA-NAME IN "HEADER LABEL" CLAUSE'
'INVALID LABEL SPECIFIER FOLLOWING "USER" IN "VALUE OF" CLAUSE'
'MISSING DATA-NAME IN "TRAILER LABEL" CLAUSE'
'MISSING RECORD-NAME IN "DATA RECORDS" CLAUSE'
'MISSING DATA-NAME OR LITERAL IN "MERIT" CLAUSE'
```

```
'MISSING DATA-NAME OR LITERAL SPECIFYING PAGE SIZE IN "LINAGE" CLAUSE'
'"FOOTING" DOES NOT FOLLOW "WITH" IN LINAGE CLAUSE'
'MISSING DATA-NAME OR LITERAL SPECIFYING "FOOTING" IN "LINAGE" CLAUSE'
'MISSING DATA-NAME OR LITERAL SPECIFYING "TOP" MARGIN IN "LINAGE" CLAUSE'
'MISSING DATA-NAME OR LITERAL SPECIFYING "BOTTOM" MARGIN IN "LINAGE" CLAUSE'
'"TOP" OR "BOTTOM" DOES NOT FOLLOW "LINES AT" IN "LINAGE CLAUSE"'
'MISSING ALPHABET-NAME IN "CODE-SET" CLAUSE'
'MISSING DATA-NAME FOLLOWING "FIELD IS" IN "CODE-SET" CLAUSE'
'INVALID "RECORDING MODE" SPECIFICATION'
'MISSING "DATA-SENSITIVE" DELIMETER IN "RECORDING MODE" CLAUSE'
'MISSING DATA-NAME IN "RECORD LENGTH" CLUASE'
'MISSING DATA-NAME IN "FEEDBACK" CLAUSE'
'MISSING DATA-NAME IN "PARTIAL RECORD" CLAUSE'
'MISSING DATA-NAME OR LITERAL IN "PAD CHARACTER" CLAUSE'
'MISSING DATA-NAME FOLLOWING 77 LEVEL'
'MISSING DATA-NAME FOLLOWING 01 LEVEL'
'MISSING DATA-NAME FOLLOWING LEVEL NUMBER'
'MISSING DATA-NAME FOLLOWING "REDEFINES"'
'MISSING PERIOD FOLLOWING DATA DESCRIPTION ENTRY'
'MISSING DATA-NAME FOLLOWING LEVEL NUMBER 66'
'MISSING DATA-NAME FOLLOWING "RENAMES"'
'MISSING DATA-NAME FOLLOWING "THROUGH" IN "RENAMES" DESCRIPTION'
'MISSING PERIOD FOLLOWING LEVEL 66 DATA DESCRIPTION ENTRY'
'MISSING CONDITION-NAME FOLLOWING LEVEL NUMBER 88'
'MISSING "VALUE IS" CLAUSE IN LEVEL 88 DATA DESCRIPTION ENTRY'
'MISSING LITERAL FOLLOWING "VALUE IS" IN LEVEL 88 DATA DESCRIPTION ENTRY'
'MISSING LITERAL FOLLOWING "THROUGH" IN LEVEL 88 DATA DESCRIPTION ENTRY'
'MISSING PERIOD FOLLOWING LEVEL 88 DATA DESCRIPTION ENTRY'
'INVALID DATA USAGE SPECIFIED IN USAGE" CLAUSE'
'"LEADING" OR "TRAILING" NOT FOUND IN "SIGN" CLAUSE'
'INTEGER DOES NOT FOLLOW "OCCURS" IN "OCCURS" CLAUSE'
'MISSING UPPER VALUE OF OCCURS RANGE IN "OCCURS" CLAUSE'
'MISSING "DEPENDING ON" IN "OCCURS" CLAUSE'
'MISSING "DEPENDING ON" DATA-NAME IN "OCCURS" CLAUSE'
'MISSING DATA-NAME FOLLOWING "KEY IS" IN "OCCURS" CLAUSE'
'MISSING DATA-NAME FOLLOWING "INDEXED BY"'
'MISSING "ZERO" IN "BLANK WHEN ZERO" CLAUSE'
'MISSING LITERAL IN "VALUE IS" CLAUSE'
'MISSING GROUP LEVEL DATA-NAME IN QUALIFICATION OF DATA-NAME'
```

```
'*UNDEFINED PROCEDURE NAME*'
'ATTEMPT TO QUALIFY SECTION NAME'
'PARAGRAPH NOT IN INDICATED SECTION'
'MULTIPLE DEFINITION OF SECTION NAME'
'MULTIPLE DEFINITION OF PARAGRAPH NAME'
'PARAGRAPH OR SECTION NAME CONFLICTS WITH PREVIOUS USAGE OF NAME'
'INSUFFICIENT QUALIFICATION OF NAME'
'QUALIFIED NAME DOES NOT EXIST'
'ELEMENTARY ITEM NAME USED IN CORRESPONDING'
'IMPROPER USE OF "SD" FILE'
'INVALID CLAUSE IN I/O STATEMENT, FILE ORGANIZATION MUST BE SEQUENTIAL'
'INVALID CLAUSE IN I/O STATEMENT, FILE ORGANIZATION MUST BE INDEXED'
'INVALID CLAUSE IN I/O STATEMENT, FILE ORGANIZATION MUST BE RELATIVE'
'INVALID CLAUSE IN I/O STATEMENT, FILE ORGANIZATION CANNOT BE SEQUENTIAL'
' ERROR MESSAGE NOT BEING USED COMPILER BUG '
'RECORD MUST BE ASSOCIATED WITH A SORT FILE'
'FILE MUST BE DECLARED WITH AN "SD" STATEMENT'
'INVALID RECORD KEY NAME'
'INVALID NAME, RECORD NAME REQUIRED'
'INVALID RECORD KEY, KEY NAME NOT ASSOCIATED WITH THE FILE'
'FILE HAS NO RECORD KEYS'
'SORT KEY NOT IN FILES RECORD AREA'
'ILLEGAL USE OF CONDITION NAME'
'ILLEGAL USE OF DATA NAME, RECORD NAME REQUIRED'
'TOO MANY LEVELS OF QUALIFICATION'
'FILE DOES NOT HAVE A LINAGE COUNTER'
'NO LINAGE COUNTER DECLARED IN PROGRAM'
'INCOMPLETE "GO TO" STATEMENT IN UNALTERABLE PARAGRAPH'
'UNDECLARED DATA NAME'
'UNDECLARED FILE NAME'
'UNDECLARED ALPHABET NAME'
'ILLEGAL ABBREVIATED COMBINED RELATIONAL CONDITION'
'ILLEGAL ABBREVIATED COMBINED RELATIONAL CONDITION'
'ILLEGAL ABBREVIATED COMBINED RELATIONAL CONDITION'
'"ADVANCING" CLAUSE ILLEGAL ON NON "PRINTER" FILES'
'NO PRIME RECORD KEY SPECIFIED ON FILE DESCRIPTION ENTRY'
'MISSING 'INDEX BY' CLAUSE IN TABLE DECLARATION'
'<KEY SERIES> ILLEGAL WITH SEQUENTIAL ACCESS FILES'
'"SEARCH" DATA-NAME DOES NOT HAVE AN "OCCURS" CLAUSE'
'A "CAM" TYPE FILE MUST BE USED IN "CALL" OF A "CAM" ROUTINE'
'COMPILER STACK OVERFLOW'
'TOO MANY LEVELS OF SUBSCRIPTING'
'ILLEGAL RELATIVE POSITION WITH KEYED ACCESS'
'LITERALS NOT ALLOWED IN CAM CALL'
'ILLEGAL RELATIVE POSITION WITH RANDOM ACCESS MODE'
'DATA RECORD NAME NOT ASSOCIATED WITH A FILE'
'UNRECOGNIZED TOKEN, TOKEN IGNORED'
'UNDECLARED DATA NAME'
'UNRECOGNIZED TOKEN, TOKEN IGNORED'
'PERIOD REQUIRED AFTER SECTION & PARAGRAPH HEADERS'
'MISSING DECLARATIVE STATEMENT'
'PERIOD REQUIRED AFTER SECTION & PARAGRAPH HEADERS'
'"DECLARATIVES" DOES NOT FOLLOW "END"'
'MISSING PROCEDURE NAME'
'SECTION HEADER EXPECTED BUT NOT FOUND'
'USE MUST FOLLOW SECTION HEADER IN DECLARATIVES SECTION'
'MISSING DATA NAME IN "ACCEPT" STATEMENT'
'INVALID DEVICE NAME FOR "ACCEPT"'
'"COUNT" DOES NOT FOLLOW "MESSAGE"'
'MISSING DATA NAME IN "ADD" STATEMENT'
```

```
'MISSING DATA NAME IN "ADD" STATEMENT'
'MISSING GROUP LEVEL DATA NAME IN "ADD CORRESPONDING" STATEMENT'
'MISSING "TO" IN "ADD CORRESPONDING" STATEMENT'
'MISSING GROUP LEVEL DATA NAME IN "ADD CORRESPONDING" STATEMENT'
'MISSING ADVANCING SPECIFIER TYPE'
'MISSING PROCEDURE NAME IN "ALTER" STATEMENT'
'MISSING "TO" IN "ALTER" STATEMENT'
'MISSING "TO" IN "ALTER" STATEMENT'
'MISSING PROCEDURE NAME IN "ALTER" STATEMENT'
'"AT END" IS NOT FOLLOWED BY A STATEMENT'
'"END" DOES NOT FOLLOW "AT"'
'"AT END-OF-PAGE" NOT FOLLOWED BY A STATEMENT'
'"END-OF-PAGE" DOES NOT FOLLOW "AT"'
'MISSING DATA NAME IN "CALL" STATEMENT'
'MISSING DATA NAME IN "CALL" STATEMENT'
'MISSING DATA NAME OR LITERAL IN "CANCEL" STATEMENT'
'MISSING FILE NAME IN "CLOSE" STATEMENT'
'"NO REWIND" DOES NOT FOLLOW "WITH"'
'"REWIND" DOES NOT FOLLOW "NO" IN "CLOSE" STATEMENT'
'"REMOVAL" DOES NOT FOLLOW "FOR"'
'"NO REWIND" OR "LOCK" DOES NOT FOLLOW "WITH"'
'"REWIND" DOES NOT FOLLOW "NO"'
'"SEQUENCE" DOES NOT FOLLOW "COLLATING"'
'INVALID ALPHABET NAME IN "COLLATING SEQUENCE" CLAUSE'
'MISSING RECEIVING ITEM DATA NAME IN "COMPUTE" STATEMENT'
'MISSING ARITHMETIC OPERATOR IN "COMPUTE" STATEMENT'
'MISSING ARITHMETIC OPERATOR IN "COMPUTE" STATEMENT'
'INVALID ARITHMETIC EXPRESSION'
'INVALID ARITHMETIC EXPRESSION'
'INVALID EXPRESSION CONSTRUCTION FOLLOWING "NOT"'
'INVALID ARITHMETIC EXPRESSION'
'MISSING RELATIONAL OPERATOR, SIGN OR CLASS CONDITIONAL OPERATOR'
'INVALID EXPRESSION FOLLOWING RELATIONAL OPERATOR'
'INVALID EXPRESSION FOLLOWS PLUS OR MINUS SIGN'
'INVALID EXPRESSION FOLLOWS PLUS OR MINUS SIGN'
'INVALID EXPRESSION FOLLOWS ARITHMETIC OPERATOR'
'INVALID EXPRESSION FOLLOWS'
'INVALID EXPRESSION FOLLOWING "("'
'MISSING ")"'
'"SUBINDEX" DOES NOT FOLLOW "DEFINE"'
'MISSING FILE NAME IN "DEFINE" STATEMENT'
'MISSING "INDEX NODE" CLAUSE IN "DEFINE" STATEMENT'
'"NODE" DOES NOT FOLLOW "INDEX" IN "DEFINE" STATEMENT.'
'MISSING DATA NAME OR LITERAL IN "NODE SIZE" CLAUSE'
'MISSING DATA NAME IN "PARTIAL RECORD" CLAUSE'
'MISSING "KEY LENGTH" CLAUSE IN "DEFINE" STATEMENT'
'"LENGTH" DOES NOT FOLLOW "KEY"'
'MISSING DATA NAME OR LITERAL IN "MAXIMUM KEY LENGTH" CLAUSE'
'"KEY" DOES NOT FOLLOW "MAXIMUM"'
'"SUBINDEX" OR "DUPLICATES" DOES NOT FOLLOW "ALLOW"'
'"COMPRESSION" DOES NOT FOLLOW "KEY"'
'MISSING FILE NAME AFTER "DELETE"'
'MISSING DATA NAME OR LITERAL IN "DISPLAY" STATEMENT'
'MISSING DEVICE NAME IN "DISPLAY" STATEMENT'
'"ADVANCING" DOES NOT FOLLOW "NO"'
'MISSING DATA NAME OR LITERAL FOLLOWING "DIVIDE"'
'"BY" OR "INTO" DOES NOT FOLLOW FIRST OPERAND IN DIVIDE STATEMENT'
'DATA NAME OR LITERAL DIVIDEND NOT FOUND IN "DIVIDE" STATEMENT'
'DATA NAME OR LITERAL DIVISOR NOT FOUND IN "DIVIDE" STATEMENT'
'MISSING RECEIVING FIELD IN "DIVIDE" STATEMENT'
```

```
'MISSING RECEIVING FIELD DATA NAME IN "DIVIDE" STATEMENT'
'MISSING DATA NAME IN "REMAINDER" CLAUSE'
'INVALID EXPRESSION FOLLOWS PLUS OR MINUS SIGN'
'INVALID EXPRESSION FOLLOWS PLUS OR MINUS SIGN'
'MISSING FILE NAME IN "EXPUNGE" STATEMENT'
'MISSING FILE NAME IN "EXPUNGE" STATEMENT'
'MISSING DATA NAME FOLLOWING "FROM"'
'"LINE" DOES NOT FOLLOW "FROM" IN "WRITE" OF MTCP FILE'
'MISSING LINE-NAME IN "FROM LINE" CLAUSE'
'NO FILE NAME FOLLOWS "GIVING"'
'"DEPENDING ON" CLAUSE EXPECTED BUT NOT FOUND IN "GO TO" STATEMENT'
'MISSING DATA NAME IN "DEPENDING ON" CLAUSE'
'INVALID QUALIFICATION OF "LINAGE COUNTER", FILE NAME EXPECTED'
'INVALID CONDITIONAL EXPRESSION FOLLOWING "IF"'
'MISSING STATEMENT FOLLOWING CONDITIONAL EXPRESSION'
'MISSING STATEMENT FOLLOWING "ELSE"'
'"PROCEDURE" DOES NOT FOLLOW "INPUT"'
'MISSING SECTION NAME IN "INPUT PROCEDURE" CLAUSE'
'MISSING SECTION NAME IN "INPUT PROCEDURE" CLAUSE'
'MISSING DATA NAME AFTER "INSPECT"'
'INVALID TALLYING CLAUSE IN "INSPECT" STATEMENT'
'"TALLYING" OR "REPLACING" DOES NOT FOLLOW "INSPECT"'
'INVALID "REPLACING" CLAUSE IN "INSPECT" STATEMENT'
'"FOR" NOT FOUND AFTER DATA NAME IN "TALLY" CLAUSE'
'"ALL", "LEADING", OR "CHARACTERS" NOT FOUND IN "TALLY" CLAUSE'
'MISSING DATA NAME OR LITERAL IN "REPLACING" CLAUSE'
'"BY" NOT FOUND IN "REPLACING" CLAUSE'
'"BY" NOT FOUND IN "REPLACING" CLAUSE'
'MISSING DATA NAME OR LITERAL FOLLOWING "BY" IN "INSPECT" STATEMENT'
'MISSING DATA NAME OR LITERAL FOLLOWING "INITIAL" IN "INSPECT" STATEMENT'
'MISSING DATA NAME AFTER "INTO"'
'"LINE" DOES NOT FOLLOW "INTO" IN "WRITE" STATEMENT'
'MISSING DATA NAME AFTER "INTO LINE" CLAUSE'
'ILLEGAL STATEMENT FOLLOWING "INVALID KEY" CLAUSE'
'MISSING DATA NAME AFTER "KEY"'
'MISSING DATA NAME AFTER "KEY IS"'
'"SUBINDEX" DOES NOT FOLLOW "LINK" '
'MISSING FILE NAME AFTER "LINK SUBINDEX"'
'"SOURCE" DOES NOT FOLLOW THE FILE NAME IN THE "LINK" STATEMENT'
'"DESTINATION" MISSING IN "LINK" STATEMENT'
'MISSING FILE NAME AFTER "MERGE"'
'MISSING LIST OF "MERGE" KEYS'
'MISSING "USING" CLAUSE IN MERGE STATEMENT'
'NO OUTPUT PROCEDURE NAME OR OUTPUT FILES SPECIFIED IN "MERGE" STATEMENT'
'MISSING SENDING FIELD DATA NAME OR LITERAL IN "MOVE" STATEMENT'
'"TO" DOES NOT FOLLOW THE SENDING FIELD'
'MISSING RECEIVING FIELD DATA NAME OR LITERAL IN "MOVE" STATEMENT'
'MISSING SENDING FIELD GROUP LEVEL NAME IN "MOVE" STATEMENT'
'"TO" DOES NOT FOLLOW THE SENDING FIELD'
'MISSING RECEIVING FIELD GROUP LEVEL NAME IN "MOVE" STATEMENT'
'MISSING DATA NAME OR LITERAL IN "MULTIPLY" STATEMENT'
'MISSING "BY" IN "MULTIPLY" STATEMENT'
'MISSING DATA NAME OR LITERAL AFTER "BY" IN "MULTIPLY" STATEMENT'
'"GIVING" NOT FOUND AFTER LITERAL IN "MULTIPLY" STATEMENT'
'MISSING RECEIVING FIELD DATA NAME IN "MULTIPLY" STATEMENT'
'"SENTENCE" DOES NOT FOLLOW "NEXT" '
'"DATA" DOES NOT FOLLOW "NO"'
'ILLEGAL STATEMENT FOLLOWING "NO DATA" CLAUSE'
'INVALID OPEN TYPE IN "OPEN" STATEMENT'
'MISSING FILE NAME FOLLOWING "INPUT" IN "OPEN" STATEMENT'
```

```
'"NO" DOES NOT FOLLOW "WITH" IN "OPEN" STATEMENT'
'"REWIND" DOES NOT FOLLOW "NO" IN "OPEN" STATEMENT'
'MISSING FILE NAME IN "OPEN OUTPUT" STATEMENT'
'"REWIND" DOES NOT FOLLOW "WITH" IN "OPEN" STATEMENT'
'"NO" DOES NOT FOLLOW "WITH" IN "OPEN" STATEMENT'
'MISSING FILE-NAME IN "OPEN I/O" STATEMENT'
'"VERIFY" DOES NOT FOLLOW "WITH" IN'
'MISSING FILE-NAME FOLLOWING "EXCLUDE" IN "OPEN" STATEMENT'
'"PROCEDURE" DOES NOT FOLLOW "OUTPUT"'
'MISSING SECTION NAME IN "OUTPUT PROCEDURE" CLAUSE'
'MISSING SECTION NAME AFTER "THROUGH" IN "OUTPUT PROCEDURE" CLAUSE'
'"OVERFLOW" DOES NOT FOLLOW "ON"'
'INVALID STATEMENT FOLLOWS "ON OVERFLOW" CLAUSE'
'PROCEDURE NAME MISSING IN "PERFORM" STATEMENT'
'MISSING PROCEDURE NAME AFTER "THROUGH" IN "PERFORM" STATEMENT'
'INVALID CONDITIONAL EXPRESSION FOLLOWING "UNTIL" IN "PERFORM" STATEMENT'
'MISSING DATA NAME AFTER VARYING IN "PERFORM" STATEMENT'
'"TIMES" DOES NOT FOLLOW DATA NAME OR LITERAL IN "PERFORM" STATEMENT'
'DATA NAME DOES NOT FOLLOW "AFTER" IN THE "PERFORM" STATEMENT'
'"FROM" DOES NOT FOLLOW THE DATA NAME IN THE "PERFORM" STATEMENT'
'DATA NAME OR LITERAL DOES NOT FOLLOW "FROM" IN THE "PERFORM" STATEMENT'
'"BY" DOES NOT FOLLOW THE DATA NAME OR LITERAL IN THE "PERFORM" STATEMENT'
'DATA NAME OR LITERAL DOES NOT FOLLOW "BY" IN THE "PERFORM" STATEMENT'
'"UNTIL" DOES NOT FOLLOW DATA NAME OR LITERAL IN THE "PERFORM" STATEMENT'
'INVALID CONDITIONAL EXPRESSION IN THE "PERFORM" STATEMENT'
'INVALID EXPRESSION FOLLOWS LEFT PAREN'
'MISSING RIGHT PAREN'
'MISSING DATA NAME QUALIFIER'
'MISSING SECTION NAME AS QUALIFIER'
'MISSING FILE NAME IN "READ" STATEMENT'
'MISSING RELATIONAL OPERATOR'
'MISSING RECORD NAME IN "RELEASE" STATEMENT'
'MISSING DATA NAME AFTER "FROM" IN "RELEASE" STATEMENT'
'MISSING FILE NAME IN "RETRIEVE" STATEMENT'
'"KEY" DOES NOT FOLLOW "HIGH" IN "RETRIEVE" STATEMENT'
'"SUBINDEX" DOES NOT FOLLOW "KEY" IN "RETRIEVE" STATEMENT'
'MISSING "INTO" IN "RETRIEVE" STATEMENT'
'MISSING DATA NAME AFTER "INTO" IN "RETRIEVE" STATEMENT'
'MISSING FILE NAME IN "RETURN" STATEMENT'
'MANDATORY "AT END" CLAUSE MISSING IN "RETURN" STATEMENT'
'MISSING RECORD NAME IN "REWRITE" STATEMENT'
'MISSING "SEARCH" FIELD DATA NAME'
'MISSING "SEARCH" FIELD DATA-NAME'
'MISSING DATA NAME AFTER "VARYING" IN "SEARCH" STATEMENT'
'"END" DOES NOT FOLLOW "AT" IN "SEARCH" STATEMENT'
'INVALID CONDITIONAL EXPRESSION AFTER "WHEN" IN "SEARCH" STATEMENT'
'MISSING "WHEN" CLAUSE IN "SEARCH" STATEMENT'
'MISSING FILE NAME IN "SEEK" STATEMENT'
'MISSING DATA NAME IN "SET" STATEMENT'
'INVALID RESULT VALUE SPECIFICATION IN "SET" STATEMENT'
'"BY" DOES NOT FOLLOW "UP" OR "DOWN"'
'MISSING DATA NAME OR LITERAL AFTER "BY" IN "SET" STATEMENT'
'MISSING "SIZE" AFTER "ON" IN "ON SIZE ERROR" CLAUSE'
'MISSING "ERROR" AFTER "SIZE" IN "ON SIZE ERROR" CLAUSE'
'INVALID STATEMENT IN "ON SIZE ERROR" CLAUSE'
'MISSING SD NAME IN "SORT" STATEMENT'
'MISSING DATA NAME OR LITERAL IN "MAXIMUM RECORDS" CLAUSE'
'NO SORT KEYS SPECIFIED FOR "SORT" STATEMENT'
'NO "INPUT PROCEDURE" OR "USING" CLAUSE IN "SORT" STATEMENT'
'NO "OUTPUT PROCEDURE" OR "GIVING" CLAUSE IN "SORT" STATEMENT'
```

```
'MISSING FILE NAME IN "START" STATEMENT'
'INVALID LOGICAL OPERAND IN "KEY IS" CLAUSE OF "START" STATEMENT'
'INVALID LOGICAL OPERAND IN "KEY IS" CLAUSE OF "START" STATEMENT'
'MISSING DATA NAME IN "KEY IS" CLAUSE OF "START" STATEMENT'
'DATA NAME OR LITERAL DOES NOT FOLLOW "BLOCK" IN "START" STATEMENT'
'DATA NAME OR LITERAL DOES NOT FOLLOW "CHARACTER" IN "START" STATEMENT'
'MISSING LITERAL OR "RUN" AFTER "STOP"'
'MISSING DATA NAME OR LITERAL IN "STRING" STATEMENT'
'MISSING "INTO" DATA NAME PHRASE IN "STRING" STATEMENT'
'MISSING DATA NAME AFTER "INTO" IN "STRING" STATEMENT'
'MISSING "POINTER" DATA NAME IN "STRING" STATEMENT'
'"POINTER" DOES NOT FOLLOW "WITH" IN "STRING" STATEMENT'
'"DELIMETED" DOES NOT FOLLOW DATA NAME OR LITERAL IN "STRING" STATEMENT'
'INVALID DELIMETER FOLLOWING "DELIMETED BY" IN "STRING" STATEMENT'
'INVALID SUBSCRIPT EXPRESSION'
'MISSING RIGHT PARENTHESIS FOLLOWING SUBSCRIPT'
'MISSING DATA NAME OR LITERAL IN "SUBTRACT" STATEMENT'
'MISSING "FROM" IN "SUBTRACT" STATEMENT'
'MISSING DATA NAME OR LITERAL FOLLOWING "FROM" IN "SUBTRACT" STATEMENT'
'"GIVING" DOES NOT FOLLOW "FROM" LITERAL'
'DATA NAME DOES NOT FOLLOW "GIVING"'
'MISSING GROUP LEVEL NAME IN "SUBTRACT CORRESPONDING" STATEMENT'
'"FROM" DOES NOT FOLLOW DATA NAME IN "SUBTRACT CORRESPONDING" STATEMENT'
'MISSING GROUP LEVEL NAME IN "SUBTRACT CORRESPONDING" STATEMENT'
'MISSING DATA NAME,LITERAL OR EXPRESSION'
'INVALID EXPRESSION FOLLOWS EXPONENTIATION OPERATOR'
'RECORD NAME DOES NOT FOLLOW "TERMINATE"'
'FILE NAME DOES NOT FOLLOW "TRUNCATE"'
'MISSING FILE NAME FOLLOWING "UNDELETE"'
'MISSING DATA NAME FOLLOWING "UNSTRING"'
'MISSING DATA NAME OR LITERAL FOLLOWING "DELIMITED BY" IN "UNSTRING" STATEMENT'
'MISSING "DELIMETED BY" OR "INTO"'
'MISSING DATA NAME FOLLOWING "INTO" IN "UNSTRING" STATEMENT'
'MISSING DATA NAME FOLLOWING "TALLYING IN" IN "UNSTRING" STATEMENT'
'MISSING DATA NAME FOLLOWING "DELIMITER IN" IN "UNSTRING" STATEMENT'
'MISSING DATA NAME FOLLOWING "COUNT IN" IN "UNSTRING" STATEMENT'
'"AFTER" OR "BEFORE" DOES NOT FOLLOW "USE"'
'"REPORTING" DOES NOT FOLLOW "BEFORE" IN "USE" STATEMENT'
'MISSING DATA NAME IN "USE BEFORE REPORTING" STATEMENT'
'MISSING "EXCEPTION" OR "ERROR" IN "USE AFTER STANDARD" STATEMENT'
'MISSING "PROCEDURE" IN "USE AFTER STANDARD" STATEMENT'
'INVALID CONDITION IN "USE AFTER STANDARD" STATEMENT'
'MISSING FILE NAME AFTER "USING"'
'MISSING "POINTER" AFTER "WITH"'
'MISSING DATA NAME AFTER "WITH POINTER"'
'MISSING RECORD NAME IN "WRITE" STATEMENT'
'ILLEGAL CHARACTER IN NUMERIC LITERAL'
'ILLEGAL NUMERIC LITERAL CONSTRUCTION'
'NUMERIC LITERAL > 18 DIGITS'
'ILLEGAL MOVE SOURCE/DESTINATION COMBINATION'
'LOCATOR TABLE OVERFLOW'
'TEMPORARY TABLE OVERFLOW'
'ILLEGAL COMBINATION OF TYPES IN A COMPARISON'
'STACK UNDERFLOW'
'ITEM EXCEEDS 32767 BYTES'
'ILLEGAL SOURCE KIND'
'INCORRECT NUMBER OF SUBSCRIPTS'
'ITEM MUST BE SUBSCRIPTED'
'ALPHABETIC TEST OF NUMERIC TEST IS MEANINGLESS'
'NUMERIC TEST ON ALPHA ITEM IS ILLEGAL'
'USER EXTERNAL SYMBOL TABLE OVERFLOW'
```

```
'DEVICE NAME EXPECTED'
'ILLEGAL DESTINATION KIND'
'STACK OVERFLOW'
'ILLEGAL TYPE IN ARITHMETIC EXPRESSION'
'SUBSCRIPT SHOULD BE AN INTEGER ITEM'
'SORT KEYS EXPECTED'
'REFERENCE TO A LINKAGE SECTION ITEM WHICH IS NOT A PARAMETER'
'UNSTRING DESTINATION MUST BE A DISPLAY ITEM'
'DELIMITERS MUST BE ALPHANUMERIC'
'INTEGER EXPECTED'
'INTEGER DATA ITEM EXPECTED'
'NUMERIC DATA ITEM EXPECTED'
'NONNUMERIC LITERAL EXPECTED'
'NON-NEGATIVELY SCALED NUMBER EXPECTED'
'ALPHANUMERIC DATA ITEM EXPECTED'
'DISPLAY DATA ITEM EXPECTED'
'LITERAL TABLE OVERFLOW'
'ONLY 18 DIGITS CAN BE MOVED TO A NUMERIC ITEM'
'NON-INTEGER NUMERIC ITEM CAN NOT BE MOVED TO A GROUP ITEM'
'NON-NUMERIC LITERAL OR DISPLAY DATA ITEM EXPECTED'
'POINTER MUST BE ABLE TO HOLD THE SOURCE ITEM LENGTH PLUS 1'
'SYMBOL TABLE OVERFLOW'
'MAGNITUDE EXCEEDS 10**(+/-64)'
'COMPILER STACK MANAGEMENT PROBLEM. RECOVERY ATTEMPTED.'
'LITERAL SUBSCRIPT OUT OF BOUNDS'
```

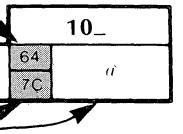
End of Appendix

Appendix E
ASCII Character Set

To find the *octal* value of a character, locate the character, and combine the first two digits at the top of the character's column with the third digit in the far left column.

LEGEND:

Character code in decimal
EBCDIC equivalent hexadecimal code
Character



OCTAL	00_		01_		02_		03_		04_		05_		06_		07_	
0	0 00	NUL	8 16	BS (BACK- SPACE)	16 10	DLE ↑P	24 18	CAN ↑X	32 40	SPACE	40 4D	(	48 F0	0	56 F8	8
1	1 01	SOH ↑A	9 05	HT (TAB)	17 11	DC1 ↑Q	25 19	EM ↑Y	33 5A	!	41 5D	)	49 F1	1	57 F9	9
2	2 02	STX ↑B	10 15	NL (NEW LINE)	18 12	DC2 ↑R	26 3F	SUB ↑Z	34 7F	" (QUOTE)	42 5C	*	50 F2	2	58 7A	:
3	3 03	ETX ↑C	11 0B	VT (VERT. TAB)	19 13	DC3 ↑S	27 27	ESC (ESCAPE)	35 7B	#	43 4E	+	51 F3	3	59 5E	;
4	4 37	EOT ↑D	12 0C	FF (FORM FEED)	20 3C	DC4 ↑T	28 1C	FS ↑\	36 5B	$	44 6B	, (COMMA)	52 F4	4	60 4C	<
5	5 2D	ENQ ↑E	13 0D	RT (RETURN)	21 3D	NAK ↑U	29 1D	GS ↑]	37 6C	%	45 60	-	53 F5	5	61 7E	=
6	6 2E	ACK ↑F	14 0E	SO ↑N	22 32	SYN ↑V	30 1E	RS ↑↑	38 50	&	46 4B	. (PERIOD)	54 F6	6	62 6E	>
7	7 2F	BEL ↑G	15 0F	SI ↑O	23 26	ETB ↑W	31 1F	US ↑←	39 7D	' (APOS)	47 61	/	55 F7	7	63 6F	?

OCTAL	10_		11_		12_		13_		14_		15_		16_		17_	
0	64 7C	@	72 C8	H	80 D7	P	88 E7	X	96 79	` (GRAVE)	104 88	h	112 97	p	120 A7	x
1	65 C1	A	73 C9	I	81 D8	Q	89 E8	Y	97 81	a	105 89	i	113 98	q	121 A8	y
2	66 C2	B	74 D1	J	82 D9	R	90 E9	Z	98 82	b	106 91	j	114 99	r	122 A9	z
3	67 C3	C	75 D2	K	83 E2	S	91	[	99 83	c	107 92	k	115 A2	s	123 C0	{
4	68 C4	D	76 D3	L	84 E3	T	92 E0	\	100 84	d	108 93	l	116 A3	t	124 4F	|
5	69 65	E	77 D4	M	85 E4	U	93	]	101 85	e	109 94	m	117 A4	u	125 D0	}
6	70 C6	F	78 D5	N	86 E5	V	94 5F	↑ or ^	102 86	f	110 95	n	118 A5	v	126 A1	~ (TILDE)
7	71 C7	G	79 D6	O	87 E6	W	95 6D	← or _	103 87	g	111 96	o	119 A6	w	127 07	DEL (RUBOUT)

SD-00217B Character code in octal at top and left of charts.

↑ means CONTROL

End of Appendix

Appendix F
EBCDIC Character Set

	0_	1_	2_	3_	4_	5_	6_	7_	8_	9_	A_	B_	C_	D_	E_	F_
0	0 NUL	16 DLE	32 DS	48	64 SPACE	80 &	96 - MINUS	112	128	144	160	176	192 {	208 }	224 \	240 0
1	1 SOH	17 DC1	33 SOS	49	65	81	97 /	113	129 a	145 j	161 ~	177	193 A	209 J	225	241 1
2	2 STX	18 DC2	34 FS	50 SYN	66	82	98	114	130 b	146 k	162 s	178	194 B	210 K	226 S	242 2
3	3 ETX	19 TM	35	51	67	83	99	115	131 c	147 l	163 t	179	195 C	211 L	227 T	243 3
4	4 PF	20 RES	36 BYP	52 PN	68	84	100	116	132 d	148 m	164 u	180	196 D	212 M	228 U	244 4
5	5 HT	21 NL	37 LF	53 RS	69	85	101	117	133 e	149 n	165 v	181	197 E	213 N	229 V	245 5
6	6 LC	22 BS	38 ETB	54 UC	70	86	102	118	134 f	150 o	166 w	182	198 F	214 O	230 W	246 6
7	7 DEL	23 IL	39 ESC	55 EOT	71	87	103	119	135 g	151 p	167 x	183	199 G	215 P	231 X	247 7
8	8	24 CAN	40	56	72	88	104	120	136 h	152 q	168 y	184	200 H	216 Q	232 Y	248 8
9	9	25 EM	41	57	73	89	105	121 ` GRAVE	137 i	153 r	169 z	185	201 I	217 R	233 Z	249 9
A	10 SMM	26 CC	42 SM	58	74	90 !	106 ¦	122 : COLON	138	154	170	186	202	218	234	250
B	11 VT	27 CU1	43 CU2	59 CU3	75 . PERIOD	91 $	107 , COMMA	123 #	139	155	171	187	203	219	235	251
C	12 FF	28 IFS	44	60 DC4	76 <	92 *	108 %	124 @	140	156	172	188	204	220	236	252
D	13 CR	29 IGS	45 ENQ	61 NAK	77 (	93)	109 _ UNDER LINE	125 ' APOS-TROPHE	141	157	173	189	205	221	237	253
E	14 SO	30 IRS	46 ACK	62	78 +	94 ;	110 >	126 =	142	158	174	190	206	222	238	254
F	15 SI	31 IUS	47 BEL	63 SUB	79 !	95 ¬	111 ?	127 " QUOTE	143	159	175	191	207	223	239	255

DECIMAL CHARACTER CODE IN UPPER LEFT CORNER OR EACH BOX.
HEXADECIMAL CHARACTER CODE AT TOP AND LEFT OF CHART.

SD-01084

End of Appendix

Appendix G
Using the Communications Access Manager with COBOL

When you use CAM (the Communications Access Manager) with a COBOL program, your control over the communications lines comes from two sources: the line characteristics supplied for each line during system generation (CGEN), and the processing parameters you include in your program code. This appendix discusses programming for CAM in COBOL. Refer to the *Communications Access Manager Reference Manual* (093-000183) for information about line characteristics.

CAM provides access to the communications system through a series of program calls, which we will refer to as CAM calls. Before you can use the calls, you must set up a CAM file in your COBOL program; you can then enter processing parameters and issue CAM calls through the CAM file. When you issue a call, it accesses a CAM software routine, which performs some communication function for you.

CAM supplies teletypewriter protocol for asynchronous lines, and the CAM calls handle the protocol. You will neither see nor be concerned with the protocol in your program.

The following sections discuss CAM call functions, tell how to set up a COBOL CAM file, and give descriptions of the individual CAM calls. The call descriptions are divided into sections for CAM System Calls and Asynchronous I/O Calls. Since ECLIPSE COBOL does not use synchronous lines, no synchronous calls are available to COBOL. This appendix concludes with a sample program in COBOL (Figure G-1).

CAM Call Functions

By combining the asynchronous I/O calls with the CAM system calls, you can have complete communications control over asynchronous lines. You can initialize the system, send and receive data, manage completion processing, monitor modem status, connect and disconnect individual lines at will, and deactivate the system when communications operations are finished.

Before you set up your COBOL CAM file and start to issue CAM calls, you should read the following sections to learn how CAM works, how it uses data transfer modes, and how it performs completion processing and data buffering.

Before you can issue any I/O calls, you must initialize the communications system by issuing the CINT system call. Once the system is initialized, you can begin to issue calls to send data, receive data, and perform other I/O functions. When you issue I/O calls, you must decide whether to suspend the calling task until the call is completed, or to allow the task to regain control as soon as the I/O operation begins.

If you are controlling only one line with the calling task, you will usually want to suspend the task until the I/O operation is complete. If you are controlling multiple lines, it is usually more efficient to issue the call unpended, so that the task can regain control immediately and go on to perform other operations while the first operation is completing.

Whether you are controlling single or multiple lines, your program needs to know when each I/O operation is complete. CAM furnishes this information through completion processing.

Completion Processing

CAM considers a send operation complete when the program sends the last character from the CAM buffer and the receiving station acknowledges reception of that character. In a receive operation, CAM completes the call by moving the received data from the CAM buffer to the program buffer, after first sending a receive acknowledgement to the sending station.

CAM performs completion processing in one of two ways, depending on whether you choose to pend (suspend) the calling task while the call is processing, or return control to the task as soon as the operation starts. When you use the pend option, CAM suspends the calling task until the operation is complete, and then allows the task to restart.

If you issue the call unpended, CAM allows the calling task to regain control as soon as the operation begins. When the operation is complete, CAM posts an entry in the completion queue, and you must issue the CIOC call to fetch the entry from the queue and notify your program that a completion has occurred. In an unpended receive operation, the received data remains in the CAM buffer until you issue a completion call; CIOC then moves the data into the program buffer as part of the completion processing function.

Data Buffer Management

CAM maintains a pool of internal buffers for asynchronous lines. You specify the number and size of these buffers during CGEN. The buffer size has no bearing on the maximum number of characters the system can send or receive, but for better memory usage, the buffers should be the same size as the most frequently transmitted or received data block.

When you are transmitting, supply the number of characters that will be transmitted in the CSND calling sequence or program buffer record. CAM uses this byte count to get the required number of CAM buffers from the pool, chains them, transfers data from your program buffer to the chained CAM buffers, and starts data transmission from the CAM buffers. Once a buffer is emptied (data is transmitted), CAM frees that buffer to the pool, then continues transmission from the next buffer in the chain. Thus transmission buffers are freed to the buffer pool one at a time.

When you are receiving, CAM acquires buffers from the pool at character level, one at a time (as they are needed) and chains them. CAM receive buffers are freed to the buffer pool after the data reception is complete and the data has been moved to the program buffer.

Data Transfer Modes

CAM transfers data over asynchronous lines in either line or sequential mode. Remember: when we speak of transferring data, we mean that your program is sending or receiving data via the CPU; we are not speaking of sending or receiving operations at the terminal end. It is important to keep this concept in mind when dealing with CAM send and receive operations.

If the CPU is sending data, we may say that it is writing; if receiving data, it is reading. Therefore, since asynchronous lines operate in either line or sequential mode, you can initiate data transfer operations in write line or write sequential mode, or in read line or read sequential mode.

In line mode, CAM interprets the data as ASCII characters and processes it using a standard teletypewriter protocol. In read line mode, you terminate every incoming data string with a carriage return, form feed, or null, or with any one of the terminating characters you specified for that line during CGEN. In write line mode, you may terminate the outgoing data string with a carriage return, form feed, or null only. During CGEN, you can select an option that causes CAM to insert a line feed (ASCII 12) after every outgoing carriage return (ASCII 15).

CAM provides the following line editing features in read line mode:

- The receive operation checks and strips the parity bit from the incoming characters (specify the parity option during CGEN).
- A rubout character (ASCII 177) deletes from the buffer the character preceding the rubout and moves the cursor one position backward on the CRT display; on hard copy terminals, a backarrow echoes on the printout to confirm that the character has been deleted from the buffer.
- A backslash (SHIFT/L) character (ASCII 134) deletes the entire line and resets the byte pointer to the beginning of the buffer. The backslash echoes at the terminal.
- A full-duplex line echoes all characters unless you select echo suppression in the .CRCV parameter list.

In sequential mode, CAM makes no assumptions about the data; it sends and receives the data as 8-bit binary characters. You will define the length of the data blocks to be received or transmitted by entering a byte count in the CAM calling sequence. In read sequential mode, you may terminate the incoming data string with any one of the terminating characters you specified for that line during CGEN. (Remember that CAM does not furnish the form feed, carriage return, or null terminators in sequential mode. You must define all terminating characters at CGEN.) CAM offers data translation from EBCDIC to ASCII (when receiving) or from ASCII to EBCDIC (when sending) as a parameter list option in sequential mode only.

Asynchronous Line Characteristics

You will set the individual line characteristics during CGEN. Such things as baud rate, character code level, parity, and number of stop bits depend on the characteristics of your hardware terminals. You must configure the lines to be compatible with the terminals. You can also define end-of-message characters on a line-by-line basis during CGEN. See the *CAM Reference Manual* for explanations of line characteristics.

CAM File Declaration

You may declare one (and only one) CAM file in a COBOL program. To declare the file, use a special SELECT statement:

SELECT *camfile* ASSIGN TO CAM.

Camfile is any filename you want to use in your COBOL communications program to reference the file. You assign the *camfile* in this way because COBOL sees the CAM program as an I/O device and treats it as such.

FD Entry

The second part of the CAM file declaration is a special FD entry in the Data Division's File Section. Use the FD entry to define the processing parameters shown in Table G-1. There are other parameters that you will enter when you issue the call itself in the Procedure Division. Check the parameter list and the individual call's calling sequence before you decide whether to set up the parameters in the list as permanent values or as variable values. You may want to be able to change some of them when you execute the CAM calls in the Procedure Division.

Remember that once the parameters are established as permanent values, the system will use them each time you issue a CAM I/O call. If your communications application is a simple send or receive of data blocks, which all use the same parameters, then the permanent values are efficient. If your application is more complex, involving sending and receiving data with more than one set of parameters, then the parameter values will have to be variables.

To establish the parameters as variables, enter the FD format like this:

```
FD CAM-FILE
      XTRN IS transl-status (or any other name you want to reference it by)
```

Then define the variable value in Working Storage:

```
WORKING-STORAGE SECTION.
77 TRANSL-STATUS    PIC 9    VALUE 0.
```

To establish the parameters as permanent values, simply enter the values in the FD statement:

```
FD CAM-FILE
      XTRN IS 0
```

A complete FD entry of the parameters as variables would look like this:

```
FD CAM-FILE
      CAM STATUS IS cam-status
      LENGTH IS cam-length
      XPND IS pend-status
      XTRN IS transl-status
      XMOD IS line-mode
      XECS IS echo-status
      XNMT IS ck-modem-ready
      TIME OUT IS time-out
```

The FD entry affects your entire communications program, so study the parameter list descriptions in Table G-1 and the descriptions of the individual CAM calls before you set it up. Remember to also consider the line characteristics you established during CGEN.

Refer to the sample COBOL program at the end of this appendix, paying particular attention to the way variable parameter values can be altered as illustrated by the CAM call statements in the Procedure Division.

Table G-1. CAM Parameter List

Parameter	Type	Description
CAM STATUS	Integer data item of 4 through 16 digits.	CAM condition code: CAM uses this parameter to return to you the status of the communications operations in your program. If the call executes successfully, CAM returns a zero to this location; if an exception condition occurs, CAM returns the condition code here. The exception condition codes are given in this chapter in octal numbers because they must be referenced in a COBOL program as alphanumeric values (equivalent to their octal value).
LENGTH	Integer data item of 1 through 16 digits.	Size of data block that was sent or received: When an I/O call is completed (either by the pended call or by a CIOC call issued to complete an unpended call), CAM returns to this location the byte (character) count of the data block that was sent or received.
XPND	Single-digit integer literal or data item.	Pend/Immediate return option: This option determines whether the call will be pended or take the immediate return. 0 = Return control to the calling task as soon as the operation begins. 1 = Suspend the calling task until the operation is complete.
XTRN	Single-digit integer literal or data item.	Data translation option: you may ask for data translation in sequential mode only. 0 = No translation: transfer data as is. 1 = Translate data between internal ASCII and external EBCDIC.

Parameter	Type	Description
XMOD	Single-digit integer literal or data item.	Data transfer mode: This option determines whether the data will be transferred in line or sequential mode. 0 = Transfer data in line mode. 1 = Transfer data in sequential mode.
XECS	Single-digit integer literal or data item.	Character echo option: 0 = Echo characters in full-duplex mode. 1 = Suppress character echo.
XNMT	Single-digit integer literal or data item.	Request modem-ready check: You may ask for this option only on modem-controlled lines. 0 = Check for modem-ready before the I/O operation begins. 1 = Do not check for modem-ready.
TIME OUT	Integer literal or data item of 1 through 16 digits.	Time out option: This option sets a time out value for I/O operations. 0 = Do not time out. *n* = The value must be greater than zero to time out the operation. Express the value in half seconds. For example, to set a time out cycle of 3 seconds, enter 6.

CAM System Control Calls

CAM System Control calls do exactly what their name implies: they control the communications system. Before you can send or receive data over individual lines you must initialize the CAM system. Then you must monitor line status while CAM is operating, and deactivate the system when the communications portion of your program has completed. You may also want to reconnect individual lines that get disconnected while the system is running. CAM System Control calls perform these tasks.

CINT

Initialize Communications System

When you are coding a communications program, you must issue the CINT call before you can execute any other CAM calls. CINT performs CAM start-up chores and readies the system for I/O processing.

In a system configured with one or more Data General DCU 50's, the CINT initialization routine does the following:

- Introduces each DCU to the operating system's interrupt structure;
- Loads each DCU 50 with its resident program and starts execution of the multiplexor initialization routine;
- Overwrites the DCU code in the host with CAM buffers and stacks; and
- Builds and initializes all data structures including queues and line tables.

In stand-alone (without DCU's) systems, the CINT initialization routine does the following:

- Introduces the multiplexor to the operating system's interrupt structure;
- Initializes the lines and builds and initializes all data structures including stacks, buffers, queues, and line tables.

The CINT routine initializes each modem-controlled line by clearing the modem signals and resetting the Data Terminal Ready signal, allowing recognition of incoming calls.

If you want to initialize the CAM system more than once in the same program, CINT is recallable. Once you have issued the first CINT, however, you must deactivate the system with the CDAC call before you can issue another CINT to reinitialize.

Calling Sequence

CALL CINT USING *camfile*

Where:

camfile is the name of your COBOL CAM file.

Returned CAM Status

0	no error
1014	illegal device code
1025	device in use
1026	configuration error
1033	DCU inoperative
1047	map error
1065	no free channel available (DCU systems only)
1066	error in opening or loading overlay file

CMOD
Modem Status

The connect status of modem-controlled lines may change while a communications program is running. For example, the other station may drop a line (for reasons of their own) without your knowledge. In order to use that line, your program must know its status; so when the line status changes, CAM logs the change in the modem status queue. You can then issue the CMOD call to retrieve the entry from the queue. CMOD retrieves the entry and returns it to you via the feedback argument in the calling sequence.

The CMOD call retrieves modem status entries from the queue one at a time; that is, each time you issue the call, CMOD retrieves the next entry in the queue. To check modem status on a continuing basis, you must periodically issue the call.

Calling Sequence

CALL CMOD USING *camfile, line, feedback*

Where:

camfile is the name of your COBOL CAM file (you enter this).

line CAM uses this argument to return to you the number of the line where the status change occurred.

feedback CAM returns a single-digit data item to this argument; the value represents one of the following:

0 a disconnect has occurred.

1 a connect has occurred.

2 a ring indicator has been recognized on a modem with auto-answer capability, but the Data Terminal Ready signal was not raised. Issue a CCNL call to raise the signal.

Returned CAM Status

0 no error
1042 no communications stack available
1046 queue empty

CCNL
Connect Line

Issue the CCNL call to reconnect an asynchronous line after it has been disconnected by a CDIS or CDDS call. CCNL raises the Data Terminal Ready signal on a modem-controlled line, allowing the modem to recognize incoming calls.

Calling Sequence

CALL CCNL USING *camfile, line*

Where:

camfile is the name of your COBOL CAM file.

line is the logical line number of the line you want connected.

Returned CAM STATUS

0 no error
1027 illegal line number
1028 line not active
1031 no line table for this line
1033 DCU inoperative
1042 no communications stack available

CDAC
Deactivate Communications System

Deactivate the communications system with the CDAC call when your program's communications activity is finished. The CDAC routine resets the hardware and software states, releases all of the CAM-associated data structures, disconnects the communications hardware from the operating system's interrupt structure, and restarts all tasks that were pended awaiting I/O completion.

NOTE: On a DCU-assisted system, if you have not issued a CDAC call, you must reset the hardware state from the computer console in order to halt the DCUs.

Calling Sequence

CALL CDAC USING *camfile*

Where:

camfile is the name of your COBOL CAM file.

Returned CAM STATUS

0 no error
1024 illegal device code
1026 configuration error
1028 device not active
1033 DCU inoperative

Asynchronous I/O Calls

The asynchronous I/O calls manage data transfer operations over local asynchronous lines or over remote modem-controlled asynchronous lines. Once you have initialized CAM with the CINT system call, you can transfer data using the send (CSND) and receive (CRCV) calls. CAM also provides a completion processing call (CIOC) and two line disconnect calls (CDIS and CDDS) for asynchronous communications.

CSND

Send Data

Issue the CSND call to start a send operation over an asynchronous line. CSND sends data over a local line or a modem-controlled line in either line or sequential mode. CAM allows you to issue the call pended or unpended. Also, you can select data translation if you are sending in sequential mode.

Calling Sequence

CALL CSND USING *camfile, rec, line, (len)*

Where:

camfile is the name of your COBOL CAM file.

rec is any record that you defined for the CAM file (or any Working Storage record) and that you will use as a program buffer. The output data will be transmitted from this location.

line is the logical line number of the line that will be used to transmit the data.

(len) is an integer literal or data item. Enter this item only if you are transmitting in sequential mode; it specifies the number of bytes (characters) in the data block. If you do not enter this value, it will default to the length of *rec*.

Returned CAM STATUS

0 no error
1027 illegal line number
1028 line not active
1029 line busy
1030 modem not ready
1031 no line table for this line
1033 DCU inoperative
1034 no communications buffers available
1035 byte count error
1036 line timeout/retry count exhausted
1042 no stack space available
1067 power fail error

CRCV
Receive Data

Issue the CRCV call to start a receive operation over an asynchronous line. CRCV receives data from a local or modem-controlled line in either read line or read sequential mode. CAM allows you to issue the call pended or unpended. Also, you can select data translation if you are receiving in read sequential mode.

CAM allows you to issue multiple CRCV calls over the same line when you use nonpending (immediate return) calls. CAM builds a communications stack for each CRCV call as it is issued. When multiple calls have been issued and the current call finds the line still busy with a previous call, CAM chains the communications stack to the line and passes control back to the task. The task can then immediately issue another call. Then, if the line is still busy, CAM again chains the stack to the line and the cycle is repeated. Any number of calls can be chained to one line.

You must issue periodic CIOC calls to monitor receive completions. CAM posts a receive completion after each operation is finished and continues receiving data with the next outstanding CRCV call. When all calls are processed, CAM stops receiving data on that line until you issue another call.

Calling Sequence

CALL CRCV USING *camfile, rec, line, (len)*

Where:

camfile is the name of your COBOL CAM file.

rec is any record that you defined for the CAM file (or any Working Storage record) and that you will use as a program buffer. The incoming data block will be stored in this location.

line is the logical line number of the line that will be used to receive the data.

(len) is an integer literal or data item. Enter this item only if you are receiving in sequential mode; it specifies the size in bytes (characters) of the incoming data block. If you do not enter this value, it will default to the length of *rec*.

Returned CAM Status

0	no error
1027	illegal line number
1028	line not active
1029	line busy
1030	modem not ready
1031	no line table for this line
1033	DCU inoperative
1034	no communications buffers available
1035	byte count error
1036	line timeout/retry count exhausted
1037	EOT character defined during CGEN was received
1042	no stack space available
1051	overrun error
1052	parity error
1053	overrun and parity
1054	framing error
1055	framing and overrun
1056	framing and parity
1057	framing, overrun, and parity
1064	line disconnected or CAM system deactivated
1067	power fail error

CIOC

Asynchronous I/O Completion Monitor

When you select the immediate return (unpended) option for an asynchronous I/O call, you must issue the CIOC call to determine when the operation is complete. CAM posts an entry in the I/O completion queue for every completed, unpended operation; CIOC checks the queue and returns the completion status of the first entry in the queue to your program. If you issue a series of unpended calls, you must issue CIOC periodically to monitor the queue on a continuing basis.

Remember that your program does not have access to received data until the call is complete. Data received from an unpended CRCV call remains in the CAM buffer until you issue CIOC to complete the call. When it finds a receive entry in the completion queue, CIOC moves the data from the CAM buffer to the program buffer.

You can issue the CIOC call itself as either pended or unpended. If you do not suspend the calling task (unpended), the call takes the normal return immediately if it finds an I/O completion in the queue. If it finds the queue empty, or finds an error condition, it takes an immediate exception return. Remember: you must issue another CIOC call to monitor the first one if you issue the call unpended.

If you select the pend option, CAM suspends the calling task until CIOC finds a completion entry in the queue. CIOC then takes the normal return to the calling task, allowing it to restart. When CAM finds an error condition, control returns to the pended task through the exception return.

Calling Sequence

CALL CIOC USING *camfile, rec, line, (xtcp)*

Where:

camfile is the name of your COBOL CAM file.

rec is any record that you defined for the CAM file (or any Working Storage record) and that you will use as a program buffer. When completing a CRCV call, CIOC stores the received data in *rec.*

line is the logical line number. CIOC returns the line number of the completed call to this location.

(xtcp) is a single-digit integer data item. CAM uses this item to return to you the type of call CIOC found in the completion queue:

0 completed call was a send;
1 completed call was a receive.

Returned CAM Status

0	no error
1035	byte count error
1036	line timeout
1046	completion queue empty

CDIS

Disconnect Line

Issue the CDIS call when you want to disconnect an asynchronous line from the communications system. After the line is disconnected, CDIS restarts any task that was pended awaiting I/O completion on the line. The line cannot be used again until you reconnect it. Issue CSND or CRCV to reconnect a local line; modem-controlled lines can only be reconnected with the CCNL CAM system call. You should issue the CMOD call to check the connect status of the line before you issue any other I/O call.

Calling Sequence

CALL CDIS USING *camfile, line*

Where:

camfile is the name of your COBOL CAM file.

line is the logical line number of the line you want to disconnect.

Returned CAM Status

0	no error
1027	illegal line number
1028	line not active
1029	line busy
1030	modem not ready
1031	no line table for this line
1033	DCU inoperative

CDDS

Disconnect Line and Store Data

Issue the CDDS call when you want to disconnect an asynchronous line from the communications system without losing any received data that may remain in the CAM buffer. When CDDS disconnects a line, it restarts any task that was pended awaiting I/O completion on the line. It also optionally moves any received data (if there is an uncompleted receive call on the line) from the CAM buffer into the program buffer.

The line cannot be used again until you reconnect it. You can reconnect a local line with a CSND or CRCV call; modem-controlled lines can only be reconnected with the CCNL CAM system call. Then you should issue the CMOD call to check the connect status of the line before you issue any other I/O call.

Calling Sequence

CALL CDDS USING *camfile, rec, line, (len)*

Where:

camfile is the name of your COBOL CAM file.

rec is any record that you defined for the CAM file (or any Working Storage record) and that you will use as a program buffer. Any received data remaining in the CAM buffer will be stored here. Do not use this item if you do not want to save the contents of the CAM buffer.

line is the logical line number of the line you want to disconnect.

(len) is the number of characters (bytes) to be moved from the CAM buffer to the program buffer. If you enter zero in this item, the data will not be transferred. If the data is moved, CAM returns the actual byte count of the data to this location.

Returned CAM Status

0	no error
1027	illegal line number
1028	line not active
1031	no line table for this line
1033	DCU inoperative

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID. DGCAM.
**** REMARKS
**** ILLUSTRATE DG COBOL INTERFACE TO CAM
**** THIS PROGRAM ILLUSTRATES TECHNIQUES FOR SETTING UP THE CAM PARAMETERS
**** AS VARIABLES IN THE DECLARATION DIVISION (THOUGH THE CAM PARAMETERS MAY
**** BE SET AS CONSTANTS ).
****  THE PROCEDURE DIVISION:
**** 1) ALLOWS THE USER TO SET UP THE LOGICAL LINE
**** 2) SET THE ECHO STATUS
**** 3) CHOOSE A CAM CALL BY USE OF CONDITIONAL PERFORM STATEMENTS
**** 4) DEMONSTRATES HOW THE ACTUAL LINE OF COBOL CODE MAY BE SET UP AND HOW
****    TO SET AND CHANGE (STATUS BITS) CAM PARAMETERS.
**** 5) ALLOWS THE USER TO REPEATEDLY EXECUTE CAM CALLS.
****
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SOURCE-COMPUTER. ECLIPSE.
 OBJECT-COMPUTER. ECLIPSE.
 SPECIAL-NAMES.
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
     SELECT CAM-FILE ASSIGN TO CAM.
 DATA DIVISION.
 FILE SECTION.
 FD  CAM-FILE
     CAM STATUS IS CAM-STATUS
     LENGTH IS CAM-LENGTH
     XPND IS PEND-STATUS
     XTRN IS TRANSL-STATUS
     XMOD IS LINE-MODE
     XECS IS ECHO-STATUS
     XNMT IS CK-MODEM-RDY
     TIME OUT IS TIME-OUT.
 01  CAM-REC              PIC X(100).
 WORKING-STORAGE SECTION.
 77  CAM-STATUS           PIC 9999        VALUE 9999.
 77  REC-LENGTH           PIC 999         VALUE 100.
 77  CAM-LENGTH           PIC 999.
 77  PEND-STATUS          PIC 9           VALUE 1.
 77  TRANSL-STATUS        PIC 9           VALUE 0.
 77  LINE-MODE            PIC 9           VALUE 1.
 77  ECHO-STATUS          PIC 9           VALUE 1.
 77  CK-MODEM-RDY         PIC 9           VALUE 1.
 77  TIME-OUT             PIC 9           VALUE 0.
 77  LINE-NR              PIC 99          VALUE 1.
 77  LINE-NR-RET          PIC 9.
 77  IO-TYPE              PIC 9.
 77  CODE-IN              PIC X.
 77  FDBK                 PIC 9           VALUE 0.
 01  TEST-SCR.
     03  FILLER           PIC X           VALUE "<15>".
     03  FILLER           PIC X           VALUE "<12>".
     03  FILLER           PIC X(10)       VALUE "0123456789".
     03  FILLER           PIC X           VALUE "<15>".
     03  FILLER           PIC X           VALUE "<12>".
 PROCEDURE DIVISION.
 000-BEGIN.
                 DISPLAY "LOGICAL LINE NUMBER? ".
                 ACCEPT LINE-NR.
****THE ABOVE LINE SETS THE PROPER LOGICAL LINE
```

Figure G-1. Sample CAM COBOL Program

```
****WHICH WILL BE USED IN THIS PROGRAM
****FOR CAM EXECUTION.  IT IS NECCESSARY THAT THIS LOGICAL
****LINE NUMBER BE ASSIGNED TO A PHYSICAL LINE IN
****THE COMMUNICATIONS SYSTEM GENERATION 'CGEN CBLCON'.
     DISPLAY "SET ECHO STATUS...".
     DISPLAY "ENTER 0 TO SET ECHO OR 1 TO SUPPRESS ECHO:".
     ACCEPT CODE-IN.
**** USER INPUTS ECHO STATUS
     IF CODE-IN NOT = "0" AND CODE-IN NOT = "1"
       GO TO 000-BEGIN.
     MOVE CODE-IN TO ECHO-STATUS.
**** SET ECHO STATUS
 010-REQUEST-CODE.
****THIS PARAGRAPH DISPLAYS THE VARIOUS CAM CALLS AND THEIR
****ASSOCIATED CODE NUMBERS FOR THIS PROGRAM.
****BEFORE EXECUTING ANY CAM I/O IT IS NECCESSARY TO INITIALIZE THE
****COMMUNICATIONS LINE THRU THE 'CINT' SYSTEM CALL.
     DISPLAY "ENTER CODE (TYPE H FOR HELP):".
     ACCEPT CODE-IN.
     IF CODE-IN NOT = "H"
     GO TO 020-CHECK-CODE.
     DISPLAY "  1 = CSND".
     DISPLAY "  2 = CRCV".
     DISPLAY "  3 = CIOC".
     DISPLAY "  4 = CINT".
     DISPLAY "  5 = CDAC".
     DISPLAY "  6 = CMOD".
     DISPLAY "  7 = CCNL".
     DISPLAY "  8 = CDIS".
     DISPLAY "  9 = END JOB".
     GO TO 010-REQUEST-CODE.
 020-CHECK-CODE.
**** THIS PARAGRAPH CALLS THE SUBROUTINES TO BE PERFORMED ACCORDING
**** TO THE CODE INPUT.
     IF CODE-IN = "1"
       PERFORM 520-CSND.
     IF CODE-IN = "2"
       PERFORM 500-CRCV.
     IF CODE-IN = "3"
       PERFORM 510-CIOC.
     IF CODE-IN = "4"
       PERFORM 530-CINT.
     IF CODE-IN = "5"
       PERFORM 540-CDAC.
     IF CODE-IN = "6"
        PERFORM 550-CMOD.
     IF CODE-IN = "7"
       PERFORM 560-CCNL.
     IF CODE-IN = "8"
       PERFORM 570-CDIS.
     IF CODE-IN = "9"
     GO TO 900-EOJ.
     DISPLAY "CAM STATUS = ", CAM-STATUS.
     GO TO 010-REQUEST-CODE.
 500-CRCV.
     MOVE 100 TO CAM-LENGTH.
**** THE NUMBER OF CHARACTERS ACTUALLY INPUT IS STORED IN THIS ITEM.
     MOVE 0 TO LINE-MODE.
**** SET MODE ON LINE MODE.
     MOVE SPACES TO CAM-REC.
     MOVE 0 TO PEND-STATUS.
**** SET STATUS TO READ-IMMEDIATE.
     CALL CRCV USING CAM-FILE, CAM-REC, LINE-NR.
**** THIS CALL STARTS A READ OPERATION OVER AN ASYNCHRONOUS
**** COMMUNICATIONS LINE.
 510-CIOC.
```

Figure G-1. Sample CAM COBOL Program (continued)

End of Appendix

Appendix H
Handling Unlabeled Magnetic Tape in AOS

The following two programs show you how to create and read unlabeled magnetic tape files in the AOS system. The first program, MTAOUT.CO (Figure H-1.), creates a simple tape file. It has 200-character fixed-length records with 2000 characters per block. The second program, MTAINN.CO (Figure H-2), reads this tape file.

The most important thing in this example is the form of the SELECT clause in the programs.

```
SELECT OUTFILE ASSIGN TO "ANYTAPE:0".
```

OUTFILE (INFILE in the second program), is the internal name used in that particular COBOL program. The ASSIGN clause specifies the system filename, i.e., the external filename, as ANYTAPE:0. ANYTAPE is a dummy name you use to communicate with the AOS system. It could be LOGICALNAME or FOO or any other handy string. The :0 indicates which file you want on the tape. If you want to read the second file on a tape, you could use ANYTAPE:1; if you want the tenth file, you could use ANYTAPE:9. The first file is always file number 0.

If you wanted to run the two programs in Figures H-1 and H-2, you would proceed as follows:

Type a command of the form

```
MOUNT ANYTAPE <message>
```

This command sends a message to the system operator telling him/her to mount a tape for you. A message like the following will appear on the operator's console:

*FROM PID #: (EXEC) ** UNIT MOUNT ** FROM USER your-name PID = #*
FROM PID #: REQUEST IS < message >

The name ANYTAPE in the mount command tells the AOS system what you and your programs want to call this magnetic tape. < *message* > is any text string you want the operator to read. For example, you might want to explain how to find the right tape and whether to write-enable it or not. Here is a sample:

```
MOUNT ANYTAPE FOR OUTPUT; USE TAPE MARKED "12/14/77"
```

This sends to the operator the message:

FROM PID□: REQUEST IS FOR OUTPUT; USE TAPE MARKED "12/14/77".

In response, the operator will mount your tape (maybe on magnetic tape unit 3), write-enable it, then type a message like this on the system console:

```
CONTROL @EXEC MOUNTED @MTA3
```

At this point, the system will know that the tape which you call ANYTAPE is mounted on MTA unit 3. You will know that all this has been completed because, after you type the mount command, your process hangs without giving you the prompt character. When the operator has finished servicing the mount request you will execute your programs in the normal way, i.e.,

```
X MTAOUT
X MTAINN
```

When you have finished with the magnetic tape, type the following command:

```
DISMOUNT ANYTAPE
```

This breaks the connection between your logical name ANYTAPE and the assigned unit. For example, if you try to run MTAOUT immediately after this, it will fail (unless you happen to have a disk subdirectory called ANYTAPE). This command also sends a message to the operator to remove your tape and use the unit for something else.

```
0001    IDENTIFICATION DIVISION.
0002    PROGRAM-ID. MTAOUTPUT.
0003    *       EXAMPLE SHOWING CREATION OF AN UNLABELED MAG TAPE FILE
0004    ENVIRONMENT DIVISION.
0005    INPUT-OUTPUT SECTION.
0006    FILE-CONTROL.
0007            SELECT OUTFILE ASSIGN TO "ANYTAPE:0".
0008    DATA DIVISION.
0009    FILE SECTION.
0010    FD      OUTFILE,
0011            BLOCK CONTAINS 2000 CHARACTERS,
0012            RECORDING MODE FIXED.
0013    01      REC.
0014      02    UPINTEGER       PIC 9(7).
0015      02    FILLER          PIC X(193).
0016
0017    WORKING-STORAGE SECTION.
0018
0019    PROCEDURE DIVISION.
0020    INITS.
0021            OPEN OUTPUT OUTFILE.
0022            MOVE ALL "." TO REC.
0023            PERFORM WRITE-REC VARYING UPINTEGER
0024                    FROM 1 BY 1 UNTIL UPINTEGER > 100.
0025            CLOSE OUTFILE.
0026            STOP RUN.
0027
0028    WRITE-REC.
0029            WRITE REC.
0030
```

Figure H-1. Unlabeled Mag Tape File - Creation

```
0001    IDENTIFICATION DIVISION.
0002    PROGRAM-ID. MTAINPUT.
0003    *       EXAMPLE SHOWING READ OF AN UNLABELED MAG TAPE FILE
0004    ENVIRONMENT DIVISION.
0005    INPUT-OUTPUT SECTION.
0006    FILE-CONTROL.
0007            SELECT INFILE ASSIGN TO "ANYTAPE:0".
0008    DATA DIVISION.
0009    FILE SECTION.
0010    FD      INFILE,
0011            BLOCK CONTAINS 2000 CHARACTERS,
0012            RECORDING MODE FIXED.
0013    01      REC.
0014      02    REC-BEGIN       PIC X(10).
0015      02    FILLER          PIC X(187).
0016      02    REC-END         PIC XXX.
0017
0018    WORKING-STORAGE SECTION.
0019
0020    PROCEDURE DIVISION.
0021    INITS.
0022            OPEN INPUT INFILE.
0023            PERFORM READ-FILE THRU READ-FILE-END.
0024            IF NOT (REC-BEGIN = "0000100..." AND
0025                    REC-END   = "...")
0026            THEN DISPLAY "?MTAINN ERROR".
0027            CLOSE INFILE.
0028            STOP RUN.
0029
0030    READ-FILE.
0031            READ INFILE; AT END GO TO READ-FILE-END.
0032            GO TO READ-FILE.
0033    READ-FILE-END.
0034
```

Figure H-2. Unlabeled Mag Tape File - Reading

End of Appendix

Index

DataGeneral

Software Documentation Remarks Form

How Do You Like This Manual?

Title _____ No. _____

We wrote the book for you, and naturally we had to make certain assumptions about who you are and how you would use it. Your comments will help us correct our assumptions and improve our manuals. Please take a few minutes to respond.

If you have any comments on the software itself, please contact your Data General representative. If you wish to order manuals, consult the Publications Catalog (012-330).

Who Are You?

- ☐ EDP Manager
- ☐ Senior System Analyst
- ☐ Analyst/Programmer
- ☐ Operator
- ☐ Other _____

What programming language(s) do you use? _____

How Do You Use This Manual?

(List in order: 1 = Primary use)

- ____ Introduction to the product
- ____ Reference
- ____ Tutorial Text
- ____ Operating Guide
- ____ _____

Do You Like The Manual?

Yes	Somewhat	No	
☐	☐	☐	Is the manual easy to read?
☐	☐	☐	Is it easy to understand?
☐	☐	☐	Is the topic order easy to follow?
☐	☐	☐	Is the technical information accurate?
☐	☐	☐	Can you easily find what you want?
☐	☐	☐	Do the illustrations help you?
☐	☐	☐	Does the manual tell you everything you need to know?

Comments?

(Please note page number and paragraph where applicable.)

From:

Name _____ Title _____ Company _____

Address _____ Date _____

SD-00742

FOLD DOWN FIRST FOLD DOWN

FIRST
CLASS
PERMIT
No. 26
Southboro
Mass. 01772

BUSINESS REPLY MAIL

No Postage Necessary if Mailed in the United States

Postage will be paid by:

Data General Corporation

Southboro, Massachusetts 01772

ATTENTION: Software Documentation

FOLD UP SECOND FOLD UP

STAPLE

DataGeneral

users group

Installation Membership Form

Name _____ Position _____ Date _____

Company, Organization or School _____

Address _____ City _____ State _____ Zip _____

Telephone: Area Code _____ No. _____ Ext. _____

1. Account Category

- ☐ OEM
- ☐ End User
- ☐ System House
- ☐ Government

2. Hardware

	Qty. Installed	Qty. On Order
M/600		
C/350, C/330, C/300		
S/250, S/230, S/200		
S/130		
AP/130		
CS Series		
N3/D		
Other NOVA		
microNOVA		
Other (Specify) _____		

3. Software

- ☐ AOS
- ☐ DOS
- ☐ SOS
- ☐ RDOS
- ☐ RTOS
- ☐ Other

Specify _____

4. Languages

- ☐ Algol
- ☐ DG/L
- ☐ Cobol
- ☐ ECLIPSE Cobol
- ☐ Business BASIC
- ☐ BASIC
- ☐ Assembler
- ☐ Interactive
- ☐ Fortran
- ☐ RPG II
- ☐ PL/1
- ☐ Other

Specify _____

5. Mode of Operation

- ☐ Batch (Central)
- ☐ Batch (Via RJE)
- ☐ On-Line Interactive

6. Communications

- ☐ RSTCP
- ☐ HASP
- ☐ RJE80
- ☐ SAM
- ☐ CAM
- ☐ 4025
- ☐ Other

Specify _____

7. Application Description

○ _____

8. Purchase

From whom was your machine(s) purchased?

- ☐ **Data General Corp.**
- ☐ Other

Specify _____

9. Users Group

Are you interested in joining a special interest or regional Data General Users Group?

○ _____

DG-05810

DataGeneral

Data General Corporation, Westboro, Massachusetts 01581, (617) 366-8911

FOLD

STAPLE

FOLD

STAPLE

FOLD

FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 26 SOUTHBORO, MA. 01772

Postage will be paid by addressee:

Data General

ATTN: Users Group Coordinator

Southboro, Massachusetts 01772