Data General

COBOL Reference Manual (AOS)

COBOL
Reference Manual
(AOS)

093-000223-01

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. 093-000223

Printed in the United States of America
Revision 01, November 1980

NOTICE

COBOL
Reference Manual
(AOS)
093-000223

Revision History:

093-000223

Original Release - June 1979

First Revision - November 1980 (COBOL (AOS) Rev. 3.00)

This manual supercedes the AOS portion of 093-000180-02. A vertical bar or an asterisk in the margin of a page indicates substantive change or deletion, respectively, from revision 00.

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 Advanced Operating System (AOS).

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 about 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	describes the COBOL source-level debugger and its commands.
Chapter 10	describes how to use COBOL with the Data General Database Management System (DG/DBMS).
Chapter 11	tells you how to compile, bind, and execute your COBOL program under AOS.
Appendix A	contains a list of COBOL key words.
Appendix B	describes differences between AOS COBOL and CS Interactive COBOL.
Appendix C	tells you how to write COBOL-callable assembly language routines for AOS.
Appendix D	contains a table of the ASCII character set and one of the EBCDIC character set.
Appendix E	tells you how to create and read unlabeled magnetic tape files using COBOL in AOS.
Appendix F	tells you how to upgrade your RDOS COBOL programs to run on AOS.

Required Manuals

You should supplement your reading of this manual with the following: *INFOS ™ Storybook* (69-000019), *AOS Programmer's Manual* (93-000120), INFOS ™ *System User's Manual (AOS)* (93-000152), *Sort/Merge Utility User's Manual (AOS)* (93-000155), *DG/DBMS Reference Manual* (93-000163), and *A Guide to Using DG/DBMS* (69-000025).

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 that 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 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

Contents

Chapter 1 - Introduction

General Description of ECLIPSE COBOL 1-1
COBOL Program Structure 1-1
Language Extensions 1-2
    I/O Extensions 1-2
    Data Extensions 1-2
    COBOL-Level Debugger 1-2
Sample Program 1-3

Chapter 2 - Language Concepts

The Character Set 2-1
Separators 2-1
COBOL Words 2-1
    Key, Optional, and Special Character Words 2-2
    Figurative Constants 2-2
    Special Register 2-2
User-Defined Words 2-3
Literals 2-3
    Numeric Literals 2-3
    Alphanumeric Literals 2-4
Source Formats 2-4
    Lines, Statements, and Sentences 2-4
        Debug Lines 2-5
        Comment Lines 2-5
        Continuation Lines 2-5
    Clauses and Phrases 2-6
    The A-Margin 2-6

Chapter 3 - The Identification Division

Chapter 4 - The Environment Division

Structure 4-1
Configuration Section 4-2
The Source-Computer Paragraph 4-2
The Object-Computer Paragraph 4-2
The Special-Names Paragraph 4-2
File Organizations 4-5
Sequential Files 4-5
Relative Files 4-6
Indexed Files 4-6
Sort/Merge Files 4-7
Database Files 4-7
File Access Modes 4-8
Sequential Access Mode 4-8
Random Access Mode 4-8
Dynamic Access Mode 4-8
Input-Output Section 4-9
The File-Control Paragraph 4-9
SELECT Clause 4-9
OPTIONAL Clause 4-14
ASSIGN Clauses 4-14
RESERVE Clause 4-15
ORGANIZATION Clause 4-16
ACCESS MODE Clause 4-16
FILE STATUS and INFOS STATUS Clauses 4-18
PARITY Clause 4-18
ALLOW SUB-INDEX and LEVELS Clauses 4-18
KEY COMPRESSION Clause 4-19
SELECT Clause Examples 4-20
The I-O-Control Paragraph 4-21

Chapter 5 - The Data Division

Structure 5-1
File Section 5-2
File Description Entry 5-3
Block, Node, and Record Sizes 5-7
RECORDING MODE Clause 5-8
LABEL RECORDS Clause 5-9
VALUE OF Clause 5-10
DATA RECORD Clause 5-11
LINAGE Clause 5-11
CODE-SET Clause 5-13
FEEDBACK Clause 5-13
PAD Clause 5-14
MERIT Clause 5-14
PARTIAL RECORD Clause 5-14
Working-Storage Section 5-15
Virtual-Storage Section 5-15
CSIZE 5-17
The /M Switch 5-17
Linkage Section 5-18

Screen Section 5-18
The DISPLAY and ACCEPT Commands 5-18
DISPLAY 5-19
ACCEPT 5-19
Syntax 5-20
CPRINT 5-25
Data Types 5-26
Numeric Data 5-26
The Data Description Entry 5-28
Level Numbers 5-29
REDEFINES Clause 5-30
OCCURS Clause 5-31
Examples of Array Declarations 5-32
PICTURE Clause 5-33
Defining Alphabetic Items 5-33
Defining Alphanumeric Items 5-33
Defining Alphanumeric Edited Items 5-33
Defining Numeric Items 5-34
Defining Numeric Edited Items 5-35
Examples 5-35
Data Editing 5-36
Alphanumeric/Alphabetic Editing 5-36
Numeric Editing 5-36
USAGE Clause 5-40
SIGN Clause 5-41
SYNCHRONIZED Clause 5-41
JUSTIFIED Clause 5-42
BLANK WHEN ZERO Clause 5-42
VALUE Clause 5-42
Examples of Data Description Entries 5-43
The RENAMES Entry 5-44
The Condition Name Entry 5-45

Chapter 6 - The Procedure Division

Structure and Concepts 6-1
Name Qualification 6-2
Procedure Name Qualification 6-2
Data Name Qualification 6-3
Condition Name Qualification 6-3
Array Name Qualification 6-4
Handling Arithmetics 6-5
Common Arithmetic Phrases 6-6
The ROUNDED Phrase 6-6
The SIZE ERROR Phrase 6-6
The CORRESPONDING Phrase 6-7
Arithmetic Expressions 6-8
Conditional Expressions 6-9
Simple Expressions 6-9
Compound Expressions 6-10
Subprogramming 6-12
Segmentation 6-13
Print File Formatting 6-13

Indexed File Record Selection 6-15
The Position Phrase 6-15
The Relative Option Phrase 6-15
The KEY Series Phrase 6-17
Indexed File Record Options 6-18
Handling I/O Exception Conditions 6-19
The AT END Phrase 6-19
The INVALID KEY Phrase 6-19
The Declaratives Section 6-19
COBOL File Status Data Items 6-20
INFOS Status Data Items 6-21

Chapter 7 - Procedure Statements

ACCEPT 7-4
ACCEPT DATE/DAY/TIME 7-6
ADD 7-7
ALTER 7-9
CALL 7-10
CALL PROGRAM 7-12
CANCEL 7-14
CLOSE 7-15
COMPUTE 7-16
DEFINE SUB-INDEX 7-17
DELETE 7-19
DELETE FILE 7-21
DISPLAY 7-22
DIVIDE 7-23
EXIT 7-24
EXIT PROGRAM 7-25
EXPUNGE 7-26
EXPUNGE SUB-INDEX 7-27
GO 7-28
IF 7-29
INSPECT 7-31
LINK SUB-INDEX 7-36
MERGE 7-38
MOVE 7-40
MULTIPLY 7-43
OPEN 7-44
PERFORM 7-46
READ for a Sequential File 7-49
READ for a Relative File 7-50
READ for an Indexed File 7-51
RELEASE 7-53
RETRIEVE 7-54
RETURN 7-56
REWRITE for a Sequential File 7-57
REWRITE for a Relative File 7-58
REWRITE for an Indexed File 7-59
SEARCH 7-61
SEEK 7-63
SET 7-64
SET UP/DOWN 7-65
SORT 7-66
START for a Sequential File 7-69
START for a Relative file 7-70

START for an Indexed File 7-71
STOP 7-72
STRING 7-73
SUBTRACT 7-75
TRUNCATE 7-77
UNDELETE 7-78
UNLOCK 7-80
UNSTRING 7-81
USE 7-83
WRITE for a Sequential File 7-85
WRITE for a Relative File 7-87
WRITE for an Indexed File 7-88

Chapter 8 - The COPY Facility

Structure 8-1
Replacement Strings 8-2

Chapter 9 - The COBOL Interactive Debugger

Operating Instructions 9-1
Comment Lines 9-1
Debug Lines 9-1
Debugger Features 9-2
    Using Breakpoints 9-2
    Checking Program Status 9-2
    Controlling Program Execution 9-2
    Using Other Programs and Files 9-2
Debugger Commands 9-2
    AUDIT 9-3
    CLEAR 9-4
    CLI 9-5
in COMPUTE 9-6
    CON 9-7
    COPY 9-8
    DISPLAY 9-9
    ENV 9-10
    MOVE 9-11
    SET 9-12
    STOP 9-13
    WALKBACK 9-14

Chapter 10 - The Data General Database Management System (DG/DBMS) Interface

Compiling and Binding DG/DBMS With a COBOL Program 10-1
DG/DBMS Subschemas In The Data Division 10-2
    The SYSTEM Record Type 10-7
    Set Types 10-7
    Declaring Free Cursors 10-10

Overview of DML Statements in The Procedure Division 10-11
READY 10-13
INITIATE 10-13
FIND 10-13
STORE, GET, MODIFY, ERASE, CONNECT, DISCONNECT, RECONNECT, and the ASSIGN Clause 10-13
COMMIT 10-13
ROLLBACK 10-14
FINISH 10-14
Positioning Within a Database 10-14
Error Handling 10-14
Subprograms 10-15
DML Statement Reference Section 10-15
Opening and Closing a Subschema 10-16
READY 10-16
FINISH 10-17
Transaction Statements 10-18
INITIATE 10-18
COMMIT 10-19
ROLLBACK 10-20
Manipulating Set Connections 10-21
CONNECT 10-21
DISCONNECT 10-22
RECONNECT 10-23
Manipulating Record Occurrences 10-24
STORE 10-24
GET 10-25
MODIFY 10-26
ERASE 10-27
Condition Checking 10-28
IF 10-28
CHECK 10-29
Locating a Record Occurrence 10-30
FIND (positional) 10-30
FIND (using data items) 10-31
FIND (duplicates) 10-32
FIND (current) 10-33
FIND (owner) 10-34
Sample COBOL Programs Using DG/DBMS 10-35

Chapter 11 - How to Use COBOL Under AOS

Compiling, Binding, and Executing 11-1
Using the Compiler 11-1
Calling the Compiler 11-2
Virtual Code 11-2
A Word About Overlays 11-2
ANSI Standard Segmentation 11-2
Compiling A Program Using Virtual Code 11-3
Compiler Switches 11-4
Global Switches 11-4
Local Switches 11-4
Example 11-4
The COBOL Map Switch 11-5
Error Messages 11-9
Warning Messages 11-9
Binding Program Files 11-10

CBIND Global Switches 11-10
CBIND Local Switches 11-11
Binding Programs (CBIND) Using Virtual Code or ANSI Segmentation 11-12
The /NODEFILE Switch 11-15
Splitting the Loading Process 11-15
Executing Your COBOL Program 11-15
Executing in the Debugger 11-16
Runtime Errors 11-16
Error Messages 11-17
Nonfatal COBOL Program Errors 11-17
Fatal COBOL Program Errors 11-17
Trace Information 11-18
Example 11-18

Appendix A - COBOL Reserved Words

Appendix B - CS Compatibility

AOS-CS Differences B-1
Identification Division Incompatibilities B-1
Environment Division Incompatibilities B-2
Procedure Division Incompatibilities B-3
Transporting Files From your Interactive COBOL System To AOS B-4

Appendix C - Writing COBOL-Callable Assembly Language Routines

Appendix D - ASCII and EBCDIC Character Sets

Appendix E - Handling Unlabeled Magnetic Tape

Appendix F - Language Upgradability From RDOS to AOS

Identification Division (Chapter 3) F-1
Environment Division (Chapter 4) F-1
Data Division (Chapter 5) F-1
Procedure Division (Chapters 6 and 7) F-2

Tables

Table	Caption	
2-1	COBOL Character Set	2-1
2-2	User-defined Words	2-3
2-3	Indicator Characters	2-5
4-1	COBOL File Handling	4-8
4-2	SELECT Clause	4-11
5-1	FD Entry Clauses	5-5
5-2	COBOL Screen Section Function Delimiter Keys	5-19
5-3	COBOL Line and Column Positioning	5-22
5-4	Sign Overpunch Characters	5-26
5-5	Binary Number Storage	5-27
5-6	PICTURE Editing Symbols	5-37
5-7	Suppression Symbol Examples	5-38
5-8	Floating Insertion Editing Examples	5-39
6-1	Arithmetic Operators	6-8
6-2	Logical Operators	6-11
6-3	Relative Access	6-18
6-4	COBOL File Status Indicators	6-20
7-1	INSPECT Table Structure for Example	7-33
7-2	MOVE Rules	7-41
10-1	Data Manipulation Language Statements	10-11
11-1	/M Switch Output	11-5
B-1	Differences in FILE STATUS Error Codes	B-2

Illustrations

Figure	Caption	
1-1	COBOL Sample Program	1-3
5-1	LINAGE Clause Example	5-12
5-2	Internal Data Structure	5-15
5-3	COBOL Program Using Screen Formatting	5-24
6-1	Multilevel Indexed File	6-15
6-2	Relative Access	6-16
7-1	IF Statement Example	7-30
7-2	PERFORM Example	7-48
10-1	A DG/DBMS Subschema in a COBOL Program	10-2
10-2	Structure of Data in the Figure 10-1 Subschema	10-4
10-3	COBOL/DBMS Subschema Record Type Description	10-6
10-4	The Function of the User Work Area (UWA)	10-7
10-5	COBOL Subschema Set Relationship	10-8
10-6	Owner-Member Record Diagram	10-9
10-7	Cursor Positions	10-15
10-8	Sample Program Number 1	10-35
10-9	Logical Diagram of Sample Program Number 1	10-38
10-10	Sample Program Number 2	10-39
11-1	Sample COBOL Source Program Lising (produced with /L compiler switch)	11-7
11-2	COBOL Map Produced by Program in Figure 11-1	11-9
11-3	Example of an Overlay and Overlay Area Structure with Two Object Modules	11-12
C-1	AOS Assembly Language Routine Example	C-3
E-1	Creating an Unlabeled Magnetic Tape File	E-2
E-2	Reading an Unlabeled Magnetic Tape File	E-3

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:

Nucleus
Table Handling
Sequential I/O
Relative I/O
Indexed I/O
Sort/Merge
Segmentation
Library
Interprogram 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 and DG/DBMS 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.

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. (Chapter 11 contains 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 CUT-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. Sample Program (continues)

```
01      SUMMARY-LINE 1.
        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.
```

Figure 1-1. COBOL Sample Program (continued)

```
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.


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. Sample Program (concluded)

End of Chapter

Chapter 2
Language Concepts

The Character Set

The COBOL character set consists of the characters listed in Table 2-1. 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 include any number of spaces. COBOL recognizes lowercase alphabetic characters only in literals and user-defined COBOL words. Lowercase characters are not permitted in the PROGRAM-ID paragraph.

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

Table 2-1. COBOL Character Set

Character	Name	Character	Name
0 - 9	digits	,	comma
A - 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

Separators

The elements of a COBOL source program (words, literals, etc.) are delimited by separators. The separators are space, comma, or 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., $\geq$), 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 that 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 (ASCII 377_8).

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 NEW LINE 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.

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

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

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 ± 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 signaled. (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

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 characters, followed by a NEW LINE or form feed. 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 NEW LINE or form feed preceded by 0 to 255 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
At least four spaces (or a 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:

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

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:

```
[ OBJECT-COMPUTER . cmt-entry    [ MEMORY SIZE int-1 { WORDS | CHARACTERS | MODULES } ]

  [ PROGRAM COLLATING SEQUENCE IS { ASCII | STANDARD-1 | NATIVE | EBCDIC | alph } ]  [SEGMENT-LIMIT IS int-2]. ]
```

Where:

cmt-entry is any comment that is not a COBOL reserved word; it represents the object computer.

int-1 is a positive integer literal that specifies the size of memory in WORDS, CHARACTERS, or MODULES, whichever you specify. COBOL ignores this clause.

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

int-2 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.

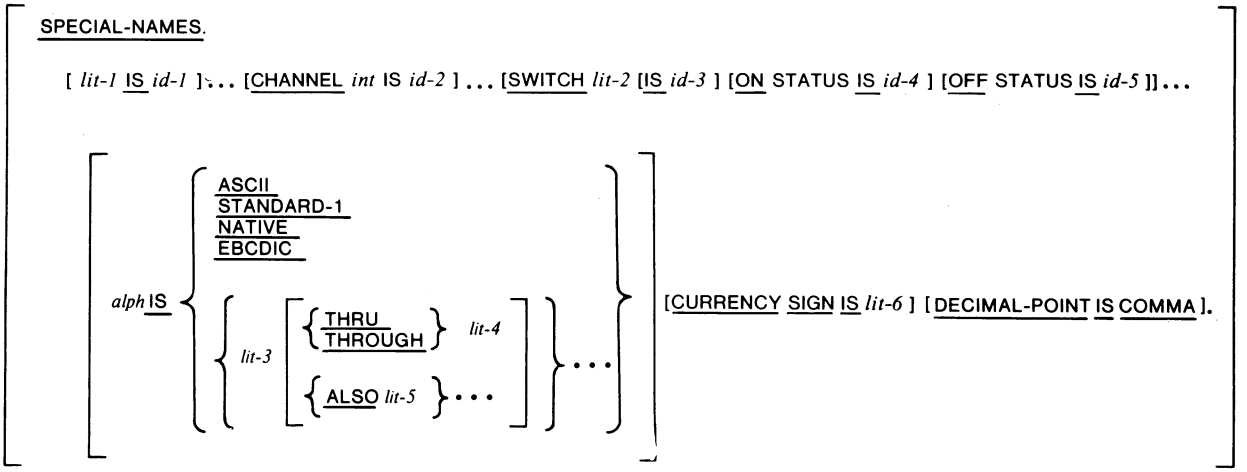
The *program collating sequence* is the relational order of the data characters which COBOL encounters internally during your program's execution. When COBOL compares character strings (alphanumeric data items or alphanumeric literals) in explicitly stated relational conditions and relational conditions 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 bound in the root context. 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, *int-2* are resident segments; those with segment numbers from *int-2* through 99 inclusive are overlay segments. See the section on virtual code in Chapter 11.

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:

lit-1 is an alphanumeric literal that specifies the name of a system I/O device or file.

id-1 is a mnemonic name that you use in your program to reference *lit-1*.

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

id-2 is the channel name you use in your program to reference the channel associated with *int*.

lit-2 is a single alphabetic character literal that specifies the program execution switch.

id-3 is the internal switch name for *lit-2*.

id-4, *id-5* are condition names associated with the ON and OFF STATUS conditions.

alph is an alphabet name.

lit-3, *lit-4*, *lit-5* are numeric or alphanumeric literals. If numeric, they must be unsigned integers, with values from 1 through 256. If alphanumeric, and you specify THROUGH or ALSO, they must be one character in length.

lit-6 is a single-character alphanumeric literal that specifies the currency character.

You use the device clause (the items *lit-1* and *id-1)* 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. Never reference the internal switch name *(id-3)* 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 D. If you specify EBCDIC, *alph* represents the EBCDIC code set/collating sequence also in Appendix D. 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-3* is less than *lit-4*, COBOL assigns successive ascending positions in the collating sequence to the contiguous characters in the ASCII set beginning with the character specified by *lit-3* and ending with the character specified by *lit-4*. If *lit-3* is greater than *lit-4* COBOL assigns successive ascending positions (reverse order) in the collating sequence to the contiguous characters in the ASCII set. Assignment begins with *lit-3* and continues backward through *lit-4*.

6. If you specify ALSO, COBOL assigns the same position in the collating sequence to the characters of the ASCII set within *lit-3* and *lit-5*.

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, {A, Z}, M, N, O, P, Q, {B, F}, E, NULL, ↑A, ↑B, ...

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.

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 four types of file organization: sequential, relative, indexed, and database. 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 data on a sequential access device (e.g., magnetic tape), or if you want to output data to a sequential device (e.g., line printer), you should use sequential file organization to create and process the 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 a 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 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, it takes time as COBOL passes 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 a 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 data, or you may reference 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 two directories or more containing the file's index and database. 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 ORGANIZATION IS INDEXED in the file's SELECT clause. You can access an indexed file only on disk devices.

*

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).

Database Files

COBOL uses the DG/DBMS (Data General Database Management System) to handle database files. Chapter 10 gives a full description of how to use DG/DBMS.

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 file handling 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, 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

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

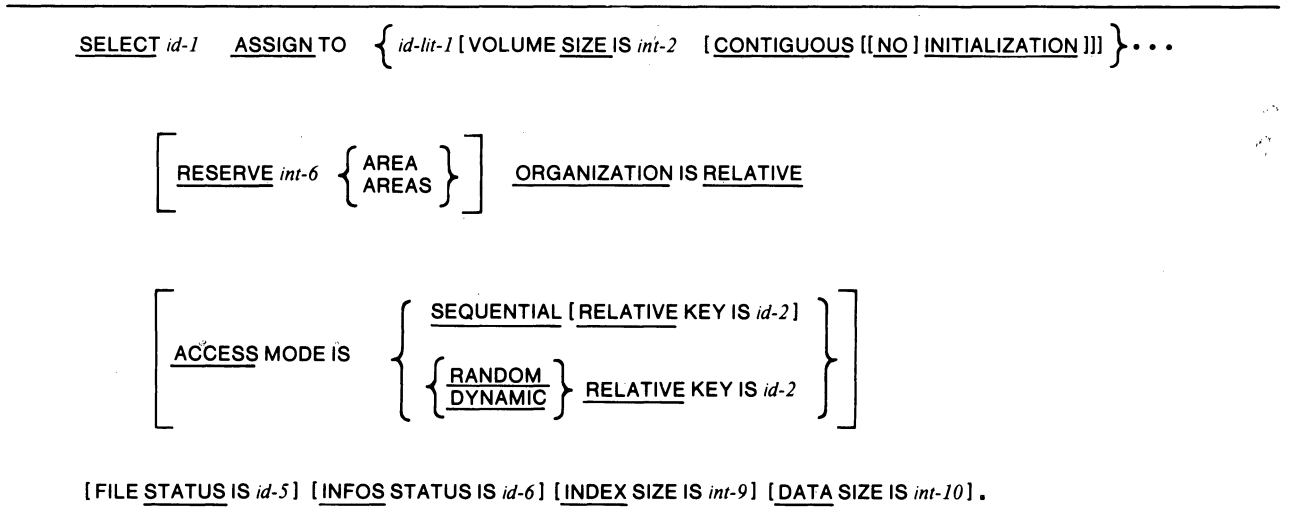
You must specify one SELECT clause for each file. In each clause, you name a file; you may also specify other file-related information. For each file you specify in this paragraph, you must include a file description entry in the Data Division (see Chapter 5).

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.

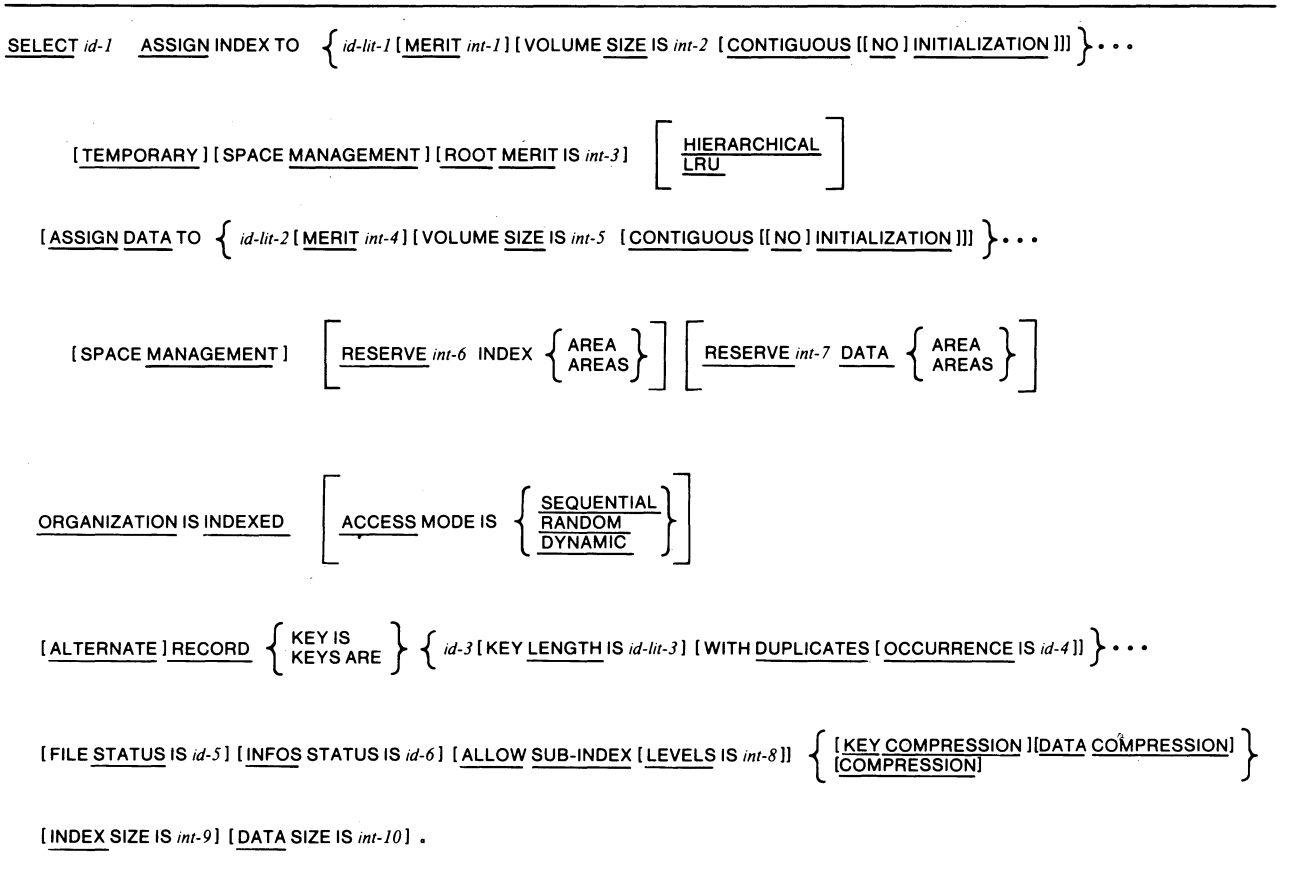
Format for a sequential file:

```
SELECT [ OPTIONAL ] id-1  ASSIGN TO { { id-lit-1 [ VOLUME SIZE IS int-2 [ CONTIGUOUS [[ NO ] INITIALIZATION ]]] } ...
                                    { PRINTER [ id-lit-1 ]
                                    { [ DISK [ id-lit-1 ]]
                                    { [ DISPLAY id-lit-1 ]
                                    { [ PRINTER-1 [ id-lit-1 ]]
                                    { [ KEYBOARD id-lit-1 ]

[ RESERVE int-6 { AREA  } ]  [ ORGANIZATION IS SEQUENTIAL ] [ ACCESS MODE IS SEQUENTIAL ]
                { AREAS }

[ FILE STATUS IS id-5 ] [ INFOS STATUS IS id-6 ]  [ PARITY IS { ODD  } ]  [ INDEX SIZE IS int-9 ] [ DATA SIZE IS int-10 ].
                              { EVEN }
```

Format for a relative file:



Format for an indexed file:



Format for a sort/merge file:

```
SELECT id-1  ASSIGN TO id-lit-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.

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 1.	
VOLUME SIZE	X	X	X			Disk files only.
CONTIGUOUS	X	X	X		COBOL will allocate blocks randomly.	
INITIALIZATION					Initialization.	COBOL ignores this clause.
PRINTER	X				@LPT	
TEMPORARY						COBOL ignores this clause.
SPACE MANAGEMENT			X		No space management.	
ROOT MERIT			X		COBOL assigns root node priority.	COBOL ignores this clause.
HIERARCHICAL/LRU						
DISK	X	X	X	X	fn	
DISPLAY	X					
PRINTER-1	X				@LPT1	
KEYBOARD	X					

(continues)

Table 4-2. SELECT Clause

Clauses and Options	Sequential Files	Relative Files	Indexed Files	Sort/Merge Files	Default	Restrictions
RESERVE (DATA)						COBOL ignores this clause.
RESERVE (INDEX)						COBOL ignores this clause.
ORGANIZATION	X	X	X		Sequential.	
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 length specified in the key item's PICTURE clause.	
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.	

(continued)

Table 4-2. SELECT Clause

Clauses and Options	Sequential Files	Relative Files	Indexed Files	Sort/Merge Files	Default	Restrictions
ALTERNATE RECORD (cont.)						
DUPLICATES			X		COBOL sends an error message if it encounters a duplicate alternate key.	
OCCURRENCE			X		None.	
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						COBOL ignores this clause.
ALLOW SUBINDEX			X		No subindexing allowed.	You cannot specify this option if you have alternate record keys.
LEVELS			X		Maximum number of levels is 32.	
KEY COMPRESSION			X		Redundant key information will be stored.	
DATA COMPRESSION			X		Redundant data record information will be stored.	
COMPRESSION			X		Redundant key and data information will be stored.	
INDEX SIZE						COBOL ignores this clause.
DATA SIZE						COBOL ignores this clause.

(concluded)

OPTIONAL Clause

```
[ OPTIONAL ] id-1
```

Where:

id-1 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

{ id-lit-1 [MERIT int-1 ] [VOLUME SIZE IS int-2 [CONTIGUOUS [[NO] INITIALIZATION]]]
  [DISK [ id-lit-1 ]]
  [DISPLAY id-lit-1 ]
  [PRINTER [ id-lit-1 ]]
  [PRINTER-1 [ id-lit-1 ]]
  [KEYBOARD id-lit-1 ] } ...

[ TEMPORARY ] [ SPACE MANAGEMENT] [ ROOT MERIT IS int-3 ] [ HIERARCHICAL | LRU ]

[ ASSIGN DATA TO { id-lit-2 [ MERIT int-4 ] [ VOLUME SIZE IS int-5 [ CONTIGUOUS [[ NO ] INITIALIZATION ]]] }
    [ SPACE MANAGEMENT ] ]
```

Where:

id-lit-1, id-lit-2 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 *id-1*.

int-1, int-4 is a positive integer literal that specifies the priority of a volume.

int-2, int-5 is a positive integer literal that specifies a number of blocks.

int-3 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 *id-1* with a storage medium and say something about its physical makeup.

You must specify at least one symbolic name *id-lit-1* 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 created the file as a new indexed file or as a new inversion of an existing indexed file, or if you intend to EXPUNGE an indexed file, you must also specify the ASSIGN DATA clause with the data record file system names to define the database filename. Otherwise, the default is <filename>.DB.

You must use the VOLUME SIZE clause in conjunction with CONTIGUOUS, or it has no effect. VOLUME SIZE sets the number of contiguous disk blocks allotted to the file. Each time the allotment is full, AOS allocates another chunk of that size to the file. Large volume sizes permit fast access to information, but unused areas in the allotment waste disk space.

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 *id-lit-1* you specify. For specific information on print file formatting, see Chapter 6.

The SPACE MANAGEMENT clause (for indexed files only) optimizes the size of an INFOS file, saving space at a slight increase in overhead.

The MERIT clause (indexed files only) assigns a merit factor to an INFOS database file. Later, when you write records to the file, you can specify a record merit factor which will place the record on a volume with the same (or if no match exists, lower) merit factor. You can replace these volumes in the database with links to files on different logical disks, thus assigning specific records to specific disks. However, creating this link structure requires AOS CLI intervention.

The ROOT MERIT clause (indexed files only) assigns a merit factor to the INFOS system index root node. Read the *INFOS System User's Manual (AOS)* for details on index node merit factors. To use this facility effectively, you must create a link structure in the same manner as for database volumes.

COBOL always ignores the TEMPORARY, and LRU clauses.

RESERVE Clause

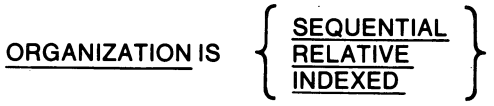
Where:

int-6 is an integer literal that specifies the number of input/output buffers you want to allocate for your file's indexes.

int-7 is an integer literal that specifies the number of input/output buffers you want to allocate for your file's database.

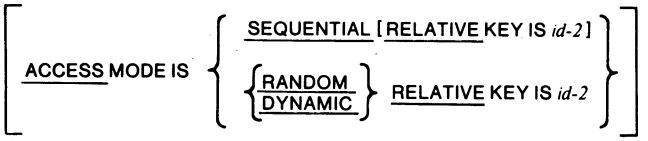
COBOL always ignores this clause.

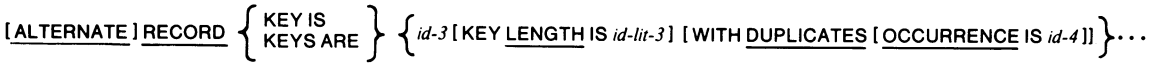
ORGANIZATION Clause



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. ORGANIZATION IS INDEXED defines an INFOS system file.

ACCESS MODE Clause





Where:

id-2 is an unsigned integer data item that specifies the relative record number of a key.

id-3 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.

id-lit-3 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.

id-4 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 *(id-2)* When randomly accessing an indexed file, COBOL accesses the record with the same value as the record key data item *(id-3)*.

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 *id-1* KEY *id-3, ...*

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

READ *id-1 [position phrase]* KEY *id-3*

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 *id-lit-3* have several functions depending on the context in which they appear:

1. When you OPEN a file for output, the value of *id-lit-3* (for the first *id-3* 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 *id-lit-3* at that time represents the number of characters (leftmost) in *id-3* 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 *id-lit-3* at that time represents the number of characters (leftmost) in *id-3* 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 *(id-3)* 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.

After executing a WRITE statement, COBOL updates the value of the occurrence number data item *(id-4)* 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 *id-5*] [INFOS STATUS *id-6*]

Where:

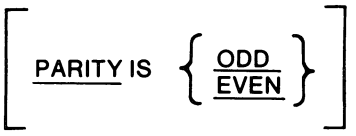
id-5 is a 2-character alphanumeric data item that receives the COBOL file status item.

id-6 is a 4-character, alphanumeric data item that receives the INFOS file status item which you must define in the Working Storage Section of the Data Division.

If you specify the FILE STATUS clause, you establish a data item *id-5* 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 *id-6* 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



COBOL always ignores this clause; parity is always odd.

ALLOW SUB-INDEX and LEVELS Clauses

[ALLOW SUB-INDEX [LEVELS IS *int-8*]]

Where:

int-8 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 index 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.

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. Note: Omitting the LEVELS clause causes the ALLOW SUB-INDEX clause to have *no effect*.

COBOL uses the ALLOW SUB-INDEX and LEVELS clauses to define the maximum number of subindex levels permitted. You can only set this option when creating an INFOS file. Do not redefine the maximum number of levels in an existing INFOS file if you expect to increase this maximum.

KEY COMPRESSION Clause

[KEY COMPRESSION]

Enabling KEY COMPRESSION saves space in INFOS system indexed files. Keys with duplicate suffixes use pointers to eliminate duplication of characters in the same subindex.

[DATA COMPRESSION]

Enabling DATA COMPRESSION saves space in INFOS system data files. The INFOS system compresses data records with duplicate information.

[COMPRESSION]

Specifying COMPRESSION enables both KEY and DATA COMPRESSION.

INDEX SIZE Clause

[INDEX SIZE IS *int-9*]

COBOL always ignores this clause.

DATA SIZE Clause

[DATA SIZE IS *int-10*]

COBOL always ignores 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:

```
I-O-CONTROL .

[ SAME { RECORD } AREA FOR id-1, ... ] ... [ MULTIPLE FILE TAPE CONTAINS { id-2 [ POSITION int ] } ... ] ... .
       { SORT   }
       { SORT-MERGE }
```

Where:

id-1 is a symbolic name that specifies a file.

id-2 is a symbolic name that specifies a labeled tape file or a system file specifier that specifies an unlabeled tape file.

int is an integer that specifies the position of a magnetic 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 second through n files begin at the same character position in memory as those declared for the first file, thereby redefining the record area declared for the first file.

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 six sections: the File Section, the Subschema Section, the Working Storage Section, the Virtual Storage Section, the Linkage Section, and the Screen 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.

Use the *Subschema Section* in conjunction with Data General Database Management System (DG/DBMS) databases. See Chapter 10 for details.

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.

The *Virtual Storage Section* places variables on disk during the run unit without the need for user controlled I/O. With this, you can have very large amounts of data maintained in your program.

The Virtual Common Section is a Virtual Storage Section that subprograms in the same run unit can use globally.

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 *Screen Section* displays CRT oriented screen formatted output. Variables in the Screen Section reference Working Storage variables.

The Data Division has the following format:

```
DATA DIVISION .

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

[ SUBSCHEMA SECTION .
  COPY subschema name. ]

[ WORKING-STORAGE SECTION.
  various record descriptions. ]

[ VIRTUAL-STORAGE SECTION.
  virtual data record descriptions. ]

[ LINKAGE SECTION .
  global record descriptions. ]

[ SCREEN SECTION .
  screen section body. ]
```

File Section

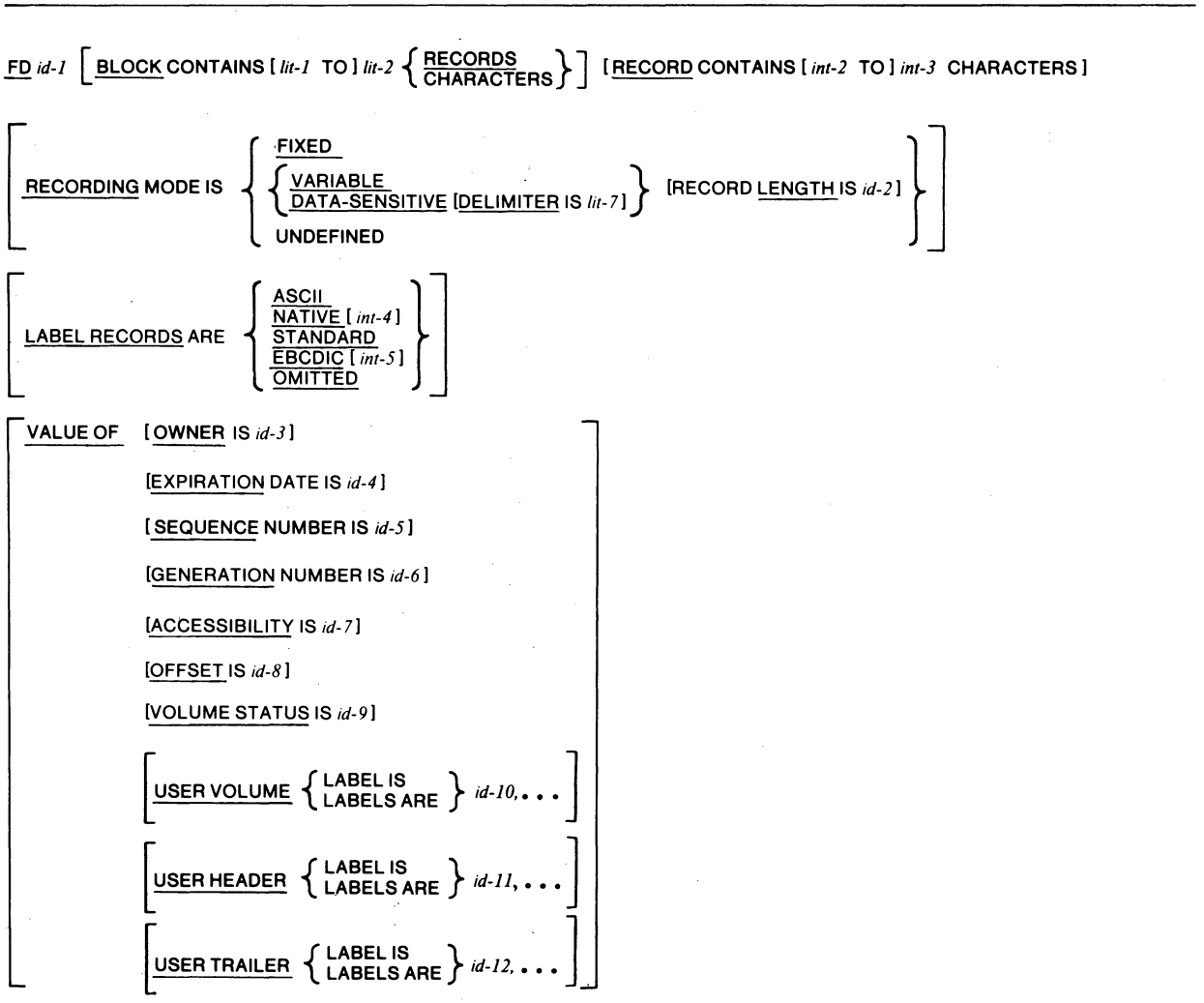
The File Section consists of FD and SD entries containing record descriptions which define the storage areas COBOL uses to transfer data to and from files. An FD or SD entry continues a file declaration that the SELECT clause began. You must specify one FD or SD entry for each file you declared in a 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 in this chapter.

File Description Entry

A file description entry may take one of four forms.

Format for a sequential file:



Format for a relative file:

```
FD id-1 [ BLOCK CONTAINS [ lit-1 TO ] lit-2  { RECORDS / CHARACTERS } ]

    [ RECORD CONTAINS [ int-2 TO ] int-3 CHARACTERS] [RECORDING MODE IS FIXED]

    [ LABEL { RECORD IS / RECORDS ARE } STANDARD / OMITTED ] [ DATA { RECORD IS / RECORDS ARE } id-13 , ... ]

    [FEEDBACK IS id-15 ] [PAD CHARACTER IS id-lit-5 ].

01 id-18 , ...
```

Format for an indexed file:

```
FD id-1  [ INDEX BLOCK CONTAINS [ lit-3 TO ] lit-4 CHARACTERS ] [ DATA BLOCK CONTAINS [ lit-5 TO ] lit-6 { RECORDS / CHARACTERS } ]

    [ INDEX NODE SIZE IS int-1 CHARACTERS ]  [RECORD CONTAINS [ int-2 TO] int-3 CHARACTERS]

    [RECORDING MODE IS VARIABLE [RECORD LENGTH IS id-2 ]]

    [ LABEL { RECORD IS / RECORDS ARE } { STANDARD / OMITTED } ]  [ DATA { RECORD IS / RECORDS ARE } id-13 , ... ]

    [FEEDBACK IS id-15 ] [MERIT IS id-16 ] [PARTIAL RECORD IS id-17 ].

    01 id-18 ,  ...
```

Format for a sort/merge file:

```
SD id-1  [ BLOCK CONTAINS lit-1 TO lit-2 { RECORDS / CHARACTERS } ]  [ RECORD CONTAINS [ int-2 TO ] int-3 CHARACTERS ]

    [ DATA { RECORD IS / RECORDS ARE } id-13 ,... ] .

01 id-18 , ...
```

In all formats, *id-1* is a symbolic name that specifies the name of the internal file you specified in the file's SELECT clause; and *id-18* is the name of an 01-level data record. *id-18* and *id-1* must have different names.

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.

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	Maximum size is one record.	
INDEX BLOCK CONTAINS			X		Maximum size is 2048 characters.	
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		X		Delimiters are CR, NL, FF, and NULL.	
LABEL RECORDS	X	X	X		ANSI Standard level 3.	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					

(continues)

Table 5-1. FD Entry Clauses

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						COBOL ignores this clause.
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 1.	
PARTIAL			X		None.	
PAD					Pad character is null.	COBOL ignores this clause.

(concluded)

Block, Node, and Record Sizes

```
[BLOCK CONTAINS [ lit-1 TO] lit-2 { RECORDS / CHARACTERS } ]

[INDEX BLOCK CONTAINS [ lit-3 TO ] lit-4 CHARACTERS ] [ DATA BLOCK CONTAINS [ lit-5 TO ] lit-6 { RECORDS / CHARACTERS } ]

[INDEX NODE SIZE IS int-1 CHARACTERS ]  [RECORD CONTAINS [ int-2 TO ] int-3 CHARACTERS ]
```

Where:

int-1, int-2 is a positive integer literal that specifies the maximum number of characters which a logical block in a sequential or relative file may contain.

lit-3, lit-4 is a positive integer literal that specifies the maximum number of characters which a logical block in an indexed file may contain.

lit-5, lit-6 is a positive integer literal that specifies the maximum number of characters or records which a logical block in a data file may contain.

int-1 is a positive integer literal that specifies the number of characters in an index node.

int-2, int-3 is a positive integer literal that specifies the number of characters in a data record.

The BLOCK CONTAINS 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.

The INDEX BLOCK CONTAINS clause specifies the size, in characters, of your index file's logical block. COBOL always ignores *lit-3*. The block size is equal to the number of characters specified in *lit-4*. If you omit this option, the default size is 2048 characters.

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 the size of one record.

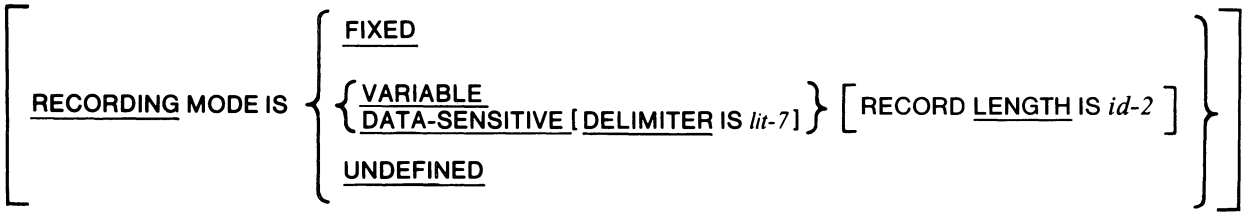
AOS permits the following block sizes (in characters):

for a tape (sequential files only):	up to 8192
for a disk (sequential or random files):	up to 8192
for indexed files:	2048 or 4096 only

The INDEX NODE SIZE clause specifies the size, in characters, of an index node in an indexed file. 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 *int-2* through *int-3*, inclusive.

RECORDING MODE Clause



Where:

lit-7 is an alphanumeric literal which specifies a character that delimits the end of a record, replacing the default delimiters.

id-2 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 *(id-2)*, 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.

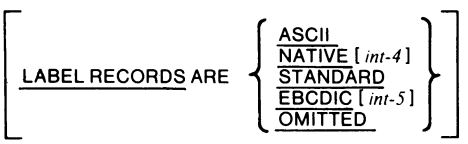
If you specify **DATA-SENSITIVE**, the length of a record is determined by the occurrence of some special character(s) *(lit-7)*. If you do not specify a delimiter character, the default characters are carriage return, line feed, null, or new line. You may also set a maximum length for a data-sensitive record by specifying the **RECORD LENGTH** clause. Count the delimiter as part of the record. COBOL uses the delimiter as the last character in the record. Your program must remove the delimiter when necessary.

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 read any section within the file, regardless of the individual records' size. For each READ operation, COBOL reads one physical block. You can only use UNDEFINED for an input magnetic tape file. *

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 *id-2*. On output in variable record format, *id-2* will specify the number of characters you want to write. On output in data-sensitive record format, *id-2* 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:

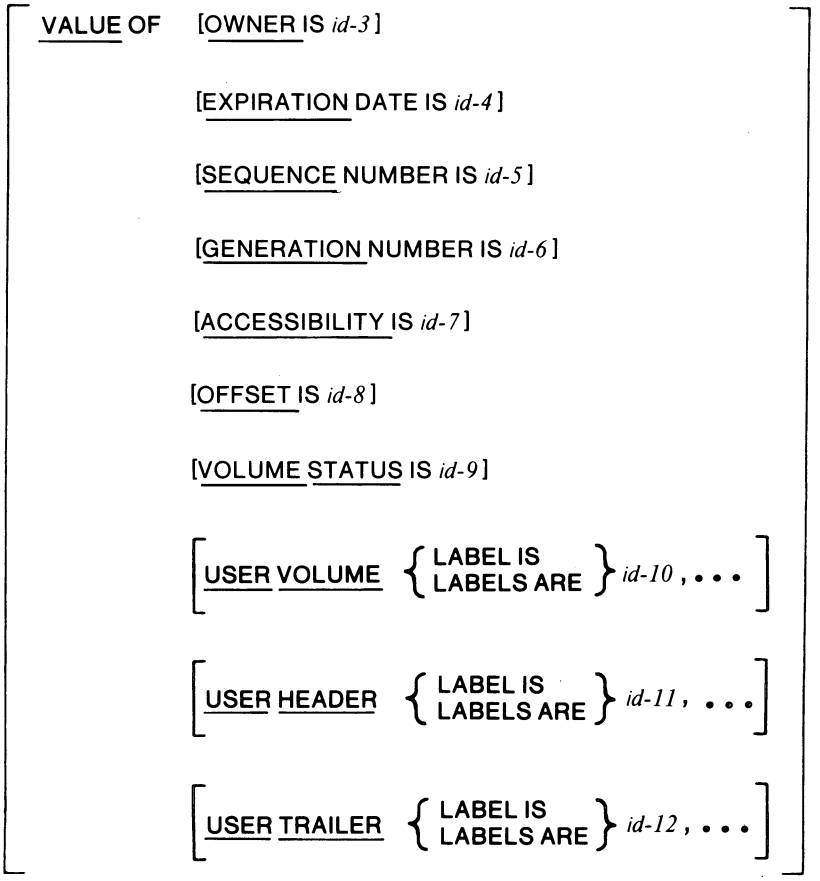
int-4 is a positive integer literal that indicates the level number of the tape, either 1, 2, or 3.

int-5 is a positive integer literal that indicates the level number of the tape, either 1 or 2.

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 *(int-4* and *int-5)* 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:

id-3 is a 14-character alphanumeric data item that specifies the owner identification (10 characters for EBCDIC).

id-4 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.

id-5 is a 4-digit unsigned numeric DISPLAY data item that specifies the number of a volume in a volume set.

id-6 is a 4-digit unsigned numeric DISPLAY data item that specifies a number to identify this file among successive generations of the file.

id-7 is a 1-character alphanumeric data item that specifies the restrictions placed on access to the file.

id-8 is a 2-digit unsigned numeric DISPLAY data item that specifies the number of characters you want ignored at the beginning of each block.

id-9 is a 4-character alphanumeric data item that receives the current status of volume and label processing.

id-10 is a 76-character alphanumeric data item that specifies volume label information.

id-11 is a 76-character alphanumeric data item that specifies file label header information.

id-12 is a 76-character alphanumeric data item that specifies file trailer label information.

You must declare *id-10, id-11,* and *id-12* contiguously in working storage.

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. COBOL always ignores the OWNER IS and SEQUENCE IS options. The OFFSET IS option does not apply to EBCDIC labels.

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 *id-9* 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 id-lit-1 LINES    [WITH FOOTING AT id-lit-2]  [LINES AT TOP id-lit-3]  [LINES AT BOTTOM id-lit-4] ]
```

Where:

id-lit-1 is a positive integer literal or an unsigned integer data item that specifies the number of lines in the page body.

id-lit-2 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.

id-lit-3 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.

id-lit-4 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 *id-1*. 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 *(id-lit-1)*,
- the footing *(id-lit-2)*,
- the top *(id-lit-3)*, and
- the bottom *(id-lit-4)*.

COBOL calculates the total logical size of a page as *id-lit-3* + *id-lit-1* + *id-lit-4*. No correspondence is established between a logical page and the physical size of the forms you are printing.

The size you specify for *id-lit-1* 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 *id-lit-2* and *id-lit-4*. 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 *id-lit-2* must be greater than zero, but less than or equal to *id-lit-1*. 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.

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.

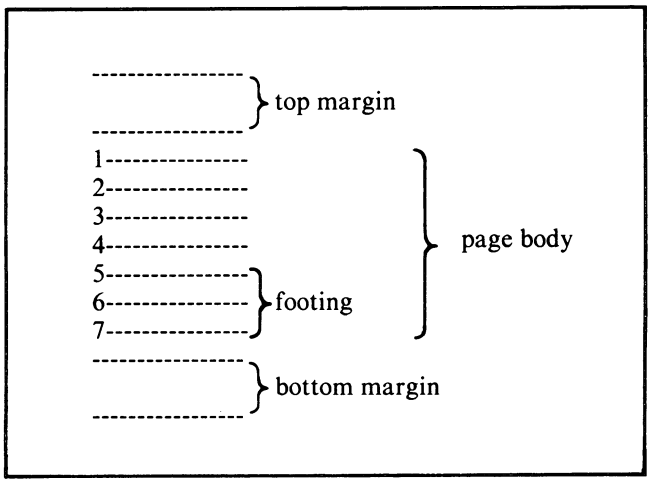
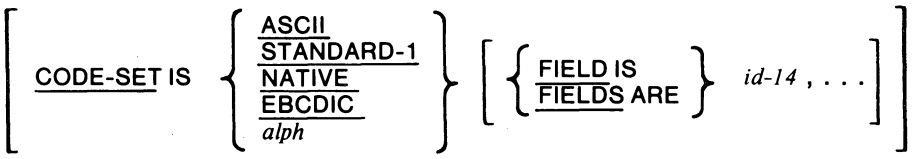


Figure 5-1. LINAGE Clause Example

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.

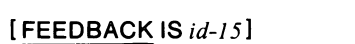
id-14 is an alphabetic, alphanumeric, unsigned numeric, or group data item that is contained in one of the record descriptions you declared for the file and that specifies the data fields you want translated.

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 E). 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. When you reference a field that is an array, COBOL will only translate the first data item. You must specify a higher level group name containing the array to reference the entire array.

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:

id-15 is a 4-byte data item that receives feedback information concerning the location of an indexed file's records.

The FEEDBACK clause allows you to access specially formatted feedback information concerning the location of records in an indexed 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 *id-15* with PIC X(4) for file FIL, then the following sequence of commands will insert the record in an inversion:

```
READ FIL.

(id-15 now contains the 4-byte block of information.)

WRITE FIL-INV INVERTED.
```

The system uses the information contained in *id-15* to write the inversion. *id-15* is unmodified.

You never need to explicitly reference the FEEDBACK item.

PAD Clause

[PAD CHARACTER IS *id-lit-5*]

Where:

id-lit-5 is a single-character alphanumeric literal or data item that should specify a character to fill unused portions of a logical block.

If you specify this clause, COBOL will always pad unused portions of your logical block with nulls (regardless of the character you specify).

MERIT Clause

[MERIT IS *id-16*]

Where:

id-16 is an integer data item that specifies the record.

This clause specifies the merit factor of a record. If your INFOS file has Optimized Record Distribution enabled, the INFOS system will then select the volume that matches the record merit factor according to INFOS system conventions. The selected volume is usually the volume with a merit factor equal to or closest to, but lower than, the record merit factor. The INFOS system places the record on the selected volume. For more information on Optimized Record Distribution, see the *INFOS System User's Manual (AOS)*.

PARTIAL RECORD Clause

[PARTIAL RECORD IS *id-17*]

Where:

id-17 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 in an INFOS system index file. This saves time because you can access this information without accessing the database file.

The size of the data item you specify in *id-17* determines the length any partial record can have. This length cannot be larger than the maximum size of the partial record set at index or subindex creation time (up to 255 characters). When COBOL accesses a data record for this file, it will return the partial record, if one exists, to *id-17*.

Working-Storage Section

Use the Working-Storage Section to describe data items that do not need an external file. Usually, you only need these items during program execution.

Working-Storage as well as File Section entries use a hierarchical structure. Levels, starting with 01, describe records. Levels "nest", that is, a lower-level item is a component (analogous to a substring) of the datanames physically above it in the source listing with lower-level numbers.

For example:

```
01 ALPHA
02 BETA PIC XX.
02 GAMMA PIC X(5).
```

defines the record shown in Figure 5-2.

The level numbers 01, 77, 66, and 88 have special meanings.

Level 01 defines a new record.

Level 77 defines a new record without any hierarchical structure permitted.

Levels 66 and 88 are defined later in this chapter.

You define the length of a record with its lowest level data items only. See "the Data Description Entry" later in this chapter for more details.

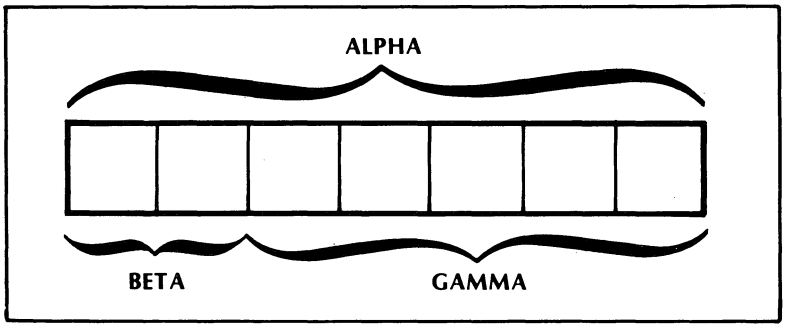


Figure 5-2. Internal Data Structure

Virtual-Storage Section

You use the Virtual-Storage Section if your program must handle large amounts of data, more data than you can fit into available memory. To use virtual data, you must specify a Virtual-Storage Section immediately after your Working-Storage Section. Essentially, the Virtual-Storage Section is a Working-Storage Section that the compiler breaks up into 2K-(or more) byte pages and stores in a file. When your program needs one of these data items, it moves the entire page containing the data into an area in memory reserved for this purpose. When you reference another data item, the page containing the new information overwrites the old page in memory.

All the rules that apply to an ordinary Working-Storage Section also apply to the Virtual-Storage Section. Please note these restrictions:

- Your elementary (lowest level) data items cannot be more than 32,767 characters each.
- You cannot access a group item greater than 32,767 characters.
- You cannot use a virtual data item in a SELECT clause, or in an FD or SD entry.
- You cannot use virtual data items in a CALL USING statement.

*

Specifying a Virtual-Storage Section automatically activates the feature; you need take no action at compile or bind time. However, you may get a runtime error as a result of your Virtual-Storage Section: ERROR IN SHARED PARTITION SET. We have to explain more about what the system is doing to tell you what this error message means.

When you specify a Virtual-Storage Section, the COBOL compiler creates a file called filename.VM. The compiler sets up all the section's initial values in this file. At runtime, the system must allocate to all the portions of your program the amount of main memory they will need. The system allocates what remains of your 64K bytes of available address space to Virtual-Storage in 2K-byte chunks. Then the system copies your .VM file into a temporary file called ?pid.CBL.VM.TMP. This temporary file, by the way, protects your virtual data from being accidentally modified by another user running the same program, and it may delay the start of your program's execution slightly while the system is creating it.

At compilation time, the COBOL compiler looks at all the commands in your COBOL program that require virtual data. The system then checks whether the space it has reserved for virtual data is large enough to contain the virtual data it needs to execute your program's commands. COBOL must be able to load all the data items involving any one command into memory simultaneously. For example, the MOVE statement requires that both the source and destination data areas fit into memory. Note that the space COBOL needs is not necessarily all the space implied by the statement. For instance, COBOL expands the statement MOVE A TO B, C, D into three statements: MOVE A TO B, MOVE A TO C, and MOVE A TO D. Therefore, COBOL does not require space in memory to fit all four data items simultaneously.

To further complicate matters, the largest data item will not necessarily be the item that requests the most space in your program. The COBOL compiler breaks up virtual data items into 2K-byte sections (exactly 2,048 bytes) contiguously in the source program list. The system places these overlays contiguously in a special file.

If one data item sits on the border of two pages, the compiler will place it on both pages. When the program references that "split" data item, the program will contiguously load both 2K-byte pages in their entirety. If that data item is not on a page border, the system will load only the page on which it resides. Therefore, the largest data items are essentially those which cross the largest number of page borders. Consequently, you will find that reordering your data items and placing dummy fillers in the Virtual-Storage Section may enable a program that previously failed to successfully run.

COBOL characters are all one byte long; however, COBOL always places level 01 and 77 data items on full word boundaries. This means that if you want to compute the placement of your data on its virtual pages, you must allow one byte per character and if a level 01 or 77 data item would have been on an odd-numbered byte, add 1. COBOL does not use the missing byte.

For example, suppose we have 4K bytes of available memory space for Virtual-Storage (4,096 bytes). We then put four group items in the section: ALPHA-RECPTS, WK-TOTL, NRECS, and MO-TOTL. ALPHA-RECPTS is a group item with a fixed size of exactly 2,000 bytes. WK-TOTL contains 50 bytes, NRECS has 1,500 bytes, and MO-TOTL also has 1,500 bytes.

If you place the four groups in the Virtual-Storage Section in their above order, WK-TOTL will overlap a page border by 2 bytes. ALPHA-RECPTS will occupy the first 2,000 bytes of page 0. (We call the first page page 0; ALPHA-RECPTS has the byte numbers 0 to 1,999). WK-TOTL will occupy the next 48 bytes of the first page (byte numbers 2,000 to 2047) as well as the first 2 bytes in page 1 (byte numbers 0 to 1). NRECS will then occupy the next 1,500 bytes in page 1 (byte numbers 2 to 1501), and MO-TOTL will overlap pages 1 and 2.

Thus, WK-TOTL and MO-TOTL both require 4K bytes of memory for the system to load them. WK-TOTL, the smallest physical item, needs the largest area of virtual memory space. Further, to load both MO-TOTL and WK-TOTL requires 6K bytes of memory (they share one page). Therefore, this program will generate an error at runtime (assuming only 4K is available). If, however, we create two dummy data items and insert them in the program, the program will run.

Let's call the first dummy data item DUMMY-1 and make it 48 bytes long. Now, we place this item between ALPHA-RECPTS and WK-TOTL. WK-TOTL will now reside entirely on page 1. Similarly, placing another dummy data item of 500 bytes between NRECS and MO-TOTL will place MO-TOTL entirely on page 2.

Now, we have "wasted" 548 bytes in our virtual data file. However, we can now run our program with only 4K bytes of available memory. Also, when the system needs a virtual data item, it never needs to read in more than one virtual data page. In reality, we did not actually waste any space, since the system will always read in a 2K-byte page whether it is full or not. In effect, all we did was fill an undefined empty area in page 2 with a defined empty area.

COBOL provides you with two utilities that can help you to optimize virtual data in your program: CSIZE and the /M switch.

CSIZE

Use this format to invoke the CSIZE utility:

CSIZE *programname* [.PR]

If your program is a COBOL program, CSIZE will return the following:

1. The number of shared pages your program uses.
2. The number of unshared pages your program uses.
3. The number of virtual pages in memory that your program requires to run.

If your program is not a COBOL program, CSIZE will still return 1 and 2, but instead of 3, it will return the message NOT A COBOL PROGRAM.

The CSIZE output looks like this:

The program file programname.PR uses m unshared and n shared o byte pages for a total of p pages.
The number of pages needed for virtual data is q. (m, n, o, p, and q are integers.)

The /M Switch

You may also specify the /M switch at compile time to check your data. The /M switch provides information about the location of all your data; it also tells you exactly how much your data overlaps a page boundary. See "The COBOL Map Switch" in Chapter 10 for details.

Keeping track of where your page borders are in this fashion is a complicated task at best. However, this should make you understand how a seemingly innocuous rearrangement or deletion of a data item from your Virtual-Storage Section may cause a runtime error. This problem should arise only when your program and your data are both very large. We suggest that a more efficient solution would be to use ANSI standard segmentation to shrink the amount of memory you need for your program.

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.

Screen Section

Data General's COBOL screen management Section allows you to control the position and appearance of the screens your program displays. You use a new Section in the Data Division, called the Screen Section, to do this. The Screen Section appears immediately after the Linkage Section.

The Screen Section defines the appearance of data items on the screen and the position of data item inputs from the screen. You must initially define each of these data items in either your Working-Storage Section, Virtual Storage Section, or File Section. (However, you need not define literal strings in these sections). The Screen Section refers to these data names and displays them in their specified positions. You can also specify:

- that the screen display will blink or the terminal bell will ring (audio tone) as the data items (or literals) are displayed,
- that the user must fill each entry field on the screen,
- that the user must entirely fill each entry field, and
- that the user need not strike NEW LINE after completing a field to move onto the next field.

You cannot reference an array in the Screen Section. The rules for a standard Data Division Section apply to the Screen Section. You can write all the clauses except the screen name, which must follow the level number, in any order.

Do not use V or P in the PICTURE clause of a screen item. However, you can use V or P in a screen item's corresponding working-storage item.

The DISPLAY and ACCEPT Commands

Once you have defined your screens in the Data Division, use the DISPLAY and ACCEPT commands to display the screen you defined and accept data from the screen, respectively. Specify the name you gave to the screen in the Screen Section as the argument for DISPLAY or ACCEPT. You can specify either an elementary or a group item; COBOL will follow normal principles for which items to use. Use screen items with ACCEPT and DISPLAY commands only. Do not use them with any other commands under any circumstances.

We discuss the DISPLAY and ACCEPT commands in Chapter 7. Read that chapter for a description of the full syntax and general use of the commands. The commands involve some special considerations when you use them with screens; we discuss them here.

DISPLAY

You will find descriptions of the FROM, TO, and USING clauses later in this section. DISPLAY acts in a special manner depending on these clauses.

When you DISPLAY a screen group or item, an implicit move occurs for each elementary item. If the elementary item is an output item, your program moves data from the identifier or literal referenced in your FROM id or USING id clause to the screen field. If the elementary screen item has a TO clause but no FROM clause, then your program will move underscores to the field. These implicit moves follow normal COBOL MOVE rules with two exceptions: we allow numeric edited to numeric and numeric edited to numeric edited moves.

When you DISPLAY a screen item that contains a FROM or USING data item clause, be sure to initialize data items before you DISPLAY them. Otherwise you will receive a runtime error.

COBOL always executes screen DISPLAYs without automatic line advancing.

ACCEPT

When your program performs an ACCEPT, an implicit move occurs for each elementary item; your program moves data from the screen field to the corresponding TO or USING data item. This implicit move follows normal COBOL MOVE rules with two exceptions: we allow numeric edited to numeric and numeric edited to numeric edited moves.

When your program executes a screen ACCEPT, it displays the values that are currently in the screen section's data names as defaults. The program then positions the cursor at the beginning of the first field, and moves on to successive fields when the user strikes a delimiter key. If the user does not enter new data, but strikes a delimiter, the program retains the default. If the user does not completely overwrite the default information, the program retains the remaining trailing characters in the data item. You can avoid these situations with the ESC, CR, and EOL keys. You can also use different delimiter function keys to control your program.

You may wish to use the ESCAPE KEY clause in the ACCEPT command within your program. The ESCAPE KEY clause represents a system-defined 2-digit data item. The number contained within the data item depends on which key was used to terminate the screen line. NEW LINE, ESCape, and the 8 unlabeled function keys across the top of a 6052 or 6053 keyboard are all valid terminators, generating different codes. Further, the function keys (not NL or ESC) will generate other codes if you strike the CTRL, SHIFT, or both keys along with a function key. Table 5-2 lists the terminators you can use and the codes they generate.

Table 5-2. COBOL Screen Section Function Delimiter Keys

Used with	Function Key							
	F1	F2	F3	F4	F5	F6	F7	F8
Fn. Key only	02	03	04	05	06	07	08	09
Shift key	10	11	12	13	14	15	16	17
Control key	18	19	20	21	22	23	24	25
CTRL & Shift	26	27	28	29	30	31	32	33
ESC generates code 01. CR, FF, or Tab generates code 00.								

Note that we use abbreviations in the text for some of the delimiter keys:

Key	Abbreviation
Escape	ESC
NEW LINE	NL
Form Feed	FF
Carriage Return	CR

The EOL, CR, ESC, and function keys perform special functions when using the screen. EOL will initialize the field. If you do not wish to use the default field, pressing EOL will clear the default value. CR is a delimiter that clears all the characters to the right of the cursor from the current cursor position. However, be careful when you use it. If you type CR at the beginning of a field, you will delete the default information and fill the field with blanks; also, the program will move on to the next field unless you are using the REQUIRED clause for the field. If you are using the SECURE clause for a field, COBOL does not permit you to use a default value; the program will always display underscores. Striking the ESC or a function key enters all the information for the current ACCEPT up to and including the current entry. The ACCEPT command then terminates; the program will not accept information for the rest of the fields specified in the ACCEPT, and control passes to the next executable statement.

Do not strike the BREAK key while executing a COBOL screen program; it will cause your process to hang.

Your program always executes screen ACCEPTs without automatic line advancing.

Syntax

Here is the full format for the Screen Section; an explanation of each clause follows it.

For Group Items:

{ *01 screen-name* [VIRTUAL] | *level-no screen-name* }

[AUTO]
[SECURE]
[REQUIRED]
[FULL]

For Literals:

{ *01 screen-name* [VIRTUAL] | *level-no screen-name* }

[BLANK SCREEN]
[BLANK LINE]
[BELL]
[BLINK]
[LINE [NUMBER IS [PLUS] *int*]]
[COLUMN [NUMBER IS [PLUS] *int*]]
[VALUE IS *lit*]

For an Elementary Item:

{ *01 screen-name* [VIRTUAL] | *level-no screen-name* }

[BLANK SCREEN]
[BLANK LINE]
[BELL]
[BLINK]
[LINE [NUMBER IS [PLUS] *int*]]
[COLUMN [NUMBER IS [PLUS] *int*]]

[PICTURE IS *format* { [FROM *id-lit*] | [TO *id*] | [USING *id*] }]

[BLANK WHEN ZERO]
[JUSTIFIED RIGHT]
[SECURE]
[AUTO]
[REQUIRED]
[FULL]

Where:

screen-name is any valid COBOL data name (that is not a reserved word) that specifies a screen name; if you omit the name, the item is a filler.

level-no is a level number ranging from 01 to 49; it behaves exactly as in the other Data Division Sections.

int is an integer that specifies a line or column position.

lit is a literal you want to display.

id-lit is a data item or literal that specifies an output field.

id is a data item that specifies an input field.

[VIRTUAL]

This clause (on an 01 item only) places the screen in Virtual-Storage. See the "Virtual-Storage Section" earlier in this chapter for details.

[BLANK SCREEN]

This clause clears the screen and positions the cursor to line 1, column 1. COBOL always executes this clause first regardless of the positions of the other attributes in the screen item. This occurs only when DISPLAYing a screen.

[LINE [NUMBER IS [PLUS] *int*]]

[COLUMN [NUMBER IS [PLUS] *int*]]

These clauses position the cursor on the screen. Four different possibilities exist for each clause:

1. No instruction
2. Only the command: LINE or COLUMN
3. The command with a number: LINE *m* or COLUMN *n*
4. The command with a plus: LINE PLUS *m* or COL PLUS *n*

Furthermore, using one form of one clause may affect the other clauses. Table 5-3 shows the relationships between and the results of the different possibilities for these clauses.

Note that the compiler resolves relative positions, therefore, line plus 1 refers to the previous item in the screen section, NOT the current cursor position. Items at level 01 default to line 1, col 1.

Table 5-3. COBOL Line and Column Positioning

LINE Clause	COLUMN Clause	Resulting Cursor Position On Screen
No Instruction	No Inst	No Change
	COL	Same Line, Column plus 1
	COL *n*	Same Line, Column *n*
	COL PLUS *n*	Same Line, Column plus *n*
LINE	No Inst	Line plus 1, Column 1
	COL	Line plus 1, Column plus 1
	COL *n*	Line plus 1, Column *n*
	COL PLUS *n*	Line plus 1, Column plus *n*
LINE *m*	No Inst	Line *m*, Column 1
	COL	Line *m*, Column plus 1
	COL *n*	Line *m*, Column *m*
	COL PLUS *n*	Line *m*, Column plus *n*
LINE PLUS *m*	No Inst	Line plus *m*, Column 1
	COL	Line plus *m*, Column plus 1
	COL *n*	Line plus *m*, Column *n*
	COL PLUS *n*	Line plus *m*, Column plus *n*

COBOL always executes this clause after a BLANK SCREEN clause, if one exists, and before any other clause on a command line, regardless of its position on the line.

[BLANK LINE]

This clause erases all characters on the same line as the cursor from the cursor's current position to the end of the line. Using this clause does not change the position of the cursor. COBOL always executes this clause after executing BLANK SCREEN and Line/Column positioning clauses, if any exist, regardless of its position on the line. This occurs only when DISPLAYing a screen.

[PICTURE IS *format*]

This half of the PICTURE clause is identical to the standard COBOL PICTURE clause. However, in the Screen Section you must use the PICTURE clause in conjunction with one of the three following clauses:

[FROM *id-lit]*

This clause specifies the field as an output field. When a DISPLAY clause that references this command line appears in the Procedure Division, COBOL will display the data specified by *id-lit* on the screen at half intensity (dim). If an ACCEPT clause in the Procedure Division references this command line, COBOL will ignore the entire Screen Section command line and continue execution of the other screen items.

[TO *id*]

This clause specifies the field as an input field. The clause follows rules for use that are opposite to those for the FROM clause. If you do not use a FROM clause on the same command line as the TO clause, when a DISPLAY clause that references this command line appears in the Procedure Division, COBOL will display underlines on the screen for the length specified by the Screen Section PICTURE clause at full intensity (bright).

Use an ACCEPT command to enter data from the screen into *id.*

[USING *id]*

This clause specifies the field for either input or output or both. The Procedure Division statement that references this clause determines its function. If an ACCEPT statement references it, the clause behaves in the same manner as the TO clause. If a DISPLAY statement references it, the clause behaves in the same manner as the FROM data-name clause, except that COBOL will display USING items at full intensity.

You can use both the FROM and TO clauses on the same command line. This will give you the equivalent of a USING clause with the ability to enter data into a different data name from the data name containing your screen output. You may not use the USING clause on the same command line as either a FROM or a TO clause.

[VALUE IS *lit*]

Use this clause to specify a literal that you wish to display. You must enclose the literal in quotes (literals are only alphanumeric fields). You cannot specify a PICTURE clause on the same command line with a literal string.

[BLINK]

This clause causes an output display to blink for elementary items. BLINK is not a group option. BLINK does not affect input fields.

[BELL]

This clause rings the terminal audio tone when DISPLAYing the field.

[SECURE]

As the user enters data into an input field, this clause echoes asterisks, instead of the actual characters, on the screen. COBOL echoes one asterisk for each character. If you use SECURE on an output field, COBOL will ignore the clause. If you use SECURE on a group item, all its subordinate items are SECURE. The user may not use a default value for a SECURE field.

[AUTO]

By specifying this clause, as soon as you fill an input field, the cursor will automatically move to the next field; you won't need to type NL. If there are no more input fields left, the ACCEPT statement automatically completes. If you use AUTO on an output field, COBOL will ignore the clause. If you use AUTO on a group item, all its subordinate items are AUTO.

[REQUIRED]

This clause requires you to enter at least one character into the input field. If you use REQUIRED on an output field, COBOL will ignore the clause. If you use REQUIRED on a group item, all its subordinate items are REQUIRED.

[FULL]

This clause requires you to fill the entire length of the input field, or COBOL will ignore your NL. If you use FULL on an output field, COBOL will ignore the clause. If you use FULL on a group item, all its subordinate items are FULL.

[BLANK WHEN ZERO]

This clause functions in the Screen Section in the same manner as it functions in the PICTURE clause.

[JUSTIFIED RIGHT]

This clause functions in the Screen Section in the same manner as it functions in the PICTURE clause.

Figure 5-3 is an example of a COBOL program using screen formatting. Please note that #W in this program is a CALL PROGRAM option. For further information, see the CALL statement in Chapter 7.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. LOGON.

* THIS IS THE STANDARD LOGON PROGRAM SUPPLIED WITH AOS COBOL.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.

DATA DIVISION.
WORKING-STORAGE SECTION.
77 LINE-NUMBER  PIC 99.
77 PROGRAM-NAME PIC X(20).
01 DATE-NOW.
        03      YY      PIC 99.
        03      MM      PIC 99.
        03      DD      PIC 99.
01 TIME-NOW.
        03      HH      PIC 99.
        03      MA      PIC 99.
        03      SA      PIC 99.
        03      FILLER  PIC 99.

SCREEN SECTION.
01 DATE-TIME-MENU.
        03 BLANK SCREEN " ".
        03 LINE 1 COLUMN 20 "AOS COBOL STANDARD LOGON PROGRAM".
        03 LINE PLUS 2 COLUMN 28.
        03 "DATE:".
        03 PIC Z9 FROM MM OF DATE-NOW.
        03 "/".
        03 PIC 99 FROM DD.
        03 "/".
        03 PIC 99 FROM YY.
        03 LINE COLUMN 28 "TIME: ".
        03 PIC Z9 FROM HH.
        03 ":".
        03 PIC 99 FROM MA.
        03 ":".
        03 PIC 99 FROM SA.
        03 LINE COLUMN 28 "TERMINAL NUMBER: ".
        03 PIC Z9 FROM LINE-NUMBER.
```

Figure 5-3. COBOL Program Using Screen Formatting (continues)

```
01 MENU-1.
        03 LINE 9 COLUMN 28.
        03 "RUN PROGRAM: ".
        03 PIC X(20) USING PROGRAM-NAME.
01 NO-PROGRAM-LINE.
        03 LINE 23 BELL "PROGRAM NOT FOUND" BLINK.
        03 COLUMN 75 VALUE " " BELL.

PROCEDURE DIVISION.
BEGIN.
        ACCEPT DATE-NOW FROM DATE.
        ACCEPT TIME-NOW FROM TIME.
        ACCEPT LINE-NUMBER FROM LINE NUMBER.
        DISPLAY DATE-TIME-MENU.

        MOVE SPACES TO PROGRAM-NAME.
        DISPLAY MENU-1.

        ACCEPT MENU-1.
        IF PROGRAM-NAME = SPACES THEN
                GO TO BEGIN.

        CALL PROGRAM PROGRAM-NAME.
        DISPLAY NO-PROGRAM-LINE WITH NO ADVANCING.
        CALL PROGRAM "#W".
        GO TO BEGIN.
        STOP RUN.
```

Figure 5-3. COBOL Program Using Screen Formatting (concluded)

CPRINT

The CPRINT utility allows you to print a hard copy of a screen image created with COBOL screen formatting. To print the screen image, your program must first send it to a filename. Use the command

DISPLAY *screen-name* UPON { *"filename"* / *dev* }

in your program. If you use *dev,* you must define a COBOL data name in your program in the Special-Names section that references an AOS filename. *dev* must take the form of a data name. This command causes your program to place a copy of the *screen-name* screen image in the file filename. However, filename will contain unprintable CRT-control characters; the line printer cannot format your screen correctly from this file. To create a file that you can print, specify this command from the AOS CLI:

CPRINT[/L= *listfilename]* *sourcefilename*

This command tells the CPRINT utility to take the unprintable CRT-control characters in *sourcefilename* and use them to format a printable output file. CPRINT places this output in *listfilename.* If you do not specify the /L switch, CPRINT outputs the file directly to @LPT.

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 and <200> thru <377>. It is stored as a string of consecutive 8-bit bytes.
- *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 and space. A space is equivalent to zero. 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 fewer digits more efficiently than it handles 17 or 18 digits.) This is the most efficient numeric data type.

A *decimal with trailing sign overpunch* datum is a string of not more than 18 ASCII digits. The sign is indicated in the right-most (low-order) digit according to Table 5-4. The sign is handled by the hardware; you never manipulate it. However, you may examine the sign character if you write it to a file. This is the most efficient signed data type.

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.

Table 5-4. 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 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 if the datum is signed or F if the datum is unsigned, 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 an undefined half-byte if there are an odd number of half-bytes (in order to align the string).

If you reference this data item as a numeric item, COBOL ignores the half-byte. Results are unpredictable if you reference this data item as a string of bytes or use it as a record key for an indexed (INFOS system) file. You can avoid this problem by always defining packed decimal items with an odd number of digits.

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 5-5.

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

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

$$\left\{ \begin{matrix} - \\ + \end{matrix} \right\} 9(i).9(f)E+99$$

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.

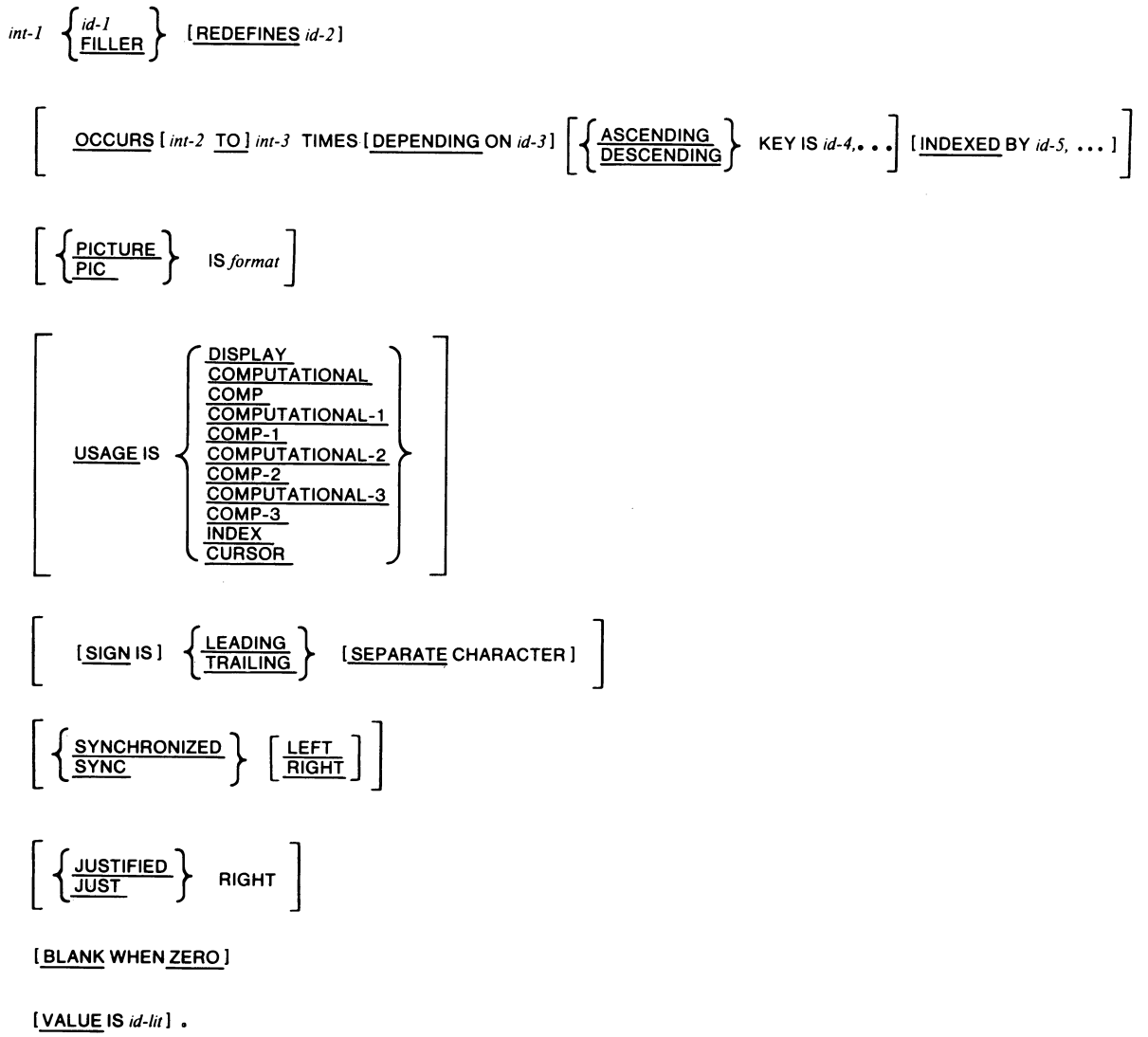
Table 5-5. Binary Number Storage

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
17 or 18	8

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:



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 *(id-1)* 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 *(id-1)* 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

Where:

int-1 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.

id-1 is a data name that specifies the item you are declaring.

You specify level numbers 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, and must have level numbers greater than the number used to describe the group item. 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 *id-2*]

Where:

id-2 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 *id-1*. In addition, the data item whose storage area you are redefining *(id-2)* must have the same level number as *id-1*. You may specify several successive redefinitions of *id-2*, each of which may reference either *id-2* itself or one of the items that redefines *id-2*. You may not declare any data item with the same level number as *id-2* and *id-1* between the declarations of *id-2* and *id-1* (except those items which are other redefinitions of *id-2)*.

When you use REDEFINES at the 01 or 77 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 01 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 *id-2* in the REDEFINES clause without subscript references.

When you specify REDEFINES, neither the declaration of *id-1* nor the declarations of any of its subordinate items may contain a VALUE clause. Also, you cannot declare an OCCURS DEPENDING clause for either *id-2, id-1,* 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 10 character positions for A. The entire storage is 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

Where:

int-2 is a positive integer literal used by the DEPENDING clause.

int-3 is a positive integer literal that specifies the number of occurrences of *dn* as an entry in an array.

id-3 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.

id-4 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.

id-5 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 specifies 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 *(int-3)*. Subscripted references may appear in the Procedure Division of your program (see the section "Array Name Qualification" in Chapter 6). All references to elements in 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 *id-3* (i.e., from the first through the *nth* entries, where *n* is the value of *id-3)*. 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 *int-2* and it must be less than *int-3*. When you execute your compiled program, the value of *id-3* must not be less than *int-2* nor greater than *int-3*. The value of *id-3* (or of *int-2)* 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 $id\text{-}5_1$, ... $id\text{-}5_n$ 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:

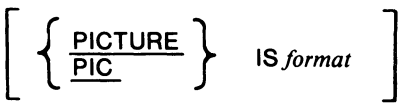
A:

	B(1)	B(2)	B(3)
C(1)	1,1	2,1	3,1
C(2)	1,1	2,2	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 itèm 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:

format is a valid combination of characters in the COBOL set. It specifies the size and type of the elementary data item *id-1*.

The PICTURE clause specifies the length and data type of the elementary data item *id-1*. 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, CURSOR, 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 As and the 9 as Xs, 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 SEPARATE 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 9s but not on both sides. The implicit decimal point is to the right of the rightmost P if there are Ps to the right of the 9s; it is to the left of the leftmost P if there are Ps to the left of the 9s. The V is redundant when used with Ps, 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 one to nine 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 (debit) 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 Xs) 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 As) 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 (debit), and $, the zero suppression characters Z and *, and the floating insertion characters +, -, and $.

Table 5-6 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 5-6. 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 asterisk (*)

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 negative.

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.

*

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 numeric character positions on both the left and right of the decimal point 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 5-7.

Table 5-7. Suppression Symbol Examples

Source Characters	Picture for Data Item	Data Item Resultant 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 $.

where n is greater than 1.

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.

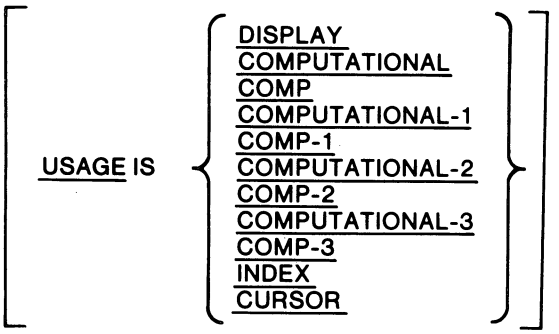
To avoid truncation, the minimum size of the picture string for a resultant data item must be the number of characters in the sending data item, plus the number of nonfloating insertion characters specified for that item, plus one for the floating insertion character.

You may not combine zero suppression editing with floating insertion editing. For examples of floating insertion editing, see Table 5-8.

Table 5-8. Floating Insertion Editing Examples

Source Characters	Picture for Data Item	Data Item Resultant 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
CURSOR	Free cursor for DG/DBMS

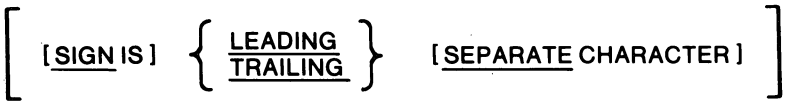
If you omit the USAGE clause, the default is USAGE IS DISPLAY. You may specify USAGE IS DISPLAY for alphabetic and alphanumeric data items, but the system will ignore it.

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 clauses in the data item's description entry.

SIGN Clause



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, internal floating point, 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

COBOL always ignores the SYNCHRONIZED clause.

JUSTIFIED Clause

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.

The JUSTIFIED clause applies only to the MOVE verb.

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 *id-lit*]

Where:

id-lit is an alphanumeric literal or a figurative constant that specifies the initial value of *id-1*.

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 requiring 8 bytes of storage.

```
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 *id-1* RENAMES *id-2* [{ THRU / THROUGH } *id-3*]

Where:

id-1 is a data name that specifies the item you are declaring.

id-2 is a data name that specifies a subordinate data item. It is part of the same record as *id-3*.

id-3 is a data name that specifies a subordinate data item. It is part of the same record as *id-2*.

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 *id-3*, the data item *id-1* is the name of a group data item containing all character positions from the first position of *id-2* through the last position of *id-3*.

However, no data item may be subordinate to *id-1* if you specify the RENAMES clause. If you omit *id-3*, *id-1* is simply an alternate name for *id-2*.

The character positions you define for *id-3* (if you specify it) must be to the right of those defined for *id-2*. 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 *id-1* (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 *id-1* itself as a qualifier.

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

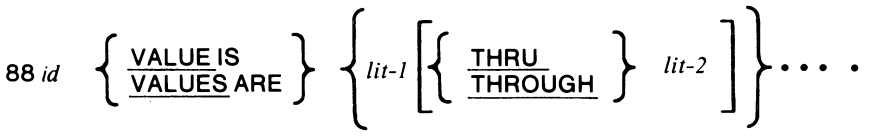
An example of data description entries which specify **RENAMES** entries is:

```
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.
```

In this example, Q is 2 characters long, redefining D and E. R is 3 characters long, redefining E, F, and H. S is 3 characters long, redefining B, D, and E.

The Condition Name Entry

Condition names provide a method for testing variables that gives greater readability in the Procedure Division. Set up a condition name by using a level 88 statement in the Data Division.



Where:

id is a condition name specifying a data item.

lit-1 specifies the minimum value for *id* that gives a "true" result.

lit-2 specifies the maximum value for *id* that gives a "true" result.

The condition *id* is true if the preceding data item falls within the range of the literal(s) specified.

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 declarative 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* refers to ANSI standard segmentation. The 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.* We describe all Procedure Division statements 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 id-1,...].
paragraph-1
paragraph-2
...
paragraph-n
```

A sectioned Procedure Division has the format:

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

Where:

id-1 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

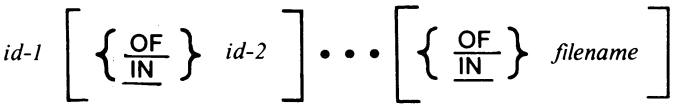
Paragraphs in the Procedure Division may have the same name if they appear in different sections. If you want to reference a paragraph that appears more than once in your program, you must qualify the paragraph name with the name of the section in which it appears. Use the format:

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, COBOL assumes you are referencing that paragraph in the current section.

Data Name Qualification

All datanames in the Working-Storage, Virtual-Storage, Screen, Subshema, 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:

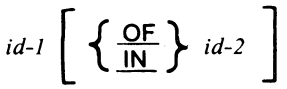
```
D OF E
D OF C OF B OF A
E OF A
```

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. The format to reference these items is:

*

$$\left\{ id \right\} (expr\text{-}1\ ,\ldots)$$

Where:

id is a data item that specifies a data name or condition name.

expr-1 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:

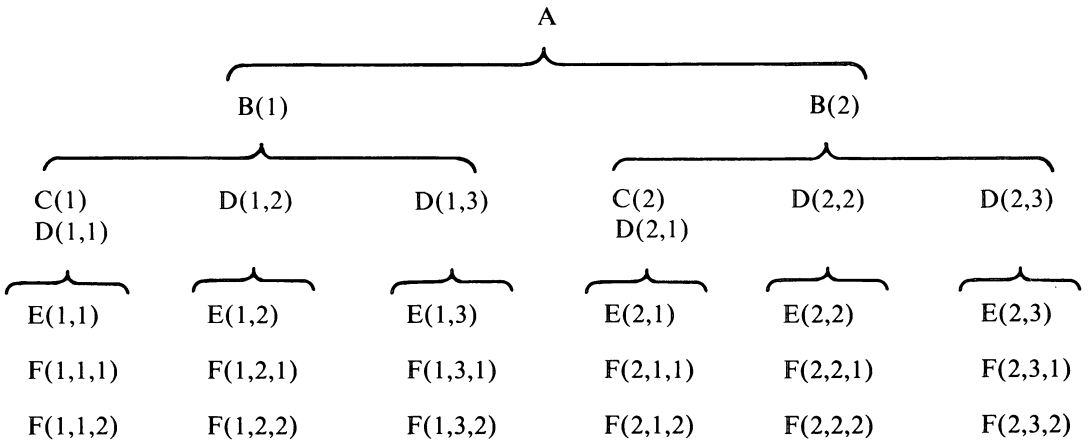
```
C OF B OF A (I,J).
```

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 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 18. 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 resultant 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 resultant item. COBOL treats the resultant 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 resultant 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 accuracy of the results that COBOL generates.

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 resultant 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 resultant 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 handles exponentiation using more than 18 digits in floating point notation. Rounding these results should improve accuracy.

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 resultant item, a size error condition occurs. In this case, COBOL leaves the resultant 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 resultant 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.

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 from left to right
4. Multiplication and division from left to right
5. Addition and subtraction from left to right

Examples of arithmetic expressions are:

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

Table 6-1. Arithmetic Operators

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

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:

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

Where:

expr-1, is an arithmetic expression, an alphanumeric literal, or an alphabetic or alphanumeric data item that *expr-2* 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:

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

Where:

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

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

If you specify POSITIVE, NEGATIVE, or ZERO, and *id* is numeric, the condition is true if *id* 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] □ *id*

Where:

id is the 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
NOT	Logical negation: the value is true if the condition is false, and false if the condition is true.
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.

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. "SUBPRO" must be the program-id of the 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 argument 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 are bound into the root context of the program file. Overlayable segments must share the same memory space, so COBOL brings segments into memory as needed. 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). *

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 Sequential 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 for a Sequential 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 B1 or B2, 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.

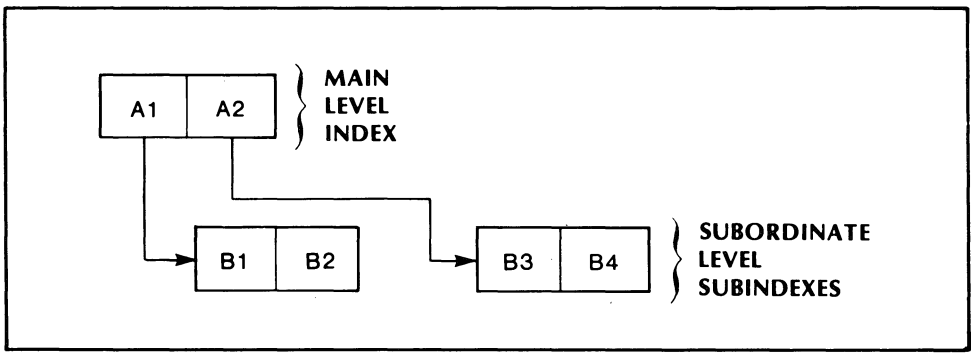
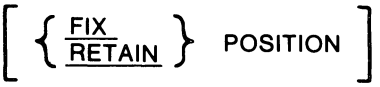


Figure 6-1. Multilevel Indexed File

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 DELETE statement always sets the record pointer to the record before the one just DELETEd. 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.

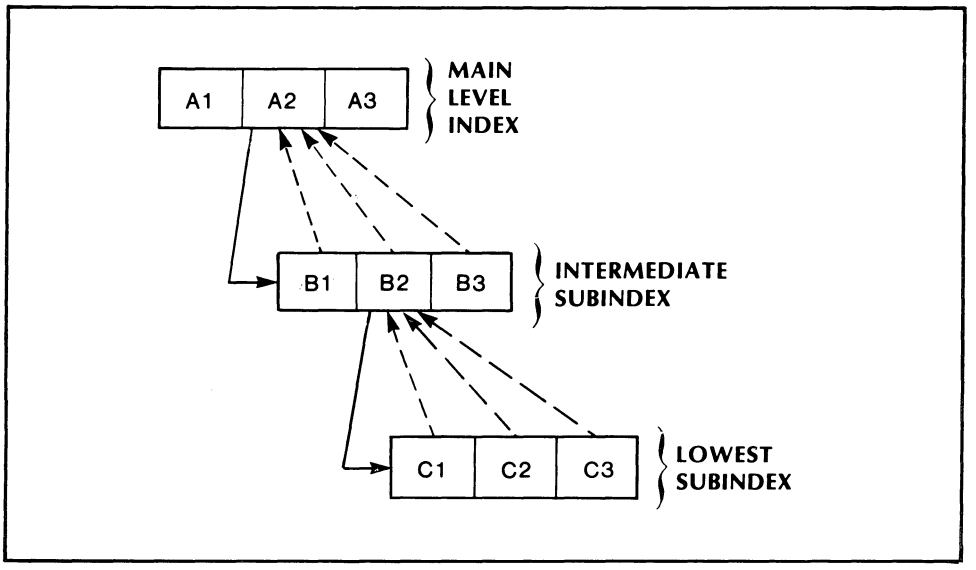
The Relative Option Phrase

Each time you access a key (and its associated database record), you can set a new current position at that key. Thus, by setting the current position with each processing operation, you may read a 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
  FORWARD
  BACKWARD
  UP
  DOWN
  UP FORWARD
  UP BACKWARD
  DOWN FORWARD
  STATIC ]
```

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).



SD-01087

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	C1
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:

Where:

id 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.

Table 6-3. Relative Access

Option	Use with		Position Without KEY Series	Position With KEY Series
	ISAM	DBAM		
FORWARD (or NEXT)	X	X	Next entry in current index.	--
BACKWARD	X	X	Previous entry in current index.	--
UP		X	Entry in higher level index that references current subindex.	Same as without key series, then apply key series.
DOWN		X	Position before the first record in the subindex.	Subordinate index referenced by current index entry, then apply key series.
UP FORWARD		X	UP, then next entry in that index.	--
UP BACKWARD		X	UP, then previous entry in that index.	--
DOWN FORWARD		X	First entry in subindex referenced by current index entry.	--
STATIC	X	X	Remain at current entry in current index.	Apply key series in current index.

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 or output the partial record associated with the referenced index entry. If you specify SUPPRESS DATA RECORD in a READ, REWRITE, or WRITE statement, execution will not input or output the data record associated with the referenced index entry.

You may suppress both partial record and data record information 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 no data; an entry that references only a subindex. Use record suppression to generate inversions for an indexed file.

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. The end-of-file condition occurs when you reach the end of any subindex. In a SEARCH statement, an end-of-file condition occurs when the index exceeds the highest permitted occurrence number of the array you are searching.

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. The two exceptions are if an end-of-file condition occurs in a statement containing an AT END phrase, or if an invalid key in a key series is encountered in a statement containing an 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.
section name.
```

The USE statement defines the conditions under which COBOL executes the associated Declaratives section(s). Each USE statement must be in a separate section. 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).

AOS COBOL permits referencing procedure names outside a section from inside that section and vice-versa. Exercise caution if you use this feature. This is a Data General extension to the ANSI Standard.

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 or file already open.
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 functionally organized summary of the procedure statements contained in this chapter.

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.
CALL PROGRAM	Chain to a specified program.
CANCEL	Restore subprogram to its initial state.
EXIT	Document end of PERFORM-type subroutine. No action performed.
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 or suspend execution of a program's run.

Console Input/Output

ACCEPT	Input data from a device.
DISPLAY	Output data to a device.

File Handling

CLOSE	Terminate processing of files.
DEFINE SUB-INDEX	Create subindex in indexed file.
DELETE	Remove record from indexed file.
DELETE FILE	Delete a disk file.
EXPUNGE	Delete a disk file (same as DELETE 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.
UNLOCK	Unlock all the records of a file your process locked.
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 { id-1 } [FROM id-lit ] [TIME-OUT AFTER id-2 SECONDS] [ON ESCAPE stmt ]
       { screen-name }
```

Where:

id-1 is a numeric or alphanumeric data item that receives the input. If *id-1* 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 *id-1* is not numeric, COBOL interprets the input data as an alphanumeric string.

screen-name is a COBOL Screen Section screen-name.

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

id-2 is a data item that specifies the number of seconds you want COBOL to wait for a response.

stmt is any COBOL imperative statement.

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 *id-1*, 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 *id-1* is larger than the transferred data, COBOL left justifies the data when storing it. If *id-1* is smaller than the transferred data, COBOL stores only the leftmost characters of the data and ignores those characters that do not fit.

If *id-1* is numeric, and the input data has more digits than are specified for *id-1*, 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.

If you specify the TIME-OUT clause, COBOL will wait *id-2* seconds for a response terminated by NEW-LINE. If the program does not receive a response in time, the ACCEPT terminates and control passes to the ON ESCAPE clause. COBOL will return code 99 in ESCAPE KEY. COBOL uses an extra task for the time-out routine.

LINE NUMBER is a 2-digit number representing the console at which the program is currently running.

ESCAPE KEY contains a 2-digit code generated by a termination key.

LINE NUMBER and ESCAPE KEY are designed primarily for use with COBOL screen management.

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 NEW LINE, 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 id FROM { DATE        }
               { DAY         }
               { TIME        }
               { LINE NUMBER }
               { ESCAPE KEY  }
```

Where:

id 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 *id* 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 an 8-digit number representing the time of day, based on a 24-hour clock. The form is HHMMSSOO, where HH is the hour, MM is the minute, SS is the second, and OO are hundredths of a second. For example, you would interpret 16350000 as 4:35 P.M. Hundredths of a second are always zero.

LINE NUMBER and ESCAPE KEY are useful with COBOL screen management. LINE NUMBER is a 2-digit number representing the terminal at which the program is currently running. ESCAPE KEY contains the 2-digit code generated by the last termination key you used.

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 id-lit, ... TO { id-1 [ROUNDED] }... [ON SIZE ERROR stmt]
```

ADD GIVING:

```
ADD id-lit, ... GIVING { id-2 [ROUNDED] }... [ON SIZE ERROR stmt]
```

ADD CORRESPONDING:

```
ADD { CORRESPONDING } id-3 TO id-4 [ROUNDED]  [ON SIZE ERROR stmt]
    { CORR         }
```

Where:

id-lit is a numeric literal or a numeric data item that specifies an addend.

id-1 is a numeric data item that specifies an addend and receives the result of an ADD TO operation.

stmt is an imperative statement to which control passes if a size error condition occurs.

id-2 is a numeric data item or a numeric edited data item that receives the result of an ADD GIVING operation.

id-3 is a group item containing addends that are numeric data items.

id-4 is a group item containing numeric data items corresponding to those in *id-3*.

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 *id-1* and storing the result in *id-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 *id-2* that follows the word GIVING.

An ADD CORRESPONDING statement adds data items in *id-3* to corresponding data items in *id-4* storing the results in the *id-4* 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 resultant 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 resultant item, then the resultant item remains unchanged. The ADD statement completes any remaining operations, stores all satisfactory results, and then transfers control to *stmt*. 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 TO [ PROCEED TO ] para-sect } . . .
```

Where:

para is the name of a paragraph in the Procedure Division that contains a single sentence consisting of a simple GO TO statement.

para-sect 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 *para-sect*.

> CAUTION: Do not use ALTER in new program development because this statement will most 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 { id-1  } [USING id-2,... ] [ON OVERFLOW stmt]
     { "lit" }
```

Where:

id-1 is an alphanumeric data item that specifies the program id of a COBOL subprogram you want to transfer control to.

lit is an alphanumeric literal that specifies the program *id* of a subprogram you want to transfer control to.

id-2 is a data item that is defined in the Working-Storage Section, File Section, or Linkage Section and that specifies a parameter you want to pass to the called subprogram.

stmt 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. The exception is that EXIT PROGRAM (see next paragraph) clears all outstanding PERFORM loops. 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 *stmt*. 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.

Examples

Example 1

```
PROGRAM-ID.    A-PROG.
   .
   .
   .
WORKING-STORAGE SECTION.

01 NUM PIC S9(5)V99 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, AGROUP.
   .
   .
   .



PROGRAM-ID.    SUBPROG.
   .
   .
   .
LINKAGE SECTION.
01 TABL.
   02 FILLER PIC X(10).
   02 TA OCCURS 150 TIMES.
      03 FILLER PIC X.
      03 T PIC 99.
01 N PIC S9(5)V99 COMP-3.
   .
   .
   .
PROCEDURE DIVISION USING N, TABL.
   .
   .
   .
```

Example 2

```
PROGRAM-ID. A-PROG.
   .
   .
WORKING-STORAGE SECTION.
01 NUM PIC S9(5)V99 COMP-3.
01 AGROUP.
   02 TAB-ITEM OCCURS 150.
      03 T PIC X.
      03 S PIC 99.
01 P-NAME PIC X(7) VALUE IS "SUBPROG".
   .
   .
PROCEDURE DIVISION.
   .
   .
   .
   CALL P-NAME USING NUM, AGROUP.
   .
   .
```

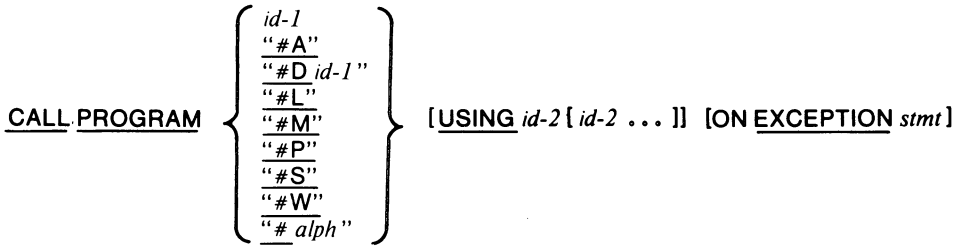
Example 3

```
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.
```

CALL PROGRAM

Chains to a specified program, overwriting the original program.

Format



Where:

id-1 is the name of the called program.

id-2 is a data item that is defined in the Working-Storage Section and that specifies a parameter which you want to pass to the called subprogram.

stmt is an imperative statement to which control passes if the calling program cannot find or execute the called program.

alph is any alphabetic character not listed among the *"# letter"* options.

Statement Execution

CALL PROGRAM functions in the same manner as the CALL statement, with the following exceptions:

1. EXIT PROGRAM will not return control to the calling program.
2. The CANCEL statement has no meaning to CALL PROGRAM; all called programs are always initialized with the CALL PROGRAM statement.

The CALL PROGRAM statement performs a chain to the called program. See the *AOS Programmer's Manual* for a description of chain functionality. If you wish to return to the original program, use another CALL PROGRAM statement in the new program.

When you return to the original program, it will have its initial values.

The *" # letter"* options in CALL PROGRAM represent system calls. They do not necessarily chain to another program. The meanings of these system calls are:

CALL	FUNCTION
"#A"	Chain to LOGON.PR.
"#M"	Chain to LOGON.PR.
"#P"	Chain to LOGON.PR.
"#L"	Chain to LOGON.PR.
"#H"	Equivalent to a COBOL STOP RUN statement.
"#S"	Equivalent to a COBOL STOP RUN statement.
"#W"	Pause program execution for 3 seconds.
" #D id-1"	Chain to *id-1* using the debugger.
" # alph"	No effect, control passes to the next executable statement.

CANCEL

Restores a previously called subprogram to its initial state.

Format

```
CANCEL id-lit, ...
```

Where:

id-lit is an alphanumeric literal or alphanumeric data item that specifies the program id of a subprogram you want to cancel.

Statement Execution

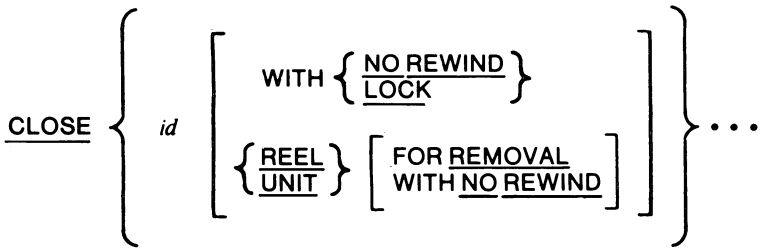
This statement resets all data items, the status and positioning of all files, and all alterable GO TO statements to the values they had before execution of the subprogram.

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.

CLOSE

Terminates the processing of files or file volumes.

Format



Where:

id 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.

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.

COBOL always ignores REMOVAL, REEL, UNIT and NO REWIND. COBOL never rewinds.

COMPUTE

Assigns the value of an arithmetic expression to one or more data items.

Format

```
COMPUTE { id [ROUNDED] }, ... = expr [ON SIZE ERROR stmt]
```

Where:

id is a numeric or numeric edited data item.

expr is any legal arithmetic expression (see the section "Arithmetic Expressions" in Chapter 6).

stmt is an imperative statement to which control passes if a size error condition occurs.

Statement Execution

A COMPUTE statement evaluates the *expr* and stores it in *id* 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 *expr* 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 1 to the rightmost digit in the resultant 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 resultant item, then the resultant item remains unchanged. The COMPUTE statement completes all operations, stores all satisfactory results, and then transfers control to *stmt*. 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 and B are both defined as whole numbers, and 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

```
DEFINE SUB-INDEX id-1

[ { FIX } POSITION ]  [ NEXT ]
                      [ FORWARD ]
                      [ BACKWARD ]
                      [ UP ]
                      [ DOWN ]
                      [ UP FORWARD ]
                      [ UP BACKWARD ]
                      [ DOWN FORWARD ]
                      [ STATIC ]

[ { KEY IS } { id-2 [ APPROXIMATE ] } ... ]
  { KEYS ARE }       [ GENERIC     ]

[FROM id-3 ]

[INDEX NODE SIZE IS int-1 ] [PARTIAL RECORD LENGTH IS id-4 ] [MAXIMUM KEY LENGTH IS int-2 ]

[KEY COMPRESSION] [ALLOW SUB-INDEX]  [ALLOW DUPLICATES]  [INVALID KEY stmt ]
```

Where:

id-1 is a filename that specifies an indexed file OPENed for output or I/O and SELECTed for ALLOW SUB-INDEX.

id-2 is an alphanumeric data item that specifies a record key associated with *id-1*.

id-3 is an alphanumeric data item which contains data in the form of an INFOS subindex definition packet and which is defined in working storage.

int-1 is an integer data item that specifies the size of a subindex node.

id-4 is an alphanumeric data item that specifies the partial record length for the subindex.

int-2 is an integer literal or an integer data item that specifies the maximum key length for a subindex.

stmt 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 *(id-2)* 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 *id-3* will define the subindex. *id-3* is 16 characters long. The second way to define the subindex is to specify the **INDEX NODE** phrase. Here you must define the node size *(int-1)* and the maximum key length *(int-2)* 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.

Be sure to specify the **ALLOW DUPLICATES** phrase if you intend to specify duplicate keys for the new subindex you are defining.

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 *stmt*. 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.
```

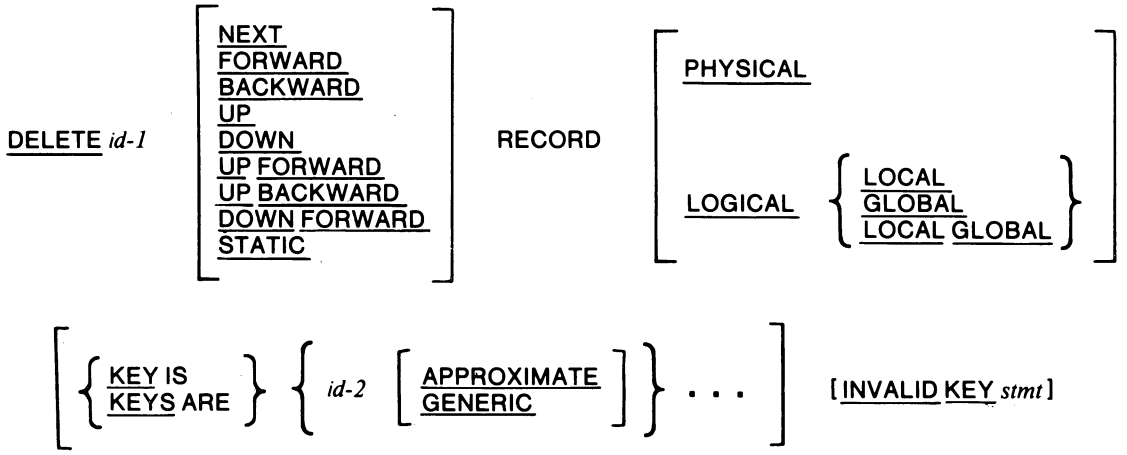
Example 2

```
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:

id-1 is a filename specifying an indexed file that you OPENed for I/O.

id-2 is an alphanumeric data item that specifies a record key associated with *id-1*.

stmt 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 *id-1* relative to the current position of the file's record pointer. If you omit this option, the default is the first (primary) key in the SELECT clause.

If you specify the KEY series phrase, you must have declared each key *(id-2)* in this file's SELECT clause.

You can delete only in a file OPENed for I-O.

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 User's Manual (AOS)*). 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 partial record 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 "COBOL file status Data Items" in Chapter 6). You can restore the partial record by using the UNDELETE statement described later on in this chapter.

DELETE (continued)

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 partial record and the data record.* You can restore the partial record 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. Be careful! You can still access and modify a logically deleted record. Be sure to check for file status code 96 in your program.

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 record pointer points to the record immediately before the deleted record.

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 *stmt*. If you omit this option, or if no invalid key condition occurs, control passes to the first executable statement following the DELETE sentence. See the Declaratives section in Chapter 6.

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.

DELETE FILE
Deletes disk files.

Format

DELETE FILE *id* , ...

Where:

id specifies an unopened disk file.

Statement Execution

For each *id* you name, AOS deletes all disk files named in the associated ASSIGN clause.

If *id* 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.

This statement is identical to the EXPUNGE statement.

DISPLAY

Makes low-volume data available for output to a specified hardware device or file.

Format

```
DISPLAY { id-lit      } ... [UPON dev] [WITH NO ADVANCING]
        { screen-name }
```

Where:

id-lit 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.

screen-name is a COBOL Screen Section screen-name.

Statement Execution

This statement transfers the source(s) *(id-lit)* to the output device or file in the order you specify them. It transfers them as a continuous 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 physical 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.

Formats

Simple DIVIDE -

```
DIVIDE id-lit-1 INTO { id-1 [ROUNDED] } ... [ON SIZE ERROR stmt]
```

DIVIDE GIVING -

```
DIVIDE { id-lit-1 INTO id-lit-2 } GIVING { id-2 [ROUNDED] } ... [REMAINDER id-3] [ON SIZE ERROR stmt]
       { id-lit-2 BY id-lit-1   }
```

Where:

id-lit-1 is a numeric literal or a numeric data item that specifies a divisor.

id-1 is a numeric data item that specifies a dividend and receives the quotient of a simple DIVIDE.

stmt is an imperative statement to which control passes if a size error condition occurs.

id-lit-2 is a numeric literal or a numeric data item that specifies a dividend.

id-2 is a numeric or numeric edited data item that receives a quotient.

id-3 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. You can only specify one ROUNDED clause if you specify the GIVING clause.

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 resultant 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 resultant item or if you attempt to divide by zero, then the DIVIDE statement completes all operations and transfers control to *stmt*. 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. Upon execution of an EXIT PROGRAM statement, the current state of the program remains the same except that it clears all active PERFORM loops.

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 *id, ...*

Where:

id is a filename that specifies an unopened disk file.

Statement Execution

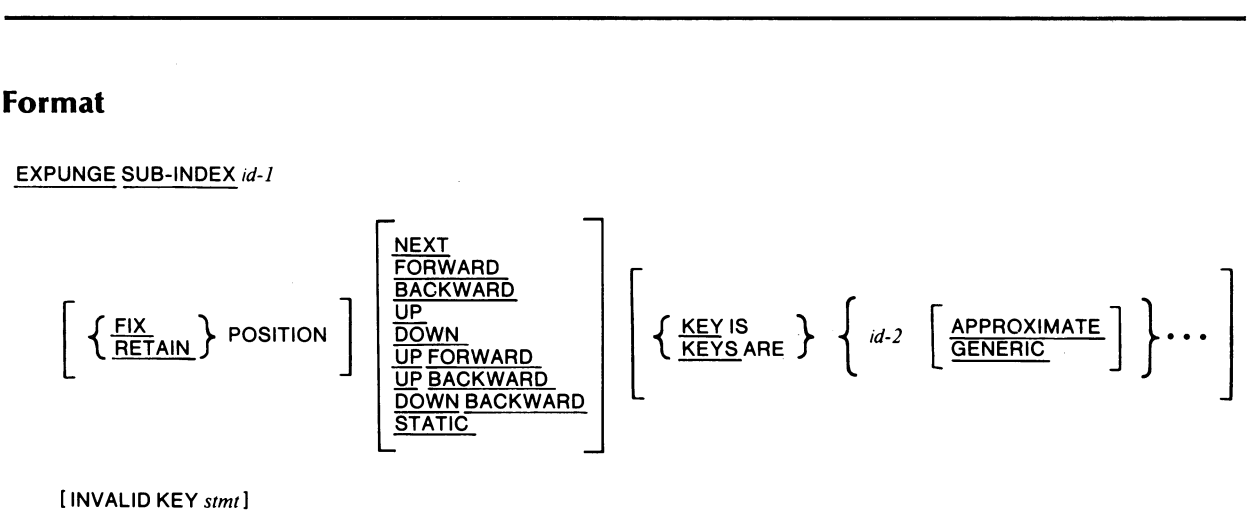
For each *id* you name, the operating system deletes all disk files named in the associated ASSIGN clause.

If *id* 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.

This statement is identical to the DELETE FILE statement.

EXPUNGE SUB-INDEX
Deletes a subindex from an indexed file.

Format



Where:

id-1 is a filename that specifies an indexed file OPENed for output or I/O and SELECTed for ALLOW SUB-INDEX.

id-2 is an alphanumeric data item that specifies a record key associated with *id-1*.

stmt 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 the first key in the SELECT clause.

If you specify the KEY series phrase, you must have declared each key *(id-1)* 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 *stmt*. 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.

Formats

Simple GO TO -

GO TO [*para-sect*]

GO TO DEPENDING -

GO TO *para-sect, ...* DEPENDING ON *int*

Where:

para-sect is a paragraph name or a section name.

int is an integer data item that specifies which *para-sect* you want to transfer control to.

Statement Execution

A simple GO TO statement transfers control to *para-sect* unless you previously modified this GO TO statement by an ALTER statement. If you do not specify *para-sect* 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 the *para-sect* argument that occupies the relative position corresponding to the value of *int*. 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 expr THEN [ { stmt-1 | NEXT SENTENCE } ] [ ELSE { stmt-2 | NEXT SENTENCE } ]
```

Where:

expr is any legal conditional expression.

stmt-1, stmt-2 is any legal Procedure Division statement, including another IF statement.

Statement Execution

A true *expr* transfers control to *stmt-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 *expr* transfers control to *stmt-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.

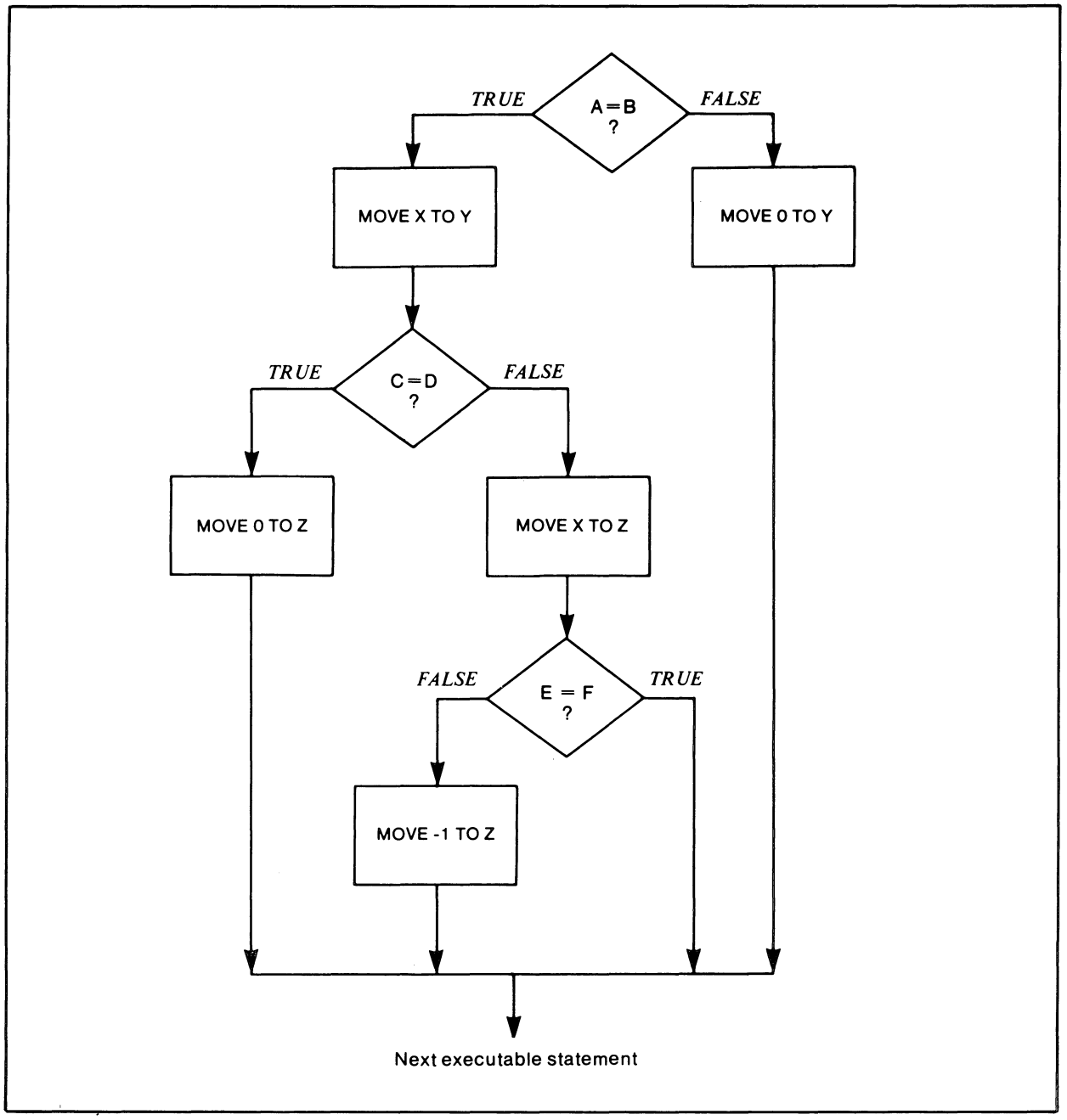
If the ELSE NEXT SENTENCE clause would be the last clause in the IF sentence, you need not specify it.

*

Example

```
IF A = B THEN
   MOVE X TO Y,
   IF C = D THEN
         MOVE 0 TO Z,
      ELSE
         MOVE X TO Z,
         IF E = F THEN
            NEXT SENTENCE,
         ELSE
            MOVE -1 TO Z.
ELSE
   MOVE 0 TO Y.
```

Figure 7-1 shows how the above IF statement works.



SD-01086

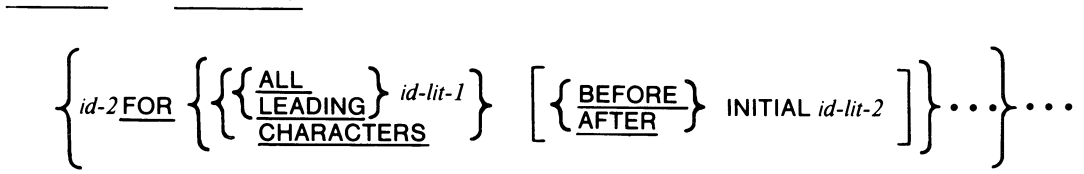
Figure 7-1. IF Statement Example

INSPECT

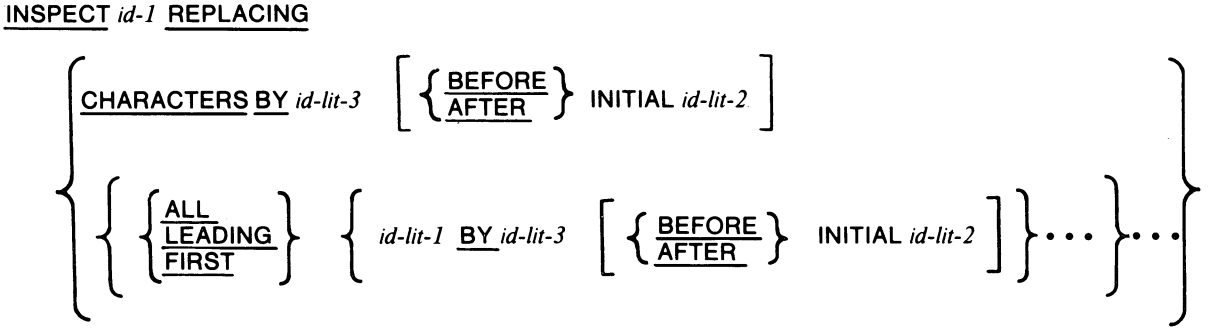
Counts, replaces, or counts and replaces occurrences of specified character strings in a data item.

Formats

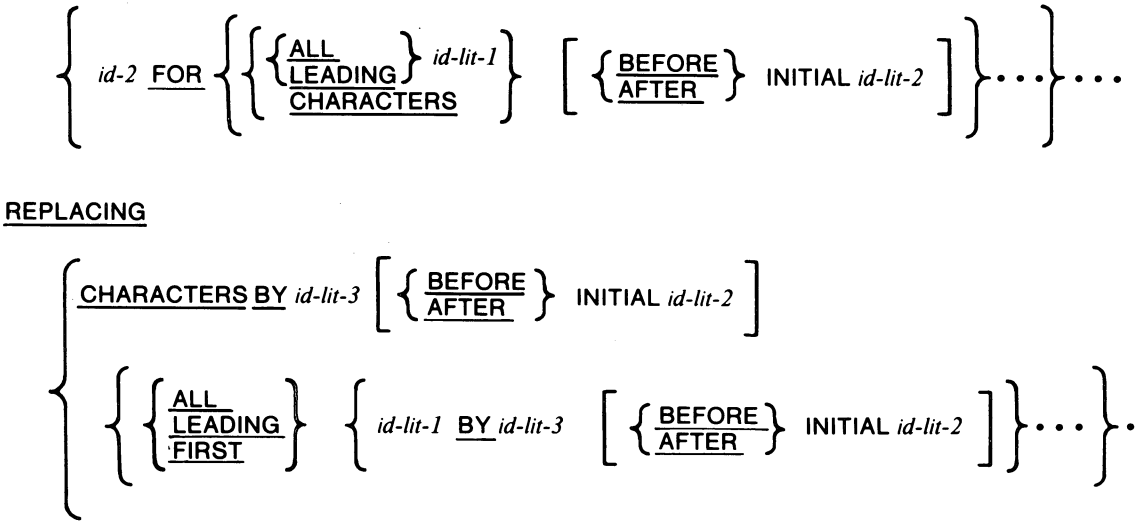
INSPECT TALLYING -



INSPECT REPLACING -



INSPECT TALLYING and REPLACING -



Where:

id-1 is an alphanumeric or numeric data item that specifies the item you want to inspect.

id-2 is a numeric data item that the INSPECT statement increments every time it finds a matching string.

id-lit-1 is an alphanumeric literal or a numeric data item that specifies the character string you want to match.

id-lit-2 is an alphanumeric literal or a numeric data item that specifies an operation delimiter.

id-lit-3 is an alphanumeric literal or a numeric data item that specifies a replacement string.

Each *id-1, id-lit-1, id-lit-2,* and *id-lit-3* 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 *(id-1)* for a character string(s), which may occur between boundaries determined by a delimiter you may optionally specify *(id-lit-2).* INSPECT compares this string to a string you specify *(id-lit-1)* and, if there is an exact match within the boundaries, then INSPECT 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 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.

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 *id-lit-1* LEADING *id-lit-1* 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.

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:

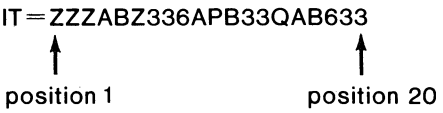


Table 7-1 shows the table-like structure that COBOL sets up before it scans IT and tallies the appropriate strings.

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

In the first operation of the preceding 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, the first character in 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 *id-1* from left to right. It considers the operations in the statement in the order you specify them. Each operation is eligible for consideration only within the boundaries that COBOL determines for it before the comparison cycle takes place (as shown previously in Table 7-1).

The comparison cycle begins by considering the leftmost character in *id-1* for a match, then continues scanning character by character until it considers the rightmost character in *id-1* 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 in 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 *id-lit-1* INSPECT REPLACING replaces each match with the associated *id-lit-3*. The lengths of each pair of *id-lit-1* and *id-lit-3* must be the same.

If you specify the FIRST option, INSPECT REPLACING replaces only the first occurrence of *id-lit-1* in *id-1*.

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 PIC 99 COMP VALUE IS 0.
01 I2 PIC 99 COMP VALUE IS 0.
01 I3 PIC 99 COMP VALUE IS 80.
01 I4 PIC 99 COMP VALUE IS 0.
01 I5 PIC 99 COMP VALUE IS 0.
01 I6 PIC 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".
```

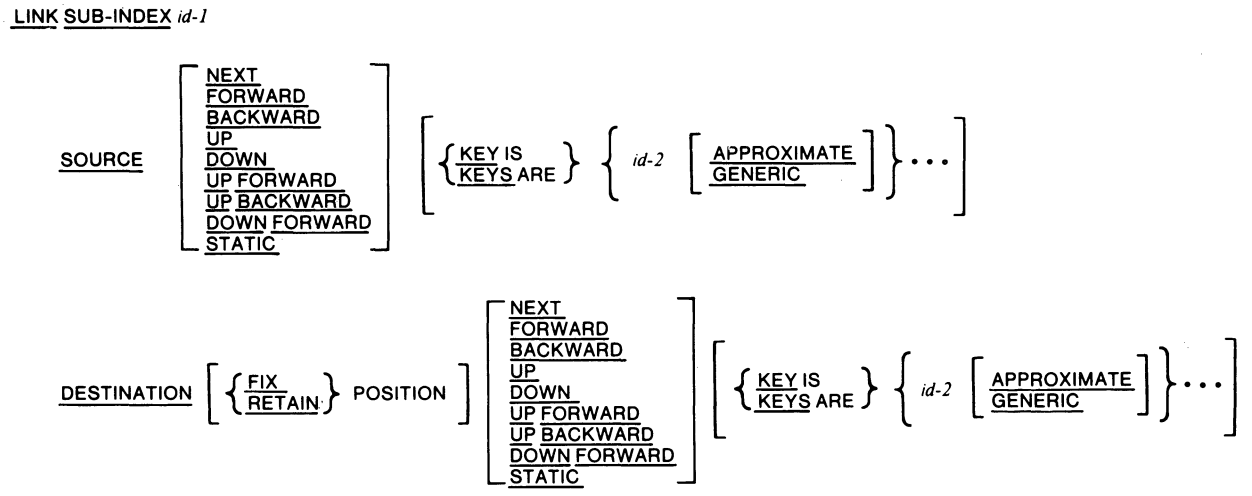
The results of this example are:

```
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:

id-1 is a filename that specifies an indexed file OPENed for output or I/O and SELECTed for ALLOW SUB-INDEX.

id-2 is an alphanumeric data item that specifies a record key associated with *id-1*.

stmt 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 the first key in the SELECT clause.

If you specify the KEY series phrase, you must have declared each key *(id-2)* 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 *stmt*. 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

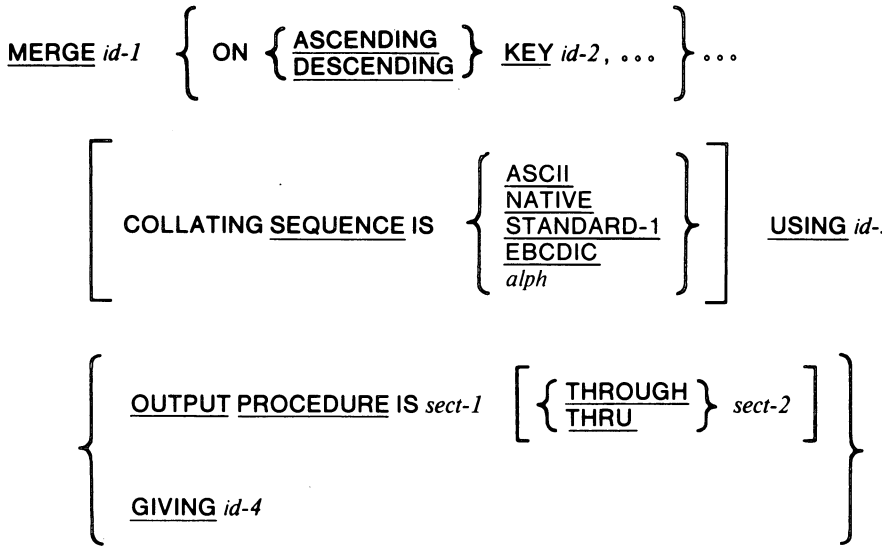
```
LINK SUB-INDEX FILE1 SOURCE
   DESTINATION DOWN FORWARD
   INVALID STOP RUN.
```

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:

id-1 is a filename that specifies a sort/merge file described in a sort/merge file description entry in the Data Division.

id-2 is an alphanumeric data item that specifies a key described in records associated with *id-1*.

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.

id-3 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.

sect-1, sect-2 is a Procedure Division section name that specifies an output procedure executed as part of the MERGE operation.

id-4 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 *id-4* 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 sequence specifiers have the same meanings as the specifiers used in the PROGRAM COLLATING SEQUENCE clause of the Environment Division (see the section "The 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 or paragraphs 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 *id-1*. The words THROUGH and THRU are equivalent.

If you specify the GIVING clause, control transfers all merged records from *id-1* to *id-4*.

The logical records in the *id-3* and in *id-4* must be equal in size to the logical records in *id-1*.

MERGE statements may appear anywhere in your program's Procedure Division except in the Declaratives section 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.

Formats

Simple MOVE -

```
MOVE id-lit TO id-1, ...
```

MOVE CORRESPONDING -

```
MOVE { CORRESPONDING } id-2 TO id-3
     { CORR         }
```

Where:

id-lit is any data item or literal that specifies a data source you want to move.

id-1 is any data item that receives a data source.

id-2 is a group data item that specifies data sources you want to move.

id-3 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 *id-1* 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 *id-2* to corresponding data items in *id-3*.

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 "The 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.

Table 7-2. MOVE Rules

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			

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.

Formats

Simple MULTIPLY -

```
MULTIPLY id-lit-1 BY { id-1 [ROUNDED] } ... [ON SIZE ERROR stmt]
```

MULTIPLY GIVING -

```
MULTIPLY id-lit-1 BY id-lit-2 GIVING { id-2 [ROUNDED] } ... [ON SIZE ERROR stmt]
```

Where:

id-lit-1 is a numeric literal or a numeric data item that specifies a multiplicand.

id-1 is a numeric data item that specifies the multiplier and receives the result of a simple MULTIPLY operation.

stmt is an imperative statement to which control passes if a size error condition occurs.

id-lit-2 is a numeric literal or a numeric data item that specifies a multiplier.

id-2 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 *(id-lit-1)* by each multiplier/result *(id-1)* and stores each result in the corresponding *id-1* 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 *(id-2)* according to MOVE rules.

If you specify ROUNDED and COBOL truncates the result of this operation to fit the given resultant item, it rounds in the following manner. COBOL adds a 1 to the rightmost digit in the resultant 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 *stmt*. 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

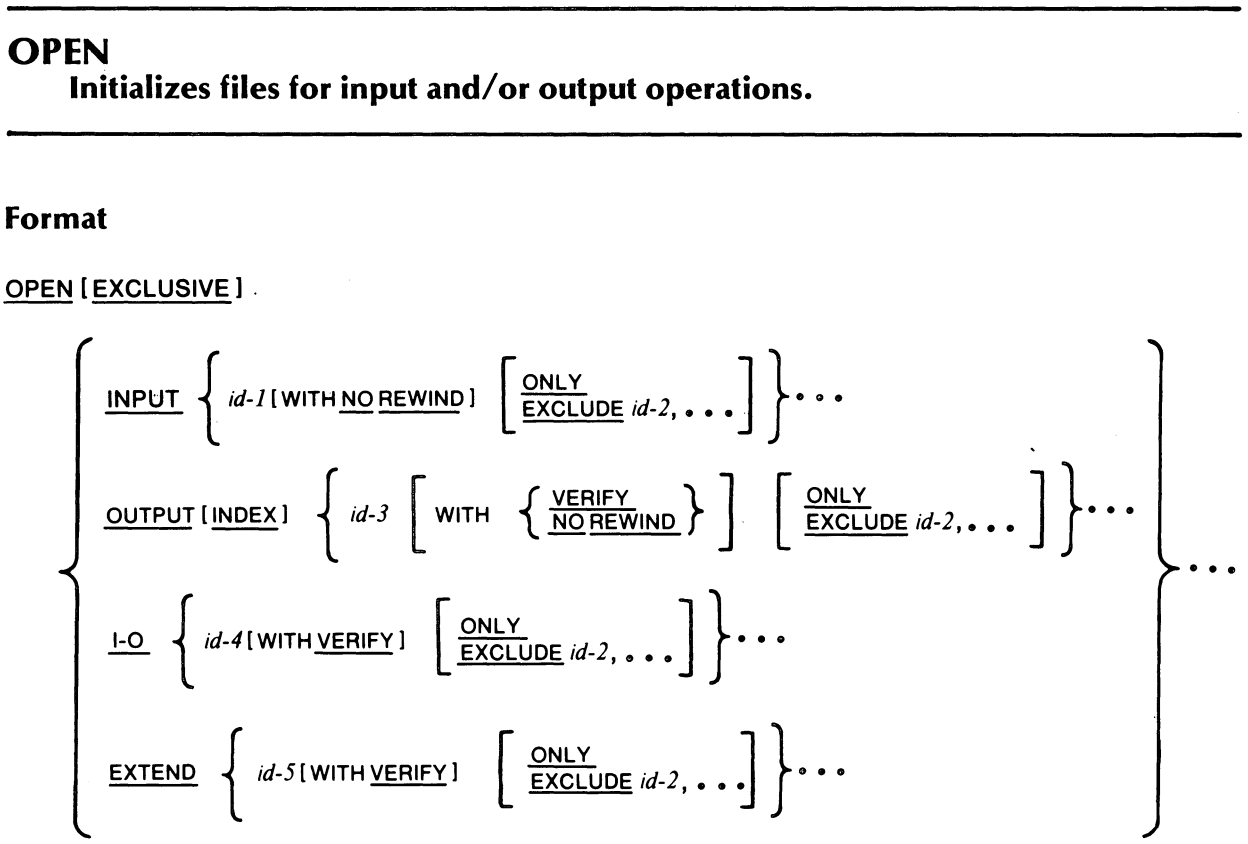
```
MULTIPLY A BY B GIVING C, D.
```

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:

id-1 is a filename that specifies a file you want to open for input.

id-2 is a filename that specifies an index you want to suppress buffer space for.

id-3 is a filename that specifies a file you want to create for output.

id-4 is a filename that specifies a file you want to open for input and output.

id-5 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 "COBOL File Status Data Items" 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.

INFOS System users note: OPENing an indexed file automatically performs a down-forward movement positioning you on the kirst key in the top level index if the ACCESS MODE is SEQUENTIAL DYNAMIC. (The INFOS system starts users above their top level index.)

COBOL always ignores the ONLY, EXCLUDE, VERIFY, and NO REWIND options.

Examples

Example 1

```
OPEN I-O MAINTAIN.
```

This statement opens the file named MAINTAIN for input and output.

Example 2

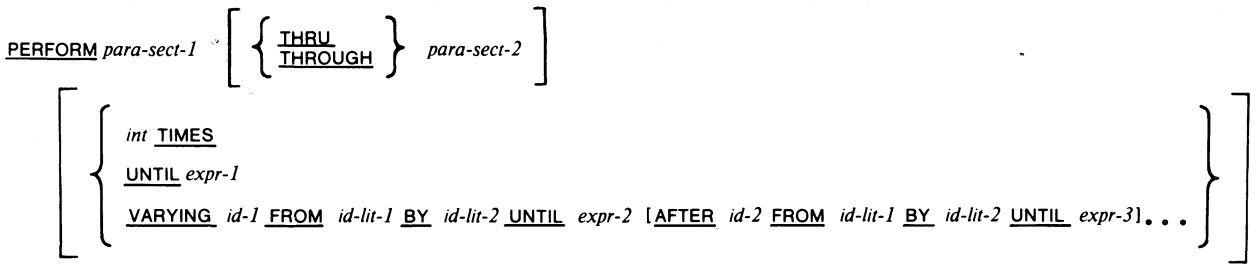
```
OPEN EXCLUSIVE INPUT SAMFILE.
```

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.

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



Where:

para-sect is the name of a procedure paragraph or section you want to transfer control to.

int is an integer literal or an integer data item that specifies the number of times you want to execute the range of the PERFORM.

expr is any legal conditional expression (see the section "Conditional Expressions" in Chapter 6).

id is a numeric data item that specifies a counter.

id-lit-1 is a numeric literal or a numeric data item that specifies an initial value.

id-lit-2 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 *para-sect-1* to the last statement in *para-sect-2* (if you specify it). If you omit *para-sect-2* the last statement in *para-sect-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 *int*. If *int* 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 *int* is a literal, it must not exceed 32767.

If you specify the UNTIL clause, the PERFORM statement executes the range of the PERFORM until the value of *expr* 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.

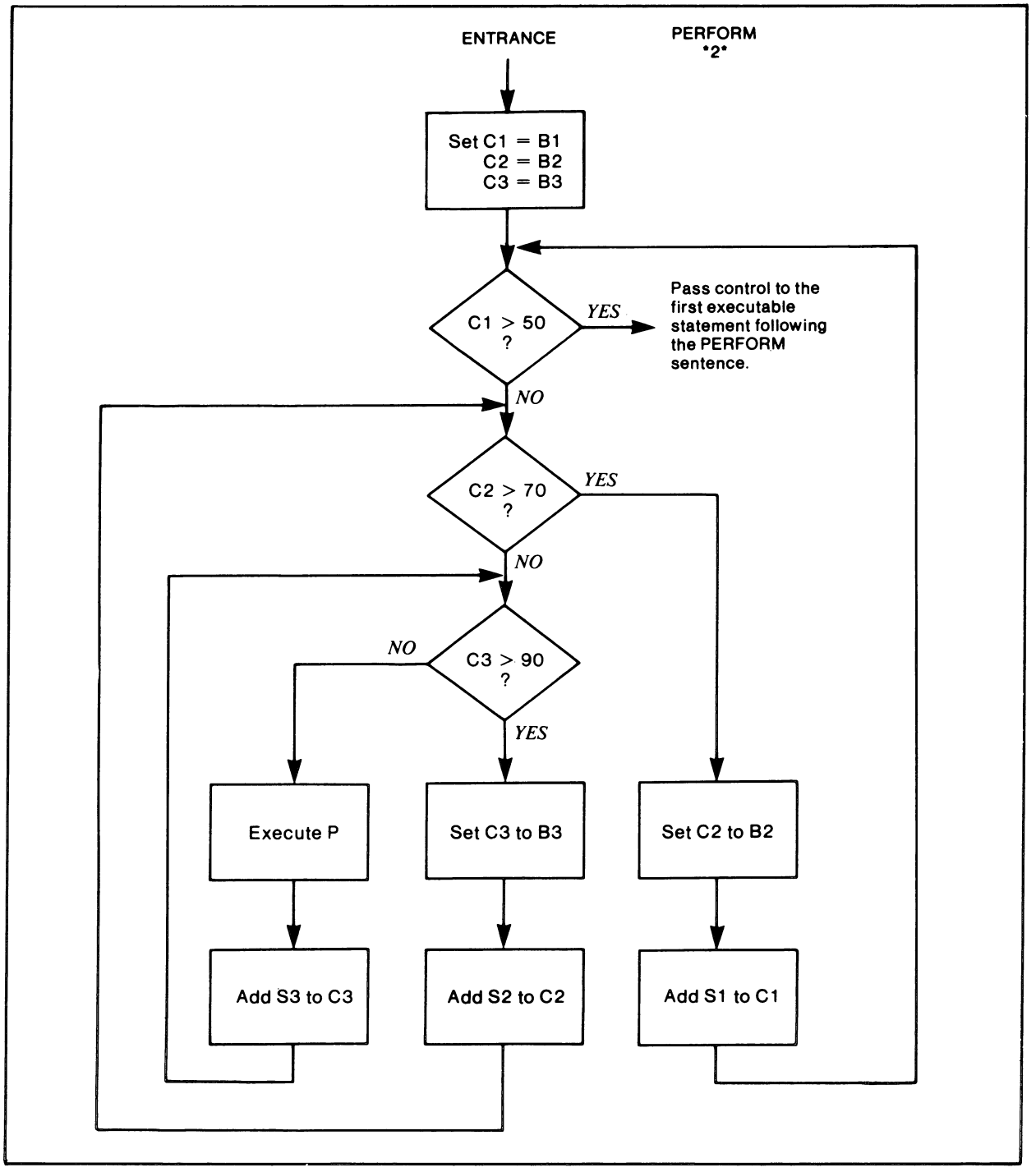
As soon as the *expr* in the VARYING phrase is true, control passes to the first executable statement following the PERFORM sentence. If the condition is false, COBOL executes the range of the PERFORM unless you specify an AFTER phrase or phrases. In this case, COBOL evaluates the condition in the first AFTER phrase. If it is true, COBOL re-evaluates 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.

PERFORMS may follow complicated interlocking structures; for example, a PERFORM loop may include another PERFORM loop and share code. However, these structures are dangerous and unclear. We do not recommend using them. 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.



SD-01085

Figure 7-2. PERFORM Example

READ for a Sequential File

Makes a record available from a sequentially organized file.

Format

READ *id-1* NEXT RECORD [LOCK / UNLOCK] [INTO *id-2*] [AT END *stmt*]

Where:

id-1 is a filename that specifies a sequential file OPENed for input or I/O.

id-2 is any data item specifying a destination that receives the file's record area.

stmt 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 *id-2* 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 AT END option and the READ operation reaches the end of the file, control passes to *stmt*. If you omit this option, or if no end-of-file condition occurs, control passes to the first executable statement following the READ sentence. See "The Declaratives Section" in Chapter 6.

COBOL always ignores the LOCK option.

Example

```
READ INVENTORY LOCK INTO WORKAREA.
```

This statement reads the next record in INVENTORY and moves it into WORKAREA.

READ for a Relative File

Makes a record available from a relative file.

Format

```
READ id-1 [NEXT] RECORD [ LOCK | UNLOCK ] [WAIT] [INTO id-2] [ AT END | INVALID KEY stmt ]
```

Where:

id-1 is a filename that specifies a relative file OPENed for input or I/O.

id-2 is any data item specifying a destination that receives the file's record area.

stmt 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 *id-2* according to MOVE rules. (See the MOVE statement earlier in this chapter.)

COBOL always ignores the LOCK and WAIT options.

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 *stmt*. If you omit this option, or if no end-of-file condition occurs, control passes to the first executable statement following the READ sentence. See "The Declaratives Section" in Chapter 6.

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 *stmt*. If you omit this option, or if no invalid key condition occurs, control passes to the first executable statement following the READ sentence. See "The Declaratives Section" in Chapter 6.

Example

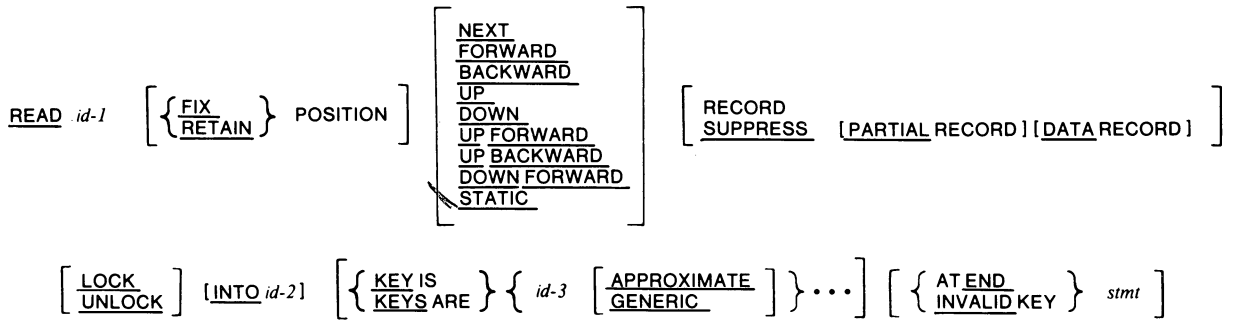
```
READ NAMES INTO LOC100
   INVALID KEY DISPLAY "KEY DOES NOT EXIST".
```

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 user terminal.

READ for an Indexed File

Makes a record available from an indexed file.

Format



Where:

id-1 is a filename that specifies an indexed file OPENed for input or I/O.

id-2 is any data item specifying a destination that receives the file's record area.

id-3 is an alphanumeric data item that specifies a key used to determine a record's location.

stmt 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 *id-2* 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 the first (primary) key in the SELECT clause.

If you specify the KEY series phrase, you must have declared each key *(id-3)* 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, you are the only one who 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. If you SUPPRESS DATA RECORD, locks are ignored.

READ for an Indexed File (continued)

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 *stmt*. 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 *stmt*. 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 *id* [FROM *id-lit*]

Where:

id is an alphanumeric data item specifying a record from the sort/merge description entry which is given in the controlling SORT statement.

id-lit is any data item or literal that specifies a source you want to pass.

Statement Execution

A RELEASE statement releases the contents of *id* as an input record to the initial phase of the SORT operation. If you specify the FROM option, RELEASE moves the contents of *id-lit* to *id* 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. *id* and *id-lit* must not reference the same storage area.

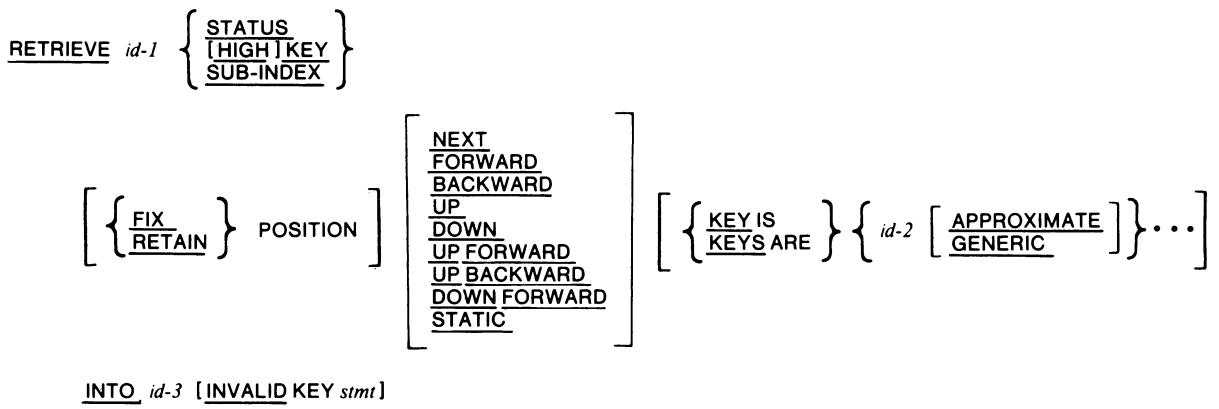
Example

```
RELEASE REC01.
```

RETRIEVE

Obtains information about an indexed file.

Format



Where:

id-1 is a filename that specifies an open indexed file.

id-2 is an alphanumeric data item that specifies a record key associated with *id-1*.

id-3 is any data item specifying a destination that receives record or key status information.

stmt 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 *id-1*, relative to the current position of *id-1* record pointer. If you omit both this option and the **KEY** series option, the default is the first key in the SELECT clause.

If you specify the **KEY** series phrase, you must have declared each key *(id-2)* in the SELECT statement for *id-1*.

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 *id-3* 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:

1st character (leftmost) - a LOGICAL LOCAL DELETE was executed for this record;

2nd character - the key is a duplicate;

3rd character - reserved;

4th character (rightmost) - a LOGICAL GLOBAL DELETE was executed for this record.

If you specify HIGH KEY, RETRIEVE moves the value of the highest key in the subindex associated with the selected record into *id-3* (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 *id-3.*

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 *id-1:*

1. The *id-3* data item in the SELECT clause that names either the last *id-2* you specified in the KEY IS option of the RETRIEVE statement or, if you omitted this option,

2. The first key *(id-3)* you specified in the KEY IS phrase of the SELECT statement.

   The variables that RETRIEVE updates are *id-lit-3* which will contain the length of the RETRIEVEd record's key, and *id-4* which will contain the occurrence number of the RETRIEVEd key.

If you specify SUB-INDEX, RETRIEVE moves into *id-3* the 12-character subindex definition packet associated with the key you specified in the KEY series phrase.

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 *stmt.* 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 *id-1* RECORD [INTO *id-2*] [AT END *stmt*]

Where:

id-1 is a filename that specifies a sort/merge file given in the controlling SORT or MERGE statement.

id-2 is any data item specifying a destination that receives the file's record area.

stmt 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 *id-1*.

If you specify the INTO clause, RETURN moves the contents of the file's record area to *id-2* 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. *id-1* and *id-2* 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 *stmt* and the contents of the file's record area remain undefined. After execution of *stmt*, 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. See "The Declaratives Section" in Chapter 6.

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 *id* [FROM *id-lit*]

Where:

id 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.

id-lit 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 *id-lit* to *id* 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 called "COBOL File Status Data Items" in Chapter 6.)

id-lit and *id* must not reference the same storage area. The number of character positions in *id* 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 *id* [IMMEDIATE] [FROM *id-lit*] [INVALID KEY *stmt*]

Where:

id 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.

id-lit is any data item or literal that specifies the source you want to write.

stmt 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 *id-lit* to *id* 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 called "COBOL File Status Data Items" in Chapter 6.)

id-lit and *id* must not reference the same storage area. The number of character positions in *id* 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.

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 *stmt*. 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. See "The Declaratives Section" in Chapter 6.

The IMMEDIATE clause immediately writes the block containing the record to the disk file. This option trades I/O efficiency for extra data security.

Example

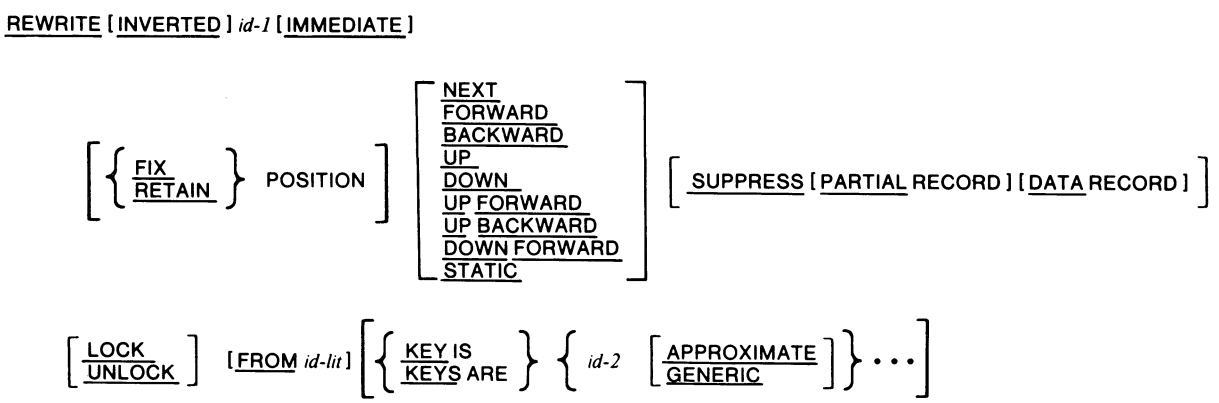
```
REWRITE COMP1 IMMEDIATE INVALID KEY DISPLAY "KEY ERROR".
```

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 user terminal.

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:

id-1 is a data name that specifies a logical record declared in the File Section of the Data Division and OPENed for I/O.

id-lit is any data item or literal that specifies the source you want to write.

id-2 is an alphanumeric data item that specifies a record key associated with the current indexed file.

stmt 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 *id-lit* to *id-1* 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 called "COBOL File Status Data Items" in Chapter 6.)

id-lit and *id-1* must not reference the same storage area. The number of character positions in *id-1* 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.

REWRITE for an Indexed File (continued)

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 *(id-2)* 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, you are the only one who 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. If you SUPPRESS DATA RECORD, COBOL ignores a lock on the record.

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.

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 *stmt*. If you omit this option or if no invalid key condition occurs, control passes to the first executable statement following the REWRITE sentence. See "The Declaratives Section" in Chapter 6.

COBOL always ignores the IMMEDIATE option.

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

Where:

id is an unsubscripted array name whose data description must contain an OCCURS clause specifying the INDEXED BY phrase.

int is an integer data item that specifies an index item.

stmt-1 is an imperative statement to which control passes if *elemt* contains a value greater than the highest possible occurrence number.

expr is any legal conditional expression (see the section "Conditional Expressions" in Chapter 6).

stmt-2 is an imperative statement to which control passes if all specified *exprs* 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 *id* data definition, then the SEARCH statement will evaluate the *expr* 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 *stmt-1*. 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 *exprs* 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 *id's* OCCURS clause) to 1.

If you specify VARYING *int* and if *int* is one of the index items listed in *id* INDEXED BY phrase, then the SEARCH operation varies that index item instead of *i-1*. If *int* is not one of the index items for *id* or if you omit the VARYING phrase, then the SEARCH operation will use *i-1* as the index, and COBOL will increment *int* 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 id RECORD    [INVALID KEY stmt]
```

Where:

id is a filename that specifies a relative file OPENed for input or I/O.

stmt is an imperative statement to which control passes if record selection indicators are invalid.

Statement Execution

COBOL always ignores the SEEK statement.

SET

Is an inverted form of the MOVE statement. It sets one or more data items equal to another data item.

Format

SET *id*, ... TO *id-lit*

Where:

id is an index, numeric, or alphanumeric data item that receives a data source.

id-lit is an index, numeric, or alphanumeric data item or literal 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 it exists), etc. COBOL evaluates any subscripting or indexing associated with *id* 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. The format of the SET statement is equivalent to:

MOVE *id-lit* TO *id*,...

SET UP/DOWN

Adds an operand to or subtracts an operand from one or more operands and stores the results.

Format

SET *id* ,... { UP / DOWN } BY *id-lit*

Where:

id is a numeric data item that specifies an addend or minuend and that receives the result of the addition or subtraction.

id-lit is a numeric literal or a numeric data item that specifies an addend or subtrahend.

Statement Execution

A SET UP statement adds the addend *(id-lit)* to each addend/result *(id)* and stores each result in the corresponding *id* according to MOVE rules. (See the MOVE statement earlier in this chapter.) The SET UP statement is equivalent to the statement:

ADD *id-lit* TO *id*, ...

A SET DOWN statement subtracts the subtrahend *(id-lit)* from the minuend/result *(id)* and stores each result in the corresponding *id* according to MOVE rules. The SET DOWN statement is equivalent to the statement:

SUBTRACT *id-lit* FROM *id*...

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

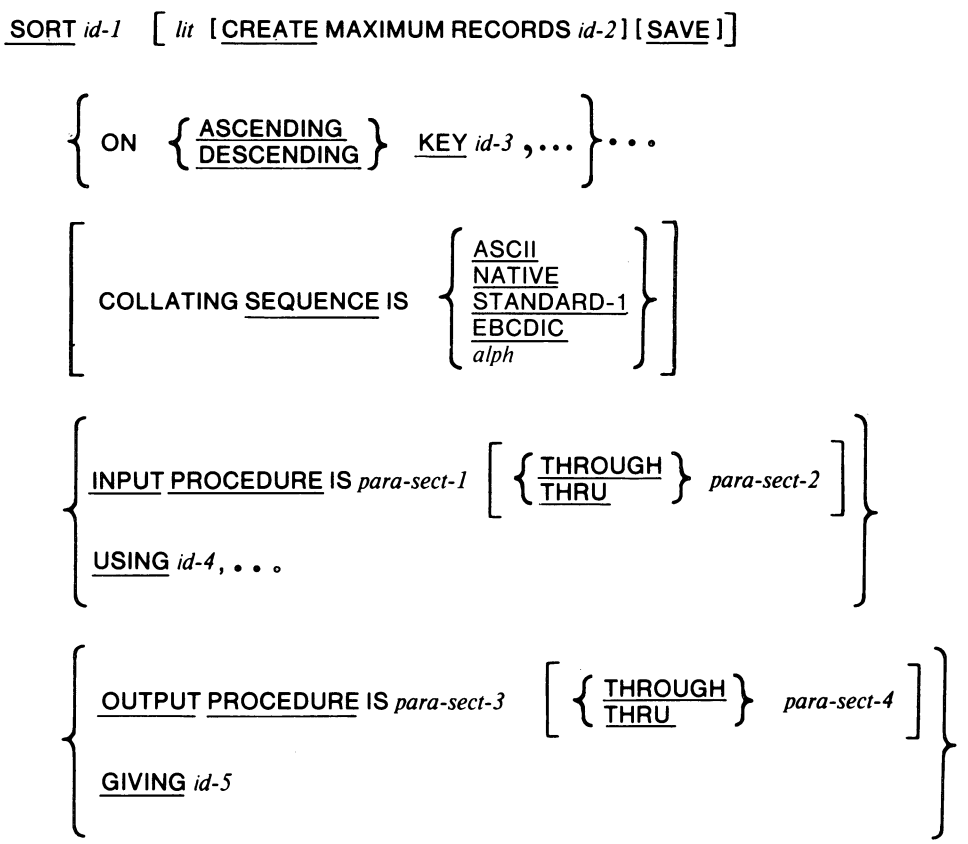
```
SET A, B, C DOWN BY D.
```

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:

id-1 is a filename that specifies a sort/merge file described in a sort/merge file description entry in the Data Division.

lit is an alphanumeric literal that would specify a work file or data item. COBOL ignores this specification.

id-2 is a numeric data item that specifies the number of records you want to allocate for *lit*. COBOL ignores this specification.

id-3 is an alphanumeric data item that specifies a key described in a record associated with *id-1*.

alph is an alphabetic 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.

para-sect-1, para-sect-2 is a Procedure Division section or paragraph name specifying an input procedure which COBOL executes as the initial phase of the SORT operation.

id-4 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.

para-sect-3, para-sect-4 is a Procedure Division section or paragraph name specifying an output procedure which COBOL executes as the final phase of the SORT operation.

id-5 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

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 sequence 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 logical records in the *para-sect-1* and *para-sect-2* and in *id-5* must be equal in size to the logical records of *id-1*.

SORT statements may appear anywhere in the Procedure Division of your program, except in the Declaratives section or in an input or output procedure associated with a SORT or MERGE statement.

After completion of the SORT operation, control passes to the first executable statement following the SORT statement.

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 *id* RECORD *id-lit-1* [CHARACTER *id-lit-2*] [AT END *stmt*]

Where:

id is a filename that specifies a sequential file on a direct access device, OPENed for input or I/O.

id-lit-1 is an integer literal or an integer data item that specifies a record number.

id-lit-2 is an integer literal or an integer data item that specifies a character offset within *id-lit-1*.

stmt 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 return the record number in *id-lit-1* plus one, offset by *id-lit-2* character position(s), if specified.

The first record in a sequential file is numbered 1. If you do not specify *id-lit-2,* 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 *stmt*. 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. (See "The Declaratives Section" in Chapter 6.)

Example

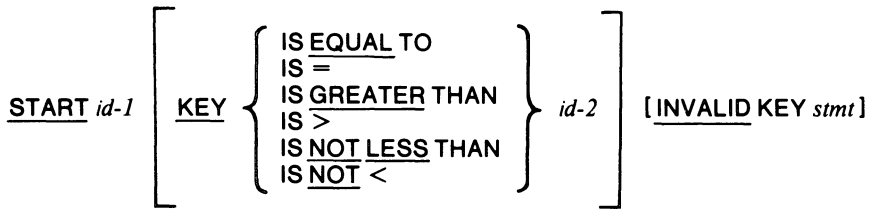
```
START MYFILE RECORD 6.
```

After execution of this statement, the next READ issued for MYFILE will begin at the seventh record.

START for a Relative file

Positions the record pointer in a relative file.

Format



Where:

id-1 is a filename that specifies a relative file OPENed for input or I/O.

id-2 is an integer data item specifying a key that appears in the RELATIVE KEY phrase of the file's SELECT clause.

stmt 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 *id-2,* 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 *id-2* if it exists. If you specify NOT LESS, START positions the file to the record indicated by *id-2.* 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 section called "COBOL File Status Data Items" in Chapter 6).

If you specify the INVALID KEY option and no existing record satisfies the stated comparison, control passes to *stmt.* 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. (See "The Declaratives Section" in Chapter 6.)

Example

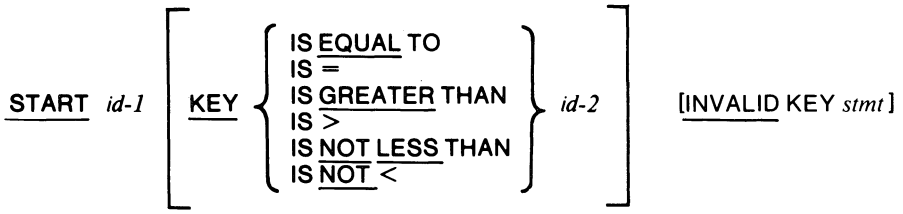
```
START INC99 KEY GREATER IT.
```

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:

id-1 is a filename that specifies an indexed file with sequential or dynamic access and OPENed for input or I/O.

id-2 is an alphanumeric data item that specifies either a record key associated with *id-1* or a subordinate data item whose leftmost character position corresponds to the leftmost character position of a record key data item.

stmt 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 *id-2,* 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 *id-2* and of the same length as *id-2,* if it exists. START also updates the value of *id-2* to the value of the new key. If you specify NOT LESS, START positions the file to the record indicated by *id-2,* or, if that does not exist, to the record whose key is the next key higher. START also updates the value of *id-2* 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 first record key clause associated in this file's SELECT clause for comparison.

If *id-2* 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 *stmt*. 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. (See "The Declarative Section" in Chapter 6.)

In a multilevel indexed file, the key refers to the top file level.

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

$$\underline{\text{STOP}} \left\{ \begin{matrix} \underline{\text{RUN}} \\ \underline{\textit{lit}} \end{matrix} \right\}$$

Where:

lit 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 *lit* statement displays *lit* and suspends execution of the current run unit. If your program is executing at your console, it will display *lit* on the program console. Type any character except ESC to continue execution; typing ESC will terminate the program. If your program is executing in the batch stream, it will display *lit* on the OP console (PID2) and block itself if the operator status is ON, otherwise, the program aborts with the error message: "OPERATOR NOT AVAILABLE." The operator can then unblock the process to continue execution or terminate the process to abort execution.

If you specify a message that is a numeric literal, it must be an unsigned integer.

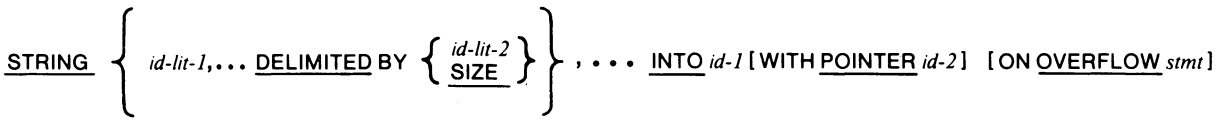
Example

```
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



Where:

id-lit-1 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.

id-lit-2 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.

id-1 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.

id-2 is an unsigned integer data item that specifies the character position in *id-1* at which the STRING operation begins.

stmt is an imperative statement to which control passes if an overflow condition occurs.

If you specify a *id-lit-1* or *id-lit-2* data item as a numeric data item, the STRING statement interprets it as alphanumeric.

If you specify a *id-lit-1* or *id-lit-2* 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 *id-1* data item.

Statement Execution

A STRING statement transfers selected characters from specified source fields *(id-lit-1)* to a destination field *(id-1)*.

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 *id-lit-2* 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 *id-lit-2* (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.

STRING (continued)

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 *(id-2)* 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 *id-2,* according to MOVE rules. The *id-2* data item must be large enough to contain a value equal to the number of characters in *id-1* plus 1.

An overflow condition occurs if, at any point in the execution of a STRING statement, the value of *id-2* 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 *stmt.* 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 id-lit-1,... FROM { id-1 [ROUNDED] } ... [ON SIZE ERROR stmt]
```

SUBTRACT GIVING -

```
SUBTRACT id-lit-1,... FROM id-lit-2 GIVING { id-2 [ROUNDED] } ... [ON SIZE ERROR stmt]
```

SUBTRACT CORRESPONDING -

```
SUBTRACT { CORRESPONDING } id-3 FROM id-4 [ROUNDED] [ON SIZE ERROR stmt]
         { CORR        }
```

Where:

id-lit-1 is a numeric literal or a numeric data item that specifies a subtrahend.

id-1 is a numeric data item that specifies a minuend and receives the result of a simple SUBTRACT operation.

stmt is an imperative statement to which control passes if a size error condition occurs.

id-lit-2 is a numeric literal or a numeric data item that specifies a minuend.

id-2 is a numeric or numeric edited data item that receives the result of a SUBTRACT GIVING operation.

id-3 is a group data item that specifies subtrahends.

id-4 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 *id-1*, and storing the result in *id-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 *id-2* following the word GIVING.

A SUBTRACT CORRESPONDING statement subtracts data items in *id-3* from corresponding data items in *id-4,* storing the results in the *id-4* 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 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 *stmt*. 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 *id* BLOCK

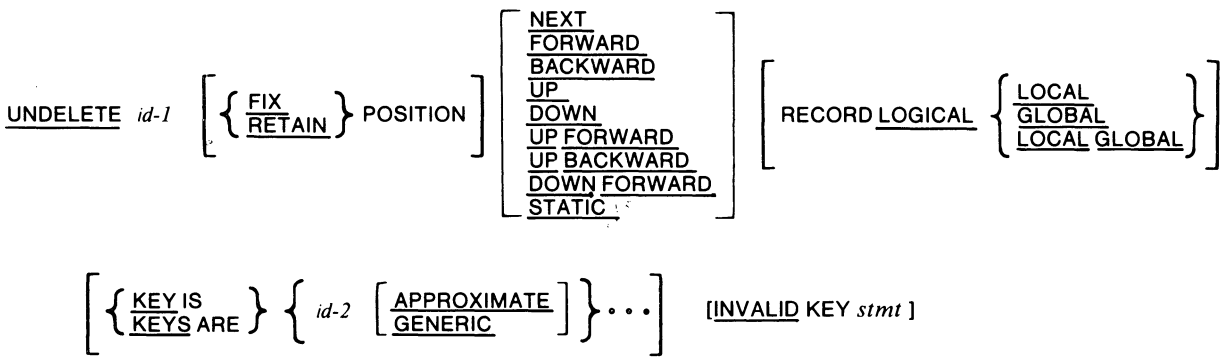
Where:

id is a filename that specifies an open, sequentially organized file.

UNDELETE

Restores a record that you logically deleted from an indexed file.

Format



Where:

id-1 is a filename that specifies an open, indexed file.

id-2 is an alphanumeric data item that specifies a record key associated with *id-1*.

stmt 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 *(id-2)* 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 do not specify the type of restoration, COBOL defaults to 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 *stmt*. If you omit this option, or if no invalid key condition occurs, control passes to the first executable statement following the UNDELETE sentence. (See "The Declaratives Section" in Chapter 6.)

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.

UNLOCK

Unlocks all the records of a file locked by your process.

Format

Where:

id is the name of the file containing the records you wish to UNLOCK.

Statement Execution

This statement unlocks all the records in *id* which were locked (with READ statements) by any program running under your process.

COBOL always ignores the RECORD and RECORDS clauses.

UNSTRING

Separates contiguous data located in a single source field and moves it to one or more destination fields.

Format

```
UNSTRING id-1 [DELIMITED BY [ALL] id-lit-1 [OR BY [ALL] id-lit-2] ... ]

        INTO { id-2 [DELIMITER IN id-3] [COUNT IN id-4] } ... [WITH POINTER id-5] [TALLYING IN id-6]

        [ON OVERFLOW stmt]
```

Where:

id-1 is an alphanumeric data item that specifies a source field.

id-lit-1, id-lit-2 is an alphanumeric data item or an alphanumeric literal that specifies a string delimiter.

id-2 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.

id-3 is an alphanumeric data item that receives a string delimiter.

id-4 is an integer data item that receives the number of characters transferred in one iteration of the UNSTRING operation.

id-5 is an integer data item that specifies the character position in *id-1* where the UNSTRING operation begins.

id-6 is an integer data item that receives the number of iterations the UNSTRING operation performs.

stmt 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 *id-1, id-5,* or *id-6* only once, at the beginning of the UNSTRING operation. COBOL evaluates any subscripting for *id-lit-1, id-lit-2, id-2, id-3,* or *id-4* immediately before transferring data to that item.

Statement Execution

An UNSTRING statement transfers selected characters from a source field *(id-1)* to one or more destination fields *(id-2)*. 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 nonnumeric or numeric, respectively.

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 *id-3*. If there was no delimiter match, COBOL space-fills *id-3*. If you specify the COUNT IN phrase, COBOL stores the number of characters transferred in *int-1*, 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 *(id-5)*. 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 *id-5*, according to MOVE rules. The *id-5* data item must be large enough to contain a value equal to the number of characters in *id-2* 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 *id-3* (if you specified it), and the number of contiguous delimiters to *id-4* (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 *id-6*. If you specified the WITH POINTER phrase, COBOL stores the current value of the source scan base in *id-5*, according to MOVE rules.

An overflow condition occurs if, at any point in the execution of an UNSTRING statement, the value of *id-5* 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 *stmt*. If you omit this option or if no overflow condition occurs, control passes to the first executable statement following the UNSTRING sentence.

Example

Given:

```
01 SR PIC X(23)
   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

```
USE AFTER STANDARD { EXCEPTION } PROCEDURE ON { id ,...  }
                   { ERROR     }              { INPUT    }
                                              { OUTPUT   }
                                              { I-O      }
                                              { EXTEND   }
```

Where:

id 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 *id* 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 *id* 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 *id*. If the error occurs in the OPEN statement itself then the Declaratives section will only be called if you specified *id*.

After execution of a USE statement, control passes to the first executable statement following the statement whose processing invoked the Declaratives section.

Example

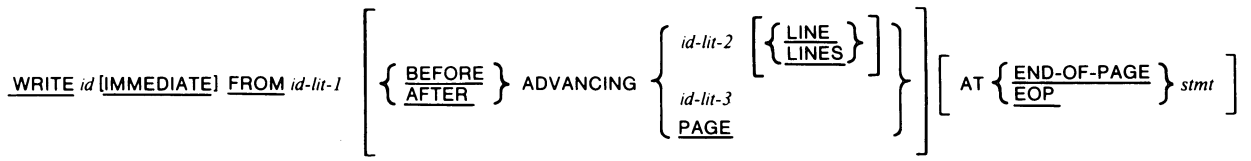
```
DECLARATIVES.
   SEC100 SECTION 2.
      USE AFTER ERROR PROCEDURE ON INPUT
            ADD 1 TO FLAG.
            MOVE COUNT TO ERR-SUM.
            .
            .
            .
END DECLARATIVES.
```

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:

id is a data name that specifies a logical record in a sequential file OPENed for output or extension.

id-lit-1 is any data item or literal that specifies the source you want to write.

id-lit-2 is a nonnegative integer literal or an unsigned integer data item that specifies the number of lines you want to advance.

id-lit-3 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.

stmt 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 *id-lit-1* to *id* according to MOVE rules, prior to performing the WRITE operation. (See the MOVE statement earlier in this chapter.) *id-lit-1* and *id* must not reference the same storage area.

The IMMEDIATE option prohibits your program from continuing until AOS flushes the information to disk. Normally, AOS holds disk information in a buffer, filling the buffer before it performs any I/O. Use of IMMEDIATE allows you greater data security at the expense of performance.

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 *id-lit-3* 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 *id-lit-2* to advance the current position to a position *id-lit-2* lines ahead. You specify *id-lit-3* 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.

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 *stmt*. 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 *stmt*.

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 *id* [IMMEDIATE] [FROM *id-lit*] [INVALID KEY *stmt*]

Where:

id is a data name that specifies a logical record in a relative file OPENed for output or I/O.

id-lit is any data item or literal that specifies the source you want to write.

stmt 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 *id-lit* to *id* according to MOVE rules, prior to performing the WRITE operation. (See the MOVE statement earlier in this chapter.) *id-lit* and *id* 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.

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 *stmt*. If you omit this option or if no invalid key condition occurs, control passes to the first executable statement following the WRITE sentence. (See "The Declaratives Section" in Chapter 6.)

The IMMEDIATE option prohibits your program from continuing until AOS flushes the information to disk. Normally, AOS holds disk information in a buffer filling the buffer before it performs any I/O. Use of IMMEDIATE allows you greater data security at the expense of performance.

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] id-1 [IMMEDIATE]

[ { FIX    } POSITION ] [ UP   ]
[ { RETAIN }          ] [ DOWN ]   [SUPPRESS [PARTIAL RECORD] [DATA RECORD]]
                        [ STATIC ]

[ LOCK   ] [FROM id-lit] [ { KEY IS  } id-2, ... ] [INVALID KEY stmt]
[ UNLOCK ]               [ { KEYS ARE }          ]
```

Where:

id-1 is a data name that specifies a logical record in an indexed file OPENed for output or extension.

id-lit is any data item or literal that specifies the source you want to write.

id-2 is an alphanumeric data item that specifies a record key associated with a file.

stmt 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 *id-lit* to *id-1* according to MOVE rules, prior to performing the **WRITE** operation. (See the MOVE statement earlier in this chapter.) *id-lit* and id-1 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 the first key in the SELECT clause.

If you specify the **KEY** series phrase, you must have declared each *id-2* 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, you are the only one who 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. If you SUPPRESS DATA RECORD, Locks are ignored.

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.

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 *stmt*. If you omit this option or if no invalid key condition occurs, control passes to the first executable statement following the WRITE sentence.

COBOL always ignores the IMMEDIATE option.

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 AOS file structures, see the *AOS Programmer's Manual*.

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.

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 Chapter 10 for a discussion of compilation instructions.

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 terminate it with a period (which is considered part of the statement). COPY statements in the Identification Division must begin in the A-margin. A COPY statement has the following format: *

COPY *id-1* [{ OF / IN } *id-2*] [REPLACING { *id-lit-1* BY *id-lit-2* } ...] .

Where:

id-1 is a symbolic name that specifies the name of the file you want to copy. You may use any valid file specifier.

id-2 is a symbolic name specifying the name of the library directory (if one exists) that contains *id-1*.

id-lit-1 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.

id-lit-2 is a single COBOL word, a numeric or alphanumeric literal, a character string, or a pseudo-text that specifies text with which you want to replace each occurrence of *id-lit-1*.

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 *id-2:id-1* and omit the option OF/IN LIB.

If all the characters in *id-1* or *id-2* 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".
```

The two statements above illustrate equivalent uses of COPY.

```
COPY DATA_TEXT OF LIB.
```

This statement is incorrect because DATA_TEXT is not a legal COBOL name.

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 *id-lit-1* in your program's copy of the library file with *id-lit-2*.

Each string can consist of ten elements: *id-lit-1* may not be a null, but *id-lit-2* may. Otherwise, *id-lit-1* or *id-lit-2* can consist of any of the following elements:

- A single COBOL reserved word, user-defined word, or undefined word
- 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

For example, when COBOL encounters the following in a source program:

```
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.
```

it inserts a file, LTEST, into that source program and makes four types of replacements. Under the above instructions, if LTEST contains the following:

```
"LMN" TRY TO
FIT 26 TO 13
IF X OF Y OF M (3,A) =
   X OF Y OF Z (3,A), ABC.
```

COBOL 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 *id-lit-1*.

End of Chapter

Chapter 9
The COBOL Interactive Debugger

The ECLIPSE COBOL system provides a source-level debugging aid that allows you to debug programs interactively at the terminal. This debugger responds to COBOL-like commands as you issue them from the terminal.

The debug program allows you to set breakpoints in your program causing it to 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.

You should also use the /L compiler switch to get a listing of the program. You will need the line numbers from the /L switch to set break points.

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 this CLI command:

```
)DEB MAINPROGRAM)
+=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 prompt character each time it is ready to accept a new command. You must terminate each debugger command with a NEW LINE.

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 a NEW LINE.

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 pass control to the debugger. The CLEAR command removes either all or specific breakpoints that you previously set.

Checking Program Status

When a 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, an error, or upon 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.

Typing STOP at the console terminates execution immediately. You will want to use this command to abort execution or to end a debugging session before a 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.

Please note that the COBOL interactive debugger does not process data names beginning with a numeric character.

AUDIT

Saves terminal dialog of user, debugger, and object program.

Format

AUDIT [*"lit"*]

Where:

lit is an alphanumeric literal that specifies the name of an operating system file (called the *audit file).*

Description

If you specify *lit,* the system opens that file and places in it a copy of all succeeding console interaction between the user, the debugger, and the object program. If you omit *lit,* the system closes the current audit file, if one exists, and terminates audit output.

You may open only one audit file at a time.

CLEAR

Removes breakpoints previously set with a SET command.

Format

CLEAR [*lit*, ...]

Where:

lit is an alphanumeric literal that specifies the name of a breakpoint defined in a previous SET command.

Description

If you specify a *lit* (or *lit's)* the system removes that particular breakpoint. If you omit *lit* altogether, the system removes all previously set breakpoints.

CLI

Allows you to enter the AOS Command Line Interpreter (CLI).

Format

CLI

Description

Before you enter the CLI, the system saves the current state of your program. Re-enter the debugger with the AOS BYE command. For more information on the CLI, see the *AOS Command Line Interpreter User's Manual* (093-000122).

COMPUTE

Sets one or more data items to the value of an arithmetic expression.

Format

$$\underline{\text{COMPUTE}} \left\{ id \left[\underline{\text{ROUNDED}} \right] \right\}, \ldots = expr$$

Where:

id 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 resultant item(s) you specify, according to the rules for the COBOL COMPUTE statement (see Chapter 7).

You may qualify and/or subscript any *id*, as appropriate.

CON

Either initiates execution of the object program or resumes program execution after encountering a breakpoint.

Format

CON [[*lit*] *int*]

Where:

lit is an alphanumeric literal that specifies the name of a breakpoint defined in a previous SET command.

int is a positive integer literal that specifies the number of times you want *lit* 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 *lit*, and *lit* 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 *lit* and *int*, and *lit* 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 *int*.

If you omit both *lit* and *int*, 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 *"lit"*

Where:

lit 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 *lit,* 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 *lit* to a depth of 5 nested COPY commands.

DISPLAY

Outputs the contents of object program data items to the current console.

Format

DISPLAY *id*, ...

Where:

id 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 multiple program environment, to which future debugger commands will apply.

Format

ENV [*id*]

Where:

id is the name of the main program ID or one of its COBOL subprogram ID in the current run unit.

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 *id*, the default is the main program. This is what the debugger assumes when you begin execution of your program.

Setting **ENV** enables you to check the variables in a program, which is in the current run unit and is not executing, when the run unit pauses at one of your breakpoints. You have access to 50 subprograms in the current run unit. CLEAR will clear all your breakpoints in all your subprograms, regardless of **ENV**.

MOVE

Sets data items in the object program to specified values.

Format

MOVE *id-lit* TO *id*, ...

Where:

id-lit is a literal or data item defined in the object program.

id is a data item defined in the object program.

Description

This command stores the value of *id-lit* 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 [*lit-1*, [*id* +] *lit-2*

Where:

lit-1 is an alphanumeric literal that specifies the breakpoint you want defined and enabled.

id is a symbolic name that specifies the object program.

lit-2 is an integer literal that indicates the number of the line at which you want to define the breakpoint.

Description

The line number, *lit-2,* does not refer to the card format sequence number. It refers to the absolute number of a line in the source file, as numbered by the COBOL compiler in the source listing.

If you specify *id,* the system sets a breakpoint at the line number indicated by *lit-2* in *id.* If you omit *id,* the system sets a breakpoint at the line number indicated by *lit-2* in the program last referenced in an ENV command. If you have not issued an ENV command, *id* 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 SET without any arguments, the system displays a list of the current breakpoints on the terminal.

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 by listing active subroutine CALLs and PERFORMs.

Format

WALKBACK [*int*]

Where:

int is an integer literal that specifies the number of CALLs and PERFORMs you want listed.

Description

If you specify *int,* the system lists the *int* 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
The Data General Database Management System (DG/DBMS) Interface

The Data General Database Management System (DG/DBMS) is a Codasyl compliant, network structured, database management system.

In general, a database is a collection of interrelated data stored in a way that eliminates redundancy, permits many distinct applications to use the information in different ways, stores the data in a method independent of the programs that use it, and controls access to the data.

DG/DBMS is a system that manages and coordinates access to all the data in the database. You can set up your application so that information from different sources exists in the same database. Many programs can then access and modify the database simultaneously. In this way, the system need not store data redundantly, and all programs can access the most recent information about a subject.

DG/DBMS also protects data by allowing only certain programs to access certain data. Some programs can modify data, others can only look at data. In addition, DG/DBMS can prevent more than one program from modifying the same piece of data simultaneously, and prevent a program from accessing information until another program finishes with it.

The Data Base Administrator (DBA) in your organization will describe and create the entire database you will access. You cannot change the defined structure of the database or create a new one from your COBOL program. You must instruct your DBA to do this. However, the DBA also creates a special structure for your program which may allow you to change entries and the relationships between entries in the database.

The discussions in this chapter use an example to explain using DG/DBMS. This example illustrates a hospital application where each doctor can have several patients, and each patient may be assigned several doctors. With this example structure, you can learn which doctors are treating a patient or, conversely, which patients are assigned to a doctor(s).

This chapter tells you what statements and clauses you must add to your COBOL program in order to use DG/DBMS, and how to bind the DG/DBMS library with your COBOL program. Because we don't include a full discussion of DG/DBMS here, you should read the *Data General/Database Management System (DG/DBMS) Reference Manual* (093-000163) before you attempt to use this chapter.

Compiling and Binding DG/DBMS With a COBOL Program

You compile and bind a COBOL program that contains DG/DBMS statements as you normally would any COBOL program. However, you must bind the module ?DBMS.OB in with your program, as follows:

CBIND / *sw/sw ... filename/sw/sw ...* ?DBMS.OB

If a library contains ?DBMS.OB, you must specify that library name in the CBIND command line.

DG/DBMS Subschemas In The Data Division

To use DG/DBMS statements in a COBOL program, you must include a description of your database in that program's Data Division. The Data Base Administrator (DBA) in your organization will build this description.

First, the DBA uses the Data Definition Language (DDL) to describe and create the entire database. The DDL specifies the record structure of each record type, the connection criteria, the paths among records, and the access rights given to your program. This description is called the *schema.* Then, the DBA creates subsets of the schema, called *subschemas,* whose source code is compatible with a COBOL program. You will copy this source code into your COBOL program.

Figure 10-1 shows an example of a subschema that we will use to illustrate using DG/DBMS in COBOL, and Figure 10-2 shows the structure of that data. Those lines that are preceded by an asterisk are comment lines used to provide information to the programmer; the DBA specified this information when defining the subschema.

```
            SUBSCHEMA NAME IS "PATIENT-SEARCH"
               WITHIN ":UDD:HARRIS:TREATMENT_DATABASE"
               ALLOWS ERASE GET MODIFY STORE

            SET NAME IS DOCTORS-BY-NAME
                ALLOWS RECONNECT
                OWNER IS SYSTEM
                MEMBER IS DOCTOR
                AUTOMATIC MANDATORY
*               ORDER IS SORTED BY KEY ASCENDING
*                   KEYS ARE:
*                       LAST-NAME
*                       FIRST-NAME
*               DUPLICATES ALLOWED
*               MEMBER LIMIT IS NONE

            SET NAME IS PATIENTS-BY-NAME
                ALLOWS RECONNECT
                OWNER IS SYSTEM
                MEMBER IS PATIENTS
                AUTOMATIC MANDATORY
*               ORDER IS SORTED BY KEY ASCENDING
*                   KEYS ARE:
*                       LAST-NAME
*                       FIRST-NAME
*               DUPLICATES ALLOWED
*               MEMBER LIMIT IS NONE

            SET NAME IS PATIENT-TREATMENTS
                ALLOWS RECONNECT
                OWNER IS PATIENTS
                MEMBER IS TREATMENTS
                AUTOMATIC MANDATORY
*               ORDER IS NEXT
*               MEMBER LIMIT IS NONE

            SET NAME IS DOCTOR-TREATMENTS
                ALLOWS RECONNECT
                OWNER IS DOCTOR
                MEMBER IS TREATMENTS
                AUTOMATIC MANDATORY
*               ORDER IS NEXT
*               MEMBER LIMIT IS NONE

                DBMS STATUS IS DBMS-STATUS.
```

Figure 10-1. A DG/DBMS Subschema in a COBOL Program (continues)

```
01      DOCTOR ALLOWS ERASE GET MODIFY STORE.
        05  LAST-NAME  PIC X(25)
                                          USAGE IS DISPLAY
                                          ALLOWS GET MODIFY.
        05  FIRST-NAME  PIC X(20)
                                          USAGE IS DISPLAY
                                          ALLOWS GET MODIFY.
        05  SPECIALTY  PIC X(15)
                                          OCCURS 5 TIMES
                                          USAGE IS DISPLAY
                                          ALLOWS GET MODIFY.
        05  INFO  PIC X(80)
                                          USAGE IS DISPLAY
                                          ALLOWS GET MODIFY.
        05  BEEPER  PIC S9(4)
                                          USAGE IS DISPLAY
                                          SIGN IS TRAILING
                                          ALLOWS GET MODIFY.

01      PATIENTS ALLOWS ERASE GET MODIFY STORE
        05  LAST-NAME  PIC X(20)
                                          USAGE IS DISPLAY
                                          ALLOWS GET MODIFY.
        05  FIRST-NAME  PIC X(15)
                                          USAGE IS DISPLAY
                                          ALLOWS GET MODIFY.
        05  WARD  PIC X(4)
                                          USAGE IS DISPLAY
                                          ALLOWS GET MODIFY.
        05  ROOM  PIC S9(3)
                                          USAGE IS DISPLAY
                                          SIGN IS TRAILING
                                          ALLOWS GET MODIFY.

01      TREATMENTS ALLOWS GET MODIFY ERASE STORE
        05  DISEASE  PIC X(100)
                                          USAGE IS DISPLAY
                                          ALLOWS GET MODIFY.
        05  MEDICATION  PIC X(25)
                                          OCCURS 5 TIMES
                                          USAGE IS DISPLAY
                                          ALLOWS GET MODIFY.
        05  DIET  PIC X(200)
                                          USAGE IS DISPLAY
                                          ALLOWS GET MODIFY.
        05  SPECIAL-INSTRUCTIONS  PIC X(40)
                                          OCCURS 5 TIMES
                                          USAGE IS DISPLAY
                                          ALLOWS GET MODIFY.
```

Figure 10-1. A DG/DBMS Subschema in a COBOL Program (concluded)

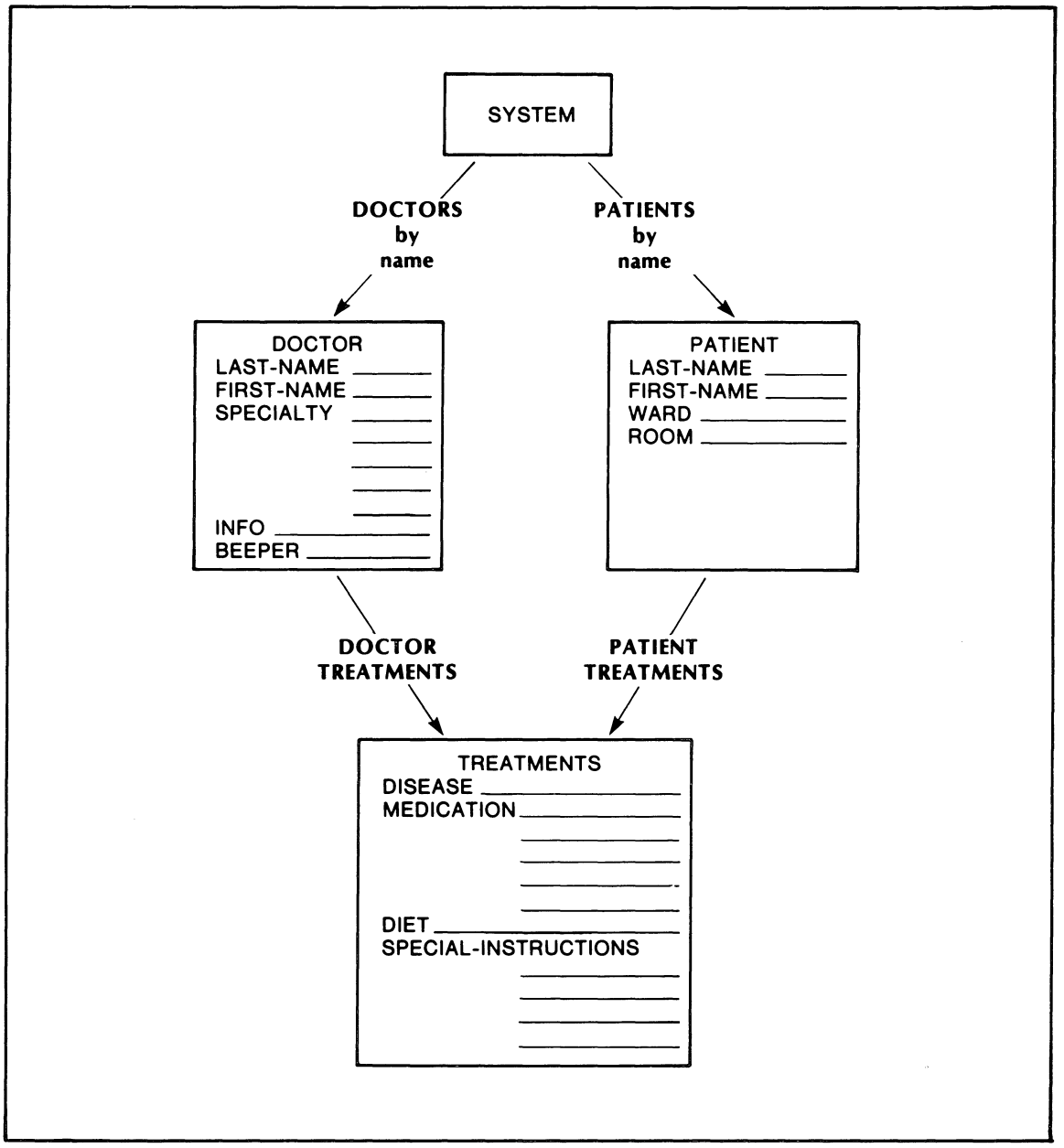


Figure 10-2. Structure of Data in the Figure 10-1 Subschema

Using a subschema is simple. Use the COBOL COPY statement to copy the subschema source code file into a special section within your program's Data Division. This special section is called the *Subschema Section;* it immediately precedes the Working-Storage Section. Your code for a subschema in the Data Division must follow this format:

```
DATA DIVISION.
FILE SECTION.
     file section body.
SUBSCHEMA SECTION.
     COPY "subschema-copy file".
WORKING-STORAGE SECTION.
     Working-Storage Section body.
```

DG/DBMS automatically produces a subschema copyfile when the DBA defines a subschema. If the DBA modifies a subschema, you must recompile your COBOL program with the new version of the copyfile. DG/DBMS deletes the old copyfile.

Subschema copyfiles have special ACL controls which determine if a user has compile, retrieve, and/or update access for a subschema. So, if you get the message FILE ACCESS DENIED on a COPY statement, see your DBA about changing the subschema access privileges. The *DG/DBMS Reference Manual* contains complete information about subschema access rights.

The subschema pathname is the AOS directory pathname for the database directory followed by either the subschema source code filename, a link, or the name of a file on your search list. Your DBA provides the subschema name.

For example, for the database with the name TREATMENT_DATABASE using the subschema name PATIENT_SEARCH located in directory :UDD:HARRIS, specify

```
COPY ":UDD:HARRIS:TREATMENT_DATABASE:PATIENT_SEARCH.COB".
```

Let's talk about our example subschema. We treat the subschema as two parts. The first part, an enhancement to COBOL syntax, describes interrecord relationships. The second part, which looks like an FD record definition in the Data Division, describes various record types for the database.

A *record type* is an 01-level record definition and its associated elementary items (see Figure 10-1). Many records of the same record type may exist in the database; we call these records *occurrences* of the record type.

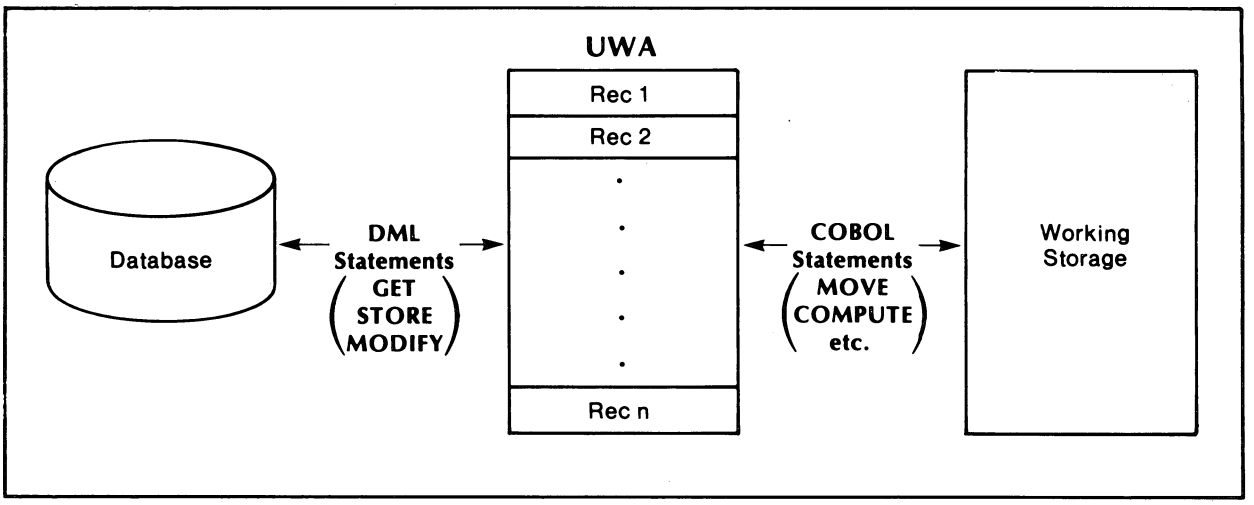
Figure 10-3 shows the record type description of our database. This part of the subschema appears to define three group data items with a total of 13 elementary data items. This is true if you reference these data items with ordinary COBOL statements (MOVE, COMPUTE, etc.). But, if you use a special set of statements, called Data Manipulation Language (DML) statements, on those data items, the data structure becomes a window into the DG/DBMS database.

When you COPY a subschema into your program, COBOL defines a storage area called the User Work Area (UWA) in the same manner as Working-Storage. Within the UWA, COBOL defines separate, physical storage areas for each record in the subschema. You use ordinary COBOL statements to move data between the UWA and other data areas in your program. You use DML statements to transfer information between the UWA and the database. Figure 10-4 illustrates this.

DML statements can transfer information between the record areas in the UWA and the database file. The record definitions in the Subschema Section describe the different types of data that a program can reference in the database. DG/DBMS extends the COBOL record definition by including an ALLOWS clause. This clause specifies the DML statements that you can use on a record type (01 level) and its subordinate items.

```
01      DOCTOR ALLOWS ERASE GET MODIFY STORE
        05  LAST-NAME  PIC X(25)
                                          USAGE IS DISPLAY
                                          ALLOWS GET MODIFY.
        05  FIRST-NAME  PIC X(20)
                                          USAGE IS DISPLAY
                                          ALLOWS GET MODIFY.
        05  SPECIALTY  PIC X(15)
                                          OCCURS 5 TIMES
                                          USAGE IS DISPLAY
                                          ALLOWS GET MODIFY.
        05  INFO  PIC X(80)
                                          USAGE IS DISPLAY
                                          ALLOWS GET MODIFY.
        05  BEEPER  PIC S9(4)
                                          USAGE IS DISPLAY
                                          SIGN IS TRAILING
                                          ALLOWS GET MODIFY.

01      PATIENTS ALLOWS ERASE GET MODIFY STORE
        05  LAST-NAME  PIC X(20)
                                          USAGE IS DISPLAY
                                          ALLOWS GET MODIFY.
        05  FIRST-NAME  PIC X(15)
                                          USAGE IS DISPLAY
                                          ALLOWS GET MODIFY.
        05  WARD  PIC X(4)
                                          USAGE IS DISPLAY
                                          ALLOWS GET MODIFY.
        05  ROOM  PIC S9(3)
                                          USAGE IS DISPLAY
                                          SIGN IS TRAILING
                                          ALLOWS GET MODIFY.

01      TREATMENTS ALLOWS GET MODIFY ERASE STORE
        05  DISEASE  PIC X(100)
                                          USAGE IS DISPLAY
                                          ALLOWS GET MODIFY.
        05  MEDICATION  PIC X(25)
                                          OCCURS 5 TIMES
                                          USAGE IS DISPLAY
                                          ALLOWS GET MODIFY.
        05  DIET  PIC X(200)
                                          USAGE IS DISPLAY
                                          ALLOWS GET MODIFY.
        05  SPECIAL-INSTRUCTIONS  PIC X(40)
                                          OCCURS 5 TIMES
                                          USAGE IS DISPLAY
                                          ALLOWS GET MODIFY.
```

Figure 10-3. COBOL/DBMS Subschema Record Type Description



SD-02100

Figure 10-4. The Function of the User Work Area (UWA)

The SYSTEM Record Type

One record type in your subschema owns other records (it is never a member); it is called the *SYSTEM record type*. The SYSTEM record type has only one occurrence. The DG/DBMS system creates that occurrence and keeps header information in it for its own use; you cannot access that occurrence. As far as you are concerned, your DBA uses the SYSTEM record type when building a schema, to "hang" other record types from. All application programs wishing to access any part of the database structure must begin that navigation through a system set type via the SYSTEM record.

Set Types

The first part of a subschema defines relationships between record types (01-level data items) in the subschema. These record types are called *sets*. Figure 10-5 shows the set types in our example subschema.

Each set type consists of the following:

- An owner record type specification
- A member record type specification
- Insertion/retention criteria for member records
- A method for ordering member records
- A clause determining whether or not duplicate sort key values are allowed
- A maximum number of member occurrences (if desired)

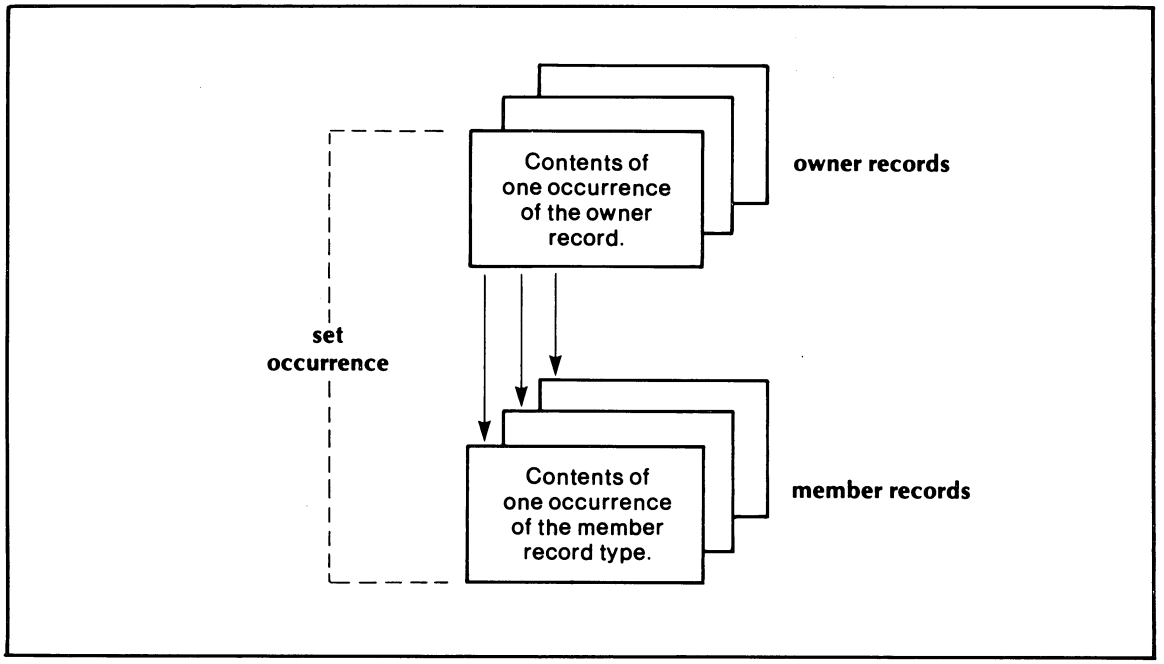
In the database, a *set occurrence* consists of at least one occurrence of the owner record type and zero or more member record occurrences. It is not necessary to have a member record occurrence in a set occurrence; but, if members are connected, they are part of the set occurrence. The rules for the set type define the owner/member relationships in the set occurrence.

We want to remind you that all the aforementioned DG/DBMS terms and relationships are discussed in detail in the *DG/DBMS Reference Manual.*

Using the analogy that Codasyl databases are "network" or "navigational" databases, you can consider set occurrences as the pathways that you travel and record occurrences as the destinations to which you go. Figure 10-6 illustrates this principal.

```
          SUBSCHEMA NAME IS "PATIENT-SEARCH"
             WITHIN ":UDD:HARRIS:TREATMENT_DATABASE"
             ALLOWS ERASE GET MODIFY STORE

          SET NAME IS DOCTORS-BY-NAME
              ALLOWS RECONNECT
              OWNER IS SYSTEM
              MEMBER IS DOCTOR
              AUTOMATIC MANDATORY
*             ORDER IS SORTED BY KEY ASCENDING
*                 KEYS ARE:
*                     LAST-NAME
*                     FIRST-NAME
*                     DUPLICATES ALLOWED
*             MEMBER LIMIT IS NONE

          SET NAME IS PATIENTS-BY-NAME
              ALLOWS RECONNECT
              OWNER IS SYSTEM
              MEMBER IS PATIENTS
              AUTOMATIC MANDATORY
*             ORDER IS SORTED BY KEY ASCENDING
*                 KEYS ARE:
*                     LAST-NAME
*                     FIRST-NAME
*                 DUPLICATES ALLOWED
*             MEMBER LIMIT IS NONE

          SET NAME IS PATIENT-TREATMENTS
              ALLOWS RECONNECT
              OWNER IS PATIENTS
              MEMBER IS TREATMENTS
              AUTOMATIC MANDATORY
*             ORDER IS NEXT
*             MEMBER LIMIT IS NONE

          SET NAME IS DOCTOR-TREATMENTS
              ALLOWS RECONNECT
              OWNER IS DOCTOR
              MEMBER IS TREATMENTS
              AUTOMATIC MANDATORY
*             ORDER IS NEXT
*             MEMBER LIMIT IS NONE

              DBMS STATUS IS DBMS-STATUS.
```

Figure 10-5. COBOL Subschema Set Relationship



SD-02101

Figure 10-6. Owner-Member Record Diagram

Let's look at the first part of Figure 10-5 and briefly discuss the components. A header appears before the set descriptions. It gives the name of the subschema:

SUBSCHEMA NAME IS "PATIENT_SEARCH"

The AOS filename for the subschema copyfile is always the subschema name with the extension .COB. The header also gives the database directoryname (AOS):

WITHIN "TREATMENT_DATABASE"

It also specifies the type of access that you have with this subschema in the clause:

ALLOWS ERASE GET MODIFY STORE

In the first part of the first set description, the clause

SET NAME IS DOCTORS-BY-NAME

identifies the name of the set type. The ALLOWS clause for this set type indicates that RECONNECT statements are allowed; CONNECT and DISCONNECT are not allowed.

The next two lines of the example specify the two record types associated with the set type. One record type is always the OWNER, and another record type is always the MEMBER. A set type can have only one owner record type and only one member record type.

The next clause defines insertion/retention criteria. Sets may have AUTOMATIC or MANUAL insertion modes and MANDATORY or OPTIONAL retention modes. AUTOMATIC and MANUAL options refer to set member connections. When you store a record occurrence, if the record type is a member in an automatic set, then DG/DBMS automatically connects the new occurrence to the current owner. If you want to connect a member into a MANUAL set, you must specify a CONNECT statement.

MANDATORY and OPTIONAL refer to set disconnections. If you specify MANDATORY, you may not disconnect a record from its owner in the set without erasing the record from the database. Application programs may move MANDATORY record occurrences from one set occurrence to another within the same set type. If you specify OPTIONAL, you can disconnect a member from a set occurrence without erasing it. You should note, however, that any record occurrence disconnected from a set remains in the database, but may permanently disappear from view if that set is the only one in which the record was connected.

You can always issue a RECONNECT statement for a record, regardless of its insertion or retention properties. That is, you can reconnect a record with a MANDATORY retention characteristic, and DG/DBMS will disconnect the record from its old set occurrence and connect it to a new owner.

The ORDER IS clause defines the relative ordering of member records within a set occurrence. It has five options.

SORTED BY KEY ASCENDING KEY IS key - Sorts the members in a set occurrence by the designated key field(s) in the member record type

FIRST - Places new record occurrences before the first member of the set occurrence

LAST - Places new record occurrences after the last member of the set occurrence

NEXT - Places new record occurrences immediately after the record occurrence (owner or member) you last accessed in the set occurrence; if current of set is on an owner, DG/DBMS inserts the record as the new first member

PRIOR - Places new record occurrences immediately before the record occurrence you last accessed in the set occurrence; if current of set is on an owner, DG/DBMS inserts the record as the new last member

The duplicates clause, DUPLICATES [NOT] ALLOWED, allows or prohibits the use of duplicate sort keys. The DBA can specify this only when specifying the SORTED BY KEY clause.

The DG/DBMS error status clause, DBMS STATUS IS DBMS-STATUS, specifies a system defined data name of type X(5). It will contain the DG/DBMS error code if and when you receive an error. 00000 indicates no error.

Declaring Free Cursors

Free cursors are pointers that you can set in the database. You declare them in the Working-Storage section of your program. With a free cursor set, it is not necessary to have *current of record* or *current of set* positioned on that record to access it. See the section *Positioning* later in this chapter for the definitions of these terms.

Use the following format to declare a free cursor in Working Storage:

```
{01}
{77} fcn [USAGE IS] CURSOR FOR recna.

01   fcn OCCURS int TIMES [USAGE IS] CURSOR FOR recna.
```

Where:

fcn is the free cursor name.

recna is the name of the record type that *fcn* is associated with.

int is an integer constant from 1 to 255.

Free cursors are only 01- or 77-level items; you do not specify a PICTURE clause for these items. DG/DBMS keeps all the information about the cursor in its own process space; COBOL has no control over the cursor information. You use DML statements to ASSIGN a free cursor to a record occurrence and to later refer to that free cursor. You cannot MOVE any information to or from a free cursor name. No data item can be subordinate to a free cursor name.

Overview of DML Statements in The Procedure Division

As we said before, you access a database using Data Manipulation Language (DML) statements. Table 10-1 lists all the DML statements you can specify in a COBOL program, along with their formats.

Table 10-1. Data Manipulation Language Statements

Opening and Closing a Subschema
READY { { EXCLUSIVE / CONCURRENT } RETRIEVAL / UPDATE } [AT ERROR *stmt*]
FINISH [AT ERROR *stmt*]
Transaction Statements
INITIATE TRANSACTION *id* [USAGE MODE IS { UPDATE / RETRIEVAL }] [AT ERROR *stmt*]
COMMIT TRANSACTION [AT ERROR *stmt*]
ROLLBACK TRANSACTION [AT ERROR *stmt*]
Manipulating Set Connections
CONNECT MEMBER { *fcn* / *recna* } TO SET *setna* [AT ERROR *stmt*]
DISCONNECT MEMBER { *fcn* / *recna* } FROM SET *setna* [AT ERROR *stmt*]
RECONNECT MEMBER *fcn* TO SET *setna* [AT ERROR *stmt*]
Manipulating Record Occurrences
STORE *recna* [ASSIGN *fcn*] [AT ERROR *stmt*]
GET [*id-1,...* FROM] { *recna* / *fcn* } [AT ERROR *stmt*]
MODIFY [*id-1,...* FROM] { *recna* / *fcn* } [AT ERROR *stmt*]
ERASE { *recna* / *fcn* } [AT ERROR *stmt*]

(continues)

Table 10-1. Data Manipulation Language Statements

Condition Checking

```
IF { [ { recna | fcn } IS [NOT] MEMBER WITHIN setna ]
   | [ setna IS [NOT] { MEMBER | OWNER } ]
   | [ { recna | fcn | setna } IS [NOT] NULL ]
   | [ setna IS [NOT] EMPTY ] } THEN stmt

CHECK STATUS OF stat id-1 id-2 [AT ERROR stmt]
```

Locating a Record Occurrence

```
FIND { FIRST | LAST | NEXT | PRIOR | id | int } MEMBER recna WITHIN setna [ASSIGN fcn] [AT ERROR stmt]

FIND { FIRST | LAST } MEMBER recna WITHIN setna USING { SORT KEY | fn-1, ... }
     [ASSIGN fcn] [AT ERROR stmt]

FIND { NEXT | PRIOR } MEMBER recna WITHIN setna WITH DUPLICATE { SORT KEY | fn-1,... }
     [ASSIGN fcn] [AT ERROR stmt]

FIND CURRENT { fcn | recna WITHIN setna } [ASSIGN fcn] [AT ERROR stmt]

FIND OWNER recna WITHIN setna [ASSIGN fcn] [AT ERROR stmt]
```

(concluded)

The following discussion presents the DML statements in the order in which you are likely to use them. While the specific order of the statements depends on your particular program's needs, certain statements must precede or follow others.

You must follow certain prerequisites when using DML statements in a COBOL program: You must READY a database before you can begin a transaction. You must begin (INITIATE) a transaction before you access the database. You must end (COMMIT) a transaction before you start another one (which also makes your modifications visible to other users). And you must perform a FINISH statement to close the database to yourself when you are done with it.

READY

The first DML statement you must use when you wish to access the database is the READY statement. Think of the READY statement as an OPEN statement for the database. The options for the READY statement exist because, unlike an ordinary file, many users may access a database simultaneously.

You cannot specify the name of the database; your subschema provides the information needed to find the right database. This means that you cannot use more than one subschema in the same program. If you need access to more information in your database than any one subschema provides, have your DBA create a new subschema for you. Most programs should execute a READY statement only once, because this statement uses a lot of overhead.

INITIATE

After you READY a subschema, you must INITIATE a transaction. A *transaction* is a user-defined set of DML and Procedure Division statements. You signal the end of a transaction with the COMMIT or ROLLBACK statement (described later), Each INITIATE-COMMIT sequence defines a DG/DBMS transaction.

You must execute all DML statements, except READY and FINISH, from within a transaction; i.e., enclosed within an INITIATE-COMMIT pair. If you attempt to execute a DML statement outside a transaction, DG/DBMS rejects it. If you attempt to INITIATE a transaction when another one is in progress, DG/DBMS rejects the second INITIATE. In normal usage, you will follow READY by many INITIATE-COMMIT sequences.

FIND

Now that you have finished the preliminaries, you must position yourself somewhere in the database to begin work. Use a series of FIND statements to locate a particular member record occurrence in a set occurrence. We discuss this in more detail in the section *Positioning* later in this chapter.

STORE, GET, MODIFY, ERASE, CONNECT, DISCONNECT, RECONNECT, and the ASSIGN Clause

You use these statements and this clause to manipulate the database. A STORE is analogous to a WRITE. A FIND followed by a GET is analogous to a READ. MODIFY changes data in an existing record occurrence. An ERASE is analogous to a DELETE. CONNECT, DISCONNECT, and RECONNECT reorganize member record occurrences in set occurrences. ASSIGN assigns a free cursor.

If you fail to position the relevant set and record cursors before using any of the statements, DG/DBMS returns a NULL CURSOR error. However, you can STORE into a system set without positioning first. See the *DG/DBMS Reference Manual* for a description of how each statement updates cursors.

COMMIT

COMMIT ends the series of DML statements that began with the last INITIATE. It also allows your updates to become visible to other users who INITIATE transactions after this COMMIT.

ROLLBACK

The ROLLBACK command discards all the modifications you have made to the database since your last INITIATE. Use this command to abort a transaction when your program has encountered inconsistent data or an error from which it cannot recover.

FINISH

When you are through with the database, FINISH closes it. If you do a FINISH while your last transaction is still active (you haven't done either a ROLLBACK or a COMMIT), DG/DBMS assumes that the transaction is still incomplete and internally issues a ROLLBACK. DG/DBMS does this in order to leave the database in a known, consistent state.

Positioning Within a Database

It is important that you understand positioning and how cursors affect database access. Therefore, you should read the section *Cursors and Currency* in Chapter 5 of the *DG/DBMS Reference Manual* before proceeding with this section.

The database system uses cursors to keep track of your position in the database. A *cursor* is a type of pointer. There are, at any given time, several different cursors in your database. There are two categories for cursors: free cursors and system cursors.

As mentioned earlier, free cursors are user-controlled cursors that you set to a particular record occurrence. Use free cursors to explicitly mark a position in the database so that you can reposition to that record later, regardless of where the system cursors point. Except for a few special cases, the system never alters free cursors. We list these exceptions under the appropriate statement descriptions given later in this chapter.

All cursors initially point to nothing (null cursors). When you access a record, DG/DBMS sets system cursors. There are two types of system cursors: record cursors and set cursors. When you access a record occurrence, you position the record cursor to that occurrence. This then becomes the *current of record* for the record type involved. DG/DBMS positions set cursors on the last record accessed in each set type. *Current of set* can therefore be on either an owner or a member record.

As illustrated by Figure 10-7, when we initially find an occurrence of record type ALPHA in set type SYSTEM-ALPHA, current of set for both sets SYSTEM-ALPHA and ALPHA-BETA points to that occurrence of ALPHA (position #1).

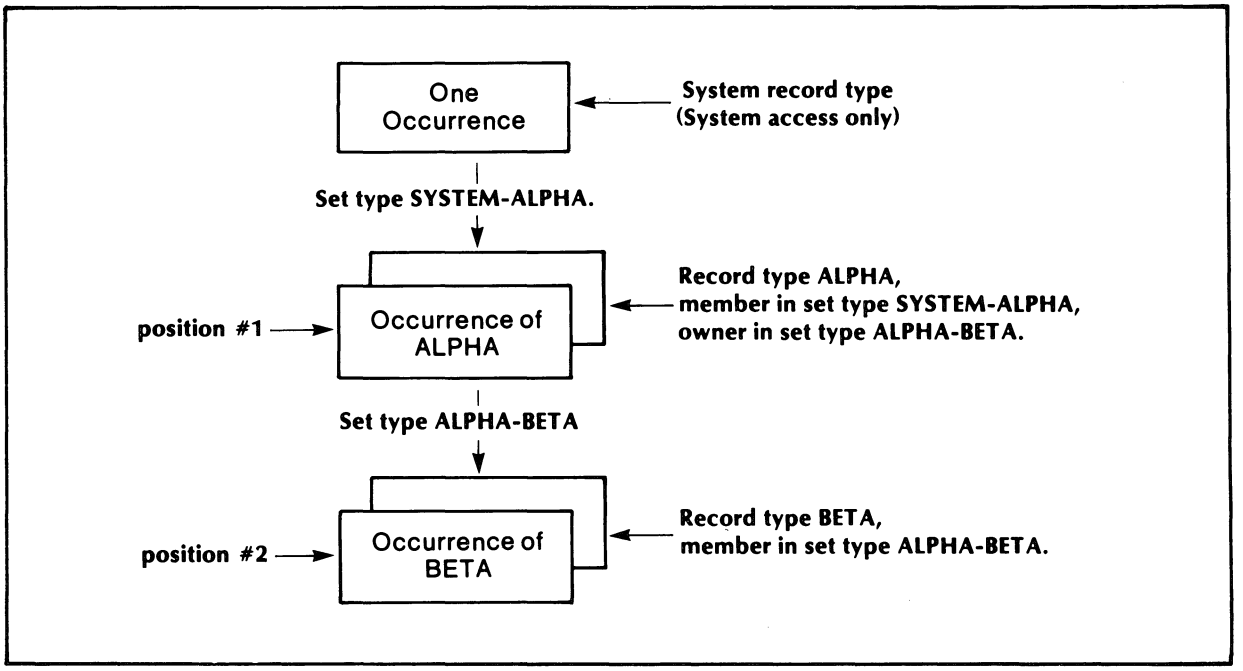
When we subsequently find an occurrence in record type BETA, current of set for ALPHA-BETA points to that occurrence of BETA (position #2). Current of set for SYSTEM-ALPHA still points to position #1.

The locations of your various cursors determine which occurrences the system will use for statements involving their related set/record types. For example, a CONNECT requires specifying a record occurrence to connect and a set type (assuming you do not use free cursors). DG/DBMS CONNECTs the member record occurrence to the owner record occurrence in the set occurrence marked by current of set for that set type. Current of set can point either to an owner record occurrence or to another member record occurrence. However, DG/DBMS always CONNECTs to the owner in the current set occurrence.

Error Handling

You always want to trap DG/DBMS errors when they occur during program execution. There are three ways to do this:

1. Check the DBMS-STATUS data item
2. Use the AT ERROR clause in statements that allow it
3. Specify a Declaratives section



SD-02102

Figure 10-7. Cursor Positions

The best way to check for a specific or expected error (e.g., END-OF-SET) is to check the DBMS-STATUS data item located in the Data Division of your program. Using this method will trap all errors from those DG/DBMS statements that do not contain an AT ERROR clause. For a complete list of error codes, see the *DG/DBMS Reference Manual.*

A more general approach is to use the AT ERROR clause. If specified for a particular statement, this clause traps any DG/DBMS error that occurs on that statement.

You can also specify a COBOL Declaratives Section to trap on DG/DBMS errors. Follow the standard rules for setting up this section (see Chapter 6). In the USE statement, specify the following:

USE AFTER DB-EXCEPTION

If an error occurs on a DG/DBMS statement, the COBOL runtime system first checks if an AT ERROR clause is present in the statement that generated the error. If there is one, COBOL executes it. If there is none, COBOL looks for a DB-EXCEPTION clause; if one exists, COBOL executes it.

Subprograms

Separately compiled subprograms (invoked by a COBOL CALL statement) can access a database. If a subprogram contains DML statements, it must also contain a READY and a FINISH statement. If you attempt to READY the database in the main program or a subprogram and then issue DML statements in another subprogram, the results will be unpredictable.

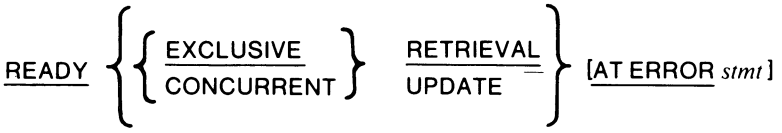
DML Statement Reference Section

This section presents each DG/DBMS DML statement in detail; we have organized them functionally.

READY

Opens a database file.

Format



Where:

stmt is any valid COBOL statement.

Statement Execution

The READY statement opens a database, alllowing access to it through the subschema specified in the program's Subschema Section.

EXCLUSIVE prevents any other user from accessing the database. You cannot specify READY EXCLUSIVE while another user has the database open.

CONCURRENT permits other users to access the database while you are accessing it.

UPDATE permits your program to modify the database.

RETRIEVAL allows your program to examine the database. It does not, however, allow your program to modify the database.

If you include the AT ERROR clause and DG/DBMS returns an error, control passes to *stmt*.

The READY statement sets all cursors to null.

FINISH

Closes a database.

Format

FINISH [AT ERROR *stmt*]

Where:

stmt is any valid COBOL statement.

Statement Execution

FINISH closes the database to your program. Note that that database itself will not close unless all users have closed it.

If you include the AT ERROR clause and DG/DBMS returns an error, control passes to *stmt*.

FINISH deletes all your free cursors, and sets your system cursors to null.

INITIATE

Starts a transaction.

Format

INITIATE TRANSACTION *id* [USAGE MODE IS { UPDATE / RETRIEVAL }] [AT ERROR *stmt*]

Where:

id is a 10-digit numeric data item you defined in Working-Storage.

stmt is any valid COBOL statement.

Statement Execution

INITIATE starts a transaction in DG/DBMS. DG/DBMS returns a transaction number in *id;* the program may use this number as a backup/recovery aid.

UPDATE allows the program to modify the database in this transaction.

RETRIEVAL allows your program to examine the database. It does not, however, allow your program to modify the database. You may not INITIATE a transaction in UPDATE mode if you READYed your database in RETRIEVAL mode.

If you include the AT ERROR clause and DG/DBMS returns an error, control passes to *stmt*.

INITIATE has no effect on cursors. However, other programs' COMMITted transactions may change the records to which your cursors point.

You can write all transaction numbers returned through INITIATE statements to an AOS sequential file. You may also include some key application data fields. If a crash occurs, you can use the AOS file (along with the CHECK command described later) to restart the application program at the appropriate place.

COMMIT

Ends a transaction, making your modifications visible to all other users of the database who subsequently INITIATE a transaction.

Format

COMMIT TRANSACTION [AT ERROR *stmt*]

Where:

stmt is any valid COBOL statement.

Statement Execution

Until you COMMIT a transaction, other users do not see the modifications you have made to the database. After the COMMIT, you need a new INITIATE command to further access or modify the database.

If you include the AT ERROR clause and DG/DBMS returns an error, control passes to *stmt*.

COMMIT has no effect on any cursors.

Note that COMMIT is an "inexpensive" command. DG/DBMS made all the database modifications during the program transactions; COMMIT simply makes these modifications visible to other users.

To end a transaction and abort all changes made during it, use the ROLLBACK command (described later).

ROLLBACK

Ends a transaction, aborting all changes made to the database in that transaction.

Format

ROLLBACK TRANSACTION [AT ERROR *stmt*]

Where:

stmt is any valid COBOL statement.

Statement Execution

ROLLBACK discards changes made to the database file since the start of the transaction. You must INITIATE a new transaction before you can again access the database.

If you specify the AT ERROR clause and DG/DBMS returns an error, control passes to *stmt*.

ROLLBACK resets all cursors to their values at the time the transaction was INITIATEd.

CONNECT

Connects the named record occurrence to the current of set for the named set type.

Format

CONNECT MEMBER { *recna* / *fcn* } TO SET *setna* [AT ERROR *stmt*]

Where:

recna is a record name.

fcn is a free cursor name.

setna is a set name. The current of set in *setna* defines the owner for the connection.

stmt is any valid COBOL statement.

Statement Execution

DG/DBMS connects the record indicated by current of record or by a free cursor to the owner record within the current of set defined by *setna*. You must be positioned within an occurrence of *setna* and have either a record cursor or a free cursor on the member record.

If you specified AT ERROR and DG/DBMS returns an error, control passes to *stmt*.

CONNECT has no effect on free cursors. It sets the current of record cursor to the connected record occurrence. It sets the current of set cursor for *setna* to the connected record occurrence.

DISCONNECT

Disconnects the referenced member record from the named set.

Format

DISCONNECT MEMBER { *recna* / *fcn* } FROM SET *setna* [AT ERROR *stmt*]

Where:

recna is a record name.

fcn is a free cursor name.

setna specifies the set type from which DG/DBMS disconnects the record *id*.

stmt is any valid COBOL statement.

Statement Execution

This statement disconnects the member record on which you are currently positioned or which you have marked by a free cursor from its owner in *setna*.

You cannot disconnect an occurrence in a mandatory set type. (You may delete the member record occurrences.)

If you specified AT ERROR and DG/DBMS returns an error, control passes to *stmt*.

DISCONNECT has no effect on free cursors. It sets current of record in the member record type to the disconnected record. It leaves current of set in the disconnected set on the new "hole" in the accessed set occurrence.

RECONNECT

Disconnects and then connects an occurrence of a member record type within the same set type. Also updates current of set in the same set type.

Format

RECONNECT MEMBER *fcn* TO SET *setna* [AT ERROR *stmt*]

Where:

fcn is a free cursor name.

setna is the name of the set type within which DG/DBMS moves the indicated record occurrence.

stmt is any valid COBOL statement.

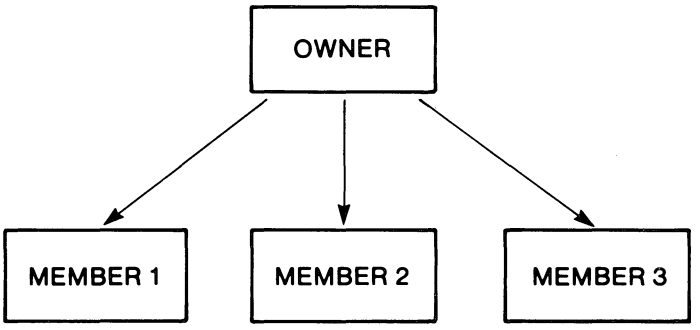
Statement Execution

DG/DBMS disconnects the record occurrence marked with free cursor *fcn* from set type *setna* and then connects it to the current of set in *setna*.

If you specified the AT ERROR clause and DG/DBMS returns an error, control passes to *stmt*.

RECONNECT has no effect on free cursors. It sets current of record to the reconnected member occurrence. It sets current of set to the reconnected member occurrence in its new set occurrence.

In addition to using RECONNECT to change the owner of a member record occurrence, you can use it to reorder members in a set occurrence. For example:



If ORDER IS NEXT, a free cursor is on MEMBER 1, and current of set is on MEMBER 2, then RECONNECTing *fcn* to *setna* will place MEMBER 1 after MEMBER 2

STORE

Stores the information residing in the User Work Area into the database.

Format

STORE *recna* [ASSIGN *fcn*] [AT ERROR *stmt*]

Where:

recna is the name of a Subschema Section record item (01-level).

fcn is a free cursor name.

stmt is any valid COBOL statement.

Statement Execution

STORE creates a new occurrence of the record type *recna*. STORE affects only the fields that the subschema includes in *recna* (according to standard COBOL Data Division rules). For each AUTOMATIC set, DG/DBMS connects the occurrence in the position determined by the set's ORDER IS clause (FIRST, LAST, NEXT, PRIOR, KEY). DG/DBMS connects the new record to the set occurrence defined by current of set. The new record occurrence becomes current of record and current of set for all sets in which the record is an owner or AUTOMATIC member. STORE has no effect on free cursors.

If you specify the ASSIGN clause, the free cursor marks the STOREd occurrence of the record type.

If you specified the AT ERROR clause and DG/DBMS returns an error, control passes to *stmt*.

GET

Moves the record pointed to by the current of record or by a free cursor into the User Work Area.

GET [*id-1,* ... [FROM]] { *recna* / *fcn* } [AT ERROR *stmt*]

Where:

id-1 is a field in record type *recna.*

recna is a record type.

fcn is a free cursor name.

stmt is any valid COBOL statement.

Statement Execution

The GET command moves data from the database into the User Work Area.

You must use the FROM clause if you specify field names in *id-1.* If you do not use the FROM clause, DG/DBMS retrieves the entire record. DG/DBMS retrieves information from the database at the current of record or free cursor position.

Record types *(recna)* are 01-level items in the Subschema Section.

If you specify the AT ERROR clause and DG/DBMS returns an error, control passes to *stmt.*

GET has no effect on free cursors or set cursors. The current of record is set to the retrieved occurrence.

MODIFY

Replaces information in the database with information from the User Work Area.

Format

MODIFY [*id-1,* ... FROM] { *recna* / *fcn* } [AT ERROR *stmt*]

Where:

id-1 is a field in a record type.

recna is a record type.

fcn is a free cursor name.

stmt is any valid COBOL statement.

Statement Execution

MODIFY moves all fields specified in the command into the current of record or free cursor occurrence, overwriting the information in the database. If you specify fields for *id-1,* you must use the FROM clause; the other fields in the occurrence do not change. If you do not specify any fields for *id-1,* DG/DBMS reads the entire record (all the fields) from the User Work Area.

If you specify AT ERROR and DG/DBMS returns an error, control passes to *stmt.*

MODIFY has no effect on free cursors. The current of record is set to the modified occurrence. The current of set is set to the modified record for all sets in which the occurrence is an owner or member. Note that DG/DBMS automatically reorders the members in a sorted set occurrence if you modify the sort key in a connected member.

ERASE

Deletes a record occurrence from the database.

Format

ERASE { *recna* / *fcn* } [AT ERROR *stmt*]

Where:

recna is a record type.

fcn is a free cursor name.

stmt is any valid COBOL statement.

Statement Execution

ERASE deletes the record occurrence indicated by current of record in *recna* or free cursor *fcn* from the database. You cannot ERASE a record occurrence if it is the owner of another record occurrence in any set; you must ERASE all its member records first.

If you specify the AT ERROR clause and DG/DBMS returns an error, control passes to *stmt*.

ERASE sets the following to null:

- free cursors that were pointing to the ERASEd occurrence
- current of record for this record type
- current of set for all sets that the ERASEd record owned

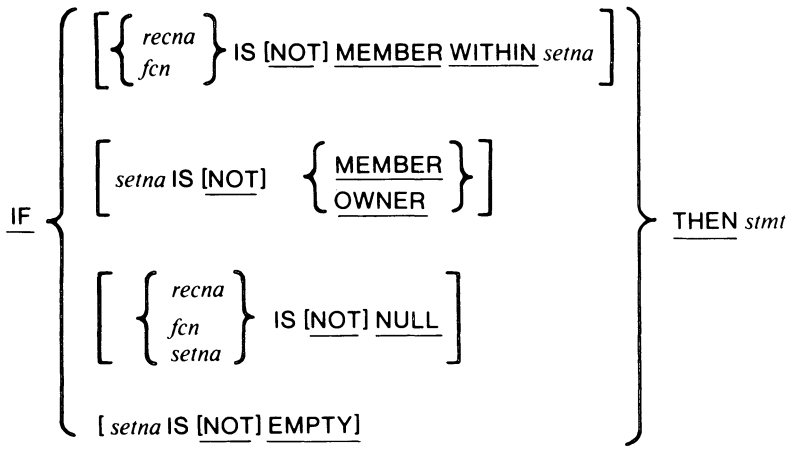
ERASE sets the current of set for all sets in which the ERASEd record was a connected member to the "hole" the record left in the set occurrence.

Note that a DG/DBMS ERASE always causes an immediate, physical deletion. The only way to undo an ERASE command is to ROLLBACK the transaction instead of COMMITting it.

IF

Tests the status of the database with special clauses.

Format



Where:

recna is a record type.

fcn is a free cursor name.

setna is a set type.

stmt is any valid COBOL statement.

These clauses are extensions to the COBOL IF statement's format. *This format does not include the entire set of optional clauses for the COBOL IF statement.* Refer to the IF statement in Chapter 7 for a complete description of this statement.

Statement Execution

Use these four clauses to test the status of the database.

1. [{ *recna* / *fcn* } IS [NOT] MEMBER WITHIN *setna*]

Tests to see if a record occurrence is connected in a set type.

2. [*setna* IS [NOT] { MEMBER / OWNER }]

Tests to see if the current of set is on an owner or a member record.

3. [{ *recna* / *fcn* / *setna* } IS [NOT] NULL]

Tests to see if a free cursor, current of record, or current of set is null (not set to a specific occurrence in the set or record type).

4. [*setna* IS [NOT] EMPTY]

Tests to see if the current of set has member record occurrences.

The IF statement never affects any cursors.

CHECK
Check the status of a transaction.

Format

CHECK STATUS OF *id-1 id-2* [AT ERROR *stmt*]

Where:

id-1 is a 10-digit numeric COBOL data item containing the transaction number.

id-2 is a one-digit numeric COBOL data item into which DG/DBMS returns a transaction status number.

stmt is any valid COBOL statement.

Statement Execution

When you issue the CHECK command, DG/DBMS returns a one-digit number describing the status of the transaction into *id-2*. The following is a list of these codes and their meanings:

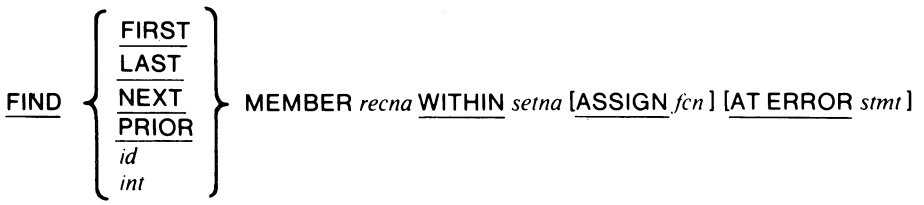
Code #	Meaning
0	Unknown transaction
1	Transaction currently active
2	Transaction successfully completed
3	Transaction backed out (ROLLBACK completed)
4	System error; transaction exists but DG/DBMS cannot determine its status

The CHECK command never affects any cursors.

FIND (positional)

Locates record occurrences by relative positioning.

Format



Where:

id is a data item in the Working-Storage Section containing an integer value.

int is a literal integer.

fcn is a free cursor name.

recna is a record name.

setna is the set type that should contain the selected member record.

stmt is any valid COBOL statement.

Statement Execution

FIND positional moves you through occurrences of a record type within a given set occurrence.

FIRST locates the first occurrence of the record type in the current set occurrence.

LAST locates the last occurrence of the record type in the current set occurrence.

NEXT locates the next occurrence of the record type from current of set within the current set occurrence.

PRIOR locates the immediately previous occurrence of the record type from current of set within the current set occurrence.

If you specify a positive *id* or *int*, DBMS locates the record that is *id* or *int* occurrences from the beginning of the set. If you specify a negative *id* or *int*, DBMS locates the record that is *id* or *int* from the end of the set.

ASSIGN *fcn* assigns a free cursor to the found record occurrence.

If you specify the AT ERROR clause and DG/DBMS returns an error, control passes to *stmt*.

FIND has no effect on free cursors. It sets current of record to the found record occurrence and sets current of set to the located record in all set types in which the record is an owner or a connected member.

FIND (using data items)

Locates the occurrence of a record type containing a specific value(s) in one or more fields.

Format

```
FIND { FIRST } MEMBER recna WITHIN setna USING { SORT KEY }
     { LAST  }                                 { fn-1, ... }

[ASSIGN fcn] [AT ERROR stmt]
```

Where:

recna is the name of the record type.

setna is the name of the set type.

fn1... is a list of one or more fields in the record.

fcn is a free cursor name.

stmt is any valid COBOL statement.

Statement Execution

Use this statement to FIND a record occurrence for which you know the contents of a specific field(s).

FIRST gives you the first occurrence of the record in the current set occurrence where the values for the listed fields or sort keys match the values found in the User Work Area.

LAST gives you the last occurrence of the record in the current set occurrence where the values for the listed fields or sort keys match the values found in the User Work Area.

SORT KEY tells DG/DBMS to use the fields defined as sort keys in the subschema. Use this option for record types that are SORTED BY KEY only.

ASSIGN *fcn* assigns a free cursor to the found record occurrence.

If you specify the AT ERROR clause and DBMS returns an error, control passes to *stmt*.

FIND has no effect on free cursors. It sets current of record to the found record occurrence. It sets current of set to the located record in all set types in which the record is an owner or a connected member.

FIND (duplicates)

Locates occurrences of records having identical fields.

Format

```
FIND { NEXT } MEMBER recna WITHIN setna WITH DUPLICATE { SORT KEY }
     { PRIOR }                                         { fn-1,...  }

[ASSIGN fcn] [AT ERROR stmt]
```

Where:

recna is the data name of the record type.

setna is the name of the set type.

fn-1,... is a list of one or more fields in the record.

fcn is a free cursor name.

stmt is any valid COBOL statement.

Statement Execution

DG/DBMS searches, from current of set, for the NEXT or PRIOR record occurrence that is within the current set occurrence and that has the same values for the specified field(s) as those found in the current of set record occurrence. Use this statement in conjunction with the FIND using data items statement. FIND using data items locates the first or last occurrence. FIND duplicates finds all other occurrences that have the same values for the specified field(s).

ASSIGN assigns a free cursor to the found record.

SORT KEY tells DG/DBMS to use the fields defined as sort keys in the subschema. Use this option for record types that are SORTED BY KEY only.

If you specify the AT ERROR clause and DG/DBMS returns an error, control passes to *stmt*.

FIND has no effect on free cursors. It sets current of record to the found record occurrence. It sets current of set to the located record in all set types in which the record is an owner or connected member.

FIND (current)

Locates the current of record, current of set, or the occurrence assigned to a free cursor.

Format

FIND CURRENT { *fcn* | *recna* WITHIN *setna* } [ASSIGN *fcn*] [AT ERROR *stmt*]

Where:

fcn is a free cursor name.

recna is the 01-level record name.

setna is the desired set name.

stmt is any valid COBOL statement.

Statement Execution

The FIND CURRENT statement is the only FIND that does not locate a new record occurrence. The statement resets all the system cursors associated with a particular record type to point to the same, previously known, record occurrence. Current of record, current of set, or a free cursor indicates this previously known record occurrence.

FIND has no effect on free cursors. It sets current of record to the found record occurrence. It sets current of set to the located record in all set types in which the record is an owner or a connected member.

Specify FIND CURRENT *recna* WITHIN *setna* to reposition all relevant cursors to the record pointed to by the current of set cursor *(setna)*. If *setna* does not point to an occurrence of *recna*, DG/DBMS returns an error. (Remember: current of set can be on an owner or a member record occurrence.)

Specify FIND CURRENT *recna* to reposition all relevant cursors to the record pointed to by the current of record cursor *(recna)*.

Specify FIND CURRENT *fcn* to reposition all relevant cursors to the record pointed to by the free cursor *(fcn)*.

ASSIGN assigns a free cursor to the found record.

If you specify the AT ERROR clause and DG/DBMS returns an error, control passes to *stmt*.

FIND (owner)

Locates the owner of an occurrence in a set type.

Format

FIND OWNER *recna* WITHIN *setna* [ASSIGN *fcn*] [AT ERROR *stmt*]

Where:

recna is the 01-level record name of the owner record type.

setna is the set name.

fcn is a free cursor name.

stmt is any valid COBOL statement.

Statement Execution

FIND OWNER locates the owner record of the set occurrence indicated by the current of set cursor for *setna*.

ASSIGN assigns a free cursor to the found record.

If you specify the AT ERROR clause and DG/DBMS returns an error, control passes to *stmt*.

FIND has no effect on free cursors. It sets current of record to the found record occurrence. It sets current of set to the located record in all set types in which the record is an owner or a connected member.

Sample COBOL Programs Using DG/DBMS

The COBOL program in Figure 10-8 uses our hospital subschema example to find all the patients under the care of Dr. Brian Hackenbush and to output that information to the terminal.

```
       IDENTIFICATION DIVISION.
       PROGRAM-ID. DBMS-EXAMPLE-I.
       AUTHOR.  HARRIS CHASEN.
       ENVIRONMENT DIVISION.
       CONFIGURATION SECTION.
       SOURCE-COMPUTER.          ECLIPSE.
       OBJECT-COMPUTER.          ECLIPSE.
       INPUT-OUTPUT SECTION.
       FILE-CONTROL.
       DATA DIVISION.
       FILE SECTION.
       SUBSCHEMA SECTION.
              COPY ":UDD:HARRIS:TREATMENT_DATABASE:PATIENT_SEARCH.COB".
       C      SUBSCHEMA NAME IS "PATIENT-SEARCH"
       C         WITHIN ":UDD:HARRIS:TREATMENT_DATABASE"
       C         ALLOWS ERASE GET MODIFY STORE
       C
       C      SET NAME IS DOCTORS-BY-NAME
       C          ALLOWS RECONNECT
       C          OWNER IS SYSTEM
       C          MEMBER IS DOCTOR
       C          AUTOMATIC MANDATORY
       C*         ORDER IS SORTED BY KEY ASCENDING
       C*             KEYS ARE:
       C*                LAST-NAME
       C*                FIRST-NAME
       C*                DUPLICATES ALLOWED
       C*         MEMBER LIMIT IS NONE
       C
       C      SET NAME IS PATIENTS-BY-NAME
       C          ALLOWS RECONNECT
       C          OWNER IS SYSTEM
       C          MEMBER IS PATIENTS
       C          AUTOMATIC MANDATORY
       C*         ORDER IS SORTED BY KEY ASCENDING
       C*             KEYS ARE:
       C*                LAST-NAME
       C*                FIRST-NAME
       C*         DUPLICATES ALLOWED
       C*         MEMBER LIMIT IS NONE
       C
       C      SET NAME IS PATIENT-TREATMENTS
       C          ALLOWS RECONNECT
       C          OWNER IS PATIENTS
       C          MEMBER IS TREATMENTS
       C          AUTOMATIC MANDATORY
       C*         ORDER IS NEXT
       C*         MEMBER LIMIT IS NONE
       C
       C      SET NAME IS DOCTOR-TREATMENTS
       C          ALLOWS RECONNECT
       C          OWNER IS DOCTOR
       C          MEMBER IS TREATMENTS
       C          AUTOMATIC MANDATORY
       C*         ORDER IS NEXT
       C*         MEMBER LIMIT IS NONE
       C
       C          DBMS STATUS IS DBMS-STATUS.
```

Figure 10-8. Sample Program Number 1 (continues)

```
C
C 01    DOCTOR ALLOWS ERASE GET MODIFY STORE
C       05  LAST-NAME  PIC X(25)
C                                           USAGE IS DISPLAY
C                                           ALLOWS GET MODIFY.
C       05  FIRST-NAME  PIC X(20)
C                                           USAGE IS DISPLAY
C                                           ALLOWS GET MODIFY.
C       05  SPECIALTY  PIC X(15)
C                                           OCCURS 5 TIMES
C                                           USAGE IS DISPLAY
C                                           ALLOWS GET MODIFY.
C       05  INFO  PIC X(80)
C                                           USAGE IS DISPLAY
C                                           ALLOWS GET MODIFY.
C       05  BEEPER  PIC S9(4)
C                                           USAGE IS DISPLAY
C                                           SIGN IS TRAILING
C                                           ALLOWS GET MODIFY.
C
C 01    PATIENTS ALLOWS ERASE GET MODIFY STORE
C       05  LAST-NAME  PIC X(20)
C                                           USAGE IS DISPLAY
C                                           ALLOWS GET MODIFY.
C       05  FIRST-NAME  PIC X(15)
C                                           USAGE IS DISPLAY
C                                           ALLOWS GET MODIFY.
C       05  WARD  PIC X(4)
C                                           USAGE IS DISPLAY
C                                           ALLOWS GET MODIFY.
C       05  ROOM  PIC S9(3)
C                                           USAGE IS DISPLAY
C                                           SIGN IS TRAILING
C                                           ALLOWS GET MODIFY.
C
C 01    TREATMENTS ALLOWS GET MODIFY ERASE STORE
C       05  DISEASE  PIC X(100)
C                                           USAGE IS DISPLAY
C                                           ALLOWS GET MODIFY.
C       05  MEDICATION  PIC X(25)
C                                           OCCURS 5 TIMES
C                                           USAGE IS DISPLAY
C                                           ALLOWS GET MODIFY.
C       05  DIET  PIC X(200)
C                                           USAGE IS DISPLAY
C                                           ALLOWS GET MODIFY.
C       05  SPECIAL-INSTRUCTIONS  PIC X(40)
C                                           OCCURS 5 TIMES
C                                           USAGE IS DISPLAY
C                                           ALLOWS GET MODIFY.
WORKING-STORAGE SECTION.
77 TX-NO PIC 9(10).
77 FLAG PIC 9.


PROCEDURE DIVISION.
DECLARATIVES.
DB-ERROR SECTION.
        USE AFTER DB-EXCEPTION.
        DISPLAY "DBMS ERROR: " DBMS-STATUS.
        STOP RUN.
ERROR-END.
        EXIT.
END DECLARATIVES.

BEGIN.
```

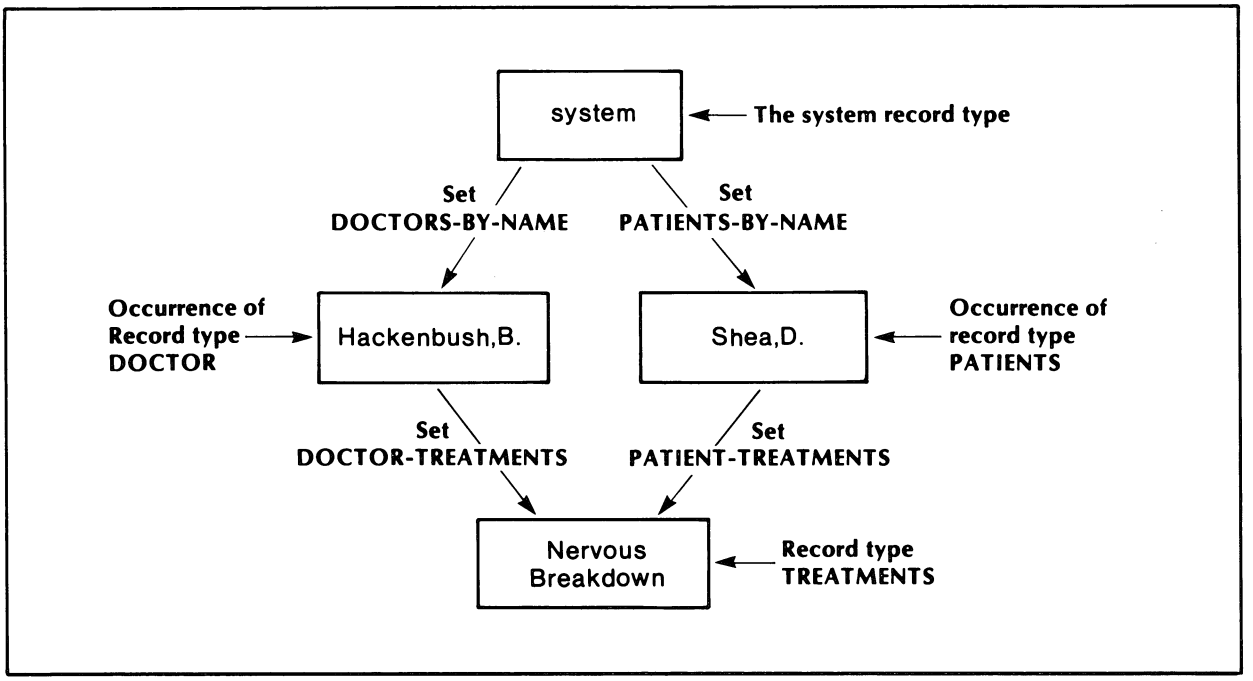
Figure 10-8. Sample Program Number 1 (continued)

```
        MOVE ZEROES TO FLAG.
        MOVE ZEROES TO TX-NO.
OPEN-DATA.
*       OPEN THE DATABASE FILE.
        READY RETRIEVAL.

*       START A TRANSACTION.
        INITIATE TX-NO USAGE RETRIEVAL.

FIND-DOC.
*       WE KNOW THE DOCTOR'S NAME, FIND HIS OCCURRENCE IN THE SET.
        MOVE "HACKENBUSH" TO LAST-NAME OF DOCTOR.
        MOVE "BRIAN" TO FIRST-NAME OF DOCTOR.
        FIND FIRST DOCTOR WITHIN DOCTORS-BY-NAME USING SORT-KEY.

*       FIND THE TREATMENT FOR THE FIRST PATIENT ON THE DOCTOR'S LIST.
        FIND FIRST MEMBER TREATMENTS WITHIN DOCTOR-TREATMENTS
                AT ERROR GO TO ERROR-PARA.

        PERFORM GET-PAT THRU NEXT-PAT UNTIL FLAG = 1.

*       TRANSACTION SUCESSFUL.  END TRANSACTION.
        COMMIT TRANSACTION.
        GO TO STOP-RUN.
GET-PAT.
*       FIND THE PATIENT'S INFORMATION.
        FIND OWNER PATIENT WITHIN PATIENT-TREATMENTS AT ERROR GO TO ERROR-PARA.

*       PUT THE PATIENT INFORMATION INTO THE UWA AND OUTPUT IT.
        GET PATIENTS.
        DISPLAY PATIENTS.
NEXT-PAT.
*       FIND NEXT GETS US THE NEXT TREATMENT RECORD OWNED BY HACKENBUSH
*       WHEN WE RUN OUT OF MEMBERS, DBMS WILL RETURN AN END-OF-SET
*       ERROR AND WE STOP THE RUN.
        FIND NEXT MEMBER TREATMENTS WITHIN DOCTOR-TREATMENTS
                AT ERROR MOVE 1 TO FLAG.

ERROR-PARA.
        DISPLAY "DATABASE ERROR, TRANSACTION ABORTED.".
        DISPLAY "TRANSACTION ID: " TX-NO " ERROR CODE: " DBMS-STATUS.
*       SINCE WE ARE NOT MODIFYING THE DATABASE INFORMATION
*       HERE, ROLLBACK IS REALLY UNNECESSARY.  WE INCLUDE
*       IT FOR ILLUSTRATION.
        ROLLBACK TRANSACTION.

STOP-RUN.
*       CLOSE THE DATABASE.
        FINISH.
        STOP RUN.
```

Figure 10-8. Sample Program Number 1 (concluded)

Figure 10-9 contains a diagram that shows the logic of our program. We first found the Hackenbush record in DOCTORS-BY-NAME. Then, we found Hackenbush's first member in set type DOCTOR-TREATMENTS. This member has another owner in set type PATIENT-TREATMENTS. The PATIENT-TREATMENTS owner is one of the doctor's patients. So, by sequentially looking up the treatment occurrences who are members attached to Hackenbush, and then finding their patient owners, we found all the patients under Dr. Hackenbush's treatment.

In Figure 10-10, we reverse the program. Using the same logic and making only a few modifications, we can find all the doctors treating a patient; in this case, the patient is named John Kelley. Compare the two sample programs. Note that they both use the same subschema.



SD-02104

Figure 10-9. Logical Diagram of Sample Program Number 1

```
IDENTIFICATION DIVISION.
PROGRAM-ID. DBMS-EXAMPLE-II.
AUTHOR.  HARRIS CHASEN.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.          ECLIPSE.
OBJECT-COMPUTER.          ECLIPSE.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
DATA DIVISION.
FILE SECTION.
SUBSCHEMA SECTION.
        COPY ":UDD:HARRIS:TREATMENT_DATABASE:PATIENT_SEARCH.COB".
C       SUBSCHEMA NAME IS "PATIENT-SEARCH"
C          WITHIN ":UDD:HARRIS:TREATMENT_DATABASE"
C          ALLOWS ERASE GET MODIFY STORE
C
C       SET NAME IS DOCTORS-BY-NAME
C           ALLOWS RECONNECT
C           OWNER IS SYSTEM
C           MEMBER IS DOCTOR
C           AUTOMATIC MANDATORY
C*          ORDER IS SORTED BY KEY ASCENDING
C*              KEYS ARE:
C*                 LAST-NAME
C*                 FIRST-NAME
C*          DUPLICATES ALLOWED
C*          MEMBER LIMIT IS NONE
C
C       SET NAME IS PATIENTS-BY-NAME
C           ALLOWS RECONNECT
C           OWNER IS SYSTEM
C           MEMBER IS PATIENTS
C           AUTOMATIC MANDATORY
C*          ORDER IS SORTED BY KEY ASCENDING
C*              KEYS ARE:
C*                 LAST-NAME
C*                 FIRST-NAME
C*          DUPLICATES ALLOWED
C*          MEMBER LIMIT IS NONE
C
C       SET NAME IS PATIENT-TREATMENTS
C           ALLOWS RECONNECT
C           OWNER IS PATIENTS
C           MEMBER IS TREATMENTS
C           AUTOMATIC MANDATORY
C*          ORDER IS NEXT
C*          MEMBER LIMIT IS NONE
C
C       SET NAME IS DOCTOR-TREATMENTS
C           ALLOWS RECONNECT
C           OWNER IS DOCTOR
C           MEMBER IS TREATMENTS
C           AUTOMATIC MANDATORY
C*          ORDER IS NEXT
C*          MEMBER LIMIT IS NONE
C
C           DBMS STATUS IS DBMS-STATUS.
C
```

Figure 10-10. Sample Program Number 2 (continues)

```
C 01     DOCTOR ALLOWS ERASE GET MODIFY STORE
C        05  LAST-NAME  PIC X(25)
C                                              USAGE IS DISPLAY
C                                              ALLOWS GET MODIFY.
C        05  FIRST-NAME  PIC X(20)
C                                              USAGE IS DISPLAY
C                                              ALLOWS GET MODIFY.
C        05  SPECIALTY  PIC X(15)
C                                              OCCURS 5 TIMES
C                                              USAGE IS DISPLAY
C                                              ALLOWS GET MODIFY.
C        05  INFO  PIC X(80)
C                                              USAGE IS DISPLAY
C                                              ALLOWS GET MODIFY.
C        05  BEEPER  PIC S9(4)
C                                              USAGE IS DISPLAY
C                                              SIGN IS TRAILING
C                                              ALLOWS GET MODIFY.
C
C 01     PATIENTS ALLOWS ERASE GET MODIFY STORE
C        05  LAST-NAME  PIC X(20)
C                                              USAGE IS DISPLAY
C                                              ALLOWS GET MODIFY.
C        05  FIRST-NAME  PIC X(15)
C                                              USAGE IS DISPLAY
C                                              ALLOWS GET MODIFY.
C        05  WARD  PIC X(4)
C                                              USAGE IS DISPLAY
C                                              ALLOWS GET MODIFY.
C        05  ROOM  PIC S9(3)
C                                              USAGE IS DISPLAY
C                                              SIGN IS TRAILING
C                                              ALLOWS GET MODIFY.
C
C 01     TREATMENTS ALLOWS GET MODIFY ERASE STORE
C        05  DISEASE  PIC X(100)
C                                              USAGE IS DISPLAY
C                                              ALLOWS GET MODIFY.
C        05  MEDICATION  PIC X(25)
C                                              OCCURS 5 TIMES
C                                              USAGE IS DISPLAY
C                                              ALLOWS GET MODIFY.
C        05  DIET  PIC X(200)
C                                              USAGE IS DISPLAY
C                                              ALLOWS GET MODIFY.
C        05  SPECIAL-INSTRUCTIONS  PIC X(40)
C                                              OCCURS 5 TIMES
C                                              USAGE IS DISPLAY
C                                              ALLOWS GET MODIFY.
WORKING-STORAGE SECTION.
77 TX-NO PIC 9(5).
77 FLAG PIC 9.
PROCEDURE DIVISION.
DECLARATIVES.
DB-ERROR SECTION.
        USE AFTER DB-EXCEPTION.
        DISPLAY "DBMS ERROR: " DBMS-STATUS.
        STOP RUN.
ERROR-END.
        EXIT.
END DECLARATIVES.

BEGIN.
        MOVE ZEROES TO FLAG.
        MOVE ZEROES TO TX-NO.
OPEN-DATA.
```

Figure 10-10. Sample Program Number 2 (continued)

```
*       OPEN THE DATABASE FILE.
        READY RETRIEVAL.

*       START A TRANSACTION.
        INITIATE TX-NO USAGE RETRIEVAL.

FIND-DOC.
*       WE KNOW THE PATIENT'S NAME, FIND HIS OCCURRENCE IN THE SET.
        MOVE "KELLEY" TO LAST-NAME OF PATIENTS.
        MOVE "JOHN" TO FIRST-NAME OF PATIENTS.
        FIND FIRST PATIENT WITHIN PATIENTS-BY-NAME USING SORT-KEY.

*       FIND THE TREATMENT FOR THE PATIENT.
        FIND FIRST MEMBER TREATMENTS WITHIN PATIENT-TREATMENTS
                AT ERROR GO TO ERROR-PARA.

        PERFORM GET-DOC THRU NEXT-DOC UNTIL FLAG = 1.

*       TRANSACTION SUCESSFUL.  END TRANSACTION.
        COMMIT TRANSACTION.
        GO TO STOP-RUN.
GET-DOC.
*       FIND THE DOCTOR.
        FIND OWNER DOCTOR WITHIN DOCTOR-TREATMENTS AT ERROR GO TO ERROR-PARA.

*       PUT THE DOCTOR INFORMATION INTO THE UWA AND OUTPUT IT.
        GET DOCTOR.
        DISPLAY DOCTOR.
NEXT-DOC.
*       FIND NEXT GETS US THE NEXT TREATMENT RECORD OWNED BY KELLEY (ANY OTHER
*       DOCTORS).  WHEN WE RUN OUT OF MEMBERS, DBMS WILL RETURN AN END-OF-SET
*       ERROR AND WE STOP THE RUN.
        FIND NEXT MEMBER TREATMENTS WITHIN PATIENT-TREATMENTS
                AT ERROR MOVE 1 TO FLAG.

ERROR-PARA.
        DISPLAY "DATABASE ERROR, TRANSACTION ABORTED.".
        DISPLAY "TRANSACTION ID: " TX-NO " ERROR CODE: " DBMS-STATUS.
*       SINCE WE ARE NOT MODIFYING THE DATABASE INFORMATION
*       HERE, ROLLBACK IS REALLY UNNECESSARY.  WE INCLUDE
*       IT FOR ILLUSTRATION.
        ROLLBACK TRANSACTION.

STOP-RUN.
*       CLOSE THE DATABASE.
        FINISH.
        STOP RUN.
```

Figure 10-10. Sample Program Number 2 (concluded)

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, Binding, and Executing

There are four stages needed to bring your COBOL program to life. Using one of Data General's text editors, you create a program source 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 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 . . .*] *id-1* [*id-2/L*] [*id-3/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 later in this chapter.

id-1 is a filename that specifies the source program file you want compiled.

id-2 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.

id-3 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 *id-1* with the extension .OB.

You must supply the source filename. If you designate any other optional filenames, be sure to 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.

Virtual Code

We summarize global compiler switches later in this chapter. Note that the /V switch listed there has some special uses and options.

A Word About Overlays

To use the information in this section, you should know what an overlay is. Briefly, an *overlay* is an area of code that exists as a unit within a larger, related unit. The system places an overlay on off-line storage (usually a disk) in an overlay file called filename.OL. When the computer executes a program and needs to use a section of the code in an overlay, it enters the entire overlay segment into a reserved overlay area in memory. Usually this involves overwriting another overlay that was in that area. Later, if the computer needs the first overlay again, it will overwrite the second overlay. This process of entering an overlay into memory is called *paging.*

ANSI Standard Segmentation

Virtual code is an alternative to ANSI Standard segmentation. If your program uses ANSI segmentation, you must specify a number after the section names in your program's Procedure Division. You then use the SEGMENT-LIMIT clause in your program's Object-Computer paragraph to select a boundary. All sections with numbers greater than or equal to this boundary will become overlays; the others will be memory resident. The SEGMENT-LIMIT default boundary number is 50; segment numbers can range from 1 to 99. You can find more information on ANSI Standard segmentation in Chapters 4 and 6.

Virtual code eliminates your need to include this information in your program. When you specify the /V switch, the COBOL compiler automatically breaks the program up into a group of overlays. The default size of these overlays is 2K bytes each. At runtime, when Data General's Advanced Operating System needs to page in a new overlay, it will decide which overlays should remain in memory by paging out the least-recently used overlay.

WARNING: As a default, the COBOL compiler will segment your entire program into 2K-byte chunks. The COBOL binder will create only one overlay area as a default; this means that only one 2K-byte segment can be in memory at any one time. Therefore, virtual code using its defaults is not very efficient.

Beware of situations where the system must execute code that is located physically far away in the source program, particularly for PERFORM statements. Every time the system executes a command specified by a PERFORM statement, the program must page in the overlay containing the PERFORM command, then page back again to execute the code.

You can minimize these problems by using CBIND/C. Read "Binding Programs (CBIND) Using Virtual Code" or ANSI Standard Segmentation" later in this chapter.

A well thought-out program using ANSI Standard segmentation will run far more efficiently than a virtual program. You should, therefore, use ANSI Standard segmentation whenever possible.

Compiling A Program Using Virtual Code

To compile your program using virtual code, specify the /V switch on the compile command line, in this manner:

COBOL [/V] *[//sw...] sourcefile [n/V] [listingfile/L] [objectfile/R]*

You can use either the local or global /V switch in the COBOL command. The first /V switch in the format is a global switch; the second one is a local switch. If you include the local /V switch, you need not specify the global switch; if you use both, the compiler will ignore the global switch.

The global /V switch creates only 2K-byte overlays for your program.

The *n* option using the local /V switch gives the size of the overlays where *n* is an integer that specifies the number of 2K bytes in each overlay. You can make *n* up to the amount of available memory (*n* = available memory (bytes)/2). If *n* is too large, the AOS binder will return an error.

The binder, instead of the compiler, returns this error because you specify the number of overlay areas you want in memory at bind time. As a result, the compiler has no way of knowing how big the total of your overlay areas will be. Also, separately compiled modules may require different overlay areas that you will not specify until all the modules are bound.

You can use any other compiler switches in the compiler command along with the /V switch.

After you have compiled your program using the virtual code option, the COBOL compiler has created a group of object files that you will bind together, using CBIND, to create a program file. You may bind together object files compiled at different times to form one large program (as long as you wrote the source files to work together).

We discuss CBIND later in this chapter. For now, you should know that the COBOL compiler creates an object file called filename.OB and a group of files called filename.nnn that are numbered sequentially. If you use ANSI segmentation, COBOL creates files named filename.section-number. If you use virtual code, COBOL creates files starting at 100 (filename.100, filename.101, filename.102,...). Each filename.nnn file contains one overlay segment. CBIND will bind all of these overlay areas together into one file called filename.OL, mentioned earlier. You use these filenames and the CBIND command line to change the default of one 2K-byte overlay area. "Binding Programs (CBIND) Using Virtual Code" or "ANSI Standard Segmentation", in this chapter, tells you how to do this.

Compiler Switches

You specify global or local switches in the COBOL compilation command line.

Global 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 @LIST.

/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 for virtual code.

/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, it will always be output to @LIST.

*

Local Switches

The local switches you may append to a COBOL filename in the compilation command line are /L and /R. The /L switch allows you to explicitly specify a list file. The /R switch allows you to explicitly specify your object file (filename) and virtual data file (filename .VM).

Example

The following command calls the COBOL compiler to compile a file named FILE1:

```
)COBOL/L/X/W FILE1 FILE1.LS/L)
```

This command compiles the source file FILE1 and produces an object file named FILE1.OB (default name). The listing file FILE1.LS will contain a source listing (/L), a cross-reference table (/X), and error messages (automatically). The compiler will suppress warning messages (/W).

The COBOL Map Switch

The /M switch provides you with a wide range of information about your COBOL program.

Setting the /M switch produces three different pages of information. Table 11-1 lists the types of information on each page.

The first page of the map contains information about the type and structure of your program's Data Division.

The second page of the map contains information about your files.

The third page of the map contains information about the structure of your program's Procedure Division. Two types of information are presented: information about sections and information about paragraphs.

Table 11-1. /M Switch Output

First Page

Heading	Meaning
NAME	The name of the data item as stated in your source program.
TYPE	The format of the data item: **Abbreviation — Meaning** **** — Undeclared Item N — Numeric XN — Extended Numeric FP — Floating Point EFP — Extended Floating Point AN — Alphanumeric ANE — Alphanumeric Edited A — Alpha NE — Numeric Edited GRP — Group
CLASS	The type of data item: **Abbreviation — Meaning** CHAR,LO — Character, Low Overpunch Sign CHAR,HI — Character, High Overpunch Sign CHAR,TS — Character, Trailing Sign CHAR,LS — Character, Leading Sign CHAR,NS — Character, No Sign PACKED — Packed Decimal BINARY — Binary FLOAT — Floating Point
LOC	The displacement of the first byte in the data item. The location number is an octal number. COBOL uses position 000000 as the location of the first data item. When the data is virtual, COBOL uses position 000000 as the location of the first byte on each page.
SIZE	The size of the data item, in bytes. The size is a decimal number.
SCL	The scale factor of the data item (see Chapter 5).
SIG	The significance of a scaled data item.
L	A flag bit that indicates whether the data item is a linkage item.

(continues)

Table 11-1. /M Switch Output

First Page

Heading	Meaning
O	A flag bit that indicates whether the data item contains an OCCURS DEPENDING clause.
J	A flag bit indicating that the data item is right justified.
B	A flag bit indicating blank when zero.
F/B	Father/Brother. If the sign is negative, the number indicates the father. If the sign is positive, the number indicates the brother. A father is the group data item containing the data item referred to by the F/B code. Father data items have lower level numbers than their sons and appear physically above their sons in the source code. A brother is a data item that is on the same level as the considered data item and that appears directly below it in the source code. The last item in a group of brothers contains the father's number. The numbers listed in the F/B column refer to the reference numbers that the map provides in the first column.
SON	The number of the first subordinate data item attached to the considered data item. The son number refers to the reference numbers in the first column of the map.
OF	Occurs Father. Similar to the father number in the F/B column, this column indicates that the data item referenced contains an OCCURS clause.
OS	Occurs Son. Similar to the son number, this column indicates that the data item referenced contains an OCCURS clause.

If your program uses virtual data, COBOL provides additional information on the next line for each data item:

Statement	Meaning
VIRTUAL DATA	The item is virtual data.
PAGE n	The number of the page on which the system has placed the data.
OFFSET m	The decimal location of the first byte in the data word, relative to the page containing the data.

Second Page

Heading	Meaning
FILES	The name of the file. The word FILE appears on each line.
ACC	File Access Type: **Abbreviation** **Meaning** SEQ Sequential RAN Random DYN Dynamic
ORG	COBOL will indicate three different types of organization: **Type** **Meaning** SEQUENTIAL Sequential access file. RELATIVE Relative access file. INDEXED INFOS system indexed file (ISAM or DBAM).

(continued)

Table 11-1. /M Switch Output

Second Page	
Heading	**Meaning**
RECORD	The record specified for your file by a DATA RECORDS clause.
PACKET ADDR.	COBOL packet information is for internal use only. Do not attempt to access or modify the COBOL packet. Unpredictable results will occur.
Third Page	
Header	**Meaning**
(none)	The name of the section or paragraph.
PARA SECT	The words PARA or SECT will indicate whether the item is a section or a paragraph.
SECTION FIRST PARA	If the item is a section, this column contains the name of the first paragraph in the section. If the item is a paragraph, this column contains the name of the section it belongs to.
LAST PARA	If the item is a section, this column contains the name of the last paragraph in the section. The column is blank if the item is a paragraph.

(concluded)

Figure 11-1 shows a sample COBOL source program listing and Figure 11-2 shows the map the compiler produces for it.

```
0001    IDENTIFICATION DIVISION.
0002    PROGRAM-ID. CTSVIRT.
0003
0004    ENVIRONMENT DIVISION.
0005    INPUT-OUTPUT SECTION.
0006    FILE-CONTROL.
0007
0008    DATA DIVISION.
0009    FILE SECTION.
0010    WORKING-STORAGE SECTION.
0011    VIRTUAL-STORAGE SECTION.
0012
0013    * PAGE 0
0014    01      FFA.
0015      02            FF1 PIC X(1024).
0016      02            FF2 PIC X(1024).
0017
0018    * PAGES 1,2
0019    01      FFB.
0020      02            FF3 PIC X(48).
0021      02            FF4 PIC X(2001).
0022
0023    * PAGES 2,3
0024    01      FFC.
0025      02            FF5 PIC X(2047).
0026      02            FF6 PIC X(48).
0027      02            FF7 PIC X(2000).
0028
```

Figure 11-1. Sample COBOL Source Program Listing (produced with /L compiler switch) (continues)

```
0029    * MANY MORE PAGES
0030    01      FFD.
0031      02    FF8     PIC X(2000).
0032    D 02    FF9     PIC X(30000).
0033
0034    01      FFE.
0035      02    FF10    PIC X(2000).
0036    D 02    FF11    PIC X(30000).
0037
0038    PROCEDURE DIVISION.
0039
0040            DISPLAY 'BEGIN VIRTUAL TEST'.
0041
0042    * INITIALIZE DATA
0043            MOVE    ALL '1' TO FF1.
0044            MOVE    ALL '2' TO FF2.
0045            MOVE    ALL '3' TO FF3.
0046            MOVE    ALL '4' TO FF4.
0047            MOVE    ALL '5' TO FF5.
0048            MOVE    ALL '6' TO FF6.
0049            MOVE    ALL '7' TO FF7.
0050            MOVE    ALL '?' TO FFD.
0051            MOVE    ALL '?' TO FFE.
0052            DISPLAY '       1 PAGE TESTS'.
0053            MOVE    FF2 TO FF1.
0054            MOVE    FF7 TO FF6.
0055            MOVE    FF7 TO FF6.
0056
0057            DISPLAY '       2 PAGE TESTS'.
0058            MOVE    FF6 TO FF3.
0059            MOVE    FF5 TO FF1.
0060
0061            DISPLAY '       3 PAGE TESTS'.
0062            MOVE    FF4 TO FF2.
0063            MOVE    FF5 TO FF6.
0064            MOVE    FF5 TO FF6.
0065
0066    * CHECK FOR ERRORS
0067            IF FF1 NOT = ALL '5' THEN DISPLAY 'ERROR 1'.
0068            IF FF2 NOT = ALL '4' THEN DISPLAY 'ERROR 2'.
0069            IF FF3 NOT = ALL '7' THEN DISPLAY 'ERROR 3'.
0070            IF FF4 NOT = ALL '4' THEN DISPLAY 'ERROR 4'.
0071            IF FF5 NOT = ALL '5' THEN DISPLAY 'ERROR 5'.
0072            IF FF6 NOT = ALL '5' THEN DISPLAY 'ERROR 6'.
0073            IF FF7 NOT = ALL '7' THEN DISPLAY 'ERROR 7'.
0074
0075    * MORE 3 PAGE TESTS
0076            MOVE    FFB TO FFA.
0077            MOVE    FFC TO FFB.
0078
0079    * CHECK FOR ERRORS
0080            IF FF2 NOT = ALL '4' THEN DISPLAY 'ERROR 8'.
0081            IF FFB NOT = ALL '5' THEN DISPLAY 'ERROR 9'.
0082
0083            DISPLAY 'END OF VIRTUAL TEST'.
```

Figure 11-1. Sample COBOL Source Program Listing (produced with /L compiler switch (concluded)

```
FILE:   CTSVIRT2

DATA ITEMS
    NAME                    TYPE CLASS     LOC    SIZE SCL SIG LOJB F/B  SON OF OS
  2 FF1                       AN  CHAR,NS 000000  1024          ----   3    0
        VIRTUAL DATA      PAGE 0  OFFSET 0
 14 FF10                      AN  CHAR,NS 003722  2000          ---- -13    0
        VIRTUAL DATA      PAGE 4  OFFSET 2002
  3 FF2                       AN  CHAR,NS 002000  1024          ----  -1    0
        VIRTUAL DATA      PAGE 0  OFFSET 1024
  5 FF3                       AN  CHAR,NS 000000    48          ----   6    0
        VIRTUAL DATA      PAGE 1  OFFSET 0
  6 FF4                       AN  CHAR,NS 000060  2001          ----  -4    0
        VIRTUAL DATA      PAGE 1  OFFSET 48
  8 FF5                       AN  CHAR,NS 000002  2047          ----   9    0
        VIRTUAL DATA      PAGE 2  OFFSET 2
  9 FF6                       AN  CHAR,NS 000001    48          ----  10    0
        VIRTUAL DATA      PAGE 3  OFFSET 1
 10 FF7                       AN  CHAR,NS 000061  2000          ----  -7    0
        VIRTUAL DATA      PAGE 3  OFFSET 49
 12 FF8                       AN  CHAR,NS 000002  2000          ---- -11    0
        VIRTUAL DATA      PAGE 4  OFFSET 2
  1 FFA                      GRP  CHAR,NS 000000  2048          ----   4    2
        VIRTUAL DATA      PAGE 0  OFFSET 0
  4 FFB                      GRP  CHAR,NS 000000  2049          ----   7    5
        VIRTUAL DATA      PAGE 1  OFFSET 0
  7 FFC                      GRP  CHAR,NS 000002  4095          ----  11    8
        VIRTUAL DATA      PAGE 2  OFFSET 2
 11 FFD                      GRP  CHAR,NS 000002  2000          ----  13   12
        VIRTUAL DATA      PAGE 4  OFFSET 2
 13 FFE                      GRP  CHAR,NS 003722  2000          ----   0   14
        VIRTUAL DATA      PAGE 4  OFFSET 2002
```

Figure 11-2. COBOL Map Produced by Program in Figure 11-1

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.

Binding 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 binds your main file and subprograms together into a single program image.

The system normally places COBOL programs and runtime routines in shared memory partitions. Of course it always places data in unshared memory.

Issue the CBIND command in the following format:

CBIND [*/sw/sw* . . .] *id-1* [*id-2* . . .] [ICALL]

Where:

sw is a global switch if appended to CBIND, and a local switch if appended to *id-1*. 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.

id-1 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.

id-2 is a filename that specifies a subprogram you want included in the loaded program.

ICALL is the COBOL interface to the INFOS system (supplied by INFOS) which you need if you use INFOS indexed files. Either use LFE on ICALL to add it to VRT.LB or include ICALL on the CBIND command line.

CBIND Global 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.

/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. (name.CK performs the bind.)

/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 96-word stack (128 if you use the ID switch).

CBIND Local Switches

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 unshared data. This is currently the only way that you can create an unshared data partition, if you are using the macroassembler.

/H Load unshared code in this module as shared code. If you apply this switch to an unshared 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.

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.

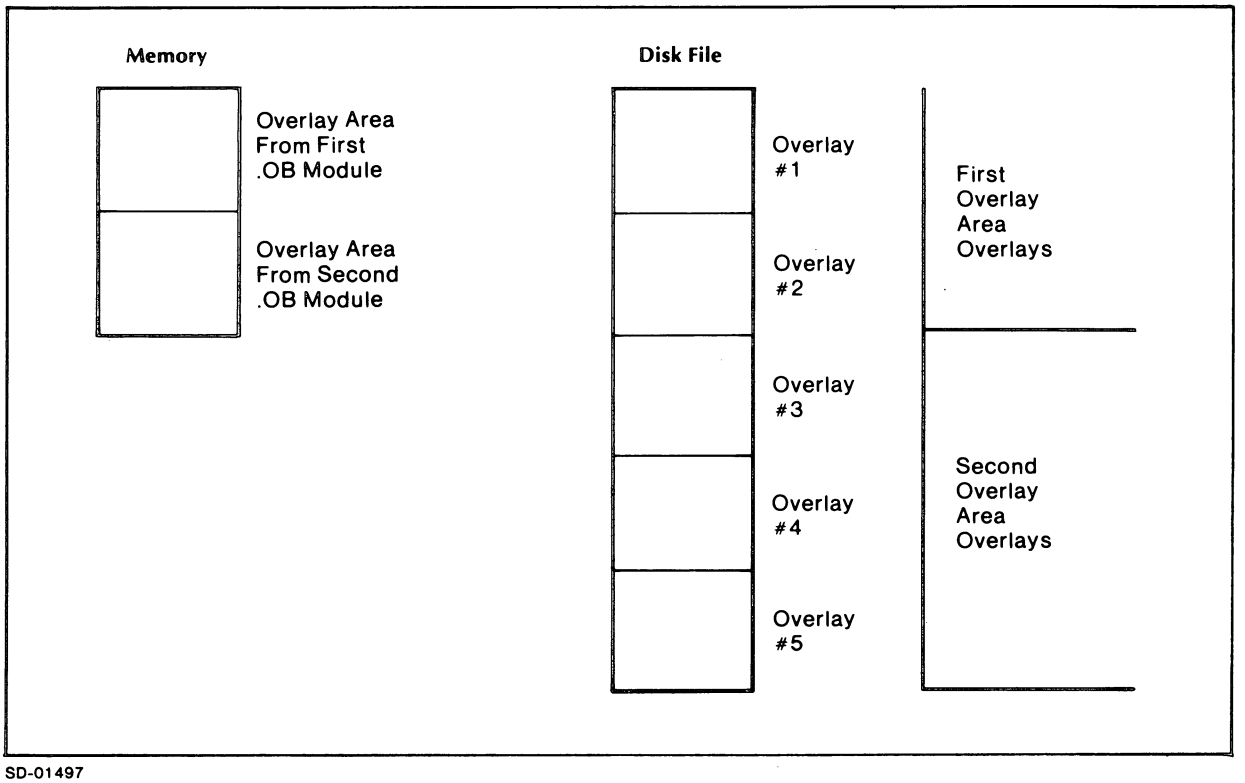
Binding Programs (CBIND) Using Virtual Code or ANSI Segmentation

Binding programs without making modifications to the compiler-created overlay structure is easy. Simply type:

CBIND *programname*

The AOS binder will then create an executable program file from your object file(s). Note that when you specified the /V switch in the COBOL compile command, the compiler created a series of files named programname.number. COBOL creates files with numbers less than 100 from ANSI segmentation numbers; the number of the file is the number you assigned to the segment in your program. The COBOL compiler creates files with numbers greater than 99; these numbers do not appear anywhere in the source program.

These files define your overlay areas for programname's compiled module. When you bind your modules together, CBIND will create one overlay area in memory and place all the overlays in one contiguous file called filename.OL. When your program executes, it will page overlays from the overlay file into the program's reserved memory area only. Figure 11-3 illustrates a slightly more sophisticated environment with two overlay areas.



SD-01497

Figure 11-3. Example of an Overlay and Overlay Area Structure with Two Object Modules

You can change an organization like the one shown in Figure 11-3, by using the /C (terminal) switch on the CBIND command line:

CBIND/C *[/sw /sw...] program 1 program 2*

Specifying the /C switch on CBIND will start an interactive dialog to reorganize your overlays. In the /C dialog, *node* is another term for overlay area.

The /C dialog creates a nodefile, which is a record of your responses to the prompts. You can use the nodefile on future CBINDs to recreate this overlay structure.

An explanation of the /C dialog follows:

PROGRAM: programname
NODEFILE: nodefilename

OVERLAY SUMMARY

programname HAS number OVERLAYS

ANSWER THE FOLLOWING QUESTIONS WITH AN INTEGER OR A NEWLINE.
NEWLINE SELECTS THE DEFAULT VALUE WHICH IS DISPLAYED WITHIN [].

HOW MANY OVERLAY NODES SHOULD THIS PROGRAM HAVE? [1]

The default displayed always equals one. The binder initially creates only one overlay area for any given program. This command resets the number of overlay areas your program will contain in memory.

HOW MANY BASIC OVERLAY AREAS IN OVERLAY NODE number? [1]

The computer will repeat this question for each overlay area. The first overlay area is overlay area 0. A Basic Overlay Area is a quantity determined at compile time. If you are using ANSI segmentation, your Basic Overlay Area will equal the largest segment in your program. If you are using virtual code only, your Basic Overlay Area will equal the number you specified with the /V switch in the compiler command times 2,096 (2K) bytes. If you did not specify a number with the /V switch, your Basic Overlay Area will equal 2K bytes.

You may wish to create overlay areas that are larger than your overlays. If you do this, COBOL will enter as many overlays as it can into the overlay area. COBOL enters these overlays as the program needs them. When COBOL fills the overlay area, the next overlay needed in memory will cause COBOL to page out the least recently used overlay which is currently in memory to make room. Please remember how large you make your overlays and overlay numbers; you will need this information for the next section.

ASSIGN A NODE NUMBER TO EACH OVERLAY.
THE NODE NUMBERS MUST BE IN THE RANGE FROM 0 TO number.
programname number [0]

Use this question to assign your overlays to an overlay area. Specify the number of the overlay area given in the previous question. If you have more than one overlay area for your program, the prompt *programname number [0]* will repeat for each overlay in the program.

DURING A FUTURE CBIND USING THIS NODEFILE, THERE MAY BE OVERLAYS PRESENT THAT ARE NOT PRESENT TODAY. THESE WILL BE ASSIGNED TO THE "DEFAULT OVERLAY NODE". ENTER THE DEFAULT OVERLAY NODE NUMBER. [0]

You use this statement in conjunction with the /NODEFILE switch. If you plan to CBIND your program again later, using different object files, COBOL will assign these files to the default overlay area. Read the section "The /NODEFILE switch," in this Chapter for details.

THANK YOU.

Your input is complete

You can also reorganize your overlay structure by editing your binder command line. Do this by using the /Q switch on CBIND. The binder will create a binder command line in a file called programname.CK and then stop without performing the bind. Later you will issue the AOS command [programname.CK]. This will cause the AOS binder to bind your programs. The section of the binder command line created by COBOL that we are concerned with looks like this:

prog1 [prog1.100,prog1.101...].

Here, *prog1* is your program and *prog.100* etc. are your overlays. The square brackets [] indicate your overlay areas; the overlays contained in a set of brackets are all assigned to the same overlay area. If you wish to increase the number of overlay areas, increase the number of sets of square brackets. If you wish to change the assignment of your overlays, place your .100 etc. files in different brackets. If you wish to increase the size of an overlay area, use the switch /AM= *number* after the close bracket] symbol for the overlay area you wish to increase. *number* is the number of Basic Overlay Areas included in the overlay area.

NOTE: Assigning only one overlay to an overlay area makes that overlay memory resident. Assigning no overlays to an overlay area wastes memory space. Assigning no overlay area to an overlay will cause unpredictable results; your program will probably not run.

For example, let's take the following program structure:

```
Name of the program : WARREN
Number of overlays: 5
WARREN.100
WARREN.101
WARREN.102
WARREN.103
WARREN.104
```

The original (compiler-generated) configuration is:

- One overlay area (0).
- All five overlays assigned to overlay area.
- Size of overlay area set to 2K bytes.

```
WARREN [WARREN.100,WARREN.101,WARREN.102,WARREN.103,WARREN.104]
```

The new user-created configuration is:

- Two overlay areas.
- Three overlays assigned to overlay area 0.
- Two overlays assigned to overlay area 1.
- Size of overlay area 0 set to 4K bytes (2K words).
- Size of overlay area 1 defaulted to 2K bytes.

```
WARREN [WARREN.103,WARREN.101,WARREN.100]/AM=2 [WARREN.102,WARREN.104]
```

The /NODEFILE Switch

You can use the overlay specifications set in an earlier CBIND to bind a new program file. (See "Binding Programs (CBIND) Using Virtual Code or ANSI Segmentation" for information about how to do this.) use the /NODEFILE switch:

CBIND/NODEFILE= *nodefilename programname [programname...]*

or

CBIND/NODEFILE *programname [programname...]*

When you use the second format, COBOL will find the overlay specifications in the nodefile for the first program. When you first use CBIND/C to bind your program for Virtual Code, COBOL creates a file containing your overlay specifications, called programname.CN. The /NODEFILE switch tells COBOL to re-use this .CN file.

If you bind new object files into the program, COBOL assigns these files' overlays to the default overlay node that you specified if you originally bound your program using the /C switch.

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 (name.CM).

After editing the command file, execute the load with the command name.CK. 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 XEQ programname at the terminal. 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, 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, the following 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. If your program does not include special code to handle I/O exceptions, COBOL displays a warning message and continues execution as best as it can (COBOL treats the I/O exception as a successful operation). If you specified any of the following, COBOL assumes the program will handle the exception and does not display a warning:

- INFOS system or COBOL status items
- Declaratives section
- Invalid Key (on Invalid-key type errors)
- AT END (on end-of-file errors)

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 father process. 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 Programmer's Manual.* There is no error-handler option; the program's execution terminates and control passes to the father process. 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 Programmer's 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 (*filename*)
SEGMENT (*number*)
RELATIVE LOCATION (*loc*)

Where:

filename 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, filename.PR, the name of the subprogram, B, which actually called the subroutine. The address given in the trace information is the relative address in filename.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
ACCESSIBILITY
ADD
ADVANCING
AFTER
ALL
ALLOW
ALLOWS
ALPHABETIC
ALSO
ALTER
ALTERNATE
AND
APPROXIMATE
ARE
AREA
AREAS
ASCENDING
ASCII
ASSIGN
AT
AUTHOR
AUTO
AUTOMATIC
BACKWARD
BECOMES
BEFORE
BELL
BIT
BLANK
BLINK
BLOCK
BOTTOM
BY
CALL
CANCEL
CD
CF
CH
CHANNEL
CHARACTER
CHARACTERS
CHECK
CLOCK-UNITS
CLOSE
COBOL
CODE
CODE-SET
COL
COLLATING
COLUMN
COMMA
COMMIT
COMMUNICATION
COMP
COMPRESSION
COMPUTATIONAL
COMP-1
COMPUTATIONAL-1
COMP-2
COMPUTATIONAL-2
COMP-3
COMPUTATIONAL-3
COMPUTE
CONCURRENT
CONFIGURATION
CONNECT
CONNECTED
CONTAINS
CONTIGUOUS
CONTROL
CONTROLS
COPY
CORR
CORRESPONDING
COUNT
CR
CREATE
CURRENCY
CURRENT
CURSOR
DATA
DATA-SENSITIVE
DATE
DATE-COMPILED
DATE-WRITTEN
DAY
DBMS
DB-EXCEPTION
DE
DEBUG-CONTENTS
DEBUG-ITEM
DEBUG-LINE
DEBUG-NAME
DEBUG-SUB-1
DEBUG-SUB-2
DEBUG-SUB-3
DEBUGGING
DECIMAL-POINT
DECLARATIVES
DEFINE
DELETE
DELIMITED
DELIMITER
DEPENDING
DESCENDING
DESTINATION
DETAIL
DISABLE
DISCONNECT
DISK
DISPLAY
DIVIDE
DIVISION
DOWN
DUPLICATE
DUPLICATES
DYNAMIC
EBCDIC
EGI
ELSE
EMI
EMPTY
ENABLE
END
END-OF-PAGE
ENTER
ENVIRONMENT
EOP
EQUAL
ERASE
ERROR
ESCAPE
ESI
EVEN
EVERY
EXCEPTION
EXCLUDE
EXCLUSIVE
EXIT
EXPIRATION
EXPUNGE
EXTEND
FD
FEEDBACK
FIELD
FIELDS
FILE
FILE-CONTROL
FILLER
FINAL
FIND
FINISH
FIRST
FIX
FIXED
FOOTING
FOR
FORWARD
FROM
FULL
GENERATE
GENERATION
GENERIC
GET
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
KEYBOARD
KEYS
LABEL
LABELS
LAST
LEADING
LEFT
LENGTH
LESS
LEVELS
LIMIT
LIMITS
LINAGE
LINAGE-COUNTER
LINE-COUNTER
LINE
LINES
LINK
LINKAGE
LOCAL
LOCK
LOGICAL
LOW-VALUE
LOW-VALUES
LRU
MANAGEMENT
MANDATORY
MANUAL
MAXIMUM
MEMBER
MEMORY
MERGE
MERIT
MESSAGE

MODE
MODIFY
MODULES
MOVE
MULTIPLE
MULTIPLY
NAME
NATIVE
NEGATIVE
NEXT
NO
NODE
NONE
NOT
NULL
NUMBER
NUMERIC
OBJECT-COMPUTER
OBTAIN
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
PREVIOUS
PRINTER
PRINTER-1
PRINTING
PRIOR
PROCEDURE
PROCEDURES
PROCEED
PROGRAM
PROGRAM-ID
PROTECTED
QUEUE
QUOTE
QUOTES
RANDOM
RD
READ
READY
RECEIVE
RECONNECT
RECORD
RECORDING
RECORDS
REDEFINES
REEL
REFERENCES
RELATIVE
RELEASE
REMAINDER
REMOVAL
RENAMES
REPLACING
REPORT
REPORTING
REPORTS
REQUIRED
RERUN
RESERVE
RESET
RETAIN
RETRIEVAL
RETRIEVE
RETURN
REVERSED
REWIND
REWRITE
RF
RH
RIGHT
ROLLBACK
ROOT
ROUNDED
RUN
SAME
SAVE
SCREEN
SD
SEARCH
SECONDS
SECTION
SECURE
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
STORE
STRING
SUB-INDEX
SUB-QUEUE-1
SUB-QUEUE-2
SUB-QUEUE-3
SUBSCHEMA
SUBTRACT
SUM
SUPPRESS
SWITCH
SYMBOLIC
SYNC
SYNCHRONIZED
SYSTEM
TABLE
TALLYING
TAPE
TEMPORARY
TERMINAL
TERMINATE
TEXT
THAN
THEN
THROUGH
THRU
TIME
TIME-OUT
TIMES
TO
TOP
TRAILER
TRAILING
TRANSACTION
TRUNCATE
TYPE
UNDEFINED
UNDELETE
UNIT
UNLOCK
UNSTRING
UNTIL
UP
UPDATE
UPON
USAGE
USE
USER
USING
VALUE
VALUES
VARIABLE
VARYING
VERIFY
VIRTUAL
VIRTUAL-COMMON
VIRTUAL-STORAGE
VOLUME
WAIT
WHEN
WITH
WITHIN
WORDS
WORKING-STORAGE
WRITE
ZERO
ZEROES
ZEROS

End of Appendix

Appendix B
CS Compatibility

Data General's CS systems' Interactive COBOL is a subset of AOS COBOL. You can code programs to run on both systems with a minimum of difficulty. Since AOS COBOL is a superset of CS COBOL, you should use the CS Interactive Cobol manual to write programs coded for use on both types of systems, not the AOS COBOL Reference Manual.

There are slight differences in syntax between the two "flavors" of this language. This appendix summarizes those differences and discusses how to transport programs and files between the two systems.

When writing a program on either system that you wish to run on both systems, you should have the following available:

- CS/AOS COBOL Compatibility manual.
- The Data General *Interactive COBOL Programmer's Reference Manual,* 045-000-011;
- This appendix ;
- The "Screen Section" in Chapter 5 of this manual;
- "The COBOL Interactive Debugger," Chapter 9 in this manual;
- "How To Use COBOL under AOS," Chapter 10 in this manual.

You will probably want to keep this appendix with its equivalent in the CS Interactive COBOL manual.

AOS-CS Differences

We discuss the differences between AOS and CS COBOL in this section. We show the differences like this:

PROBLEM: A description of the problem.

AOS: What the AOS system does.
CS: What the CS system does.
Fix: What you should do to make your program compatible.

Identification Division Incompatibilities

PROBLEM: AOS will not accept hyphens in the program ID.

AOS: Converts hyphens to dollar signs.
CS: No action taken.
Fix: None! Simply be aware that AOS will change all the hyphens in your program ID to dollar signs.

Environment Division Incompatibilities

PROBLEM: The SELECT statement.

AOS: A literal must follow the DISPLAY and KEYBOARD clauses.
CS: The literal is optional.
Fix: Always specify a literal when using these clauses.

WARNING: Opening the console as a file will override the AOS Screen Section characteristics, as a result, your screen may not work properly. Do not open the console as a file in a program that uses screen management.

PROBLEM: Filenames.

AOS: Filenames can be up to 32 characters long.
CS: Filenames can be up to 10 characters long.
Fix: Limit the length of your filenames to 10 characters.

PROBLEM: Duplicate alternate keys.

AOS: Does not allow duplicate alternate keys unless you specify ALLOW DUPLICATES.
CS: Always allows duplicate alternate keys. Treats the ALLOW DUPLICATES clause as a comment field.
Fix: Specify ALLOW DUPLICATES on every key (in the SELECT statement).

PROBLEM: INDEX SIZE and DATA SIZE clauses.

AOS: Treats these clauses as comment fields.
CS: Determines the size of contiguous files.
Fix: None! Although your file organization may be different on the two systems as a result, your programs will still run.

PROBLEM: Some FILE STATUS codes have different meanings.

Fix: Check Table B-1 for differences; code your program appropriately.

Table B-1. Differences in FILE STATUS Error Codes

Code	AOS Meaning	CS Meaning
02	Successful completion, duplicate key.	None.
21	None.	Invalid key.
24	Invalid key; key value too large.	Index depth exceeded.
95	OPEN labeled tape error.	None.
96	Record logically deleted.	OPEN error (directory not initialized).
97	Illegal REWRITE or DELETE attempted, indexed file.	Record-lock limit exceeded.
99	Other INFOS system error.	Line printer access table full.

Procedure Division Incompatibilities

PROBLEM: The ACCEPT id statement.

AOS: Does not prompt the user or allow corrections.
CS: Prompts the user and allows corrections.
Fix: Use the ACCEPT screen-name option instead of ACCEPT id.

PROBLEM: ACCEPT statement error conditions.

AOS: Returns a code 99 to the ESCape key in the event of an ACCEPT error.
CS: Does not return an error code.
Fix: Be careful.

PROBLEM: The ACCEPT id FROM EXCEPTION-STATUS command.

AOS: Does not recognize this clause.
CS: Recognizes this clause.
Fix: Do not use this clause in your ACCEPT statements.

PROBLEM: The DELETE statement.

AOS: Defaults to a physical delete.
CS: Always performs a logical delete.
Fix: Always specify logical deletes. Specify DELETE LOGICAL LOCAL GLOBAL.

PROBLEM: The DISPLAY id statement.

AOS: Displays the item(s) at full intensity (bright).
CS: Displays the item(s) at half intensity (dim).
Fix: If you desire consistency, use DISPLAY screen-name.

PROBLEM: The OPEN EXTEND statement.

AOS: Does not lock the file.
CS: Locks the file (performs an Exclusive Open).
Fix: Be careful when you open a file under AOS (lock all your records).

PROBLEM: File Input-Output.

AOS: Does not permit:
1. Opening an existing file for output.
2. Opening a nonexistent file for I-O.
3. OPEN EXTEND for a nonexistent file.

CS: Permits all three.
Fix: Use the AOS conventions for opening files.

PROBLEM: The STOP RUN Statement.

AOS: Returns to the father process.
CS: Chains to logon (LOGON.PR).
Fix: Use CALL PROGRAM "#A" instead of STOP RUN statements.

PROBLEM: Attempting to read a locked record.

AOS: Returns an error.
CS: Does not return an error if you opened your file for input.
Fix: Be careful.

PROBLEM: Logically deleted records.

AOS: Reads the records and returns a warning flag.
CS: Does not read logically deleted records.
Fix: Code your programs to avoid using AOS logically deleted records.

Transporting Files From your Interactive COBOL System To AOS

ICOS ISAM files are not compatible with AOS INFOS ISAM files. The internal structure of the two files is totally different. As a result, to transport an ISAM database file, you must convert your files to sequential files on the CS system before you place them on a tape. Then, you will reconstruct your ISAM file on the AOS system.

To create the sequential file on your CS System, you can use the ICOS REORG utility. Then, you can use the AOS Sort/Merge utility to recreate your ISAM file. Use the CS Interactive COBOL manual to find out about REORG. The *AOS Sort/Merge User's Manual* tells you how to use Sort.

An example of how to perform this file conversion follows. Consult the two manuals mentioned above for a detailed explanation of all the commands. *You must follow the syntax shown in the following example exactly;* do not omit any commas, spaces, etc. or include any extra ones. Also, this example only places one database file on a physical tape.

First, place a blank tape on your CS tape drive and specify on your ICOS system:

REORG/A *filename/I* MTa/T/U

Copy all the information returned to you by REORG. Then, use the AOS INFOS system ICREATE utility to create an ISAM file with the same filename as your CS file. Consult the *INFOS System User's Manual (AOS)* for information about ICREATE.

Next, place the tape on your AOS tape drive, and specify on your AOS system:

```
X SORT/C
INPUT FILE IS '@ MTAb',
RECORDS ARE c CHARS.
OUTPUT INFOS INDEX IS 'filename',
RECORD IS 1/ c.
KEY 1/ d.
COPY.
END.
```

Where:

a is the number assigned to the tape drive on your CS system.

b is the number assigned to the tape drive on your AOS system.

c is from REORG; it's the size of the records.

d is from REORG; it's the length of your record's keys.

Sequential files, random access files, and program source files need little modification. Modify your source programs in line with the exceptions listed in this appendix. Then, dump your files from your CS system, and X RDOS LOAD them onto your AOS System.

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
```

Be careful not to put a start label on the .END statement at the end of your assembler routine. If you do, the binder will take the label as the starting address of the entire run unit instead of the COBOL entry point, and the program will likely trap on execution.

An example of a COBOL-callable assembly language routine is shown in Figure C-1.

```
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-1. AOS Assembly Language Routine Example (continues)

```
; 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-1. AOS Assembly Language Routine Example (concluded)

End of Appendix

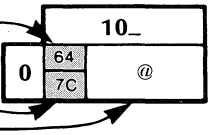
Appendix D
ASCII and EBCDIC Character Sets

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 06	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 8D	[	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 9D	]	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-00217 Character code in octal at top and left of charts.

↑ means CONTROL

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 E
Handling Unlabeled Magnetic Tape

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 E-1.), creates a simple tape file. It has 200-character fixed-length records with 2000 characters per block. The second program, MTAINN.CO (Figure E-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.

To run the two programs in Figures E-1 and E-2, you 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 "12/14/79"
```

This sends to the operator the message:

FROM PID #: REQUEST IS FOR OUTPUT; USE TAPE MARKED "12/14/79".

In response, the operator will mount your tape (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 free the unit for another use.

```
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 E-1. Creating an Unlabeled Magnetic Tape File

```
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 E-2. Reading an Unlabeled Magnetic Tape File

End of Appendix

Appendix F
Language Upgradability From RDOS to 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:
   - 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:

- You may specify your own PAD character in RDOS; AOS uses the null character.
- 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.

Procedure Division (Chapters 6 and 7)

1. The following features function in an RDOS environment. You can use them in AOS, but they have no meaning:

   - REWINDing magnetic tape volumes (OPEN and CLOSE statements); AOS never rewinds;
   - 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;
   - Setting up a work file in a SORT statement (default is WORK.W1); AOS uses the default work filename only.

2. The features which 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, REWRITE, and DELETE statements).

     In the AOS INFOS system any user can use RETRIEVE KEY successfully on a locked record. In RDOS, after encountering a locked record you can use a READ NEXT statement to continue with the next record. AOS will return the same locked record error (AOS will not move). You must RETRIEVE KEY NEXT before you READ NEXT to pass a locked record with relative access.

   - 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).
   - The /V switch in RDOS compiles virtual overlays which are paged in from extended memory; AOS compiles for virtual code. (See Chapter 5, "Virtual - Storage Section.")

4. The Communications Access Manager (CAM) statements do not exist in AOS.

End of Appendix

Index

Within this index, the letter "f" means "and the following page"; "ff" means "and the following pages". Also, primary references are listed first.

01-level data description 5-29, 5-15, 5-20f
66-level data description 5-44f, 5-15, 5-29
77-level data description 5-29, 5-15
88-level data description 5-45, 5-15, 5-29

* (asterisk) indicator
    comment 2-2, 2-5
    zero suppression 5-37f
, currency symbol 5-36f, 2-1
- (hyphen) 2-1f, 2-5
\ (backslash) indicator 2-5

A

/A global compiler switch 11-4, 11-18
A-margin 2-6
ACCEPT statement 7-4f, 5-18ff, 4-3
ACCEPT DATE/DAY/TIME statement 7-6f
ACCESS MODE clause 4-16f
access modes 4-8
ADD statement 7-7f
addition
    ADD statement 7-7f
    operators 6-8
    SET UP/DOWN statement 7-65
address map 11-1
ADVANCING
    clause 5-12
    phrase 6-13f
ALL figurative constant 2-2
ALLOW SUB-INDEX and LEVELS clauses 4-18f
alphabet
    clause 4-4
    name (table) 2-3
alphabetic data
    edited 5-36
    nonedited 5-33
alphanumeric data
    edited 5-33, 5-36
    literal 2-4
    nonedited 5-33
ALTER statement 7-9
ALTERNATE RECORD clause 4-16f
American National Standards Institute 1-1 (see ANSI)
American Standard Code for Information Interchange (see ASCII)
angle brackets (< >) 2-4, (table) 2-1
ANSI (American National Standards Institute) 1-1, 5-17, 5-32, 11-2
apostrophe (') 2-4, (table) 2-1
APPROXIMATE, KEY series phrase 6-17
arguments
    error message 11-17
    subprogramming 6-12
arithmetic
    ADD statement 7-7f
    COMPUTE statement 7-16
    CORRESPONDING phrase 6-7
    DIVIDE statement 7-23
    error message 11-17
    expressions 6-8
    MULTIPLY statement 7-43
    operations 6-5f
    operators (table) 6-8
    referencing data items 6-6
    ROUNDED phrase 6-6
    SET UP/DOWN statement 7-65
    sharing storage areas 6-6
    SIZE ERROR phrase 6-6f
    statement summary 7-1
    SUBTRACT statement 7-75f
array
    about 5-31
    declarations 5-31f
    name qualification 6-4
ASCII (American Standard Code for Information Interchange) 1-2, 2-1, 4-2, 5-13, (table) D-1
assembly language routines C-1ff
ASSIGN clause 4-14f, 10-13
asterisk (*)
    comment line 2-5, 2-2
    zero suppression 5-38, (table) 5-37
AT END phrase 6-19
AT END-OF-PAGE phrase 6-13f
AT ERROR clause 10-15
AUDIT command 9-3, 9-2
AUTHOR paragraph 3-1

B

/B global binder switch 11-10
B insertion character 5-36, (table) 5-37
backslash (/) indicator 2-5
BACKWARD, relative option phrase 6-16ff
binary
  datum 5-27
  number storage (table) 5-27
  object file 3-1, 11-1ff
binding 11-10ff
  about 11-10
  CBIND command 11-10
  DG/DBMS with a COBOL program 10-1
  switches 11-10ff (see also switches)
  using ANSI segmentation 11-12ff
  using virtual code 11-12ff
BLANK WHEN ZERO clause 5-42
BLOCK CONTAINS clause 5-7, 5-3ff
block size 5-7
brackets, angle (< >) 2-4
breakpoint 1-2, 9-2, 9-4, 9-12

C

/C
  global compiler switch 11-4
  local binder switch 11-11, 11-3, 11-13
calendar information 7-6
CALL statement 6-12, 7-10f
  in WALKBACK command 9-14
CALL PROGRAM statement 7-12f
CANCEL statement 7-14
card format 2-4, 11-4
carriage return (CR) 2-1, 2-4
CBIND
  command 11-10 (see binding)
  switches 11-10f (see switches, binder)
CHANNEL clause 4-4
channel name (table) 2-3
character
  ASCII 1-2, 2-1, 4-2, 5-13, (table) D-1
  currency 4-3, 4-5
  EBCDIC 1-2, 5-13, (table) D-2
  set 2-1
  string comparison 4-2
CHECK statement (DG/DBMS) 10-30, (table) 10-12
class condition 6-10
clause
  about 2-6
  data description entry
    BLANK WHEN ZERO 5-42
    JUSTIFIED 5-42
    OCCURS 5-31f
    PICTURE 5-33
    REDEFINES 5-30
    SIGN 5-41
    SYNCHRONIZED 5-41
    USAGE 5-40
    VALUE 5-42
  file-control paragraph
    SELECT 4-9ff, (table) 4-11ff (see SELECT clause)
  file description entry
    CODE-SET 5-13
    DATA RECORD 5-11
    FEEDBACK 5-13f
    LABEL RECORDS 5-9
    LINAGE 5-11f
    MERIT 5-14
    PAD 5-14
    PARTIAL RECORD 5-14
    RECORDING MODE 5-8
    VALUE OF 5-10f
  Object-Computer paragraph
    SEGMENT-LIMIT 4-3
    PROGRAM COLLATING SEQUENCE 4-2
  Special-Names paragraph
    alphabet 4-4
    CHANNEL 4-4
    CURRENCY SIGN 4-5
    DECIMAL-POINT IS COMMA 4-5
    SWITCH 4-4
CLEAR command 9-4, 9-2
CLI command 9-5, 9-2
CLOSE statement 7-15
COBOL
  compilation command 11-2ff
  concepts 2-1ff
    character set 2-1
    literals 2-3ff (see literals)
    separators 2-1
    user-defined words 2-3
  extensions 1-2
  words 2-1ff (see words)
CODE-SET clause 5-13
code set translation 4-3ff, 1-2
collating sequence
  clause 4-2ff
  conversion 1-2, 2-2
column positioning (table) 5-22
comma (,)
  clause 4-5
  insertion character (table) 5-37
  separator 2-1
commands
  ACCEPT 5-18ff
  AUDIT 9-3, 9-2
  CBIND 11-10ff, 11-3
  CLEAR 9-4, 9-2
  CLI 9-5, 9-2
  COBOL 11-2ff
  COMPUTE 9-6, 9-2
  CON 9-7, 9-2
  COPY 9-8, 9-2
  CPRINT 5-25
  debugger, about 9-2
  DISPLAY 9-9
  ENV 9-10, 9-2
  MOVE 9-11, 9-2
  SET 9-12, 9-2

STOP 9-13, 9-2
WALKBACK 9-14, 9-2
XEQ 11-15
comment
entry 3-1
line 2-5, 9-1
COMMIT statement (DG/DBMS) 10-19, 10-13, (table) 10-11
compiling
about 11-1
command 11-2
DG/DBMS with a COBOL program 10-1
error messages 11-9
switches 11-4ff
using ANSI standard segmentation 11-2f
using virtual code 11-2f
warning messages 11-9f
compound expressions 6-10ff
COMPUTATIONAL item 5-40
COMPUTE
command, debugger 9-6, 9-2
statement 7-16
CON debugging command 9-7, 9-2
condition name
condition 6-9
entry 5-45, (table) 2-3
evaluation (IF) 7-29f
qualification 6-3
conditional expressions
compound 6-10ff
simple 6-9ff
class condition 6-10
condition name condition 6-9
relation condition 6-9
switch condition 6-10
Configuration Section 4-2ff
Object-Computer paragraph 4-2f
SEGMENT-LIMIT clause 4-3
Source-Computer paragraph 4-2
PROGRAM COLLATING SEQUENCE clause 4-2
Special-Names paragraph 4-3ff
alphabet clause 4-4
CHANNEL clause 4-4
CURRENCY SIGN clause 4-5
DECIMAL-POINT IS COMMA clause 4-5
SWITCH clause 4-4
CONNECT statement (DG/DBMS) 10-21, 10-13, (table) 10-11
console input/output
statement summary 7-2
ACCEPT statement 7-4f
DISPLAY statement 7-22
constant, figurative 2-2
CONTIGUOUS clause 4-15
continuation lines 2-5
COPY facility
about 8-1f
command 9-8, 9-2
CORRESPONDING phrase 6-7
CPRINT utility 5-25
CR
as NEW LINE 2-1, 2-4
credit sign insertion 5-37
CS compatibility B-1ff
CSIZE utility 5-17
CURRENCY SIGN clause 4-5, 5-38
currency sign insertion 5-36ff, (table) 5-37, (example) 5-38
cursors
free 10-10, 10-14, 10-16ff
system 10-14, 10-16ff

D

/D
global binder switch 11-10
global compiler switch 11-4
local binder switch 11-11
D debug line 2-5
data
description entry 5-28ff (see data description entry)
editing 5-36ff (see editing)
extensions 1-2
floating point 2-3, 5-35
handling 4-1
map 11-4ff
name 2-3, 6-3
type 5-26f (see data type)
Data Base Administrator (DBA) 10-1
data description entry 5-28ff
about 5-28
BLANK WHEN ZERO clause 5-42
examples 5-43
FILLER 5-28
JUSTIFIED clause 5-42
level numbers 5-29
OCCURS clause 5-31f
PICTURE clause 5-33ff (see PICTURE clause)
data editing in 5-36ff
defining items 5-33ff
REDEFINES clause 5-30
SIGN clause 5-41
SYNCHRONIZED clause 5-41
USAGE clause 5-40
VALUE clause 5-42
Data Division 5-1ff
condition name entry 5-45
data description entry 5-28ff (see data description entry)
data type 5-26f (data type)
File Section 5-2ff (see File Section)
Linkage Section 5-8, 5-1 (see Linkage Section)
Screen Section 5-18ff (see Screen Section)
structure 5-1
Subschema Section 10-2ff (see subschema)

Virtual-Storage Section 5-15ff (see Virtual-Storage Section)
Working-Storage Section 5-15, 5-1 (see Working-Storage Section)
data editing 5-36ff (see editing)
Data General/Database Management System Interface (see DG/DBMS interface)
data manipulation and editing
INSPECT statement 7-31ff
MOVE statement 7-40ff
SEARCH statement 7-61f
SET statement 7-64
statement summary 7-1
STRING statement 7-73f
UNSTRING statement 7-81f
Data Manipulation Language (DML) 10-5, 10-11ff (see DML)
data name 6-3, (table) 2-3
DATA RECORD clause 5-11, (format) 5-4
data-sensitive record 1-2, 4-6, 5-7
DATA SIZE clause 4-19
data type 5-26f
alphabetic 5-26
alphanumeric 5-26
alphanumeric edited 5-26
numeric 5-26f
database files 4-7, 10-1ff
DBA (Data Base Administrator) 10-1
DEB command 9-1
debit sign (DB) 5-36f, (table) 5-37
debug line (D) 2-5
debugger
about 9-1f
AUDIT command 9-3, 9-2
binding in with COBOL program 11-10f
CLEAR command 9-4, 9-2
CLI command 9-5, 9-2
comment lines 9-1
COMPUTE command 9-6, 9-2
CON command 9-7, 9-2
COPY command 9-8, 9-2
DISPLAY command 9-9
ENV command 9-10, 9-2
executing the 11-16
MOVE command 9-11, 9-2
SET command 9-12, 9-2
STOP command 9-13, 9-2
WALKBACK command 9-14, 9-2
decimal point (.) insertion character 5-36, (table) 5-37
DECIMAL-POINT IS COMMA clause 4-5
Declaratives Section 6-19f, 10-15
DEFINE SUB-INDEX statement 7-17f
DELETE statement 7-19f
DELETE FILE statement 7-21
delimiters 2-4
DEPENDING clause 5-31
device clause 4-3
DG/DBMS (Data General/Database Management System) interface
about 10-1
compiling and binding 10-1
program, sample 10-36ff, 10-40ff
statements, DML 10-11ff (see statements, DML (DG/DBMS))
subschema 10-2ff (see subschema)
DISCONNECT statement (DG/DBMS) 10-22, 10-13, (table) 10-11
DISPLAY
debugging command 9-9
statement 7-22, 5-18f, 5-25, 4-3
DIVIDE statement 7-23
DML (Data Manipulation Language) 10-5
error handling 10-14f
positioning within a database 10-14, (figure) 10-15
program, sample 10-36ff, 10-40ff
statements (see statements, DML (DG/DBMS))
condition checking 10-28ff, (table) 10-12
locating a record occurrence 10-31ff, (table) 10-12
manipulating record occurrences 10-24ff, (table) 10-11
manipulating set connections 10-21ff, (table) 10-11
opening and closing a subschema 10-16f, (table) 10-11
transaction statements 10-18ff, (table) 10-11
subprograms 10-15
dollar sign ($) insertion character 5-36, (table) 5-37, (table) 2-1
DOWN, relative option phrase 6-16ff
DOWN FORWARD, relative option phrase 6-16ff
DUPLICATES clause 4-16f, (format) 4-10, (table) 4-13
dynamic access mode 4-8

E

/E
global binder switch 11-10
global compiler switch 11-4, 2-4
EBCDIC (Extended Binary-Coded-Decimal Interchange Code) 1-2, 5-13, (table) D-1
editing (data)
alphanumeric/alphabetic 5-36
BLANK WHEN ZERO clause 5-42
examples 5-43
floating insertion 5-39
JUSTIFIED clause 5-42
numeric 5-36ff
PICTURE clause 5-33ff
SIGN clause 5-41
SYNCHRONIZED clause 5-41
USAGE clause 5-40
VALUE clause 5-42

elementary data item 5-29
ENV command 9-10, 9-2
Environment Division
Configuration Section 4-2ff (see Configuration Section)
file access modes 4-8 (see file, access modes)
file organization 4-5ff (see file, organization)
Input-Output Section 4-9ff (see Input-Output Section)
structure 4-1
ERASE statement (DG/DBMS) 10-27, 10-13, (table) 10-11
errors
compiler messages 11-9
DG/DBMS 10-14f
fatal 11-16f
runtime
I/O exception conditions 6-19ff, 11-16
program, fatal 11-16f
program, nonfatal 11-16
system, fatal 11-17
trace 11-18
execution, program
about 11-15
command (XEQ) 11-15
debugger (/D) 11-16
errors (see errors, runtime)
EXIT statement 7-24
EXIT PROGRAM statement 7-25
Extended Binary-Coded-Decimal Interchange Code (see EBCDIC)
exponentiation 6-8, 2-3
expressions
arithmetic 6-8
compound 6-10ff
conditional 6-9f
EXPUNGE statement 7-26
EXPUNGE SUB-INDEX statement 7-27

F

fatal errors 11-16f
FD entry 5-3ff (see file description entry)
FEEDBACK clause 5-13f
field translation 5-13f
figurative constant 2-2
file
about 5-1f
access modes 4-8
File Control paragraph, Environment Division 4-9ff
ACCESS MODE clause 4-16f
ALLOW SUB-INDEX and LEVELS clauses 4-18f
ASSIGN clauses 4-14f
FILE STATUS clause 4-18
INFOS STATUS clause 4-18
KEY COMPRESSION clause 4-19
OPTIONAL clause 4-14
ORGANIZATION clause 4-16
PARITY clause 4-18
RESERVE clause 4-15
SELECT clause 4-9ff, 5-2, 5-16
file description (FD) entry, Data Division 5-3ff
block size 5-7
CODE-SET clause 5-13
DATA RECORD clause 5-11
FEEDBACK clause 5-13f
indexed file format 5-4
LABEL RECORDS clause 5-9
LINAGE clause 5-11f
MERIT clause 5-14
node size 5-7
PAD clause 5-14
PARTIAL RECORD clause 5-14
record size 5-7
RECORDING MODE clause 5-8f
relative file format 5-4
sequential file format 5-3
sort/merge file format 5-4
table of clauses 5-6
VALUE OF clause 5-10f
I/O 5-1ff (see input/output)
organization 4-5ff
database 4-7
indexed (see also indexed file)
about 4-6f
format 4-10, 5-4
table 4-11ff, 5-5f
relative (see also relative file)
about 4-6
format 4-10, 5-4
table 4-11ff, 5-5f
sequential (see also sequential file)
about 4-5f
format 4-9, 5-3
table 4-11ff, 5-5f
sort/merge (see also sort/merge process)
about 4-7
format 4-11, 5-4
table 4-11ff, 5-5f
file-handling statements
CLOSE 7-15 DEFINE SUB-INDEX 7-17f
DELETE 7-19f
DELETE FILE 7-21
EXPUNGE 7-26
EXPUNGE SUB-INDEX 7-27
LINK SUB-INDEX 7-36f
OPEN 7-44f, 4-7
READ, indexed file 7-51f
READ, relative file 7-50
READ, sequential file 7-49
RETRIEVE 7-54f, 4-17
REWRITE, indexed file 7-59f
REWRITE, relative file 7-58
REWRITE, sequential file 7-57
SEEK 7-63
START, indexed file 7-71
START, relative file 7-70

START, sequential file 7-69
TRUNCATE 7-77
UNDELETE 7-78f
UNLOCK 7-80
USE 7-83f
WRITE, indexed file 7-88f
WRITE, relative file 7-87
WRITE, sequential file 7-85f
File Section 5-3ff
FILE STATUS clause 4-18
file status data item 6-20f
filename 2-3
FILLER 5-28
FIND statements (DG/DBMS)
about 10-13
current 10-34
duplicates 10-33
owner 10-35
positional 10-31
table 10-12
using data items 10-32
FINISH statement (DG/DBMS) 10-17, 10-14, (table) 10-11
fixed insertion editing 5-37
fixed-length record 1-2, 4-6, 5-7
floating insertion editing 5-39
floating point data
error message 11-17
external representation 5-35
literal 2-3
footing area, page 5-11f
formatted data 4-3, 5-1
FORWARD, relative option phrase 6-16ff
free cursors 10-10, 10-14, 10-16ff
function delimiter keys, Screen Section (table) 5-19

G

/G global compiler switch 11-4
GENERIC, KEY series phrase 6-17
GET statement (DG/DBMS) 10-25, 10-13, (table) 10-11
global switches (see switches)
GO statement 7-28, 7-9
group data item 5-29, 5-31

H

/H
global binder switch 11-10
local binder switch 11-11
hexadecimal format 11-10
HIGH-VALUE(S) figurative constant 2-2
hyphen (see also minus sign)
continuation line 2-5
in a word 2-1

I

/I global binder switch 11-10
Identification Division 3-1
IF statement
COBOL 7-29f
DG/DBMS 10-28f, (table) 10-12
imperative statement 6-1
INDEX BLOCK CONTAINS clause 5-7
INDEX NODE SIZE clause 5-4f, 5-7
INDEX SIZE clause 4-19, B-2
indexed file
about 4-6f, 1-2
file description entry (format) 5-4, (table) 5-5f
KEY series phrase 6-17f
multilevel indexed file 6-17, (table) 6-15, 1-2, 4-6, 5-8, 5-14
organization 5-4
POSITION phrase 6-15
record options 6-19
record selection 6-15ff
relative option phrase 6-16f
SELECT clause (format) 4-10, (table) 4-11ff
simple indexed file 6-17, 4-6, 5-8
statements
CLOSE 7-15
DEFINE SUB-INDEX 7-17f
DELETE 7-19f
DELETE FILE 7-21
EXPUNGE 7-26
EXPUNGE SUB-INDEX 7-27
LINK SUB-INDEX 7-36f
OPEN 7-44f
READ 7-51f
RETRIEVE 7-54f
REWRITE 7-59f
START 7-71
UNDELETE 7-78f
UNLOCK 7-80
WRITE 7-88f
indicator characters (table) 2-5
INFOS 1-2, 6-20, 11-1
INFOS STATUS clause 4-18
INFOS status data item 6-21
INITIATE statement (DG/DBMS) 10-18, 10-13, (table) 10-11
I/O (see input/output)
Input-Output (I-O) Section 4-9ff
File Control paragraph, SELECT clauses 4-9ff
ACCESS MODE clause 4-16f
ALLOW SUB-INDEX and LEVELS clauses 4-18f
ASSIGN clauses 4-14f
DATA SIZE clause 4-19
examples 4-20
FILE STATUS clause 4-18
INDEX SIZE clause 4-19
INFOS STATUS clause 4-18
KEY COMPRESSION clause 4-19

OPTIONAL clause 4-14
ORGANIZATION clause 4-16
PARITY clause 4-18
RESERVE clause 4-15
I-O Control paragraph 4-21
MULTIPLE FILE clause 4-21
SAME RECORD AREA clause 4-21
input/output
device 4-3
error handling (USE statement) 7-83f
exception conditions 6-19ff
AT END phrase 6-19
Declaratives Section 6-19f
file status data items 6-20
INFOS status data items 6-21
INVALID KEY phrase 6-19
file 4-3, 5-1
language extensions 1-2
statements
ACCEPT 7-4ff, 5-18ff, 4-3
CLOSE 7-15
DISPLAY 7-22, 5-18f, 5-25
DEFINE SUB-INDEX 7-17f
DELETE 7-19f
DELETE FILE 7-21
EXPUNGE 7-26
EXPUNGE SUB-INDEX 7-27
LINK SUB-INDEX 7-36f
MERGE 7-38f, 4-2, 4-4
OPEN 7-44f, 4-7
READ 7-49ff
RELEASE 7-53, 4-7
RETRIEVE 7-54f, 4-17
RETURN 7-56, 4-7
REWRITE 7-57f
SEEK 7-63
SORT 7-66ff, 4-2, 4-4
START 7-69ff
TRUNCATE 7-77
UNDELETE 7-78f, 4-7
UNLOCK 7-80
USE 7-83f, 6-19, 4-6
WRITE 7-85ff
insertion editing
fixed character 5-37
simple character 5-36f
special character 5-36ff
INSPECT statement 7-31ff
INSTALLATION paragraph 3-1
integer 2-3
interactive debugger 9-1ff (see debugger)
INVALID KEY phrase 6-19
inversion, file 4-7
ISAM B-4

J

JUSTIFIED clause 5-42, 5-28

K

/K=n global binder switch 11-10
key 4-7
about 4-7, 1-2
approximate 6-17, 1-2
compression 4-13, 4-19
length (KEY LENGTH clause) 4-17, 6-18
KEY COMPRESSION clause 4-19
KEY LENGTH clause 4-17, 6-18
KEY series phrase 6-17f
APPROXIMATE 6-17
GENERIC 6-17
KEY LENGTH 6-18
keyword (table) A-1, 2-2

L

/L
global binder switch 11-11, 11-4
global compiler switch 11-4, (figure) 11-7f
local compiler switch 11-4
/L=name global binder switch 11-11
LABEL RECORDS clause 5-9
labeled magnetic tape 4-6, 4-23, 5-9
level numbers 5-29
LEVELS clause 4-18f
library, COPY file 8-1
LINAGE clause 5-11f, 6-13f, 2-2
LINAGE-COUNTER 2-2, 6-14
line
about 2-4
and column positioning (table) 5-22
comment 2-5
continuation 2-5
debug 2-5
number (LINAGE-COUNTER) 2-2, 6-14
sentence 2-5
statement 2-5
text format 2-4
link loading 11-12f
LINK SUB-INDEX statement 7-36f
Linkage Section 5-18, 5-1
listing file 11-2, 11-4
literals 2-3
alphanumeric 2-4
numeric 2-3
loading, link 11-12f
local switches (see switches)
logical
block 5-7
operators 6-11
page 5-11
LOW-VALUE(S) figurative constant 2-2

M

/M global compiler switch 11-5ff, 11-1, 11-4, 5-17
magnetic tape
  labeled 4-6, 4-23, 5-9
  unlabeled E-1ff
main program 11-10
Map switch (/M) 11-5ff, 11-1, 11-4, 5-17
margins, page 5-11f
MEMORY SIZE clause 4-2
MERGE statement 7-38f, 4-2, 4-4
MERIT clause 5-14, 4-15, 5-4, 5-6
minus sign or hyphen (-)
  blank or insertion character 5-37
  continuation line 2-5
  within a word 2-1f
mnemonic name 2-3, 4-3
MODIFY statement (DG/DBMS) 10-26, 10-13, (table) 10-11
MOVE
  statement 7-40ff
  command 9-11, 9-2
multilevel indexed file 6-17, (table) 6-15, 1-2, 4-6, 5-8, 5-14
MULTIPLE FILE clause 4-21
multiplication 7-43
MULTIPLY statement 7-43

N

/N global binder switch 11-11
name
  about 2-3
  array 6-4
  condition 6-3
  data 6-3
  procedure 6-2
  qualification 6-2ff
name/V=number local binder switch 11-11
NATIVE collating sequence 4-2, 5-13
NEW LINE (NL) 2-1f, 2-4
NEXT, relative option phrase 6-16ff
node size, file description entry 5-7
/NODEFILE 11-15, 11-13
nondeclaratives procedure 6-1
nonexecutable program file 11-10
nonfatal program errors 11-16f
nonshared code 11-11
null 2-4
numeric data
  byte-aligned binary 5-27
  byte-aligned floating point 5-27
  decimal with leading separate sign 5-27
  decimal with leading sign overpunch 5-26
  decimal with trailing separate sign 5-27
  decimal with trailing sign overpunch 5-26
  defining items (PICTURE) 5-34ff
  edited 5-26
  error message 11-17
  external floating point 5-27
  group data 5-27
  literal 2-3
  packed decimal 5-27
  sign overpunch characters (table) 5-26
  unsigned decimal 5-26

O

/O
  global binder switch 11-11
  local binder switch 11-11
Object-Computer paragraph 4-2
  PROGRAM COLLATING SEQUENCE clause 4-2
  SEGMENT-LIMIT clause 4-3
object file, binary 3-1, 11-1ff
occurrence number 1-2
OCCURS clause 5-31f
octal code 2-4
ON/OFF STATUS 4-3f
OPEN statement 7-44f, 4-7
operating instructions
  about 11-1, 1-1
  binding
    about 11-10
    command (CBIND) 11-10
    switches (see switches, binder)
    using ANSI standard segmentation 11-12ff
    using virtual code 11-12ff
  compiling 11-1ff
    about 11-1
    calling the compiler 11-2
    error messages 11-9
    overlays 11-12ff, 11-2
    switches (see switches, compiler)
    using ANSI standard segmentation 11-2f
    using virtual code (/V) 11-2f
    warning messages 11-9f
  executing 11-15
    with the debugger (/D) 11-16
  loading 11-15
  runtime errors (see errors, runtime)
operators
  arithmetic (table) 6-8
  logical 6-10f
OPTIONAL clause 4-14
ORGANIZATION clause 4-16f
overflow error message 11-17
overlay 11-12ff, 4-3

P

/P global compiler switch 11-4
PAD clause 5-14, 5-3, 5-6
paging 11-2

paragraph
definition 6-1
naming 2-6, (table) 2-3
parameter, subprogramming 6-12
PARITY clause 4-18
PARTIAL RECORD clause 5-14, 6-18
PERFORM statement 7-46ff
example 7-48
in WALKBACK command 9-14
period 2-1, 2-3
phrase 2-6
AT END 6-19
CORRESPONDING 6-7
INVALID KEY 6-19
KEY series 6-17f
position 6-15
relative option 6-16f
ROUNDED 6-6
SIZE ERROR 6-6f
PICTURE clause 5-33f, 5-18, 5-22ff
alphabetic items in 5-33
alphanumeric edited items in 5-33
alphanumeric items in 5-33
data editing in 5-36ff
numeric edited items in 5-35
numeric items 5-34
plus sign (+) 5-35, (table) 5-37, 2-2, (table) 2-1
POSITION phrase 6-15
positioning within a database 10-14
print file 2-2
print file formatting 6-13f
ADVANCING phrase 6-13f
AT END-OF-PAGE phrase 6-13f
LINAGE clause 6-13f
PRINTER clause 4-9, 4-11, 4-14
Procedure Division 6-1ff, 7-1ff
about 6-1ff
arithmetics 6-6ff
declaratives 6-2
exception conditions, I/O 6-19ff
AT END phrase 6-19
declaratives section 6-19f
file status data items 6-20f
INFOS status data items 6-21
INVALID KEY phrase 6-19
expressions
arithmetic 6-8
conditional, compound 6-10ff
conditional, simple 6-9f
operators 6-10f
indexed file (see indexed file)
name qualification 6-2ff
array 6-4
condition 6-3
data 6-3
procedure 6-2
paragraph 6-1
phrases
about 6-1
AT END 6-19
CORRESPONDING 6-7
INVALID KEY 6-19
KEY series 6-17f
position 6-15
relative option 6-16f
ROUNDED 6-6
SIZE ERROR 6-6f
print file formatting 6-13f
relative file (see relative file)
sequential file (see sequential file)
section 6-1
segmentation 6-13, 6-1
sentence 6-1
statements 7-1ff, 6-1 (see statements, Procedure Division)
subprogramming 6-12
program
control, transfer of
ALTER statement 7-9
CALL statement 6-12, 7-10f, 9-14
CALL PROGRAM statement 7-12f
CANCEL statement 7-14
EXIT statement 7-24
EXIT PROGRAM statement 7-25
GO statement 7-28
IF statement 7-29f
PERFORM statement 7-46ff
in WALKBACK command 9-14
STOP statement 7-72
execution switch 4-3
name (table) 2-3
sample
COBOL 1-3
DG/DBMS 10-35ff, 10-40ff, 10-39
segmentation 4-2
structure 1-1
PROGRAM COLLATING SEQUENCE clause 4-2
pseudo-text 8-2

Q

/Q
global binder switch 11-11, 11-14
global compiler switch 11-4
qualification, name
array 6-4
condition 6-3
data 6-3
procedure 6-2
quotation mark (") 2-4, (table) 2-1
QUOTE(S) figurative constant 2-2

R

/R
  local binder switch 11-11
  local compiler switch 11-4
random access mode 4-8
RDOS upgradability to AOS F-1f
READ statement
  indexed file 7-51f
  relative file 7-50
  sequential file 7-49
READY statement (DG/DBMS) 10-16, 10-13, (table) 10-11
RECONNECT statement (DG/DBMS) 10-16, 10-13, (table) 10-11
record
  area 5-2
  clauses
    RECORD CONTAINS 5-7
    RECORD LENGTH 5-8
  description 5-2, 4-7
  format
    data-sensitive 4-6, 1-2, 5-7
    fixed-length 4-6, 1-2, 5-7
    undefined-length 4-6, 1-2, 5-7
    variable-length 4-6, 1-2, 5-7
  Input/Output statements
    DELETE 7-19f
    READ 7-49ff
    RELEASE 7-53, 4-7
    RETRIEVE 7-54f, 4-17
    RETURN 7-56, 4-7
    REWRITE 7-57f
    SEEK 7-63
    START 7-69ff
    TRUNCATE 7-77
    UNDELETE 7-78f, 4-7
    UNLOCK 7-80
    WRITE 7-85ff
  pointer positioning 7-69ff
RECORD phrase 6-19
RECORD CONTAINS clause 5-7
RECORD LENGTH clause 5-8
RECORDING MODE clause 5-8
REDEFINES clause 5-30
relation condition 6-9
relative file
  about 4-6
  access mode with (table) 4-8
  file description entry (format) 5-4, (table) 5-5f
  I/O 4-10ff
  organization 5-4
  relative record number 4-6
  SELECT clause (format) 4-10, (table) 4-11ff
  statements
    CLOSE 7-15
    DELETE FILE 7-21
    EXPUNGE 7-26
    OPEN 7-44f
    READ 7-50
    REWRITE 7-58
    SEEK 7-63
    START 7-70
    WRITE 7-87
    UNLOCK 7-80
relative option phrase
  BACKWARD 6-16ff
  DOWN 6-16ff
  DOWN FORWARD 6-16ff
  FORWARD 6-16ff
  NEXT 6-16ff
  STATIC 6-16ff
  UP 6-16ff
  UP BACKWARD 6-16ff
  UP FORWARD 6-16ff
RELEASE statement 7-53, 4-7
RENAMES entry 5-44f
REORG utility B-4
replacement string 8-2
RESERVE clause 4-15
reserved word
  in COPY replacement string 8-2
  list A-1f
RETRIEVE statement 7-54f, 4-17
RETURN statement 7-56, 4-7
REWRITE statement
  indexed file 7-59f
  relative file 7-58
  sequential file 7-57
ROLLBACK statement (DG/DBMS) 10-20, 10-14, (table) 10-11
root context 11-11
ROOT MERIT clause 4-15
ROUNDED phrase 6-6
rubout 2-4
runtime
  errors 11-16ff (see errors, runtime)
  library 11-1

S

/S
  global binder switch 11-11
  global compiler switch 11-4
  local binder switch 11-11
SAME RECORD AREA clause 4-21
save file 11-18
scale factoring 5-34
Screen Section
  about 5-1
  clauses
    AUTO 5-23
    BELL 5-23
    BLANK LINE 5-22
    BLANK SCREEN 5-21
    BLANK WHEN ZERO 5-24

BLINK 5-23
COLUMN 5-21
FROM 5-22
FULL 5-24
JUSTIFIED 5-24
LINE 5-21
PICTURE IS 5-22
REQUIRED 5-23
SECURE 5-23
TO 5-23
USING 5-23
VALUE 5-23
VIRTUAL 5-21
commands 5-18ff
ACCEPT 5-18ff
CPRINT 5-25
DISPLAY 5-18f
format 5-20ff
elementary item 5-20
example 5-24f
group items 5-20
literals 5-20
function delimiter keys 5-19
SD entry 5-1ff, (format) 5-4, (table) 5-5f
SEARCH statement 7-61f
section
File 5-3ff (see File Section)
Input-Output 4-9ff (see Input-Output Section)
Linkage 5-18, 5-1 (see Linkage Section)
name 2-6, (table) 2-3
Procedure Division 6-1
Screen 5-18ff (see Screen Section)
Subschema 5-1, 10-2ff (see Subschema Section)
Virtual-Storage 5-15ff (see Virtual-Storage Section)
Working-Storage 5-15, 5-1 (see Working-Storage Section)
SEEK statement 7-63
SEGMENT LIMIT clause 6-13, 4-3
segmentation, program 4-3, 6-13, 11-2f, 11-12ff
SELECT clause 4-9ff, 5-2, 5-16, 4-3
ACCESS MODE clause 4-16f
ALLOW SUB-INDEX clause 4-18
ASSIGN clause 4-14f
DUPLICATES clause 4-16f
examples 4-20
FILE STATUS clause 4-18
format, indexed 4-10
format, relative 4-10
format, sequential 4-9
INFOS clause 4-18
KEY COMPRESSION clause 4-19
LEVELS clause 4-18
OPTIONAL clause 4-14
ORGANIZATION clause 4-16
PARITY clause 4-18
RESERVE clause 4-15
table 4-11ff
with unlabeled magnetic tape E-1
semicolon (;) 2-1
sentence 2-5, 6-1
separators 2-1
sequential access mode 4-8
sequential file
about 4-5f
file description entry (format) 5-3, (table) 5-5f
format 4-6, 4-9, 4-11ff
relative record number 4-6
SELECT clause (format) 4-9, (table) 4-11ff
statements
CLOSE 7-15
DELETE FILE 7-21
EXPUNGE 7-26
OPEN 7-44f
READ 7-49
REWRITE 7-57
START 7-69
TRUNCATE 7-77
WRITE 7-85f
UNLOCK 7-80
set
occurrences 10-8f
types 10-7f
identifying name of 10-9
insertion/retention criteria 10-9
MEMBER 10-9
ORDER IS 10-10
OWNER 10-9
reconnections 10-10
SET
statement 7-64
command 9-12, 9-2
SET UP/DOWN statement 7-65
shared code 11-11
SIGN clause 5-41
sign overpunch characters (table) 5-26
signed datum 5-34
size error 11-16
SIZE ERROR phrase 6-6f
slash (/)
comment line 2-5, (table) 2-1, 2-2
insertion character 5-36f
SORT
command B-4
statement 7-66ff, 4-2, 4-4
sort/merge process
about 4-7
file description entry (format) 5-4, (table) 5-5f
SELECT clause (format) 4-11, (table) 4-11
statement summary 7-3
statements
MERGE 7-38f, 4-2, 4-4
RELEASE 7-53, 4-7
RETURN 7-56, 4-7
SORT 7-66ff, 4-2, 4-4
source file 11-1
Source-Computer paragraph 4-1

space
insertion character 5-36, (table) 5-37
separator 2-1
SPACE(S) figurative constant 2-2
SPACE MANAGEMENT clause 4-15
Special-Names paragraph 4-3f
alphabet clause 4-4
CHANNEL clause 4-4
CURRENCY SIGN clause 4-5
DECIMAL-POINT IS COMMA clause 4-5
SWITCH clause 4-4
STANDARD-1 collating sequence 4-2, 5-13
START statement
indexed file 7-71
relative file 7-70
sequential file 7-69
statement 2-5, 6-1
statements, Procedure Division 7-1ff, 10-11ff
arithmetic operations
ADD
COMPUTE 7-16
DIVIDE 7-23
MULTIPLY 7-43
SET UP/DOWN 7-65
SUBTRACT 7-75f
console Input/Output
ACCEPT 7-4f
DISPLAY 7-22, 4-3
data manipulation and editing
INSPECT 7-31ff
MOVE 7-40ff
SEARCH 7-61f
SET 7-64
STRING 7-73f
UNSTRING 7-81f
DG/DBMS Data Manipulation Language (DML)
about 10-13ff
CHECK 10-30
COMMIT 10-19
CONNECT 10-21
DISCONNECT 10-21
ERASE 10-27
FIND (current) 10-34
FIND (duplicates) 10-33
FIND (owner) 10-35
FIND (positional) 10-31
FIND (using data items) 10-32
FINISH 10-17
GET 10-25
IF 10-28f
INITIATE 10-18
MODIFY 10-26
READY 10-16
RECONNECT 10-23
ROLLBACK 10-20
STORE 10-24
table 10-11f
file handling
CLOSE 7-15 DEFINE SUB-INDEX 7-17ff
DELETE 7-19f
DELETE FILE 7-21
EXPUNGE 7-26
EXPUNGE SUB-INDEX 7-27
LINK SUB-INDEX 7-36f
OPEN 7-44f, 4-7
READ, indexed file 7-51f
READ, relative file 7-50
READ, sequential file 7-49
RETRIEVE 7-54f, 4-17
REWRITE, indexed file 7-59f
REWRITE, relative file 7-58
REWRITE, sequential file 7-57
SEEK 7-63
START, indexed file 7-71
START, relative file 7-70
START, sequential file 7-69
TRUNCATE 7-77
UNDELETE 7-78f, 4-7
UNLOCK 7-80
USE 7-83f, 6-19, 4-6
WRITE, indexed file 7-88f
WRITE, relative file 7-87
WRITE, sequential file 7-85f
miscellaneous
ACCEPT DAY/DATE/TIME 7-6f
sort/merge
MERGE 7-38f, 4-2, 4-6
RELEASE 7-53, 4-7
RETURN 7-56, 4-7
SORT 7-66ff, 4-2, 4-4
summary 7-1ff
STATIC, relative option phrase 6-16ff
STOP
command 9-13, 9-2
statement 7-72
STOP RUN statement B-3
STORE statement (DG/DBMS) 10-24, 10-13, (table) 10-11
string
comparison 4-2
concatenation (STRING statement) 7-73f
counting (INSPECT statement) 7-31ff
replacement (INSPECT statement) 7-31ff
separation (UNSTRING statement) 7-81ff
STRING statement 7-73f
subindex
create (DEFINE SUB-INDEX statement) 7-17f, 4-7
delete (EXPUNGE SUB-INDEX statement) 7-27, 4-7
link (LINK SUB-INDEX statement) 7-36f, 4-7
subprograms 6-12
arguments 6-12
CALL statement 6-12, 7-10
CANCEL statement 7-14
DG/DBMS 10-15
EXIT statement 7-24
parameters 6-12
PERFORM statement 7-46ff
USING phrase 6-12

subschema (DG/DBMS) 10-2ff
about 5-1
COPY 10-5
data definition language (DDL) 10-2
data manipulation language (DML) 10-5
example 10-2ff
free cursors 10-10
occurrences 10-5
record type 10-5, (figure) 10-6, 10-9
schema 10-2
set types 10-7ff, 10-8
SYSTEM record type 10-7
user work area 10-5, (figure) 10-7
subscript 6-4
SUBTRACT statement 7-75f
subtraction
operator 6-8
SET UP/DOWN statement 7-65
SUBTRACT statement 7-75f
suppression symbol, zero (table) 5-37
suspend program execution (STOP statement) 7-72
switch
condition 6-10
name (table) 2-3
SWITCH clause 4-4
switches
binder, global
/B 11-10
/D 11-10
/E 11-10
/H 11-10
/I 11-10
/K=n 11-10
/L 11-11, 11-4
/L=name 11-11
/N 11-11
/O 11-11
/Q 11-11, 11-14f
/S 11-11
/T=n 11-11
/Z=n 11-11
binder, local
name/B 11-11
/C 11-11, 11-3, 11-13
/D 11-11, 11-16
/H 11-11
/O 11-1!
/R 11-11
/S 11-11
/U 11-11
name/V=number 11-11
n/Z 11-11
/NODEFILE 11-15, 11-13
compiler, global
/A 11-4, 11-18
/C 11-4, 11-3
/D 11-4
/E 11-4, 2-4
/G 11-4
/L 11-4, 11-18
/M 11-4 (see Map switch)
/P 11-4
/Q 11-4
/S 11-4
/V 11-4, 11-3, 11-12, 11-2
/W 11-4, 11-9
/X 11-4
compiler, local
/L 11-4
/R 11-4
/M 11-5ff (see Map switch)
SYNCHRONIZED clause 5-41
symbols
editing (table) 5-37
zero suppression (table) 5-38
system errors, fatal 11-17
SYSTEM record type 10-7

T

/T=n global binder switch 11-11
tab 2-5
table searching (SEARCH statement) 7-61f
temporary storage 5-1
text format 2-1, 2-5
textual substitutions 8-1 (see also COPY)
trace information 11-18
TRUNCATE statement 7-77

U

/U local binder switch 11-11
undefined-length record 4-6, 1-2, 5-7
UNDELETE statement 7-78f, 4-7
unlabeled magnetic tape E-1ff
creating E-2
reading E-3
SELECT clause E-1
UNLOCK statement 7-80
UNSTRING statement 7-81f, 4-2
UP, relative option phrase 6-16ff
UP BACKWARD, relative option phrase 6-16ff
UP FORWARD, relative option phrase 6-16ff
USAGE clause 5-40
USE statement 7-83f, 6-19, 4-6
user-defined word 2-3
in COPY replacement string 8-2
USING phrase 6-12

V

/V global compiler switch 11-4, 11-3, 11-12, 11-2
VALUE clause 5-42
VALUE OF clause 5-10f
variable-length record 1-2, 4-6, 5-7
virtual code 11-2f
ANSI standard segmentation 11-2f
compiling (/V) 11-2f
overlays 11-2, 11-12ff

Virtual-Storage Section 5-15ff
about 5-1
data item 5-16f
files 5-16
memory space 5-16f
optimizing virtual data 5-17
VOLUME SIZE clause 4-15

W

/W global compiler switch 11-4, 11-9
WALKBACK command 9-14, 9-2
warning messages, compiler 11-9f
words (COBOL)
in COPY replacement string 8-2
figurative constants 2-2
ALL 2-2
CR 2-2
HIGH-VALUE(S) 2-2
length 2-2
LOW-VALUE(S) 2-2
QUOTE(S) 2-2
SPACE(S) 2-2
used interchangeably with literals 2-2
ZERO(E)(S) 2-2
optional 2-2
special character 2-2
special register 2-2
Working-Storage Section 5-15, 5-1
WRITE statement
indexed file 7-88f
relative file 7-87
sequential file 7-85f

X

/X global compiler switch 11-4

Z

/Z local binder switch 11-11
/Z=n global binder switch 11-11
zero
insertion character 5-36, (table) 5-37
suppression 5-38, (table) 5-37
ZERO(E)(S) figurative constant 2-2

Data General Corporation, Westboro, MA 01580