COBOL

COBOL-81 RSX-11M / M-PLUS User's Guide

Order No. AA-M179B-TC

digital

software

COBOL-81 RSX-11M/M-PLUS User's Guide

Order No. AA-M179B-TC

July 1983

This manual describes how to use COBOL-81 on the RSX-11M/M-PLUS operating systems.

OPERATING SYSTEM AND VERSION:	RSX-11M	V4
	RSX-11M-PLUS	V2
SOFTWARE VERSION:	COBOL-81	V2

digital equipment corporation, maynard, massachusetts

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

The software described in this document is furnished under a license and may be used or copied only in accordance with the terms of such license.

No responsibility is assumed for the use or reliability of software on equipment that is not supplied by DIGITAL or its affiliated companies.

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

The following are trademarks of Digital Equipment Corporation:

digital ™
DEC
DECmate
DECsystem-10
DECSYSTEM-20
DECUS
DECwriter
DIBOL
MASSBUS
PDP
P/OS
Professional
Rainbow
RSTS
RSX
UNIBUS
VAX
VMS
VT
Work Processor

Commercial Engineering Publications typeset this manual using DIGITAL's TMS–11 Text Management System.

Part I Developing COBOL-81 Programs

Contents
Using COBOL-81 on Your Operating System
Creating and Entering a COBOL-81 Program
Compiling a COBOL-81 Program
Linking a COBOL-81 Program
Running a COBOL-81 Program

Appendix A, Compiler Implementation Limitations
Appendix B, Compiler Error Messages
Appendix C, Run-Time Error Messages
Appendix D, Using COBOL-81 MCR/CCL Commands

Part II Using COBOL-81 Programming Options

Contents
Using the COBOL-81 Reformat Utility
Troubleshooting
Debugging Your Program
Reducing Your Task Size
Improving Program Performance
Interprogram Communication

Appendix A, Debugger Error Messages

Part III Handling Data with COBOL-81

Contents
Numeric Character Handling
Nonnumeric Character Handling
Table Handling
Data Handling Optimization

Part IV Processing Files and Records with COBOL-81

The Basics of Handling COBOL-81 Files and Records
Processing Sequential Files
Processing Relative Files
Processing Indexed Files
Input/Output Exception Conditions Handling
Sharing Files and Protecting Records
File Optimization Techniques
Producing Printed Reports With COBOL-81
Forms for Video Terminals
Sorting Records and Merging Files

Appendix A, Designing Your Form with Escape Sequences
Appendix B, Logical Unit Number (LUN) Assignments

Master Index

Book Map

To the Reader

Objectives

This manual explains how to use COBOL-81 on the RSX-11M/M-PLUS operating system.

The information in this manual supplements the description of the COBOL programming language in the *COBOL-81 Language Reference Manual.*

Intended Audience

This documentation set is for the experienced COBOL programmer. It does not attempt to teach the COBOL language or operating system concepts and procedures. If you are a new COBOL user, you should read introductory COBOL textbooks and take DIGITAL COBOL courses – either self-paced or classroom.

Prerequisites

Those unfamiliar with the RSX-11 operating system should refer to either the *RSX-11M/RSX-11S Information Directory and Index,* or the *RSX-11M-PLUS Information Directory and Index*. The directory appropriate for your operating system lists and describes all manuals in the system documentation set.

Structure of This Document

The *COBOL-81 User's Guide* is divided into four parts:

Part I	Developing COBOL-81 Programs
Part II	Using COBOL-81 Programming Options
Part III	Handling Data with COBOL-81
Part IV	Processing Files and Records with COBOL-81

The Book Map, which follows the title page, lists the contents of all four parts. A detailed table of contents precedes each of these four parts of the manual.

The User's Guide contains a master index of all topics discussed in the *COBOL-81 Language Reference Manual* and the *COBOL-81 User's Guide*.

Associated Documents

- The *COBOL-81 Language Reference Manual*, Order No. AA-J434B-TC, describes the COBOL programming language rules and formats.
- The *COBOL-81 Pocket Guide*, Order No. AV-H630C-TC, provides quick reference information needed to create, compile, link, and run COBOL programs.
- The *COBOL-81 Installation Guide/Release Notes*, Order No. AA-M181C-TC, describes the installation and certification procedures for the COBOL-81 compiler on the RSX-11M/M-PLUS operating system.
- The *PDP-11 COBOL to COBOL-81 Translator Utility*, Order No. AA-N339A-TC, tells users how to convert PDP-11 COBOL application programs to COBOL-81 programs.

Conventions Used in This Document

Throughout this manual, commands are displayed in the Digital Command Language (DCL) format. See Part I, Appendix D, for the Monitor Console Routine (MCR) equivalents. Additional conventions follow:

Convention	Meaning
RET	A symbol with a one- to three-character abbreviation indicates that you must press a key on the terminal; for example, RET and TAB indicate that you press the RETURN key and the TAB key on your terminal.
CTRL/x	The symbol CTRL/x indicates that you must press a key labeled CTRL while you simultaneously press another key; for example, CTRL/C, CTRL/O.
`COBOL`RET `File:          PAYROLL`RET	Black ink indicates all output lines or prompting characters that the system prints or displays. Red ink indicates all user-entered commands.

`PROCEDURE DIVISION.`
`BEGIN-PROGRAM.`
.
.
.
`END-PROGRAM.`
	A vertical series of periods, or ellipses, means that not all the data a user would enter is shown.

Summary of Technical Changes

This section lists, by part and chapter, the major technical changes documented in Version 2 of the *COBOL-81 User's Guide*. These modifications reflect new features in existing software as well as changes and additions to COBOL program development.

Part I Developing COBOL-81 Programs

- Chapter 1, Using COBOL-81 on RSX-11M/M-PLUS, provides a description of the DCL HELP facility.
- Chapter 3, Compiling a COBOL-81 Program, describes the flagging of a destructive reference with the CROSS_REFERENCE compiler qualifier.
- Appendix D, MCR Commands for COBOL-81, includes the following new switches that are used with the BLDODL utility:

  /CLU

  /ULIB

  /FMS

Part II Using COBOL-81 Programming Options

- Chapter 3, Debugging Your Program, includes qualification support for the following Debugger commands:

  SET BREAKPOINT

  CANCEL BREAKPOINT

  DISPLAY

  MOVE

  DEFINE

  UNDEFINE

Part III Handling Data with COBOL-81

- Chapter 3, Table Handling, describes changes to the following features of the COBOL-81 SEARCH statement:

  OCCURS DEPENDING ON clause

  KEY IS phrase

  DEPENDING ON phrase

Part IV Processing Files and Records with COBOL-81

- Chapter 3, Processing Relative Files, describes the use of fixed-size record cells and the use of a key to retrieve records.
- Chapter 5, Input/Output Exception Conditions Handling, provides information on the following special registers containing status values from the RMS-11 file system:

  RMS-STS

  RMS-STV

- Chapter 7, File Optimization Techniques, describes the APPLY WINDOW clause, which corresponds to window pointers.
- Chapter 8, Producing Printed Reports with COBOL-81, discusses use of the LINAGE clause to define the logical page.
- Chapter 9, Forms For Video Terminals, describes ACCEPT/DISPLAY extensions for screen formatting. These options allow you to do the following:

  Erase parts or all of the screen

  Use relative and absolute cursor positioning

  Specify form attributes on data to be displayed and accepted

  Convert data to appropriate usage when accepting data

  Handle errors in filling out the form

  Provide field and screen protection by limiting the number of characters typed on the terminal

  Accept data with no echo

  Specify default values for ACCEPT statements

  Define and handle special control keys

  Translate any data item to usage DISPLAY for terminal use

- Chapter 10, Sorting Records and Merging Files, discusses a wide range of sorting capabilities and options provided with the SORT and MERGE verbs.

Incompatibilities with VAX–11 COBOL

COBOL-81 is a subset of VAX–11 COBOL, but the two products have some incompatibilities due to differences between the PDP-11 and the VAX–11 computer systems. The /STA:VAX compiler switch tells the compiler to flag COBOL-81 code that is incompatible with VAX–11 COBOL. Appendix D in the *COBOL-81 Language Reference Manual*, Ensuring COBOL-81 Compatibility with VAX–11 COBOL, lists and describes all known incompatibilities.

Acknowledgment

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

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

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

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

Contents

Page

PART I

Chapter 1 Using COBOL-81 on RSX-11M/M-PLUS

1.1 Operating System Commands 1-2

1.1.1 Aids to Entering Commands 1-2
1.1.2 Getting HELP 1-2

1.2 File Specifications and Defaults 1-3

1.2.1 Accounts, Directories, and Files 1-3
1.2.2 Logical Names 1-5

1.3 File Creation and Maintenance 1-6

Chapter 2 Creating and Entering a COBOL-81 Program

2.1 Creating the Source File 2-1
2.2 Choosing a Reference Format 2-2

2.2.1 Terminal Reference Format 2-2
2.2.2 ANSI Reference Format 2-2

2.3 Using the COPY Statement 2-3

Chapter 3 Compiling a COBOL-81 Program

3.1 Functions of the Compiler 3-1
3.2 Command Line Format 3-2
3.3 Command Qualifiers 3-2
3.4 Examples 3-6
3.5 Common COBOL-81 Command Line Errors 3-7
3.6 Compiler Diagnostics Summary 3-7
3.7 Compiler Limitations Summary 3-8

Chapter 4 Linking a COBOL-81 Program

4.1 Functions of the LINK Command 4-1
4.2 Using the LINK/C81 Command to Build an Executable Image 4-1

4.2.1 Library Routine Qualifiers 4-2
4.2.2 Output File Qualifiers 4-3

4.3 Linking Error Message Summary 4-3

Chapter 5 Running a COBOL-81 Program

5.1 Functions of the COBOL-81 Object Time System (OTS) 5-1
5.2 Command Line Format 5-1
5.3 Run-Time Error Message Summary 5-2

Appendix A COBOL-81 Compiler Implementation Limitations

Appendix B Compiler Error Messages

Appendix C Run-Time Error Messages

Appendix D MCR Commands for COBOL-81

D.1 Compiling D-1

D.1.1 Using the Compiler. D-2

D.1.1.1 Switches Requesting Special Functions D-4
D.1.1.2 Switches Suppressing Functions. D-5
D.1.1.3 Switches Altering Functions D-6
D.1.1.4 Examples of Switches D-7

D.2 Using the BLDODL Utility. D-7

D.2.1 BLDODL Command Line and Switches D-8
D.2.2 BLDODL Utility Command Line Defaults D-10

D.3 Task-Building D-10

D.3.1 Using the CMD File as Input. D-10
D.3.2 Using a Direct Command Line as Input D-11
D.3.3 Results of the Task Build D-11

D.4 Executing D-12
D.5 Examples D-12

Figure

1-1 COBOL-81 Program Development 1-4

Tables

1-1 Default File Types 1-5
1-2 Commands for File Operations 1-6
3-1 Qualifiers 3-2
D-1 Summary of Compiler Switches D-3

Chapter 1
Using COBOL-81 on RSX-11M/M-PLUS

The RSX-11M and RSX-11M-PLUS operating systems and their command language, DCL (DIGITAL Command Language), provide numerous tools and utilities for program development. If your terminal is not set to DCL mode, enter the following command:

```
SET /DCL=TI:
```

where:

TI: is the physical device name of your terminal.

This chapter summarizes the fundamental information you need to develop your COBOL-81 programs, including:

- The DCL commands you use to create, compile, link, and execute COBOL-81 programs
- The rules for specifying input and output files for commands and programs
- The commands you use to create, modify, and maintain files

For an introduction to these concepts, see the *RSX-11M/M-PLUS Guide to Program Development.* For detailed definitions of DCL commands and file specifications, see the *RSX-11M/M-PLUS Command Language Manual.*

1.1 Operating System Commands

To develop COBOL-81 programs, you use four DCL commands:

- The EDIT command allows you to create the source file.
- The COBOL command invokes the COBOL-81 compiler.
- The LINK/C81 command produces an executable image of your program.
- The RUN command executes the program.

1.1.1 Aids to Entering Commands

The next few chapters of this manual describe in detail the commands you use to develop COBOL-81 programs and the qualifiers that modify the operation of these commands. The following hints can help you enter commands easily and accurately:

- You can abbreviate any command name or qualifier name to four characters. In most cases, fewer than four characters are accepted, as long as there is no ambiguity about the name of the command.
- You must precede each qualifier in the command line with a single slash character (/) or a space.
- If you omit a required parameter (for example, a file specification), the DCL command interpreter prompts you to enter it.
- You can enter a command on as many lines as you want, as long as you end each continued line with a hyphen (-) and the maximum line length does not exceed 80 characters for RSX-11M or 255 characters for RSX-11M-PLUS.
- After you have entered a complete command, you must press the RETURN key to pass the command to the system for processing.
- You can cancel a command before the final RETURN by typing CTRL/U.

If you make an error entering a command (for example, if you misspell a command or qualifier name), the command line interpreter issues an error message, and you must reenter the entire command line including any qualifiers.

1.1.2 Getting HELP

The HELP command invokes the RSX-11M/M-PLUS HELP utility, which gives you online information about a command, its parameters, and qualifiers. When you type HELP, the utility displays information available in the system help files or in any help library that you specify.

To obtain information about the COBOL compile command, you enter the following command:

```
HELP COBOL
```

The HELP command response displays a description of the COBOL command and a list of its qualifiers.

1.2 File Specifications and Defaults

Because many DCL commands and qualifiers affect files, it is helpful to understand the relationship between accounts and directories when you work with files.

1.2.1 Accounts, Directories, and Files

You log into an account when you begin an RSX-11M/M-PLUS session. Your account name identifies you to the system.

The system uses your account name to keep track of the resources you use, such as the amount of time you access the computer's memory or are logged in and the amount of storage space your files require.

A directory is a list of the files in your account that are kept in a specific location on a disk. Each directory stores such information as the name and size of each file stored on a particular mass-storage device under a particular User File Directory (UFD).

You identify a file by specifying its location and its name. A file's complete location consists of:

- The device. Files are kept on mass-storage devices such as disks or magnetic tapes.
- UFD. Files are contained in User File Directories (UFDs). The UFD is a two-number code in the form [g,m] that is in every file specification, either explicitly or by default, and that locates the file. In the form [g,m], g signifies group, while m signifies member.

A file's name, chosen by the person who creates the file, consists of:

- The file name (one to nine alphanumeric characters)
- The file type (zero to three alphanumeric characters), preceded by a period
- The version number, in octal, of the file, preceded by a semicolon

A file specification is the full name and location of a file. In its complete form, the file specification includes:

```
device:[g,m]filename.typ;version
```

The delimiters in a file specification are brackets, commas, colons, and semicolons. Brackets ([ ]) surround the UFD. A comma (,) separates the two numbers of the UFD. A single colon follows the device name. A semicolon separates the file type from the version number.

A complete file specification that does not assume any defaults is:

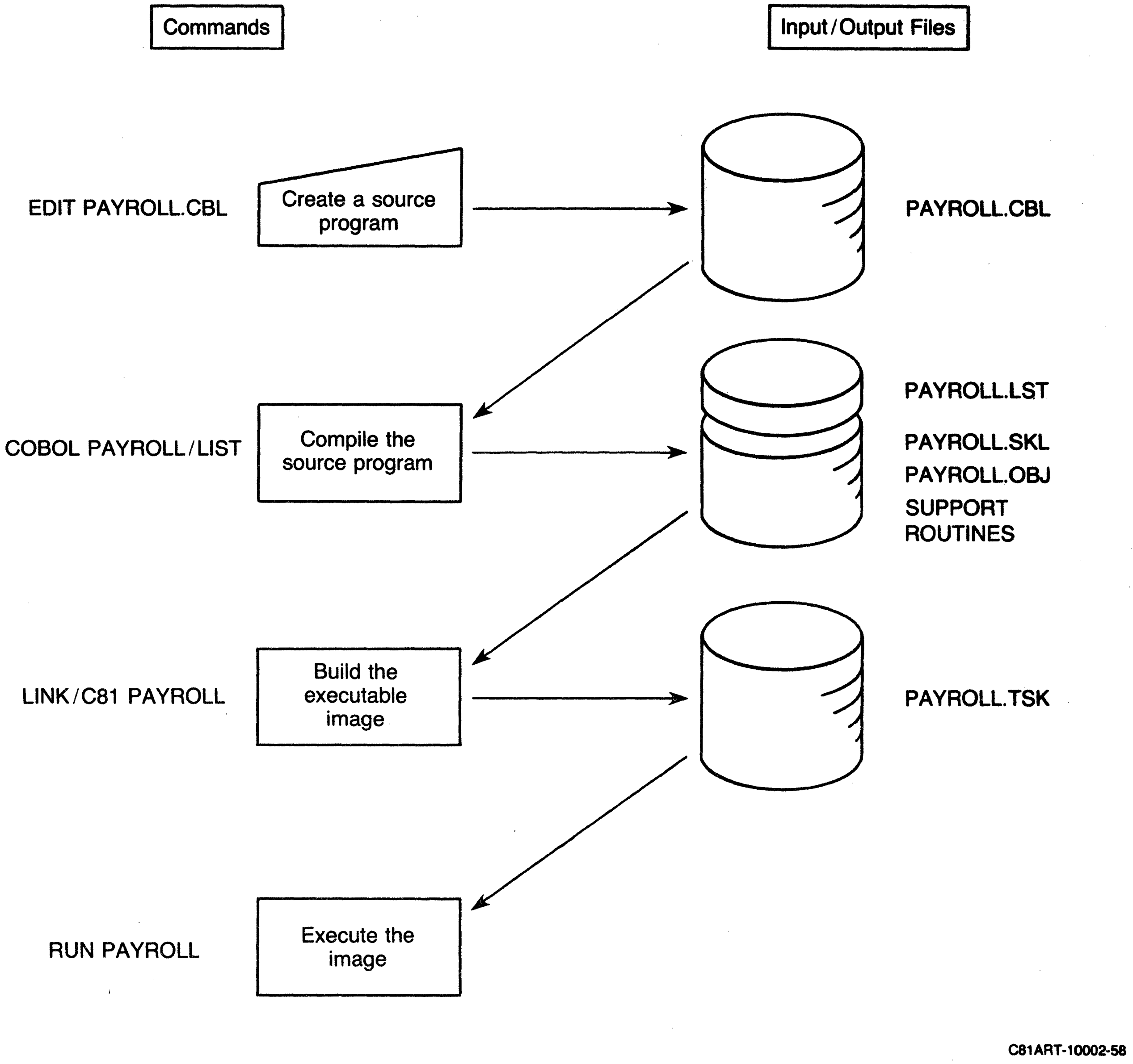
```
DB0:[30,10]PAYROLL.CBL;3
```

You need not give a complete file specification each time you refer to a file. A simple file specification, one that uses system defaults, consists of the file name and type:

```
PAYROLL.CBL
```

In Figure 1-1, all input and output files are given in their simplest forms. To define a unique COBOL-81 source file, you need only give it a unique name and a file type of CBL.

Figure 1-1: COBOL-81 Program Development



The following DCL commands appear in Figure 1-1:

```
EDIT PAYROLL.CBL
COBOL PAYROLL/LIST
LINK/C81 PAYROLL
RUN PAYROLL
```

For these commands, the following defaults are in effect:

- All the commands shown use the current default device and the user's UFD to locate a file that you specify.
- The EDIT command does not assume a default file type. By explicitly specifying CBL as a file type when you create the source program, you can omit the file type in your compile command line.
- The COBOL command assumes that if no file type has been specified for a source file, its file type is CBL. Unless you use qualifiers to change the names of its output files, the compiler uses the default file types LST and OBJ for the listing and object files, respectively. The compiler also produces a skeleton overlay descriptor file with a default file type of SKL.
- The LINK/C81 command assumes that if no file type has been specified for an input file, its file type is SKL. To use the LINK/C81 command, you must be sure that the SKL file is available in addition to the OBJ file. If no qualifiers override the default output file types, the LINK/C81 command assigns the default file type TSK for the executable file.
- The RUN command assumes that if no file type has been specified for a file, its file type is TSK.

Table 1-1 summarizes the default file types.

Table 1-1: Default File Types

Type of File	Default Value
Input to compiler	CBL
Output from compiler	OBJ, SKL
Compiler listing file	LST
Input to LINK/C81	SKL
Output from LINK/C81	TSK
Input to RUN command	TSK

1.2.2 Logical Names

Logical names allow you to keep programs and batch control files independent of the physical locations of files. They also provide a convenient, shorthand way to specify devices and directories that you refer to frequently.

With the ASSIGN command, you can assign a logical name to a physical device. The logical names you assign can be system-wide or local, and you can assign any number.

The logical names that you assign for devices do not depend on the physical device specifications. Unlike a physical name, a logical name is independent of the drive on which the medium is mounted. Device logical names make it easier to adapt a program for use on different drives: You can use logical names within file specifications in your COBOL-81 source code, and you do not need to reference specific devices until run time.

1.3 File Creation and Maintenance

You create a source file with the CREATE command or with a text editor. The default DIGITAL Standard Editor available on RSX-11M/M-PLUS is EDT. Consult the *EDT Editor Manual* for information about how to use this editor. You can also use EDI, but EDT is now the default editor. (EDI is described in the *RSX-11M/11M-PLUS Utilities Manual.*)

Table 1-2 describes some of the basic DCL file-handling commands available to programmers. For online assistance in entering a command or determining its parameters, qualifiers, or options, use the HELP command.

Table 1-2: Commands for File Operations

	Creating and Modifying Files
CREATE	Creates a file from records or data that you input following the command; for example, lines entered from a terminal or placed in a batch input file.
EDIT	Invokes the EDT text editor.
	Displaying Files and File Names
TYPE	Displays the contents of individual files at the terminal.
DIRECTORY	Displays information about the files in a UFD.
	Copying, Renaming, and Appending Files
COPY	Copies the contents of a file or files to another file or files.
RENAME	Assigns a new name to a file.
APPEND	Concatenates a file to the end of another file.
	Deleting Files
DELETE	Deletes files from a directory.

Chapter 2
Creating and Entering a COBOL-81 Program

To create a source program file you use a text editor. EDT, the DIGITAL Standard Editor, is the default editor for DCL on RSX-11M/M-PLUS. When you use the EDIT command, the /EDT qualifier is optional. The text editor enables you to type in your source code and store it in a system file. Once you have stored the program, you can use EDT to make any desired changes.

2.1 Creating the Source File

To invoke EDT on an RSX-11M/M-PLUS system, you type: EDIT filename.typ. For example:

```
EDIT PAYROLL.CBL
```

If the file you specify already exists, you can then modify it. If the file does not exist, EDT prints:

```
Input file does not exist

[EOB]

*
```

The asterisk is EDT's prompt for the next command. Following is a sample program created with EDT:

```
IDENTIFICATION DIVISION.
PROGRAM-ID.    TEST-1.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 MESSAGE-AREA      PIC X(24) VALUE "THIS IS A TEST PROGRAM".
PROCEDURE DIVISION.
000-BEGIN.
    DISPLAY MESSAGE-AREA.
    STOP RUN.
```

Consult the *EDT Editor Manual* for further instructions on using EDT.

2.2 Choosing a Reference Format

Before you can develop a COBOL-81 program, you must decide on a source reference format and prepare your source program accordingly. The COBOL-81 compiler accepts source programs written in either DIGITAL's terminal reference format or ANSI reference format. However, you cannot mix reference formats in the same compile command line, even when you are copying text from a COBOL-81 library.

2.2.1 Terminal Reference Format

Use DIGITAL's terminal reference format when you create source files from interactive terminals. The COBOL-81 compiler accepts terminal format as the default reference format unless the default was changed by your system manager during installation. Terminal format eliminates the line-number and identification fields of ANSI format and allows horizontal tab characters and short lines. This saves disk space and decreases compile time. Because the spacing requirements of terminal format are more flexible, it is usually easier to edit source programs written in this format.

There are four rules regarding this format.

1. The maximum number of characters you can put on a line is 200. However, the listing that the compiler produces displays only the first 120 characters.
2. The indicator area, if used, is the first character position of the line. Valid characters in this area are hyphen (-), slash (/), and asterisk (*).
3. If an indicator is present in character position 1, Area A occupies positions 2 through 5; if not, Area A occupies positions 1 through 4.
4. Area B immediately follows Area A.

You can use the TAB key or the SPACE bar to position source entries in a line. Terminal format recognizes a RETURN as the end of a line. When you use indicators such as continuation (-), comment (*), or skip-to-top-of-page (/) characters, you must enter these characters in position 1. Area A then occupies positions 2 through 5 and Area B occupies positions 6 through 200.

For more information, see the *COBOL-81 Language Reference Manual.*

2.2.2 ANSI Reference Format

ANSI format (defined in the *COBOL-81 Language Reference Manual*) is useful on a card-oriented system. If your program is in ANSI format, you must compile it using the /ANSI_FORMAT qualifier (see Chapter 3) or convert it to terminal format using the REFORMAT utility (see Part II, Chapter 1). You can choose this format if your COBOL program was written for a compiler that used ANSI reference format.

The REFORMAT utility allows you to convert from terminal format to ANSI format or from ANSI format to terminal format. You can also use REFORMAT to match the formats of source files and COBOL-81 library files when their formats are not the same. See Part II, Chapter 1, for a full description of the REFORMAT utility.

2.3 Using the COPY Statement

The COPY statement allows you to access COBOL-81 libraries at compile time. These libraries contain source text that can be merged with one or more COBOL-81 programs at installation.

The simplest form of the statement is: COPY text-name. For example:

```
COPY CUSTRC.
```

A complex application can consist of many separately compiled programs that share the same structure declaration or variable declarations. In such cases, it is convenient to maintain only one copy of the declaration of the variables and to include this declaration in each source program.

Although most statements contained in your program specify actions taken at run time, the COPY statement specifies an action taken at compile time. When your program contains a COPY statement, the compiler creates a "temporary" source file that is a composite of the CBL file you submit to the compiler and the library text you include with the COPY statement. The compiler processes this temporary source file rather than the file you originally submit to it. Therefore, your program listing (LST file) contains all text included by the COPY statement, but your original source file is unchanged. For more information on the COPY statement, see Chapter 6 of the *COBOL-81 Language Reference Manual*.

Chapter 3
Compiling a COBOL-81 Program

Once you have created your source program and are satisfied with it, you are ready to compile it. This chapter describes how to use the COBOL command to compile your source programs into object files. Topics include:

- Functions of the COBOL-81 compiler
- COBOL command syntax and qualifiers
- Compiler diagnostics and limitations

3.1 Functions of the Compiler

The primary functions of the COBOL-81 compiler are to:

- Verify COBOL source program statements and issue diagnostic messages.
- Generate machine language instructions in the form of an object module (OBJ file) from the source program.
- Produce a skeleton overlay descriptor language file (SKL file) used by the Task Builder to identify independent segments of the source program.
- Create a listing with diagnostics, a data allocation map, and cross-references.

The following is a sample compile command:

```
COBOL PAYROLL
```

This COBOL command invokes the COBOL-81 compiler and specifies that the source file is PAYROLL.CBL. The file type is optional in the command line because CBL is the default. If the compilation is successful, the output is assembled in an object module named PAYROLL.OBJ and a skeleton overlay descriptor language file named PAYROLL.SKL.

3.2 Command Line Format

The format of the command line to the compiler is:

```
COBOL[/qualifiers] file-spec[/qualifiers]...
```

where:

file-spec specifies the files that contain the COBOL-81 source program. If you do not specify a file type, the compiler assumes CBL as the default.

/qualifiers specify compiler options.

3.3 Command Qualifiers

You can use qualifiers to select or suppress compiler options. Table 3-1 lists the COBOL-81 command qualifiers and their normal defaults. Following the table are complete descriptions of the compiler command qualifiers. The default qualifiers are indicated by (D). These defaults can be changed during the installation of the compiler.

Table 3-1: Qualifiers

Qualifier	Summary Description
/ANSI_FORMAT	Accepts a source program in ANSI format.
/NOANSI_FORMAT (D)	Accepts a program in terminal reference format.
/CHECK (D)	Enables range checking of subscripts, indexes, and nested PERFORM statements at run time.
/CHECK:BOUNDS	Enables range checking of subscripts and indexes only.
/CHECK:PERFORM	Enables range checking of nested PERFORM statements at run time.
/NOCHECK /CHECK:NOBOUNDS /CHECK:NOPERFORM	Disables range checking.
/CODE:[NO]CIS	Specifies object code (with or without the Commercial Instruction Set) appropriate for the system that will execute the program.
/CROSS_REFERENCE	Produces a cross-reference table of data and procedure-names in your LST file.
/NOCROSS_REFERENCE (D)	Suppresses production of a cross-reference table.
/DEBUG	Creates symbol information in the object code for use by the Symbolic Debugger.
/NODEBUG (D)	Suppresses creation of the symbol information used by the Symbolic Debugger.
/DIAGNOSTICS[:filename]	Produces a DIA file of the compilation.
/NODIAGNOSTICS (D)	Suppresses production of a DIA file.
/LIST[:filename]	Produces a LST file of the compilation.
/NOLIST (D)	Suppresses production of a LST file.
/NAMES:xx	Changes the PSECT kernel in your object file from SC (the default) to the value you specify for xx.

(continued on next page)

Table 3-1: Qualifiers (Cont.)

Qualifier	Summary Description
/OBJECT[:filename] (D)	Creates an OBJ file as output.
/NOOBJECT	Suppresses the production of an OBJ file.
/SHOW /SHOW:MAP	Produces Procedure Division and Data Division offset maps in your LST file.
/NOSHOW (D)	Suppresses the production of offset maps.
/SUBPROGRAM	Treats the source program as a subprogram.
/NOSUBPROGRAM (D)	Examines the Procedure Division statement to determine if the program is the main or the subprogram.
/TEMPORARY[:device]	Changes the storage area for temporary work files from SY: (the default) to the value you specify for device.
/TRUNCATE	Performs decimal, rather than binary, truncation on COMP data items.
/NOTRUNCATE (D)	Performs binary truncation on COMP data items.
/WARNINGS[:INFORMATIONAL] (D)	Tells the compiler to issue diagnostics including informational messages.
/NOWARNINGS /WARNINGS[:NOINFORMATIONAL]	Suppresses informational diagnostics.

The following are complete descriptions of the COBOL-81 compiler qualifiers:

/ANSI_FORMAT
/NOANSI_FORMAT (D)

The /ANSI_FORMAT qualifier tells the compiler that your program is in conventional (or ANSI) format, rather than the default, terminal format.

/NOANSI_FORMAT is the default.

/CHECK (D)
/CHECK:BOUNDS
/CHECK:PERFORM
/CHECK:NOBOUNDS
/CHECK:NOPERFORM
/NOCHECK

The /CHECK:BOUNDS qualifier compares subscript and index ranges at run time against the ranges defined by corresponding OCCURS clauses. If any range is exceeded during program execution, COBOL-81 issues an error message.

The /CHECK:PERFORM qualifier determines whether or not your program's PERFORM statements are nested properly (if nested at all). If COBOL-81 detects improper nesting during program execution, it issues an error message.

With /CHECK:NOBOUNDS, COBOL-81 does not check subscript and index ranges at run time against the ranges defined by OCCURS clauses. If any range is exceeded during execution, COBOL-81 does not issue an error message.

Similarly, with /CHECK:NOPERFORM, the compiler does not check to determine whether your program's PERFORM statements are nested properly (if at all). If COBOL-81 detects improper nesting during execution, it issues an error message. If you use /CHECK:NOPERFORM, the compiler does not produce diagnostics when PERFORM statements are nested improperly. Do not use the /CHECK:NOBOUNDS and the /CHECK:NOPERFORM qualifiers in the same command line. COBOL-81 issues a message indicating conflicting qualifiers. Instead, use the /NOCHECK qualifier.

The /NOCHECK qualifier tells the compiler to suppress range checking for both subscripts and indexes and for the nesting of PERFORM statements. The purpose of the suppression of checking is to reduce task size and improve performance.

/CHECK is the default.

/CODE:[NO]CIS

The /CODE:CIS qualifier tells the compiler to use CIS (Commercial Instruction Set) in the object code it produces. If the system manager set the default to non-CIS code when COBOL-81 was installed, and if your machine does have CIS, this qualifier overrides that default. This qualifier is needed when you are developing a program to run on a system other than your own. See the system manager if you do not know whether or not your machine has CIS.

/CROSS_REFERENCE
/NOCROSS_REFERENCE (D)

The /CROSS_REFERENCE qualifier tells the compiler to add two cross-reference tables to the end of your list file: one for data-names and one for procedure-names. In each table, the names you used in your program are listed alphabetically. Opposite each name is a list of every line number in which that name occurs. A "D" after a number indicates the line in which you defined the name. An asterisk (*) after a line number indicates a destructive reference, such as a value assignment to a data-name.

Here is an excerpt from a list file (SAMPLE.LST) that resulted from the command line COBOL SAMPLE/CROSS_REFERENCE/LIST:

```
CROSS REFERENCE IN ALPHABETICAL ORDER

DATA NAMES and MNEMONIC NAMES

END-OF-DATA           25D      75       85
FAKE-CARD             18       19D      84
F-NUMBER              22D      82*
```

This qualifier is particularly useful if, for example, one variable yields unexpected results when you run your program. You can trace the variable through your program, and the table gives you a list of the lines to check (see Part II, Chapter 2).

The cross-reference tables are also helpful when you use the Symbolic Debugger.

/NOCROSS_REFERENCE is the default.

/DEBUG
/NODEBUG (D)

The /DEBUG qualifier tells the compiler that you intend to use the COBOL-81 Symbolic Debugger (see Part II, Chapter 3). The compiler then generates symbol information in the object module for all data-names and procedure-names. This increases the size of the object file. However, when you finish debugging and no longer need the symbols, you can recompile without this qualifier.

If you include the Symbolic Debugger in your program, you must also use the /DEBUG qualifier to the LINK/C81 command.

/DIAGNOSTICS[:filename]
/NODIAGNOSTICS (D)

The /DIAGNOSTICS qualifier enables the creation of a diagnostics file with the same file name as the source file and with the file type DIA. The DIA file contains the compiler diagnostic summary. You can specify a different file name or type for this diagnostics file.

/NODIAGNOSTICS is the default.

/LIST[:filename]
/NOLIST (D)

The /LIST qualifier tells the compiler to produce a LST file containing the source code and any diagnostic messages. /LIST is necessary when you want to use the /CROSS_REFERENCE or the /SHOW qualifiers.

If you append the /LIST qualifier to an input file specification instead of to the compile command, the resulting LST file has the same name as the qualified file.

/NOLIST is the default.

/NAMES:xx

The /NAMES qualifier tells the compiler to use the two alphanumeric characters you specify as the PSECT kernel for this program. The only time you need this qualifier is when your executable image uses both subprograms and segmentation. See Part II, Chapter 5 for a detailed explanation.

/OBJECT[:filename]
/NOOBJECT

The /OBJECT qualifier allows you to specify a file other than the default as the compiled object file. /NOOBJECT suppresses the creation of an object file.

If you append the /OBJECT qualifier to an input file specification instead of to the compile command, the resulting OBJ file has the same name as the qualified file.

/OBJECT is the default.

/SHOW
/SHOW:MAP
/NOSHOW (D)
/SHOW:NOMAP (D)

The /SHOW and /SHOW:MAP qualifiers are equivalent. They tell the compiler to add two offset maps to the list file, one referring to the Data Division and one referring to the Procedure Division. The compiler provides these maps for use with ODT (Online Debugging Tool). Consult the *IAS/RSX-11 ODT Reference Manual* for more information.

/NOSHOW or /SHOW:NOMAP is the default

/SUBPROGRAM
/NOSUBPROGRAM (D)

The /SUBPROGRAM qualifier tells the compiler that it is compiling a subprogram. You must use this qualifier only if the subprogram does not use parameters from the main program; that is, if it does not contain the PROCEDURE DIVISION USING header.

/NOSUBPROGRAM is the default.

/TEMPORARY[:device]

The /TEMPORARY qualifier tells the compiler to store its temporary working files on the device you specify during compilation. This qualifier is useful if there is little system disk space available and you want to specify a device other than SY:, which is the default.

/TRUNCATE
/NOTRUNCATE (D)

The /TRUNCATE qualifier tells the compiler to perform decimal truncation on the values of COMPUTATIONAL (or COMP) data items. By default, COBOL-81 performs binary truncation. With binary truncation, the maximum value a COMP item can contain depends on its storage allocation. If you specify the /TRUNCATE qualifier, the maximum value depends on the item's PICTURE character-string.

/NOTRUNCATE is the default.

/WARNINGS (D)
/WARNINGS:INFORMATIONAL
/WARNINGS:NOINFORMATIONAL
/NOWARNINGS

The /WARNINGS and /WARNINGS:INFORMATIONAL qualifiers are equivalent. They tell the compiler to include informational diagnostics during compilation. The /NOWARNINGS and /WARNINGS:NOINFORMATIONAL qualifiers are equivalent. They prevent the compiler from issuing informational diagnostics during compilation. If you use either of these qualifiers, only warning and fatal diagnostics are included in the list file, diagnostic file, and diagnostic summary.

/WARNINGS (or /WARNINGS:INFORMATIONAL) is the default.

3.4 Examples

In addition to producing OBJ and SKL files, the compile command lines in the following examples show the use of various qualifiers.

1.
```
COBOL YEARLY/WARNINGS:NOINFORMATIONAL
```

   Gives you a summary display of warning and fatal errors only.

2.
```
COBOL ANNUAL/LIST/SHOW:MAP/CROSS_REFERENCE
```

   Creates the list file ANNUAL.LST with offset maps and cross-reference tables.

3. `COBOL TEST/TEMPORARY:DK2`

   Uses DK2: for storing temporary files during compilation.

4. `COBOL TEMP/CROSS_REFERENCE`

   Is a meaningless use of the /CROSS_REFERENCE qualifier because no list file has been specified to contain the cross references. COBOL-81 ignores the qualifier, proceeds with the compilation, and gives you a summary of the results.

3.5 Common COBOL-81 Command Line Errors

Some common errors to avoid when you enter COBOL command lines include:

- Omitting the /ANSI_FORMAT qualifier from source programs that are in ANSI format
- Including contradictory qualifiers, such as /SHOW with /NOLIST
- Forgetting to include a file type in a file specification when you do not want the default file type

3.6 Compiler Diagnostics Summary

During a compilation, the COBOL-81 compiler checks your program against COBOL syntax and semantic rules. It issues a diagnostic message for each violation it finds. A diagnostic belongs to one of three classes, depending on the function it serves: I (Informational), W (Warning), or F (Fatal).

Informational

> If the violation in your source code has no effect on the rest of the program and the recovery is obvious, the compiler can overlook it. However, it issues an informational message to remind you of the COBOL-81 statement the code has violated or to point out potential problems the code creates.

Warning

> If the recovery action taken by the compiler for a particular error can affect the rest of the program, a warning message is issued.

Fatal

> If no recovery action can be taken by the compiler for an error in the code, a fatal message is issued. When a fatal error is found, the compilation temporarily stops. Before resuming, the compiler sometimes skips a section of source code. If source code has been skipped, an informational message is issued to show you where the compilation resumes.

If only informational or warning diagnostics occur, the program is said to have compiled successfully. However, you should check all informationals and all warnings. If the program contains any fatal errors, the compilation is unsuccessful and neither an object file nor an SKL file is created.

For example, assume you typed this line to compile the source program and get an object file and a list file:

```
COBOL DATE/LIST
```

There are several possible responses from the COBOL-81 compiler. One of the possible responses follows:

```
C81 - 0 FATAL ERRORS
C81 - 2 WARNINGS
C81 - 3 INFORMATIONALS
```

This indicates successful compilation because there are no fatal errors; that is, DATE.OBJ and DATE.SKL were created. However, you should examine DATE.LST to see which rules the source code violated.

Another possible response includes the following lines:

```
C81 - 2 FATAL ERRORS - object deleted
C81 - 2 WARNINGS
C81 - 4 INFORMATIONALS
```

Due to the two fatal errors, the compilation is unsuccessful and the compiler did not create DATE.OBJ or DATE.SKL. You must examine DATE.LST for the diagnostic messages, correct DATE.CBL, and compile it again.

The compiler issues (that is, puts in the list and diagnostic files) a maximum of 500 errors. The compilation continues and the summary count continues to be updated, but you have no way of knowing what errors occurred after the first 500.

See Appendix B for a list of the COBOL-81 compiler error messages.

3.7 Compiler Limitations Summary

There are several implementation limits to the COBOL-81 compiler, and you receive diagnostics if you exceed them. For a list of these limitations, see Appendix A.

Chapter 4
Linking a COBOL-81 Program

After you have compiled your source program, you must link the resulting object modules(s) to create an executable (or task) image of your program. The LINK/C81 command accepts COBOL-81 OBJ files and SKL files as input, invokes the COBOL-81 BLDODL utility to create ODL and CMD files, and calls the Task Builder, which produces an executable image (TSK file).

This chapter describes the use of the DCL LINK/C81 command. See Appendix D for information about using MCR (Monitor Console Routine) commands to produce an executable image. You must refer to Appendix D if you plan to use BLDODL utility options that DCL does not provide.

4.1 Functions of the LINK Command

The primary functions of the LINK/C81 command are to allocate memory within the executable image, to resolve symbolic references among the modules being linked, to assign values to relocatable global symbols, and to perform relocation.

4.2 Using the LINK/C81 Command to Build an Executable Image

To link your program, you specify the SKL file as the input file specification to the LINK/C81 command. The resulting output (or task) file (containing the executable image) has the same file name as the input file, but its file type is TSK. The format of the LINK/C81 command is:

```
LINK/C81[/qualifier] file-spec[, ...] [/qualifier][, ...]
```

where:

file-spec specifies the COBOL-81 object file(s) to be linked.

/qualifiers specify options to the command.

There are two categories of qualifiers to the LINK/C81 command. Library routine qualifiers are discussed in Section 4.2.1 and output file qualifiers are discussed in Section 4.2.2

4.2.1 Library Routine Qualifiers

In most cases, the LINK/C81 command supplies your program with the run-time and I-O support you need by default, and you do not need to use any qualifiers to include:

- Disk-resident RMS-11 libraries for Record Management Services I-O, if your program requires RMS
- Disk-resident COBOL-81 OTS library C81LIB or C81CIS, depending on whether or not your system has the Commercial Instruction Set (CIS)

For example:

```
LINK/C81/RMS:RES/OTS:RES PAYROLL
```

This command includes memory-resident libraries when you have them installed on your system, and it clusters these libraries without requiring any input from you. Only use this command if both libraries have been installed as memory-resident on your system.

The following qualifiers allow you to specify libraries other than the defaults:

/FMS

Using the /FMS qualifier indicates that you are including FMS (File Management Support) library support in your task image. You must specify FMS support if you call FMS routines from your program.

/OTS:RESIDENT
/OTS:NORESIDENT

The /OTS:RESIDENT qualifier includes memory-resident OTS, either C81CIS or C81LIB, in your task if memory-resident OTS is installed. If you use the /OTS:RESIDENT qualifier and your system does not have memory-resident OTS, an error message will be issued at link time.

The /OTS:NORESIDENT qualifier creates a reference to the disk-resident OTS library.

/RMS:RESIDENT
/RMS:NORESIDENT

The /RMS:RESIDENT qualifier creates a reference to the shared RMS-11 memory-resident library. The resulting executable image is smaller and uses the resident library, RMSRES, at run time.

The /RMS:NORESIDENT qualifier creates a reference to the disk-resident RMS library.

4.2.2 Output File Qualifiers

The following qualifiers control the output files produced by the LINK/C81 command.

/MAP
/NOMAP

Use the /MAP qualifier to indicate that you want the LINK/C81 command to produce a memory map file. The /NOMAP qualifier indicates that you do not want a memory map file.

If you use /MAP, you can also assign a file specification for the memory map file that is different from the file specification in your command line. If you omit the file specification, LINK/C81 produces a map file on the same device, with the same directory and name, as the executable file. The default file type is MAP.

/NOMAP is the default.

/DEBUG
/NODEBUG

Use the /DEBUG qualifier to indicate that you are including the COBOL-81 Symbolic Debugger in your task. You must have used the /DEBUG compiler qualifier to the COBOL command to use the /DEBUG qualifier to the LINK/C81 command.

/NODEBUG is the default.

4.3 Linking Error Message Summary

If the Task Builder detects any errors while linking object modules, it displays messages indicating the cause and severity of the error. If any fatal error conditions occur, that is, errors with severities of F, the Task Builder does not produce an image file. MAP files with diagnostics are produced, however, if you use the /MAP qualifier.

A common error that can occur during linking is that a reference to a symbol name remains unresolved. This error occurs when you omit required library qualifiers from the LINK/C81 command and the Task Builder cannot locate the definition for a specified global symbol reference. It can also occur when a subprogram to be called from the main program has been omitted from the LINK/C81 command line. If an error occurs when you build modules, you can often correct the error by reentering the command line and specifying the correct modules or libraries.

If execution of the LINK/C81 command is unsuccessful, an error message is issued to your terminal before the return to the system prompt.

For example, command execution is unsuccessful if the resulting image is too large. The following message indicates this condition:

```
SEGMENT seg-name HAS ADDR OVERFLOW: ALLOCATION DELETED
```

Seg-name is the name of the object file the Task Builder was processing when the overflow occurred. To recover, you must make more efficient use of memory by overlaying sections of the task. Use the COBOL-81 segmentation facility to overlay your object code or the BLDODL utility to overlay RMS-11 routines. See Part II, Chapters 4 and 5 for information about these facilities.

Refer to the *RSX-11M/M-PLUS Task Builder Manual* for an explanation of all other link-time errors.

Chapter 5
Running a COBOL-81 Program

Your COBOL-81 TSK file is an executable form of the declarations and instructions represented in your COBOL source program. It includes I/O routines and other subprograms inserted by the Task Builder as a result of your commands or the contents of the ODL file. It also includes the COBOL-81 run-time system, which is a library of predefined routines that perform standard functions for your program, such as arithmetic and data movement. The run-time system is also referred to as the Object Time System (OTS).

5.1 Functions of the COBOL-81 Object Time System (OTS)

The principal functions of the COBOL-81 OTS include:

- Arithmetic operations
- Input-output operations
- Subscripting, indexing, and table handling
- String operations
- Error handling

5.2 Command Line Format

To run a COBOL-81 program, type:

```
RUN task-file
```

where:

task-file names your executable (or task) image. The default file type is TSK.

5.3 Run-Time Error Message Summary

The COBOL-81 Object Time System checks adherence to COBOL-81 general rules and issues error messages to your terminal if there is a general rule violation in your program.

If a run-time error occurs, the OTS issues two lines of information to your terminal. The first line contains the number associated with the error message and a description of the error. The second line contains the location of the error in your source code.

The general format of this information is:

nn message-text
Error occurred in program *program-name* at line number n.

where:

nn — identifies the number associated with the error. For a complete listing of run-time error messages, see Appendix C.

message-text — describes the error.

program-name — is the name that appears in the PROGRAM-ID paragraph of the source program in which the error has occurred.

n — is the compiler-generated line number of the source statement that caused the error.

For example, if PAYROLL.TSK attempts to read a record from a file that is not open, this message is issued:

```
33  Program attempted an I/O operation on a file that is not open.
    Error occurred in program PAYROLL at line number 32.
```

If the program is executing within one or more PERFORM statements when the error occurs, the line number of each active PERFORM statement will be issued in addition to the message (unless you compiled the program with the /NOCHECK or the /CHECK:NONE qualifier). For example:

```
2   Program attempted to exit PERFORMs in the wrong order.
    Error occurred in program REPORT at line number 22.

The currently active perform line number(s) are:
25
22
```

If the error occurs in a called program, the calling sequence will be issued in addition to the message. For example:

```
42  Program attempted division by zero.
    Error occurred in program SUBR5 at line number 15.

SUBR5 was called from line number 20 in program SUBR4.
SUBR4 was called from line number 43 in program SUBR3.
SUBR3 was called from line number 35 in program SUBR2.
```

See Appendix C for the complete list of run-time error messages, along with suggested recovery actions.

Appendix A
COBOL-81 Compiler Implementation Limitations

This appendix lists the implementation limitations for the COBOL-81 compiler. The compiler issues diagnostics whenever you exceed any of these limits.

1. The run-time storage that the compiler allocates for object code and data cannot exceed 65535 bytes.
2. The run-time storage that the compiler allocates for an indexed file's RECORD KEY or ALTERNATE KEY data item cannot be greater than 255 bytes.
3. The number of ALTERNATE KEY data items for an indexed file cannot exceed 254.
4. The value of the integer in the EXTENSION option of the APPLY clause must be from 0 to 65535 (inclusive).
5. The number of SAME AREA or SAME RECORD AREA clauses cannot exceed 30.
6. The length of any record in a file description cannot exceed 16384 bytes.
7. The physical block size for a sequential tape file must be from 18 to 8192 bytes (inclusive).
8. A PICTURE character-string cannot contain more than 30 characters.
9. PICTURE character-strings for alphanumeric edited or numeric edited data items cannot represent more than 255 standard data format characters (each character occupies one byte in storage).

10. PICTURE character-strings for alphanumeric or alphabetic data items cannot represent more than 65535 standard data format characters.

11. A nonnumeric literal cannot contain more than 256 characters.

12. The number of operands in the USING phrase of a CALL statement cannot be greater than 255.

13. The number of operands in a DISPLAY statement cannot be greater than 254.

14. The number of operands in a GO TO DEPENDING statement cannot be greater than 512.

15. The integer in the TIMES option of the PERFORM statement cannot exceed 2,147,483,648.

16. The number of SORT keys is limited to 16. Each of these must be less than 256 characters. The sum of the keys must be less than or equal to 512 characters.

Appendix B
Compiler Error Messages

This appendix lists the COBOL-81 compiler error messages that are preceded by an asterisk (*). The compiler error messages have been made as self-explanatory as possible; however, some require further clarification. In these cases, the compiler issues the asterisk along with the message to tell you that more information is available in this appendix. For each message, a numeric code and severity level are provided. There are three severity levels: Informational (I), Warning (W), and Fatal (F).

004 I ***This literal contains a non-printing character.**

If you intended to include the non-printing character in the literal, no further action is necessary. If not (that is, if the character is there by error), use a text editor to correct the literal.

010 W ***This character is not in the COBOL character set.**

The compiler ignores the character.

016 F ***This picture string is too long.**

This is due to one of the following:

- More than 30 characters in the PICTURE string
- More than 18 digits represented for a numeric item
- More than 256 characters represented for a nonnumeric item

019 F ***This is not a valid picture string.**

See the *COBOL-81 Language Reference Manual* for the syntax rules regarding PICTURE character-strings.

025 F ***Invalid character used with repeat count.**

See the *COBOL-81 Language Reference Manual* for the syntax rules regarding PICTURE strings.

201 I ***The redefining item is smaller than the redefined item.**

If the difference in the items' sizes is not intentional, correct the source code.

203 I *Fill bytes have been inserted in this record or item.

Fill bytes will make this item longer than you might expect. To see how much space the compiler allocates for this item, use the /SHOW:MAP qualifier when you compile the program.

229 F *The VALUE clause is incompatible with the category of this item.

A numeric item can have only a numeric literal value. An alphanumeric, alphanumeric edited, numeric edited, or alphabetic item can have only a nonnumeric literal value.

233 F *This value exceeds the maximum for this parameter.

See the *COBOL-81 Language Reference Manual* for the rules and limits for the particular statement causing the error.

263 F *Multiply defined name.

Each name in a COBOL-81 program can belong to only one set of user-defined words. Condition-names, data-names, and record-names belong to the same set; therefore, you can define a data-name, record-name, and condition-name with the same word. In these cases, you can qualify the word so that there is no ambiguity when you reference an item.

However, if you define a word to be in one of the following sets, you cannot use that word to define an item in any of the other sets:

- Alphabet-name
- File-name
- Index-name
- Mnemonic-name
- Paragraph-name
- Program-name
- Section-name
- Text-name

Revise your source code so that the name does not duplicate a name in another set.

268 F *Use of this file in a SAME AREA clause implies its usage in more than one SAME RECORD AREA clause.

If any file in a SAME AREA clause is in a SAME RECORD AREA clause, all files in that SAME AREA clause must be in the SAME RECORD AREA clause.

300 I *The redefining item is larger than the redefined item.

If the difference in the items' sizes is not intentional, correct the source code.

310 W *The RECORD CONTAINS value is greater than length of longest record.

When the program reads or writes a record, it will access the number of characters specified by the RECORD CONTAINS clause, including characters you did not define as part of the record.

311 W *The length of longest record is greater than RECORD CONTAINS value.

The RECORD CONTAINS value is ignored. When the program reads or writes a record, it will access the number of characters specified by the longest record description.

312 W *The upper bound on RECORD VARYING clause is greater than longest record length.

The upper bound will be used to determine the maximum number of characters that can be read or written.

316 I *Fill bytes have been inserted before this data item.

Fill bytes will make this item longer than you might expect. To see how much space the compiler allocates for this item, use the /SHOW:MAP qualifier when you compile the program.

353 W *Index data items are allocated differently using VAX–11 COBOL.

The compiler issues this message only if you compile your program with the /STA:VAX switch.

USAGE IS INDEX data items in COBOL-81 are two bytes long. In VAX–11 COBOL, they are four bytes long. You must not store these items in COBOL-81 files if a VAX–11 COBOL program also accesses those files. Appendix D of the *COBOL-81 Language Reference Manual* explains how you must use index data items in your program to be compatible with VAX–11 COBOL.

354 W *The storage allocation of this item is incompatible with VAX–11 COBOL.

The compiler issues this message only if you compile your program with the /STA:VAX switch.

COBOL-81 aligns USAGE IS COMP items on a word boundary. VAX–11 COBOL aligns these items on any byte boundary. If your program generates files containing COMP data items and a VAX–11 COBOL program must access those files, you must resolve this incompatibility. The easiest way to do this is to specify the SYNCHRONIZED clause for all COMP items in your program. Appendix D of the *COBOL-81 Language Reference Manual* also explains a manual method (explicit FILLER item insertion) you can use to resolve COMP alignment differences.

453 F *Ambiguous reference.

Your reference to this item is not unique; that is, your reference points to more than one user-defined word in your program. Remember that data-names and procedure-names need to be unique only if you refer to them in Procedure Division statements. This message appears when you define a name more than once in your program and cannot use qualification to make it unique. In this case, you must change one of the duplicate names to a unique one.

501 I *Compilation resumed at this point.

This message may be issued after the compiler detects a fatal error. It indicates that the compiler skipped source code from the last error to this point, and in doing so it might have overlooked errors or created new errors. For example, skipping a SELECT clause will cause an error when the compiler encounters the file description.

512 F ***A required operand is missing.**

See the *COBOL-81 Language Reference Manual* for the rules regarding the statement causing this error.

520 F ***Invalid PROCEDURE DIVISION header.**

See the *COBOL-81 Language Reference Manual* for the rules regarding the two formats for this header.

532 F ***Illegal combination of sending and receiving items.**

See the *COBOL-81 Language Reference Manual* for tables of valid MOVE or SET operand combinations.

543 F ***This program-name contains an invalid character.**

A program-name can contain characters from the set A through Z, a through z, 0 through 9, and hyphen (-).

637 F ***Invalid use of the INTO phrase.**

When you use the INTO phrase of a READ or RETURN statement, one of the following conditions must be true:

- Only one record description applies to the file.
- All record descriptions for the file and the receiving data item are group items or alphanumeric elementary items.

Either change your program so that these conditions are true or delete the INTO phrase from the statement.

Appendix C
Run-Time Error Messages

This appendix lists the COBOL-81 run-time error messages. COBOL-81 displays these messages when it detects errors during program execution.

Each message is followed by an explanation and one or more suggested solutions.

Messages preceded by asterisks (*) indicate Synchronous System Traps, which are described in the documentation set for your operating system. A Synchronous System Trap can occur if:

- You compile your program with any of the following qualifiers: /NOCHECK, /CHECK:NONE, /CHECK:NOPERFORM, or /CHECK:NOBOUNDS
- You get a link-time error but execute the task anyway

When a Synchronous System Trap occurs, the line number of the source code is not available; COBOL-81 displays the object-address instead. To associate this address with the source program statement that caused the error, use the memory allocation map produced by the LINK/C81 command (when its /MAP qualifier is specified). Revise the source code so that the PERFORM statement (or the subscript or index) is within the range specified in the program.

COBOL-81 also issues message 15, RESERVED INSTRUCTION, if a program was compiled with the /CODE:CIS qualifier and you attempt to run it on a system without the Commercial Instruction Set (CIS). In this case, you must recompile the program with the /CODE:NOCIS qualifier and then recreate the task image.

If the corrective actions just described do not solve the problem, submit an SPR (Software Problem Report).

1 **Program attempted to PERFORM a range that is already being performed.**

An active PERFORM range must be exited before it can be performed again. Revise the program logic.

2 **Program attempted to exit PERFORMs in the wrong order.**

All PERFORM ranges, whether physically nested or not, must be exited in reverse order. Revise the program logic.

3 **A subprogram attempted to CALL itself either directly or indirectly.**

The EXIT PROGRAM statement must be executed in a subprogram before the subprogram can be called again. Revise the program logic.

4 **Number of parameters used by subprogram does not equal number in CALL statement.**

The number of arguments in the calling program's CALL statement must equal the number of arguments in the called program's PROCEDURE DIVISION USING header. Revise the source code.

5 **Program evaluated a subscript outside the range of the OCCURS clause.**

A subscript value must be within the range defined by the OCCURS clause, and not less than one. Determine why the program is using a value outside the range and revise the code.

6 **Program attempted to OPEN more than one file specified in a SAME AREA clause.**

Only one file in a SAME AREA clause can be open at one time. Revise the program logic.

7 **Version number of program does not match that of the OTS.**

This error will occur in two situations:

- When a new version of the OTS has been installed and the version numbers of the main and subprograms do not match. In this case, recompile all the programs in the task and build a new executable image.
- When a new version of the compiler has been installed but a new OTS has not been installed. In this case, install the new OTS and build a new executable image.

8 **Environmental integrity fault.**

Part of the run-time system has been damaged by the program or by an error in the run-time system itself. This error could result from an out-of-range subscript in a program compiled with qualifiers that suppress subscript range checking (/NOCHECK, /CHECK:NONE, /CHECK:NOBOUNDS). Recompile the program with the /CHECK:BOUNDS qualifier and correct any errors the bounds checking detects. If the error still occurs, submit an SPR (Software Problem Report).

9 **Program attempted to OPEN more than the maximum of 15 files.**

No more than 15 files can be open at the same time. Revise the source code.

10 **Error in an input-output operation. The RMS error code is *n*.**

This is an RMS-specific error; see the *RMS-11 User's Guide* for more information.

An appropriate USE procedure in the program can handle this error during execution. The file status key value for this error is 98 if the operation was CLOSE, and 30 for all other operations.

*11 **ODD ADDRESS FAULT.**

*12 **MEMORY PROTECTION VIOLATION.**

*13 **BPT INSTRUCTION.**

*14 **IOT INSTRUCTION.**

*15 **RESERVED INSTRUCTION.**

*16 **INVALID EMT INSTRUCTION.**

*17 **TRAP INSTRUCTION.**

*18 **FLOATING POINT EXCEPTION.**

19 **Program could not find a record with the record key specified.**

Determine why the record was not in the file. If this error is to be expected, add a routine to the source code that will handle it during execution. It can be handled by either:

- The INVALID KEY phrase.
- An appropriate USE procedure. The file status key value for this error is 23.

20 **Program attempted to WRITE beyond the boundary of the file.**

This error occurs if the system cannot extend the file. To recover, make the file noncontiguous (see your operating system documentation for instructions).

To handle this error during execution, add a routine to the source code. The error can be handled by either:

- The INVALID KEY phrase.
- An appropriate USE procedure. The file status key value for this error is 34 if the organization is sequential, and 24 if it is indexed.

21 **Program attempted to OPEN a file that another program is accessing.**

A file cannot be accessed for writing or updating by more than one program at a time unless ALLOWING ALL was specified in its OPEN statement. Either revise the OPEN statement or add an appropriate USE procedure to the program to handle the error during execution. The file status key value for this error is 91.

22 **Program attempted to access a record that another program is writing or updating.**

A record cannot be written or updated by more than one program at a time. An appropriate USE procedure in the program can handle this error during execution. The file status key value for this error is 92.

23 **Program attempted an I-O operation that is invalid with the OPEN mode for the file.**

Either revise the source code (changing the OPEN mode or the I-O operation) or add an appropriate USE procedure to the program to handle the error during execution. The file status key value for this error is 94.

24 **Program attempted a DELETE or REWRITE not immediately preceded by a successful READ.**

Either revise the source code or add an appropriate USE procedure to the program to handle the error during execution. The file status key value for this error is 93.

25 **Program attempted to WRITE or REWRITE a record with a nonascending key value.**

If nonascending keys are to be allowed, change the access mode to RANDOM or DYNAMIC. Otherwise, add a routine to the program to handle the error during execution. The error can be handled by either:

- The INVALID KEY phrase.
- An appropriate USE procedure. The file status key value for this error is 21.

26 **Program attempted to WRITE or REWRITE a record with a duplicate key value.**

If the error occurred on an alternate key, recreate the file specifying DUPLICATES. Otherwise, add a routine to the program to handle the error during execution. The error can be handled by either:

- The INVALID KEY phrase.
- An appropriate USE procedure. The file status key value for this error is 22.

27 **Program attempted to REWRITE a record after changing the value of its primary key.**

For indexed files in sequential access mode, the values of the primary keys in the record to be replaced and the last record read from the file must be equal. Either revise the source code (changing the access mode to RANDOM or DYNAMIC) or add a routine to the program to handle the error during execution. The error can be handled by either:

- The INVALID KEY phrase.
- An appropriate USE procedure. The file status key value for this error is 21.

28 **Program attempted to READ from a missing optional file.**

Add a routine to the program to handle this error during execution. It can be handled by either:

- The AT END phrase.
- An appropriate USE procedure. The file status key value for this error is 15.

29 **Program could not find a file specified to be OPENed.**

Determine whether the values in the ASSIGN and VALUE OF ID clauses are incorrect or the file is actually missing. An appropriate USE procedure in the program can handle this error during execution. The file status key value for this error is 97.

30 **Program attempted to create a file but could not find enough space.**

This error occurs if the device in the file specification does not contain the preallocation amount of space. (If the program does not specify a preallocation amount in its APPLY clause, the RMS-11 default is four blocks for a sequential file and four times the bucket size for an indexed file.) Either change the preallocation amount in the APPLY clause, create more space on the device, or add an appropriate USE procedure to the program to handle the error during execution. The file status key value for this error is 95.

31 **Program attempted an I-O operation on a file that was already closed.**

Either revise the program logic or add an appropriate USE procedure to the program to handle the error during execution. The file status key value for this error is 94.

32 **Program attempted to READ a record but the end of the file was reached.**

Add a routine to the program to handle this error during execution. It can be handled by either:

- The AT END phrase.
- An appropriate USE procedure. The file status key value for this error is 13 for the first occurrence, and 16 for each subsequent occurrence before a valid record position is established.

33 **Program attempted an I-O operation on a file that is not open.**

Either revise the program logic or add an appropriate USE procedure to the program to handle the error during execution. The file status key value for this error is 94.

34 **Program attempted to OPEN a file that was previously closed WITH LOCK.**

Either revise the program logic or add an appropriate USE procedure to the program to handle the error during execution. The file status key value for this error is 94.

35 **Program attempted to OPEN a file that is already open.**

Either revise the program logic or add an appropriate USE procedure to the program to handle the error during execution. The file status key value for this error is 94.

36 **Program attempted to use an exponent greater than the 99 maximum.**

Determine how the exponent was generated and revise the source code.

37 **Program attempted to use a non-integer exponent.**

Determine how the exponent was generated and revise the source code.

38 **Program generated an intermediate value that exceeded the maximum of 10 to the 99th.**

Determine how the value was generated and revise the source code.

39 **Program failed in an attempt to ACCEPT data.**

This is an indication of a hardware error. The terminal is busy, off line, or otherwise unavailable.

40 **Program failed in an attempt to DISPLAY data.**

This is an indication of a hardware error. The terminal is busy, off line, or otherwise unavailable.

41 **Program encountered a protection code violation when it tried to OPEN a file.**

This condition can be handled two ways:

- By an appropriate USE procedure in the program. The file status key value for this error is 30.
- By lowering the file's protection code. Use the SET PROTECTION command.

42 **Program attempted division by zero.**

Determine how the zero value was generated and revise the source code.

43 **Program attempted to raise zero to the zero power.**

Determine how the values were generated and revise the source code.

44 **Program attempted to execute a PERFORM statement beyond the maximum number of times.**

The maximum number of times is 2,147,483,648. Determine how the value was generated and revise the source code.

45 **SELECT clause organization does not match organization of file opened.**

File organization is fixed when the file is created and cannot be subsequently changed. Revise the ORGANIZATION clause in the source code.

46 **SELECT clause index key description does not match that of file opened.**

The data descriptions of all record keys, and their relative locations in the record, must be the same as when the file was created. Revise the SELECT clause in the source code.

47 **Program attempted to create a print file on a tape without specifying PRINT CONTROL.**

The PRINT-CONTROL phrase of the APPLY clause must be specified for print files on magnetic tape. Revise the source code.

48 **Program attempted to activate a USE PROCEDURE which is already active.**

One USE AFTER EXCEPTION procedure can invoke another. However, a USE AFTER EXCEPTION procedure must return control to the routine that invoked it before it can be invoked again. Revise the source code.

49 **The BY operand in a PERFORM VARYING is zero.**

The value for the index-name following the VARYING BY phrase must not be zero. Determine how the zero value was generated and revise the source code.

50 **Invalid LINAGE value.**

The values for data items specified as *page-lines* or *footing-line* must be greater than zero. Determine how the zero value was generated and revise the source code.

51 **LINAGE value is less than FOOTING value.**

In the LINAGE clause, the value for a data-name specified as *page-lines* must be greater than or equal to the value for a data-name specified as *footing-line*. Determine which value is incorrect and revise the source code.

52 **Enter the numbers of switches you want "ON" during program execution.**

The program you are running uses external switches to determine internal logic paths. Consult the documentation accompanying your program to determine which switches should be set "ON" for your application or environment. Then, enter the numbers of the switches you want set (separated by commas, spaces, or tabs). Or, if you want to set all switches in the program, you can simply enter an asterisk (*).

53 **OCCURS DEPENDING ON item is not within the specified range.**

In the OCCURS clause, all values for a data-name specified in the DEPENDING ON phrase must be within the range delimited by OCCURS *min-times* TO *max-times*. Determine why an invalid value is being generated and revise the source code.

54 **Attempt to WRITE a record that already exists in a relative file.**

An appropriate USE procedure in the program can handle this error during execution. The file status key value for this error is 22.

55 **Switch input is invalid or out of range.**

COBOL-81 programs can test the status of no more than 16 switches. Switch numbers must be integers within the range 1 to 16 (inclusive). Switch input is out of range if you entered an integer that is less than 1 or greater than 16. Input is invalid if you entered a switch number that is not an integer or is not specified in your program.

When you enter the input character string, each switch number must be separated by a space, a comma, or a tab character. Typing the string incorrectly results in input that is invalid or out of range.

Re-execute your program and enter valid switch numbers correctly.

56 **Switch input string must be 64 characters or less.**

The string of input characters you enter is restricted to a maximum of 64 characters. Only integers (1 through 16, inclusive), separated by commas, spaces, or tabs should appear in the input string. Re-enter the characters correctly.

58 **Attempt to start a SORT/MERGE when one is already in progress.**

A task must execute SORT and MERGE statements one at a time; that is, it cannot execute a SORT or MERGE statement before another has finished execution. This error could occur when a task contains subprograms and two programs in the task attempt simultaneous sort or merge operations.

59 **Attempt to RELEASE/RETURN a record from the wrong sort or merge file.**

The SD entry for the sort or merge file does not contain the record specified in the SORT or MERGE statement. Revise the source code.

60 **Attempt to RELEASE a record during a merge operation.**

The MERGE statement cannot specify an input procedure. Therefore, RELEASE is an invalid statement during a merge operation. Revise the source code.

61 **Error in a sort or merge operation. The SORT/MERGE error code is *n*.**

Submit an SPR (Software Problem Report).

Appendix D
MCR Commands for COBOL-81

All of the examples in this manual for compiling, linking, and running COBOL-81 programs were written using the DIGITAL Command Language (DCL). This appendix provides equivalent Monitor Console Routine (MCR) commands.

You must use the MCR interface for program development when:

- You want to edit a command (CMD) file or an overlay descriptor language (ODL) file for input to the Task Builder.
- You want to specify disk resident RMS-11 libraries that are less heavily overlaid than the LINK/C81/RMS:NORESIDENT default.
- You want to specify a user library as input to the Task Builder.
- You want to store or delete SKL files in a directory other than the one in which the OBJ files reside.

If you are in DCL and you want to change to MCR to use the commands in this appendix, you type:

```
SET TERMINAL MCR
```

You will remain in MCR mode until you log off. If you want MCR to be the default command line interpreter for your terminal, have the system manager change your account file.

D.1 Compiling

The compiler scans your source statements for syntax and semantic errors. Once it has finished, it responds with a diagnostic summary of any errors it found. You can also request additional compiler functions depending on the command line you use.

D.1.1 Using the Compiler

To invoke the COBOL-81 compiler, type:

```
C81 RET
```

The RETURN key is optional. If you press it, the compiler gives you this prompt:

```
C81>
```

Then you must enter a command line as input to the compiler. If you do not press the RETURN key, you must enter compiler commands on the same line.

If you type /HELP instead of a command line, the compiler displays a help message on your screen. The message summarizes the command line format and switches. The HELP text also specifies current default switches for the compile command line.

The format of the command line to the compiler is:

```
[obj-file],[list-file],[diag-file]=source-file[/switch] ...
```

This command produces an object file, a list file, and a diagnostics file. The diagnostics file is simply a subset of the list file. It contains only the diagnostics issued, along with the line of source code to which each diagnostic applies. A fourth file, with the file type SKL, is produced along with each object file. It consists of skeleton overlay descriptor language, which is needed as input to the COBOL-81 BLDODL utility. You control the production of all four files through your input to the compiler.

The default file types are:

OBJ	for the object file
LST	for the list file
DIA	for the diagnostics file
CBL	for the source file

The object, list, and diagnostics files are optional; each is produced only if you give a file name for it. The compiler expects the file names, separated by commas, in the order shown in the command line format. If you want to stop the compiler from producing one file but not the one following it in the command line, you must still type the comma for the one you wish to omit.

To illustrate the use of the commas, here are sample command lines showing the eight possible combinations of input and output files:

1. `C81 FILE1,FILE2,FILE3 = FILE`

   Creates FILE1.OBJ, FILE1.SKL, FILE2.LST, and FILE3.DIA from FILE.CBL.

2. `C81 MONTH, MONTH = MONTH`

   Creates MONTH.OBJ, MONTH.SKL, and MONTH.LST from MONTH.CBL.

3. `C81 LABEL, , LABEL = PROGRM`

   Creates LABEL.OBJ, LABEL.SKL, and LABEL.DIA from PROGRM.CBL.

4.
   ```
   C81 , LIST, TT: = MYFILE.CRG
   ```

   Creates LIST.LST from the source file MYFILE.CRG. The diagnostics are output directly to your terminal; that is, they are displayed on the screen rather than stored in a file.

5.
   ```
   C81 INVEST = TANZAN.ITE
   ```

   Creates the object module INVEST.OBJ and its accompanying SKL file (INVEST.SKL) from TANZAN.ITE.

6.
   ```
   C81 , TI: = GRADE
   ```

   Creates only a temporary list file from GRADE.CBL, and outputs it directly to the terminal.

7.
   ```
   C81 , , REPORT = REPORT
   ```

   Creates the diagnostics file REPORT.DIA from REPORT.CBL.

8.
   ```
   C81 = TEST.311
   ```

   Compiles the source file TEST.311 but creates no output files. You get the diagnostic summary, however.

You can request various compiler functions by indicating compiler switches after you specify the source file in the command line. Table D-1 summarizes the available compiler switches and the functions they perform.

Table D-1: Summary of Compiler Switches

You Type	To Tell the Compiler
/BLD	To create an ODL and a CMD file to submit to the Task Builder
/CIS	To use CIS (Commercial Instruction Set) in the object code
/CRF	To produce a cross-reference table of data and procedure names in your LST file
/CVF	To accept a source program in conventional (ANSI) format
/DEB	To create symbol information in the object code for use by the Symbolic Debugger
/MAP	To produce Procedure Division and Data Division offset maps in your LST file
/STA:VAX	To flag COBOL-81 code that might be incompatible with VAX–11 COBOL code
/SUB	To treat the source program as a subprogram
/TRU	To perform decimal, rather than binary, truncation on COMP data items
/-BOU	Not to produce the code needed for checking subscript and index ranges at run time
/-CIS	Not to use CIS in the object code
/-INF	Not to issue informational diagnostics
/-PER	Not to produce the code needed for checking nested PERFORM statements at run time
/-SKL	Not to produce a skeleton overlay descriptor language file
/FIPS:74	To change values for FILE STATUS data items and sizes for some arithmetic operations
/KER:xx	To change the PSECT kernel in your object file from SC (the default) to the value you specify for "xx"
/TMP:dev	To change the storage area for temporary work files from SY: (the default) to the value you specify for "dev"

As shown in the table, the 17 available switches can be divided into three groups:

- Those requesting special functions
- Those suppressing normally performed functions
- Those altering normally performed functions

D.1.1.1 Switches Requesting Special Functions — There are nine special functions available. By default, the compiler does not perform any of these, unless the default is changed by your system manager when installing the compiler.

/BLD

Tells the compiler to create an ODL and a CMD file. The Task Builder needs both files if your task uses overlaid or resident library I/O, uses segmentation, or includes the Symbolic Debugger. Using this switch is equivalent to using the BLDODL utility without specifying any switches in the BLDODL command line. See Section D.2 on BLDODL.

The /BLD switch cannot be used when compiling subprograms or programs that call subprograms.

/CIS

Tells the compiler to use CIS (Commercial Instruction Set) in the object code it produces. If the system manager set the default to non-CIS code when COBOL-81 was installed, and your machine does have CIS, this switch overrides that default. See the system manager if you do not know whether or not your machine has CIS.

/CRF

Tells the compiler to add two cross-reference tables to the end of your list file: one for data-names and one for procedure-names. In each table, the names you used in your program are listed alphabetically. Opposite each name is a list of every line number in which that name occurs. A "D" after a number indicates the line in which you defined the name. An asterisk (*) after a number indicates a destructive reference.

Here is an excerpt from a list file (SAMPLE.LST) which resulted from the command line "C81 , SAMPLE = SAMPLE/CRF":

```
CROSS REFERENCE IN ALPHABETICAL ORDER

DATA NAMES and MNEMONIC NAMES

END-OF-DATA          25D     75      85
FAKE-CARD            18      19D     84
F-NUMBER             22D     82*
```

This switch is particularly useful if, for example, one variable yields unexpected results when you run your program. You can trace the variable through your program, and the table gives you a list of the lines to check.

The cross-reference tables are also helpful when you use the Symbolic Debugger.

/CVF

Tells the compiler that your program is in conventional (or ANSI) rather than terminal format.

/DEB

Tells the compiler that you intend to use the Symbolic Debugger (see Part II, Chapter 3). The compiler then generates symbol information in the object module for all data names and procedure names. This increases the size of the object file. When you finish debugging and no longer need the symbols, you can recompile without this switch.

If you include the Symbolic Debugger in your program, you must also do one of the following:

- Use the compiler's /BLD switch. For example:

```
C81 TAXDAT = TAXDAT/DEB/BLD
```

- Use the BLDODL utility, specifying its /DEB switch. For example:

```
BLD>TAXDAT = TAXDAT/DEB
```

/MAP

Tells the compiler to add two offset maps to the list file, one referring to the Data Division and one referring to the Procedure Division. The compiler provides these maps for use with ODT (Online Debugging Tool); consult the *IAS/RSX-11 ODT Reference Manual* for more information.

/STA:VAX

Tells the compiler to flag COBOL-81 code that is incompatible with VAX–11 COBOL code in the following situations:

- some COMP items (without SYNC clause)
- USAGE IS INDEX items
- RMS-STS AND RMS-STV

/SUB

Tells the compiler that it is compiling a subprogram. You must use this switch only if the subprogram does not use parameters from the main program; that is, if it does not contain the PROCEDURE DIVISION USING header.

/TRU

Tells the compiler to perform decimal truncation on the values of COMP data items. By default, COBOL-81 performs binary truncation. With binary truncation, the maximum value a COMP item can contain depends on its storage allocation. If you specify this switch, the maximum value depends on the item's PICTURE character-string.

D.1.1.2 Switches Suppressing Functions — Four switches suppress functions that the compiler normally performs (unless the defaults have been changed by your system manager). Each is prefaced by a minus sign, indicating that you are "turning off" the function.

/–BOU

Stops the compiler from generating code for checking subscripts and indexes. By default, COBOL-81 checks each one at run time against the ranges defined by its data name's OCCURS clause. If any range is exceeded during execution, COBOL-81 issues an error message to that effect. However, if this switch is used to suppress checking, an out-of-range subscript or index does not generate an error message, and the program does not produce valid results.

/–CIS

Stops the compiler from using CIS (Commercial Instruction Set) in the object code it produces. If the system manager set the default to CIS code when COBOL-81 was installed, this switch overrides that default.

/–INF

Stops the compiler from issuing informational diagnostics during the compilation. If you use this switch, only warning and fatal diagnostics appear in the list file, diagnostic file, and diagnostic summary.

/–PER

Stops the compiler from generating code needed for checking PERFORM statement ranges. At run time, COBOL-81 uses this code to determine if your program's PERFORM ranges are nested properly (if nested at all). If COBOL-81 detects improper nesting during execution, it issues an error message to that effect. If you use this switch, however, and the program's PERFORM statements are nested incorrectly, the program does not produce valid results.

Note

Both /–PER and /–BOU can save execution time and decrease the size of the task image.

/–SKL

Stops the compiler from producing the skeleton overlay descriptor language file. This file is normally produced each time the compiler creates an object file.

Note

The SKL file must be produced if you want to use the compiler's /BLD switch or the BLDODL utility.

D.1.1.3 Switches Altering Functions — There are three switches you can use to alter compiler functions:

/FIPS:74

Compilation with this switch produces the following:

- The value 10 replaces file status values 13, 15, and 16.
- The maximum size of intermediate values generated during arithmetic computations is 19 digits, rather than 18 digits.

/KER:xx

Tells the compiler to use the two alphanumeric characters you specify as the PSECT kernel for this program. The only time you need this switch is when your task image uses both subprograms and segmentation; see Part II, Chapter 5, for a detailed explanation.

/TMP:dev

Tells the compiler to store its temporary working files on the device you specify by *dev* during compilation. Since the default device is SY:, this switch is useful if there is limited system disk space available, or if you have a high-speed swapping device, such as a fixed-head disk or electronic memory, available.

D.1.1.4 Examples of Switches — These command lines illustrate the use of various switches:

1.
```
C81 = YEARLY/-INF
```
   Gives you a summary of warning and fatal errors only.

2.
```
C81 , ANNUAL = ANNUAL/MAP/CRF
```
   Creates the list file ANNUAL.LST with offset maps and cross-reference tables.

3.
```
C81 TEST = TEST/TMP:DK2:
```
   Uses DK2: for storing temporary files during compilation.

4.
```
C81 MAIL = MAIL/CRF
```
   Is a meaningless use of the /CRF switch, because no list file has been specified to contain the cross-references. COBOL-81 ignores the switch and proceeds with the compilation.

D.2 Using the BLDODL Utility

This section explains the use of the COBOL-81 BLDODL utility and the optional functions it provides.

For an explanation of the RMS-11 concepts referred to in this section, see the *RMS-11 User's Guide*.

To invoke BLDODL, type:

```
BLDODL RET
```

The RETURN key is optional. If you type it, BLDODL will give you this prompt:

```
BLD>
```

You then enter a command line. If you do not press the RETURN key, you must enter commands on the same line as BLDODL, If you type /HELP instead of a command line, BLDODL displays a help message on your terminal. This message summarizes the format of the command and its switches.

To exit to the system prompt, press CTRL/Z.

D.2.1 BLDODL Command Line and Switches

The format of the command line is:

```
output[/switch]... = input1[/switch]...[,inputx[/switch]...]...
```

where:

output is the file specification you want BLDODL to use for the ODL and CMD files it produces.

input1 is the file specification of the SKL file you are processing. (Remember, the compiler gives the SKL file the same file specification as the one you defined for the OBJ file.)

inputx specifies either another SKL file (created by the compiler), a user library, or an ODL file you have created to describe the overlay structure for the RMS-11 portion of your task. If you specify an ODL file, you must follow it with the /IO:USEROV switch. If you specify a user library, you must follow it with the /ULIB switch. See the *RSX-11M/M-PLUS Task Builder Manual* for more information.

switch is one of the following:

/CLU:RESLIB1:RESLIB2

Allows you to cluster up to two other memory-resident libraries with a COBOL-81 OTS resident library. The RMS resident library, RMSRES.TSK, is one library you can cluster with a COBOL-81 OTS resident library. The following BLDODL command line illustrates how to do this:

```
TEST = TEST/CLU:RMSRES
```

If you use the /CLU switch to cluster with RMS, you cannot use the /IO: switch.

/ULIB

Allows you to include one user library per task image. Append /ULIB to a library name of six or fewer characters.

/DEB

Indicates that you are including the COBOL-81 Symbolic Debugger in your task. You must use this switch in the BLDODL command line if you used the compiler's /DEB switch.

/FMS

Indicates that you are including the FMS library in your task image.

/MAP

Creates a request in your CMD file for a Task Builder memory allocation map (or MAP file).

/MER

Creates an ODL file that is a concatenation of the SKL files in your task. This ODL file contains every line from every SKL file used. If you do not use this switch, BLDODL produces an abbreviated ODL file. This file includes only a one-line reference to each SKL file. An abbreviated ODL file is smaller, but a concatenated ODL file eliminates the need for keeping each SKL stored on disk. If you do not plan to recompile any part of your program, you can use /MER and then delete each SKL file from your directory.

/OBJ:dev:project:programmer

Specifies the location of your OBJ file. Use this switch on an input SKL file when the SKL file and its corresponding OBJ file are in different directories.

/IO:

Specifies how you want the RMS-11 routines included in your task image. There are four choices:

/IO:DECOV is the default if you do not use an /IO: switch. It specifies that the I/O routines are to be overlayable (that is, that they will share memory) in your task. If your program requires RMS-11 support for indexed files, the routines will occupy 9K bytes in the task. Support for sequential I/O occupies only 8K bytes. In your ODL file, BLDODL includes DIGITAL-supplied instructions that specify how the Task Builder must overlay the RMS-11 routines.

/IO:NONOV includes the I/O routines so that they are not overlayable. The amount of memory occupied by the routines depends on the amount of I/O your program performs. Execution speed may be considerably better with this switch than it would be with the default, /IO:DECOV.

/IO:MEMRES specifies that your task will use the memory-resident RMS-11 library, RMSRES. It occupies 16K bytes in your task. See the *RMS-11 User's Guide* for an explanation of RMS-11 resident libraries.

/IO:USEROV indicates a locally written ODL file. Use this switch only if you specify an ODL file as one of the input files in the BLDODL command line. The switch must immediately follow the ODL file specification.

/LRG

Specifies a large overlay structure (12K bytes) for RMS-11 routines. With a large overlay, execution speed will increase, but your task size will be larger.

If you are not overlaying RMS-11 routines (with the /IO:DECOV switch or by accepting the default), and your task does not perform indexed I/O, BLDODL ignores this switch.

/RES

Creates a reference in the CMD file to the shared OTS resident library. The resulting task image will be smaller; it will use the resident library at run time. Use of the resident library will save memory space if several COBOL-81 tasks are executing at the same time.

Do not use the /RES switch if the resident library is not installed on your system.

/DIA

Invokes a dialog from BLDODL that prompts you for each BLDODL option. This switch is available primarily for former PDP-11 COBOL users who are accustomed to this dialogue from the PDP-11 COBOL Merge Utility (documented in the *PDP-11 COBOL User's Guide*).

D.2.2 BLDODL Utility Command Line Defaults

A BLDODL command line with no switches produces default ODL and CMD files. These default files:

- Do not include the Symbolic Debugger
- Do not produce a Task Builder map
- Do not concatenate SKL files (each SKL file is referred to indirectly)
- Use the device and project-programmer number of the output file specified to find each SKL and OBJ
- Use DIGITAL-supplied overlay descriptors for RMS-11 routines
- Use the small RMS-11 overlay structure (9K bytes) for indexed file support if the task requires that support

The examples at the end of this appendix (Section D.5) illustrate the use of BLDODL in the process of producing a task image.

D.3 Task-Building

The RSX-11M/M-PLUS Task Builder produces an executable (or task) image of your program. It builds the image either by processing a direct command line, or by processing the command (CMD) file produced when you use either the BLDODL utility or the compiler's /BLD switch.

This section first explains using the CMD file as input to the Task Builder. Then, it shows you how to build tasks with a direct command line.

D.3.1 Using the CMD File as Input

To build a task using the CMD file, type:

```
TKB @command-file
```

where:

TKB — invokes the Task Builder.

@ — indicates to the Task Builder that the file specified is a command file (that is, it contains Task Builder commands).

command-file — is the name of the CMD file produced by the compiler when you used the /BLD switch or the BLDODL utility.

The Task Builder processes the commands in the file specified and produces an executable image. It uses the same file name as that of the command file, and the file type TSK.

D.3.2 Using a Direct Command Line as Input

The format of the Task Builder command line is:

```
TKB task-file[,map-file[,symbol-table-file]]=object-file(s),library-file/LB,LB:[1,1]RMSLIB/LB
```

where:

TKB — invokes the Task Builder.

task-file — specifies the name of the task image. The default file type is TSK.

object-file(s) — specifies the names of each object file to be included in the task. If you specify more than one file, one must be a main program and the others subprograms. The order in which they are specified is not important. The default file type for an object file is OBJ.

library-file — specifies a system library file needed by the Task Builder to process all COBOL-81 programs. Use one of the following:

LB:[1,1]C81CIS — if your object code contains CIS (/CIS)

LB:[1,1]C81LIB — if your object code does not contain CIS (/–CIS)

LB:[1,1]RMSLIB — is necessary only if your task performs file I/O. This file specification refers to a system library file of RMS-11 routines.

If you specify the RMS-11 library file (LB:[1,1]RMSLIB), the Task Builder builds nonoverlaid RMS-11 routines into your task. This is equivalent to using the BLDODL utility with the /IO:NONOV switch. Your task will use more memory with nonoverlaid I/O routines, but you will achieve the fastest execution speed this way.

D.3.3 Results of the Task Build

A successful task build creates an executable image and returns to the system prompt.

If the task build is unsuccessful, a Task Builder error message is issued to your terminal before the return to the system prompt.

The task build will be unsuccessful if the resulting image is too large. The following message indicates this condition:

```
SEGMENT seg-name HAS ADDR OVERFLOW: ALLOCATION DELETED
```

Seg-name is the name of the object file the Task Builder was processing when the overflow occurred. To recover, you must make more efficient use of memory by overlaying sections of the task. Use the COBOL-81 segmentation facility to overlay your object code (see Part II, Chapter 5), or the BLDODL utility to overlay RMS-11 routines.

If you are unsure of whether your compiler produces CIS or non-CIS code and you task build your object file with the incorrect COBOL-81 OTS library (LB:[1,1]C81CIS/LB or LB:[1,1]C81LIB/LB), the Task Builder will be unable to define certain symbols. You will get the following error message if you have built CIS object code with the non-CIS library (LB:[1,1]C81LIB/LB):

```
n UNDEFINED SYMBOLS SEGMENT seg-name

     $ENCIS
```

You will get the following message if you have task built non-CIS object code with the CIS library (LB:[1,1]C81CIS/LB):

```
n UNDEFINED SYMBOLS SEGMENT seg-name

     $ENLIB
```

Refer to the *RSX-11M/M-PLUS Task Builder Manual* for all other Task Builder errors.

D.4 Executing

Just as in DCL, the command to execute a task image is RUN. The format for executing a task image is:

RUN task-file

Task-file is the name of the executable image created by the Task Builder. The default file type is TSK.

D.5 Examples

To illustrate some of the options COBOL-81 provides, here are some examples of compiling, task building, and executing programs, and of using the BLDODL utility:

1. The following example shows how to compile, task build, and run a simple COBOL program that does not perform file I/O. It uses a direct command line, rather than a CMD file, as input to the Task Builder. All that is required, in addition to the object file created by the compiler, is the Object Time System library (in this case, the non-CIS library):

```
C81 TEST1 = TEST1
TKB TEST1 = TEST1, LB:[1,1]C81LIB/LB
RUN TEST1
```

   Here is another method that produces the same results. It uses the CMD and ODL files (produced as a result of the compiler /BLD switch) as input to the Task Builder:

```
C81 TEST1 = TEST1/BLD
TKB @TEST1
RUN TEST1
```

2. The BLDODL command line shown here requests a Task Builder memory allocation map (or a MAP file):

   ```
   C81 TEST2 = TEST2
   BLDODL TEST2 = TEST2/MAP
   TKB @TEST2
   RUN TEST2
   ```

   The Task Builder processes TEST2.CMD and TEST2.ODL to produce both TEST2.TSK and TEST2.MAP.

3. If you want to use a locally written ODL file (shown here as USRRMS.ODL) to specify the RMS-11 overlay structure in your task, you use a BLDODL command line similar to the following one:

   ```
   C81 TEST3 = TEST3
   BLDODL TEST3 = TEST3, USRRMS.ODL/IO:USEROV
   TKB @TEST3
   RUN TEST3
   ```

4. This example shows how to compile, build, and run a COBOL-81 task that uses subprograms (BLDODL is used to create single CMD and ODL files referring to each of the programs):

   ```
   C81 MAIN = MAIN
   C81 SUB1 = SUB1
   C81 SUB2 = SUB2
   BLDODL TEST4 = MAIN, SUB1, SUB2
   TKB @TEST4
   RUN TEST4
   ```

5. Here are two different ways to include the Symbolic Debugger in your task image:

   ```
   C81 TEST5 = TEST5/DEB/BLD
   TKB @TEST5
   RUN TEST5
   ```

   ```
   C81 TEST5 = TEST5/DEB
   BLDODL TEST5 = TEST5/DEB
   TKB @TEST5
   RUN TEST5
   ```

6. The following example shows how to include FMS support in your task:

   ```
   C81 TEST6 = TEST6
   BLD TEST6 = TEST6/FMS
   TKB @TEST6
   RUN TEST6
   ```

Contents

Page

PART II

Chapter 1 Using the COBOL-81 REFORMAT Utility

1.1 ANSI-to-Terminal Format Conversion 1-1
1.2 Terminal-to-ANSI Format Conversion 1-3
1.3 REFORMAT Error Messages 1-4

Chapter 2 Troubleshooting

2.1 Reading a Program Listing 2-1
2.2 Program Run Errors 2-6
2.2.1 Faulty Data 2-6
2.2.2 Common Logic Errors 2-7
2.2.3 COBOL-81 Symbolic Debugger 2-8

Chapter 3 Debugging Your Program

3.1 Overview of the Debugger 3-1
3.2 Preparing the Program 3-2
3.3 Using the Debugger Commands 3-3
3.3.1 Using the HELP Command 3-5
3.3.2 Using the DISPLAY Command 3-5
3.3.3 Using the MOVE Command 3-6
3.3.4 Using Breakpoints 3-7
3.3.4.1 SET BREAKPOINT 3-7
3.3.4.2 CANCEL BREAKPOINT 3-8
3.3.4.3 SHOW BREAKPOINTS 3-8
3.3.5 Using Synonyms 3-9
3.3.5.1 DEFINE 3-9
3.3.5.2 UNDEFINE 3-10
3.3.5.3 SHOW SYNONYMS 3-10
3.3.6 Using the PROCEED Command 3-10
3.3.7 Interrupting Program Execution 3-11
3.3.8 Using the STOP Command 3-11

Chapter 4 Reducing Your Task Size

4.1 When to Use Task Size Reduction Techniques 4-1
4.2 Reduction Techniques Available 4-1
4.3 Selecting Library Support Routines 4-2
4.3.1 Disk Libraries, Resident Libraries, and Clustered Resident Libraries . . . 4-2
4.3.2 Estimating COBOL-81 OTS Support 4-5
4.3.3 Using Disk Libraries Only 4-5
4.3.4 Using Resident Libraries 4-6

4.4 Using Subprograms with Implicit Overlays 4-6
4.5 Using the COBOL-81 Segmentation Facility 4-7
4.5.1 Programming Considerations. 4-7
4.5.2 Creating a Segmented Task Image 4-8
4.5.2.1 Segmenting a Single-Program Task 4-8
4.5.2.2 Segmenting a Multiple-Program Task 4-9
4.5.3 Reading a Memory Allocation Map. 4-11

Chapter 5 Improving Program Performance

5.1 Introduction 5-1
5.2 Performance Versus Task Image Reduction. 5-1
5.3 Using Compiler Qualifiers to Improve Performance 5-1
5.3.1 Using the /NOCHECK and /CHECK Qualifiers 5-2
5.3.2 Using the /TEMPORARY:dev Qualifier 5-2
5.4 Using BLDODL Switches to Improve Performance 5-2
5.4.1 Using the BLDODL/IO:NONOV Switch. 5-2
5.4.2 Using the BLDODL/LRG Switch 5-2
5.5 Using Terminal Format Source Programs. 5-3
5.6 Data Handling Techniques for Improving Performance 5-3
5.7 Using File Optimization to Improve Performance 5-3
5.8 Using Subprograms 5-3

Chapter 6 Interprogram Communication

6.1 A Multiple COBOL-81 Program Task 6-1
6.1.1 Identifying a COBOL-81 Subprogram 6-2
6.1.2 Compiling Main and Subprograms 6-2
6.1.3 Transferring Execution Control with the CALL Statement. 6-3
6.1.3.1 The CALL Statement 6-3
6.1.3.2 The EXIT PROGRAM Statement. 6-3
6.1.3.3 Sharing Execution Control 6-4
6.1.3.4 Nesting CALL Statements 6-4
6.2 Accessing Another Program's Data Division 6-7
6.2.1 The USING Phrase 6-7
6.2.2 The Linkage Section 6-8
6.2.3 Examples 6-9
6.2.4 COBOL-81 OTS – Error Checking 6-12
6.3 Including Non-COBOL-81 Programs in a Task 6-12
6.3.1 MACRO Programs and COBOL-81 Programs 6-13
6.3.1.1 Calling a MACRO Program from a COBOL-81 Program . . 6-13
6.3.2 Calling a COBOL-81 Program from a MACRO Program 6-14
6.3.3 Using the Argument Address List 6-15

Appendix A Debugger Error Messages

Examples

4-1 Sample Segmented Program 4-12
4-2 Sample Memory Allocation Map 4-13
6-1 Sharing Program Execution Control Between a Main Program and Multiple Subprograms. 6-5
6-2 Redefining a Calling Program's Data Items in the Called Subprogram's Linkage Section 6-10

Figures

2-1 Partial Listing of the Program REPORT. 2-3
4-1 How Use of Libraries Affects Task Size 4-4
4-2 Memory Allocation of a Segmented Program. 4-9
6-1 Sharing Execution Control Between a Main Program and One Subprogram. . . .6-4
6-2 Nesting CALL Statements 6-4
6-3 Sharing Execution Control Between a Main Program and Multiple Subprograms .6-5
6-4 Correspondence of Single Sets of Data 6-7
6-5 Sharing Execution Control and Data Between a Main Program and One Subprogram. 6-9
6-6 Sharing Nested Execution Control and Data Between a Main Program and Multiple Subprograms. 6-10
6-7 Argument Address List General Format 6-15
6-8 Sample Argument Address List 6-16

Tables

3-1 Debugger Commands. 3-2
6-1 Correspondence of Data-Names 6-8

Chapter 1
Using the COBOL-81 REFORMAT Utility

COBOL-81 accepts source programs written in either ANSI (conventional) reference format or DIGITAL terminal format.

- ANSI format results in source programs that are compatible with the reference format of other COBOL compilers. If your program is in ANSI format, you must compile it using the /ANSI_FORMAT qualifier, unless the default was changed by the system manager during installation.

- Terminal format works with text editors on an online keyboard. It is the format expected by the COBOL-81 compiler, unless the default was changed by the system manager during installation. Terminal format eliminates the line-number and identification fields of ANSI format. It saves disk space and decreases compile time.

The *COBOL-81 Language Reference Manual* explains both formats in detail.

COBOL-81 provides the REFORMAT Utility to convert terminal format source programs to ANSI format and vice versa. This chapter shows you how to use REFORMAT to do both types of conversions.

1.1 ANSI-to-Terminal Format Conversion

REFORMAT converts each ANSI-format source line to terminal format by:

- Removing the six-character sequence field in the first six character positions of the ANSI-format line.

- Moving any continuation (-) or comment (* or /) symbols from character position 7 to character position 1.

- Replacing spaces with horizontal tabs immediately to the right of Margin B and every eight character positions thereafter until the end of the line. This occurs only in those source lines not containing a nonnumeric literal.

- Removing the identification entry field in character positions 73 through 80 of the ANSI-format line.
- Removing insignificant trailing spaces before character position 73 of the ANSI-format line.
- Replacing every form feed character with a line containing a slash (/) in character position 1.
- Placing the converted code in positions 1 through the end of the line, thereby creating a terminal-format line.

Because spaces are not converted to tabs in lines containing nonnumeric literals, those lines might be aligned differently from the rest when you use a text editor on the program. However, the list file produced by the compiler will be aligned correctly.

To run REFORMAT:

- On RSTS/E you type:

```
RFM RET
```

- On RSX-11M/M-PLUS you type:

```
MCR RFM RET
```

REFORMAT executes and prompts you with this message:

```
REFORMAT - ANSI-to-terminal conversion mode [ Y / N ]?
```

For an ANSI-to-terminal conversion, type "Y" and press the RETURN key. REFORMAT confirms your choice with this message:

```
REFORMAT - ANSI-to-terminal format selected
```

REFORMAT then asks for input and output file specifications:

```
REFORMAT -       ANSI-format input file spec :
REFORMAT - Terminal-format output file spec :
```

REFORMAT reads the input file and writes a terminal-format output file. After processing the last source line, REFORMAT displays these messages:

```
REFORMAT - n ANSI COBOL source lines converted to terminal format
REFORMAT - ANSI-to-terminal format conversion mode [ Y / N ]?
```

The first message indicates the number of input source lines converted to terminal format; the second message prompts you for conversion of another file. Type CTRL/Z to end execution.

1.2 Terminal-to-ANSI Format Conversion

REFORMAT converts each terminal-format source record to ANSI format by:

- Placing a six-character line number (000010) in the first six character positions of the line and increasing it by 000010 for each subsequent line.
- Moving any continuation (-) or comment (* or /) symbols from character position 1 to character position 7.
- Replacing horizontal tabs with space characters at every eighth character position, starting at character position 5 until the end of the line.
- Moving spaces into remaining character positions after the last character of code and before character position 73.
- Expanding a terminal line with more than 65 characters into two or more ANSI-format lines and right justifying these lines at character position 72.
- Placing either identification characters (if you supply them when you run REFORMAT) or spaces into character positions 73-80.
- Right justifying (at position 72) the first line of a continued nonnumeric literal. This makes sure that the literal remains the same length as it was in the default format.
- Replacing every form feed character with a line containing a slash (/) in position 7 and space characters in positions 8 through 72.
- Placing the converted code in character positions 8 through 73, thereby creating one or more ANSI-format lines.

To run REFORMAT:

- On RSTS/E you type:

```
RFM RET
```

- On RSX-11M/M-PLUS you type:

```
MCR RFM RET
```

REFORMAT prompts you with this message:

```
REFORMAT - ANSI-to-terminal conversion mode [Y / N ]?
```

For a terminal-to-ANSI conversion, type "N" and press the RETURN key. REFORMAT confirms your choice with this message:

```
REFORMAT - Terminal-to-ANSI format selected
```

REFORMAT then asks for input and output file specifications:

```
REFORMAT - Terminal-format input file spec :
REFORMAT -    ANSI-format output file spec :
```

After you enter the file specifications, REFORMAT asks for an identification entry in columns 73 through 80:

```
REFORMAT - Columns 73 to 80:
```

If you want an identification entry, type from one to eight characters. REFORMAT places these characters, left justified, in columns 73 through 80 of each output line. If you do not want an identification entry, type a carriage return.

REFORMAT reads the input file and writes the output file in 80-character ANSI-format lines. After processing the last line, REFORMAT displays these messages:

```
REFORMAT - n Terminal COBOL source lines converted to ANSI format
REFORMAT - ANSI-to-terminal format conversion mode [ Y / N ]?
```

The first message indicates the number of input source code lines converted to ANSI format; the second message prompts you for conversion of another file. Type CTRL/Z to end execution.

1.3 REFORMAT Error Messages

If any of your responses to the prompts are incorrect, REFORMAT displays messages. It replaces the parentheses and the parenthetical text in the following examples with the appropriate format type you specified:

```
REFORMAT - Error in opening (ANSI or terminal) format input file:
REFORMAT -         (ANSI or terminal) format input file spec:
```

REFORMAT could not open the file; either the file is not on the specified device or you typed the file specification incorrectly. The default device is SY:, and the default directory is your current directory.

To recover from this error, examine the input file specification and type a corrected version. To process another file, type a new input file specification. To end execution, type CTRL/Z.

```
REFORMAT - Error in opening (ANSI or terminal) format output file:
REFORMAT -   (ANSI or terminal) format output file spec:
```

REFORMAT could not open the output file. An incorrectly typed file specification causes this error. The default device is SY:, and the default directory is your current directory.

To continue, examine the output file specification and type a corrected version. To end execution, type CTRL/Z.

```
REFORMAT - (ANSI or terminal) format input file is empty
REFORMAT -      (ANSI or terminal) format input file spec:
```

REFORMAT issues this message if it opens an empty file; that is, one that contains no source code. To continue, type a new input file specification. To end execution, type CTRL/Z.

```
REFORMAT - Error in reading (ANSI or terminal) format input file
REFORMAT - Reformatting aborted
REFORMAT - n (ANSI or terminal) COBOL source lines converted to
                (ANSI or terminal) format
REFORMAT - ANSI-to-terminal format conversion mode [ Y / N ]?
```

You will receive these messages if REFORMAT failed to read a source line from the input file. This error ends the conversion process. REFORMAT closes both files and displays the number of converted input lines.

At this point, you can either convert another file or end the session by typing CTRL/Z. Before REFORMAT can completely convert the file that contains the error, you will have to examine the file and make the necessary correction.

```
REFORMAT - Error in writing (ANSI or terminal) format output file
REFORMAT - Reformatting aborted
REFORMAT - n (ANSI or terminal) COBOL source records converted to
                (ANSI or terminal) format
REFORMAT - ANSI-to-terminal format conversion mode [ Y or N ]?
```

REFORMAT failed the attempt to write an output record. It ends execution and closes both files.

To process another file, type a new input file specification and continue the prompting message sequence. To end execution, type CTRL/Z.

Chapter 2
Troubleshooting

This chapter discusses how to find and correct program logic errors. It explains how to read a program listing and discusses techniques that you can use for program debugging.

2.1 Reading a Program Listing

The circled numbers on the program listing REPORT (see Figure 2-1) correspond to the following numbered text explanations:

❶ The program name as declared in PROGRAM-ID.

❷ The date and time of compilation.

❸ The creation date and time of the file specified in 5.

❹ The version of the COBOL-81 compiler.

❺ The source file specification in file-spec format (device:[directory]filename.type).

❻ Source line numbers assigned by the compiler. The COBOL-81 Symbolic Debugger uses these line numbers as location specifications.

❼ Source text. Although a terminal line can contain 200 characters, a source listing line contains a maximum 120 characters.

❽ Identification field. If the source file is in ANSI format, this field contains the identification field (positions 73 through 80).

❾ Error pointer. The caret (^) points to the closest approximation of where the error occurred.

❿ Error message line. This line gives the error severity code, the error message number, and the error message.

⓫ A summary total, by diagnostic level, of compiler-generated diagnostics. Diagnostics can be Informational (I), Warning (W), or Fatal (F).

⓬ COBOL-81 command qualifiers. The first line of qualifiers is the compiler command line; the second group shows the remaining qualifiers and qualifier defaults in effect at compile time.

13 Procedure Division Map. This is a list of procedure-names and their attributes that you get by specifying the /MAP command qualifier.

14 A list of procedure-names.

15 The source line number, where the procedure-name is defined.

16 PSECT. This gives the procedure-name's program section identification.

17 OFFSET. This gives the offset in octal and decimal from the beginning of the PSECT.

18 Lists the procedure-name type (either PARAGRAPH or SECTION).

19 REF indicates whether or not the section or paragraph was referenced in the program.

20 Data Division Map. This lists file names, data items, and their attributes. You get this listing by specifying the /MAP command qualifier.

21 The data item's level number.

22 The name of the data item.

23 The source line number where the data item is defined.

24 PSECT. This identifies the data item's PSECT.

25 OFFSET. This gives the offset in octal and decimal from the beginning of the PSECT.

26 REF indicates whether or not a data item is referenced in the program.

27 The category of data described. Category classifications include:

FILE NAME
GROUP ITEM
ALPHABETIC
ALPHANUMERIC
ALPHANUMERIC EDITED
NUMERIC
NUMERIC EDITED
DECIMAL SIGNED
DECIMAL UNSIGNED

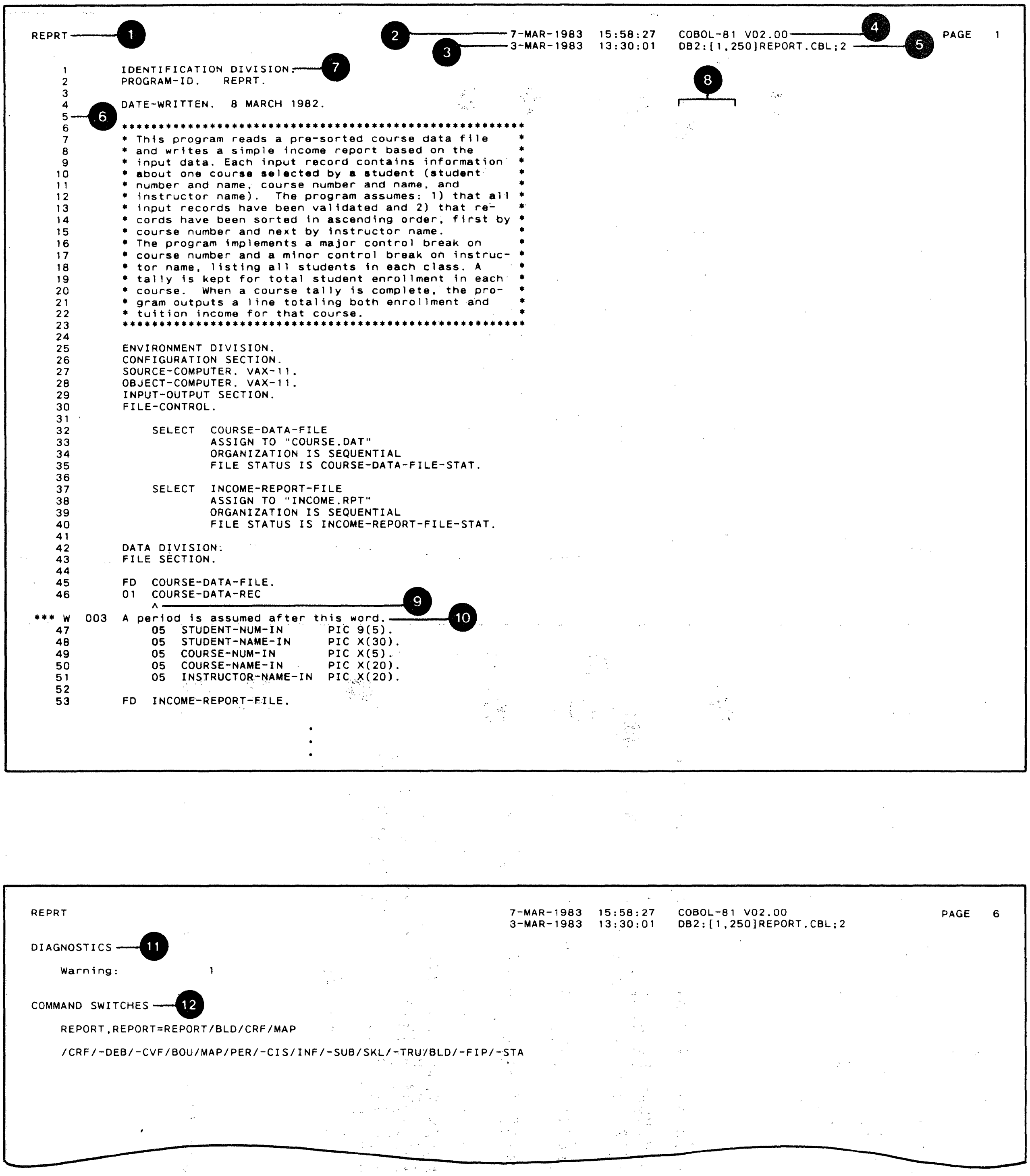
28 Indicates the number of occurrences specified by the OCCURS clause of a data definition entry. This column is blank if the data definition entry does not define a table.

29 The number of bytes allocated to the data item. For numeric values, field size and bytes can be different.

30 Alphabetical Cross Reference of Data Items. This table displays the line numbers for each occurrence of a data item. The letter "D" is suffixed to the line number at which the data item was defined. An asterisk (*) indicates the line numbers of destructive references, such as assignment statements that change the value of a data item.

31 Alphabetical Cross Reference of Procedure Names. This table displays the line numbers at which each procedure-name is referenced. The letter "D" is suffixed to the line number at which the procedure begins.

Figure 2-1: Partial Listing of the Program REPORT

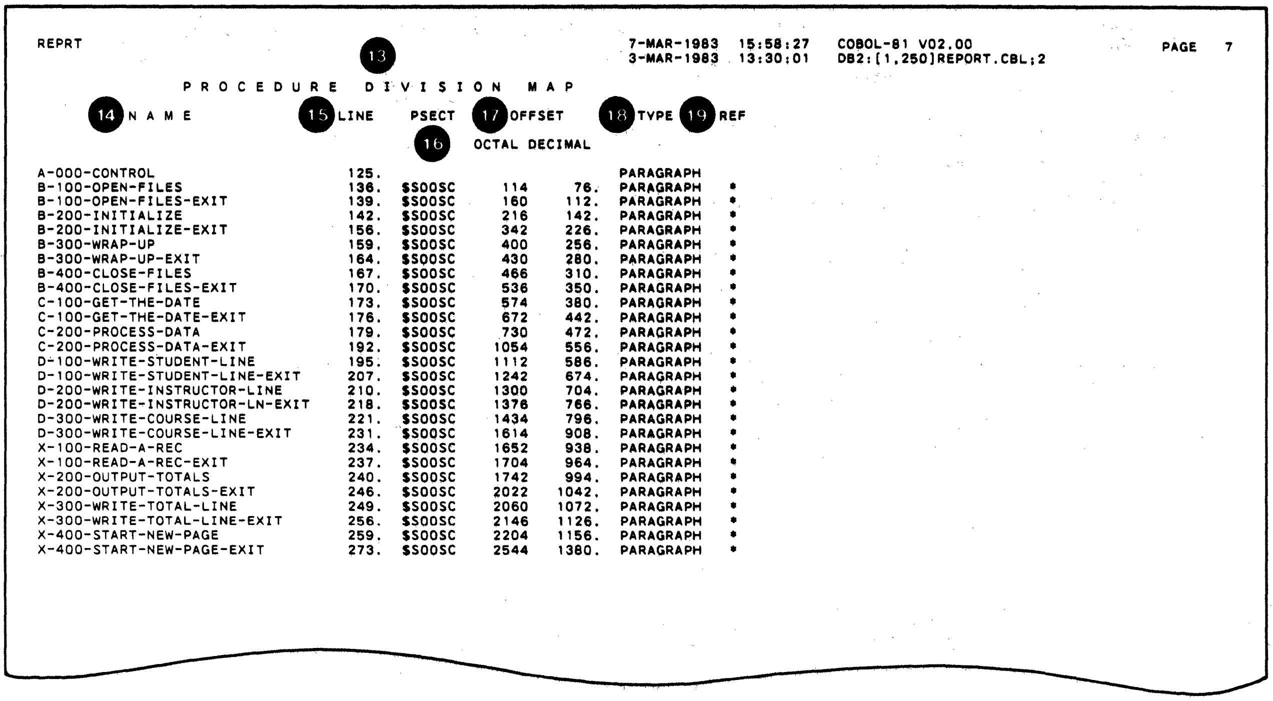


```
REPRT                                         7-MAR-1983  15:58:27   COBOL-81 V02.00                PAGE   1
                                              3-MAR-1983  13:30:01   DB2:[1,250]REPORT.CBL;2

      1        IDENTIFICATION DIVISION.
      2        PROGRAM-ID.   REPRT.
      3
      4        DATE-WRITTEN.  8 MARCH 1982.
      5
      6        ****************************************************
      7        * This program reads a pre-sorted course data file  *
      8        * and writes a simple income report based on the    *
      9        * input data. Each input record contains information *
     10        * about one course selected by a student (student   *
     11        * number and name, course number and name, and      *
     12        * instructor name).  The program assumes: 1) that all *
     13        * input records have been validated and 2) that re-  *
     14        * cords have been sorted in ascending order, first by *
     15        * course number and next by instructor name.        *
     16        * The program implements a major control break on   *
     17        * course number and a minor control break on instruc- *
     18        * tor name, listing all students in each class. A   *
     19        * tally is kept for total student enrollment in each *
     20        * course.  When a course tally is complete, the pro- *
     21        * gram outputs a line totaling both enrollment and  *
     22        * tuition income for that course.                   *
     23        ****************************************************
     24
     25        ENVIRONMENT DIVISION.
     26        CONFIGURATION SECTION.
     27        SOURCE-COMPUTER. VAX-11.
     28        OBJECT-COMPUTER. VAX-11.
     29        INPUT-OUTPUT SECTION.
     30        FILE-CONTROL.
     31
     32            SELECT  COURSE-DATA-FILE
     33                    ASSIGN TO "COURSE.DAT"
     34                    ORGANIZATION IS SEQUENTIAL
     35                    FILE STATUS IS COURSE-DATA-FILE-STAT.
     36
     37            SELECT  INCOME-REPORT-FILE
     38                    ASSIGN TO "INCOME.RPT"
     39                    ORGANIZATION IS SEQUENTIAL
     40                    FILE STATUS IS INCOME-REPORT-FILE-STAT.
     41
     42        DATA DIVISION.
     43        FILE SECTION.
     44
     45        FD  COURSE-DATA-FILE.
     46        01  COURSE-DATA-REC
                   ^
*** W   003  A period is assumed after this word.
     47            05  STUDENT-NUM-IN       PIC 9(5).
     48            05  STUDENT-NAME-IN      PIC X(30).
     49            05  COURSE-NUM-IN        PIC X(5).
     50            05  COURSE-NAME-IN       PIC X(20).
     51            05  INSTRUCTOR-NAME-IN   PIC X(20).
     52
     53        FD  INCOME-REPORT-FILE.
                              .
                              .
                              .
```

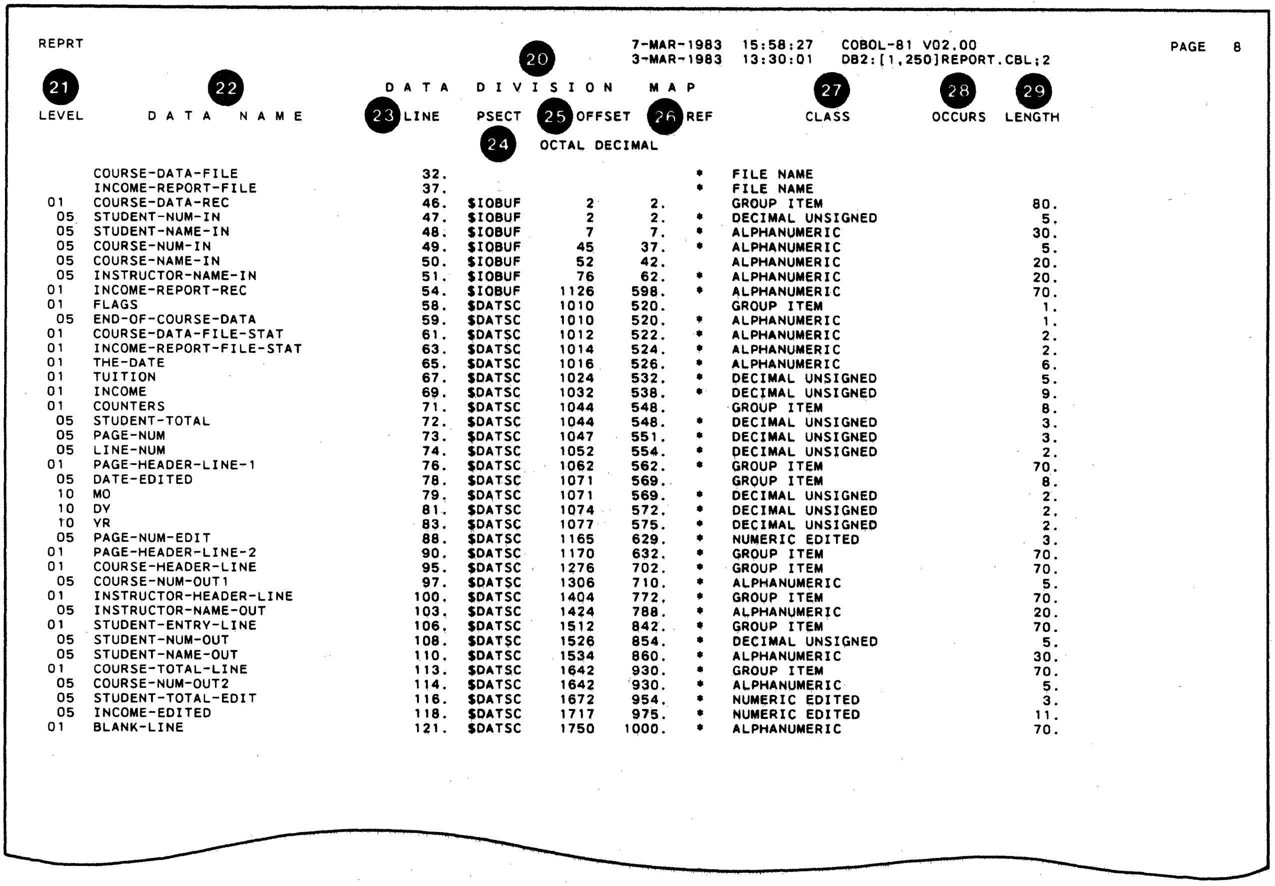
```
REPRT                                         7-MAR-1983  15:58:27   COBOL-81 V02.00                PAGE   6
                                              3-MAR-1983  13:30:01   DB2:[1,250]REPORT.CBL;2

DIAGNOSTICS

    Warning:          1

COMMAND SWITCHES

    REPORT,REPORT=REPORT/BLD/CRF/MAP

    /CRF/-DEB/-CVF/BOU/MAP/PER/-CIS/INF/-SUB/SKL/-TRU/BLD/-FIP/-STA
```

(continued on next page)

Figure 2-1: Partial Listing of the Program REPORT (Cont.)


13
14
15
16
17
18
19

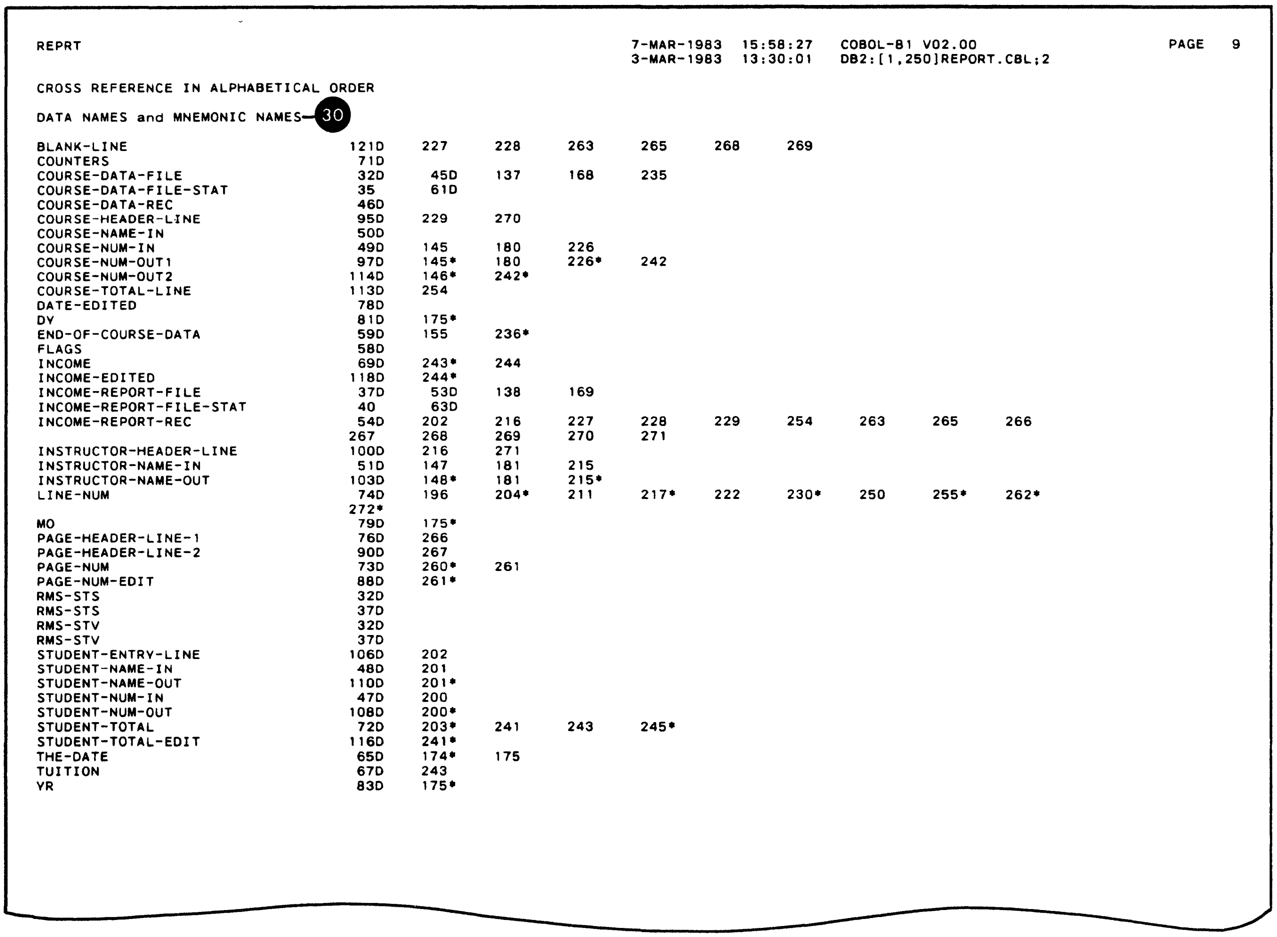
```
REPRT                                                    7-MAR-1983  15:58:27   COBOL-81 V02.00           PAGE  7
                                                         3-MAR-1983  13:30:01   DB2:[1,250]REPORT.CBL;2

                  P R O C E D U R E   D I V I S I O N   M A P

    N A M E                       LINE    PSECT     OFFSET      TYPE     REF
                                                 OCTAL DECIMAL

A-000-CONTROL                    125.                          PARAGRAPH
B-100-OPEN-FILES                 136.   $SOOSC     114     76.  PARAGRAPH   *
B-100-OPEN-FILES-EXIT            139.   $SOOSC     160    112.  PARAGRAPH   *
B-200-INITIALIZE                 142.   $SOOSC     216    142.  PARAGRAPH   *
B-200-INITIALIZE-EXIT            156.   $SOOSC     342    226.  PARAGRAPH   *
B-300-WRAP-UP                    159.   $SOOSC     400    256.  PARAGRAPH   *
B-300-WRAP-UP-EXIT               164.   $SOOSC     430    280.  PARAGRAPH   *
B-400-CLOSE-FILES                167.   $SOOSC     466    310.  PARAGRAPH   *
B-400-CLOSE-FILES-EXIT           170.   $SOOSC     536    350.  PARAGRAPH   *
C-100-GET-THE-DATE               173.   $SOOSC     574    380.  PARAGRAPH   *
C-100-GET-THE-DATE-EXIT          176.   $SOOSC     672    442.  PARAGRAPH   *
C-200-PROCESS-DATA               179.   $SOOSC     730    472.  PARAGRAPH   *
C-200-PROCESS-DATA-EXIT          192.   $SOOSC    1054    556.  PARAGRAPH   *
D-100-WRITE-STUDENT-LINE         195.   $SOOSC    1112    586.  PARAGRAPH   *
D-100-WRITE-STUDENT-LINE-EXIT    207.   $SOOSC    1242    674.  PARAGRAPH   *
D-200-WRITE-INSTRUCTOR-LINE      210.   $SOOSC    1300    704.  PARAGRAPH   *
D-200-WRITE-INSTRUCTOR-LN-EXIT   218.   $SOOSC    1376    766.  PARAGRAPH   *
D-300-WRITE-COURSE-LINE          221.   $SOOSC    1434    796.  PARAGRAPH   *
D-300-WRITE-COURSE-LINE-EXIT     231.   $SOOSC    1614    908.  PARAGRAPH   *
X-100-READ-A-REC                 234.   $SOOSC    1652    938.  PARAGRAPH   *
X-100-READ-A-REC-EXIT            237.   $SOOSC    1704    964.  PARAGRAPH   *
X-200-OUTPUT-TOTALS              240.   $SOOSC    1742    994.  PARAGRAPH   *
X-200-OUTPUT-TOTALS-EXIT         246.   $SOOSC    2022   1042.  PARAGRAPH   *
X-300-WRITE-TOTAL-LINE           249.   $SOOSC    2060   1072.  PARAGRAPH   *
X-300-WRITE-TOTAL-LINE-EXIT      256.   $SOOSC    2146   1126.  PARAGRAPH   *
X-400-START-NEW-PAGE             259.   $SOOSC    2204   1156.  PARAGRAPH   *
X-400-START-NEW-PAGE-EXIT        273.   $SOOSC    2544   1380.  PARAGRAPH   *
```


20
21
22
23
24
25
26
27
28
29

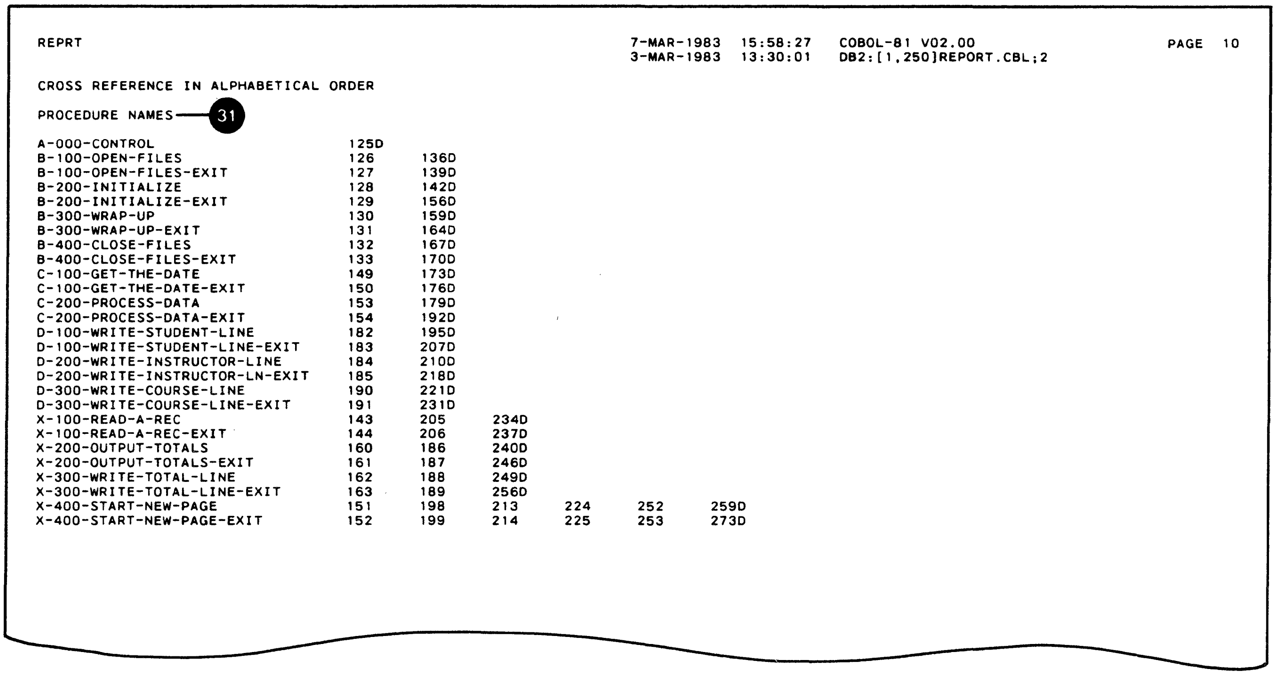
```
REPRT                                                       7-MAR-1983  15:58:27   COBOL-81 V02.00          PAGE  8
                                                            3-MAR-1983  13:30:01   DB2:[1,250]REPORT.CBL;2

                                        D A T A   D I V I S I O N   M A P
LEVEL      D A T A   N A M E          LINE    PSECT    OFFSET    REF        CLASS             OCCURS  LENGTH
                                                     OCTAL DECIMAL

      COURSE-DATA-FILE                 32.                         *   FILE NAME
      INCOME-REPORT-FILE               37.                         *   FILE NAME
 01   COURSE-DATA-REC                  46.   $IOBUF      2      2.     GROUP ITEM                       80.
  05  STUDENT-NUM-IN                   47.   $IOBUF      2      2.  *  DECIMAL UNSIGNED                  5.
  05  STUDENT-NAME-IN                  48.   $IOBUF      7      7.  *  ALPHANUMERIC                     30.
  05  COURSE-NUM-IN                    49.   $IOBUF     45     37.  *  ALPHANUMERIC                      5.
  05  COURSE-NAME-IN                   50.   $IOBUF     52     42.     ALPHANUMERIC                     20.
  05  INSTRUCTOR-NAME-IN               51.   $IOBUF     76     62.  *  ALPHANUMERIC                     20.
 01   INCOME-REPORT-REC                54.   $IOBUF   1126    598.  *  ALPHANUMERIC                     70.
 01   FLAGS                            58.   $DATSC   1010    520.     GROUP ITEM                        1.
  05  END-OF-COURSE-DATA               59.   $DATSC   1010    520.  *  ALPHANUMERIC                      1.
 01   COURSE-DATA-FILE-STAT            61.   $DATSC   1012    522.  *  ALPHANUMERIC                      2.
 01   INCOME-REPORT-FILE-STAT          63.   $DATSC   1014    524.  *  ALPHANUMERIC                      2.
 01   THE-DATE                         65.   $DATSC   1016    526.  *  ALPHANUMERIC                      6.
 01   TUITION                          67.   $DATSC   1024    532.  *  DECIMAL UNSIGNED                  5.
 01   INCOME                           69.   $DATSC   1032    538.  *  DECIMAL UNSIGNED                  9.
 01   COUNTERS                         71.   $DATSC   1044    548.     GROUP ITEM                        8.
  05  STUDENT-TOTAL                    72.   $DATSC   1044    548.  *  DECIMAL UNSIGNED                  3.
  05  PAGE-NUM                         73.   $DATSC   1047    551.  *  DECIMAL UNSIGNED                  3.
  05  LINE-NUM                         74.   $DATSC   1052    554.  *  DECIMAL UNSIGNED                  2.
 01   PAGE-HEADER-LINE-1               76.   $DATSC   1062    562.  *  GROUP ITEM                       70.
  05  DATE-EDITED                      78.   $DATSC   1071    569.     GROUP ITEM                        8.
  10  MO                               79.   $DATSC   1071    569.  *  DECIMAL UNSIGNED                  2.
  10  DY                               81.   $DATSC   1074    572.  *  DECIMAL UNSIGNED                  2.
  10  YR                               83.   $DATSC   1077    575.  *  DECIMAL UNSIGNED                  2.
  05  PAGE-NUM-EDIT                    88.   $DATSC   1165    629.  *  NUMERIC EDITED                    3.
 01   PAGE-HEADER-LINE-2               90.   $DATSC   1170    632.  *  GROUP ITEM                       70.
 01   COURSE-HEADER-LINE               95.   $DATSC   1276    702.  *  GROUP ITEM                       70.
  05  COURSE-NUM-OUT1                  97.   $DATSC   1306    710.  *  ALPHANUMERIC                      5.
 01   INSTRUCTOR-HEADER-LINE          100.   $DATSC   1404    772.  *  GROUP ITEM                       70.
  05  INSTRUCTOR-NAME-OUT             103.   $DATSC   1424    788.  *  ALPHANUMERIC                     20.
 01   STUDENT-ENTRY-LINE              106.   $DATSC   1512    842.  *  GROUP ITEM                       70.
  05  STUDENT-NUM-OUT                 108.   $DATSC   1526    854.  *  DECIMAL UNSIGNED                  5.
  05  STUDENT-NAME-OUT                110.   $DATSC   1534    860.  *  ALPHANUMERIC                     30.
 01   COURSE-TOTAL-LINE               113.   $DATSC   1642    930.  *  GROUP ITEM                       70.
  05  COURSE-NUM-OUT2                 114.   $DATSC   1642    930.  *  ALPHANUMERIC                      5.
  05  STUDENT-TOTAL-EDIT              116.   $DATSC   1672    954.  *  NUMERIC EDITED                    3.
  05  INCOME-EDITED                   118.   $DATSC   1717    975.  *  NUMERIC EDITED                   11.
 01   BLANK-LINE                      121.   $DATSC   1750   1000.  *  ALPHANUMERIC                     70.
```

(continued on next page)

Figure 2-1: Partial Listing of the Program REPORT (Cont.)



```
REPRT                                                      7-MAR-1983  15:58:27   COBOL-81 V02.00                PAGE   9
                                                           3-MAR-1983  13:30:01   DB2:[1,250]REPORT.CBL;2

CROSS REFERENCE IN ALPHABETICAL ORDER

DATA NAMES and MNEMONIC NAMES

BLANK-LINE                          121D    227     228     263     265     268     269
COUNTERS                             71D
COURSE-DATA-FILE                     32D     45D    137     168     235
COURSE-DATA-FILE-STAT                35      61D
COURSE-DATA-REC                      46D
COURSE-HEADER-LINE                   95D    229     270
COURSE-NAME-IN                       50D
COURSE-NUM-IN                        49D    145     180     226
COURSE-NUM-OUT1                      97D    145*    180     226*    242
COURSE-NUM-OUT2                     114D    146*    242*
COURSE-TOTAL-LINE                   113D    254
DATE-EDITED                          78D
DY                                   81D    175*
END-OF-COURSE-DATA                   59D    155     236*
FLAGS                                58D
INCOME                               69D    243*    244
INCOME-EDITED                       118D    244*
INCOME-REPORT-FILE                   37D     53D    138     169
INCOME-REPORT-FILE-STAT              40      63D
INCOME-REPORT-REC                    54D    202     216     227     228     229     254     263     265     266
                                    267     268     269     270     271
INSTRUCTOR-HEADER-LINE              100D    216     271
INSTRUCTOR-NAME-IN                   51D    147     181     215
INSTRUCTOR-NAME-OUT                 103D    148*    181     215*
LINE-NUM                             74D    196     204*    211     217*    222     230*    250     255*    262*
                                    272*
MO                                   79D    175*
PAGE-HEADER-LINE-1                   76D    266
PAGE-HEADER-LINE-2                   90D    267
PAGE-NUM                             73D    260*    261
PAGE-NUM-EDIT                        88D    261*
RMS-STS                              32D
RMS-STS                              37D
RMS-STV                              32D
RMS-STV                              37D
STUDENT-ENTRY-LINE                  106D    202
STUDENT-NAME-IN                      48D    201
STUDENT-NAME-OUT                    110D    201*
STUDENT-NUM-IN                       47D    200
STUDENT-NUM-OUT                     108D    200*
STUDENT-TOTAL                        72D    203*    241     243     245*
STUDENT-TOTAL-EDIT                  116D    241*
THE-DATE                             65D    174*    175
TUITION                              67D    243
YR                                   83D    175*
```



```
REPRT                                                      7-MAR-1983  15:58:27   COBOL-81 V02.00                PAGE  10
                                                           3-MAR-1983  13:30:01   DB2:[1,250]REPORT.CBL;2

CROSS REFERENCE IN ALPHABETICAL ORDER

PROCEDURE NAMES

A-000-CONTROL                       125D
B-100-OPEN-FILES                    126     136D
B-100-OPEN-FILES-EXIT               127     139D
B-200-INITIALIZE                    128     142D
B-200-INITIALIZE-EXIT               129     156D
B-300-WRAP-UP                       130     159D
B-300-WRAP-UP-EXIT                  131     164D
B-400-CLOSE-FILES                   132     167D
B-400-CLOSE-FILES-EXIT              133     170D
C-100-GET-THE-DATE                  149     173D
C-100-GET-THE-DATE-EXIT             150     176D
C-200-PROCESS-DATA                  153     179D
C-200-PROCESS-DATA-EXIT             154     192D
D-100-WRITE-STUDENT-LINE            182     195D
D-100-WRITE-STUDENT-LINE-EXIT       183     207D
D-200-WRITE-INSTRUCTOR-LINE         184     210D
D-200-WRITE-INSTRUCTOR-LN-EXIT      185     218D
D-300-WRITE-COURSE-LINE             190     221D
D-300-WRITE-COURSE-LINE-EXIT        191     231D
X-100-READ-A-REC                    143     205     234D
X-100-READ-A-REC-EXIT               144     206     237D
X-200-OUTPUT-TOTALS                 160     186     240D
X-200-OUTPUT-TOTALS-EXIT            161     187     246D
X-300-WRITE-TOTAL-LINE              162     188     249D
X-300-WRITE-TOTAL-LINE-EXIT         163     189     256D
X-400-START-NEW-PAGE                151     198     213     224     252     259D
X-400-START-NEW-PAGE-EXIT           152     199     214     225     253     273D
```

C81ART-10003-180

2.2 Program Run Errors

If your program terminates abnormally, you receive one of the COBOL-81 run-time error messages to identify the problem. Appendix C of Part I lists and describes these error messages.

However, your program can run to completion and still not yield the results you expect. These incorrect or undesirable program results are usually caused by data errors or program logic errors. You can resolve most of these errors by "desk-checking" your program and by using the COBOL-81 Symbolic Debugger.

2.2.1 Faulty Data

Faulty or incorrectly defined data can often produce incorrect results. Data errors can sometimes be attributed to:

- Incorrect picture size. If the picture size of a receiving data item is too small, data may be truncated.
- Incorrect file definition. The block size you specify when accessing a file should be the same block size you used when creating the file.
- Incorrect record field position. The record field positions that you specify in your program might not agree with a file's record field positions. For example, a file could have this record description:

```
01  PAY-RECORD.
    03  P-NUMBER        PIC X(5).
    03  P-WEEKLY-AMT    PIC S9(5)V99  COMP-3.
    03  P-MONTHLY-AMT   PIC S9(5)V99  COMP-3.
    03  P-YEARLY-AMT    PIC S9(5)V99  COMP-3.
        .
        .
        .
```

Incorrectly positioning these fields can produce faulty data.

An attempt to read the file according to the following input record definition would place monthly data in P-YEARLY-AMT and annual data in P-MONTHLY-AMT.

```
01  PAY-RECORD.
    03  P-NUMBER        PIC X(5).
    03  P-WEEKLY-AMT    PIC S9(5)V99  COMP-3.
    03  P-YEARLY-AMT    PIC S9(5)V99  COMP-3.
    03  P-MONTHLY-AMT   PIC S9(5)V99  COMP-3.
        .
        .
        .
PROCEDURE DIVISION.
ADD-TOTALS.
    ADD P-MONTHLY-AMT TO TOTAL-MONTHLY-AMT.
        .
        .
        .
```

You can minimize file definition and record field position errors by writing frequently accessed file and record descriptions to a library file and then using the COPY statement in programs that access those files.

Your choice of test data can minimize faulty data problems. Rather than using "live" or ideal data, use test files that include data extremes. For example, test data for an update program should contain tests for duplicate adds, a delete to a nonexistent master record, multiple change records, and so forth. Give particular attention to the first and last records read into the program. Many errors occur at these key points.

Determining when a program produces incorrect results can often help your debugging effort. You can do this by maintaining audit counts (such as total master in = nnn, total transactions in = nnn, total deletions = nnn, total master out = nnn) and displaying the audit counts when the program ends.

2.2.2 Common Logic Errors

When checking your program for logic errors, first examine your program for some of the more obvious bugs, such as the following:

1. Hidden periods. Periods inadvertently placed in a statement usually produce unexpected results. For example:

```
050-DO-WEEKLY-TOTALS.
    IF W-CODE = "W"
       PERFORM 100-WEEKLY-SUMMARY
       ADD WEEKLY-AMT TO WEEKLY-TOTALS.
       GO TO 000-READ-A-MASTER.
    WRITE NEW-MASTER-REC.
```

   The period at the end of ADD WEEKLY-AMT TO WEEKLY-TOTALS changes the logic of the statement by transforming GO TO 000-READ-A-MASTER from a conditional to an unconditional GO TO. As a result, the statement following the GO TO will never be executed.

2. Testing for equality, rather than inequality. Executing a procedure until a test condition is met can cause errors:

```
PERFORM ABC-ROUTINE UNTIL A-COUNTER = 10
```

   If, during execution, the program increments A-COUNTER by an integer other than 1 (1.5, for example), A-COUNTER might never equal 10, causing an infinite loop in ABC-ROUTINE. You can prevent this type of error by changing the statement to:

```
PERFORM ABC-ROUTINE UNTIL A-COUNTER > 9
```

3. Combining two negative test conditions with an OR. The intent of the following statement is to execute GO TO 200-PRINT-REPORT when TEST-FIELD contains any character except "A" or "B". However, the GO TO always executes because the logical equivalent (IF TEST-FIELD NOT = ("A" AND "B")) for the stated test condition can never be true. A single character input (TEST-FIELD) cannot be equal to two characters ("A" and "B") at the same time.

```
IF TEST-FIELD NOT = "A" OR NOT = "B"
   GO TO 200-PRINT-REPORT.
   .
   .
   .
```

You can correct this logic error by changing the statement to:

```
IF TEST-FIELD NOT = "A" AND NOT = "B"
   GO TO 200-PRINT-REPORT.
   .
   .
   .
```

2.2.3 COBOL-81 Symbolic Debugger

The COBOL-81 Symbolic Debugger lets you debug a COBOL program at run time. With the Debugger, you can interactively examine and change the contents of data fields and control the order of statement execution. For information on how to use the Debugger, see Chapter 3.

Chapter 3
Debugging your Program

The COBOL-81 Symbolic Debugger helps you debug programs written for the COBOL-81 compiler. The Debugger lets you control and monitor your program as it runs by referring to the source version rather than the object code produced by the compiler.

This chapter shows you how to prepare your program for using the Debugger and how to use each Debugger command. It begins with an overview of the functions the Debugger provides. Appendix A lists the Debugger error messages.

3.1 Overview of the Debugger

The Debugger gives you control over your program's execution by letting you specify breakpoints, which are positions in your program where execution temporarily stops.

At these breakpoints, you can examine the contents of data items and, if necessary, assign new values to them.

The Debugger also lets you associate synonyms with data-names and positions in your program. Once you have defined a synonym, you can use it in any Debugger command rather than typing the actual name or position. If you refer to particular data items or positions often during a debugging session, defining synonyms for them saves time.

Table 3-1 shows the available Debugger commands and the functions they provide. The letters underlined in each command indicate that command's abbreviation. You can use an abbreviation in place of the full command at any time.

Table 3-1: Debugger Commands

Command	Description
SET BREAKPOINT	Specifies a point at which program execution will be interrupted.
CANCEL BREAKPOINT	Removes a breakpoint.
SHOW BREAKPOINTS	Displays information about the breakpoints currently set.
DISPLAY	Displays the contents of data items on the terminal.
MOVE	Changes the contents of data items.
DEFINE	Associates a synonym with a data-name or a position.
UNDEFINE	Deletes a synonym.
SHOW SYNONYMS	Displays information about the synonyms currently in use.
PROCEED	Begins execution or continues execution after a breakpoint.
STOP	Stops execution.
HELP	Displays information about a Debugger command or topic.

3.2 Preparing the Program

To include the Debugger in your program, compile it using the /DEBUG qualifier (see Part I, Chapter 3.) For example:

```
COBOL PROGRM/DEBUG
```

If the program calls subprograms, each subprogram must also be compiled with the /DEBUG qualifier.

After compiling the program(s), use the LINK/C81 command with the /DEBUG qualifier.

When you include the Debugger in your task, the Task Builder creates a "symbols file." This file has the same file name as your task image and a file type of STB. It contains information the Debugger needs to know about your program's data-names, procedure-names, and line numbers. You must not delete this file from your directory if you want to use the Debugger. You can delete it, however, once you have finished debugging the program.

The Debugger occupies under 1K words to 6K words in your task image if your program performs file I/O. PDP-11 Record Management Services (RMS-11) routines required by the Debugger for support are already part of your task. In this case, if you do not include a COBOL-81 resident library, the size of the Debugger depends on how much of the COBOL-81 OTS your program uses. The more OTS your program uses, the less address space the debugger will require. If you do include a COBOL-81 resident library, the Debugger occupies the full 6K words in your task image.

The Debugger can occupy up to 10K words in your task image if your program does not perform file I/O because the RMS-11 routines required by the Debugger are not already part of your task image. The Debugger must therefore add these routines to your task image, increasing your task size by an additional 4K words.

If the addition of the Debugger makes your task too large to fit in memory, the Task Builder issues this message:

```
SEGMENT seg-name HAS ADDR OVERFLOW:  ALLOCATION DELETED
```

To correct this problem, you have to make more sections of your task overlayable. See Chapter 4 for information on how to use the COBOL-81 segmentation facility to overlay your object code, and on how to use the BLDODL utility to overlay RMS-11 routines if you have not already done so.

To begin the debugging session, type RUN followed by the name of your task image, as you normally would to execute your program. For example:

```
RUN PROGRM
```

The Debugger then initializes its internal tables. The initialization can take several minutes if your program is large. Once it is finished, the Debugger gives you this prompt:

```
CDB>
```

You also see this prompt whenever the Debugger assumes control after a breakpoint.

In response to the prompt, you can type any Debugger command. The commands are explained in detail in the next section.

A Debugger command can continue over several lines, but it cannot exceed 200 characters. To continue a Debugger command line, type a hyphen (-) at the end of the line to be continued. The Debugger gives you a shortened version of its prompt, as this example shows:

```
CDB>MOVE "This is an exam-
>ple of line continuation" TO ITEMA
```

The Debugger considers the character immediately preceding the hyphen and the character immediately following the shortened prompt to be contiguous. The previous command lines are equivalent to this single line:

```
CDB>MOVE "This is an example of line continuation" TO ITEMA
```

3.3 Using the Debugger Commands

This section shows you the syntax of each Debugger command, along with examples illustrating its use.

The format of each command's syntax uses the following conventions (which are the same as those used in the *COBOL-81 Language Reference Manual*):

- Braces, { }, enclose lists from which you must choose one element.
- Brackets, [ ], enclose optional elements.
- Uppercase words and letters mean that you type the word or letter as shown. The letters underlined are those needed to uniquely define the command to the Debugger. That is, you can use the underlined letters as abbreviations.
- Lowercase words mean that you substitute a word or value of your choice.

Some commands refer to "position," which you must supply using this format:

$$[\ \text{program-name}\ \backslash\ ]\ \begin{Bmatrix} [\ \underline{\text{LINE}}\ ]\ \text{line-number} \\ [\ \underline{\text{PARA}}\text{GRAPH}\ ]\ \text{paragraph-name}\ \left[\begin{Bmatrix}\underline{\text{IN}}\\ \underline{\text{OF}}\end{Bmatrix}\ \text{section-name-2}\right] \\ [\ \underline{\text{SECTION}}\ ]\ \text{section-name-1}\end{Bmatrix}$$

where:

program-name\ — is necessary only if you are referring to a position in a program other than the one currently executing. For example, if the Debugger stops at a breakpoint you set in SUB1, and you want to refer to line 112 of MAIN, you must type:

```
MAIN\LINE 112
```

line-number — is one of the line numbers the compiler assigned to your source code. Your LST file contains these numbers.

paragraph-name
section-name-1 — refer to paragraphs or sections you defined in your program's Procedure Division.

section-name-2 — qualifies your reference to paragraph-name (when needed to make this reference unique). Section-name-2 refers to the Procedure Division section to which paragraph-name is subordinate.

As the syntax for position shows, you do not have to type the words LINE, PARAGRAPH, or SECTION. However, if a section-name or paragraph-name in your program is numeric, you must specify SECTION or PARA to distinguish it from a line number.

The data-names you use in Debugger commands must also be unique, and they can be qualified or subscripted. See Part III, Chapter 3, Table Handling, for further information on subscripting and qualifying. Use this format when specifying a data-name:

$$[\ \text{program-name}\ \backslash\ ]\ \text{data-name-1}\ \left[\begin{Bmatrix}\underline{\text{IN}}\\ \underline{\text{OF}}\end{Bmatrix}\ \text{data-name-2}\right]\ \ldots\ \left[\ (\ \text{literal...}\ )\ \right]$$

where:

program-name\ — must be specified only if the data-name is defined in a program other than the one currently executing. For example, if you are stopped at a breakpoint in MAIN and you want to refer to ITEMA in SUB1, type:

```
SUB1\ITEMA
```

data-name-1 — refers to a data item in your program. If the item is a table element, you must specify a subscript value.

data-name-2 — qualifies your reference to data-name-1 (when needed to make this reference unique). Data-name-2 refers to the data item to which data-name-1 is subordinate.

3.3.1 Using the HELP Command

To get information about a Debugger command or topic, use the HELP command. The format of this command is:

```
HELP  [ topic-word ]
```

If you do not specify a topic-word, the Debugger will give you a list of topics for which information is available. You can then use any item in the list as a topic-word.

Examples

```
CDB>HELP
```

Displays a list of Debugger commands and topics.

```
CDB>H DATA-NAME
```

Displays information about data-names.

3.3.2 Using the DISPLAY Command

Use the DISPLAY command to display the contents of data items on your terminal. The format of this command is:

```
DISPLAY  [ data-name  [ BYTE  ] ]
                      [ ASCII ]
```

This is similar to the COBOL-81 DISPLAY statement, except that it edits numeric data into SIGN TRAILING with decimal point.

If data-name is subscripted, you can specify either a single subscripted item or a range of subscripted items. For example:

```
CDB>DISPLAY ITEMB(3, 6:8)
```

Displays the contents of three data items: ITEMB(3,6), ITEMB(3,7), and ITEMB(3,8).

As the syntax shows, you do not have to specify a data-name. Whenever you specify a data-name in a DISPLAY or MOVE command, that data-name becomes the current data-name. The Debugger keeps track of this current data-name and uses it in each subsequent DISPLAY and MOVE command until you specify another one. To avoid ambiguity, however, you cannot specify the BYTE or ASCII option unless you also specify a data-name.

The BYTE and ASCII options let you override the format of the data item as specified in the COBOL-81 source program. You can use them, for example, to see the exact values of nonprinting characters in an alphanumeric variable. BYTE displays the octal value of each byte of the item. ASCII outputs the item as a series of ASCII characters, with any nonprinting characters in the item represented by the backslash (\).

Examples

```
CDB> DISPLAY SALES-TOTAL
```

Displays the contents of the item SALES-TOTAL.

```
CDB> D MY-ARRAY (5)
```

Displays the contents of the item MY-ARRAY(5).

```
CDB> DIS ITEM-NAME BYTE
```

Displays the octal value of each byte in the item ITEM-NAME.

```
CDB> DISPLAY SALE-TABLE(1, 2:20)
```

Displays the contents of SALE-TABLE(1,2), SALE-TABLE(1,3), ...,SALE-TABLE(1,20).

```
CDB> DISPLAY MATRIX-ONE(1:2, 1:2, 1:2)
```

Displays the contents of the following:

```
MATRIX-ONE(1,1,1)
MATRIX-ONE(1,1,2)
MATRIX-ONE(1,2,1)
MATRIX-ONE(1,2,2)
MATRIX-ONE(2,1,1)
MATRIX-ONE(2,1,2)
MATRIX-ONE(2,2,1)
MATRIX-ONE(2,2,2)
```

```
CDB>  DIS PART1 IN TOOL63
```

Displays the contents of the PART1 subordinate to TOOL63.

3.3.3 Using the MOVE Command

Use the MOVE command to change the value of a COBOL-81 data item. The format of this command is:

```
MOVE literal [ [ TO ] data-name ]
```

This command simulates a COBOL-81 MOVE statement.

The source of the move must be either a numeric or nonnumeric literal.

As the syntax shows, you do not have to specify a destination (that is, a data-name). Whenever you specify a data-name in a DISPLAY or MOVE command, that data-name becomes the current data-name. The Debugger keeps track of this current data-name and uses it in each subsequent DISPLAY and MOVE command until you specify another data-name.

All moves behave as though the MOVE statement had appeared in the COBOL-81 source program.

Examples

```
CDB>MOVE -100.5 TO TAX-RELIEF
```

Moves the numeric value –100.5 to the item TAX-RELIEF.

```
CDB>MOVE "JOHN BULL" EMPLOYEE-NAME
```

Moves the character string JOHN BULL to the item EMPLOYEE-NAME.

```
CDB>M 0
```

Moves the value 0 to the data item you specified most recently.

3.3.4 Using Breakpoints

Three Debugger commands apply to breakpoints:

- SET BREAKPOINT inserts a breakpoint in your program.
- CANCEL BREAKPOINT removes one or all breakpoints.
- SHOW BREAKPOINTS displays information about each breakpoint you have currently set.

3.3.4.1 SET BREAKPOINT — The format of the SET BREAKPOINT command is:

```
                                                  ┌            ┌ integer ┐ ┐
SET BREAKPOINT position [ DISPLAY data-name ]     │ PROCEED   <           > │
                                                  └            └ ALWAYS  ┘ ┘
```

This command inserts a breakpoint at the position indicated. The Debugger assumes control just before the first executable statement that occurs after that position. For this reason, the Debugger considers a section and its first paragraph, and a paragraph and its first line, to be at the same "position".

When program execution reaches a breakpoint, the Debugger issues a message of the form:

```
Breakpoint at position in module program-name
```

If you use the DISPLAY option, the contents of data-name is displayed on your terminal. This option, therefore, saves you from typing a separate DISPLAY command every time the breakpoint is reached.

Using the PROCEED option saves you from having to type a separate PROCEED command to resume execution after the data-name is displayed. If you specify an integer, execution does not stop until the breakpoint has been encountered the specified number of times. At that point, you receive the CDB> prompt.

Specifying PROCEED ALWAYS lets you set up a "watchpoint" for the displayed data item. That is, each time the program passes the breakpoint, the contents of data-name is displayed but the program does not stop. This is useful, for example, if you want to check the way your program is changing the value of data-name as execution loops.

Examples

```
CDB> SET BREAK 46
```

Sets a breakpoint at line 46 in your program.

```
CDB> S B SECTION ALPHA
```

Sets a breakpoint at section ALPHA in your program.

```
CDB> SET BREAKPOINT LINE 101 DIS MONEY
```

Sets a breakpoint at line 101 in your program. Each time the breakpoint is reached, the contents of the data item MONEY are displayed.

3.3.4.2 CANCEL BREAKPOINT — The format of the CANCEL BREAKPOINT command is:

$$\underline{\text{C}}\text{ANCEL }\underline{\text{B}}\text{REAKPOINT} \begin{Bmatrix} \text{position} \\ \underline{\text{ALL}} \end{Bmatrix}$$

This command removes breakpoints. If you specify a position, only the breakpoint at that position is removed. The ALL option removes all the breakpoints currently set in the main program and in any subprograms.

Examples

```
CDB> C B EXT\2346
```

Cancels the breakpoint on line 2346 in program EXT.

```
CDB> CANCEL BREAK ALL
```

Cancels all the breakpoints in the main program and subprograms.

3.3.4.3 SHOW BREAKPOINTS — Use the SHOW BREAKPOINTS command to display information about each breakpoint you have set. The format for this command is:

$\underline{\text{SH}}\text{OW }\underline{\text{B}}\text{REAKPOINTS}$

When you enter this command:

```
CDB> SH B
```

The Debugger responds with the following:

1. The position of each breakpoint in the main program and subprograms
2. For each breakpoint, the name of any data item you have specified to be displayed
3. For each breakpoint, any proceed count you have specified

The information appears in this format:

```
position      Display:  data-name      Proceed:  integer
```

Examples

```
line 281 in module DATFD
line 273 in module DATFD     Display: PACKED-UNSIGNED-FD
line 265 in module DATFD
```

The examples indicate the information displayed if three breakpoints are in effect.

3.3.5 Using Synonyms

Three Debugger commands apply to synonyms:

- DEFINE associates a synonym with a data-name or a position.
- UNDEFINE deletes a synonym. It is no longer recognized by the Debugger.
- SHOW SYNONYMS displays information about the currently used synonyms.

3.3.5.1 DEFINE — The format of the DEFINE command is:

```
                         ⎧ data-name ⎫
DEFINE synonym [ = ]     ⎨           ⎬
                         ⎩ position  ⎭
```

After you use this command to define a synonym, you can use that synonym, rather than its corresponding data-name or position, in Debugger commands. If you refer to a particular data-name or position frequently during the debugging session, using a synonym for it saves time.

The synonym you specify must be unique. No data-name or procedure-name in the program can have the same name, and the name must not be a synonym that the Debugger is already using.

Examples

```
CDB>DEFINE X = SALESMAN-CODE
```

Defines X as a synonym for SALESMAN-CODE.

```
CDB>DEFINE TA = SALES-TOT OF STORE-A
```

Defines TA as a synonym for the data-name SALES-TOT that is subordinate to the data-name STORE-A.

```
CDB>DE Y SUB1\INITIAL-PARA
```

Defines Y as a synonym for the paragraph-name INITIAL-PARA in the program SUB1.

```
CDB>DEFINE A EMPTAB(5,4,16)
```

Defines A as a synonym for the table element EMPTAB(5,4,16).

Defining synonyms uses extra workspace. Conscientious use of UNDEFINE will avoid exhaustion of this resource. (See error message 1 in Appendix A)

3.3.5.2 UNDEFINE — The format of the UNDEFINE command is:

UNDEFINE synonym

After you use this command, the Debugger no longer recognizes the synonym. You can then define that synonym as some other data-name or position.

Examples

```
CDB> UNDEFINE SYN1
```

Removes the synonym SYN1 from the Debugger's synonym list.

```
CDB> U X
```

Removes the synonym X from the Debugger's synonym list.

3.3.5.3 SHOW SYNONYMS — The format of the SHOW SYNONYMS command is:

SHOW SYNONYMS

This command displays a list of all the currently recognized synonyms, along with their actual names in this format:

```
synonym:          actual name
```

Examples

```
C1: CMP-1
C2: CMP-2
C3: CMP-3
```

Displayed if three synonyms are in effect.

3.3.6 Using the PROCEED Command

To start program execution or to continue execution of your program after a breakpoint, use the PROCEED command. The format is:

PROCEED [integer]

If you specify an integer, the program ignores (integer – 1) breakpoints. That is, the Debugger does not give you control until the nth breakpoint is reached, where n = integer.

Examples

```
CDB> P
```

Begins or continues your program.

```
CDB> PROCEED 20
```

Begins or continues your program and tells it to ignore the next 19 breakpoints. Execution stops at the 20th breakpoint, and the Debugger will prompt you for another command.

3.3.7 Interrupting Program Execution

You can type CTRL/C to stop execution, rather than waiting for your program to encounter a breakpoint. After you type CTRL/C, you receive the Debugger prompt. You can enter any Debugger command at this point.

3.3.8 Using the STOP Command

To stop your program and end the debugging session, use the STOP command. The format of this command is:

STOP [RUN]

This command is equivalent to the COBOL-81 STOP RUN statement. Your program stops, the COBOL-81 OTS closes any open files, and control returns to the operating system.

Chapter 4
Reducing Your Task Size

4.1 When to Use Task Size Reduction Techniques

Your program is too large to fit into memory when it requires more than 32K words of address space at any one time. This 32K word area in memory must provide for the support routines your program needs when it runs, as well as the data specifications and procedures that originate in your source file. In the following discussion, the term *task image*, or simply *task*, refers to all code that one 32K word memory area must accommodate. The term *TSK file* refers to the file you execute with the RUN command.

When you reduce task size, you are changing the way the operating system accesses your program code and support routines. Task reduction techniques are the methods that you use to ensure that one 32K word partition in memory can handle everything your program specifies or needs in order to run.

All of the techniques available to you for reducing program task size cause varying degrees of performance degradation. Because of this, use the techniques described in this chapter only when your program is too large to fit into memory.

4.2 Reduction Techniques Available

You can use the following techniques to reduce your COBOL-81 task:

- Clustered resident libraries
- Overlayable RMS-11 input/output disk routines
- Callable subprograms with implicit overlays
- Segmentation
- File-handling optimization

This chapter discusses all of the task reduction techniques in this list except the last. Refer to Part IV, Chapter 7, File Optimization Techniques, for information on using file-handling techniques to reduce task size.

The techniques you use for a particular task depend primarily on whether or not (1) your system supports resident libraries and (2) your task image was created using COBOL-81 defaults.

4.3 Selecting Library Support Routines

All programs automatically include COBOL-81 OTS (Object Time System) support routines. In addition, RMS-11 (PDP-11 Record Management Services) support routines are automatically included if your program opens and closes files. FMS (PDP-11 Forms Management System) and user-defined routines also can be included, but only if you select the appropriate options when you create the task image.

The support routines included in your task image reside in libraries. OTS, RMS-11, and FMS libraries are DIGITAL-supplied. User-defined libraries are those created at your installation. All systems support disk versions of these libraries. Optionally, systems can support resident versions of these libraries. However, provisions for resident libraries must be included when your system manager generates your operating system, and then each library must be separately installed on your system.

To determine if your system supports resident libraries, and, if it does, which ones are installed:

- For a RSTS/E system, type:

```
SYSTAT/C
```

- For an RSX-11M/M-PLUS system, type:

```
SHOW COMMON
```

When you enter this command, the resident libraries that are installed on your system are named in the terminal display.

Including the most appropriate library support for your program can be the best way to reduce the size of your COBOL-81 task and minimize the effect of library use on performance. The following sections explain some of the major differences between disk and resident libraries and the options you can have on your system for a particular task.

4.3.1 Disk Libraries, Resident Libraries, and Clustered Resident Libraries

When you use disk libraries, their routines are actually built into your TSK file and add to its size. However, only those routines in the library that apply to your program are added, not the whole library. Programs that include only disk library support can be run on any system large enough to support the size of the task; that is, when you include only disk library support, your program is "transportable" to systems that do not support resident libraries. However, as the number of COBOL-81 programs your installation has to store and run increases, disk library support for these programs becomes more costly in terms of system disk and memory space.

When you use resident libraries, their routines are not included in your TSK file. (However, some OTS routines are an exception to this rule. See Section 4.3.2.) This means a savings in disk space, if many of the TSK files being stored use the same support routines. Resident libraries can be shared by other programs that run simultaneously with yours. This is a big advantage to your system because disk and memory space is not wasted by duplicates of program support routines. However, when you use resident libraries, your task probably includes support routines that your program does not need. This disadvantage (at least from the perspective of a single program) can be more than offset by using the clustering option that is available for resident libraries. The following paragraphs discuss disk and resident library use in greater detail.

Figure 4-1 illustrates the differences in using disk libraries, resident libraries, and clustered resident libraries in terms of disk and memory space allocation for a program. The figure assumes that:

- Two COBOL-81 applications (YOUR.TSK and OTHER.TSK) run simultaneously.
- Both applications require RMS-11 support.
- Neither program uses segmentation. (If the programs included the SEGMENT LIMIT clause, they would occupy more disk space than memory space.)

If you look at a single "typical" COBOL-81 program that requires RMS-11 as well as OTS support, using resident libraries always results in a smaller TSK file than using disk libraries. However, depending both on your program and on the system environment in which it runs, using resident libraries might or might not conserve memory space from your system's point of view.

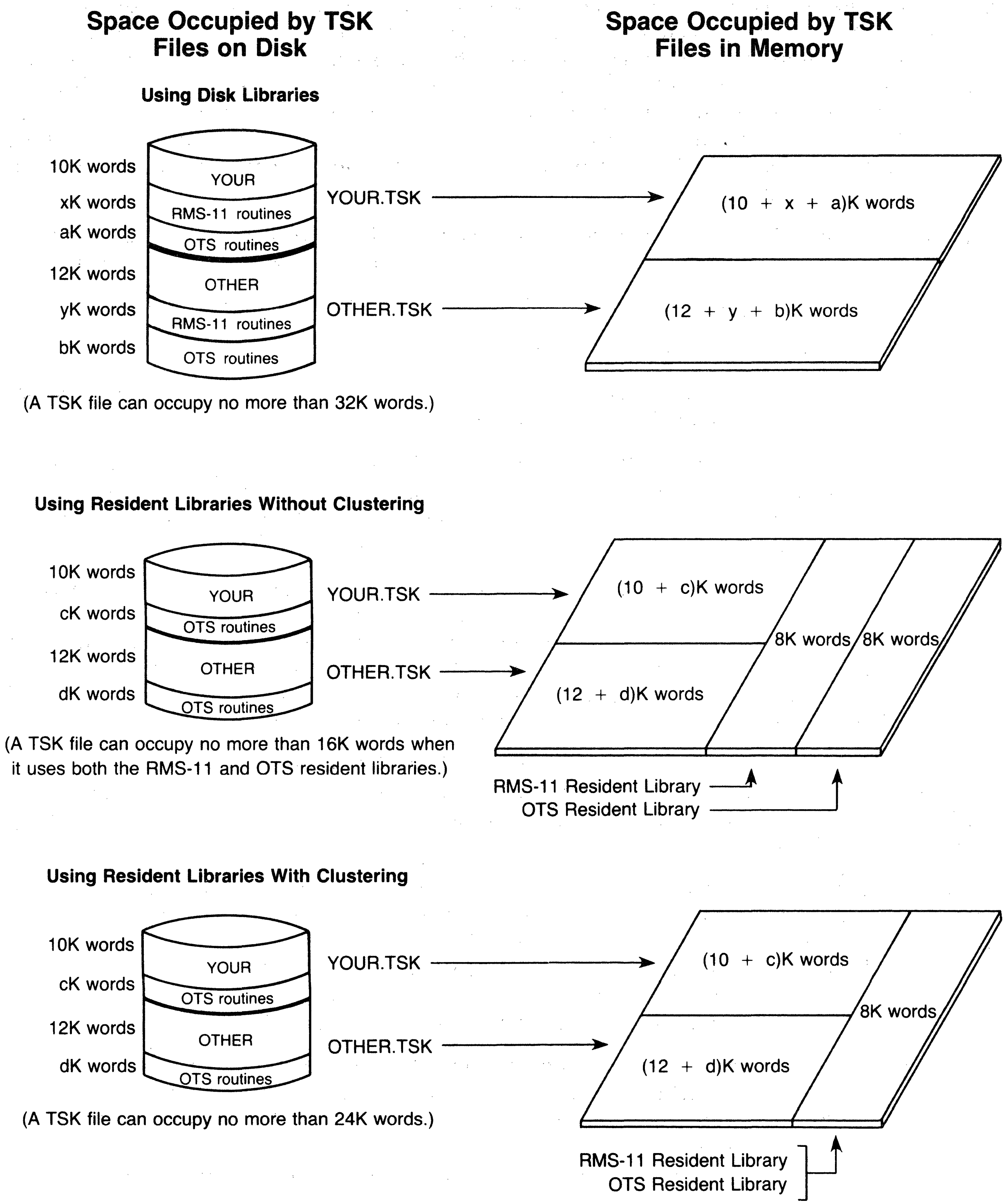
When your program runs with resident library support, the system must allocate space in memory for a whole library. Resident libraries remain in memory as long as they are being used, and they can be shared by programs with a wide range of support requirements. Therefore, *all* the support routines in the library are in memory with your program, not just the ones your program needs. For this reason, systems typically install support routines as resident libraries only when they (1) will be heavily used and (2) will be accessed by two or more programs running at the same time.

The clustering option for resident libraries is a newer DIGITAL feature than are resident libraries themselves. It allows two or more resident libraries to share the same address space. When you use the LINK/C81 command or the COBOL-81 BLDODL utility to create a task image, you can cluster up to three resident libraries.

Figure 4-1 illustrates the difference in memory allocation that clustering makes when you use resident libraries. The specific advantage of clustering resident libraries is that you can use resident libraries efficiently to conserve address space for your task. This is because clustering requires only enough words of address space for the largest resident library being used. In most cases, clustering requires only 8K words of address space no matter how many libraries you specify. (The only exception is when you specify a user-defined resident library that requires more than 8K words.)

To use clustered resident libraries (for one program) to both your and the system's advantage, your program must (1) access at least two resident libraries and (2) require support routines that in disk form total more than 8K words. If your program is a "typical" COBOL-81 program (one that opens and closes files), using clustered resident library support always (1) allows you to fit a larger program into memory than using disk libraries and (2) results in maximum memory conservation for your operating system.

Figure 4-1: How Use of Libraries Affects Task Size



Legend: *x* and *y* represent values that depend on a program's requirements for RMS-11 library support
a and *b* represent values that depend on a program's total requirements for OTS library support
c and *d* represent values that depend on a program's requirements for supplementary OTS support
(support not provided in the OTS resident library)
K represents the value 1000

C81ART-10004-50

4.3.2 Estimating COBOL-81 OTS Support

The COBOL-81 OTS support a program requires can range from 2K to 12K words.

If your program requires *only* COBOL-81 OTS support, including the OTS as disk library routines might allow you to fit a larger program into memory than using the resident library. The COBOL-81 resident library requires 8K words in your task image. If your program requires fewer OTS routines than are included automatically using the resident library, then using the disk library could be an advantage to you.

Also, the COBOL-81 resident library does not include all OTS support routines. If your task uses the COBOL-81 resident library and requires OTS routines not included in the resident library, those OTS routines are included in the TSK file.

If your task requires support from libraries in addition to the one containing the COBOL-81 OTS, using OTS disk routines generally loses all space-saving advantages. Most COBOL-81 programs require considerably more than 2K words of OTS support. For this reason, clustering the COBOL-81 resident library with other resident libraries results in significant task size reduction.

4.3.3 Using Disk Libraries Only

Some systems are too small to allow resident library installation. Other systems can support resident libraries, but for one reason or another, do not have the ones you require installed. In this case, you are limited to disk library support routines. This section discusses your task reduction options when you are limited to disk library use.

If your task image was created using COBOL-81 disk library defaults, it includes only those OTS routines and the smallest overlayable RMS-11 routine structure that your program needs.

If your task image was created using the COBOL-81 BLDODL utility (and then the Task Builder), check to see if either the BLDODL /LRG switch or /IO:NONOV switch was specified. Both of these switches are intended primarily to improve program performance. They add more space in the task image for RMS-11 routines than does the default BLDODL command line.

If you rerun BLDODL without the /LRG switch and then rebuild your task, you can reduce your task size by either 3K or 4K bytes (depending on whether your program requires sequential or indexed file support).

If you rerun BLDODL without the /IO:NONOV switch and then recreate your task image, you can reduce your task size more than 4K bytes. The amount of reduction depends on how many input/output routines your program requires.

At this point, the remaining options you can use to reduce your task size are callable subprograms and segmentation. Subprograms are discussed in Section 4.4 and Chapter 6 and segmentation in Section 4.5. You should also discuss use of resident libraries with your system manager if you think that using them would be to both your and the system's advantage.

4.3.4 Using Resident Libraries

This section discusses your task reduction options when you have the support routines your program needs installed as resident libraries on your system.

If your task was created using the LINK/C81 command:

- On a RSTS/E system, the LINK/C81 command line default includes resident library support (if available) and clusters resident libraries. If your program needs *only* COBOL-81 OTS support, you might be able to fit your program into memory by recreating a task image and specifying the /OTS:NORESIDENT qualifier in the LINK/C81 command line.
- On an RSX-11M/M-PLUS system, the LINK/C81 command line default includes disk library support in your task image. If your program needs only COBOL-81 OTS support, you might have the most size-efficient task image you can achieve by manipulating library support. However, if your program requires support from RMS-11 as well, using the clustering option with resident libraries can obtain more memory space for your program. If you specify resident library qualifiers in the LINK/C81 command line, clustering is invoked by default. This command line has the following format:

```
LINK/C81/OTS:RESIDENT/RMS:RESIDENT program-name
```

If your TSK file was created using the COBOL-81 BLDODL utility defaults and then the Task Builder, it includes disk library support. Therefore, if your program requires support from at least two resident libraries, you can reduce your task size by rerunning the BLDODL utility with the /CLU switch. Refer to Part I, Appendix D, for instructions on using the BLDODL utility.

4.4 Using Subprograms with Implicit Overlays

If you organize your application as subprograms, COBOL-81 can overlay some parts of your TSK file for you. This implicit overlaying significantly reduces your task size if your program contains many data items and literals and/or if its Procedure Division is large.

When you use subprograms, COBOL-81 can overlay the storage space required for each program's descriptors. (For each data item or literal in a program, COBOL-81 allocates three words of storage for its "descriptor".) Therefore, the space required for a task's descriptors has to be only as large as needed for the program that has the most data items and literals. Also, Procedure Division code for each program in the task can be overlaid. As with the case for descriptors, the space required for Procedure Division code has to be only as large as needed for the largest Procedure Division among the programs.

To take advantage of this COBOL-81 feature, you must specify the SEGMENT-LIMIT clause in the OBJECT-COMPUTER paragraph of each program included in your TSK file. Within this clause, you can specify any segment number within the integer range 1 through 49. You do not need to explicitly segment the Procedure Divisions of any program in the task (unless you need additional overlaying for a very large TSK file).

For more information on using callable subprograms, refer to Chapter 6, Interprogram Communication.

4.5 Using the COBOL-81 Segmentation Facility

COBOL-81 allows you to divide the Procedure Division into overlayable and nonoverlayable program segments to optimize memory use. Only enough memory space is allocated in the task image to store the largest overlayable program segment. Overlayable program segments are read into memory only when needed.

An overlayable program segment can be overlayed by and can overlay any other overlayable segment. A nonoverlayable program segment, however, can never be overlayed within the program. All code generated for the Identification Division through the Data Division becomes part of the nonoverlayable portion of the task image.

4.5.1 Programming Considerations

Using segmentation allows you to specify those segments you want to be overlayable and those you want to be nonoverlayable. To use segmentation, you must first define a segment limit by specifying the SEGMENT-LIMIT clause in the OBJECT-COMPUTER paragraph of the Environment Division of your source program. The integer value you specify in this clause is used by the compiler to determine whether a program segment is overlayable or nonoverlayable. A segment consists of one or more COBOL-81 sections. Each COBOL-81 section should be composed of a series of closely related operations designed to collectively perform a particular function.

In the Procedure Division, you specify a number in each section header that assigns that section to a segment. For example:

```
INITIALIZATION SECTION 10.
```

In this section header, INITIALIZATION is the user-defined word that identifies the section, SECTION is a COBOL-81 required word, and 10 is the number of the segment to which the section is assigned.

If you specify a segment-number whose value is less than the value specified in the SEGMENT-LIMIT clause, you have defined the section as being nonoverlayable. A segment-number whose value is greater than or equal to the value specified in the SEGMENT-LIMIT clause defines the segment as being overlayable. All segment-numbers specified must be in the range 0 through 49.

The most frequently referenced sections of your program should be made nonoverlayable. This reduces the number of I-O reads and improves performance. Assign segment-numbers that are less than the value specified in the SEGMENT-LIMIT clause to those sections.

Infrequently used sections should be made overlayable. Assign segment-numbers that are greater than or equal to the value specified in the SEGMENT-LIMIT clause to those sections. Sections that communicate with each other should be assigned to the same segment. Thus, the communicating sections are read into memory as a unit, again improving performance. Sections having identical segment-numbers are regrouped and assigned to the same segment. This regrouping does not affect the program's logical flow.

If your program task consists of a main program and callable subprograms, you can assign duplicate segment-numbers in any or all of the programs. However, the Task Builder expects to receive unique Program Section (PSECT) names, and you must ensure this by using the /NAMES:XX qualifier when you compile the main and subprograms.

Do not use more segmentation than required to get your task to fit into memory; the more heavily segmented a task is, the worse the performance. Keep the following three guidelines in mind when deciding how many overlayable segments you need and how large each should be:

1. The overlayable portion of your task is only large enough to accommodate the largest overlayable segment.
2. A few large segments run faster than many small segments.
3. Segments that are equal (or nearly equal) in size use memory most efficiently.

As a first try, choose a segment size that you feel will reduce the total task size enough to fit it into memory. Segment your program(s) accordingly and try to create the task image. If the attempt to create the task is successful, read the memory allocation map and note the size of the total task and the sizes of the segments. With this information, you can now tailor your program segments to maximize performance. To achieve the best run-time performance, you must use the minimum amount of segmentation necessary to fit your task in memory.

4.5.2 Creating a Segmented Task Image

The Task Builder needs segmented object code along with other information to include segmentation in the task image. This information is included automatically when you use DCL commands (the COBOL command followed by the LINK/C81 command).

However, if you intend to use the Task Builder's TKB command, the information is provided by the compile-time /BLD switch or by the COBOL-81 BLDODL utility. Remember that you can use the compiler /BLD switch only if the task image does not include subprograms.

4.5.2.1 Segmenting a Single-Program Task — To illustrate the concept of using the segmentation facility on a single-program task, the following program skeleton is presented. The value specified in the SEGMENT-LIMIT clause is 16. Therefore, any section assigned a segment-number equal to or greater than 16 belongs to an overlayable segment.

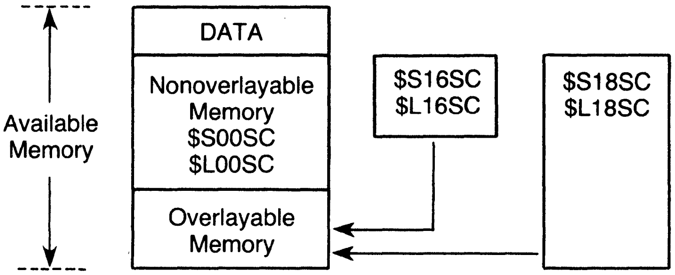
```
IDENTIFICATION DIVISION.
PROGRAM-ID. SEG-EXAMP.
ENVIRONMENT DIVISION.
SOURCE COMPUTER. PDP-11.
OBJECT-COMPUTER. PDP-11
            SEGMENT-LIMIT IS 16.
DATA DIVISION.
PROCEDURE DIVISION.

   SECT-NAME-1 SECTION 10.
          .
          .
          .
   SECT-NAME-2 SECTION 16.
          .
          .
          .
   SECT-NAME-3 SECTION 12.
          .
          .
          .
   SECT-NAME-4 SECTION 18.
          .
          .
          .
   SECT-NAME-5 SECTION 14.
```

The compiler produces the following default PSECT names, along with their literals and literal descriptors:

```
SEG-EXAMP

$S00SC          $L00SC
$S16SC          $L16SC
$S18SC          $L18SC
```

Figure 4-2 illustrates the way memory is allocated for the sample segmented program above. Program segments 10, 12, and 14 are nonoverlayable, while segments 16 and 18 are overlayable. The code and literals for sections with segment-numbers 10, 12, and 14 are located in PSECTs $S00SC and $L00SC. The relative sizes of the two overlayable segments are represented by the size of the segment blocks. Note that the memory space provided for the overlayable parts of the program is only large enough to accommodate the largest overlayable segment. Also, each overlayable segment is read into memory only when needed.

Figure 4-2: Memory Allocation of a Segmented Program



C81ART-10005-18

4.5.2.2 Segmenting a Multiple-Program Task — Segmenting a multiple-program task is very much like segmenting a single-program task. First you must specify the SEGMENT-LIMIT clause in each program to be segmented. Next you segment the Procedure Division of those programs you want to segment by assigning segment-numbers to sections.

More than likely, you will have to use duplicate segment-numbers in the various programs, particularly if there are numerous sections or more than one programmer is writing the source code. This poses no problem as long as you ensure that the compiled object code for each program in the task contains unique PSECT names. You do this by using the /NAMES:XX qualifier when you compile each segmented program. XX can be any of the characters $, A-Z, or 0-9. For example, consider a five-program task comprised of one main program and four subprograms, three of which include explicit segmentation:

1. Main program "MAIN":

```
SEGMENT-LIMIT IS 10.
                          SECTION 00.
                          SECTION 11.
                          SECTION 12.
                          SECTION 13.
                          SECTION 14.
```

(continued on next page)

2. Subprogram "SUBA":

```
SEGMENT-LIMIT IS 10.
              SECTION 00.
              SECTION 11.
              SECTION 12.
              SECTION 13.
              SECTION 15.
```

3. Subprogram "SUBB":

```
SEGMENT-LIMIT IS 10.
              SECTION 00.
              SECTION 11.
              SECTION 12.
              SECTION 16.
              SECTION 17.
```

4. Subprogram "SUBC":

```
SEGMENT-LIMIT IS 10.
              SECTION 00.
              SECTION 11.
              SECTION 18.
              SECTION 19.
              SECTION 20.
```

5. Subprogram "SUBD":

```
SEGMENT-LIMIT IS 10.
              SECTION 00.
```

If you do not use the /NAMES:XX qualifier, the compiler generates the following default PSECT names:

			Program		
Segment-Number	**MAIN**	**SUBA**	**SUBB**	**SUBC**	**SUBD**
11	$S11SC	$S11SC	$S11SC	$S11SC	—
	$L11SC	$L11SC	$L11SC	$L11SC	—
12	$S12SC	$S12SC	$S12SC	—	—
	$L12SC	$L12SC	$L12SC	—	—
13	$S13SC	$S13SC	—	—	—
	$L13SC	$L13SC	—	—	—
14	$S14SC	—	—	—	—
	$L14SC	—	—	—	—
15	—	$S15SC	—	—	—
	—	$L15SC	—	—	—
16	—	—	$S16SC	—	—
	—	—	$L16SC	—	—
17	—	—	$S17SC	—	—
	—	—	$L17SC	—	—
18	—	—	—	$S18SC	—
	—	—	—	$L18SC	—
19	—	—	—	$S19SC	—
	—	—	—	$L19SC	—
20	—	—	—	$S20SC	—
	—	—	—	$L20SC	—

Note the duplicate PSECT names. For example, each program (except SUBD) generates $S11SC and $L11SC PSECT names for segment number 11.

The Task Builder requires unique PSECT names. It recognizes only the first of any duplicate names when it creates a task.

For example, programs MAIN, SUBA, SUBB, and SUBC can reference only their own segment 11 routines at run time. However, the Task Builder recognizes only one name, $S11SC, for this segment number. Therefore, each program executes the same segment 11 routine.

By specifying the /NAMES:XX qualifier when compiling each program, you override the compiler's default PSECT names. For example, if you compile each program as follows, the resulting PSECT names are shown.

```
COBOL MAIN/LIST/NAMES:MM
COBOL SUBA/LIST/NAMES:AA
COBOL SUBB/LIST/NAMES:BB
COBOL SUBC/LIST/NAMES:CC
COBOL SUBD/LIST/NAMES:DD
```

			Program		
Segment-Number	**MAIN**	**SUBA**	**SUBB**	**SUBC**	**SUBD**
11	$S11MM	$S11AA	$S11BB	$S11CC	—
	$L11MM	$L11AA	$L11BB	$L11CC	—
12	$S12MM	$S12AA	$S12BB	—	—
	$L12MM	$L12AA	$L12BB	—	—
13	$S13MM	$S13AA	—	—	—
	$L13MM	$L13AA	—	—	—
14	$S14MM	—	—	—	—
	$L14MM	—	—	—	—
15	—	$S15AA	—	—	—
	—	$L15AA	—	—	—
16	—	—	$S16BB	—	—
	—	—	$L16BB	—	—
17	—	—	$S17BB	—	—
	—	—	$L17BB	—	—
18	—	—	—	$S18CC	—
	—	—	—	$L18CC	—
19	—	—	—	$S19CC	—
	—	—	—	$L19CC	—
20	—	—	—	$S20CC	—
	—	—	—	$L20CC	—

Now each PSECT name is unique. Therefore, when you link these programs, each program accesses its own segment number 11 routines. When MAIN references segment number 11 routines, it accesses its own $S11MM segment.

4.5.3 Reading a Memory Allocation Map

The memory allocation map provides the means for you to determine the exact amount of memory each segment requires. This section explains how to obtain a memory allocation map and how to read it.

You request a memory map listing by specifying the /MAP qualifier to the LINK/C81 command. An alternative way to request the listing is to specify the /MAP switch in the BLDODL utility command line.

A complete memory allocation map is comprised of numerous memory sub listings. What you are interested in is the memory allocation synopsis for each main or subprogram and for each overlayable segment. These are the only parts of the memory allocation map discussed in this section. Example 4-1 shows a sample skeleton program using segmentation. With a SEGMENT-LIMIT of 10, SECTION 10 and SECTION 12 are overlayable, and SECTION 5 is nonoverlayable. The parts of the memory allocation map applicable to segmentation are shown in Example 4-2.

Example 4-1: Sample Segmented Program

```
IDENTIFICATION DIVISION.
PROGRAM-ID. XSEG21.
ENVIRONMENT DIVISION.
SOURCE-COMPUTER. PDP-11.
OBJECT-COMPUTER. PDP-11
          SEGMENT-LIMIT IS 10.

DATA DIVISION.
PROCEDURE DIVISION.
MAINSTUFF SECTION.
* the root segment
MASTERSTUFF.
              .
              .
              .

* overlayable segment
SECTION-1 SECTION 10.
PROC-1.
              .
              .
              .

* overlayable segment
SECTION-2 SECTION 12.
PROC-2.
              .
              .
              .

* nonoverlayable segment
SECTION-3 SECTION 5.
PROC-3.
              .
              .
              .
          STOP RUN.
```

Example 4-2 shows a sample memory allocation map.

Example 4-2: Sample Memory Allocation Map

```
segment: $A00SC (entire program)

Memory allocation synopsis:

Section

$CROSC:(RO,D,GBL,REL,CON)      013674 000006 00006.  ←
$L00SC:(RO,D,GBL,REL,CON)      013702 000144 00100.  ←
$S00SC:(RO,I,GBL,REL,CON)      014046 000304 00196.  ←
$$ALVC:(RO,I,LCL,REL,CON)      014352 000000 00000.
$$RTS :(RO,I,GBL,REL,OVR)      013556 000002 00002.


segment: $A10SC (SECTION 10 overlayable segment)

Memory allocation synopsis:

Section

$L10SC:(RO,D,GBL,REL,CON)      014354 000036 00030.  ←
$S10SC:(RO,I,GBL,REL,CON)      014412 000116 00078.  ←
$$ALVC:(RO,I,LCL,REL,CON)      014530 000000 00000.
$$RTS: (RO,I,GBL,REL,OVR)      013556 000002 00002.


segment: $A12SC (SECTION 12 overlayable segment)

Memory allocation synopsis:

Section

$L12SC:(RO,D,GBL,REL,CON)      014354 000012 00010.  ←
$S12SC:(RO,I,GBL,REL,CON)      014366 000034 00028.  ←
$$ALVC:(RO,I,LCL,REL,CON)      014422 000000 00000.
$$RTS: (RO,I,GBL,REL,CON)      013556 000002 00002.
```

In the following discussions of the memory allocation synopses, the rows of interest (indicated with arrows) are those where the Section PSECT name is of the form $XXXSC. In these particular rows, the column of interest is the one at the far right, which lists, in decimal bytes, the memory allocated for the named PSECT.

Segment $A00SC is the memory allocation synopsis for the entire program. You are interested in the nonoverlayable portions. Row $CROSC lists the memory allocated for data descriptors, row $L00SC that for literals, and row $S00SC the memory for program sections whose section number is less than the segment limit number.

If, after segmentation, the program task image is still too large, one of your options is to reduce, if possible, the nonoverlayable portion of the image. If the entire Procedure Division is already overlayed, the only options left are to reduce the size of the largest overlayable segment or to divide the program into subprograms.

Segments $A10SC and $A12SC are the memory allocation synopses for the two overlayable segments in the program. From these notice that PSECT $S10SC has been allocated 108 bytes (30 for literals ($L10SC) and 78 bytes for the remainder ($S10SC)) while PSECT $S12SC has been allocated 38 bytes (10 for literals ($L12SC) and 28 bytes for the remainder ($S12SC)). To use memory most efficiently, you should make overlayable program segments as nearly equal in size as possible. If your Procedure Division is already fully overlayed, your only option at this point is to reduce the largest overlayable segment. This might mean breaking up present segments into additional smaller segments. Breaking up segment $A10SC into two smaller overlays would reduce the memory requirements for the overlays. Although fine-grained segmentation degrades performance, there is no other option to reducing your task size.

Chapter 5
Improving Program Performance

5.1 Introduction

If your COBOL-81 application program is large and/or processes large quantities of data, you will probably be interested in improving run-time performance. In addition, if you are compiling very large programs on a busy system, you may want to improve compile-time performance as well. This chapter discusses some general concepts that you can use to improve performance and provides references to other parts of this manual that give more detailed information.

5.2 Performance Versus Task Image Reduction

In general, significant reduction of run time can be accomplished only at the expense of increasing task image size. If you are forced to use segmentation/overlaying techniques to get your COBOL-81 program to fit into the available main memory, a significant reduction in run time might not be possible, but some minor improvements can be made. However, to optimize the run-time performance of your program, limit overlaying to the absolute minimum required to fit your program task image into main memory. See Chapter 4 for a discussion of program segmentation and overlaying techniques.

5.3 Using Compiler Qualifiers to Improve Performance

There are three compiler qualifiers that you can use to improve run-time and/or compile-time performance. These qualifiers are in addition to any other compiler qualifiers that you might have to specify for other reasons. The following sections discuss these qualifiers.

5.3.1 Using the /NOCHECK and /CHECK Qualifiers

By default, the compiler generates the code necessary to check valid ranges for subscripts, indexes, and nested PERFORM statements. This extra code both increases task image size and degrades performance. You can override these defaults with the /NOCHECK, /CHECK:NOBOUNDS, and /CHECK:NOPERFORM qualifier. Specify /CHECK:NOBOUNDS to turn off subscript and index range checking, /CHECK:NOPERFORM to turn off nested PERFORM statement checking, and /NOCHECK to suppress range checking for both subscripts and indexes, and for the nesting of PERFORM statements. Use of these qualifiers will improve run-time performance and also reduce your task image size. Use these qualifiers only after you have completely debugged your program. See Part I, Appendix D, for additional information.

5.3.2 Using the /TEMPORARY:dev Qualifier

The /TEMPORARY:dev qualifier instructs the compiler to store its temporary working files on the device you specify by "dev". By default, the compiler stores these files on the system disk. If for "dev", you specify a disk that has more free blocks, compile-time performance will improve.

5.4 Using BLDODL Switches to Improve Performance

There are two BLDODL switches that you can specify to significantly improve run-time performance. Both affect the overlaying of RMS-11 I/O routines. If your program performs I/O functions, the compiler builds into the task image those I/O routines required. These routines are provided by RMS-11. Whether or not these routines are overlayed, and if so how much space is allocated for their storage, has a significant effect on run-time performance. These switches are in addition to other BLDODL switches that you might have to specify for other reasons. The following sections discuss these BLDODL switches.

5.4.1 Using the BLDODL/IO:NONOV Switch

If you do not specify any BLDODL/IO switches, by default the required RMS-11 I/O routines are overlayable and increase the task image size. For sequential file support, 8K bytes are allocated and for indexed files 9K bytes are allocated. If it is not absolutely necessary for you to overlay the RMS-11 routines, specify the BLDODL/IO:NONOV switch. This switch specifies that RMS-11 routines are not overlayable, but instead are to be included as part of the task image. Because there is no longer a need to read the overlayable routines into memory, performance improves considerably. See Part I, Appendix D for additional information.

5.4.2 Using the BLDODL/LRG Switch

If you do overlay RMS-11 routines you can improve run-time performance by allocating more storage space for the routines. To specify a large overlay structure for RMS-11 routines, use the BLDODL/LRG switch. Instead of the 8K or 9K bytes allocated for the routines, 12K bytes are now allocated, thus improving performance by reducing the number of reads to memory. This might mean that you will have to rely more heavily on other segmentation/overlaying techniques to obtain the additional memory space needed. See Part I, Appendix D, for additional information.

5.5 Using Terminal Format Source Programs

You can write your source programs using either the conventional ANSI format or the DIGITAL terminal format. The DIGITAL terminal format eliminates line-number and identification fields and allows tab characters and short lines, thus saving disk space and reducing compile time. If your source program is written in ANSI format you can convert it to terminal format before compiling by using the REFORMAT Utility. See Chapter 1 for instructions on using the REFORMAT Utility.

5.6 Data Handling Techniques for Improving Performance

There are several data handling techniques you can use to improve program performance. These include:

- Using the same scale in arithmetic operations
- Reducing the number of significant digits in numeric data types
- Using indexes instead of subscripts in table handling
- Avoiding decimal truncation
- Using the ADD, SUBTRACT, MULTIPLY, and DIVIDE statements instead of the COMPUTE statement
- Using the GO TO DEPENDING statement instead of IF ... GO TO sequences
- Using the SEARCH ALL statement instead of the SEARCH statement

These techniques are explained in Part III, Chapter 4, Data Handling Optimization

5.7 Using File Optimization to Improve Performance

There are several file optimization techniques available to you to improve program performance. However, some of these techniques may involve significant trade offs. These file optimizing techniques include:

- Optimizing file I/O ... the APPLY clause
- Sharing record areas
- Reserving additional I/O buffer space
- Tailoring I/O buffers
- Optimizing file design

These techniques are explained in Part IV, Chapter 7, File Optimization Techniques.

5.8 Using Subprograms

You can reduce the compile time for a large program if you break it into subprograms. This way, you only need to recompile those subprograms with coding changes. This technique also simplifies program segmentation if you keep segmentation in mind when you break up your large programs. A properly designed overlay structure will have minimal effect on run-time performance.

Chapter 6
Interprogram Communication

COBOL-81 allows you to link separately compiled COBOL-81 and non-COBOL-81 programs into a single task image. The CALL statement then allows these separately compiled programs to communicate with each other when the task executes.

This chapter introduces you to multiple program (COBOL-81 and non-COBOL-81) tasks. It discusses and presents examples of how to transfer execution control and data from one program to another within the task.

The first section explains multiple COBOL-81 program tasks. The remaining sections then discuss the inclusion of non-COBOL-81 programs into the task image and how to communicate with them from a COBOL-81 program.

6.1 A Multiple COBOL-81 Program Task

A multiple COBOL-81 program task must consist of both:

- One main COBOL-81 program
- One or more COBOL-81 subprograms

A main program calls subprograms but cannot be called in return. Task execution begins and ends in its Procedure Division. It contains one or more CALL statements and is a *"calling"* program.

A subprogram must always be called by a main program or another subprogram. It contains none, one, or more than one CALL statement. If it contains a CALL statement, it is both a *"calling"* and a *"called"* program. If it does not contain a CALL statement it is a *"called"* program only.

6.1.1 Identifying a COBOL-81 Subprogram

At compile time, the COBOL-81 compiler must be able to distinguish between main programs and subprograms. Depending on the need to pass parameters between programs, there are two methods of identifying subprograms.

1. When passing parameters between programs, you must specify the USING phrase in the Procedure Division header of the called subprogram. The USING phrase identifies this particular program as a subprogram.

   For example:

   ```
   PROCEDURE DIVISION USING A, B.
   ```

   The use of the data-name is discussed in Section 6.2.

2. When you do not need to pass parameters, you do not need to specify the USING phrase. To identify the program as a subprogram, you must specify the /SUBPROGRAM compiler qualifier.

   For example:

   ```
   COBOL SUB1 = SUB1/SUBPROGRAM
   ```

6.1.2 Compiling Main and Subprograms

When you use subprograms and also use segmentation in more than one of the programs, you must ensure that all PSECT names generated by the compiler and used by the Task Builder are unique. You do this by specifying the /NAMES:XX compiler qualifier when you compile any of the programs using segmentation. XX is any two-character alphanumeric combination you choose to uniquely identify PSECT names. This is in addition to any other compiler qualifiers that you might have specified.

For example, a multiple program task consists of:

MAIN (with segmentation)
SUB1 (no segmentation)
SUB2 (with segmentation)

You compile these as follows:

```
COBOL MAIN = MAIN/NAMES:MM
COBOL SUB1 = SUB1
COBOL SUB2 = SUB2/NAMES:AA
```

Now all PSECT names assigned by the compiler will be unique and of the form *$SnnXX* where nn are the section numbers you assigned to the program segments in the Procedure Division and XX are the alphanumeric characters you specified in the /NAMES qualifier. See Chapter 4 for a more complete discussion of program segmentation.

6.1.3 Transferring Execution Control with the CALL Statement

You control a multiple program execution sequence in much the same way that you control the execution sequence in a single COBOL-81 program.

In a single COBOL-81 program, you execute a GO TO or PERFORM statement to change its logic flow. In a multiple COBOL-81 program, you execute both:

- A controlling CALL statement in the calling program (main or subprogram).
- An EXIT PROGRAM statement in the called subprogram.

6.1.3.1 The CALL Statement — Execution of a CALL statement causes the task's execution control to pass from the calling program to the beginning of the called subprogram's Procedure Division. The first time the called subprogram assumes execution control, its state is that of a fresh copy of the program. Each subsequent time it is called, its state is as it was upon the last exit from that program.

Note

If you are passing parameters between programs, you must also specify the USING phrase in the CALL statement. See the *COBOL-81 Language Reference Manual* for a complete discussion of the CALL statement and the USING phrase.

6.1.3.2 The EXIT PROGRAM Statement — To return execution control to the calling program, the called subprogram executes an EXIT PROGRAM statement. The EXIT PROGRAM format for the called program is:

<u>EXIT</u> <u>PROGRAM</u>

You can include more than one EXIT PROGRAM statement in a subprogram. However, if it appears in a consecutive sequence of imperative statements, it must appear as the last statement of the sequence. For example:

```
IF A = B DISPLAY "A equals B", EXIT PROGRAM.

READ INPUT-FILE AT END DISPLAY "End of input file"
                       PERFORM END-OF-FILE-ROUTINE
                       EXIT PROGRAM.
```

If you do not specify an EXIT PROGRAM statement, control returns to the calling program after the last executable statement.

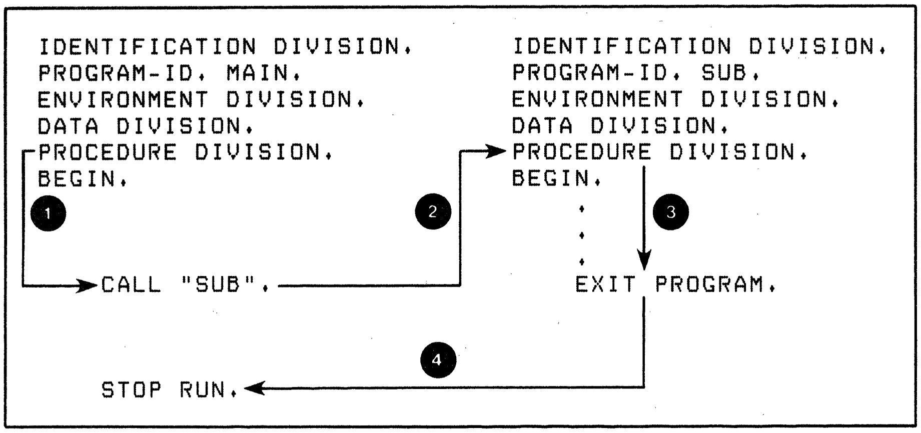
Control returns to the next statement following the CALL statement when an EXIT PROGRAM statement executes.

Note

When the EXIT PROGRAM executes, the called program is considered to have reached the ends of the ranges of all PERFORM statements. Thus, an error does not occur if the called program is entered again during image execution.

6.1.3.3 Sharing Execution Control — Figure 6-1 shows how execution control is shared between a main program and a subprogram.

Figure 6-1: Sharing Execution Control Between a Main Program and One Subprogram



```
IDENTIFICATION DIVISION.        IDENTIFICATION DIVISION.
PROGRAM-ID. MAIN.               PROGRAM-ID. SUB.
ENVIRONMENT DIVISION.           ENVIRONMENT DIVISION.
DATA DIVISION.                  DATA DIVISION.
PROCEDURE DIVISION.             PROCEDURE DIVISION.
BEGIN.                          BEGIN.
                                    .
                                    .
                                    .
    CALL "SUB".                     EXIT PROGRAM.


    STOP RUN.
```

C81ART-10006-18

6.1.3.4 Nesting CALL Statements — A called subprogram can itself transfer execution control to a subprogram. This technique is known as CALL statement nesting. For example, consider a nested task that executes a series of three call statements from three separate programs:

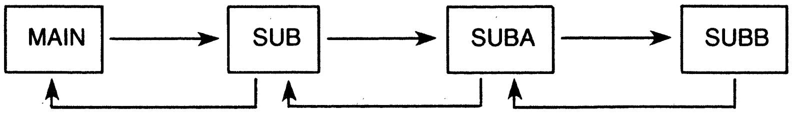
MAIN calls SUB,
SUB then calls SUBA,
SUBA then calls SUBB.

In Figure 6-2 SUBB cannot directly call SUBA, nor can it call SUB or MAIN.

Note

The COBOL-81 OTS issues a fatal error message if a called subprogram either directly or indirectly calls a subprogram already in a nest.

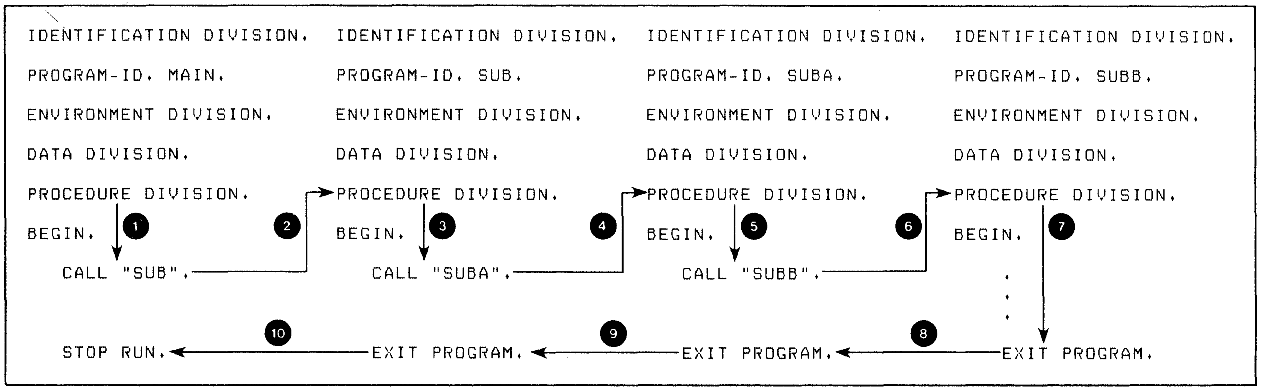
Figure 6-2: Nesting CALL Statements



C81ART-10007-6

Figure 6-3 shows how execution control is shared between a main program and multiple subprograms.

Figure 6-3: Sharing Execution Control Between a Main Program and Multiple Subprograms



```
IDENTIFICATION DIVISION.  IDENTIFICATION DIVISION.  IDENTIFICATION DIVISION.  IDENTIFICATION DIVISION.
PROGRAM-ID. MAIN.         PROGRAM-ID. SUB.          PROGRAM-ID. SUBA.         PROGRAM-ID. SUBB.
ENVIRONMENT DIVISION.     ENVIRONMENT DIVISION.     ENVIRONMENT DIVISION.     ENVIRONMENT DIVISION.
DATA DIVISION.            DATA DIVISION.            DATA DIVISION.            DATA DIVISION.
PROCEDURE DIVISION.       PROCEDURE DIVISION.       PROCEDURE DIVISION.       PROCEDURE DIVISION.
BEGIN.                    BEGIN.                    BEGIN.                    BEGIN.
   CALL "SUB".               CALL "SUBA".              CALL "SUBB".              .
                                                                                 .
                                                                                 .
   STOP RUN.                 EXIT PROGRAM.             EXIT PROGRAM.             EXIT PROGRAM.
```

C81ART-10008-26

The following three programs illustrate their execution sequence through the display of a series of 12 messages on the default terminal. Task execution begins in MAIN with message number 1. It ends in MAIN with message number 12. The task's message sequence is shown following the program example.

Example 6-1: Sharing Program Execution Control Between a Main Program and Multiple Subprograms

```
IDENTIFICATION DIVISION.
*
* MAIN is a calling program only
*
PROGRAM-ID. MAIN.
ENVIRONMENT DIVISION.
DATA DIVISION.
PROCEDURE DIVISION.
BEGIN.
    DISPLAY " 1. MAIN has the first execution control.     ".
    DISPLAY " 2. MAIN transfers execution control to SUB1  ".
    DISPLAY "         upon executing the following CALL.   ".
    CALL "SUB1".
    DISPLAY "11. MAIN has the last execution control.      ".
    DISPLAY "12. MAIN terminates the entire task upon      ".
    DISPLAY "         execution of the STOP RUN statement  ".
    STOP RUN.

IDENTIFICATION DIVISION.
*
* SUB1 is both a called and a calling subprogram
*
*      It is called by MAIN
*
*      It then calls SUB2
*
PROGRAM-ID. SUB1.
ENVIRONMENT DIVISION.
DATA DIVISION.
PROCEDURE DIVISION.
```

(continued on next page)

Example 6-1: Sharing Program Execution Control Between a Main Program and Multiple Subprograms (Cont.)

```
BEGIN.
    DISPLAY " 3.      This is the entry point to SUB1.    ".
    DISPLAY " 4. SUB1 now has execution control.          ".
    DISPLAY " 5. SUB1 transfers execution control to SUB2 ".
    DISPLAY "         upon executing the following CALL.  ".
    CALL "SUB2".
    DISPLAY " 9. SUB1 regains execution control.          ".
    DISPLAY "10. SUB1 returns execution control to MAIN   ".
    DISPLAY "         after executing the following       ".
    DISPLAY "         EXIT PROGRAM statement.             ".
    EXIT PROGRAM.

IDENTIFICATION DIVISION.
*
* SUB2 is a called subprogram only
*
*     It is called by SUB1
*
PROGRAM-ID. SUB2.
ENVIRONMENT DIVISION.
DATA DIVISION.
PROCEDURE DIVISION.
BEGIN.
    DISPLAY " 6.      This is the entry point to SUB2. ".
    DISPLAY " 7. SUB2 now has execution control.       ".
    DISPLAY " 8. SUB2 returns execution control to SUB1".
    DISPLAY "         upon executing the following     ".
    DISPLAY "         EXIT PROGRAM statement.          ".
    EXIT PROGRAM.
```

The message sequence results displayed on the terminal are:

```
 1. MAIN has the first execution control.
 2. MAIN transfers execution control to SUB1
         upon executing the following CALL.
 3.      This is the entry point to SUB1.
 4. SUB1 now has execution control.
 5. SUB1 transfers execution control to SUB2
         upon executing the following CALL.
 6.      This is the entry point to SUB2.
 7. SUB2 now has execution control.
 8. SUB2 returns execution control to SUB1
         upon executing the following
         EXIT PROGRAM statement.
 9. SUB1 regains execution control.
10. SUB1 returns execution control to MAIN
         after executing the following
         EXIT PROGRAM statement.
11. MAIN has the last execution control.
12. MAIN terminates the entire task upon
         execution of the STOP RUN statement.
```

6.2 Accessing Another Program's Data Division

In a multiple COBOL-81 program task, a called subprogram can have access to its calling program's Data Division; however, the calling program controls how much of it is to be accessible to the called subprogram through:

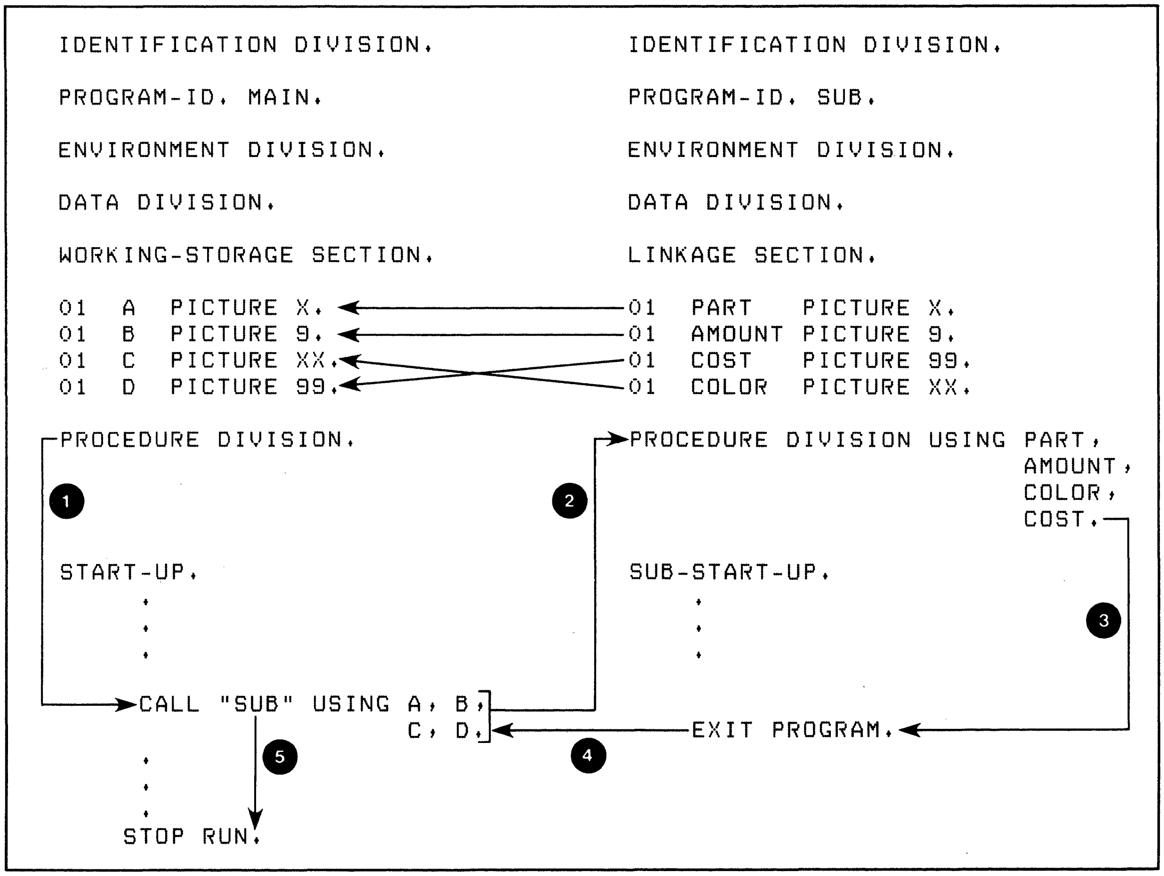
1. The USING phrase in both the CALL statement and the Procedure Division header
2. The Linkage Section

6.2.1 The USING Phrase

To access a calling program's Data Division use a CALL statement and a Procedure Division USING phrase. The CALL statement's and the Procedure Division's USING phrase must contain equal numbers of data-names. For more information on the USING phrase, see the *COBOL-81 Language Reference Manual.*

The order of appearance of USING identifiers in both calling and called programs determines the correspondence of single sets of data available to the called subprogram. The correspondence is positional and not by name. Figure 6-4 shows the correspondence of single sets of data.

Figure 6-4: Correspondence of Single Sets of Data



```
IDENTIFICATION DIVISION.          IDENTIFICATION DIVISION.
PROGRAM-ID. MAIN.                 PROGRAM-ID. SUB.
ENVIRONMENT DIVISION.             ENVIRONMENT DIVISION.
DATA DIVISION.                    DATA DIVISION.
WORKING-STORAGE SECTION.          LINKAGE SECTION.
01  A  PICTURE X.                 01  PART    PICTURE X.
01  B  PICTURE 9.                 01  AMOUNT  PICTURE 9.
01  C  PICTURE XX.                01  COST    PICTURE 99.
01  D  PICTURE 99.                01  COLOR   PICTURE XX.

PROCEDURE DIVISION.               PROCEDURE DIVISION USING PART,
                                                           AMOUNT,
                                                           COLOR,
                                                           COST.
START-UP.                         SUB-START-UP.
     .                                 .
     .                                 .
     .                                 .
         CALL "SUB" USING A, B,
                          C, D.          EXIT PROGRAM.
     .
     .
     .
   STOP RUN.
```

C81ART-10009-35

When execution control passes to SUB, the called program can access the four data items in the calling program by referring to the data-names in its own Procedure Division USING phrase. The data-names correspond as shown in Table 6-1.

Table 6-1: Correspondence of Data-Names

Calling Program Data-Name	Called Subprogram Data-Name
A	PART
B	AMOUNT
C	COLOR
D	COST

6.2.2 The Linkage Section

You must define each data-name in the Procedure Division USING phrase data-item-list in the called subprogram's Linkage Section. For example:

```
LINKAGE SECTION.

01 PART     PICTURE ...
01 AMOUNT   PICTURE ...
01 INVOICE  PICTURE ...
01 COLOR    PICTURE ...
01 COST     PICTURE ...

PROCEDURE DIVISION USING PART, AMOUNT, COLOR, COST.
```

Of those items you define in the Linkage Section, only those appearing in the Procedure Division USING phrase data-item-list are accessible to the called program. In the above example, INVOICE is not accessible to the called program because it is not in the Procedure Division USING phrase data-item-list.

Whenever a subprogram references a data-name from the Procedure Division USING phrase data-item-list, the subprogram processes it according to the definition in its own Linkage Section.

The compiler does not allocate storage space for data items in the called subprogram's Linkage Section. Subprogram references to those items are resolved at run time. The OTS equates the reference's address in the subprogram to the location in the calling program. For index-names, no such correspondence exists; index-name references in the calling and called programs always refer to separate indexes.

A called program's Procedure Division can reference data items in its Linkage Section only if it references one of the following:

- Any data item in the Procedure Division USING data-item-list.
- A data item that is subordinate to a Linkage Section data item in the Procedure Division USING data-item-list.
- Any other association with a data item in the Procedure Division USING data-item-list. For example, index-name, redefinition, etc.

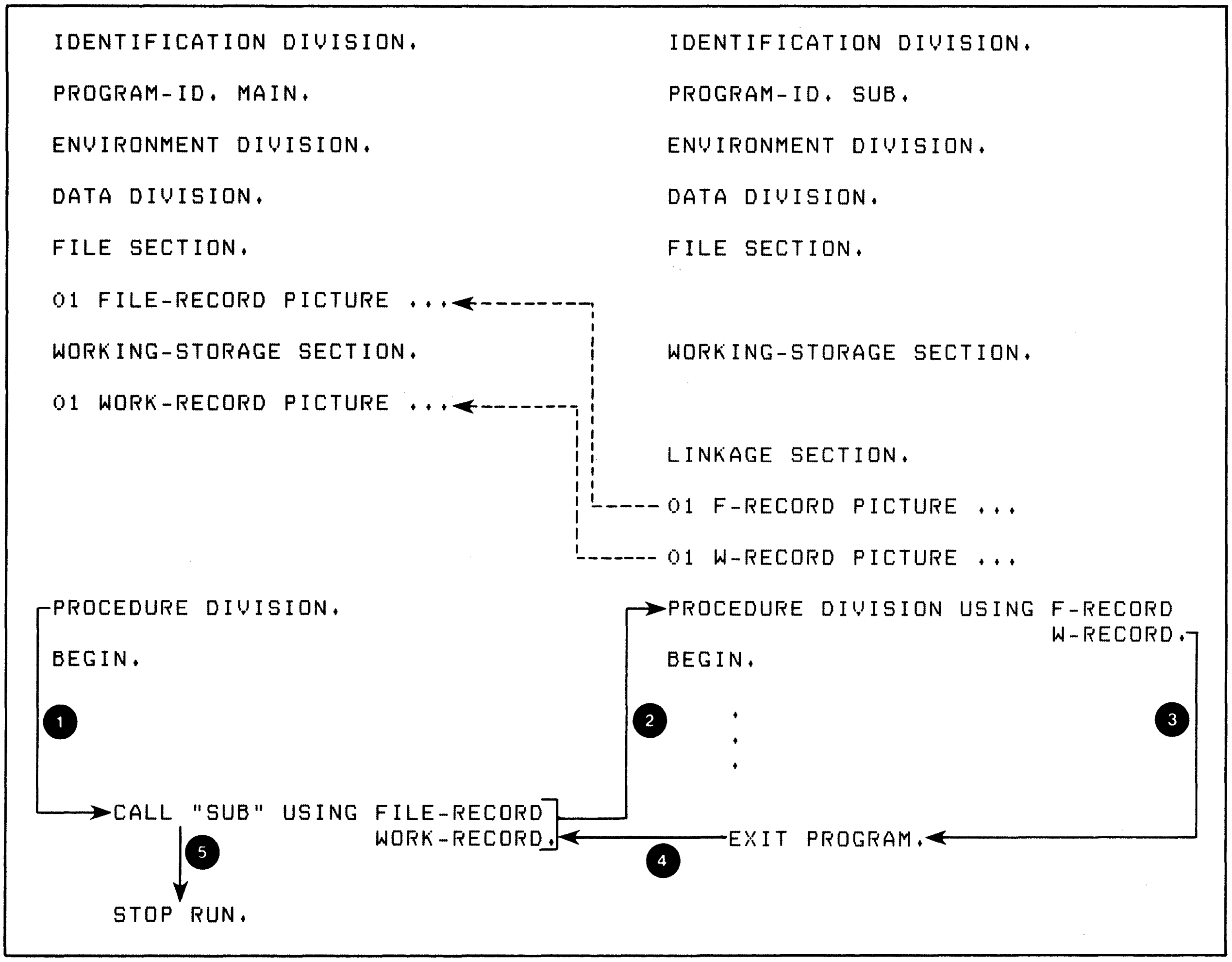
6.2.3 Examples

In the example in Figure 6-5, SUB is called by MAIN (see the solid arrows). Because MAIN includes FILE-RECORD and WORK-RECORD in its CALL "SUB" USING statement, SUB can reference these data items just as if they were in its own Data Division. However, SUB accesses these two data items with its own data-names, F-RECORD and W-RECORD (see the broken line arrows).

In the example in Figure 6-6, SUBA references data items in both MAIN and SUB.

Example 6-2 shows how a subprogram (SUB1) redefines data items in its own Linkage Section.

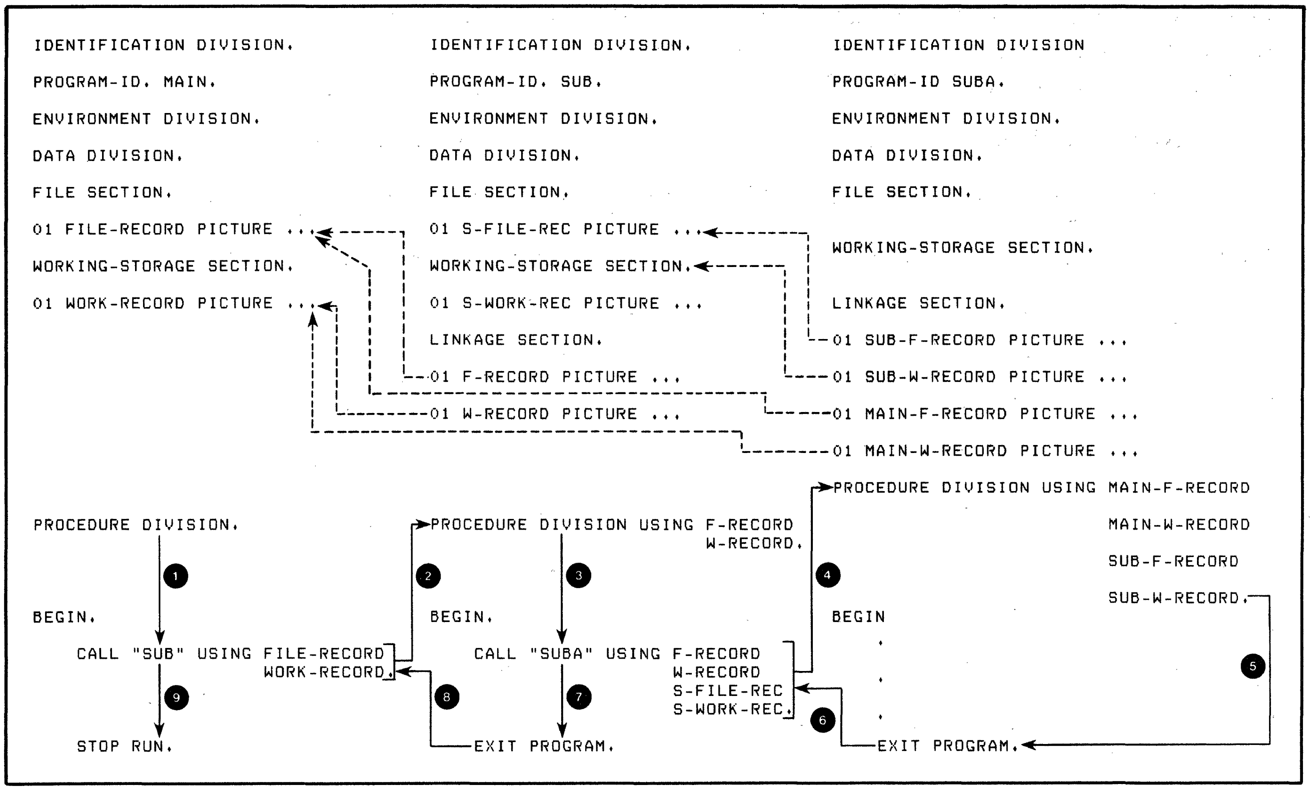
Figure 6-5: Sharing Execution Control and Data Between a Main Program and One Subprogram



```
IDENTIFICATION DIVISION.                    IDENTIFICATION DIVISION.
PROGRAM-ID. MAIN.                           PROGRAM-ID. SUB.
ENVIRONMENT DIVISION.                       ENVIRONMENT DIVISION.
DATA DIVISION.                              DATA DIVISION.
FILE SECTION.                               FILE SECTION.
01 FILE-RECORD PICTURE ...
WORKING-STORAGE SECTION.                    WORKING-STORAGE SECTION.
01 WORK-RECORD PICTURE ...
                                            LINKAGE SECTION.
                                            01 F-RECORD PICTURE ...
                                            01 W-RECORD PICTURE ...
PROCEDURE DIVISION.                         PROCEDURE DIVISION USING F-RECORD
                                                                     W-RECORD.
BEGIN.                                      BEGIN.
                                                .
                                                .
                                                .
        CALL "SUB" USING FILE-RECORD
                         WORK-RECORD.               EXIT PROGRAM.

        STOP RUN.
```

C81ART-10010-40

Figure 6-6: Sharing Nested Execution Control and Data Between a Main Program and Multiple Subprograms



```
IDENTIFICATION DIVISION.
PROGRAM-ID. MAIN.
ENVIRONMENT DIVISION.
DATA DIVISION.
FILE SECTION.
01 FILE-RECORD PICTURE ...
WORKING-STORAGE SECTION.
01 WORK-RECORD PICTURE ...

PROCEDURE DIVISION.

BEGIN.
    CALL "SUB" USING FILE-RECORD
                     WORK-RECORD.

    STOP RUN.
```

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SUB.
ENVIRONMENT DIVISION.
DATA DIVISION.
FILE SECTION.
01 S-FILE-REC PICTURE ...
WORKING-STORAGE SECTION.
01 S-WORK-REC PICTURE ...
LINKAGE SECTION.
01 F-RECORD PICTURE ...
01 W-RECORD PICTURE ...

PROCEDURE DIVISION USING F-RECORD
                         W-RECORD.

BEGIN.
    CALL "SUBA" USING F-RECORD
                      W-RECORD
                      S-FILE-REC
                      S-WORK-REC.

EXIT PROGRAM.
```

```
IDENTIFICATION DIVISION
PROGRAM-ID SUBA.
ENVIRONMENT DIVISION.
DATA DIVISION.
FILE SECTION.

WORKING-STORAGE SECTION.

LINKAGE SECTION.
01 SUB-F-RECORD PICTURE ...
01 SUB-W-RECORD PICTURE ...
01 MAIN-F-RECORD PICTURE ...
01 MAIN-W-RECORD PICTURE ...
PROCEDURE DIVISION USING MAIN-F-RECORD
                         MAIN-W-RECORD
                         SUB-F-RECORD
                         SUB-W-RECORD.
BEGIN
      .
      .
      .
EXIT PROGRAM.
```

C81ART-20900-30

Example 6-2: Redefining a Calling Program's Data Items in the Called Subprogram's Linkage Section

```
IDENTIFICATION DIVISION.
*
* MAIN is a calling program
*

PROGRAM-ID. MAIN.

ENVIRONMENT DIVISION.
DATA DIVISION.
FILE SECTION.

WORKING-STORAGE SECTION.

01 A PIC X.
01 B PIC 9.
01 C PIC X(5).
01 D PIC S9(5)V99.
01 E PIC ZZZ,ZZZ.99-.
01 F PIC XXXXX.
01 G PIC 99999.
01 H.
   03 H1 PIC X.
   03 H2 PIC XX.
   03 H3 PIC XXX.
```

(continued on next page)

Example 6-2: Redefining a Calling Program's Data Items in the Called Subprogram's Linkage Section (Cont.)

```
01 I.
   03 I1 PIC 999.
   03 I2 PIC 99.
   03 I3 PIC 9.
01 J.
   03 J1.
      05 J1-1 PIC X(11).
      05 J1-2 PIC X(12).
   03 J2.
      05 J2-1 PIC X(21).
      05 J2-2 PIC X(22).
01 K PIC X(1000).
01 L PIC X(132).

PROCEDURE DIVISION.

BEGIN.
    CALL "SUB1" USING A, B, C, D, E, F, G, H, I, J.
    STOP RUN.

IDENTIFICATION DIVISION.
*
* SUB1 is a called program
*

PROGRAM-ID. SUB1.

ENVIRONMENT DIVISION.
DATA DIVISION.

WORKING-STORAGE SECTION.
01 WORK-AREA PIC X(1000).

LINKAGE SECTION.
01 SUBA PIC X.
01 SUBB PIC 9.
01 SUBC PIC 9(5).
01 SUBD PIC S9(5)V99.
01 SUBE PIC X(11).
01 SUBF PIC XXXXX.
01 SUBG PIC XXXXX.
01 SUBH PIC X(6).
01 SUBI PIC 9(6).
01 SUBJ PIC X(66).
PROCEDURE DIVISION USING SUBA, SUBB, SUBC, SUBD, SUBE,
            SUBF, SUBG, SUBH, SUBI, SUBJ.
BEGIN.
    DISPLAY "This is the entry point to SUB1                ".
    DISPLAY "SUB1 can access ten data items in MAIN         ".
    DISPLAY "     Just as if they are in SUB1.              ".
    DISPLAY "     However, note that SUB1 redefines some    ".
    DISPLAY "     of the MAIN data descriptions.            ".
    DISPLAY "     For example, SUB1 references SUBC as a    ".
    DISPLAY "     five character numeric item, whereas MAIN ".
    DISPLAY "     references C as a five character alpha-   ".
    DISPLAY "     numeric item.                             ".
    DISPLAY "SUB1 cannot however reference any other data   ".
    DISPLAY "     in MAIN that has not been listed in the   ".
    DISPLAY "     Using phrase. For example, K and L in MAIN".
    DISPLAY "     cannot be referenced by SUB1.             ".
    DISPLAY "Execution control returns to MAIN after        ".
    DISPLAY "     executing the next statement.             ".
    EXIT PROGRAM.
```

6.2.4 COBOL-81 OTS – Error Checking

At execution, the COBOL-81 OTS performs a check to ensure that the number of arguments passed to a called subprogram is the same as the number expected. That is, the subprogram Procedure Division USING phrase data-item-list must contain the same number of data-names as the USING phrase in the calling program's CALL statement. If the number of arguments is not equal, the OTS issues a diagnostic error message and aborts the task. No checks are made to ensure that the passed arguments are the same size as the expected arguments. It is the programmer's responsibility to ensure that these size limits are compatible.

Recursive calls to COBOL-81 subprograms are not allowed. If a subprogram contains a call that directly or indirectly causes a subprogram to be reentered before it has exited from its original entry, the OTS issues a diagnostic error and aborts the task.

6.3 Including Non-COBOL-81 Programs in a Task

COBOL-81 object modules can be task-built with other non-COBOL-81 object modules. This capability is often useful, especially when a feature is not available in COBOL-81, but is available in another language.

> **Note**
>
> Non-COBOL-81 programs must not include nor use the file management services provided by RMS-11 (Record Management Services) if a COBOL-81 program performs file I/O in the same task. (COBOL-81 programs reference RMS-11 to perform file I/O.) Other file management services are available to the non-COBOL-81 program. This note of caution is very important, because the PDP-11 programming languages do not share a common Object Time System (OTS). For more information on alternative file management services, see *RSTS/E System Directives Manual* or *RSX-11 I/O Operations Reference Manual.*

To use BLDODL to include a non-COBOL-81 object module in a task image, you must:

1. Create a standard COBOL-81 SKL file (using the text editor).
2. Specify this SKL file as input to BLDODL.

A standard COBOL-81 SKL file for a non-COBOL-81 object module contains at least one of the following directive lines:

- Object Program ID Line. This line is required. It identifies the object module to be included in the task image. The format of this line is:

```
;COBOBJ=XXXXXX.OBJ
```

where:

XXXXXX.OBJ   is the name of the object module to be included in the task image.

- Main Program ID Line. This line is present only for non-COBOL-81 object modules that are main programs rather than subprograms. The format of this line is:

  ```
  ;COBMAIN
  ```

- Commercial Instruction Set (CIS) ID Line. This line is required only if your COBOL program was compiled with the /CODE:CIS switch. The /CODE:CIS switch can be implicitly or explicitly specified. The format of this line is:

  ```
  ;CIS
  ```

Consider the following examples:

Example 1:

MACRO program START.OBJ is a main program in a task consisting of a main program and several subprograms. The /CODE:CIS switch was specified at compile time. The SKL file to be hand-generated is:

```
;COBOBJ=START.OBJ
;COBMAIN
;CIS
```

Example 2:

MACRO subprogram SUBX.OBJ is to be part of a task image consisting of several COBOL-81 subprograms and a COBOL-81 main program. The /CODE:NOCIS switch was specified at compile time. The SKL file to be hand-generated is:

```
;COBOBJ=SUBX.OBJ
```

To activate a COBOL-81 subprogram, a non-COBOL-81 calling program must contain the equivalent of a COBOL-81 CALL statement. If data is being passed to the COBOL-81 subprogram, program register R5 must be set to the address of an argument list. The argument list must contain pointers to the data being passed. (See Figure 6-7, Argument List Format, in Section 6.3.1.3)

A non-COBOL-81 subprogram, to be activated by a COBOL-81 program, must contain the equivalent of the COBOL-81 PROGRAM-ID statement and the EXIT PROGRAM statement. If data is being passed, the non-COBOL-81 subprogram can access that data through program register R5. The following sections further describe how to include non-COBOL object modules in a task.

6.3.1 MACRO Programs and COBOL-81 Programs

6.3.1.1 Calling a MACRO Program from a COBOL-81 Program — When calling a MACRO program from a COBOL-81 program, you specify the global entry point in the MACRO program for the program name in the CALL statement. For example:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. CALLMAC.
ENVIRONMENT DIVISION.
DATA DIVISION.
```

(continued on next page)

```
WORKING-STORAGE SECTION.
01 BOFFIN PICTURE ...
01 BOMBUR PICTURE ...
01 BOFUR PICTURE ...
PROCEDURE DIVISION.
BEGIN.
        .
        .
        .
    CALL "BILBO" USING BOFFIN, BY DESCRIPTOR BOMBUR, BOFUR.
```

Note

This CALL statement specifies both calling mechanisms; BOFFIN is BY REFERENCE (the default mechanism), and both BOMBUR and BOFUR are BY DESCRIPTOR. This is because the MACRO program BILBO defines them this way.

The MACRO program, BILBO, must contain:

```
.GLOBL BILBO
 BILBO:           ;entry point - equivalent to PROGRAM-ID
        .
        .
        .
   RTS PC         ;return point - equivalent to EXIT PROGRAM
```

If there are any arguments to be passed to the called program (BOFFIN, BOMBUR, and BOFUR in this example), these arguments can be accessed through program register R5.

6.3.2 Calling a COBOL-81 Program from a MACRO Program

When calling a COBOL-81 subprogram from a MACRO program you use the command:

```
JSR PC,subprogram-name
```

where:

subprogram-name is the first six characters of the COBOL-81 program-name.

For example, if the MACRO program contains:

```
.GLOBL  FRODO
          .
          .
          .
     MOV #ARGLST,R5      ;point R5 to argument list
     JSR PC,FRODO        ;subprogram call statement
```

The COBOL-81 subprogram contains:

```
PROGRAM-ID. FRODO.

LINKAGE SECTION.

01 BOFFIN PICTURE ...

01 BOMBUR PICTURE ...

PROCEDURE DIVISION USING BOFFIN, BOMBUR.

      .
      .
      .

   EXIT PROGRAM.
```

Note

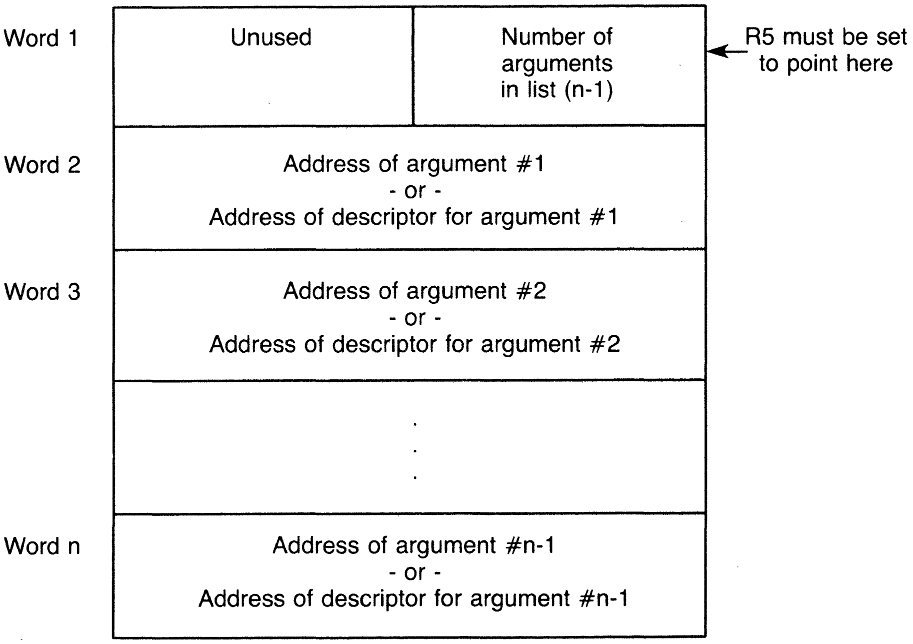
All calls to COBOL-81 subprograms from non-COBOL-81 programs specify the BY REFERENCE (default) mechanism. Non-COBOL-81 programs cannot specify other passing mechanisms.

The MACRO program in this example has set register R5 to point to the argument list expected by the COBOL-81 program. The COBOL-81 OTS will use R5 to access the passed arguments BY REFERENCE only.

6.3.3 Using the Argument Address List

The COBOL-81 compiler generates one Argument Address List for each CALL statement. Its general format is shown in Figure 6-7.

Figure 6-7: Argument Address List General Format



Word 1	Unused	Number of arguments in list (n-1)
Word 2	Address of argument #1 - or - Address of descriptor for argument #1	
Word 3	Address of argument #2 - or - Address of descriptor for argument #2	
	. . .	
Word n	Address of argument #n-1 - or - Address of descriptor for argument #n-1	

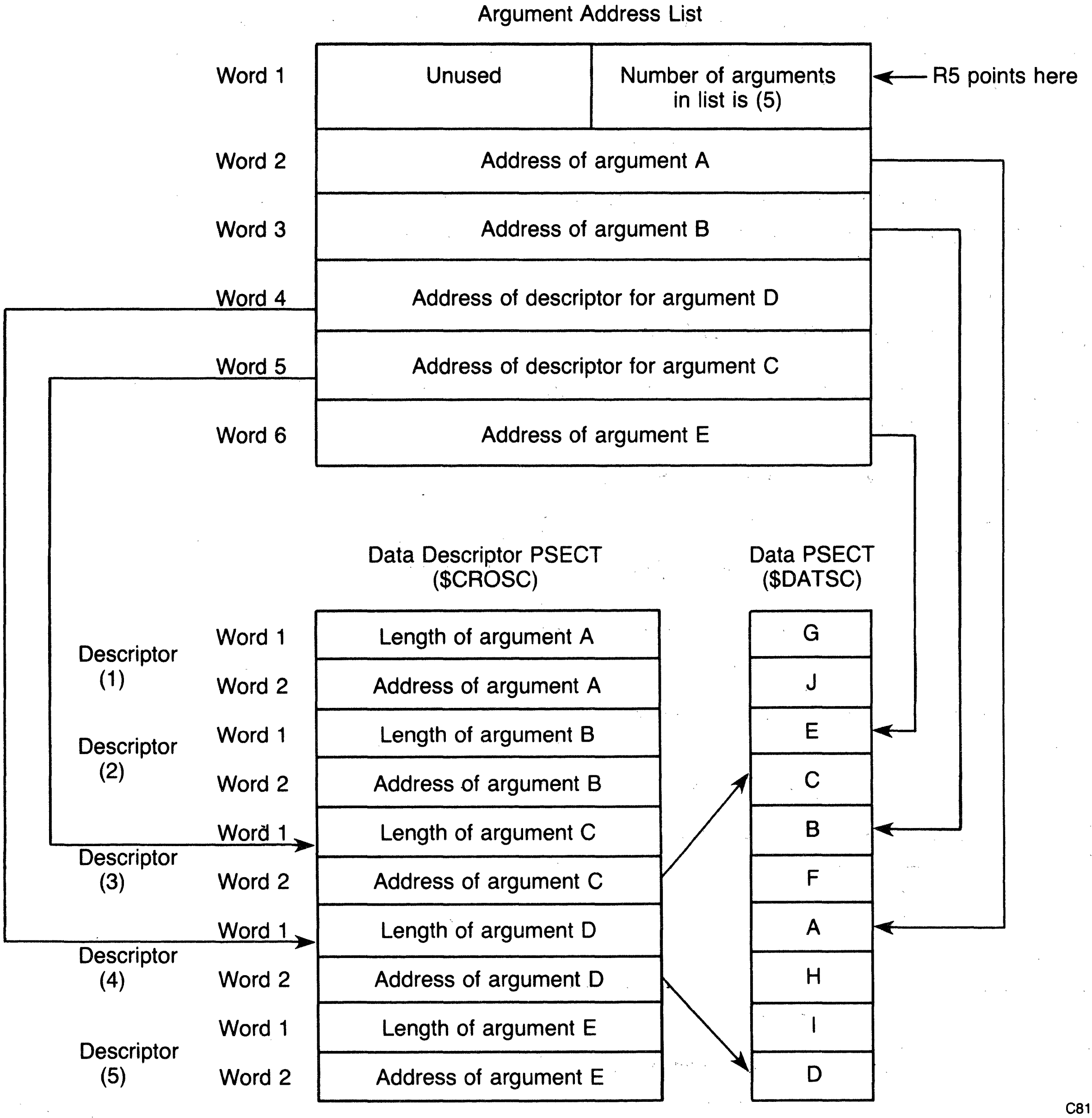
C81ART-10012-26

The sequence of arguments in the Argument Address List corresponds to the sequence of arguments in the USING phrase. For example, a COBOL-81 program calls the MACRO program MACK:

```
CALL "MACK" USING A, B, BY DESCRIPTOR D, C,
                  BY REFERENCE E.
```

MACK can then access five data items in the COBOL-81 program through the address stored in program register R5. MACK accesses a specific data item by selecting the appropriate Word in the Argument Address List. If it references Word (5) in the Argument Address List, the address of the *descriptor* for C in the COBOL-81 program's Data Descriptor PSECT (C is called BY DESCRIPTOR) is made available. C's length (Word 1) and argument address (Word 2) can then be determined by examining Descriptor (3). If it references Word (2) in the Argument Address List, the address of A in the COBOL-81 program's Data PSECT (A is called BY REFERENCE, the default) is made available. Figure 6-8 shows the Argument Address List.

Figure 6-8: Sample Argument Address List



C81ART-10013-50

Appendix A
Debugger Error Messages

This appendix lists the COBOL-81 Symbolic Debugger error messages and their explanations. Italicized words in the error message substitute for the actual numbers or data-names that appear on the terminal during a debugging session.

Some messages refer to RMS error codes. See the *RMS-11 User's Guide* for an explanation of these codes.

1 **No more work space available for synonym definitions.**

You may not execute any more DEFINE commands.

2 **No more free memory. Please delete some breakpoints or synonyms.**

The Debugger has a limited amount of memory in which to define breakpoints and synonyms.

3 **Line number *x* does not occur in "*module*."**

4 **Line number *x* in "*module*" does not contain a statement.**

5 **RMS error *x* occurred while Debugger tried to access its work file.**

6 **Symbol file contains an illegal ISD record type (*type-number*).**

The symbol file contains illegal information that has been generated by the compiler or by the Task Builder. Report all displayed information to local support or submit an SPR with the program source.

7 **The Debugger cannot find a symbol (STB) file corresponding to *filename*.**

8 **RMS error *x* occurred while Debugger tried to open *filename*.**

9 **RMS error *x* occurred while Debugger tried to access *filename*.**

10 **"*Filename*" is not a valid file specification.**

The filename you have typed contains illegal characters, is incorrectly terminated, or is incorrectly formatted.

11 **Time stamp in *filename* does not match the stamp in the task image.**

Find the correct symbol file (the one that was created with the task image).

12 **A module named *"module"* is already being used by the Debugger.**

13 **Symbol file contains an illegal ISD item code (*code-number*).**

The symbol file contains illegal information generated by the compiler or Task Builder. Report all displayed information to local support or submit an SPR with the program source.

14 **Symbol file contains illegal correlation information.**

The symbol file contains illegal information generated by the compiler or Task Builder. Report all displayed information to local support or submit an SPR with the program source.

15 **The data-name used in this command is ambiguous.**

You have used a name in your program that is used elsewhere in the program for a different entity.

16 **Illegal structure information for symbol *"symbol."***

There has been an internal error. Report all displayed information to local support or submit an SPR with the program source.

17 **RMS error *x* occurred while Debugger tried to open its work file.**

18 **RMS error *x* occurred while Debugger tried to access its work file.**

19 **RMS error *x* occurred while Debugger tried to access its symbol (STB) file.**

20 **A breakpoint already exists at *position.***

21 **No breakpoint is currently set at *position.***

Use SHOW BREAKPOINTS to find out where breakpoints are currently set.

22 **A command line cannot be longer than 200 characters.**

31 **The word "BREAKPOINT" must follow "CANCEL" in this command.**

32 **Please specify either a position or "ALL" after "BREAKPOINT".**

33 **Please specify a data-name in this command.**

34 **Please specify either a position or data-name in this command.**

35 ***"Synonym"* has already been defined as a synonym.**

Use SHOW SYNONYMS to get a list of currently recognized synonyms and their actual names.

36 ***"Synonym"* is already in use as a paragraph or section name.**

Use SHOW SYNONYMS to get a list of currently recognized synonyms and their actual names.

37 ***"Synonym"* is already in use as a data-name.**

Use SHOW SYNONYMS to get a list of currently recognized synonyms and their actual names.

38 **The word "BREAKPOINT" must follow "SET" in this command.**

39 **Please specify a position in this command.**

40 **Please specify a data-name in this command.**

41 **Please specify either an integer or "ALWAYS" after "PROCEED".**

42 **"*String*" is not a currently defined synonym.**

Use SHOW SYNONYMS to get a list of the currently recognized synonyms and their actual names.

43 **This task is too large for use with the Debugger.**

44 **No room on the system device to create the Debugger work file.**

45 **This command is incorrectly terminated by "*string*".**

Either the command was terminated by something other than <CR><LF>, or the end of the command was reached before the end of the line. When this error occurs, the Debugger displays the part of the command line it could not recognize, and executes the command anyway (unless it is STOP or PROCEED).

46 **This is not a valid Debugger command.**

47 **"*String*" is not a data-name.**

48 **Too many subscripts specified for this item.**

49 **Subscript ranges cannot be specified here.**

50 **Subscripts cannot be specified here.**

51 **Command specifies an invalid module name.**

52 **The module named "*string*" does not exist.**

53 **"*String*" is already being used by the Debugger.**

54 **"*String*" is neither a data-name nor a synonym.**

55 **Command specifies an illegal section or paragraph name.**

56 **Please specify an integer as a line number in this command.**

57 **Only one module name can be specified in this command.**

58 **Command specifies an illegal subscript.**

59 **"*String*" is not a valid receiving field for a MOVE operation.**

60 **The nonnumeric literal in a MOVE cannot be longer than 80 characters.**

61 **A nonnumeric literal must be terminated with a quotation mark.**

62 **Only a nonnumeric literal can be moved to an alphanumeric item.**

63 Only a numeric literal can be moved to a numeric item.

64 Debugger cannot find Help file "*filename*"; inform system manager.

65 RMS error *x* occurred while Debugger tried to open Help file ("*filename*").

66 RMS error *x* occurred while Debugger tried to access the Help file.

67 RMS error *x* occurred while Debugger tried to access the Help file.

68 The numeric value in this command is specified incorrectly.

69 A numeric literal cannot have more than 18 digits.

70 The subscript value specified is out of the legal range.

71 Command is missing either a position, data-name, or number.

72 Command is missing either a numeric or nonnumeric literal.

73 There is no current data-name; this command must specify one.

74 Either "BREAKPOINTS" or "SYNONYMS" must follow "SHOW".

75 Please specify a position instead of "*string*".

76 The synonym was already defined as being subscripted.

77 Debugger cannot read symbol (STB) file; inform system manager.

78 Command specifies too few subscripts for *data-name*.

79 "PROCEED ALWAYS" cannot be specified unless the "DISPLAY" option precedes it.

80 The numeric value in this command must be > 0.

81 The numeric value in this command must be < 65536.

82 A synonym cannot be used as a qualifier.

83 A module name is illegal with a qualifier.

84 The position used in this command is ambiguous.

85 The qualified name cannot be found.

86 Maximum number of qualifiers exceeded.

87 A Synonym cannot be qualified.

101 Number of logical units assigned to resulting task exceeds system maximum.

Refer to Part IV, Appendix B, Logical Unit Number (LUN) Assignments, for the LUN assignment restrictions placed on a COBOL-81 program.

Contents

Page

PART III

Chapter 1 Numeric Character Handling

1.1 How the Compiler Stores Numeric Data . . . 1-1

1.1.1 COMP and COMP SYNC Usage . . . 1-2
1.1.2 COMPUTATIONAL-3 Usage . . . 1-5
1.1.3 DISPLAY Usage . . . 1-6

1.2 Decimal Scaling Position . . . 1-6
1.3 Sign Conventions . . . 1-6

1.3.1 Sign Storage for COMP and COMP SYNC Items . . . 1-7
1.3.2 Sign Storage for COMP-3 Items . . . 1-7
1.3.3 Sign Storage for DISPLAY Items . . . 1-7

1.4 Illegal Values in Numeric Items . . . 1-9
1.5 Testing Numeric Items . . . 1-9

1.5.1 Numeric Relation Tests . . . 1-9
1.5.2 Numeric Sign Tests . . . 1-10
1.5.3 Numeric Class Tests . . . 1-10

1.6 Using the MOVE Statement . . . 1-11

1.6.1 Group Moves . . . 1-11
1.6.2 Elementary Numeric Moves . . . 1-11
1.6.3 Elementary Numeric Edited Moves . . . 1-13
1.6.4 Common Move Errors . . . 1-14

1.7 Using the Arithmetic Statements . . . 1-15

1.7.1 Intermediate Results . . . 1-15
1.7.2 Binary Truncation of COMP SYNC and COMP Items . . . 1-15
1.7.3 Using the ROUNDED Phrase . . . 1-16
1.7.4 Using the SIZE ERROR Phrase . . . 1-17
1.7.5 Using the GIVING Phrase . . . 1-18
1.7.6 Multiple Operands in ADD and SUBTRACT Statements . . . 1-18
1.7.7 Common Errors in Arithmetic Statements . . . 1-19

1.8 Arithmetic Expression Processing . . . 1-20

Chapter 2 Nonnumeric Character Handling

2.1 Data Organization . . . 2-2

2.1.1 Group Items . . . 2-2
2.1.2 Elementary Items . . . 2-2

2.2 Special Characters . . . 2-2
2.3 Testing Nonnumeric Items . . . 2-3

2.3.1 Relation Tests of Nonnumeric Items . . . 2-3

2.3.1.1 Classes of Data . . . 2-4
2.3.1.2 Comparison Operations . . . 2-4

2.3.2 Class Tests for Nonnumeric Items . . . 2-5

2.4 Data Movement . . . 2-5

2.5 Using the MOVE Statement 2-6
2.5.1 Group Moves 2-7
2.5.2 Elementary Moves 2-7
2.5.2.1 Edited Moves 2-8
2.5.2.2 Justified Moves 2-9
2.5.3 Multiple Receiving Items 2-9
2.5.4 Subscripted Moves 2-10
2.5.5 Common Nonnumeric Item MOVE Statement Errors 2-10
2.5.6 Using the MOVE CORRESPONDING Statement for Nonnumeric Items 2-10
2.6 Concatenating Data Using the STRING Statement. 2-11
2.6.1 Multiple Sending Items 2-11
2.6.2 Using the POINTER Phrase 2-12
2.6.3 Using the DELIMITED BY Phrase 2-12
2.6.4 Using the OVERFLOW Phrase 2-14
2.6.5 Subscripted Items in STRING Statements 2-15
2.6.6 Common STRING Statement Errors 2-17
2.7 Separating Data Using the UNSTRING Statement 2-18
2.7.1 Multiple Receiving Items 2-18
2.7.2 Controlling Moved Data Using the DELIMITED BY Phrase 2-20
2.7.2.1 Multiple Delimiters 2-23
2.7.3 Counting UNSTRING Characters Using the COUNT Phrase 2-24
2.7.4 Saving UNSTRING Delimiters Using the DELIMITER Phrase 2-25
2.7.5 Controlling UNSTRING Scanning Using the POINTER Phrase 2-26
2.7.6 Counting UNSTRING Receiving Items Using the TALLYING Phrase 2-27
2.7.7 Exiting an UNSTRING Statement Using the OVERFLOW Phrase . . . 2-28
2.7.8 Using Subscripted Items in UNSTRING Statements 2-29
2.7.9 Common UNSTRING Statement Errors 2-31
2.8 Examining and Replacing Characters Using the INSPECT Statement 2-31
2.8.1 Restricting Data Inspection Using the BEFORE/AFTER Phrase. 2-32
2.8.2 Implicit Redefinition 2-33
2.8.3 Examining the INSPECT Operation 2-34
2.8.3.1 Setting the Scanner 2-35
2.8.3.2 Active/Inactive Arguments 2-35
2.8.3.3 Finding an Argument Match 2-36
2.8.4 Subscripted Items in INSPECT Statements 2-37
2.8.5 The TALLYING Phrase 2-37
2.8.5.1 The Tally Counter 2-38
2.8.5.2 The Tally Argument. 2-38
2.8.5.3 The Tally Argument List 2-39
2.8.5.4 Interference in Tally Argument Lists 2-40
2.8.6 Using the Replacing Phrase 2-43
2.8.6.1 The Search Argument 2-44
2.8.6.2 The Replacement Value 2-45
2.8.6.3 The Replacement Argument 2-45
2.8.6.4 The Replacement Argument List 2-45
2.8.6.5 Interference in Replacement Argument Lists 2-46
2.8.7 Common INSPECT Statement Errors 2-47

Chapter 3 Table Handling

3.1 Introduction 3-1
3.2 Defining Tables 3-1
3.2.1 Defining Fixed-Length, One-Dimensional Tables 3-2
3.2.2 Defining Fixed-length, Multidimensional Tables 3-3
3.2.3 Defining Variable-Length Tables 3-5
3.2.4 Storage Allocation for Tables 3-5
3.2.4.1 Tables without COMP, COMP SYNC, or USAGE INDEX Items 3-5
3.2.4.2 Tables with COMP or COMP SYNC Items 3-6
3.3 Initializing Values of Table Elements 3-8
3.4 Accessing Table Elements 3-10
3.4.1 Subscripting 3-10
3.4.2 Subscripting with Literals 3-10
3.4.3 Subscripting with Data-Names 3-12
3.4.4 Subscripting with Indexes 3-12
3.4.5 Relative Indexing. 3-13
3.4.6 Index Data Items. 3-13
3.4.7 Assigning Index Values Using the SET Statement 3-13
3.4.7.1 Assigning an Integer Index Value with a SET Statement 3-13
3.4.7.2 Incrementing an Index Value with the SET Statement . . . 3-14
3.4.8 Identifying Table Elements Using the SEARCH Statement 3-14
3.4.8.1 Implementing a Sequential Search 3-14
3.4.8.2 Implementing a Nonsequential (Binary) Search 3-15

Chapter 4 Data Handling Optimization

4.1 Numeric Data Representation 4-1
4.1.1 Scaling and Mixing Data Types 4-2
4.1.2 Significant Digits. 4-2
4.1.3 Indexing Instead of Subscripting 4-3
4.1.4 Avoid Using Decimal Truncation. 4-3
4.2 Procedure Division Statements 4-4
4.2.1 ADD, SUBTRACT, MULTIPLY and DIVIDE Instead of COMPUTE. . . . 4-4
4.2.2 GO TO DEPENDING ON Instead of IF, GO TO 4-4
4.2.3 SEARCH ALL Instead of SEARCH. 4-5

Examples

1-1 Sample Record Description 1-2
1-2 Sample Record Description 1-3
1-3 Sample Record Description 1-3
1-4 Sample Record Description 1-4
2-1 Item Concatenation. 2-6
2-2 Sample Record Description Using the MOVE CORRESPONDING Statement . . 2-11
2-3 Sample Overflow Condition 2-15

3-1 One-Dimensional Table . . . 3-2
3-2 Multiple Data Items in a One-Dimensional Table . . . 3-2
3-3 Defining a Table with an Index and an Ascending Search Key . . . 3-3
3-4 Defining a Two-Dimensional Table . . . 3-3
3-5 Defining a Three-Dimensional Table . . . 3-4
3-6 Defining a Variable-Length Table . . . 3-5
3-7 Sample Record Description Defining a Table . . . 3-5
3-8 Record Description Containing a COMP SYNC Item . . . 3-6
3-9 Adding an Item Without Changing the Table Size . . . 3-7
3-10 Adding 3 Bytes That Adds 4 Bytes to the Element Length . . . 3-7
3-11 Initializing Tables . . . 3-8
3-12 Initializing Mixed Usage Items . . . 3-9
3-13 Initializing Alphanumeric Items . . . 3-9
3-14 Using a Literal Subscript to Access a Table . . . 3-10
3-15 Subscripting a Multidimensional Table . . . 3-11
3-16 Subscripting with Index-Name Items . . . 3-12
3-17 A Serial Search . . . 3-18
3-18 Using SEARCH and Varying an Index Other Than the First Index . . . 3-18
3-19 Using SEARCH and Varying an Index Data Item . . . 3-18
3-20 Using SEARCH and Varying an Index Not Associated with the Target Table . . . 3-19
3-21 Doing a Serial Search Without Using the Varying Phrase . . . 3-19
3-22 A Multiple-Key Binary Search . . . 3-20

Figures

1-1 Word and Byte Representation in Storage for Example 1-1 . . . 1-2
1-2 Storage Allocation for COMP Items for Example 1-2 . . . 1-3
1-3 Storage Allocation for Two-Word COMP SYNC Items for Example 1-3 . . . 1-4
1-4 Storage Allocation for a Four-Word COMP SYNC Item for Example 1-4 . . . 1-4
1-5 Storage of COMP-3 Data Items . . . 1-5
1-6 Sign Storage in COMP-3 Items . . . 1-7
1-7 Results of the ROUNDED Phrase . . . 1-16
1-8 Results of the ROUNDED Phrase . . . 1-17
2-1 Sequence of Subscript Evaluation . . . 2-30
2-2 Sample INSPECT Statement . . . 2-35
2-3 The Replacement Argument . . . 2-45
3-1 Organization of the One-Dimensional Table in Example 3-1 . . . 3-2
3-2 Organization of Multiple Data Items in a One-Dimensional Table . . . 3-2
3-3 Organization of a Table with an Index and an Ascending Search Key . . . 3-3
3-4 Organization of a Two-Dimensional Table . . . 3-4
3-5 Organization of a Three-Dimensional Table . . . 3-4
3-6 Memory Map for Example 3-7 . . . 3-6
3-7 Memory Map for Example 3-8 . . . 3-6
3-8 Memory Map for Example 3-9 . . . 3-7
3-9 Memory Map for Example 3-10 . . . 3-8
3-10 Memory Map for Example 3-11 . . . 3-8
3-11 Memory Map for Example 3-12 . . . 3-9
3-12 Memory Map for Example 3-13 . . . 3-10
3-13 Subscripting with Data-Names . . . 3-12
3-14 Using SEARCH to Access This Sample Table . . . 3-17

Tables

1-1 Memory Allocation for COMP Items 1-2
1-2 Memory Storage of COMP Items 1-5
1-3 Overpunched Character for Sign and Digit Combinations 1-8
1-4 Sign Tests 1-10
1-5 Numeric Editing 1-14
1-6 Maximum Values with Decimal and Binary Truncation 1-16
1-7 Rules for Intermediate Result Size. 1-22
2-1 Relational Operator Descriptions 2-3
2-2 Nonnumeric Elementary Moves 2-7
2-3 Data Movement with Editing Symbols. 2-8
2-4 Data Movement with No Editing 2-9
2-5 Results of Sample Overflow Statements 2-15
2-6 Results of Sample Delimiter Subscripts 2-17
2-7 Values Moved into the Receiving Items Based on the Sending Item Value . . . 2-19
2-8 Handling a Short Sending Item 2-19
2-9 Results of Delimiting with an Asterisk 2-20
2-10 Results of Delimiting Multiple Receiving Items 2-21
2-11 Results of Delimiting with Two Asterisks. 2-22
2-12 Results of Delimiting with ALL Asterisks 2-22
2-13 Results of Delimiting with ALL Double Asterisks 2-23
2-14 Results of Multiple Delimiters 2-24
2-15 Matching Delimiter Characters to Characters in a Field 2-33
2-16 Values Resulting from Implicit Redefinition 2-34
2-17 Relationship Between INSPECT Argument, Delimiter, Item Value, and Argument Active Position 2-36
2-18 LEADING Delimiter of the Inspection Operation 2-39
2-19 Results of the Scan with Separate Tallies. 2-40
3-1 Subscripting Rules for a Multidimensional Table 3-11
4-1 Relative Efficiency of COBOL-81 Numeric Data Types 4-1

Chapter 1
Numeric Character Handling

This chapter describes how COBOL-81 stores, represents, moves, and manipulates numeric data.

1.1 How the Compiler Stores Numeric Data

Understanding how data is stored is particularly important when you are defining data items that will participate in group moves or be the subject of a REDEFINES clause. When moving a complex record consisting of several levels of subordination, you should be sure that the receiving item is large enough to prevent data truncation. You can also use the concepts of data storage to your advantage to minimize storage space, particularly when the data file is very large. The storage considerations applicable to table handling are discussed in Chapter 3.

For each numeric data item, COBOL-81 stores the numeric value, a scaling factor (if a V or a P appears in the PICTURE), and a sign (if an S appears in the PICTURE). Each of these subjects is discussed separately in the following sections.

The USAGE clause of a numeric data item specifies the data's internal format in storage. COBOL-81 has three formats for numeric data storage:

- COMPUTATIONAL (COMP)
- COMPUTATIONAL-3 (COMP-3)
- DISPLAY

When you do not specify a usage in a PICTURE clause, the default usage is DISPLAY. (The special case of INDEX usage for tables is discussed in Chapter 3.)

The basic unit of data storage is the byte. Depending on the item usage, stored data is aligned on one-byte, one-, two-, or four-word boundaries or on multiples of one-, two-, or four-word boundaries when several items make up the record. A word is comprised of 2 bytes.

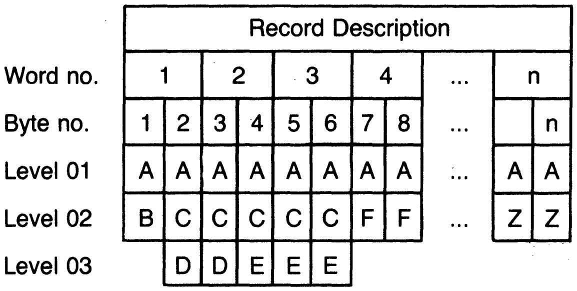
Alignment is always described in relation to the beginning of the record, which is always defined by an 01 level number. All items subordinate to the 01 level are part of the record.

Example 1-1 and Figure 1-1 show the basic representation of words and bytes in storage.

Example 1-1: Sample Record Description

```
01  A.
    02 B        PIC 9.
    02 C.
       03  D    PIC 99.
       03  E    PIC 999.
    02 F        PIC 99.
    .
    .
    .
    02 Z        PIC 99.
```

Figure 1-1: Word and Byte Representation in Storage for Example 1-1



	Record Description										
Word no.	1		2		3		4		...	n	
Byte no.	1	2	3	4	5	6	7	8	...		n
Level 01	A	A	A	A	A	A	A	A	...	A	A
Level 02	B	C	C	C	C	C	F	F	...	Z	Z
Level 03		D	D	E	E	E					

C81ART-10014-20

1.1.1 COMP and COMP SYNC Usage

COMP and COMP SYNC data items are stored in standard binary format as a binary value and an optional sign. Sign storage is discussed in Section 1.3.1. Depending on the size of the item defined by the PICTURE clause, both COMP and COMP SYNC items are stored as one, two, or four words as shown in Table 1-1.

Table 1-1: Memory Allocation for COMP Items

PICTURE Range	Storage Allocated
S9 to S9(4)	1 word (2 bytes)
S9(5) to S9(9)	2 words (4 bytes)
S9(10) to S9(18)	4 words (8 bytes)

Although both COMP and COMP SYNC items require the same number of storage words, the storage alignment in a record description for each item is quite different. This can lead to significant differences in the total amount of storage space required for a record description. COMP SYNC items are aligned on a one-, two-, or four-word boundary depending on the number of decimal digits specified in the PICTURE clause. The alignment boundary is the same as the number of words required for storage. COMP items, however, are always aligned on a one-word boundary regardless of the item size.

Note

Data defined as COMP SYNC usage in COBOL-81 is compatible with data defined as COMP SYNC usage in VAX–11 COBOL.

Figures 1-2, 1-3 and 1-4 show the difference in total storage space required to store a record defined for COMP and COMP SYNC usage. In Figures 1-2 and 1-3 ITEM-C requires two words of storage because the item is defined as containing seven decimal digits. For the record defined as COMP usage, ITEM-C must start at a one-word boundary. Thus, the compiler adds one fill byte to the first word. The total storage requirement is three words. The record defined as COMP SYNC usage, however, requires a total of four words of storage because the SYNC clause requires that ITEM-C start on a two-word boundary. (There would be no difference if ITEM-C required only one word of storage.)

Figure 1-4 is similar to Figures 1-2 and 1-3; however, ITEM-C now defines 12 decimal digits. Because of its increased size, ITEM-C requires four words of storage and must start on a four-word boundary. Notice the seven implicit fill bytes added by the compiler to align ITEM-C on a four-word boundary.

Any item that is to be a receiving item for RECORD-A must be defined so that its size is large enough to accommodate the subordinate items plus any fill bytes.

Example 1-2: Sample Record Description

```
01  RECORD-A.
    02  ITEM-B    PIC X.
    02  ITEM-C    PIC 9(7) COMP.
```

Figure 1-2: Storage Allocation for COMP Items for Example 1-2

<table>
<tr><td></td><td colspan="6">Record Description</td></tr>
<tr><td>Word no.</td><td colspan="2">1</td><td colspan="2">2</td><td colspan="2">3</td></tr>
<tr><td>Byte no.</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td></tr>
<tr><td>Level 01</td><td>A</td><td>A</td><td>A</td><td>A</td><td>A</td><td>A</td></tr>
<tr><td>Level 02</td><td>B</td><td>f</td><td>C</td><td>C</td><td>C</td><td>C</td></tr>
</table>

Legend: f = Fill byte added by the compiler for data-item alignment

C81ART-10015-18

Example 1-3: Sample Record Description

```
01  RECORD-A.
    02  ITEM-B   PIC X.
    02  ITEM-C   PIC 9(7) COMP SYNC.
```

Figure 1-3: Storage Allocation for Two-Word COMP SYNC Items for Example 1-3

	Record Description							
Word no.	1		2		3		4	
Byte no.	1	2	3	4	5	6	7	8
Level 01	A	A	A	A	A	A	A	A
Level 02	B	f	f	f	C	C	C	C

Legend: f = Fill bytes added by the compiler for data-item alignment

C81ART-10016-18

Example 1-4: Sample Record Description

```
01   RECORD-A.
     02  ITEM-B   PIC X.
     02  ITEM-C   PIC 9(12)
                  COMP SYNC.
```

Figure 1-4: Storage Allocation for a Four-Word COMP SYNC Item for Example 1-4

	Record Description															
Word no.	1		2		3		4		5		6		7		8	
Byte no.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Level 01	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
Level 02	B	f	f	f	f	f	f	f	C	C	C	C	C	C	C	C

Legend: f = Fill bytes added by the compiler for data-item alignment

C81ART-10017-20

Table 1-2 shows the memory storage of COMP data items.

Table 1-2: Memory Storage of COMP Items

addressed word	
high byte (2)	low byte (1)

one-word COMP data item

addressed word		**next word**	
high byte (2)	low byte (1)	high byte (4)	low byte (3)

two-word COMP data item

addressed word		**next word**		**next word**		**next word**	
high byte (2)	low byte (1)	high byte (4)	low byte (3)	high byte (6)	low byte (5)	high byte (8)	low byte (7)

four-word COMP data item

1.1.2 COMPUTATIONAL-3 Usage

COMP-3 data items are stored in packed-decimal format with an optional sign. Sign storage is discussed in Section 1.3.2. COMP-3 items are stored two decimal digits per byte. The 4 rightmost bits of the rightmost byte are reserved for the sign. If there are an even number of digits in the item, the leftmost 4 bits of the leftmost byte contain a zero.

Figure 1-5 represents the storage of COMP-3 items with one, two, and three digits. The blank portion of the byte is reserved for the sign.

Figure 1-5: Storage of COMP-3 Data Items

1st byte	
5	

PICTURE 9
value: 5

1st byte		2nd byte	
0	3	2	

PICTURE 9(2)
value: 32

1st byte		2nd byte	
2	6	2	

PICTURE 9(3)
value: 262

C81ART-10018-14

1.1.3 DISPLAY Usage

A numeric item with DISPLAY usage is stored as an ASCII character string with an optional sign. DISPLAY items are stored one digit per byte. Sign storage is discussed in Section 1.3.3.

1.2 Decimal Scaling Position

The assumed decimal scaling position, or scaling factor, is not stored as part of an actual numeric value. However, it is used to control operations on numeric data items.

The maximum size of all COBOL-81 numeric items is 18 decimal digits, regardless of the decimal scaling position. In the following example, both NUM-1 and NUM-2 represent COMP-3 items of maximum size:

```
03  NUM-1 PIC S9(18)        USAGE IS COMP-3.
03  NUM-2 PIC S9(6)V9(12)   USAGE IS COMP-3.
```

The following example shows how the scaling factor is used to control a numeric operation:

```
01 ORDER-PRICE PIC 99V99 COMP VALUE 12.34.
```

COBOL-81 stores this item as a one-word binary number. The word contains the integer value 1234 and another location contains the scaling factor, which is 2 in this example. The scaling factor indicates that this integer has two decimal positions to the right of the implied decimal point. Thus, the Object Time System (OTS) knows that the stored binary integer is 100 times larger than the programmer intends it to be.

Suppose the compiler subsequently encounters this statement:

```
ADD 1 TO ORDER-PRICE.
```

It then adds 1 to the 1234 stored in ORDER-PRICE. The OTS, however, scales the literal 1 up by two decimal places and adds the resultant literal, 100, to the number in ORDER-PRICE. Thus, after the ADD operation, ORDER-PRICE contains the new value 1334, which is actually 13.34 with the stored decimal scaling position.

Thus, the COBOL-81 compiler and OTS manipulate the data in COMP-3 and DISPLAY data items in much the same way. The usages have exactly the same accuracy and precision and can be freely mixed in a program. The only advantage of specifying a binary (COMP) or packed-decimal (COMP-3) usage over a DISPLAY usage is that they reduce the space required for most numbers and can speed up the execution of arithmetic statements.

1.3 Sign Conventions

All COBOL-81 numeric items can be signed or unsigned. However, all COBOL-81 arithmetic operations yield signed results. If you store a signed result in an unsigned item, only the absolute value is stored. Thus, unsigned items only contain the value zero or positive values. The way COBOL-81 stores signed results in signed data items depends on the usage and the presence of the SIGN clause. Each usage type is discussed in the following sections.

In general, do not use unsigned numeric items. They are usually a source of programming errors and are handled less efficiently than signed numeric items.

1.3.1 Sign Storage for COMP and COMP SYNC Items

Both COMP types are stored in two's complement format with the sign represented by the high-order bit.

1.3.2 Sign Storage for COMP-3 Items

The results of arithmetic operations are stored in COMP-3 usage items in packed decimal format. The rightmost four bits of the last byte are reserved for the sign.

Signs resulting from operations in which the receiving item usage is COMP-3 are:

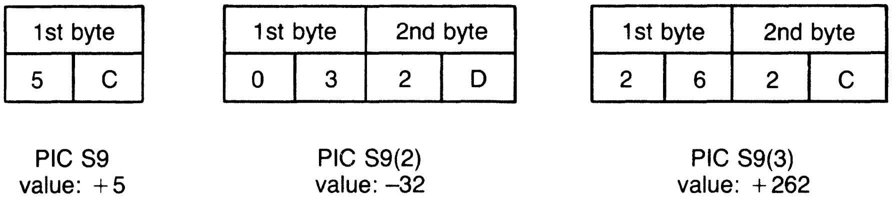
Positive sign:	binary 1100, hexadecimal C
Negative sign:	binary 1101, hexadecimal D
Unsigned:	binary 1111, hexadecimal F

The following signs are recognized as being valid. However, they do not result from program operations.

Positive signs:	binary 1010, hexadecimal A
	binary 1110, hexadecimal E
Negative signs:	binary 1011, hexadecimal B

Figure 1-6 represents the storage of COMP-3 signed items of one, two, and three digits.

Figure 1-6: Sign Storage in COMP-3 Items



C81ART-10019-12

1.3.3 Sign Storage for DISPLAY Items

The position and format of the sign storage for DISPLAY items depends on the contents of the SIGN clause. SIGN LEADING and SIGN TRAILING clauses result respectively in left overpunched and right overpunched formats. Overpunching is the result of the sign sharing a byte with a digit. For SIGN LEADING, the sign and the most significant digit (leftmost digit) share a byte, while the sign and the least significant digit (rightmost digit) share a byte for SIGN TRAILING. When a signed DISPLAY item description contains no SIGN clause the default is SIGN TRAILING. Table 1-3 shows the overpunched characters resulting from all combinations of signs and digit values. These are the actual ASCII characters that would be printed. Where more than one character appears, the first is the character generated as the result of machine operations.

Table 1-3: Overpunched Character for Sign and Digit Combinations

Digit Value	0		1		2		3		4		5		6		7		8		9	
Sign	+	–	+	–	+	–	+	–	+	–	+	–	+	–	+	–	+	–	+	–
Overpunched Char	{,[?,0	},] ;,!	A 1	J	B 2	K	C 3	L	D 4	M	E 5	N	F 6	O	G 7	P	H 8	Q	I 9	R

C81ART-10020-16

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

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

The following program example shows the results obtained when you use the SIGN LEADING and the SIGN TRAILING clauses and the default when you use no SIGN clause:

```
01  A PIC S999 SIGN LEADING.
01  B PIC S999 SIGN TRAILING.
01  C PIC S999.
    .
    .
    .
MOVE +123 TO A, B, C.
DISPLAY A.
DISPLAY B.
DISPLAY C.
```

```
Statement           Result

DISPLAY A            A23
DISPLAY B            12C
DISPLAY C            12C
```

When you specify the SIGN LEADING SEPARATE or the SIGN TRAILING SEPARATE clause, the sign is stored in a separate byte ahead of the most significant digit or after the least significant digit respectively. The actual ASCII character stored is the ASCII plus sign (octal 053) or the ASCII minus sign (octal 055).

The following program example shows the results when you specify the SIGN LEADING SEPARATE and the SIGN TRAILING SEPARATE clauses.

```
01  A PIC S999 SIGN LEADING SEPARATE.
01  B PIC S999 SIGN TRAILING SEPARATE.
    .
    .
    .
MOVE +123 TO A, B.
DISPLAY A.
DISPLAY B.
    .
    .
    .
```

```
Statement      Result

DISPLAY A      +123
DISPLAY B       123+
```

1.4 Illegal Values in Numeric Items

All COBOL-81 arithmetic operations store legal values in their result items. However, it is possible to store data in numeric items that does not conform to the data definitions of those items. For example, you can place signed values into unsigned items and place nonnumeric or improperly signed data into signed numeric display items. This can happen when you use invalid input data, redefine items, and perform group moves.

The results of arithmetic operations that use invalid data in numeric items are undefined.

1.5 Testing Numeric Items

COBOL-81 provides three kinds of tests for evaluating numeric items:

1. Relation tests that compare the item's contents to another numeric value
2. Sign tests that examine the item's sign to see if it is positive or negative
3. Class tests that inspect the item's digit positions for legal numeric values

The following sections explain these tests in detail.

1.5.1 Numeric Relation Tests

A relation test compares two numeric quantities and determines if the specified relation between them is true. For example, the following statement compares item FIELD1 to item FIELD2 and determines if the numeric value of FIELD1 is greater than the numeric value of FIELD2:

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

If the relation condition is true, the program control takes the true path of the statement.

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

The size of the items (including numeric literals) in a numeric relation test do not have to be the same. The comparison operation aligns both items on their assumed decimal positions through scaling or filling with leading or trailing zeros.

The comparison operation always compares the signs of nonzero items and considers positive items to be greater than negative items. However, since it does not compare them, positive zeros and negative zeros are equal. A negative zero could be placed in an item through redefinition of the item or a move to a group item. The operation considers unsigned numeric items to be positive.

The form of representation of the number (COMP, COMP-3, or DISPLAY usage) and the various methods of storing DISPLAY usage signs have no effect on numeric relation tests.

The results of relation tests involving illegal (nonnumeric) data in a numeric item are undefined.

1.5.2 Numeric Sign Tests

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

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

Now consider the following sign test:

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

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

Table 1-4 shows the sign tests and their equivalent relation tests as applied to FIELD1.

Table 1-4: Sign Tests

Sign Test	Equivalent Relation Test
`IF FIELD1 POSITIVE ...`	`IF FIELD1 > 0 ...`
`IF FIELD1 NOT POSITIVE ...`	`IF FIELD1 NOT > 0 ...`
`IF FIELD1 NEGATIVE ...`	`IF FIELD1 < 0 ...`
`IF FIELD1 NOT NEGATIVE ...`	`IF FIELD1 NOT < 0 ...`
`IF FIELD1 ZERO ...`	`IF FIELD1 = 0 ...`
`IF FIELD1 NOT ZERO ...`	`IF FIELD1 NOT = 0 ...`

Sign tests have no execution speed advantage over relation tests because the compiler substitutes the equivalent relation test for every correctly written sign test. (Sections 1.3 and 1.4 discuss the acceptable sign values and the treatment of illegal sign values.)

1.5.3 Numeric Class Tests

The class test inspects an item to determine if it contains numeric or alphabetic data – and uses the result to alter the program flow control. For example, the following statement determines if FIELD1 contains numeric data:

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

If the item is numeric, the test condition is true, and program control takes the true path of the statement.

Both relation and sign tests treat illegal characters in DISPLAY usage items as zeros. Both tests only determine if an item's contents are within a certain range. Therefore, certain items in newly prepared data can pass both the relation and sign tests and still contain data preparation errors.

The NUMERIC class test checks numeric or alphanumeric DISPLAY usage items for valid numeric digits.

If the item being tested contains a sign (whether carried as an overpunched character or as a separate character), the test checks it for a valid sign value. If the character position carrying the sign contains an illegal sign value, the NUMERIC class test rejects the item, and program control takes the false path of the IF statement.

The ALPHABETIC class test checks alphabetic or alphanumeric items for valid alphabetic characters and the space character. If all the character positions of the item contain ASCII characters A to Z (upper or lower case) or the space character, the item passes the ALPHABETIC class test and causes program control to take the true path of the IF statement. (For further information concerning the ALPHABETIC class test, see Part I, Chapter 2, Creating and Entering a COBOL-81 Program.)

1.6 Using the MOVE Statement

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

```
MOVE FIELD1 TO FIELD2.
```

This section considers MOVE statements as applied to numeric data items. These MOVE statements can be grouped into the following categories:

- Group moves
- Elementary moves with numeric receiving items
- Elementary moves with numeric edited receiving items

The following sections discuss each of these categories separately.

1.6.1 Group Moves

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

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

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

1.6.2 Elementary Numeric Moves

If both items of a MOVE statement are elementary items and the receiving item is numeric, the OTS considers the move to be an elementary numeric move. The sending item can be either numeric or alphanumeric. The numeric receiving item can be COMP, COMP-3 or DISPLAY usage. The elementary numeric move converts the data format of the sending item to the data format of the receiving item.

An alphanumeric sending item can be either:

- An elementary data item
- Any alphanumeric literal other than the figurative constants SPACE, QUOTE, LOW-VALUE, HIGH-VALUE, or ALL literal

The elementary numeric move accepts the figurative constant ZERO and considers it to be equivalent to the numeric literal 0. It treats alphanumeric sending items as unsigned integers of DISPLAY usage.

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

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

```
01 AMOUNT1 PIC 99V99 VALUE ZEROS.
             .
             .
             .
   MOVE 123.321 TO AMOUNT1.
```

Before execution: 00^00

After execution: 23^32

If the sending item has fewer digit positions than the receiving item, the move operation supplies zeros for all unfilled digit positions. The caret (^) indicates the stored decimal scaling position:

```
01  TOTAL-AMT PIC 999V99 VALUE ZEROS.
    .
    .
    .
    MOVE 1 TO TOTAL-AMT.
```

Before execution: 000^00

After execution: 001^00

The following statement produces the same results:

```
MOVE 001.00 TO TOTAL-AMT.
```

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

Statement	TOTAL-AMT After Execution
`MOVE 00100 TO TOTAL-AMT`	100^00
`MOVE "00100" TO TOTAL-AMT`	100^00

Literals with leading or insignificant trailing zeros have no significant advantage in space or execution speed with COBOL-81, and the zeros are often lost by decimal point alignment.

The MOVE statement's receiving item dictates how the sign will be moved. A signed DISPLAY usage receiving item causes the sign to be moved as a separate quantity. An unsigned DISPLAY usage receiving item causes no sign movement. A COMP usage receiving item; whether signed or unsigned, causes the sign to be moved; however, if the receiving item is unsigned, the OTS takes the absolute value of the sending item and stores it in the receiving item.

1.6.3 Elementary Numeric Edited Moves

The COBOL-81 OTS considers an elementary numeric move to a numeric edited receiving item to be an elementary numeric edited move. The sending item of an elementary numeric edited move can be either numeric or alphanumeric. If it is numeric, its usage can be COMP, COMP-3, or DISPLAY. The OTS treats alphanumeric sending items in numeric edited moves as unsigned DISPLAY usage integers.

The OTS considers the receiving item to be numeric edited if its PICTURE character-string contains either a BLANK WHEN ZERO clause or a combination of the following symbols:

B	Space insertion character
P	Decimal scaling position character
V	Assumed decimal point location character
Z	Leading zero suppression and space replacement character
0	Zero insertion character
9	Numeral position character
/	Slash insertion character
,	Comma insertion character
.	Decimal point insertion character
*	Leading zero suppression and asterisk replacement character
+	Positive editing sign control symbol
–	Negative editing sign control symbol
CR	Credit editing sign control symbol
DB	Debit editing sign control symbol
CS	Currency insertion symbol

A numeric edited item can contain 9, V, and P, but it also must contain one or more of the other symbols to qualify as numeric edited.

The numeric edited move operation first converts the sending item to DISPLAY usage and aligns both items on their decimal point locations. The sending item is truncated or zero filled until it has the same number of digit positions on both sides of the decimal point as the receiving item. The operation then moves the sending item to the receiving item, following the COBOL-81 editing rules.

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

- Suppress leading zeros with either spaces or asterisks
- Float a currency sign and a plus or minus sign through suppressed zeros, inserting the sign at either end of the item
- Insert zeros and spaces
- Insert commas and a decimal point

Table 1-5 illustrates several of these functions with the statement:

```
MOVE FLD-B TO TOTAL-AMT.
```

Assume that FLD-B is described as S9999V99.

Table 1-5: Numeric Editing

FLD-B	TOTAL-AMT	
	PICTURE String	**Contents After MOVE**
0023^00	ZZZZ.99	23.00
0085^9P	+ + + + .99	–85.97
1234^00	Z,ZZZ.99	1,234.00
0012^34	$,$$$.99	$12.34
0000^34	$,$$9.99	$0.34
1234^00	$$,$$$.99	$1,234.00
0012^34	$$9,999.99	$0,012.34
0012^34	$$$$,$$$.99	$12.34
0000^00	$$$,$$$.$$	
0012^3M	+ + + + .99	–12.34
0012^34	$***,***.99	$*****12.34

The currency symbol ($ or other currency sign) and the editing sign control symbols (+ –) are the only floating symbols. To float them, enter a string of two or more occurrences of the symbol, one for each character position over which you want the symbol to float.

1.6.4 Common Move Errors

The most common errors programmers make when writing MOVE statements are:

- Placing an incorrect number of replacement characters in a numeric edited item
- Moving nonnumeric data into numeric items with group moves

- Trying to float the $ or + insertion characters past the decimal point to force zero values to appear as .00 instead of spaces. Use $$.99 or .99
- Forgetting that the $ or + insertion characters require an additional position on the leftmost end that cannot be replaced by a digit, unlike the * insertion character that can be completely replaced.

1.7 Using the Arithmetic Statements

The COBOL-81 arithmetic statements allow programs to perform arithmetic operations on numeric data. The following sections explain how to use the COBOL-81 arithmetic statements.

1.7.1 Intermediate Results

Most forms of the arithmetic statements perform their operations in temporary work locations, then move the results to the receiving items, aligning the decimal points and truncating or zero filling the resultant values. This temporary work item, called the intermediate result item, has a maximum size of 18 numeric digits. The actual size of the intermediate result varies for each statement and is determined at compile time based on the sizes of the operands used by the statement.

When the compiler determines that the size of the intermediate result exceeds 18 digits, it goes to the software floating point and keeps the most significant 18 digits, bypassing leading zeros.

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

1.7.2 Binary Truncation of COMP SYNC and COMP Items

By default, COBOL-81 truncates values of COMP SYNC and COMP items according to the amount of storage allocated for them. This is called binary (as opposed to decimal) truncation.

With binary truncation, the maximum value both COMP items can contain depends on its storage allocation. With decimal truncation, the maximum value depends on the item's PICTURE character-string.

To understand the difference between binary and decimal truncation, consider the following data descriptions:

```
01  ITEMA PIC 9 COMP.
01  ITEMB PIC 9(4) COMP.
```

The PICTURE character-string of ITEMA imposes a one-digit limit on the item's value; therefore, the maximum value of ITEMA is 9. Likewise, the maximum value of ITEMB is 9999.

The compiler allocates one word (2 bytes, 16 bits) of memory for each of the above items. Therefore, both ITEMA and ITEMB can contain maximum values of (2 ** 15) – 1 or 32,767. Recall that the high order bit is reserved for the sign. Table 1-6 shows maximum values with decimal and with binary truncation.

Table 1-6: Maximum Values with Decimal and Binary Truncation

	Maximum Values	
	With Decimal Truncation	With Binary Truncation
ITEMA	9	32,767
ITEMB	9,999	32,767

Binary truncation does not occur when either COMP item appears in an arithmetic statement that uses the ON SIZE ERROR phrase. In this instance, COBOL-81 performs decimal truncation.

If you want COBOL-81 to perform decimal truncation in all numeric data manipulations, specify the /TRUNCATE compiler qualifier.

1.7.3 Using the ROUNDED Phrase

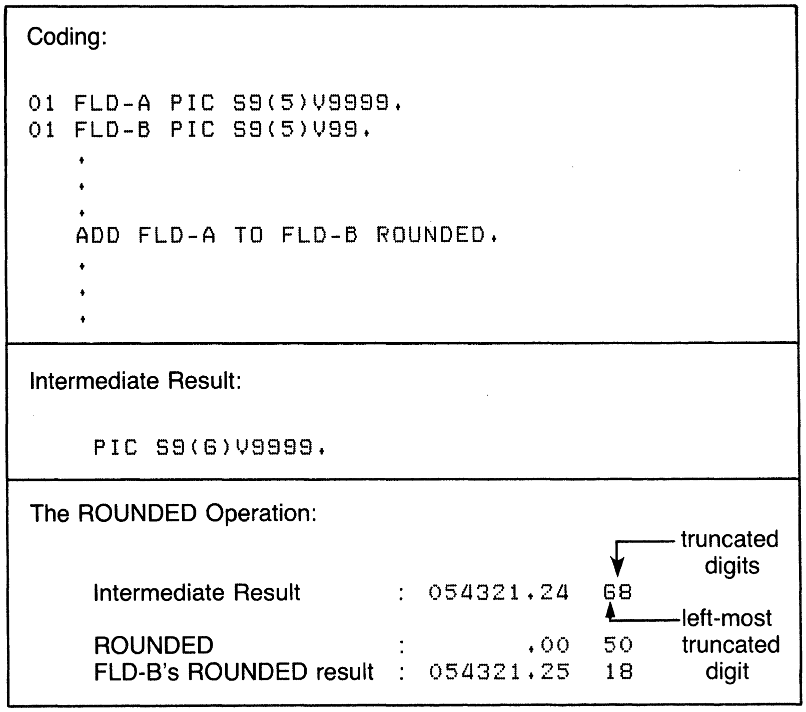
Rounding off is an important tool with most arithmetic operations. The ROUNDED phrase causes the OTS to round off the results of COBOL-81 arithmetic operations.

The phrase can be used with any COBOL-81 arithmetic statement. Rounding off takes place only when the ROUNDED phrase requests it – and then only if the intermediate result has more low-order digits than the result.

COBOL-81 rounds off by adding a 5 to the leftmost truncated digit of the absolute value of the intermediate result before it stores that result.

Figure 1-7 shows the results of rounding off an intermediate value of 54321.2468.

Figure 1-7: Results of the ROUNDED Phrase



Coding:

```
01 FLD-A PIC S9(5)V9999.
01 FLD-B PIC S9(5)V99.
    .
    .
    .
    ADD FLD-A TO FLD-B ROUNDED.
    .
    .
    .
```

Intermediate Result:

```
    PIC S9(6)V9999.
```

The ROUNDED Operation:

```
Intermediate Result       : 054321.24  68
ROUNDED                   :       .00  50
FLD-B's ROUNDED result    : 054321.25  18
```

C81ART-10021-30

Figure 1-8 rounds off to the decimal scaling position (P). Assume an intermediate result of 24680. (Section 1.7.5 discusses the GIVING phrase in numeric operations.)

Figure 1-8: Results of the ROUNDED Phrase

Coding:

```
01 AMOUNT1 PIC 9999.
01 AMOUNT2 PIC 9999PP.
   .
   .
   .
   MULTIPLY AMOUNT1 BY 10
      GIVING AMOUNT2 ROUNDED.
   .
   .
   .
```

Intermediate Result:

```
    PIC 999999.
```

The ROUNDED Operation:

```
                                         ┌──── truncated
                                         ▼        digits
    Intermediate Result        :  0246   80.
    ROUNDED                    :         50.

    AMOUNT2's ROUNDED result   :  0247   30.
```

C81ART-10022-30

1.7.4 Using the SIZE ERROR Phrase

The SIZE ERROR phrase detects the loss of high-order nonzero digits in the results of COBOL-81 arithmetic operations.

The phrase can be used in any COBOL-81 arithmetic statement.

When the execution of a statement with no SIZE ERROR phrase results in a size error, the OTS truncates the high-order digits and stores the result without notifying the user. When the same statement includes a SIZE ERROR phrase, the OTS discards the entire result without altering the receiving items in any way and executes the SIZE ERROR imperative phrase.

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

The SIZE ERROR phrase cannot be used with numeric MOVE statements. Thus, if a program moves a numeric quantity to a smaller numeric item, it can inadvertently lose high-order digits. For example, consider the following move of an item to a smaller item:

```
01   AMOUNT-A PIC 9(8)V99.
01   AMOUNT-B PIC 9(4)V99.
     .
     .
     .
MOVE AMOUNT-A TO AMOUNT-B.
```

This MOVE operation always loses four of AMOUNT-A's high-order digits. Either of the following two statements could determine whether these digits are zero or nonzero and could be tailored to any size item:

1.
```
IF AMOUNT-A NOT > 9999.99

MOVE AMOUNT-A TO AMOUNT-B

ELSE ...
```

2.
```
ADD ZERO AMOUNT-A GIVING AMOUNT-B

ON SIZE ERROR ...
```

Both alternatives allow the MOVE operation to occur only if AMOUNT-A loses no significant digits. If the value in AMOUNT-A is too large, both avoid altering AMOUNT-B and take the alternate execution path.

1.7.5 Using the GIVING Phrase

The GIVING phrase moves the intermediate result of an arithmetic operation to a receiving item. The phrase acts exactly like a MOVE statement in which the intermediate result serves as the sending item, and the data item following the word GIVING serves as the receiving item.

The phrase can be used with the ADD, SUBTRACT, MULTIPLY, and DIVIDE statements.

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

1.7.6 Multiple Operands in ADD and SUBTRACT Statements

Both the ADD and SUBTRACT statements can contain a string of operands preceding the word TO, FROM, or GIVING.

Multiple operands in either of these statements cause the OTS to add the string of operands together and use the intermediate result of that operation as a single operand to be added to, or subtracted from, the receiving item. TEMP is an intermediate result item. Consider the following examples:

1. Statement:
```
ADD A B C D TO E F G H.
```
   Equivalent coding:
```
ADD  A B,     GIVING TEMP.
ADD TEMP,  C, GIVING TEMP.
ADD TEMP,  D, GIVING TEMP.
ADD TEMP,  E, GIVING E.
ADD TEMP,  F  GIVING F.
ADD TEMP,  G  GIVING G.
ADD TEMP,  H  GIVING H.
```

2. Statement:
```
SUBTRACT A, B, C, FROM D.
```
   Equivalent coding:
```
ADD A, B,              GIVING TEMP.
ADD TEMP, C            GIVING TEMP.
SUBTRACT TEMP FROM D GIVING D.
```

(continued on next page)

3. Statement:

```
ADD A B C D GIVING E.
```

Equivalent coding:

```
ADD A B       GIVING TEMP.
ADD TEMP C    GIVING TEMP.
ADD TEMP D    GIVING E.
```

As in all COBOL-81 statements, any commas in these statements are optional.

Only statement 3 can have a numeric edited receiving item, because it is the only statement containing a GIVING phrase.

1.7.7 Common Errors in Arithmetic Statements

The most common errors programmers make when using arithmetic statements are:

- Using an alphanumeric item in an arithmetic statement. The MOVE statement allows data movement between alphanumeric items and certain numeric items, but arithmetic statements require that all items be numeric.
- Writing the ADD or SUBTRACT statements without the GIVING phrase, and attempting to put the result into a numeric edited item.
- Using an ADD statement with both the words TO and GIVING as in the following example:

  ```
  ADD A TO B GIVING C.
  ```

- Subtracting a 1 from a numeric counter that was described as an unsigned quantity and then testing for a value of less than zero.
- Forgetting that the MULTIPLY statement, without the GIVING phrase, stores the result back into the second operand (multiplier).
- Performing a series of calculations that generates an intermediate result larger than 18 digits when the final result will be fewer digits. You can prevent this problem by interspersing divisions with multiplications or by dropping nonsignificant digits after multiplying large numbers or numbers with many decimal places.
- Performing an operation on an item that contains a value greater than the precision of its data description. This can happen only if the item was disarranged by a group move or redefinition.
- Forgetting that you must specify the ROUNDED phrase for each item in an arithmetic statement containing multiple receiving items.
- Forgetting that the ON SIZE ERROR phrase applies to all receiving items in an arithmetic statement containing multiple receiving items. Only those receiving items for which a size error condition is raised are left unaltered. The ON SIZE ERROR imperative statement is executed after all the receiving items are processed by the OTS.

1.8 Arithmetic Expression Processing

COBOL-81 provides the arithmetic statements ADD, SUBTRACT, MULTIPLY, and DIVIDE and the facilities of arithmetic expressions using the +, –, *,/, and ** operators. You can perform a given arithmetic computation in any of several ways. For example, if you want to compute a salesman's total yearly sales as the sum of the four individual sales quarters, you might use this sample code:

```
     .
     .
     .
MOVE 0 TO TEMP.
ADD 1ST-SALES TO TEMP.
ADD 2ND-SALES TO TEMP.
ADD 3RD-SALES TO TEMP.
ADD 4TH-SALES TEMP GIVING TOTAL-SALES.
     .
     .
     .
```

In this example, a series of single ADD statements compute the final value of TOTAL-SALES by holding the partial sums in a temporary location called TEMP, which you defined in the Data Division of the program. You specify the class, usage and number of integer and decimal places to be maintained.

Another possible solution to the problem is:

```
     .
     .
     .
ADD 1ST-SALES, 2ND-SALES, 3RD-SALES, 4TH-SALES
                    GIVING TOTAL-SALES.
     .
     .
     .
```

In this example, the program computes TOTAL-SALES using a single ADD statement. As in the previous example, an intermediate result is required to develop the partial sums of the four quarterly sales quantities. However, in this example, the compiler defines the intermediate result in a manner transparent to the source program. It allocates storage for and assigns various attributes to this result according to the rules defined by COBOL-81. (Refer to the section on arithmetic operations in Chapter 5 of the *COBOL-81 Language Reference Manual*.) In particular, the composite of the ADD statement operands determines the number of integer and decimal places, and the usage assigned to the intermediate result. (See the *COBOL-81 Language Reference Manual* for details of the composite of operands for the arithmetic statements.) In the next example, consider another computational method:

```
     .
     .
     .
COMPUTE TOTAL-SALES = 1ST-SALES + 2ND-SALES + 3RD-SALES + 4TH-SALES.
     .
     .
     .
```

This sample coding uses a single COMPUTE statement with an embedded arithmetic expression. Again, an intermediate result is required and is defined by the compiler. The compiler generates the intermediate result using the two following rules:

1. Arithmetic operations are combined without restrictions on the composite of operands and/or receiving items.
2. Each COBOL-81 compiler implementor indicates techniques used in handling arithmetic expressions.

Thus, you can and should expect differences between various implementations of American National Standard COBOL 1974. The rest of this section describes how the COBOL-81 compiler computes the sizes of intermediate results.

The compiler computes the size of an intermediate result for each component operation of an arithmetic expression. Each component operation can be stated as:

OP1 OPR OP2

where:

OP1	is the first operand.
OPR	is an arithmetic operator.
OP2	is the second operand.

The compiler describes the size of an intermediate result in terms of the number of integer places (IP) and the number of decimal places (DP), both of which are a function of the integer and decimal places contained in the component operation. The symbol DPEXP represents the maximum number of decimal places in the entire arithmetic expression. Table 1-7 gives the rules for determining the intermediate result size for each of the arithmetic operators.

Note

If IP plus DP is greater than 18, arithmetic is done in a temporary work area where only the 18 most significant digits are held.

Table 1-7: Rules for Intermediate Result Size

Arithmetic Operator (OPR)	Intermediate Result Size

+ and −	IP = max(IP(OP1), IP(OP2)) + 1
	DP = max(DP(OP1), DP(OP2))

*	IP = IP(OP1) + IP(OP2)
	DP = DP(OP1) + DP(OP2)

/	IP = IP(OP1) + DP(OP2)
	DP = max(DPEXP, max(DP(OP1), DP(OP2) + 1))

**	For exponents that convert to one-word values:
	a = OP2
	b = OP2 + DP(OP1)
	Otherwise,
	a = 9, if IP(OP2) = 1
	otherwise, a = 19
	b = DPEXP
	and
	IP = IP(OP1) * a
	DP = max(DPEXP, DP(OP1) * b)

Chapter 2
Nonnumeric Character Handling

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

Items in a COBOL program belong to any of three data classes – alphanumeric, alphabetic, or numeric. Numeric items contain only numeric values. Alphabetic items contain only A to Z (uppercase or lowercase) and space characters. Alphanumeric items can contain values that are:

- all alphabetic
- all numeric
- a mixture of alphabetic and numeric
- any character from the ASCII character set

The item's data description specifies which class the item belongs to.

Classes are further subdivided into categories. For alphanumeric and numeric data items, class and category are synonymous. Alphanumeric items can be numeric edited, alphanumeric edited, or alphanumeric. Every elementary item, except for an index data item, belongs to one of the classes and its categories. The class of a group item is treated at run time as alphanumeric regardless of the classes of subordinate elementary items.

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

When you are reading the following sections of this chapter, keep in mind the distinction between the class or category of a data item and the actual value that the item contains.

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

Regardless of the class of an item, it is usually possible to store a value in the item, at run time, that is "illegal". Thus, nonnumeric ASCII characters can be placed in an item described as numeric, and an alphabetic item can be loaded with nonalphabetic characters.

2.1 Data Organization

A COBOL-81 record must have an 01 level number and consists of a set of data description entries that describe record characteristics. A data description entry can be either a group item or an elementary item. A group item is a data item that is followed by one or more elementary items or other group items, all of which have higher valued level numbers than the group to which they are subordinate. An elementary item has no higher valued subordinate level number. The record must have an 01 or a 77 level number.

All of the records used by COBOL-81 programs (except for certain registers and switches) must be described in the Data Division of the source program. The compiler allocates memory space for these items (except for Linkage Section items) and fixes them in size at compilation time.

The following sections explain how the compiler handles group and elementary data items.

2.1.1 Group Items

The size of a group item is the sum of the sizes of its subordinate elementary items. The compiler considers group items to be alphanumeric DISPLAY items, and it ignores the structure of the data they contain.

2.1.2 Elementary Items

The size of an elementary item is determined by the number of symbols that represent character positions contained in the PICTURE character-string. For example, consider this record description:

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

Both elementary items require seven bytes of memory; however, item FIELD-1 contains seven alphanumeric bytes while item FIELD-2 contains seven decimal digits and an operational sign. Operations on such items are independent of the mapping of the item into memory words (16-bit words that hold two 8-bit bytes). An item can begin in the leftmost or rightmost byte of a word with no effect on the function of any operations that refer to that item.

In effect, the compiler sees memory as a continuous array of bytes, not words. This becomes particularly important when you are defining a table using the OCCURS clause (see Chapter 3).

Records, items with a 77 level number, and all literal values given in the Procedure Division automatically begin on even byte addresses.

2.2 Special Characters

COBOL-81 allows you to manipulate any of the 128 characters of the ASCII character set as alphanumeric data, even though many of the characters are control characters, which usually control input/output devices. Generally, alphanumeric data manipulations attach no meaning to an 8-bit byte. Thus, you can move and compare these control characters in the same manner as alphabetic and numeric characters.

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

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

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

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

The ASCII character set listed in Appendix B of the *COBOL-81 Language Reference Manual* indicates the decimal value for any ASCII character.

2.3 Testing Nonnumeric Items

The following sections describe the relation and class tests applicable to nonnumeric items.

2.3.1 Relation Tests of Nonnumeric Items

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

An IF statement with a relation condition compares two operands, either of which can be an identifier or a literal, except that both cannot be literals. If the stated relation exists between the two operands, the relation condition is true.

When coding a relational operator, leave a space before and after each reserved word. When the reserved word NOT is present, the compiler considers it and the next key word or relational character to be one relational operator defining the comparison. Table 2-1 shows the meanings of the relational operators.

Table 2-1: Relational Operator Descriptions

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

2.3.1.1 Classes of Data — COBOL-81 allows comparison of both numeric class operands and nonnumeric class operands; however, it handles each class of data differently. For example, it allows a comparison of two numeric operands regardless of the formats specified in their respective USAGE clauses, but it requires that all other comparisons (including comparisons of any group items) be between operands with the same usage. It compares numeric class operands with respect to their algebraic values and nonnumeric (or a numeric and a nonnumeric) class operands with respect to a specified collating sequence.

If only one of the operands is numeric, it must be an integer data item or an integer literal, and it must be DISPLAY usage. The manner in which the compiler handles numeric operands depends on the nonnumeric operand.

1. If the nonnumeric operand is an elementary item or a literal, the compiler treats the numeric operand as if it had been moved into an alphanumeric data item the same size as the numeric operand and then compared. This causes any operational sign, whether carried as a separate character or as an overpunched character, to be stripped from the numeric item so that it appears to be an unsigned quantity.

   In addition, if the PICTURE character-string of the numeric item contains trailing P characters indicating that there are assumed integer positions that are not actually present, they are filled with zero digits. Thus, an item with a PICTURE character-string of S9999PPP is moved to a temporary location where it is described as 9999999. If its value is 432J (–4321), the value in the temporary location will be 4321000. The numeric digits, stored as ASCII bytes, take part in the comparison.

2. If the nonnumeric operand is a group item, the compiler treats the numeric operand as if it had been moved into a group item the same size as the numeric operand and then compared. This is equivalent to a group move.

   The compiler ignores the description of the numeric item (except for length) and, therefore, includes in its length any operational sign, whether carried as a separate character or as an overpunched character. Overpunched characters are never ASCII numeric digits. They are characters ranging from A through R, {, or }. Thus, the sign and the digits, stored as ASCII bytes, take part in the comparison, and zeros are not supplied for P characters in the PICTURE character-string.

The compiler does not accept a comparison between a noninteger numeric operand and a nonnumeric operand. If you try to compare these two items, you receive a diagnostic message at compile time.

2.3.1.2 Comparison Operations — If the two operands are acceptable, the compiler compares them byte for byte. The comparison starts at the first byte and compares the corresponding bytes until it either encounters a pair of unequal bytes or reaches the last byte of the longer operand.

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

If the operands have different lengths, the comparison proceeds as though the shorter operand were extended on the right by sufficient ASCII spaces (octal 40) to make them both the same length.

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

2.3.2 Class Tests for Nonnumeric Items

An IF statement with a class condition (NUMERIC or ALPHABETIC) tests the value in a nonnumeric data item (USAGE DISPLAY only) to determine whether it contains numeric or alphabetic data and uses the result to alter the flow of control in the program. For example:

```
IF ITEM-1 IS NUMERIC...
IF ITEM-2 IS ALPHABETIC...
IF ITEM-3 IS NOT NUMERIC...
```

If the data item consists entirely of the ASCII characters 0 through 9, with or without the operational sign, the class condition is NUMERIC. If the item consists entirely of the ASCII characters A through Z (upper or lower case) and spaces, the class condition is ALPHABETIC.

When the reserved word NOT is present, the compiler considers it and the next key word as one class condition defining the class test to be executed. For example, NOT NUMERIC determines if an operand contains at least one nonnumeric byte.

If the item being tested is described as a numeric data item, it can only be tested as NUMERIC or NOT NUMERIC. The NUMERIC test cannot examine either of the following:

- An item described as alphabetic
- A group item containing elementary items whose data descriptions indicate the presence of operational signs

For further information on using class conditions with numeric items, see the *COBOL-81 Language Reference Manual*, Chapter 5.

2.4 Data Movement

Three COBOL-81 statements (MOVE, STRING, and UNSTRING) perform most of the data movement operations required by business-oriented programs. The MOVE statement simply moves data from one item to another. The STRING statement concatenates a series of sending items into a single receiving item. The UNSTRING statement disperses a single sending item into multiple receiving items. Section 2.5 describes the MOVE statement, Section 2.6 describes STRING, and Section 2.7 describes UNSTRING.

The MOVE statement handles most data movement operations on character strings. However, it is limited in its ability to handle multiple items. For example, it cannot, by itself, concatenate a series of sending items into a single receiving item or disperse a single sending item into several receiving items.

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

Example 2-1 demonstrates item concatenation using two MOVE statements.

Example 2-1: Item Concatenation

```
01  SEND-1          PIC X(5) VALUE "FIRST".
01  SEND-2          PIC X(6) VALUE "SECOND".

01  RECEIVE-GROUP.
    05  REC-1       PIC X(5).
    05  REC-2       PIC X(6).

PROCEDURE DIVISION.
A00-BEGIN.

    MOVE SEND-1 TO REC-1.
    MOVE SEND-2 TO REC-2.

    DISPLAY RECEIVE-GROUP.
    STOP RUN.
```

The result of the concatenation is:

```
FIRSTSECOND
```

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

The STRING and UNSTRING statements handle concatenation and dispersion more easily.

2.5 Using the MOVE Statement

The MOVE statement moves the contents of one item into another. For example:

```
MOVE FIELD1 TO FIELD2

MOVE CORRESPONDING FIELD1 TO FIELD2
```

FIELD1 is the sending item name, and FIELD2 is the receiving item name.

The first statement causes the compiler to move the contents of FIELD1 into FIELD2. The two items need not be the same size, class, or usage; they can be either group or elementary items. If the two items are not the same length, the compiler aligns them on one end or the other. It also truncates or space-fills the other end. The movement of group items and nonnumeric elementary items is discussed in the next section.

The MOVE statement alters the contents of every character position in the receiving item.

2.5.1 Group Moves

If either the sending or receiving item is a group item, the compiler considers the move to be a group move. It treats both the sending and receiving items as if they were alphanumeric items.

If the sending item is a group item, and the receiving item is an elementary item, the compiler ignores the receiving item description except for the size description, in bytes, and any JUSTIFIED clause. It conducts no conversion or editing on the receiving item.

If a receiving item contains an OCCURS ... DEPENDING ON clause, you must either initialize the item using the VALUE clause or move a value into *depending-item* anytime prior to the group move.

2.5.2 Elementary Moves

If both items of a MOVE statement are elementary items, their PICTURE character-strings control their data movement. If the receiving item was described as numeric or numeric edited, the rules for numeric moves control the data movement. (See Chapter 1, Numeric Character Handling.)

Table 2-2 shows the legal and illegal nonnumeric elementary moves.

Table 2-2: Nonnumeric Elementary Moves

Sending Item Category	Receiving Item Category	
	Alphabetic	Alphanumeric Alphanumeric Edited
ALPHABETIC	Legal	Legal
ALPHANUMERIC	Legal	Legal
ALPHANUMERIC EDITED	Legal	Legal
NUMERIC INTEGER (DISPLAY ONLY)	Illegal	Legal
NUMERIC EDITED	Illegal	Legal

In all legal moves, the compiler treats the sending item as though it had been described as PIC X(n). A JUSTIFIED clause in the sending item's description has no effect on the move. If the sending item's PICTURE character-string contains editing characters, the compiler uses them only to determine the item's size.

Numeric items must be in DISPLAY (byte) format and must be integers.

If the description of the numeric data item indicates the presence of an operational sign (either as a character or an overpunched character) or if there are P characters in its character-string, the compiler first moves the item to a temporary location. It removes the sign and fills out any P character positions with zero digits. It then uses the temporary value as the sending item as if it had been described as PIC X(n). The temporary value can be shorter than the original if a separate sign was removed – or longer if P character positions were filled in with zeros.

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

A numeric integer data item sending item has no effect on the justification of the receiving item. It the numeric sending item is shorter than the receiving item, the compiler fills the receiving item with spaces.

In legal, nonnumeric elementary moves, the receiving item controls the movement of data. All of the following characteristics of the receiving item affect the move:

- Its size
- Editing characters in its description
- The JUSTIFIED RIGHT clause in its description

The JUSTIFIED clause and editing characters are mutually exclusive.

When an item that contains no editing characters or JUSTIFIED clause in its description is used as the receiving item of a nonnumeric elementary MOVE statement, the compiler moves the characters starting at the leftmost position of the item and scans across, character-by-character, to the rightmost position. If the sending item is shorter than the receiving item, the compiler fills the remaining character positions with spaces. If the sending item is longer than the receiving item, truncation occurs on the right.

2.5.2.1 Edited Moves — Alphabetic or alphanumeric items can contain editing characters. Consider the following insertion editing characters:

B	Blank insertion position
0	Zero insertion position
/	Slash insertion position

When an item with an insertion editing character in its PICTURE character-string is the receiving item of a nonnumeric elementary MOVE statement, each receiving character position corresponding to an editing character receives the insertion byte value. Table 2-3 illustrates the use of such symbols with the following statement, where FIELD1 is described as PIC X(7):

```
MOVE FIELD1 TO FIELD2
```

Table 2-3: Data Movement with Editing Symbols

FIELD1	FIELD2	
	Character-String	Contents After MOVE
070476	XX/99/XX	07/04/76
04JUL76	99BAAAB99	04 JUL 76
2351212	XXXBXXXX/XX/	235 1212/ /
123456	0XB0XB0XB0X	01 02 03 04

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

2.5.2.2 Justified Moves — A JUSTIFIED RIGHT clause in the receiving item's data description causes the compiler to reverse its usual data movement conventions. It starts with the rightmost characters of both items and proceeds from right to left. If the sending item is shorter than the receiving item, the compiler fills the remaining leftmost character positions with spaces. If the sending item is longer than the receiving item, truncation occurs on the left. Table 2-4 illustrates various PICTURE character-string situations for the following statement (with no editing):

```
MOVE FIELD1 TO FIELD2,
```

Table 2-4: Data Movement with No Editing

FIELD1		FIELD2	
PICTURE Character-string	Contents	PICTURE Character-string (and JUST Clause)	Contents After MOVE
XXX	ABC	XX	AB
		XXXXX	ABC
		XX JUST	BC
		XXXXX JUST	ABC

2.5.3 Multiple Receiving Items

If you write a MOVE statement containing more than one receiving item, the compiler moves the same sending item value to each of the receiving items. It has essentially the same effect as a series of separate MOVE statements, all with the same sending item. For information on subscripted items, see Section 2.6.4.

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

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

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

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

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

2.5.4 Subscripted Moves

Any item of a MOVE statement can be subscripted, and the referenced item can be used to subscript another name in the same statement.

When more than one receiving item is named in the same MOVE statement, the order in which the compiler evaluates the subscripts affects the results of the move. Consider the following examples:

Example 1:

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

Example 2:

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

In Example 1, the compiler evaluates FIELD1(FIELD2) only once, before it moves any data to the receiving items. It is as if the statement were replaced with the following statements:

```
MOVE FIELD1(FIELD2) TO TEMP.

MOVE TEMP TO FIELD2.

MOVE TEMP TO FIELD3.
```

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

```
MOVE FIELD1 TO FIELD2.

MOVE FIELD1 TO FIELD3(FIELD2).
```

2.5.5 Common Nonnumeric Item MOVE Statement Errors

The compiler considers any MOVE statement that contains a group item to be a group move. If an elementary item contains editing characters, or a numeric integer, these attributes of the receiving item, which determine the action of an elementary move, have no effect on the action of a group move.

2.5.6 Using the MOVE CORRESPONDING Statement for Nonnumeric Items

The MOVE CORRESPONDING statement allows you to move multiple items from one group item to another group item using a single MOVE statement. See the *COBOL-81 Language Reference Manual* for rules on the CORRESPONDING phrase. When you use the CORRESPONDING phrase, the compiler performs an independent move operation on each pair of corresponding items from the operands and checks the legality of each. Example 2-2 shows the use of the MOVE CORRESPONDING statement.

Example 2-2: Sample Record Description Using the MOVE CORRESPONDING Statement

```
01 A-GROUP.                       01 B-GROUP.
   02 FIELD1.                        02 FIELD1.
      03 A PIC X.                       03 A PIC X.
      03 B PIC 9.                       03 C PIC XX.
      03 C PIC XX.                      03 E PIC XXX.
      03 D PIC 99.
      03 E PIC XXX.

MOVE CORRESPONDING A-GROUP TO B-GROUP.
```

Because FIELD1 qualifies the elementary items for both the A-GROUP and B-GROUP items named in the MOVE CORRESPONDING statement, the preceding example is equivalent to the following series of MOVE statements:

```
MOVE A OF A-GROUP TO A OF B-GROUP.

MOVE C OF A-GROUP TO C OF B-GROUP.

MOVE E OF A-GROUP TO E OF B-GROUP.
```

2.6 Concatenating Data Using the STRING Statement

The STRING statement concatenates the contents of two or more sending items into a single receiving item.

The statement has many forms; the simplest is equivalent in function to a nonnumeric MOVE statement. Consider the following example:

```
STRING1 FIELD1 DELIMITED BY SIZE INTO FIELD2.
```

If the two items are the same size, or if the sending item (FIELD1) is larger, the statement is equivalent to the following statement:

```
MOVE FIELD1 TO FIELD2.
```

If the sending item is shorter than the receiving item, the compiler does not replace unused positions in the receiving item with spaces. Thus, the STRING statement can leave some portion of the receiving item unchanged.

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

2.6.1 Multiple Sending Items

The STRING statement can concatenate a series of sending items into one receiving item. Consider the following example of the STRING statement:

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

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

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

If the sending items do not fill the receiving item, the operation stops when the last character of the last sending item (FIELD1C) is moved. It does not alter the contents nor space-fill the remaining character positions of the receiving item.

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

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

2.6.2 Using the POINTER Phrase

Although the STRING statement normally starts scanning at the leftmost position of the receiving item, the POINTER phrase makes it possible to start scanning at another point within the item. The scanning, however, remains left-to-right. Consider the following example:

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

The value of P determines the starting character position in the receiving item. In this example, the 5 in P causes the compiler to move the first character of FIELD1A into character position 5 of FIELD2 (the leftmost character position of the receiving item is character position 1) and leave positions 1 through 4 unchanged.

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

2.6.3 Using the DELIMITED BY Phrase

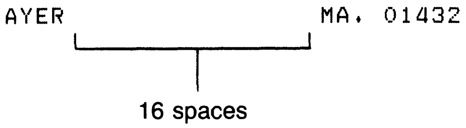
Although the sending items of the STRING statement are fixed in size at compile time, the sending items are frequently filled with spaces. For example, a 20-character city item can contain only the word MAYNARD followed by 13 spaces. The STRING statement using the DELIMITED BY SIZE phrase would move the word "MAYNARD" and the unwanted 13 spaces, assuming the receiving item is at least 20 characters long. The DELIMITED BY phrase, written with a data-name or literal, eliminates this type of problem.

The delimiter can be a literal, a data item, a figurative constant, or the word SIZE. It cannot be ALL literal since ALL literal has an indefinite length. When the phrase contains the word SIZE, the compiler moves each sending item in total, until it either exhausts the sending item or fills the receiving item.

Consider the following example:

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

If CITY is a 20-character item, the result of the STRING operation might look like the following:



```
AYER                MA, 01432
```

C81ART-10023-6

A far more attractive and functional report can be produced by having the STRING operation produce this line:

```
AYER, MA, 01432
```

To accomplish this, use the figurative constant SPACE as a delimiter on the sending item:

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

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

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

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

The sample STRING statement cannot handle two-word city names, such as New York, since the compiler considers the space between the two words as a match for the delimiter SPACE. A longer delimiter, such as two or three spaces (nonnumeric literal), can solve this problem. Only when a sequence of characters matches the delimiter does the movement stop for that data item. With a two-character delimiter, the same statement can be rewritten in a simpler form:

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

Since only the CITY item contains two consecutive spaces (the entire STATE item is only two characters long), the delimiter's search of the other items will always be unsuccessful, and the effect is the same as moving the full item (delimiting by SIZE).

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

Remember the remainder of the receiving item is not space-filled as with a MOVE statement. If ADDRESS-LINE is to be printed on a mailing label, for example, the STRING statement should be preceded by the statement:

```
MOVE SPACES TO ADDRESS-LINE.
```

This guarantees a space-fill to the right of the concatenated result. Alternatively, the last item concatenated by the STRING statement can be an item previously set to SPACES. This sending item must be moved under control of a delimiter other than SPACE.

2.6.4 Using the OVERFLOW Phrase

When the SIZE option of the DELIMITED BY phrase controls the STRING operation, and the pointer value is either known or the POINTER phrase is not used, you can add the sending items together to see if the receiving item is large enough to hold the sending items. However, if the DELIMITED BY phrase contains a literal or an identifier, or if the pointer value is not predictable, it can be difficult to tell whether or not the size of the receiving item is large enough. An overflow can occur if this is the case.

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

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

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

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

Example 2-3 illustrates the overflow condition.

Example 2-3: Sample Overflow Condition

```
DATA DIVISION.
       .
       .
       .
01 FIELD1A PIC XXX VALUE "ABC".
01 FIELD2 PIC XXXX.

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

The STRING statement numbers in the example point to corresponding results shown in Table 2-8.

Table 2-5: Results of Sample Overflow Statements

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

2.6.5 Subscripted Items in STRING Statements

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

Since you can use the pointer value as a subscript on one or more items in the statement, it is important to understand the order in which the compiler evaluates the subscripts and exactly when it updates the pointer.

Note

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

The compiler updates the pointer after it moves the last character out of each sending item. Consider the following example:

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

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

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

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

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

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

The compiler evaluates the subscript of a receiving item only once – at the start of the STRING operation. Therefore, if you use the pointer as a subscript on the receiving items, changes made to the pointer during execution of the STRING statement do not change the choice of which receiving string is altered.

You can subscript the delimiter field using a data-name or the pointer. The compiler reevaluates the delimiter subscript once for each sending item, immediately before it compares the delimiter to the item. Thus, by subscripting it with the pointer value, the delimiter can be changed for each sending item. This has the peculiar effect of choosing the next sending item's delimiter based on the position (in the receiving item) into which its first character will fall. Consider the following example:

```
01 DTABLE.
   03 D PIC X OCCURS 7 TIMES.

   MOVE 1 TO P.
   STRING "ABC"
          "ABC"
          "ABC" DELIMITED BY D(P)
          INTO R WITH POINTER P.
```

Table 2-6 shows the values moved from the three "ABC" literals to receiving item (R) for the DTABLE values shown in the left column:

Table 2-6: Results of Sample Delimiter Subscripts

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

However, if the pointer item is not used as a subscript on any of the items in the statement, the point at which the compiler evaluates the subscripts is immaterial to the execution of the statement.

Note

By avoiding the use of the pointer as a subscript, you can expect uniform results from all COBOL compilers that adhere to 1974 ANSI COBOL.

2.6.6 Common STRING Statement Errors

The most common errors you are likely to make when writing STRING statements are:

- Using the word "TO" instead of "INTO"
- Forgetting to write "DELIMITED BY SIZE"
- Forgetting to initialize the pointer
- Initializing the pointer to 0 instead of 1
- Forgetting to provide for space-filling of the receiving item when it is desirable

2.7 Separating Data Using the UNSTRING Statement

The UNSTRING statement disperses the contents of a single sending item into multiple receiving items.

The statement has many forms; the simplest is equivalent in function to a nonnumeric MOVE statement. Consider the following example:

```
UNSTRING FIELD1 INTO FIELD2.
```

The sample statement is equivalent to MOVE FIELD1 TO FIELD2, regardless of the relative sizes of the two items.

The sending item (FIELD1) can be either (1) a group item or (2) an alphanumeric or alphanumeric edited elementary item. The receiving item (FIELD2) can be alphabetic, alphanumeric, or numeric, but it cannot specify any type of editing.

If the receiving item is numeric, it must be DISPLAY usage. The PICTURE character-string of a numeric receiving item can contain any of the legal numeric description characters except for P and the editing characters. The UNSTRING statement moves the sending item to the numeric receiving item as if the sending item had been described as an unsigned integer. It automatically truncates or zero-fills as required.

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

2.7.1 Multiple Receiving Items

The UNSTRING statement can disperse one sending item into several receiving items. Consider the following example of the UNSTRING statement written with multiple receiving items:

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

The compiler performs the UNSTRING operation by scanning across FIELD1, the sending item, from left to right. When the number of characters scanned equals the number of characters in the receiving item, the compiler moves the scanned characters into that item and begins scanning the next group of characters for the next receiving item.

If each of the receiving items in the preceding example (FIELD2A, FIELD2B, and FIELD2C) is five characters long, and FIELD1 is 15 characters long, the compiler scans across FIELD1 until the number of characters scanned equals the size of FIELD2A (five). It then moves those first five characters to FIELD2A, and it sets the scanner to the sixth character position in FIELD1. Next, the compiler scans across FIELD1 from character position six, until the number of scanned characters equals the size of FIELD2B (five). The compiler then moves the sixth through the tenth characters to FIELD2B, and it sets the scanner to the next (eleventh) character position in FIELD1. For the last move in this example, it moves characters 11 through 15 of FIELD1 into FIELD2C.

Each data movement acts as an individual MOVE statement, the sending item of which is an alphanumeric item equal in size to the receiving item. If the receiving item is numeric, the move operation converts the data to numeric form. For example, consider what would happen if the items under discussion had the data descriptions and were manipulating the values shown in Table 2-7.

Table 2-7: Values Moved into the Receiving Items Based on the Sending Item Value

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

FIELD2A is an alphanumeric item. Therefore, the compiler simply conducts an elementary nonnumeric move with the first five characters.

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

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

If the sending item is shorter than the sum of the sizes of the receiving items, the compiler ignores the remaining receiving items. If it reaches the end of the sending item before it reaches the end of one of the receiving items, the compiler moves the scanned characters into that receiving item. It left-justifies and fills the remaining character positions with spaces for alphanumeric data, or it decimal point aligns and zero fills the remaining character positions for numeric data.

Consider the following statement with reference to the corresponding PICTURE character-strings and values in Table 2-8.

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

FIELD2A is a three-character alphanumeric item. It receives the first three characters of FIELD1 (ABC) in every operation. FIELD2B, however, runs out of characters every time before filling, as Table 2-8 illustrates.

Table 2-8: Handling a Short Sending Item

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

2.7.2 Controlling Moved Data Using the DELIMITED BY Phrase

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

UNSTRING delimiters can be literals, figurative constants (including ALL literal), or identifiers (identifiers can even be subscripted data-names). This section discusses the use of these three types of delimiters. Subsequent sections cover multiple delimiters, the COUNT phrase, and the DELIMITER phrase. Subscripting delimiters is discussed at the end of this section.

Consider the following sample UNSTRING statement with the figurative constant, SPACE, as a delimiter:

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

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

Table 2-9 shows the results of the following UNSTRING operation that uses a literal asterisk delimiter:

```
UNSTRING FIELD1 DELIMITED BY "*"
         INTO FIELD2.
```

Table 2-9: Results of Delimiting with an Asterisk

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

Legend: △ = space

If the delimiter matches the first character in the sending item, the compiler considers the size of the sending item to be zero. The operation still takes place, however, and fills the receiving item with spaces if it is nonnumeric or zeros if it is numeric.

A delimiter can also be applied to an UNSTRING statement that has multiple receiving items:

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

The compiler scans FIELD1 searching for a character that matches the delimiter. If it finds a match, it moves the scanned characters to FIELD2A and sets the scanner to the next character position to the right of the character that matched. It then resumes scanning FIELD1 for a character that matches the delimiter. If the compiler finds a match, it moves all of the characters between the character that first matched the delimiter and the character that matched on the second scan, and it sets the scanner to the next character position to the right of the character that matched.

The DELIMITED BY phrase handles additional items in the same manner as it handled FIELD2B.

Table 2-10 illustrates the results of a delimited UNSTRING operation into multiple receiving items:

```
UNSTRING FIELD1 DELIMITED BY "*"
         INTO FIELD2A FIELD2B.
```

Table 2-10: Results of Delimiting Multiple Receiving Items

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

Legend: Δ = space

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

Table 2-11 shows the results of an UNSTRING operation using a two-character delimiter:

```
UNSTRING FIELD1 DELIMITED BY "**"
         INTO FIELD2A FIELD2B.
```

Table 2-11: Results of Delimiting with Two Asterisks.

	Values After UNSTRING Operation	
FIELD1 PIC X(8) VALUE IS:	FIELD2A PIC XXX	FIELD2B PIC XXX JUSTIFIED
ABC**DEF	ABC	DEF
A*B*C*D*	A*B	△△△
AB***C*D	AB△	C*D
AB**C*D*	AB△	*D*
AB**CD**	AB△	△CD
AB***CD*	AB△	CD*
AB*****CD	AB△	△△△

Legend: △ = space

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

Table 2-12 shows the results of an UNSTRING operation using an ALL delimiter:

```
UNSTRING FIELD1 DELIMITED BY ALL "*"
         INTO FIELD2A FIELD2B.
```

Table 2-12: Results of Delimiting with ALL Asterisks.

	Values After UNSTRING Operation	
FIELD1 PIC X(8) VALUE IS:	FIELD2A PIC XXX	FIELD2B PIC XXX JUSTIFIED
ABC*DEF*	ABC	DEF
ABC**DEF	ABC	DEF
A******F	A△△	△△F
A*F*****	A△△	△△F
A*CDEFG	A△△	EFG

Legend: △ = space

Table 2-13 shows the results of an UNSTRING operation that combines ALL with a two-character delimiter:

```
UNSTRING FIELD1 DELIMITED BY ALL "**"
         INTO FIELD2A FIELD2B.
```

Table 2-13: Results of Delimiting with ALL Double Asterisks.

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

Legend: Δ = space

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

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

The data-name DEL1, must be alphanumeric. It can be a group or elementary item. If the delimiter contains a subscript, the subscript can be varied as a side effect of the UNSTRING operation. The evaluation of subscripts is discussed later in this section.

2.7.2.1 Multiple Delimiters — The UNSTRING statement scans a sending item, searching for a match from a list of delimiters. This list can contain ALL delimiters and delimiters of various sizes. The only requirement of the list is that delimiters must be connected by the word OR.

The following sample statement unstrings a sending item into three receiving items. The sending item consists of three strings separated by one of the following: (1) any number of spaces, (2) a comma followed by a single space, (3) a single comma, (4) a tab character, or (5) a carriage-return character. The comma and space must precede the single comma in the list if the comma and space are to be recognized.

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

Table 2-14 shows the potential of this statement. The tab and carriage-return characters represent single-character items containing the ASCII horizontal tab and carriage-return characters.

Table 2-14: Results of Multiple Delimiters

FIELD1 PIC X(12)	FIELD2A PIC XXX	FIELD2B PIC 9999	FIELD2C PIC XXX
A,0,C(RET)	A△△	0000	C△△
A(TAB)456, E	A△△	0456	E△△
A 3 9	A△△	0003	9△△
A(TAB)(TAB)B(RET)	A△△	0000	B△△
A,,C	A△△	0000	C△△
ABCD, 4321,Z	ABC	4321	Z△△

Legend: (RET) = carriage-return character
(TAB) = tab character
△= space

2.7.3 Counting UNSTRING Characters Using the COUNT Phrase

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

The length of a delimited sending item can vary from zero to the full length of the item. Some programs require knowledge of this length. For example, some data is truncated if it exceeds the size of the receiving item, so the program's logic requires this information.

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

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

The compiler counts the number of characters between the leftmost position of FIELD1 and the first asterisk in FIELD1 and places the count into COUNT2A. The compiler does not include the delimiter in the count because it is not a part of the string. The data preceeding the first asterisk is then moved into FIELD2A.

The compiler then counts the number of characters between the last contiguous asterisk in the first scan and the next asterisk in the second scan and places the count in COUNT2B. The data between the delimiters of the second scan is moved into FIELD2B.

The third scan begins at the first character after the last contiguous asterisk in the second scan. Any data between the delimiters of this scan is moved to FIELD2C.

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

If the receiving item is shorter than the value placed in the count item, the compiler truncates the sending string. If the number of integer positions in a numeric item is smaller than the value placed into the count item, high-order numeric digits have been lost. If the compiler finds a delimiter match on the first character it examines, it places a zero in the count item.

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

2.7.4 Saving UNSTRING Delimiters Using the DELIMITER Phrase

The DELIMITER phrase causes the actual character or characters that delimited the sending item to be stored in a user-supplied data area. This phrase is most useful when:

- The UNSTRING statement contains a delimiter list
- Any one of the delimiters in the list might have delimited the item
- Program logic flow depends on the delimiter match found

By using the DELIMITER and COUNT phrases, you can make program logic flow dependent on both the size of the sending string and the delimiter terminating the string.

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

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

After moving the first sending string to FIELD2A, the compiler takes the character (or characters) that delimited that string and places it in DELIMA. In this example, DELIMA contains either a comma, a tab, a carriage return, or any number of spaces. Because the delimiter string is moved under the rules of the elementary nonnumeric MOVE statement, the compiler truncates or space-fills with left- or right-justification.

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

When a sending string is delimited by the end of the sending item rather than a match on a delimiter, the delimiter string is of zero length and the DELIMITER item is space-filled. The phrase should be used only where needed. In this example, the character that delimits the last sending string is not needed, so no DELIMITER phrase follows FIELD2C.

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

When you use both DELIMITER and COUNT phrases, the DELIMITER phrase must precede the COUNT phrase. Both of the data items named in these phrases can be subscripted or indexed. If they are subscripted, the subscript can be varied as a side effect of the UNSTRING operation. The evaluation of subscripts is discussed in Section 2.7.8.

2.7.5 Controlling UNSTRING Scanning Using the POINTER Phrase

Although the UNSTRING statement scan usually starts at the leftmost position of the sending item, the POINTER phrase lets you control the character position where the scan starts. Scanning, however, remains left-to-right.

When a sending item is to be unstrung into multiple receiving items, the choice of delimiters and/or the size of subsequent receiving items can depend on the size of the first sending string and/or the character that delimited that string. Thus, the program needs to move the first sending item, hold its scanning position in the sending item, and examine the results of the operation to determine how to handle the sending items that follow.

This is done by using an UNSTRING statement with a POINTER phrase that fills only the first receiving item. When the first string has been moved to a receiving item, the compiler begins the next scanning operation one character beyond the delimiter that caused the interruption. The program examines the new position, the receiving item, the delimiter value, and the sending string size. It resumes the scanning operation by executing another UNSTRING statement with the same sending item and pointer data item. In this way, the UNSTRING statement moves one sending string at a time, with the form of each succeeding move depending on the context of the preceding string of data.

The POINTER phrase must follow the last receiving item in the UNSTRING statement. You are responsible for initializing the pointer before the UNSTRING statement executes. Consider the following two UNSTRING statements with their accompanying POINTER phrases and tests:

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

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

Because the compiler considers the leftmost character to be character position 1, the value of PNTR can be used to examine the next character. To do this, describe the sending item as a table of characters and use PNTR as a sending item subscript. This is shown in the following example:

```
01 FIELD1.
   02 FIELD1-CHAR OCCURS 40 TIMES.
   .
   .
   .
   UNSTRING FIELD1
            .
            .
            .
            WITH POINTER PNTR.
   IF FIELD1-CHAR(PNTR) = "X" ...
```

Another way to examine the next character of the sending item is to use the UNSTRING statement to move it to a one-character receiving item:

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

The program must decrement PNTR in order to work, because the second UNSTRING statement increments the pointer by 1.

The program must initialize the POINTER phrase data item before the UNSTRING statement uses it. The compiler will terminate the UNSTRING operation if the initial value of the pointer is less than one or greater than the length of the sending item. Such a pointer value causes an overflow condition. Overflow conditions are discussed in Section 2.7.7.

Sending items can also be subscripted. For example, the following statement uses subscripts to concatenate the elements of a table (A-TABLE) into a single item (A-FOUR). SUB1 can be either a subscript or an index-name.

```
STRING A-TABLE(SUB1) A-TABLE(SUB1+1) A-TABLE(SUB1+2) A-TABLE(SUB1+3)
        DELIMITED BY SIZE INTO A-FOUR.
```

2.7.6 Counting UNSTRING Receiving Items Using the TALLYING Phrase

The TALLYING phrase counts the number of receiving items that received data from the sending item.

When an UNSTRING statement contains several receiving items, there are not always as many sending strings as there are receiving items. The TALLYING phrase provides a convenient method for keeping a count of how many receiving items actually received strings. The following example shows how to use the TALLYING phrase.

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

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

The TALLYING data item always contains the sum of its initial contents plus the number of receiving items receiving data. Thus, you might want to initialize the tally count before each use.

You can use the POINTER and TALLYING phrases together in the same UNSTRING statement, but the POINTER phrase must precede the TALLYING phrase. Both phrases must follow all of the item names, the DELIMITER phrase, and the COUNT phrase. The data items for both phrases must contain numeric integers without editing characters or the symbol P in their PICTURE character-strings; both data items can be either COMP or DISPLAY usage. They can be signed or unsigned and, if they are DISPLAY usage, they can contain any desired sign option.

The data items for both phrases can be subscripted or indexed, or they can be used as subscripts on other items in the statement. The evaluation of subscripts is discussed in Section 2.8.8. A convenient use of the TALLYING phrase data item is as a subscript of a receiving item. The following example causes program control to execute the UNSTRING statement repeatedly until it exhausts the sending item:

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

Program control loops through the UNSTRING statement, using pointer PNTR to scan FIELD1 with successive executions. Each comma isolates a sending string until the scan reaches either a carriage return or the end of FIELD1. If the scan reaches the end of the item without encountering a carriage return, the compiler places a space into delimiter item DEL2, and control falls through the IF statement and out of the loop.

Because TALLYING item TLY is incremented by 1 after each string movement, TLY serves as a subscript on the receiving item. In effect, this causes the compiler to unpack the value in FIELD1 into an array of fixed-size items.

An array of COUNT data items can be supplied and loaded using the UNSTRING/TALLYING statement by adding the COUNT IN phrase. For example:

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

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

2.7.7 Exiting an UNSTRING Statement Using the OVERFLOW Phrase

The OVERFLOW phrase detects the overflow condition and causes an imperative statement to be executed when it detects the condition. An overflow condition exists:

1. When the UNSTRING statement is about to execute and its pointer data item contains a value less than one or greater than the size of the sending item. The compiler executes the OVERFLOW phrase before it moves any data, and the values of all the receiving items remain unchanged.

2. When data still remains in the sending item after the UNSTRING statement has filled all the receiving items. The compiler executes the OVERFLOW phrase after it has executed the UNSTRING statement. The value of each receiving item is updated, but some data is still unmoved.

If the UNSTRING operation causes the scan to move past the rightmost position of the sending item (thus exhausting it), the compiler does not execute the OVERFLOW phrase.

The following set of instructions causes program control to execute the UNSTRING statement repeatedly until it exhausts the sending item. The TALLYING data item is a subscript that indexes the receiving item. Compare this loop with the previous loop, which accomplishes the same thing.

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

Note

The overflow condition also occurs if the value of a pointer data item lies outside the sending item at the start of execution of the UNSTRING statement (the value is less than one or greater than the length of the sending item). This type of overflow is not distinguishable from the overflow condition described at the start of this section, except that this condition causes the UNSTRING statement to terminate before any data movement takes place. Then, the values of all receiving items remain unchanged.

2.7.8 Using Subscripted Items in UNSTRING Statements

Because the flexibility of the UNSTRING statement is enhanced by subscripting and indexing, it is important to understand how often and exactly when the compiler evaluates these subscripts and indexes. This section discusses the frequency and timing of subscript evaluation.

The compiler evaluates the subscripts and indexes of some data items only once before the UNSTRING statement executes. Any changes to the subscripts and indexes during execution of the UNSTRING statement have no effect on the data items. These data items are as follows:

- Sending item
- POINTER data item
- TALLYING data item

The compiler evaluates subscripts and indexes of some data items immediately before moving data into them and moves data into these items in the order that they are listed in the UNSTRING statement – the same order as the following:

1. Receiving item
2. DELIMITER data-item
3. COUNT data-item

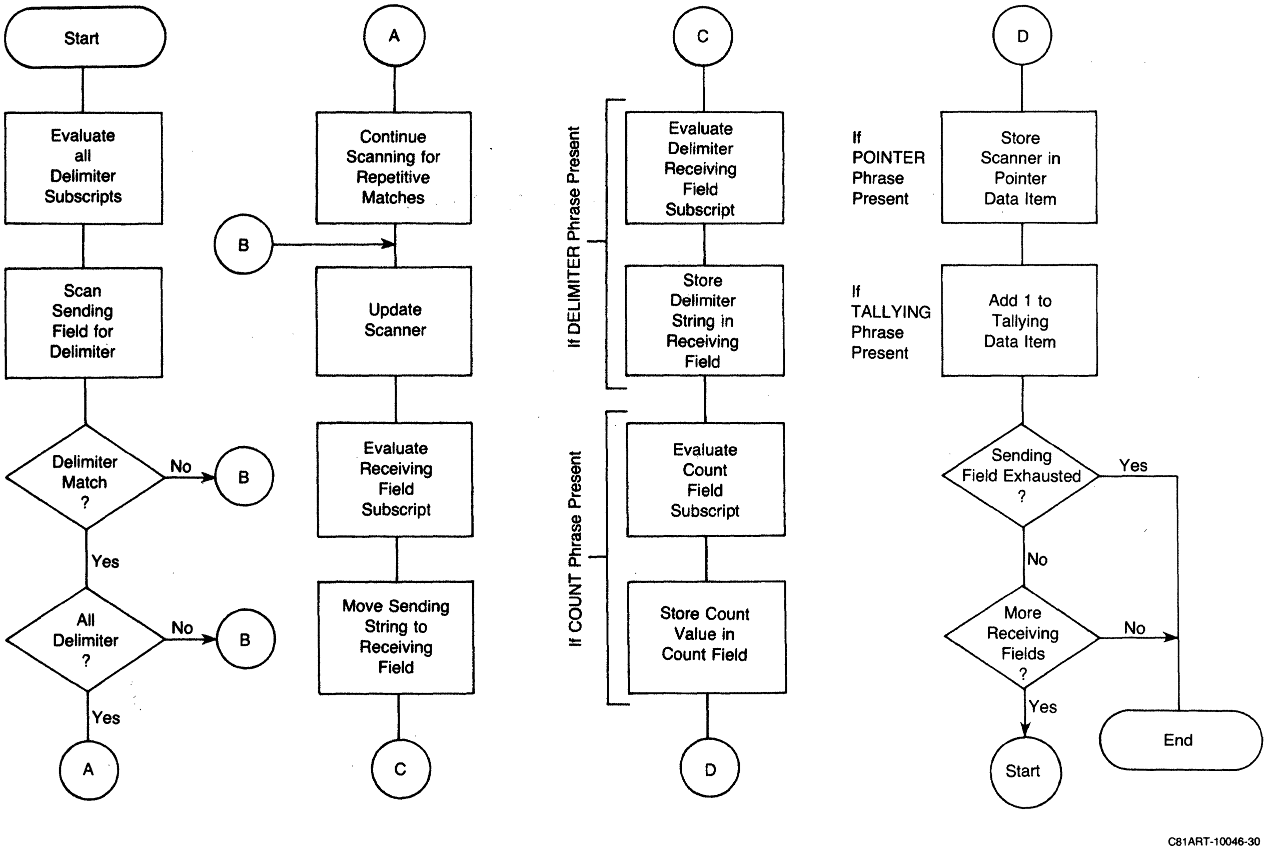
The compiler evaluates any subscripts and indexes on the delimiter data-names in the DELIMITED BY phrase immediately before it scans each sending string looking for a delimiter match. Thus, it reevaluates these subscripts and indexes once for each receiving item in the UNSTRING statement.

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

- POINTER data-item
- TALLYING data-item
- COUNT data-item
- Another receiving item

Figure 2-1 shows a flow chart of the sequence of evaluation operations:

Figure 2-1: Sequence of Subscript Evaluation


Start
Evaluate all Delimiter Subscripts
Scan Sending Field for Delimiter
Delimiter Match ?
No
B
Yes
All Delimiter ?
No
B
Yes
A
A
Continue Scanning for Repetitive Matches
B
Update Scanner
Evaluate Receiving Field Subscript
Move Sending String to Receiving Field
C
If DELIMITER Phrase Present
C
Evaluate Delimiter Receiving Field Subscript
Store Delimiter String in Receiving Field
If COUNT Phrase Present
Evaluate Count Field Subscript
Store Count Value in Count Field
D
D
If POINTER Phrase Present
Store Scanner in Pointer Data Item
If TALLYING Phrase Present
Add 1 to Tallying Data Item
Sending Field Exhausted ?
Yes
No
More Receiving Fields ?
No
Yes
Start
End
C81ART-10046-30

Note

The rules concerning the exact point at which the compiler evaluates DELIMITED BY phrase identifiers, and the point at which it updates the POINTER and TALLYING data items, are specified by 1974 American National Standard COBOL. There are no such rules for the STRING statement.

2.7.9 Common UNSTRING Statement Errors

The most common errors made when writing UNSTRING statements are:

- Leaving the OR connector out of a delimiter list
- Misspelling or interchanging the words DELIMITED and DELIMITER
- Writing the DELIMITER and COUNT phrases in the wrong order when both are present (DELIMITER must precede COUNT)
- Omitting the word INTO (or writing it as TO) ahead of the receiving item list
- Repeating the word INTO in the receiving item list as shown in this example:

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

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

2.8 Examining and Replacing Characters Using the INSPECT Statement

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

Like the STRING and UNSTRING operations, INSPECT operations scan across the item from left to right. Included in the INSPECT statement is an optional phrase that allows scanning to begin or terminate upon detection of a delimiter match. This feature allows scanning to begin within the item as well as at the leftmost position.

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

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

INSPECT FIELD1 REPLACING ALL SPACE BY ZERO.
```

The first statement causes the compiler to scan FIELD1 looking for the character B. Each time a B is found, TLY is incremented by 1.

The second statement causes the compiler to scan FIELD1 looking for spaces. Each space found is replaced with a zero.

You can use both the TALLYING and REPLACING phrases in the same INSPECT statement. However, when used together, the TALLYING phrase must precede the REPLACING phrase. An INSPECT statement with both phrases is equivalent to two separate INSPECT statements. In fact, the compiler compiles such a statement into two distinct INSPECT statements. To simplify debugging, it is best to initially write the two phrases in separate INSPECT statements.

2.8.1 Restricting Data Inspection Using the BEFORE/AFTER Phrase

The BEFORE/AFTER phrase acts as a delimiter and can restrict the area of the item being inspected.

The following sample statement counts only the zeros that precede the percent sign (%) in FIELD1:

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

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

- If the delimiter is an identifier, it must be an elementary data item of DISPLAY usage. It can be alphabetic, alphanumeric, or numeric, and it can contain editing characters. The compiler always treats the item as if it had been described as an alphanumeric string. It does this by implicit redefinition of the item, as described in Section 2.8.2.
- If the delimiter is a literal, it must be nonnumeric.

The compiler repeatedly compares the delimiter characters against an equal number of characters in the item being inspected. If none of the characters matches the delimiter, or if insufficient characters remain in the rightmost position of the item for a full comparison, the compiler considers the comparison to be unequal.

The examples of the INSPECT statement in Table 2-15 illustrate the way the delimiter character finds a match in the item being inspected. The portion of the item the statement ignores as a result of the BEFORE/AFTER phrase delimiters is crossed out with a slash, and the portion it inspects is underlined.

Table 2-15: Matching Delimiter Characters to Characters in a Field

Instruction	FIELD1 Value
INSPECT FIELD1...BEFORE "E".	ABCD~~EFGHI~~
INSPECT FIELD1...AFTER "E".	~~ABCDE~~FGHI
INSPECT FIELD1...BEFORE "K".	ABCDEFGHI
INSPECT FIELD1...AFTER "K".	~~ABCDEFGHI~~
INSPECT FIELD1...BEFORE "AB".	~~ABCDEFGHI~~
INSPECT FIELD1...AFTER "AB".	~~AB~~CDEFGHI
INSPECT FIELD1...BEFORE "HI".	ABCDEFG~~HI~~
INSPECT FIELD1...AFTER "HI".	~~ABCDEFGHI~~
INSPECT FIELD1...BEFORE "I ".	ABCDEFGHI
INSPECT FIELD1...AFTER "I ".	~~ABCDEFGHI~~

The ellipses represent the position of the TALLYING or REPLACING phrase. The compiler scans the item for a delimiter match before it scans for the inspection operation (TALLYING or REPLACING), thus establishing the limits of the operation before beginning the actual inspection. Section 2.8.3 further discusses the importance of the separate scan.

2.8.2 Implicit Redefinition

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

- If the item is alphabetic, alphanumeric edited, or unsigned numeric, the compiler redefines it as alphanumeric. This is a compile-time operation; no data movement occurs at run time.
- If the item is signed numeric, the compiler first removes the sign and then redefines the item as alphanumeric. If the sign is a separate character, the compiler ignores that character, essentially shortening the item, and that character does not participate in the implicit redefinition. If the sign is an "overpunch" on the leading or trailing digit, the compiler actually removes the sign value and leaves the character with only the numeric value that was stored in it.

The compiler alters the digit position containing the sign before beginning the INSPECT operation and restores it to its former value after the operation. If the sign's digit position does not contain a valid ASCII signed numeric digit, the action of the redefinition causes the value to change.

Table 2-16 shows these original, altered, and restored values.

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

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

Table 2-16: Values Resulting from Implicit Redefinition

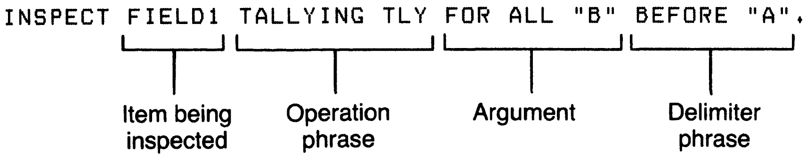
Original Value	Altered Value	Restored Value
} (173)	0 (60)	} (173)
A (101)	1 (61)	A (101)
B (102)	2 (62)	B (102)
C (103)	3 (63)	C (103)
D (104)	4 (64)	D (104)
E (105)	5 (65)	E (105)
F (106)	6 (66)	F (106)
G (107)	7 (67)	G (107)
H (110)	8 (70)	H (110)
I (111)	9 (71)	I (111)
{ (175)	0 (60)	{ (175)
J (112)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

2.8.3 Examining the INSPECT Operation

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

Figure 2-2 shows an example of the INSPECT statement. The item to be inspected must be named (FIELD1), and the item name must be followed by an operation phrase (TALLYING TLY). The operation phrase must be followed by one or more identifiers or literals (B). These identifiers or literals comprise the "arguments" (items to be compared to the item being inspected). More than one argument makes up the "argument list."

Figure 2-2: Sample INSPECT Statement

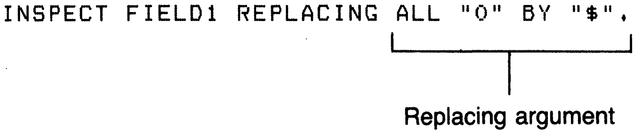


```
INSPECT FIELD1 TALLYING TLY FOR ALL "B" BEFORE "A".
```

C81ART-10024-10

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

Each argument in an argument list used in a REPLACING operation must have a replacement item associated with it. The compiler uses the replacement item to replace each string of characters in the item that matches the argument. A typical REPLACING phrase is shown in the following example (with $ as the replacement item):



```
INSPECT FIELD1 REPLACING ALL "O" BY "$".
```

C81ART-10025-4

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

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

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

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

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

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

Table 2-17 shows an INSPECT...TALLYING statement that is scanning FIELD1, tallying in TLY, and looking for the arguments and delimiters listed in the left column. Assume that TLY is initialized to 0.

Table 2-17: Relationship Between INSPECT Argument, Delimiter, Item Value, and Argument Active Position

Argument and Delimiter	FIELD1 Value	Argument Active at Position	Contents of TLY After Scan
"B" AFTER "XX"	BXBXXXXBB	6	2
	XXXXXXXX	3	0
	BXBXBBBBXX	never	0
"X" AFTER "XX"	BXBXXBXXB	6	2
	XXXXXXXX	3	6
	BBBBBBXX	never	0
"B" AFTER "XB"	BXYBXBXX	7	0
	XBXBXBXB	3	3
	BBBBBBXB	never	0
"BX" AFTER "XB"	XXXXBXXXX	6	0
	XXXXBBXXX	6	1
	XXBXXXXBX	4	1

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

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

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

When an active argument does find a match, the compiler ignores any remaining arguments in the list and conducts the TALLYING or REPLACING operation on the character. The scanner moves to a new position and the next inspection operation begins with the first argument in the list. The INSPECT statement can contain additional conditions, which are described later in this section; without them however, the argument match is allowed to take place, and inspection continues following the match.

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

2.8.4 Subscripted Items in INSPECT Statements

Any identifier named in an INSPECT statement can be subscripted or indexed. The compiler evaluates all subscripts in an INSPECT statement once, before the inspection begins; therefore, if the action of the INSPECT statement alters one of the subscripts, the new subscript value has no effect on the selection of operands during that inspection operation. For example, consider the following:

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

In this sample statement, the compiler evaluates the address of X(TLY) only once, before it begins inspecting the item; hence, it evaluates X(TLY) as X(1). If the action of inspecting and tallying alters TLY, the value of TLY has no effect on the choice of the X operand. The value (1) will be used throughout the operation.

Note

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

2.8.5 The TALLYING Phrase

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

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

Each time the tally argument matches the delimited string being inspected, the compiler adds one to the tally counter.

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

2.8.5.2 The Tally Argument — The tally argument specifies a character-string and a condition under which that string should be compared to the delimited string being inspected.

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

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

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

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

- An identifier must be an elementary item of DISPLAY usage. It can be any data class. However, if it is other than alphanumeric, the compiler performs an implicit redefinition of the item. This redefinition is identical to the BEFORE/AFTER delimiter redefinition discussed earlier in Section 2.8.1.

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

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

- LEADING specifies that only adjacent matches of the TALLY argument at the leftmost position of the delimited character string be counted. At the first failure to match the tally argument, the compiler terminates counting and causes the argument to become inactive. The sample statement INSPECT...TALLYING, (scanning FIELD1, tallying in TLY, and looking for the arguments and delimiters listed in the left column) gives the results in Table 2-18 if the program initializes TLY to 0.

Table 2-18: LEADING Delimiter of the Inspection Operation

Argument and Delimiter	FIELD1 Value	Contents of TLY After Scan
LEADING "*" AFTER "0".	F***0**F	2
	F**0F**	0
	F**F**0	0
	0***F**	3
LEADING "**" AFTER "0".	F**0**F***	1
	F**F0***FF**	1
	F**F0****F**	2
	F**F**0*	0

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

The following three examples show INSPECT statements with argument lists. The text following each one explains how that list is processed.

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

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

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

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

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

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

Table 2-19 shows various values of FIELD1 and the contents of the three tally counters after the scan. Assume that the counters are initialized to 0 before the INSPECT statement.

Table 2-19: Results of the Scan with Separate Tallies

	Contents of Tally Counters After Scan		
FIELD1 Value	**T1**	**T2**	**T3**
A.C;D.E,F	1	2	1
A.B.C.D	0	1	0
A,B,C,D	3	0	0
A;B;C;D	0	0	3
*,B,C,D	0	0	0

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

2.8.5.4 Interference in Tally Argument Lists — When several tally arguments contain one or more identical characters active at the same time, they can interfere with each other so that when one of the arguments finds a match, the scanner steps past the matching character(s), preventing those character(s) from being considered for any other match.

These two identical tally arguments do not interfere with each other since they are not active at the same time. The first A in FIELD1 causes the first argument to become inactive and the second argument to become active:

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

However, the following identical tally arguments interfere with each other since both are active at the same time:

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

For any given position of the scanner, the arguments are applied to FIELD1 in the order in which they appear in the statement. When one of them finds a match, the scanner moves to the next position and ignores the remaining arguments in the argument list. Each comma in FIELD1 causes T1 to be incremented by 1 and the second argument to be ignored. Thus, T1 always contains an accurate count of all the commas in FIELD1, and T2 is always unchanged.

The following INSPECT statement arguments only partially interfere with each other:

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

The first argument does not become active until the scanner encounters an A. The second argument tallies all commas that precede the A. After the A, the first argument counts all commas and causes the second argument to be ignored. Thus, T1 contains the number of commas that precede the first A, and T2 contains the number of commas that follow the first A. This statement works well as written, but it could be difficult to debug.

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

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

If FIELD1 contains "ABCABC" after the scan, T1 is incremented by two, and T2 is unaltered. The successful matching of the argument includes each B in the item. Each match resets the scanner to the character position to the right of the B and causes the second argument never to be successfully matched. The results remain the same even if the order of the arguments is reversed. Only separate INSPECT statements can develop the desired counts.

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

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

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

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

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

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

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

T1 counts only leading asterisks in FIELD1; the occurrence of any other character causes the first tally argument to become inactive. T2 keeps a count of any remaining asterisks in FIELD1.

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

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

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

An argument with both a LEADING condition and a BEFORE phrase can sometimes sucessfully "delimit" the item being inspected as in the following example:

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

These statements count the number of "words" in the English statement in FIELD1, assuming that no more than three spaces separate the words in the sentence, that the sentence ends with a period, and that the period immediately follows the last word. When FIELD1 has been scanned, T2 contains the number of spaces between the words. Since a count of the spaces renders a number that is one less than the number of words, the conditional statement adds one to the count.

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

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

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

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

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

- T1 contains the number of leading spaces.
- T2 contains the number of periods.
- T3 contains the number of plus signs.
- T4 contains the number of minus signs.
- T5 contains the number of remaining characters (assumed to be numeric).

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

2.8.6 Using the REPLACING Phrase

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

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

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

2.8.6.1 The Search Argument — The search argument of the REPLACING phrase names a character string and a condition under which the character string should be compared to the delimited string being inspected.

The CHARACTERS form of the search argument specifies that every character in the delimited string being inspected should be considered to match an imaginary character that serves as the search argument. Thus, the replacement value replaces each character in the delimited string. For example:

```
INSPECT ITEMA REPLACING CHARACTERS ...
```

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

```
INSPECT ITEMA REPLACING ALL ...
```

The necessary literal and identifier characteristics are as follows:

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

- An identifier must represent an elementary item of DISPLAY usage. It can be any class. However, if it is other than alphabetic, the compiler performs an implicit redefinition of the item. This redefinition is identical to the BEFORE/AFTER delimiter redefinition discussed in Section 2.8.1.

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

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

- LEADING specifies that only adjacent matches of the search argument at the leftmost position of the delimited character-string be replaced. At the first failure to match the search argument, the compiler terminates the replacement operation and causes the argument to become inactive.

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

2.8.6.2 The Replacement Value — Whenever the search argument finds a match in the item being inspected, the matched characters are replaced by the replacement value. The word BY followed by an identifier or literal specifies the replacement value. For example:

```
INSPECT ITEMA REPLACING ALL "A" BY "X" ALL "D" BY "X".
```

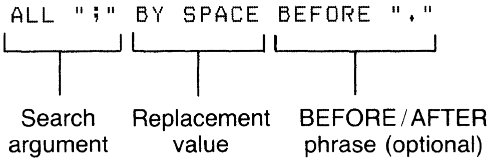
The replacement value must always be the same size as its associated search argument.

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

If the replacement value is an identifier, it must be an elementary item of DISPLAY usage. It can be any class. However, if it is other than alphanumeric, the compiler conducts an implicit redefinition of the item. This redefinition is the same as the BEFORE/AFTER redefinition discussed in Section 2.8.1.

2.8.6.3 The Replacement Argument — The replacement argument consists of the search argument (with its condition and character-string), the replacement value, and an optional BEFORE/AFTER phrase as shown in Figure 2-3.

Figure 2-3: The Replacement Argument



C81ART-10026-10

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

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

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

The previous three replacement arguments all have the same replacement value, SPACE, and are active over the entire item being inspected. The next statement replaces all commas, periods, and semicolons with space characters and leaves all other characters unchanged:

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

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

Note

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

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

The next sample statement also exchanges zeros and ones except that here the first occurrence of a space in FIELD1 causes both arguments to become inactive.

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

The first space causes the three replacement arguments to become inactive. This argument list exchanges zeros for ones, ones for zeros, and asterisks for all other characters that are in the delimited area. If the BEFORE phrase is removed from the third argument, that argument will remain active across all of FIELD1. Within the area delimited by the first space character, the third argument replaces all characters except ones and zeros with asterisks. Beyond this area, it replaces all characters (including the space that delimited FIELD1 for the first two arguments and any zeros and ones) with asterisks.

2.8.6.5 Interference in Replacement Argument Lists — When several search arguments, all active at the same time, contain one or more identical characters, they can interfere with each other – and consequently have an effect on the replacement operation. This interference is similar to the interference that occurs between tally arguments.

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

The action of a search argument is never affected by the characters of any replacement value, since the scanner does not inspect the replaced characters again during execution of the INSPECT statement. Interference between search arguments, therefore, depends on the order of the arguments, the values of the arguments, and the active/inactive status of the arguments. The discussion in Section 2.8.5.4 about interference in tally argument lists applies generally to replacement arguments as well.

The following rules help minimize interference in replacement argument lists:

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

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

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

2.8.7 Common INSPECT Statement Errors

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

- Leaving the FOR out of an INSPECT...TALLYING statement
- Using the word ''WITH'' instead of ''BY'' in the REPLACING phrase
- Failing to initialize the tally counter
- Omitting the word ''ALL'' ahead of the comparison character-string.

Chapter 3
Table Handling

3.1 Introduction

COBOL-81 requires that all data items be uniquely defined. This is usually done by assigning a unique name to each data item or by qualifying the data items. In many applications, however, this is tedious or inconvenient. Programs often contain numerous data items that have different data names but contain the same type of information. Tables provide a solution to this problem. COBOL-81 provides full table-handling capabilities.

A table is one or more repetitions of one element, comprised of one or more data items, stored in contiguous memory locations. You define a table by using an OCCURS clause following a data description entry. The literal integer value you specify in the OCCURS clause determines the number of repetitions, or occurrences, of the data description entry, thus creating a table. You can define one-, two-, and three-dimensional tables.

After you have defined a table, you can load it with data in two ways. In the most direct way, you assign values, with the VALUE clause, as part of the data description entry when you define the table. The second method allows you to store table values in a separate data file and then read and move the values into the table (with the READ and MOVE statements) when you need them.

To access data stored in tables, use subscripted or indexed procedural instructions. In either case, you can directly access a known table element occurrence or search for an occurrence based on some known condition.

This chapter discusses the procedures required to define, initialize, and access tables accurately and efficiently.

3.2 Defining Tables

To define a table you specify the OCCURS clause in the data description entry. You can define either fixed-length tables or variable-length tables. The following sections describe how to use the OCCURS clause and its options.

3.2.1 Defining Fixed-Length, One-Dimensional Tables

To define fixed-length tables, use Format 1 of the OCCURS clause (refer to the *COBOL-81 Language Reference Manual*). This format is useful when you are storing large amounts of stable, frequently used reference data. Options allow you to define single or multiple keys and/or indexes.

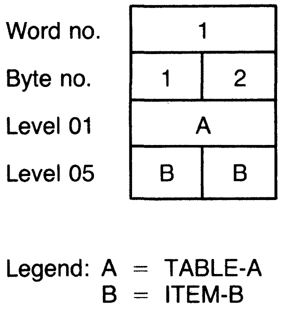
A definition of a one-dimensional table is shown in Example 3-1. In Example 3-1, the integer 2 in the OCCURS 2 TIMES clause determines the number of element repetitions. For the table to have any real meaning, this integer must be equal to or greater than two.

Example 3-1: One-Dimensional Table

```
01   TABLE-A.
     05   ITEM-B PIC X OCCURS 2 TIMES.
```

The organization of this table is shown in Figure 3-1.

Figure 3-1: Organization of the One-Dimensional Table in Example 3-1



Legend: A = TABLE-A
B = ITEM-B

C81ART-20010-15

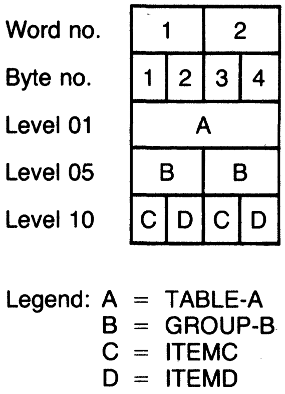
Example 3-1 specifies only a single data item. However, you can specify as many data items as you need in the table. Use of multiple data items is shown in Example 3-2.

Example 3-2: Multiple Data Items in a One-Dimensional Table

```
01   TABLE-A.
     05   GROUP-B OCCURS 2 TIMES.
          10   ITEMC PIC X.
          10   ITEMD PIC X.
```

The organization of this table is shown in Figure 3-2.

Figure 3-2: Organization of Multiple Data Items in a One-Dimensional Table



Legend: A = TABLE-A
B = GROUP-B
C = ITEMC
D = ITEMD

C81ART-20020-20

Neither Examples 3-1 nor 3-2 use the KEY IS or INDEXED BY optional phrases. If you use either the SEARCH or the SEARCH ALL verbs, you must specify at least one index. The SEARCH ALL verb also requires that you specify at least one key. Specify the search key using the ASCENDING/ DESCENDING KEY IS phrase. See Section 3.4.8 for information about the SEARCH verbs and Section 3.4.4 for information about indexing. When you use the INDEXED BY phrase, the index is internally defined and need not be defined elsewhere. Example 3-3 defines a table with an ascending search key and an index.

Example 3-3: Defining a Table with an Index and an Ascending Search Key

```
01  TABLE-A.
    05  ELEMENTB OCCURS 5 TIMES
                 ASCENDING KEY IS ITEMA
                 INDEXED BY INDX1.
        10  ITEMC PIC X.
        10  ITEMD PIC X.
```

The organization of this table is shown in Figure 3-3.

Figure 3-3: Organization of a Table with an Index and an Ascending Search Key

<table>
<tr><td>Word no.</td><td colspan="2">1</td><td colspan="2">2</td><td colspan="2">3</td><td colspan="2">4</td><td colspan="2">5</td></tr>
<tr><td>Byte no.</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td></tr>
<tr><td>Level 01</td><td colspan="10">TABLE-A</td></tr>
<tr><td>Level 05</td><td colspan="2">B</td><td colspan="2">B</td><td colspan="2">B</td><td colspan="2">B</td><td colspan="2">B</td></tr>
<tr><td>Level 10</td><td>C</td><td>D</td><td>C</td><td>D</td><td>C</td><td>D</td><td>C</td><td>D</td><td>C</td><td>D</td></tr>
</table>

Legend: B = ELEMENTB
C = ITEMC
D = ITEMD

C81ART-20030-20

3.2.2 Defining Fixed-Length, Multidimensional Tables

In addition to one-dimensional tables, COBOL-81 also allows you to define two- and three-dimensional tables. You define a two-dimensional table by defining another one-dimensional table within each element of the one-dimensional table. This process can be carried one step further by defining another one-dimensional table within each element of the two-dimensional table. This defines a three-dimensional table. A two-dimensional table is shown in Example 3-4.

Example 3-4: Defining a Two-Dimensional Table

```
01  2D-TABLE-X.
    05  LAYER-Y OCCURS 2 TIMES.
        10  LAYER-Z OCCURS 2 TIMES.
            15  CELLA PIC X.
            15  CELLB PIC X.
```

The organization of this two-dimensional table is shown in Figure 3-4.

Figure 3-4: Organization of a Two-Dimensional Table

<table>
<tr><td>Word no.</td><td colspan="2">1</td><td colspan="2">2</td><td colspan="2">3</td><td colspan="2">4</td></tr>
<tr><td>Byte no.</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td></tr>
<tr><td>Level 01</td><td colspan="8">2D-TABLE-X</td></tr>
<tr><td>Level 05</td><td colspan="4">LY</td><td colspan="4">LY</td></tr>
<tr><td>Level 10</td><td colspan="2">LZ</td><td colspan="2">LZ</td><td colspan="2">LZ</td><td colspan="2">LZ</td></tr>
<tr><td>Level 15</td><td>A</td><td>B</td><td>A</td><td>B</td><td>A</td><td>B</td><td>A</td><td>B</td></tr>
</table>

Legend: LY = LAYER-Y
LZ = LAYER-Z
A = CELLA
B = CELLB

C81ART-20040-20

Example 3-5 shows a three-dimensional table.

Example 3-5: Defining a Three-Dimensional Table

```
01 TABLE-A.
   05  LAYER-B OCCURS 2 TIMES.
       10  ITEMC PIC X.
       10  ITEMD PIC X OCCURS 3 TIMES.
       10  ITEME OCCURS 2 TIMES.
           15  CELLF PIC X.
           15  CELLG PIC X OCCURS 3 TIMES.
```

The organization of this three-dimensional table is shown in Figure 3-5.

Figure 3-5: Organization of a Three-Dimensional Table

<table>
<tr><td>Word no.</td><td colspan="2">1</td><td colspan="2">2</td><td colspan="2">3</td><td colspan="2">4</td><td colspan="2">5</td><td colspan="2">6</td><td colspan="2">7</td><td colspan="2">8</td><td colspan="2">9</td><td colspan="2">10</td><td colspan="2">11</td><td colspan="2">12</td></tr>
<tr><td>Byte no.</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td><td>12</td><td>13</td><td>14</td><td>15</td><td>16</td><td>17</td><td>18</td><td>19</td><td>20</td><td>21</td><td>22</td><td>23</td><td>24</td></tr>
<tr><td>Level 01</td><td colspan="24">A</td></tr>
<tr><td>Level 05</td><td colspan="12">B</td><td colspan="12">B</td></tr>
<tr><td>Level 10</td><td>C</td><td>D</td><td>D</td><td>D</td><td colspan="4">E</td><td colspan="4">E</td><td>C</td><td>D</td><td>D</td><td>D</td><td colspan="4">E</td><td colspan="4">E</td></tr>
<tr><td>Level 15</td><td colspan="4"></td><td>F</td><td>G</td><td>G</td><td>G</td><td>F</td><td>G</td><td>G</td><td>G</td><td colspan="4"></td><td>F</td><td>G</td><td>G</td><td>G</td><td>F</td><td>G</td><td>G</td><td>G</td></tr>
</table>

Legend: A = TABLE-A
B = LAYER-B
C = ITEMC
D = ITEMD
E = ITEME
F = CELLF
G = CELLG

C81ART-20050-25

3.2.3 Defining Variable-Length Tables

To define a variable-length table, use Format 2 of the OCCURS clause (refer to the *COBOL-81 Language Reference Manual*). Options allow you to define single or multiple keys and/or indexes.

Example 3-6 illustrates how to define a variable-length table. It uses from two to four occurrences depending on the integer value assigned to NUM-ELEM. You determine the table's minimum and maximum size with the OCCURS (minimum size) TO (maximum size) clause. The minimum size value must be equal to or greater than zero and the maximum size value must be greater than the minimum size value. The data-name in the DEPENDING ON clause must be within the minimum to maximum range.

Unlike fixed-length tables, you can dynamically alter the number of element occurrences in variable-length tables.

By generating the variable-length table in Example 3-6, you are, in effect, saying: "Build a table that can contain at least two occurrences, but no more than four occurrences, and set its present number of occurrences equal to the value specified by NUM-ELEM".

Example 3-6: Defining a Variable-Length Table

```
01  NUM-ELEM PIC 9.
        .
        .
        .
01  VAR-LEN-TABLE.
    05  TAB-ELEM OCCURS 2 TO 4 TIMES DEPENDING ON NUM-ELEM.
        10 A PIC X.
        10 B PIC X.
```

3.2.4 Storage Allocation for Tables

The compiler maps the table elements into memory, following mapping rules that depend on the presence or absence of COMP or COMP SYNC data items in the table element.

3.2.4.1 Tables Without COMP, COMP SYNC, or USAGE INDEX Items — When there are no COMP, COMP SYNC, or USAGE INDEX data items in a table definition, successive data items are mapped into adjacent memory locations using a left-to-right allocation scheme. Consider the following data description of a table and the way it is mapped into memory, illustrated by Example 3-7 and Figure 3-6.

Note

To determine exactly how much space your tables use, specify the /SHOW:MAP compiler qualifier. This gives you an offset map of both the Data Division and the Procedure Division.

Example 3-7: Sample Record Description Defining a Table

```
01  TABLE-A.
    03 GROUP-G PIC X(5) OCCURS 5 TIMES.
```

Figure 3-6: Memory Map for Example 3-7

<table>
<tr><td>Word no.</td><td colspan="2">1</td><td colspan="2">2</td><td colspan="2">3</td><td colspan="2">4</td><td colspan="2">5</td><td colspan="2">6</td><td colspan="2">7</td><td colspan="2">8</td><td colspan="2">9</td><td colspan="2">10</td><td colspan="2">11</td><td colspan="2">12</td><td colspan="2">13</td></tr>
<tr><td>Byte no.</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td><td>12</td><td>13</td><td>14</td><td>15</td><td>16</td><td>17</td><td>18</td><td>19</td><td>20</td><td>21</td><td>22</td><td>23</td><td>24</td><td>25</td><td>26</td></tr>
<tr><td>Level 01</td><td colspan="25">TABLE-A</td><td></td></tr>
<tr><td>Level 03</td><td colspan="5">GROUP-G</td><td colspan="5">GROUP-G</td><td colspan="5">GROUP-G</td><td colspan="5">GROUP-G</td><td colspan="5">GROUP-G</td><td></td></tr>
</table>

C81ART-20060-15

Alphanumeric data items require 1 byte of storage per character. Therefore, each occurrence of GROUP-G occupies 5 bytes. The first byte of the first element is automatically aligned at the left record boundary and the first 5 bytes occupy words 01, 02, and half of 03. A memory word is comprised of 2 bytes. Succeeding occurrences of GROUP-G are assigned to the next 5 adjacent bytes so that TABLE-A is comprised of 5 five-byte elements for a total of 25 bytes. Each table element, after the first, is allowed to start in either the first or second byte of a word with no regard for word boundaries.

3.2.4.2 Tables with COMP or COMP SYNC Items — Both COMP and COMP SYNC data items can occupy one, two, or four words of storage, depending on the number of digits you specify in the data definition. COMP data items, regardless of their size, always align on one-word boundaries. COMP SYNC data items, on the other hand, align on either one-, two-, or four-word boundaries depending on the number of storage words. This can have a significant effect on the amount of memory required to store tables containing COMP and COMP SYNC data items. See Chapter 1 for a discussion of COMP and COMP SYNC data items.

Example 3-8 describes a table containing a COMP SYNC data item. Figure 3-7 illustrates how it is mapped into memory.

Example 3-8: Record Description Containing a COMP SYNC Item

```
01 A-TABLE.
   03 GROUP-G OCCURS 4 TIMES.
      05 ITEM1 PIC X.
      05 ITEM2 PIC S9(5) COMP SYNC.
```

Figure 3-7: Memory Map for Example 3-8

<table>
<tr><td>Word no.</td><td colspan="2">1</td><td colspan="2">2</td><td colspan="2">3</td><td colspan="2">4</td><td colspan="2">5</td><td colspan="2">6</td><td colspan="2">7</td><td colspan="2">8</td><td colspan="2">9</td><td colspan="2">10</td><td colspan="2">11</td><td colspan="2">12</td><td colspan="2">13</td><td colspan="2">14</td><td colspan="2">15</td><td colspan="2">16</td></tr>
<tr><td>Byte no.</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td><td>12</td><td>13</td><td>14</td><td>15</td><td>16</td><td>17</td><td>18</td><td>19</td><td>20</td><td>21</td><td>22</td><td>23</td><td>24</td><td>25</td><td>26</td><td>27</td><td>28</td><td>29</td><td>30</td><td>31</td><td>32</td></tr>
<tr><td>Level 01</td><td colspan="32">A-TABLE</td></tr>
<tr><td>Level 03</td><td colspan="8">GROUP-G</td><td colspan="8">GROUP-G</td><td colspan="8">GROUP-G</td><td colspan="8">GROUP-G</td></tr>
<tr><td>Level 05</td><td>1</td><td>f</td><td>f</td><td>f</td><td>2</td><td>2</td><td>2</td><td>2</td><td>1</td><td>f</td><td>f</td><td>f</td><td>2</td><td>2</td><td>2</td><td>2</td><td>1</td><td>f</td><td>f</td><td>f</td><td>2</td><td>2</td><td>2</td><td>2</td><td>1</td><td>f</td><td>f</td><td>f</td><td>2</td><td>2</td><td>2</td><td>2</td></tr>
</table>

Legend: 1 = ITEM1
2 = ITEM2
f = fill byte

C81ART-20070-20

Because a five-digit COMP SYNC item requires two words (4 bytes) of storage, ITEM2 must start on a two-word boundary. This adds 3 fill bytes after ITEM1, and each GROUP-G occupies 8 bytes. In the previous example, A-TABLE requires 32 bytes to store 4 elements of 8 bytes each.

If, in the previous example, you define ITEM2 as a COMP data item of the same size, the storage required would be considerably less. Although ITEM2 would still require two words of storage, it would align on a one-word boundary. Two less fill bytes would be needed between ITEM1 and ITEM2, and A-TABLE would require a total of 24 bytes.

If you now add a 3-byte alphanumeric item (ITEM3) to Example 3-8 and locate it between ITEM1 and ITEM2 (see Example 3-9), the new item occupies the space formerly occupied by the 3 fill bytes. This adds 3 data bytes without changing the table size, as Figure 3-8 illustrates:

Example 3-9: Adding an Item Without Changing the Table Size

```
01 A-TABLE.
   03 GROUP-G OCCURS 4 TIMES.
      05 ITEM1 PIC X.
      05 ITEM3 PIC XXX.
      05 ITEM2 PIC 9(5) COMP SYNC.
```

Figure 3-8: Memory Map for Example 3-9

Word no.	1		2		3		4		5		6		7		8		9		10		11		12		13		14		15		16	
Byte no.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
Level 01	A-TABLE																															
Level 03	GROUP-G								GROUP-G								GROUP-G								GROUP-G							
Level 05	1	3	3	3	2	2	2	2	1	3	3	3	2	2	2	2	1	3	3	3	2	2	2	2	1	3	3	3	2	2	2	2

Legend: 1 = ITEM1
2 = ITEM2
3 = ITEM3

C81ART-20080-20

If, however, you place ITEM3 after ITEM2, the additional 3 bytes adds its own length plus another fill byte. The additional fill byte is added after the third ITEM3 character to ensure that all occurrences of the table element are mapped in an identical manner. Now, each element requires 12 bytes, and the complete table occupies 48 bytes. This is illustrated by Example 3-10 and Figure 3-9.

Example 3-10: Adding 3 Bytes That Adds 4 Bytes to the Element Length

```
01 A-TABLE.
   03 GROUP-G OCCURS 4 TIMES.
      05 ITEM1 PIC X.
      05 ITEM2 PIC 9(5) COMP SYNC.
      05 ITEM3 PIC XXX.
```

Figure 3-9: Memory Map for Example 3-10

Word no.	1		2		3		4		5		6		7		8		9		10		11		12		...
Byte no.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	...
Level 01	A-TABLE																								...
Level 03	GROUP-G												GROUP-G												...
Level 05	1	f	f	f	2	2	2	2	3	3	3	f	1	f	f	f	2	2	2	2	3	3	3	f	...

Legend: 1 = ITEM1
2 = ITEM2
3 = ITEM3
f = fill byte

C81ART-20090-22

3.3 Initializing Values of Table Elements

You can initialize a table that contains only DISPLAY items to any desired value. To initialize a table, you specify a VALUE clause in the record level preceding the record description of the item containing the OCCURS clause. Initialization is illustrated by Example 3-11 and Figure 3-10.

Example 3-11: Initializing Tables

```
01 A-TABLE VALUE IS "JANFEBMARAPRMAY
-        "JUNJULAUGSEPOCTNOVDEC".
    03 MONTH-GROUP PIC XXX USAGE DISPLAY
               OCCURS 12 TIMES.
```

Figure 3-10: Memory Map for Example 3-11

Word no.	1		2		3		4		5		6		...	13		14		15		16		17		18	
Byte no.	1	2	3	4	5	6	7	8	9	10	11	12	...	25	26	27	28	29	30	31	32	33	34	35	36
Level 01	A-TABLE																								
Level 03	M			M			M			M			...	M			M			M			M		
Byte contents	J	A	N	F	E	B	M	A	R	A	P	R	...	S	E	P	O	C	T	N	O	V	D	E	C

Legend: M = Month-Group

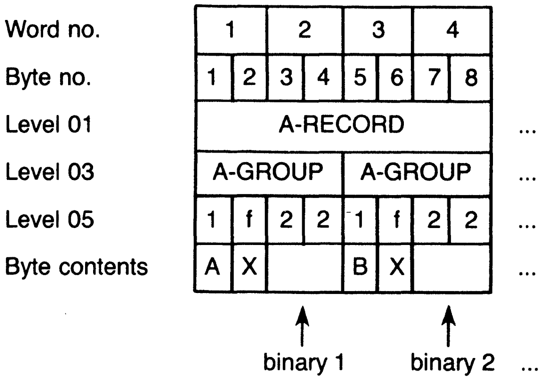
C81ART-20100-20

Often a table is too long to initialize using a single literal, or it contains numeric, alphanumeric, COMP, or COMP SYNC items that cannot be initialized. In these situations, you can initialize individual items by redefining the group level that precedes the level containing the OCCURS clause. Consider the sample table descriptions illustrated in Examples 3-12 and 3-13. Each fill byte between ITEM1 and ITEM2 in Example 3-12 is initialized to X. Figure 3-11 shows how this is mapped into memory.

Example 3-12: Initializing Mixed Usage Items

```
01 A-RECORD-ALT.
   05 FILLER PIC XX VALUE "AX".
   05 FILLER PIC S99 COMP VALUE 1.
   05 FILLER PIC XX VALUE "BX".
   05 FILLER PIC S99 COMP VALUE 2.
   .
   .
   .
01 A-RECORD REDEFINES A-RECORD-ALT.
   03 A-GROUP OCCURS 26 TIMES.
      05 ITEM1 PIC X.
      05 ITEM2 PIC S99 COMP.
```

Figure 3-11: Memory Map for Example 3-12



Legend: 1 = ITEM1
2 = ITEM2
f = fill byte

C81ART-20110-25

In Example 3-13, and as shown in Figure 3-12, each FILLER item initializes three 10-byte table elements.

Example 3-13: Initializing Alphanumeric Items

```
01 A-RECORD-ALT.
    03 FILLER PIC X(30) VALUE IS
       "AAAAAAAAAABBBBBBBBBBCCCCCCCCCC".
    03 FILLER PIC X(30) VALUE IS
       "DDDDDDDDDDEEEEEEEEEEFFFFFFFFFF".
    .
    .
    .
01 A-RECORD REDEFINES A-RECORD-ALT.
    03 ITEM1 PIC X(10) OCCURS 26 TIMES.
```

Figure 3-12: Memory Map for Example 3-13

<table>
<tr><td>Word no.</td><td colspan="2">1</td><td colspan="2">2</td><td colspan="2">3</td><td colspan="2">4</td><td colspan="2">5</td><td colspan="2">6</td><td colspan="2">7</td><td colspan="2">8</td><td colspan="2">9</td><td colspan="2">10</td><td colspan="2">11</td><td colspan="2">12</td><td>...</td></tr>
<tr><td>Byte no.</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td><td>12</td><td>13</td><td>14</td><td>15</td><td>16</td><td>17</td><td>18</td><td>19</td><td>20</td><td>21</td><td>22</td><td>23</td><td>24</td><td>...</td></tr>
<tr><td>Level 01</td><td colspan="24">A-RECORD</td><td>...</td></tr>
<tr><td>Level 03</td><td colspan="10">ITEM1</td><td colspan="10">ITEM1</td><td colspan="4">ITEM1</td><td>...</td></tr>
<tr><td>Byte contents at initialization time</td><td>A</td><td>A</td><td>A</td><td>A</td><td>A</td><td>A</td><td>A</td><td>A</td><td>A</td><td>A</td><td>B</td><td>B</td><td>B</td><td>B</td><td>B</td><td>B</td><td>B</td><td>B</td><td>B</td><td>B</td><td>C</td><td>C</td><td>C</td><td>C</td><td>...</td></tr>
</table>

C81ART-20120-20

When redefining or initializing table elements, allow space for any fill bytes that might be added due to synchronization. You do not have to initialize fill bytes, but you can do so if desired. If you do initialize fill bytes to an uncommon value, you can use them as a debugging aid in situations where a Procedure Division statement refers to the record level preceding the OCCURS clause or to another record redefining that level.

Sometimes the length and format of table items are such that they are best initialized using Procedure Division statements.

3.4 Accessing Table Elements

Once tables have been created using the OCCURS clause, the program must have a method of accessing the individual elements of those tables. Subscripting and indexing are the two methods COBOL-81 provides for accessing individual table elements. To refer to a particular element within a table, follow the name of the desired element with a subscript or index enclosed in parentheses. The following sections describe how to identify and access table elements using subscripts and indexes.

3.4.1 Subscripting

A subscript is an integer or a data-name that has an integer value. The integer value represents the desired element of the table. An integer value of 3, for example, refers to the third element.

3.4.2 Subscripting with Literals

A literal subscript is an integer value, enclosed in parentheses, that represents the desired table element. In Example 3-14, the literal subscript (2) in the MOVE instruction moves the contents of the second element of A-TABLE to I-RECORD.

Example 3-14: Using a Literal Subscript to Access a Table

Table Description:

```
01 A-TABLE.
   03 A-GROUP PIC X(5)
       OCCURS 10 TIMES.
```

Procedural Instruction:

```
MOVE A-GROUP(2) TO I-RECORD.
```

If the table is multidimensional, follow the data name of the desired data item with a list of subscripts, one for each OCCURS clause to which the item is subordinate. The first subscript in the list applies to the first OCCURS clause to which the desired item is subordinate. This is the most inclusive level, and is represented by A-GROUP in Example 3-15. The second subscript applies to the next most inclusive level and is represented by ITEM3 in the example. And finally, the last subscript applies to the least inclusive level, represented by ITEM5. In Example 3-15, the subscripts (2,11,3) in the MOVE instruction move the third repetition of ITEM5 in the eleventh repetition of ITEM3 in the second repetition of A-GROUP to I-FIELD5. For illustration simplicity, I-FIELD5 is not defined. ITEM5(1,1,1) refers to the first occurrence of ITEM5 in the table, and ITEM5(5,20,4) refers to the last occurrence of ITEM5.

Example 3-15: Subscripting a Multidimensional Table

Table Description:

```
01 A-TABLE.
   03 A-GROUP OCCURS 5 TIMES.
      05 ITEM1 PIC X.
      05 ITEM2 PIC 99 COMP OCCURS 20 TIMES.
      05 ITEM3 OCCURS 20 TIMES.
         07 ITEM4 PIC X.
         07 ITEM5 PIC XX OCCURS 4 TIMES.
```

Procedural Instruction:

```
MOVE ITEM5(2, 11, 3) TO I-FIELD5.
```

Note

Because ITEM5 is not subordinate to ITEM2, an occurrence number for ITEM2 is not permitted in the subscript list. The ninth occurrence of ITEM2 in the fifth occurrence of A-GROUP would be selected by ITEM2(5,9).

Table 3-1 shows the subscripting rules applicable to Example 3-15.

Table 3-1: Subscripting Rules for a Multidimensional Table

Name of Item	Number of Subscripts Required to Refer to the Name Item	Size of Item
A-TABLE	NONE	1110
A-GROUP	ONE	222
ITEM1	ONE	1*
ITEM2	TWO	2
ITEM3	TWO	9
ITEM4	TWO	1
ITEM5	THREE	2

* Plus a fill byte

3.4.3 Subscripting with Data-Names

You can also use data-names to specify subscripts. To use a data-name as a subscript, define it with COMP, COMP-3, or DISPLAY usage and with a numeric integer value. If the data-name is signed, the sign must be positive at the time the data-name is used as a subscript.

The sample subscripts and data-names used in Figure 3-13 refer to the table previously defined in Example 3-15.

Figure 3-13: Subscripting with Data-Names

Data Descriptions of Subscript Data-Names

```
01 SUB1 PIC 99 USAGE DISPLAY.
01 SUB2 PIC 99 USAGE COMP.
01 SUB3 PIC S99.
```

Procedural Instructions

```
   MOVE 2 TO SUB1.
   MOVE 11 TO SUB2.
   MOVE 3 TO SUB3.
   GO TO TABLERTN.
   .
   .
   .
TABLERTN.
   MOVE ITEM5(SUB1,SUB2,SUB3) TO I-FIELD5.
```

3.4.4 Subscripting with Indexes

The same rules apply for the specification of indexes as apply to subscripts except that the index must be named in the INDEXED BY phrase of the OCCURS clause.

You cannot access index items as normal data items; you cannot use them, redefine them, or write them to a file. However, the SET statement can change their values, and relation tests can examine their values. The index integer you specify in the SET statement must be in the range of one to the integer value in the OCCURS clause. The sample MOVE statement shown in Example 3-16 moves the contents of the third element of A-GROUP to I-FIELD. For illustration simplicity, I-FIELD is not defined.

Example 3-16: Subscripting with Index-Name Items

Table Description:

```
01 A-TABLE
   03 A-GROUP OCCURS 5 TIMES
      INDEXED BY IND-NAME.
```

Procedural Instructions:

```
SET IND-NAME TO 3.
MOVE A-GROUP (IND-NAME) TO I-FIELD.
```

Note

COBOL-81 initializes the value of all indexes to 1. Initializing indexes is an extension to the ANSI COBOL standards. Users who write COBOL programs that adhere to standard COBOL should not rely on this feature.

3.4.5- Relative Indexing

To perform relative indexing when referring to a table element, you follow the index name with a plus or minus sign and an integer literal. Although easy to use, relative indexing generates additional overhead each time a table element is referenced in this fashion. The run-time overhead for relative indexing of variable-length tables is significantly greater than that required for fixed-length tables. If any of the range checks reveals an out-of-range index value, program execution terminates, and an error message is issued.

The following sample MOVE statement moves the fourth repetition of A-GROUP to I-FIELD, provided IND-NAME has not been changed by a SET statement:

```
MOVE A-GROUP(IND-NAME + 3) TO I-FIELD.
```

Run-time table access using relative indexing is faster because the literal index value is calculated and stored during compilation.

3.4.6 Index Data Items

Often a program requires that the value of an index be stored outside of that item. COBOL-81 provides the index data item to fulfill this requirement.

Index data items are stored as one-word COMP items and must be declared with a USAGE IS INDEX phrase in the item description. Index data items can be explicitly modified only with the SET verb.

3.4.7 Assigning Index Values Using the SET Statement

The SET statement assigns values to indexes associated with tables so that you can reference particular table elements. Two SET statement formats are available to you as shown in the *COBOL-81 Language Reference Manual*. They are discussed in the following sections.

3.4.7.1 Assigning an Integer Index Value with a SET Statement — When you use the SET statement, the index is set to the value you specify. The most straightforward use of the SET statement is to set an index name to an integer literal value. This example references the fifth occurrence of the table containing the specified index name:

```
SET IND-1 TO 5.
```

You can also set an index name to an integer data item. For example:

```
SET INDEX-A TO COUNT-1.
```

More than one index can be set with a single set statement. For example:

```
SET TAB1-IND TAB2-IND TO 15.
```

Table indexes specified in INDEXED BY phrases cannot be displayed, moved, or manipulated in any manner. You do this by setting an index data item to the present value of an index. You could, for example, set an index data item and then display its value as shown in the following example:

```
SET INDEX-ITEM TO TAB-IND.
          .
          .
          .
DISPLAY INDEX-ITEM.
```

However, you can display, move, and manipulate the value of the table index with an index data item.

3.4.7.2 Incrementing an Index Value with the SET Statement — You can use the SET statement with the UP BY/DOWN BY clause to arithmetically alter the value of an index. A numeric literal is added to (UP BY) or subtracted from (DOWN BY) a table index. For example:

```
SET TABLE-INDEX UP BY 12.

SET TABLE-INDEX DOWN BY 5.
```

3.4.8 Identifying Table Elements Using the SEARCH Statement

The SEARCH statement is used to search a table for an element that satisfies a known condition. The statement provides for sequential and binary (nonsequential) searches, which are described in the following sections.

3.4.8.1 Implementing a Sequential Search — The SEARCH statement allows you to perform a sequential search of a table. The table description entry OCCURS clause must contain the INDEXED BY phrase. If more than one index is specified in the INDEXED BY phrase, the first index is the controlling index for the table search unless you specify otherwise in the SEARCH statement.

The search begins at the current index setting and progresses through the table, checking each element against the conditional expression. The index is incremented by one as each element is checked. If the conditional expression is true, the associated imperative statement executes (if one is present), or program control passes to the next procedural sentence. This terminates the search, and the index points to the current table element that satisfied the conditional expression.

If no table element is found that satisfies the conditional expression, the AT END exit path is taken, provided you specified one. Otherwise, program control passes to the next procedural sentence.

You can use the optional VARYING phrase of the SEARCH statement by specifying any of the following:

1. VARYING index name associated with table search
2. VARYING index data item or integer data item
3. VARYING index name not associated with table search

Regardless of which method you use, the index specified in the INDEXED BY phrase of the table being searched is incremented. This controlling index, when compared against the allowable number of occurrences in the table, dictates the permissible search range. When the search terminates, either successfully or unsuccessfully, the index remains at its current setting. At this point, you can reference the data in the table element pointed to by the index unless the AT END condition is true. If the AT END condition is true, under these circumstances, the compiler issues an error message indicating that the subscript is out of range.

The index is not initialized when the search begins. It is your responsibility to ensure that the initial index setting is the appropriate one. If you want to ensure that the entire table is included in the search range, set the controlling index to 1 before starting the search. Otherwise, the search starts at the current setting of the controlling index.

When you are varying an index associated with the table being searched, the index name can be any index you specify in the INDEXED BY phrase. It becomes the controlling index for the search and is the only index incremented. See Figure 3-14 and Example 3-18 at the end of this chapter, for an example of how to vary an index other than the first index.

When you are varying an index data item or an integer data item, either the index data item or the integer data item is incremented. The first index name you specify in the INDEXED BY phrase of the table being searched becomes the controlling index and is also incremented. The index data item or the integer data item you are varying does not function as an index; it merely allows you to maintain an additional pointer to elements within a table. See Figure 3-14 and Example 3-19 at the end of this chapter for an example of how to vary an index data item or an integer data item.

When you are varying an index associated with a table other than the one you are searching, the controlling index is the first index you specify in the INDEXED BY phrase of the table you are searching. Each time the controlling index is incremented, the index you specify in the VARYING phrase is incremented. In this manner, you can search two tables in synchronization. See Figure 3-14 and Example 3-20 at the end of this chapter for an example of how to vary an index not associated with the table you are searching.

When you omit the VARYING phrase, the first index you specify in the INDEXED BY phrase becomes the controlling index. Only this index is incremented during the serial search. See Figure 3-14 and Example 3-21 at the end of this chapter for an example of how to perform a serial search without using the VARYING phrase.

3.4.8.2 Implementing a Nonsequential (Binary) Search — You can use the SEARCH statement to perform a nonsequential table search. A nonsequential search is also known as a binary search.

To perform a binary search, you must specify an index name in the INDEXED BY phrase and a search key in the KEY IS phrase of the OCCURS clause of the table being searched.

A binary search depends on the ASCENDING/DESCENDING KEY attributes. If you specify an ASCENDING KEY, the data in the table must either be stored in ascending order or sorted in ascending order prior to the search. The same is true for a DESCENDING KEY.

During a binary search, the first, or only, index you specify in the INDEXED BY phrase of the OCCURS clause of the table being searched is the controlling index. You do not have to initialize an index in a binary search because index manipulation is automatic.

In addition to being generally faster, a binary search allows multiple equality checks.

The following object-time search sequence is presented to help you better understand the capabilities inherent in a binary search. At program execution time, the object-time system:

1. Examines the range of permissible index values, selects the median value, and assigns this value to the index.
2. Checks for equality in WHEN and AND clauses.
3. Terminates the search if all equality statements are true. If you use the imperative statement after the final equality clause, that statement executes; otherwise program control passes to the next procedural sentence, the search exits, and the index retains its current value.
4. Takes the following actions if the equality test of a table element is false:
    a. Executes the imperative statement associated with the AT END statement (if present) if all table elements have been tested. If there is no AT END statement, program control passes to the next procedural statement.
    b. Determines which half of the table is to be eliminated from further consideration. This is based on whether the key being tested was specified as ASCENDING or DESCENDING and whether the test failed because of a greater than or less than comparison. For example, if the key values are stored in ascending order, and the median table element presently being tested is greater than the value of the argument, then all key elements following the one being tested must also be greater. Therefore, the upper half of the table is removed from further consideration and the search continues at the median point of the lower half.
    c. Begins processing all over again at Step 2.

A useful variation of the binary search is that of specifying multiple search keys. Multiple search keys allow you to select a specified table element from among several elements that have duplicate low order keys. A typical example is a telephone listing where more than one person has the same last and first names – but different middle initials. All specified keys must be either ascending or descending. Example 3-22 shows how to use multiple search keys.

Figure 3-14 presents an example of a table. The table is followed by several examples of how to search it as shown in Examples 3-17 to 3-21.

Figure 3-14: Using SEARCH to Access This Sample Table

```
DATA DIVISION.
      .
      .
      .
WORKING-STORAGE SECTION.
      .
      .
      .
01  FED-TAX-TABLES.
    02  ALLOWANCE-DATA.
        03  FILLER                      PIC X(70) VALUE
            "0001440
-           "0202880
-           "0304320
-           "0405760
-           "0507200
-           "0608640
-           "0710080
-           "0811520
-           "0912960
-           "1014400".
    02  ALLOWANCE-TABLE REDEFINES ALLOWANCE-DATA.
        03  FED-ALLOWANCES OCCURS 10 TIMES
            ASCENDING KEY IS ALLOWANCE-NUMBER
            INDEXED BY IND-1.
            04  ALLOWANCE-NUMBER        PIC XX.
            04  ALLOWANCE               PIC 999V99.
    02  SINGLES-DEDUCTION-DATA.
        03  FILLER                      PIC X(112) VALUE
            "0250006700000016
-           "0670011500067220
-           "1150018300163223
-           "1830024000319621
-           "2400027900439326
-           "2790034600540730
-           "3460099999741736".
    02  SINGLE-DEDUCTION-TABLE REDEFINES SINGLES-DEDUCTION-DATA.
        03  SINGLES-TABLE OCCURS 7 TIMES
            ASCENDING KEY IS S-MIN-RANGE S-MAX-RANGE
            INDEXED BY IND-2, TEMP-INDEX.
            04  S-MIN-RANGE             PIC 999V99.
            04  S-MAX-RANGE             PIC 999V99.
            04  S-TAX                   PIC 99V99.
            04  S-PERCENT               PIC V99.
    02  MARRIED-DEDUCTION-DATA.
        03  FILLER                      PIC X(119) VALUE
            "0480009600000017
-           "0960017300081620
-           "1730026400235617
-           "2640034600390325
-           "3460043300595328
-           "4330050000838932
-           "5000099999105333G".
    02  MARRIED-DEDUCTION-TABLE REDEFINES MARRIED-DEDUCTION-DATA.
        03 MARRIED-TABLE OCCURS 7 TIMES
           ASCENDING KEY IS M-MIN-RANGE M-MAX-RANGE
           INDEXED BY IND-0, IND-3.
           04  M-MIN-RANGE              PIC 999V99.
           04  M-MAX-RANGE              PIC 999V99.
           04  M-MAX                    PIC 999V99.
           04  M-PERCENT                PIC V99.
01  TEMP-INDEX                  USAGE INDEX.
```

Example 3-17: A Serial Search

```
PROCEDURE DIVISION.
BEGIN.
   .
   .
   .
SINGLE.
       IF TAXABLE-INCOME < 02499
          GO TO END-FED-COMP.
       SET IND-2 TO 1.
       SEARCH SINGLES-TABLE VARYING IND-2 AT END
          GO TO TABLE-2-ERROR
         WHEN TAXABLE-INCOME = S-MIN-RANGE(IND-2)
               MOVE S-TAX(IND-2) TO FED-TAX-DEDUCTION OF
                    OUTPUT-MASTER
               GO TO STORE-FED-TAX
         WHEN TAXABLE-INCOME < S-MAX-RANGE(IND-2)
               SUBTRACT S-MIN-RANGE(IND-2) FROM TAXABLE-INCOME
               MULTIPLY TAXABLE-INCOME BY S-PERCENT(IND-2) ROUNDED
               ADD TAXABLE-INCOME TO FED-TAX-DEDUCTION OF
                     OUTPUT-MASTER.
```

Example 3-18: Using SEARCH and Varying an Index Other Than the First Index

```
PROCEDURE DIVISION.
BEGIN.
   .
   .
   .
MARRIED.
       IF TAXABLE-INCOME < 04799
               MOVE ZEROS TO FED-TAX-DEDUCTION OF OUTPUT-MASTER,
               GO TO END-FED-COMP.
       SET IND-3 TO 1.
       SEARCH MARRIED-TABLE VARYING IND-3
               AT END GO TO TABLE-3-ERROR
         WHEN TAXABLE-INCOME = M-MIN-RANGE(IND-3)
       MOVE M-TAX(IND-3) TO FED-TAX-DEDUCTION OF OUTPUT-MASTER,
               GO TO STORE-FED-TAX,
         WHEN TAXABLE-INCOME < M-MAX-RANGE(IND-3)
       MOVE M-TAX(IND-3) TO FED-TAX-DEDUCTION OF OUTPUT-MASTER,
       SUBTRACT M-MIN-RANGE(IND-3) FROM TAXABLE-INCOME ROUNDED,
       MULTIPLY TAXABLE-INCOME BY M-PERCENT(IND-3) ROUNDED,
       ADD TAXABLE-INCOME TO FED-TAX-DEDUCTION
                 OF OUTPUT-MASTER ROUNDED,
               GO TO STORE-FED-TAX.
```

Example 3-19: Using SEARCH and Varying an Index Data Item

```
PROCEDURE DIVISION.
BEGIN.
   .
   .
   .
```

(continued on next page)

Example 3-19: Using SEARCH and Varying an Index Data Item (Cont.)

```
SINGLE.
        IF TAXABLE-INCOME < 02499
               GO TO END-FED-COMP.
        SET IND-2 TO 1.
        SEARCH SINGLES-TABLE VARYING TEMP-INDEX AT END
               GO TO TABLE-2-ERROR
          WHEN TAXABLE-INCOME = S-MIN-RANGE(IND-2)
               MOVE S-TAX(IND-2) TO FED-TAX-DEDUCTION OF
                      OUTPUT-MASTER
               GO TO STORE-FED-TAX
          WHEN TAXABLE-INCOME < S-MAX-RANGE(IND-2)
               SUBTRACT S-MIN-RANGE(IND-2) FROM TAXABLE-INCOME
               MULTIPLY TAXABLE-INCOME BY S-PERCENT(IND-2) ROUNDED
               ADD TAXABLE-INCOME TO FED-TAX-DEDUCTION OF
                      OUTPUT-MASTER.
```

Example 3-20: Using SEARCH and Varying an Index Not Associated with the Target Table

```
PROCEDURE DIVISION.
BEGIN.
   .
   .
   .
SINGLE.
        IF TAXABLE-INCOME < 02499
               GO TO END-FED-COMP.
        SET IND-2 TO 1.
        SEARCH SINGLES-TABLE VARYING IND-0 AT END
               GO TO TABLE-2-ERROR
          WHEN TAXABLE-INCOME = S-MIN-RANGE(IND-2)
               MOVE S-TAX(IND-2, TO FED-TAX-DEDUCTION OF
                     OUTPUT-MASTER
               GO TO STORE-FED-TAX
          WHEN TAXABLE-INCOME < S-MAX-RANGE(IND-2)
               SUBTRACT S-MIN-RANGE(IND-2) FROM TAXABLE-INCOME
               MULTIPLY TAXABLE-INCOME BY S-PERCENT(IND-2) ROUNDED
               ADD TAXABLE-INCOME TO FED-TAX-DEDUCTION OF
                     OUTPUT-MASTER.
```

Example 3-21: Doing a Serial Search Without Using the Varying Phrase

```
PROCEDURE DIVISION.
BEGIN.
   .
   .
   .
FED-DEDUCT-COMPUTATION.
          SET IND-1 TO 1.
          SEARCH ALL FED-ALLOWANCES AT END GO TO TABLE-1-ERROR
            WHEN ALLOWANCE-NUMBER(IND-1) = NR-DEPENDENTS OF
                 OUTPUT-MASTER,
            SUBTRACT ALLOWANCE(IND-1) FROM GROSS-WAGE OF OUTPUT-MASTER
                 GIVING TAXABLE-INCOME ROUNDED.
          IF MARITAL-STATUS OF OUTPUT-MASTER = "M"
                 GO TO MARRIED.
```

Example 3-22: A Multiple-Key Binary Search

```
IDENTIFICATION DIVISION.
PROGRAM-ID. MULTI-KEY-SEARCH.
DATA DIVISION.
WORKING-STORAGE SECTION.

01 DIRECTORY-TABLE.
   05 NAMES-NUMBERS.
      10 FILLER                PIC X(30)
         VALUE "SMILEY    HAPPY     T.213-4332".
      10 FILLER                PIC X(30)
         VALUE "SMITH     ALAN      C.881-4987".
      10 FILLER                PIC X(30)
         VALUE "SMITH     CHARLES   J.345-2398".
      10 FILLER                PIC X(30)
         VALUE "SMITH     FREDERICK   745-0223".
      10 FILLER                PIC X(30)
         VALUE "SMITH     HARRY     C.573-3306".
      10 FILLER                PIC X(30)
         VALUE "SMITH     HARRY     J.295-3485".
      10 FILLER                PIC X(30)
         VALUE "SMITH     LARRY     X.976-5504".
      10 FILLER                PIC X(30)
         VALUE "SMITHWOOD ALBERT    J.349-9927".
   05 PHONE-DIRECTORY-TABLE REDEFINES NAMES-NUMBERS OCCURS 8 TIMES
                                  ASCENDING KEY IS LAST-NAME
                                                   FIRST-NAME
                                                   MID-INIT
                                  INDEXED BY DIR-INDX.
         15 LAST-NAME          PIC X(10).
         15 FIRST-NAME         PIC X(10).
         15 MID-INIT           PIC XX.
         15 PHONE-NUM          PIC X(8).

PROCEDURE DIVISION.
MULTI-KEY-BINARY-SEARCH.
    SEARCH ALL PHONE-DIRECTORY-TABLE
           WHEN LAST-NAME(DIR-INDX) = "SMITH"
           AND FIRST-NAME(DIR-INDX) = "HARRY"
           AND MID-INIT(DIR-INDX) = "J."
               NEXT SENTENCE.
DISPLAY-RESULTS.
    DISPLAY LAST-NAME(DIR-INDX)","
            FIRST-NAME(DIR-INDX)
            MID-INIT(DIR-INDX) " "
            PHONE-NUM(DIR-INDX).
```

Chapter 4
Data Handling Optimization

You can decrease processing time and save storage space by using compiler optimization features when you write your COBOL-81 programs. This chapter shows how certain numeric data types and Procedure Division statements can help you optimize your COBOL-81 programs.

4.1 Numeric Data Representation

Optimizing numeric data is the best way to improve program performance. The most efficient type of numeric data depends on the item's size and whether or not CIS (Commercial Instruction Set) is available on your machine. Table 4-1 shows the relative efficiency of COBOL-81 data types on both CIS and non-CIS machines. Refer to Chapter 1 for discussions of the individual numeric data types.

Table 4-1: Relative Efficiency of COBOL-81 Numeric Data Types

	PIC S9 to S9(9)		PIC S9(10) to S9(18)	
	CIS	**non-CIS**	**CIS**	**non-CIS**
Most Efficient	COMP	COMP	COMP-3	COMP
Intermediate	COMP-3	DISPLAY	DISPLAY	DISPLAY
Least Efficient	DISPLAY	COMP-3	COMP	COMP-3

In general, you describe numeric data items as USAGE COMP when:

- The data item(s) is part of an arithmetic operation and is less than 4 words (S9 to S9(9)).
- The data item(s) is used as a subscript. In this case, allocate 1 word by specifying PIC S9 to PIC S9(4).

Although not as storage-efficient as USAGE COMP, data items with a USAGE COMP-3 let you store two digits per byte rather than one digit per byte for USAGE DISPLAY items. For example:

USAGE DISPLAY		**USAGE COMP-3**	
PICTURE	Storage	PICTURE	Storage
S9(5)V99	7 bytes	S9(5)V99	4 bytes
S9(12)V9	13 bytes	S9(12)V9	7 bytes

To calculate the number of storage bytes for a COMP-3 item, divide the PICTURE size by 2 (without rounding) and add 1 to the result. You can check the allocation with the /SHOW:MAP compiler qualifier. In general, you describe numeric data items as USAGE COMP-3 when the size of the data item is from 10 to 18 decimal digits long, and you are using CIS.

4.1.1 Scaling and Mixing Data Types

Scaling is the process of aligning decimal points for COMP and COMP-3 numeric data items. This is important when two or more data items are to be used in an arithmetic operation. If the data items do not have the same scaling factor, they must be rescaled before the arithmetic operation can be performed. Two data items have the same scaling factor when they have an equal number of specified digits to the right of the implied decimal point. Digits specified to the left of the implied decimal point play no part in the scaling factor. For example, data items defined as PIC S9V99 and PIC S99V99 have the same scaling factor.

Whenever you know that two or more data items are to be used in an arithmetic operation, use the same data type for all. This avoids a conversion to a common data type.

For example, consider these three WORKING-STORAGE examples for the statement ADD A TO B:

Poor

```
01 A PIC S99V999   COMP.      Requires conversion and rescaling
01 B PIC S99V9999  COMP-3.
```

Improved

```
01 A PIC S99V9     COMP.      Requires rescaling
01 B PIC S99V999   COMP.
```

Best

```
01 A PIC S99V999   COMP.      No rescaling or conversion required
01 B PIC S99V999   COMP.
```

4.1.2 Significant Digits

In general, the fewer significant digits in an item, the better the performance (except as described in Section 4.1.1.). For example, for a numeric data item to contain a number from 1 to 999, declare it as PIC S9(3), not S9(10).

4.1.3 Indexing Instead of Subscripting

Using index-names for table handling is generally more efficient than using COMP-3 or numeric DISPLAY subscripts, because the compiler declares index-names as one-word binary data items. Subscript data items described in WORKING-STORAGE as one-word binary items are as efficient as index-items. Indexing also provides more flexibility in table-handling operations, because it allows you to use the SEARCH statement for sequential and binary searches. See Chapter 3 for more information about indexing and subscripting in table handling.

The efficiency for indexing and subscripting in decreasing order is:

1. Index-names or subscript data items described as one-word COMP
2. Subscript data items described as COMP-3
3. Subscript data items described as two- or four-word COMP
4. Subscript data items described as numeric DISPLAY

Example

```
WORKING-STORAGE SECTION.
01  TABLE-SIZE.
    03  FILLER                    PIC X(300).
01  THE-TABLE REDEFINES TABLE-SIZE.
    03  TABLE-ENTRY OCCURS 30 TIMES PIC X(10).
01  SUB-1                         PIC S9(4) COMP VALUE ZEROS.
```

This is as efficient as:

```
WORKING-STORAGE SECTION.
01  TABLE-SIZE.
    03  FILLER                    PIC X(300).
01  THE-TABLE REDEFINES TABLE-SIZE.
    03  TABLE-ENTRY OCCURS 30 TIMES PIC X(10)
                    INDEXED BY IND-1.
```

If applicable, use a numeric literal to access a table. For example:

```
MOVE TABLE-ENTRY (numeric literal) TO ...
```

This is faster than either of these:

```
MOVE TABLE-ENTRY (SUB-1) TO ...

MOVE TABLE-ENTRY (IND-1) TO ...
```

4.1.4 Avoid Using Decimal Truncation

The final factor in optimizing numeric data handling is to avoid decimal truncation. If performance is an important factor, do not use the /TRUNCATE qualifier when compiling your programs. Because of the additional operations required to implement decimal truncation, use of this qualifier will cause performance to slow down.

4.2 Procedure Division Statements

Some Procedure Division statements make better use of the COBOL-81 compiler than others. This section discusses these statements and shows how to use them.

4.2.1 ADD, SUBTRACT, MULTIPLY and DIVIDE Instead of COMPUTE

The ADD, SUBTRACT, MULTIPLY, and DIVIDE statements are generally faster than the COMPUTE statement; they usually execute fewer instructions. For example, consider the following separate arithmetic and COMPUTE statements using the listed record definitions.

Record definitions:

```
01 A     PIC S9(4)V99 COMP-3.
01 B     PIC S9(4)V99 COMP-3.
01 C     PIC S9(4)V99 COMP-3.
01 D     PIC S9(4)V99 COMP-3.
01 E     PIC S9(4)V99 COMP-3.
01 F     PIC S9(4)V99 COMP-3.
01 TEMP1 PIC S9(4)V99 COMP-3.
01 TEMP2 PIC S9(4)V99 COMP-3.
```

Separate arithmetic statements:

```
ADD A, B GIVING TEMP1.
ADD C, D GIVING TEMP2.
SUBTRACT TEMP2 FROM TEMP1.
DIVIDE TEMP1 BY 2 GIVING TEMP2.
MULTIPLY TEMP2 BY E GIVING F.
```

Computed statement:

```
COMPUTE F = ((((A + B) - (C + D)) / 2) * E).
```

4.2.2 GO TO DEPENDING ON Instead of IF, GO TO

The GO TO DEPENDING ON statement generates fewer instructions than a sequence of IF and GO TO statements; it can also improve a program's readability. For example:

```
GO TO 100-PROCESS-MARRIED
      200-PROCESS-SINGLE
      300-PROCESS-DIVORCED
      400-PROCESS-WIDOWED
          DEPENDING ON MARITAL-STATUS.
```

This example generates fewer instructions and is easier to read than:

```
IF MARITAL-STATUS = "1"
   GO TO 100-PROCESS-MARRIED.
IF MARITAL-STATUS = "2"
   GO TO 200-PROCESS-SINGLE.
IF MARITAL-STATUS = "3"
   GO TO 300-PROCESS-DIVORCED.
IF MARITAL-STATUS = "4"
   GO TO 400-PROCESS-WIDOWED.
```

Remember, data items referenced by the DEPENDING ON clause must contain a numeric value that is: (a) greater than zero and (b) not greater than the number of procedure-names in the statement. Otherwise, control passes to the next executable statement.

4.2.3 SEARCH ALL Instead of SEARCH

When you are performing table look-up operations, SEARCH ALL – a binary search operation – is usually faster than SEARCH – a sequential search operation. A binary search determines a table's size, finds the median table entry, and, by using compare processes, searches the table in sections. A sequential search manipulates the contents of an index to search the table sequentially. However, SEARCH ALL requires the table to be in ascending or descending order by search key, while SEARCH imposes no restrictions on table organization. Chapter 3 contains samples of binary and sequential table-handling operations.

Contents

Page

PART IV

Chapter 1 The Basics of Handling COBOL-81 Files and Records

1.1 Record Management Services 1-1
1.2 COBOL-81 File Organizations 1-2
1.3 File Attributes 1-3
1.4 COBOL-81 Record Attributes 1-3

1.4.1 Record Format 1-3

1.4.1.1 Fixed-Length Records 1-4
1.4.1.2 Variable-Length Records 1-4

1.4.2 Print-Controlled File 1-8

1.5 File Design Considerations 1-8
1.6 File Handling 1-8

1.6.1 Identifying a File from Your COBOL-81 Program 1-9

1.6.1.1 Using the VALUE OF ID Clause for Device Independence . 1-10
1.6.1.2 Using Logical Names 1-11

1.6.2 Choosing File Organization and Record Access Mode 1-11

1.6.2.1 File Organizations 1-11
1.6.2.2 Record Access Modes 1-12

1.7 Opening and Closing Files 1-13

Chapter 2 Processing Sequential Files

2.1 Sequential File Organization 2-1
2.2 Design Considerations 2-2
2.3 Statements for Sequential File Processing 2-3
2.4 Defining a Sequential File 2-3
2.5 Creating Sequential Files 2-4
2.6 Reading Sequential Files 2-6
2.7 Rewriting Records in a Sequential File 2-7
2.8 Extending Sequential Files 2-8
2.9 Backing Up Your Sequential Files 2-9

Chapter 3 Processing Relative Files

3.1 Relative File Organization 3-1
3.2 Design Considerations 3-2
3.3 Statements for Relative File Processing 3-3
3.4 Defining a Relative File 3-4
3.5 Creating Relative Files 3-4

3.5.1 Sequential Access Mode Creation 3-5
3.5.2 Random Access Mode Creation 3-5

3.6 Reading Relative Files . . . 3-6
3.6.1 Sequential Reading . . . 3-6
3.6.2 Random Reading . . . 3-7
3.6.3 Dynamic Reading . . . 3-8
3.7 Updating Relative Files . . . 3-9
3.7.1 Rewriting Relative Records . . . 3-9
3.7.1.1 Sequential Access Mode Rewriting . . . 3-10
3.7.1.2 Random Access Mode Rewriting . . . 3-11
3.7.2 Deleting Relative Records . . . 3-12
3.7.2.1 Sequential Access Mode Deletion . . . 3-12
3.7.2.2 Random Access Mode Deletion . . . 3-13
3.8 Backing Up Your Relative Files . . . 3-14

Chapter 4 Processing Indexed Files

4.1 Indexed File Organization . . . 4-1
4.2 Design Considerations . . . 4-2
4.3 Statements for Indexed File Processing . . . 4-3
4.4 Defining an Indexed File . . . 4-4
4.5 Creating and Populating Indexed Files . . . 4-4
4.6 Reading Indexed Files . . . 4-6
4.6.1 Sequential Reading . . . 4-6
4.6.2 Random Reading . . . 4-8
4.6.3 Dynamic Reading . . . 4-9
4.7 Updating Indexed Files . . . 4-10
4.7.1 Sequential Updating . . . 4-11
4.7.2 Random Updating . . . 4-12
4.7.3 Dynamic Updating . . . 4-15
4.8 Backing Up Your Indexed Files . . . 4-15

Chapter 5 Input/Output Exception Conditions Handling

5.1 Planning for the At End Condition . . . 5-2
5.2 Planning for the Invalid Key Condition . . . 5-2
5.3 Using File Status Values . . . 5-3
5.3.1 COBOL-81 File Status Values . . . 5-3
5.3.2 RMS-11 File Status Values . . . 5-5
5.4 Using Declarative Procedures to Handle Exception Conditions . . . 5-6

Chapter 6 Sharing Files and Protecting Records

6.1 File Sharing and Record Locking Concepts . . . 6-1
6.2 Ensuring Successful File Sharing . . . 6-3
6.2.1 Providing Disk Residency . . . 6-3
6.2.2 Using File Protection . . . 6-3
6.2.3 Determining the Intended Access Mode to a File . . . 6-4
6.2.4 Indicating the Access Allowed to Other Streams . . . 6-5

6.3 Describing Types of Access Streams. 6-5
6.4 Summarizing Related File-Sharing Criteria 6-6
6.5 Checking File Operations 6-8
6.6 Specifying the OPEN EXTEND with a Sequential File 6-8
6.7 Using Record Locking. 6-9

Chapter 7 File Optimization Techniques

7.1 Using the APPLY Clause. 7-1
7.2 Current Record Area 7-3
7.3 Sharing Record Areas 7-4
7.4 I/O Buffers 7-5
7.5 Reserving Additional I/O Buffer Space for Your Files 7-6
7.6 Sharing Buffer Areas 7-7
7.7 Tailoring I/O Buffers to Increase Speed of I/O Operations. 7-7
7.8 Optimizing File Design 7-8
7.8.1 Sequential Files 7-8
7.8.1.1 Buffer Size Calculations for Sequential Files 7-9
7.8.1.2 Unit of Transfer for Sequential Files on Magnetic Tape. . . 7-10
7.8.1.3 Unit of Transfer for Sequential Files on Disk 7-12
7.8.2 Relative Files 7-13
7.8.2.1 Maximum Record Number (MRN) 7-13
7.8.2.2 Cell Size 7-14
7.8.2.3 Bucket Size 7-14
7.8.2.4 File Size. 7-15
7.8.2.5 Bucket Size and Buffer Size Calculations for Relative Files 7-16
7.8.3 Indexed Files 7-18
7.8.3.1 Records 7-19
7.8.3.2 Alternate Keys 7-19
7.8.3.3 Bucket Size 7-20
7.8.3.4 Index Depth 7-21
7.8.3.5 File Size. 7-21
7.8.3.6 Alternate Key Index Level Calculations 7-23
7.8.3.7 Caching Index Roots 7-23
7.8.3.8 Bucket Size and Buffer Size Calculations for Indexed Files . 7-23

Chapter 8 Producing Printed Reports with COBOL-81

8.1 Designing the Report 8-1
8.2 Components of a Report. 8-4
8.3 The Logical Page and the Physical Page 8-6
8.3.1 Horizontal Spacing for the Logical Page. 8-6
8.3.2 Vertical Spacing for the Logical Page 8-6
8.4 Modes for Printing Reports. 8-6
8.4.1 Online Printing 8-7
8.4.2 Spooling to a Mass Storage Device 8-7

8.5 Accumulating and Reporting Totals 8-8
8.6 Programming the Conventional COBOL-81 Report 8-10
8.6.1 Defining the Logical Page 8-10
8.6.2 How to Advance to the Next Logical Page. 8-10
8.6.3 Programming for the Page-Overflow Condition. 8-11
8.6.4 Using a Line Counter 8-12
8.6.5 A Special Forms Example 8-12
8.7 Programming the Linage-File Report 8-15
8.7.1 Defining the Logical Page with the LINAGE Clause 8-15
8.7.2 Advancing to the Next Logical Page 8-17
8.7.3 Programming for the Page-Overflow Condition. 8-17
8.7.4 Using the LINAGE-COUNTER 8-21
8.7.5 A Special Forms Example 8-22
8.8 How to Print Your Report 8-24
8.8.1 Printing the Conventional Report. 8-24
8.8.2 Printing a Linage-File Report. 8-25
8.9 Solving Report Problems 8-25
8.9.1 Printing More Than One Logical Line on a Single Physical Line . . . 8-25
8.9.2 Group Indicating. 8-29
8.9.3 Fitting Reports on the Page 8-30
8.9.4 Printing Totals Before Detail Lines 8-30
8.9.5 Underlining Items in Your Reports 8-31
8.9.6 Bolding Items in Your Reports 8-31

Chapter 9 Forms for Video Terminals

9.1 Designing Your Form with ACCEPT/DISPLAY Options 9-2
9.1.1 Clearing a Screen Area 9-3
9.1.2 Horizontal and Vertical Positioning of the Cursor. 9-5
9.1.3 Assigning Character Attributes to Your Format Entries 9-8
9.1.4 Handling Data with ACCEPT Options. 9-9
9.1.4.1 Using CONVERSION with ACCEPT Data 9-9
9.1.4.2 Using ON EXCEPTION When Accepting Data with CONVERSION. 9-9
9.1.4.3 Protecting Your Screen 9-11
9.1.4.4 Using NO ECHO With ACCEPT Data 9-13
9.1.4.5 Assigning Default Values to Data Fields 9-14
9.1.5 Using Keys on Your Terminal to Define Special Program Functions 9-17
9.1.6 Using the CONVERSION Clause to Display Data. 9-24

Chapter 10 Sorting Records and Merging Files

10.1 ASCENDING and DESCENDING KEY Phrases 10-1
10.2 USING and GIVING Phrases. 10-2
10.3 INPUT PROCEDURE and OUTPUT PROCEDURE Phrases 10-2
10.4 WITH DUPLICATES IN ORDER Phrase 10-4
10.5 File Organization. 10-5
10.6 Multiple Sorts 10-5

10.7 Sorting Variable-Length Records 10-6
10.8 Preventing I/O Aborts. 10-6
10.9 The MERGE Statement 10-7
10.10 Sample Programs 10-8

Appendix A Designing Your Form with Escape Sequences

A.1 Defining the Escape. A-1
A.2 Defining Items for Line and Column Numbers A-2
A.3 Sending Control Character Sequences to the Terminal A-2
A.3.1 Sending to VT100 Terminals. A-3
A.3.2 Sending to VT52 Terminals A-4
A.3.3 Switching from VT100 Mode to VT52 Mode A-6
A.4 Examples A-7
A.4.1 VT100 Example A-7
A.4.2 VT52 Example A-8
A.4.3 Example of Double-Height and Double-Width Lines on a VT100 . . . A-10

Appendix B Logical Unit Number (LUN) Assignments

Examples

1-1 Sample Record Description 1-3
1-2 Defining a Fixed-Length Record 1-4
1-3 Defining Variable-Length Records with the DEPENDING ON Phrase 1-5
1-4 Defining Variable-Length Records with Multiple Record Descriptions 1-6
1-5 Defining Variable-Length Records with the OCCURS DEPENDING ON Phrase . . 1-6
1-6 Defining Variable-Length Records for Compatibility with VAX–11 COBOL 1-7
1-7 Defining Fixed-Length Records with Multiple Record Descriptions 1-8
1-8 Defining a Disk File 1-9
1-9 Defining a Magnetic Tape File 1-9
1-10 How to Override or Supplement a File Specification at Run Time. 1-10
1-11 Sequential File SELECT Statements 1-12
1-12 Relative File SELECT Statements 1-12
1-13 Indexed File SELECT Statements 1-13
2-1 Defining a Sequential File 2-4
2-2 Creating a Sequential File 2-5
2-3 Reading a Sequential File 2-6
2-4 Rewriting a Sequential File 2-7
2-5 Extending a Sequential File 2-8
3-1 Defining a Relative File 3-4
3-2 Creating a Relative File in Sequential Access Mode 3-5
3-3 Creating a Relative File in Random Access Mode 3-5
3-4 Sequentially Reading Relative Files 3-7
3-5 Randomly Reading a Relative File 3-8
3-6 Dynamically Reading a Relative File 3-9
3-7 Rewriting Relative Records in Sequential Access Mode 3-10
3-8 Rewriting Relative Records in Random Access Mode 3-11
3-9 Deleting Relative Records in Sequential Access Mode 3-12
3-10 Deleting Relative Records in Random Access Mode. 3-13
4-1 Defining an Indexed File 4-4

4-2 Creating and Populating an Indexed File. 4-5
4-3 Sequentially Reading an Indexed File 4-7
4-4 Randomly Reading an Indexed File 4-8
4-5 Dynamically Reading an Indexed File. 4-10
4-6 Sequentially Updating an Indexed File 4-11
4-7 Randomly Updating an Indexed File 4-13
5-1 Handling the At End Condition. 5-2
5-2 Handling the Invalid Key Condition. 5-2
5-3 Defining a File Status for a File. 5-3
5-4 Using the File Status Value in an Exception Handling Routine 5-4
5-5 Referencing RMS-STS and RMS-STV Values 5-5
5-6 The Declarative Skeleton 5-6
5-7 A Declarative Procedure Skeleton 5-6
5-8 Five Types of Declarative Procedures 5-7
6-1 Creating Two Access Streams to a File from the Same Program 6-5
6-2 Automatic Record Locking. 6-9
7-1 Sharing Record Areas 7-4
7-2 Statements Causing Three Files to Share One I/O Buffer 7-7
8-1 Directly Allocating a Printer for Immediate Use. 8-7
8-2 Spooling a Report to a Disk Pack for Later Printing 8-7
8-3 Checking for the Page-Overflow Condition 8-11
8-4 Page Advancing and Line Skipping 8-13
8-5 Checking for Page-Overflow on a 28-Line Logical Page 8-19
8-6 A Sample LINAGE Clause for a 20-Line Logical Page 8-22
8-7 Programming a 20-Line Logical Page defined by the LINAGE Clause 8-23
8-8 Printing Labels Four-Up 8-26
8-9 Printing Labels Four-Up in Sort Order. 8-28
9-1 Erasing a Screen 9-3
9-2 Cursor Positioning 9-5
9-3 Use of PLUS for Cursor Positioning 9-7
9-4 Character Attributes. 9-8
9-5 Use of ON EXCEPTION 9-10
9-6 Use of SIZE 9-12
9-7 Use of NO ECHO 9-13
9-8 Use of the DEFAULT Clause 9-15
9-9 Use of the CONTROL KEY Clause 9-21
9-10 Use of CONVERSION. 9-25
10-1 Using the ASCENDING KEY and DESCENDING KEY Phrases. 10-1
10-2 Using the USING and GIVING Phrases 10-2
10-3 Using the INPUT PROCEDURE and OUTPUT PROCEDURE Phrases 10-3
10-4 Replacing the INPUT PROCEDURE Phrase with the USING Phrase 10-4
10-5 Sorting With and Without the DUPLICATES IN ORDER Phrase 10-4
10-6 Multiple Sorts in the Same Program. 10-5
10-7 A Declarative Procedure for a Sort 10-7
10-8 Merging Two Files into One File 10-7
10-9 SORTA – Sorting with the USING and GIVING Phrases 10-8
10-10 SORTB – Sorting with the USING and OUTPUT PROCEDURE Phrases 10-9
10-11 SORTC – Sorting with the INPUT PROCEDURE and OUTPUT PROCEDURE Phrases 10-10
10-12 SORTE – Sorting a File and Expanding Its Output Records 10-12
10-13 MERGE01 – Merging Three Files 10-14

Figures

2-1 Sequential File Organization . . . 2-1
2-2 A Multiple Volume Sequential File . . . 2-2
3-1 Relative File Organization . . . 3-2
4-1 Indexed File Organization . . . 4-2
6-1 Multiple Access to a File . . . 6-1
6-2 File Sharing and Record Locking Relationship . . . 6-2
6-3 The Overwrite Condition . . . 6-9
7-1 Current Record Areas for Three Files . . . 7-4
7-2 Two Files Sharing a Record Area . . . 7-5
7-3 A Program's Buffer Area in a Task Structure . . . 7-5
7-4 Using One I/O Buffer to Process Three Files . . . 7-7
8-1 A Sample Report Layout Worksheet. . . . 8-2
8-2 Typical Report Layout. . . . 8-3
8-3 Components of a Report. . . . 8-5
8-4 Subtotals, Crossfoot Totals, and Rolled Forward Totals . . . 8-9
8-5 Logical Page Area for a Conventional Report. . . . 8-10
8-6 A 20-Line Logical Page . . . 8-12
8-7 A Double-Spaced Master Listing . . . 8-13
8-8 Logical Page Areas for a Linage-File Report . . . 8-16
8-9 A 28-Line Logical Page . . . 8-18
8-10 A 20-Line Logical Page . . . 8-22
8-11 Printing Labels Four-Up . . . 8-25
8-12 Printing Labels Four-Up in Sort Order . . . 8-27
9-1 Adding Information to a Master File with a Video Form . . . 9-2
9-2 Effects of the ERASE Option . . . 9-4
9-3 Positioning the Data on Line 19 Column 5. . . . 9-6
9-4 Cursor Positioning Using the PLUS Option. . . . 9-7
9-5 Screen Display with Character Attributes . . . 9-9
9-6 Accepting Data with the ON EXCEPTION Option. . . . 9-11
9-7 Screen Display of NUM-DATA Using the PROTECTED Option . . . 9-13
9-8 Accepting Data with the NO ECHO Option . . . 9-14
9-9 Accepting Data with the DEFAULT Option . . . 9-16
9-10 COBOL-81's Control Keys on the Standard VT52 Keypad . . . 9-18
9-11 COBOL-81's Control Keys on the Standard VT100 Keypad. . . . 9-19
9-12 COBOL-81's Control Keys on the Standard Professional Keypad . . . 9-19
9-13 Screen Display of Program SPECIAL. . . . 9-24
9-14 Sample Run of Program CONVERT . . . 9-26
A-1 Double-Height and Double-Width Lines on a VT100 . . . A-11

Tables

1-1 COBOL-81 File Organizations – Advantages and Disadvantages . . . 1-2
2-1 Valid I/O Statements for Sequential Files. . . . 2-3
3-1 Valid I/O Statements for Relative Files. . . . 3-3
4-1 Valid I/O Statements for Indexed Files. . . . 4-3
6-1 File Sharing and Intended COBOL-81 Operations. . . . 6-4
6-2 File Protection and Open Mode Requirements for File Sharing . . . 6-6
6-3 File Sharing Options . . . 6-7

6-4 COBOL-81 File Status Values and RMS-11 Completion/Error Codes 6-8
7-1 Permissible APPLY WINDOW Values 7-3
7-2 Default Buffer Areas Reserved by the COBOL-81 OTS 7-6
8-1 Results of Group Indicating 8-30
9-1 Cursor Positioning Requirements for ERASE Options 9-3
9-2 Available Character Attributes by Terminal Type 9-8
9-3 COBOL-81 Characters Returned by the OTS for Cursor Positioning, Program Function, and Auxiliary Keypad Keys 9-17
9-4 Characters Returned by the OTS for the Professional's Top Row Function and Editing Keys 9-19
B-1 Logical Unit Assignments for COBOL-81 Features and Files Open at the Same Time B-2

Chapter 1
The Basics of Handling COBOL-81 Files and Records

Input and output services require a complex management system; otherwise the programmer is left with the task of producing detailed Input/Output control for each program. With the PDP-11 operating systems, complete I/O services are provided for handling, controlling, and spooling I/O needs or requests. Record Management Services (RMS-11) gives a wide range of file management techniques while remaining transparent to you. This chapter introduces you to:

- Record Management Services
- COBOL-81 file organizations
- COBOL-81 file attributes
- COBOL-81 record attributes
- COBOL-81 record access modes
- COBOL-81 OPEN and CLOSE statements

1.1 Record Management Services

COBOL-81 provides extensive capabilities for data storage, retrieval, and modification for the COBOL-81 programmer through Record Management Services (RMS-11). You can select from one of several file organizations and access techniques – each of which is suited to a particular application – from the simplest sequential search of a sequentially organized file to a sophisticated dynamic access of an indexed file based on several alternate key fields.

The three file organizations built by COBOL-81 and RMS-11 – sequential, relative, and indexed – are variously available to three different access modes: sequential, random, and dynamic. COBOL-81 also supports access mode switching, or dynamic access, a useful feature that allows your program to switch from sequential to random access and back during file processing.

You should consider learning about how RMS-11 works and become familiar with the information contained in the RMS-11 documentation set. It discusses the following topics:

1. Application design
2. File design
3. Task design
4. Common optimization techniques
5. RMS-11 utilities
6. Magnetic tape handling

If you do not learn how RMS-11 works, you probably will not get the best performance possible from your application. You can even get less performance because of the defaults you're accepting without knowing it.

1.2 COBOL-81 File Organizations

Table 1-1 lists the three file organizations available to you and includes their advantages and disadvantages. Chapters 2, 3, and 4 further discuss each of these file organizations.

Table 1-1: COBOL-81 File Organizations – Advantages and Disadvantages

File Organizations	Advantages and Disadvantages	
Sequential	Advantages	Uses disk and memory efficiently Provides optimal usage if the application accesses all records sequentially on each run Provides the most flexible record format Allows data to be stored on many types of media, in a device-independent manner Allows easy file extension
	Disadvantages	Allows sequential access only Allows records to be added only to the end of a file Allows write access by multiple, concurrent users, but only in very restricted cases
Relative	Advantages	Allows sequential, random, and dynamic access Provides random record deletion and insertion Allows records to be read- and write-shared Requires that files contain a record cell for each record stored in the file; that is, files may not be densely populated because not all record cells may be used
	Disadvantages	Allows data to be stored on disk only Requires that record cells be the same size
Indexed	Advantages	Allows sequential, random, and dynamic access modes Allows random record deletion and insertion Allows records to be read- and write-shared Allows variable-length records to change length on update Allows easy file extension
	Disadvantages	Allows data to be stored on disk only Requires more disk space Uses more memory to process records Generally requires multiple disk accesses to randomly process a record

1.3 File Attributes

In COBOL-81 programs, you specify a file's attributes in the Environment and Data Divisions:

- The SELECT statement specifies the file organization
- File description entries specify record format and record blocking
- Record description entries specify physical record size(s)

Chapters 2, 3, and 4 present and discuss examples of each type of file organization supported by COBOL-81.

Your system uses these attributes to create a file and stores them with the file. When a program accesses a file, it should specify the same attributes stored when the file was created. For example, a program cannot read a sequential file as an indexed file, because no index keys exist.

1.4 COBOL-81 Record Attributes

A record is a group of related data elements. The space a record needs on a physical device depends on:

- The file organization
- The record format
- The number of bytes the record contains

If a file has more than one record description, the records automatically share the same record area in memory. The Object- or Run-Time System does not clear this area before it executes the READ statement. Therefore, if the record read by the latest READ statement does not fill the entire record area, the area not overlaid by the incoming record remains unchanged.

1.4.1 Record Format

The compiler determines record format from a combination of record description entries and the RECORD CONTAINS clause.

In Example 1-1, a file contains a company's stock information (part number, supplier, quantity, price.) Within this file, the information is divided into records. All information for a single piece of stock constitutes a single record.

Each record in the stock file is itself divided into discrete pieces of information known as elementary items. You give the item a specific location in the record, give it a name, and define its size. The part number is an item in the part record, as is supplier, quantity, and price. In this example, PART-RECORD defines four elementary items: PART-NUMBER, PART-SUPPLIER, PART-QUANTITY, and PART-PRICE.

Example 1-1: Sample Record Description

```
01  PART-RECORD.
    02  PART-NUMBER                   PIC 9999.
    02  PART-SUPPLIER                 PIC X(20).
    02  PART-QUANTITY                 PIC 99999.
    02  PART-PRICE                    PIC S9(5)V99.
```

You can completely control the grouping of elementary items into records and records into files. COBOL-81 programs either build records and pass them to RMS-11 for storage in a file, or they issue requests for records while RMS-11 performs the necessary operations to retrieve the records from a file.

The maximum size of a record depends on its format:

- For fixed-length records, the maximum size is the record size
- For variable-length records, the maximum size is the size of the largest record plus the number of overhead bytes needed by the storage medium

In either case, the length of any record in a file description entry cannot exceed 16384 bytes.

1.4.1.1 Fixed-Length Records — Files with a fixed-length record format contain the same size records. The compiler generates the fixed-length format when all of the following conditions are true:

1. The file has only one record description. If the file has multiple record descriptions, the largest record description determines record size.
2. The file description does not contain a RECORD CONTAINS phrase or a RECORD VARYING phrase.
3. The program does not specify a print-controlled file by referring to the file with:
   a. The ADVANCING phrase in a WRITE statement
   b. An APPLY PRINT-CONTROL clause in the Environment Division
   c. A LINAGE clause in the file description

Fixed-length record size is determined by either the record description or the record size specified by the RECORD CONTAINS phrase, whichever is larger. In Example 1-2, the RECORD CONTAINS phrase specifies a record size larger than the record description; therefore, record size is 100 characters.

Example 1-2: Defining a Fixed-Length Record

```
FD  FIXED-FILE
    RECORD CONTAINS 100 CHARACTERS.
01  FIXED-REC      PIC X(75).
```

1.4.1.2 Variable-Length Records — Files with a variable-length record format can contain different length records. The system stores the record's size in bytes in a record-length field that precedes each record.

- For disk files, the record-length field is a 2-byte value specifying record length in bytes. A record's length does not include this 2-byte field, however.
- For ANSI magnetic tape files, the record-length field is a 4-byte decimal value specifying record length in bytes. A record's length does not include this 4-byte field.

The compiler generates the variable-length attribute for a file when the file description contains a RECORD VARYING phrase or a RECORD CONTAINS phrase. (See also Section 1.4.2, Print-Controlled Files.)

Examples 1-3, 1-4, and 1-5 show you the three ways COBOL-81 lets you create a variable-length record file.

In Example 1-3, the DEPENDING ON phrase sets the OUT-REC record length. The IN-TYPE data field determines the OUT-LENGTH field's contents.

Example 1-3: Defining Variable-Length Records with the DEPENDING ON Phrase

```
FILE SECTION.
FD  INFILE
    LABEL RECORDS ARE STANDARD.
01  IN-REC.
    03  IN-TYPE       PIC X.
    03  REST-OF-REC   PIC X(499).
FD  OUTFILE
    RECORD VARYING FROM 200 TO 500 CHARACTERS
    DEPENDING ON OUT-LENGTH.
01  OUT-REC           PIC X(500).
WORKING-STORAGE SECTION.
01  OUT-LENGTH        PIC 999 COMP VALUE ZEROES.
PROCEDURE DIVISION.
000-OPEN-FILES.
    OPEN INPUT INFILE OUTPUT OUTFILE.
010-READ-INPUT.
    READ INFILE AT END
        CLOSE INFILE OUTFILE STOP RUN.
    IF IN-TYPE = "A"
*************************************
* Output is a 200-character record. *
*************************************
        MOVE 200 TO OUT-LENGTH
        WRITE OUT-REC FROM IN-REC
        GO TO 010-READ-INPUT.
    IF IN-TYPE = "B"
*************************************
* Output is a 300-character record. *
*************************************
        MOVE 300 TO OUT-LENGTH
        WRITE OUT-REC FROM IN-REC
        GO TO 010-READ-INPUT.
    IF IN-TYPE = "C"
*************************************
* Output is a 400-character record. *
*************************************
        MOVE 400 TO OUT-LENGTH
        WRITE OUT-REC FROM IN-REC
        GO TO 010-READ-INPUT.
    IF IN-TYPE = "D"
*************************************
* Output is a 500-character record  *
*************************************
        MOVE 500 TO OUT-LENGTH
        WRITE OUT-REC FROM IN-REC
        GO TO 010-READ-INPUT.
    DISPLAY "INVALID RECORD TYPE " IN-TYPE.
    DISPLAY "INPUT RECORD IS BYPASSED"
    GO TO 010-READ-INPUT.
```

In Example 1-4, the length of the output record determines the record length.

Example 1-4: Defining Variable-Length Records with Multiple Record Descriptions

```
FILE SECTION.
FD  FILE-A
    LABEL RECORDS ARE STANDARD.
01  A-RECORD          PIC X(200).
FD  FILE-B
    LABEL RECORDS ARE STANDARD.
01  B-RECORD          PIC X(500).
FD  OUTFILE
    RECORD VARYING FROM 200 TO 500 CHARACTERS.
01  OUT-REC-1         PIC X(200).
01  OUT-REC-2         PIC X(500).
PROCEDURE DIVISION.
000-OPEN-FILES.
    OPEN INPUT FILE-A FILE-B OUTPUT OUTFILE.
010-READ-FILE-A.
    READ FILE-A AT END
       CLOSE FILE-A
       GO TO 020-READ-FILE-B.
    WRITE OUT-REC-1 FROM A-RECORD.
    GO TO 010-READ-FILE-A.
020-READ-FILE-B.
    READ FILE-B AT END
       CLOSE FILE-B OUTFILE STOP RUN.
    WRITE OUT-REC-2 FROM B-RECORD.
    GO TO 020-READ-FILE-B.
```

Example 1-5 creates variable-length records by using the OCCURS DEPENDING ON phrase in the record description. COBOL-81 determines record length by adding the sum of the variable record's fixed portion to the size of the table described by the number of table occurrences at execution time.

In this example, the variable record's fixed portion size is 113 characters. (This is the sum of P-PART-NUM, P-PART-INFO, and P-BIN-INDEX.) If P-BIN-INDEX contains a 7 at execution time, P-BIN-NUMBER would be 35 characters long. Therefore, PARTS-REC's length would be 148 characters; the fixed portion's length is 113 characters, and the table entry's length at execution time is 35 characters.

Example 1-5: Defining Variable-Length Records with the OCCURS DEPENDING ON Phrase

```
FILE SECTION.
FD  TRANS-FILE
    LABEL RECORDS ARE STANDARD.
01  TRANS-REC.
    03 T-PART-NUM      PIC X(10).
    03 T-PART-INFO     PIC X(100).
    03 T-BIN-NUMBER    PIC X(5).
FD  PARTS-MASTER
    RECORD VARYING 118 TO 163 CHARACTERS.
01  PARTS-REC.
    03 P-PART-NUM      PIC X(10).
    03 P-PART-INFO     PIC X(100).
    03 P-BIN-INDEX     PIC 999.
    03 P-BIN-NUMBER    PIC X(5)
        OCCURS 1 TO 10 TIMES DEPENDING ON P-BIN-INDEX.
WORKING-STORAGE SECTION.
01  INITIAL-READ       PIC X VALUE "Y".
PROCEDURE DIVISION.
```

(continued on next page)

Example 1-5: Defining Variable-Length Records with the OCCURS DEPENDING ON Phrase (Cont.)

```
000-OPEN-FILES.
    OPEN INPUT TRANS-FILE OUTPUT PARTS-MASTER.
010-READ-TRANS.
    READ TRANS-FILE AT END
        WRITE PARTS-REC
        CLOSE TRANS-FILE PARTS-MASTER STOP RUN.
    IF INITIAL-READ = "Y"
        MOVE ZEROES TO P-BIN-INDEX
        MOVE "N" TO INITIAL-READ
        PERFORM 040-SETUP-PARTS-REC
        GO TO 010-READ-TRANS.
020-COMPARE-PART-NUMBER.
    IF T-PART-NUM = P-PART-NUM
        ADD 1 TO P-BIN-INDEX
        MOVE T-BIN-NUMBER TO P-BIN-NUMBER (P-BIN-INDEX)
        GO TO 010-READ-TRANS.
030-UNEQUAL-PART-NUMBER.
    WRITE PARTS-REC.
    MOVE SPACES TO PARTS-REC.
    MOVE ZEROES TO P-BIN-INDEX.
    PERFORM 040-SETUP-PARTS-REC.
    GO TO 010-READ-TRANS.
040-SETUP-PARTS-REC.
    MOVE T-PART-NUM TO P-PART-NUM.
    MOVE T-PART-INFO TO P-PART-INFO.
    ADD 1 TO P-BIN-INDEX.
    MOVE T-BIN-NUMBER TO P-BIN-NUMBER (P-BIN-INDEX).
```

If you describe a record with both the RECORD VARYING...DEPENDING ON clause and the OCCURS DEPENDING ON clause, COBOL-81 specifies record length as the value of data-name-1.

If your variable-length record requirements include compatibility with COBOL-74, the system generates variable-length attributes when you use the RECORD CONTAINS...CHARACTERS clause in place of VAX–11 COBOL's RECORD VARYING clause. For example:

Example 1-6: Defining Variable-Length Records for Compatibility with VAX–11 COBOL

```
FD  PARTS-MASTER
    RECORD CONTAINS 200 TO 500 CHARACTERS.
01  PARTS-REC-1     PIC X(200).
01  PARTS-REC-2     PIC X(300).
01  PARTS-REC-3     PIC X(400).
01  PARTS-REC-4     PIC X(500).
PROCEDURE-DIVISION.
    .
    .
    .
100-WRITE-REC-1.
    MOVE IN-REC TO PARTS-REC-1.
    WRITE PARTS-REC-1.
    GO TO ...
200-WRITE-REC-2.
    MOVE IN-REC TO PARTS-REC-2.
    WRITE PARTS-REC-2.
    GO TO ...
    .
    .
    .
```

If you have multiple record-length descriptions for a file and omit either the RECORD VARYING clause or the RECORD CONTAINS clause, all records written to the file will have a length equal to the length of the longest record described for the file. See Example 1-7.

Example 1-7: Defining Fixed-Length Records with Multiple Record Descriptions

```
FD PARTS-MASTER.
01  PARTS-REC-1    PIC X(200).
01  PARTS-REC-2    PIC X(300).
01  PARTS-REC-3    PIC X(400).
01  PARTS-REC-4    PIC X(500).
    .
    .
    .
PROCEDURE DIVISION.
    .
    .
    .
100-WRITE-REC-1.
    MOVE IN-REC TO PARTS-REC-1.
    WRITE PARTS-REC-1.
    GO TO ...
200-WRITE-REC-2.
    MOVE IN-REC TO PARTS-REC-2.
    WRITE PARTS-REC-2
    GO TO ...
    .
    .
    .
```

Writing PARTS-REC-1, PARTS-REC-2, or PARTS-REC-3 produces records equal in length to the longest record, PARTS-REC-4.

1.4.2 Print-Controlled File

Print-controlled files contain form-advancing information with each record. COBOL-81 places explicit form-control bytes directly into the file. Therefore, any COBOL-81 program trying to read a print-controlled file can have unpredictable results.

If you use the WRITE AFTER ADVANCING, the LINAGE, or the APPLY PRINT-CONTROL statement, the compiler generates variable-length print-controlled records.

1.5 File Design Considerations

The importance of design is proportional to the complexity of the file organization. That is, design is least important for applications using sequential organization, more important for relative organization, and most important for indexed organization. Chapters 2, 3, and 4 discuss file design for sequential, relative, and indexed files, respectively.

1.6 File Handling

Before your program can perform I/O on a file, it must identify the file to the operating system, specify the file's organization and access modes, and make the file available by opening it. A program must follow these steps whenever creating the file or processing one that has already been created.

1.6.1 Identifying a File from Your COBOL-81 Program

A file description entry defines a file's logical structure and associates the file with a file name that is unique within the program. The program uses this file name in the OPEN, READ, START, DELETE, and CLOSE statements.

You must establish a link between the file name your program uses and the file specification that RMS-11 uses. The SELECT and ASSIGN clauses do this. Together these clauses define a file connector. A file connector is storage area that contains information about a file. It links:

- A file name and a physical file
- A file name and its associated record area

The program must include a SELECT clause, followed by an ASSIGN clause, for every file description entry (FD) it contains. The file name you specify in the SELECT clause must match the file name in the file description entry. In the ASSIGN clause following a SELECT clause, you specify a literal that associates the file name with a file specification. This literal can be a complete file specification or one that relies on operating system defaults.

To understand the relationships between the SELECT clause, the ASSIGN clause, and the FD entry, consider two examples. In Example 1-8, because the file name specified in the FD entry of Example 1-8, DAT-FILE, all I/O statements in the program referring to that file must use the name DAT-FILE. RMS-11 uses the ASSIGN clause to interpret DAT-FILE as REPORT.DAT and requires that REPORT.DAT be in the account under which the program is running.

Example 1-8: Defining a Disk File

```
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT DAT-FILE
           ASSIGN TO "REPORT.DAT"
           .
           .
           .
DATA DIVISION.
FD  DAT-FILE
           .
           .
           .
```

The I/O statements in Example 1-9 refers to MYFILE-PRO, which the ASSIGN clause identifies to the operating system as MARCH.311. Additionally, the operating system looks for the file in directory [2,202] on the magnetic tape mounted on MM1:.

Example 1-9: Defining a Magnetic Tape File

```
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT MYFILE-PRO
           ASSIGN TO "MM1:[2,202]MARCH.311"
           .
           .
           .
```

(continued on next page)

Example 1-9: Defining a Magnetic Tape File (Cont.)

```
DATA DIVISION.
FD  MYFILE-PRO
           .
           .
           .
PROCEDURE DIVISION.
A000-BEGIN.
    OPEN INPUT MYFILE-PRO.
           .
           .
           .
    READ MYFILE-PRO.
           .
           .
           .
    CLOSE MYFILE-PRO.
```

1.6.1.1 Using the VALUE OF ID Clause for Device Independence — If the file specification is subject to change, it is inconvenient to edit the ASSIGN clause and recompile the program every time you run it. To avoid this problem, you can use a partial file specification in the ASSIGN clause and complete it by using the optional VALUE OF ID clause of the FD entry.

The VALUE OF ID clause completes or overrides the file specification in the ASSIGN clause. This lets you keep the file specification a variable until run time.

Example 1-10 illustrates how to use the VALUE OF ID clause to complete a partial file specification MARCH, with operator input. Notice how the Procedure Division statements prompt the operator for a file specification. This technique allows:

- Maximum flexibility for file access. The operator can override any part of the file specification in the ASSIGN clause.
- Maximum use of system hardware. The operator can mount a tape (or any other volume) on any available tape drive and direct the program to it.
- Maximum use of computer operator and operating system. The operator and operating system no longer have to wait for one job to finish using a specific tape drive before the next job can be started.

Example 1-10: How to Override or Supplement a File Specification at Run Time

```
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT MYFILE-PRO
           ASSIGN TO "MARCH"
           .
           .
           .
DATA DIVISION.
FILE SECTION.
FD  MYFILE-PRO
********************************************
*
    VALUE OF ID IS USER-EXTENSION.
*
********************************************
           .
           .
           .
```

(continued on next page)

Example 1-10: How to Override or Supplement a File Specification at Run Time (Cont.)

```
WORKING-STORAGE SECTION.

*****************************************
*
01  USER-EXTENSION      PIC X(20).
*
*****************************************

PROCEDURE DIVISION.
A000-BEGIN.

*****************************************
*
    DISPLAY "Enter file specification".
    ACCEPT USER-EXTENSION.
*
*****************************************
    OPEN INPUT MYFILE-PRO.
            .
            .
            .
    READ MYFILE-PRO.
            .
            .
            .
    CLOSE MYFILE-PRO.
```

1.6.1.2 Using Logical Names — Another option you have when using the ASSIGN clause is to use a logical name in the file specification. For system specific information on the use of logical names, refer to Part I, Chapter 1, of your *COBOL-81 User's Guide*.

1.6.2 Choosing File Organization and Record Access Mode

Your program always specifies – either explicitly or implicitly – a file's organization and access mode before the program opens the file. The ORGANIZATION and ACCESS clauses of the FILE-CONTROL paragraph, if present, specify these two attributes. If these clauses are not present, the compiler assumes sequential organization and sequential access.

1.6.2.1 File Organizations — COBOL-81 supports three types of file organizations:

- ORGANIZATION IS SEQUENTIAL – This organization is useful for programs that normally access each record, as in a payroll or mailing list file.
- ORGANIZATION IS RELATIVE – This organization lets you access records randomly according to their key values (relative record numbers).

  This organization is less flexible than indexed organization because you cannot insert a record in the middle of your file unless you have an empty cell to contain it.
- ORGANIZATION IS INDEXED – This organization lets you access records randomly according to their key values. Therefore, it is a useful way to organize a file in which records will be added, changed, or deleted upon demand.

If you do not use the ORGANIZATION clause, COBOL-81 assumes the file organization is sequential.

1.6.2.2 Record Access Modes — The methods for retrieving and storing records in a file are called record access modes. COBOL-81 supports three types of record access modes:

1. ACCESS MODE IS SEQUENTIAL

   - With sequential files, sequential access retrieves the records in the same sequence established by the WRITE statements that created the file.

   - With relative files, sequential access retrieves the records in the order of ascending record key values (or relative record numbers).

   - With indexed files, sequential access retrieves records in the order of ascending record key values.

2. ACCESS MODE IS RANDOM – The value of the record key your program specifies indicates the record to be accessed.

3. ACCESS MODE IS DYNAMIC – This access mode allows you to switch from sequential access mode to random access mode and back to sequential access mode while processing a file. You can switch back and forth as much as you like; the only limitation is that the file must support the selected access mode.

If you do not use the ACCESS clause, COBOL-81 assumes sequential access.

A different access mode can be used to process records within the file each time it is opened. A program can also change access mode during the processing of its file. Chapters 2, 3, and 4 discuss the access mode(s) applicable to sequential, relative, and indexed file organization, respectively.

Example 1-11 shows a sample SELECT statement for a sequential file with sequential access modes:

Example 1-11: Sequential File SELECT Statements

```
              (1)                                   (2)

FILE-CONTROL.                               FILE-CONTROL.
    SELECT LIST-FILE                            SELECT PAYROLL
           ASSIGN TO "MAIL.LIS"                        ASSIGN TO "PAYROL.DAT".
           ORGANIZATION IS SEQUENTIAL
           ACCESS IS SEQUENTIAL.
```

COBOL-81 assumes sequential organization and sequential access unless you specify otherwise.

Sample statements for a relative file are shown in Example 1-12.

Example 1-12: Relative File SELECT Statements

```
              (1)                                   (2)

FILE-CONTROL.                               FILE-CONTROL.
    SELECT MODEL                                SELECT PARTS
           ASSIGN TO "ACTOR.DAT"                       ASSIGN TO "PARTS.DAT"
           ORGANIZATION IS RELATIVE                    ORGANIZATION IS RELATIVE
           ACCESS IS SEQUENTIAL.                       ACCESS IS DYNAMIC
                                                       RELATIVE KEY IS PART-NO.
```

Sample statements for an indexed file are shown in Example 1-13.

Example 1-13: Indexed File SELECT Statements

```
             (1)                                         (2)

FILE-CONTROL.                                 FILE-CONTROL.
    SELECT GROUP                                 SELECT TEAS
           ASSIGN TO "RFCBA.PRO"                        ASSIGN TO "TETLY"
           ORGANIZATION IS INDEXED                      ORGANIZATION IS INDEXED
           ACCESS MODE IS DYNAMIC                       RECORD KEY IS LEAVES.
           RECORD KEY IS WRITER
           ALTERNATE RECORD KEY IS EDITOR.
```

Because the default organization is sequential, both the relative and indexed examples require the ORGANIZATION clause.

1.7 Opening and Closing Files

A COBOL-81 program must open a file with the OPEN statement before any other I/O statement can reference it. Files can be opened more than once in the same program as long as they are closed before the second and subsequent opens.

Opening a file allocates the buffers, creates or checks the file labels, and positions the I/O device to the start of the file. Closing a file writes out any remaining records in the output buffers, writes an end-of-file label on magnetic output files, and optionally rewinds and/or locks magnetic tape files. Examples of OPEN and CLOSE statements are:

```
OPEN INPUT MASTER-FILE.

OPEN OUTPUT REPORT-FILE.

OPEN I-O    MASTER-FILE
            TRANS-FILE
     OUTPUT REPORT-FILE.

CLOSE MASTER-FILE.

CLOSE TRANS-FILE
      REPRT-FILE.
```

The OPEN statement must specify one of four open modes: INPUT, OUTPUT, I-O, or EXTEND. Your choice, along with the file's organization and access mode, determines which I/O statements you can use. Sections 2.3, 3.3, and 4.3 discuss the I/O statements for sequential, relative, and indexed files respectively.

When your program performs an OPEN statement, the following steps take place:

1. RMS-11 builds a file specification by using the contents of the VALUE OF ID clause, if any, to alter or complete the file specification in the ASSIGN clause.
2. If the file was named in a SAME AREA clause, the Run-Time System checks the status of all other files named in the clause. If any are open, the OPEN statement fails.
3. The Run-Time System checks the file's current status. If the file is open, or if it was closed WITH LOCK, the OPEN statement fails.

4. If the file attributes specified by the program attempting an OPEN operation (INPUT, I-O, or EXTEND) differ from the attributes specified when the file was created, the OPEN statement fails.

5. If the file specification names an invalid device, or contains any other errors, the Run-Time System generates an error message and the OPEN statement fails.

6. The Run-Time System takes one of the following actions if it cannot find the file:

   a. If the file's OPEN mode is OUTPUT, it creates the file.

   b. If the file's OPEN mode is EXTEND, or I-O with RANDOM or DYNAMIC access, it creates the file.

   c. If the file's OPEN mode is INPUT, and its SELECT clause includes the OPTIONAL phrase, the OPEN statement is successful. The first read on that file causes the AT END condition.

   d. If none of the previous conditions is met, the OPEN fails and the USE procedure (if any) gets control. If no USE procedure exists, the Run-Time System aborts the program.

7. If the file's OPEN mode is OUTPUT, and a file by the same name already exists, the pre-existing file is replaced.

After the program successfully opens the file, the Run-Time System enables or disables all I/O statements that refer to the file, depending on the file organization, access mode, and open mode.

RMS-11 creates an end-of-file marker whenever a program closes a sequential file. If the file is on magnetic tape, RMS-11 rewinds it. To close a tape without rewinding it, use the NO REWIND phrase. This speeds processing when another file is to be written beyond the end of the first file. For example:

```
CLOSE MASTER-FILE NO REWIND.
```

You can also close a file and prevent it from being open again by the program in the same run. For example:

```
CLOSE MASTER-FILE WITH LOCK.
```

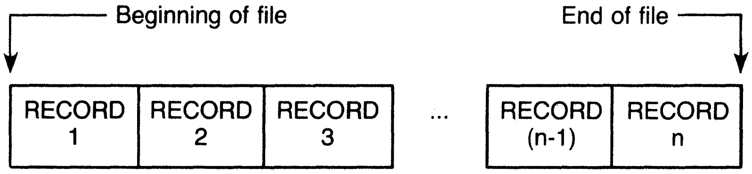
Chapter 2
Processing Sequential Files

Sequential input/output, in which records are written and read in sequence, is the simplest and most common form of I/O. It can be performed on all I/O devices, including magnetic tape, disk, terminals, and line printers.

2.1 Sequential File Organization

In sequential file organization, records are arranged consecutively in the order in which they were written to the file. Figure 2-1 illustrates sequential file organization.

Figure 2-1: Sequential File Organization



C81ART-20200-10

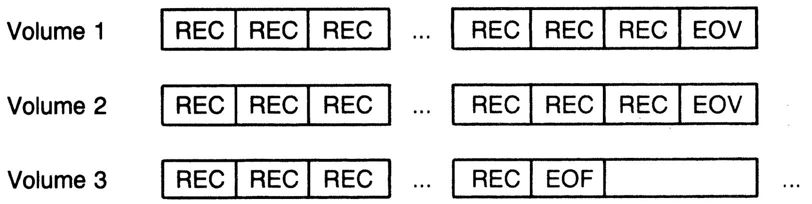
Sequential files always contain an end-of-file mark that designates the end of the file. COBOL-81 statements can write over the end-of-file mark and, thus, extend the length of the file. (RMS-11 inserts another end-of-file mark after the last record written.) Since the end-of-file mark indicates the end of useful data, COBOL-81 provides no method for reading beyond the end-of-file mark, even though the amount of space reserved for the file exceeds the amount actually used.

Occasionally a file with sequential organization is so large that it requires more than one volume, such as a multiple reel magnetic tape file. An end-of-volume label marks the end of recorded information on each volume and signals the file system to switch to a new volume. On multiple volume files, the end-of-file mark appears only once, at the end of the last record on the last volume. See Figure 2-2.

Note

RSTS/E does not support multiple volume files.

Figure 2-2: A Multiple Volume Sequential File


Volume 1
REC REC REC ... REC REC REC EOV
Volume 2
REC REC REC ... REC REC REC EOV
Volume 3
REC REC REC ... REC EOF ...

C81ART-20210-15

2.2 Design Considerations

With sequential files, design considerations include the selection of both:

1. Record format (See Chapter 1)
   - Fixed-length
   - Variable-length
2. Medium type – Sequential files can be accessed on disk, magnetic tape, and unit record devices (for example: printers and card readers). When you select the medium for your file, consider the following:
   - Speed of access – tape is significantly slower than disk.
   - Frequency of use – use tape to store files and save your disk for more immediate purposes.
   - Transportability – use tape files if you need to use the file across systems (RSTS/E disk structure is not compatible with RSX-11M/M-PLUS or VAX/VMS).
3. Allocation – at time of file creation and file extension
4. Compiler limitations

For more information on sequential file design, see Chapter 7, File Optimization Techniques, and the *RMS-11 User's Guide*.

2.3 Statements for Sequential File Processing

Processing a sequential file involves:

1. Opening the file with the OPEN statement
2. Processing the file with valid I/O statements
3. Closing the file with the CLOSE statement

Table 2-1 lists the valid I/O statements and illustrates the following relationships:

- Organization determines valid access modes.
- Organization and access mode determine valid open modes.
- All three (organization, access, and open mode) enable or disable I/O statements.

Table 2-1: Valid I/O Statements for Sequential Files

			Open Mode			
File Organization	**Access Mode**	**Statement**	**INPUT**	**OUTPUT**	**I-O**	**EXTEND**
SEQUENTIAL	SEQUENTIAL	READ REWRITE WRITE	Yes No No	No No Yes	Yes Yes No	No No Yes

2.4 Defining a Sequential File

Each sequential file in a COBOL-81 program is given a name, or file name, in a separate SELECT clause in the Environment Division. Refer to Example 2-1 for these file names: MASTER-FILE, TRANS-FILE, and REPRT-FILE. These names are referred to by statements in the COBOL-81 program.

The ASSIGN clause associates the file name with a file specification. The file specification points the operating system to the file's physical and logical location on a specific hardware device. For example:

1. MASTER-FILE is located on disk unit DB1:, directory [1,10], and is called MASTER.DAT.
2. TRANS-FILE is located on magnetic tape unit 1, and is called TRANS.DAT.
3. REPRT-FILE is the line printer.

Each file is then further described in the program with a file description (FD) entry in the File Section of the Data Division; for example, MASTER-FILE, TRANS-FILE, and REPRT-FILE. The FD entry is then followed immediately by the file's record description; for example, MASTER-RECORD, TRANSACTION-RECORD, and REPORT-LINE.

You need not specify either the ORGANIZATION IS SEQUENTIAL phrase or the ACCESS MODE IS SEQUENTIAL phrase in the SELECT clause. COBOL-81 assumes sequential organization and sequential access mode unless you specify otherwise.

Example 2-1: Defining a Sequential File

```
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT   MASTER-FILE   ASSIGN   TO   "DB1:[1,10]MASTER.DAT".
    SELECT   TRANS-FILE    ASSIGN   TO   "MM1:TRANS.DAT".
    SELECT   REPRT-FILE    ASSIGN   TO   "LP0:".

DATA DIVISION.
FILE SECTION.

FD  MASTER-FILE...
01  MASTER-RECORD.
    02  MASTER-DATA       PIC X(80).
    02  MASTER-SIZE       PIC 99.
    02  MASTER-TABLE      OCCURS 0 to 50 TIMES
                          DEPENDING ON MASTER-SIZE.
        03  MASTER-YEAR   PIC 99.
        03  MASTER-COUNT  PIC S9(5)V99.

FD  TRANS-FILE...
01  TRANSACTION-RECORD    PIC X(25).

FD  REPRT-FILE...
01  REPORT-LINE           PIC X(132).
```

2.5 Creating Sequential Files

A program creates a sequential file by:

1. Opening the file as OUTPUT or EXTEND
2. Executing the WRITE statement

Each WRITE statement releases a logical record to the end of an output file, thereby creating an entirely new record in the file. The WRITE statement releases records to files that are OPEN:

- OUTPUT – The output mode can create these two kinds of files:
    1. Storage files – A storage file remains on tape or disk for future reference or processing.
    2. Print files – The LINAGE clause, APPLY PRINT-CONTROL clause, or the ADVANCING phrase in the WRITE statement designates a file as a print file. One or more records containing carriage-control characters are written to perform line spacing. The WRITE statement does not have to release print files directly to a storage file. It can release them directly to the printer for immediate printing. A storage file can also be a print file.
- EXTEND – The extend mode adds new records in sequence after the last record of the file (see Section 2.8).

There are two ways of writing records:

1. WRITE record-name FROM source-area
2. WRITE record-name

However, the first way is best used for program readability when working with multiple record types. For example, statements (1) and (2) in this example are logically equivalent:

```
FILE SECTION.
FD   STOCK-FILE.
01   A-STOCK-RECORD      PIC X(80).
01   B-STOCK-RECORD      PIC X(80).

WORKING-STORAGE SECTION.
01   STOCK-WORK       PIC X(80).
```

```
                  (1)                                     (2)

WRITE A-STOCK-RECORD FROM STOCK-WORK.    MOVE STOCK-WORK TO A-STOCK-RECORD.
                                         WRITE A-STOCK-RECORD.
```

When you omit the FROM phrase, you process the records directly in the record area or buffer (for example, A-STOCK-RECORD).

The following example writes the record PRINT-LINE to the device assigned to that record's file, then skips three lines. When it reaches the end of the page (as specified by the LINAGE clause), it causes program control to transfer to HEADER-ROUTINE.

```
WRITE PRINT-LINE BEFORE ADVANCING 3 LINES
      AT END-OF-PAGE PERFORM HEADER-ROUTINE.
```

For a WRITE statement, if the source area is shorter than the file's record length, the source area is padded on the right with spaces; if longer, the source area is truncated on the right.

Example 2-2 creates a sequential file.

Example 2-2: Creating a Sequential File

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SEQ01.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT   TRANS-FILE    ASSIGN   TO   "TRANS.DAT".
DATA DIVISION.
FILE SECTION.

FD   TRANS-FILE.
01   TRANSACTION-RECORD    PIC X(25).

PROCEDURE DIVISION.
A000-BEGIN.
    OPEN OUTPUT TRANS-FILE.
    PERFORM  A010-PROCESS-TRANS UNTIL TRANSACTION-RECORD = "END".
    CLOSE TRANS-FILE.
    STOP RUN.
A010-PROCESS-TRANS.
    DISPLAY "Enter next record  - X(25)"
    DISPLAY "Enter END to terminate the session"
    DISPLAY "-------------------------"
    ACCEPT TRANSACTION-RECORD
    IF TRANSACTION-RECORD NOT = "END"
       WRITE TRANSACTION-RECORD.
```

2.6 Reading Sequential Files

To read a sequential file you must:

1. Open the file as INPUT or I-O
2. Execute the READ statement

Each READ statement reads a single logical record and makes its contents available to the program in the record area. There are two ways of reading records:

1. READ file-name INTO destination-area
2. READ file-name

For example, statements (1) and (2) in this example are logically equivalent:

```
FILE SECTION.
FD  STOCK-FILE.
01  STOCK-RECORD      PIC X(80).

WORKING-STORAGE SECTION.
01  STOCK-WORK        PIC X(80).


                 (1)                                 (2)

READ STOCK-FILE INTO STOCK-WORK.   READ STOCK-FILE.
                                   MOVE STOCK-RECORD TO STOCK-WORK.
```

When you omit the INTO phrase you process the records directly in the record area or buffer; for example, STOCK-RECORD. The record is also available in the record area if you use the INTO phrase.

In a READ, if the destination area is shorter than the length of the record being read, the record is truncated on the right; if longer, the destination area is filled on the right with blanks.

Example 2-3 reads a sequential file and displays its contents on the terminal.

Example 2-3: Reading a Sequential File

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SEQ02.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT   TRANS-FILE    ASSIGN   TO    "TRANS.DAT".
DATA DIVISION.
FILE SECTION.

FD  TRANS-FILE.
01  TRANSACTION-RECORD    PIC X(25).

PROCEDURE DIVISION.
A000-BEGIN.
    OPEN INPUT TRANS-FILE.
    PERFORM A100-READ-TRANS-FILE UNTIL TRANSACTION-RECORD = "END".
    CLOSE TRANS-FILE.
    STOP RUN.

A100-READ-TRANS-FILE.
    READ TRANS-FILE AT END MOVE "END" TO TRANSACTION-RECORD.
    IF TRANSACTION-RECORD NOT = "END" DISPLAY TRANSACTION-RECORD.
```

2.7 Rewriting Records in a Sequential File

To rewrite a record in a sequential file you must:

1. OPEN the file as INPUT-OUTPUT
2. READ the target record
3. REWRITE the target record

The REWRITE statement places the record just read back into its file. The REWRITE statement completely replaces the contents of the target record with new data. You can use the REWRITE statement for files on mass storage devices only; for example, disk units. There are two ways of rewriting records:

1. REWRITE record-name FROM source-area
2. REWRITE record-name

For example, statements (1) and (2) in this example are logically equivalent:

```
FILE SECTION.
FD  STOCK-FILE.
01  STOCK-RECORD       PIC X(80).

WORKING-STORAGE SECTION.
01  STOCK-WORK         PIC X(80).
```

```
                 (1)                                    (2)

REWRITE STOCK-RECORD FROM STOCK-WORK.    MOVE STOCK-WORK TO STOCK-RECORD.
                                         REWRITE STOCK-RECORD.
```

When you omit the FROM phrase, you process the records directly in the record area or buffer; for example, STOCK-RECORD.

For a REWRITE statement, the record being rewritten must be the same length as the record being replaced.

Example 2-4 reads a sequential file and rewrites as many records as the operator wants.

Example 2-4: Rewriting a Sequential File

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SEQ03.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT   TRANS-FILE    ASSIGN   TO    "TRANS.DAT".
DATA DIVISION.
FILE SECTION.

FD  TRANS-FILE.
01  TRANSACTION-RECORD     PIC X(25).

WORKING-STORAGE SECTION.
01  ANSWER                 PIC X.
```

(continued on next page)

Example 2-4: Rewriting a Sequential File (Cont.)

```
PROCEDURE DIVISION.
A000-BEGIN.
    OPEN I-O TRANS-FILE.
    PERFORM A100-READ-TRANS-FILE UNTIL TRANSACTION-RECORD = "END".
    CLOSE TRANS-FILE.
    STOP RUN.

A100-READ-TRANS-FILE.
    READ TRANS-FILE AT END MOVE "END" TO TRANSACTION-RECORD.
    IF TRANSACTION-RECORD NOT = "END"
       PERFORM A300-GET-ANSWER UNTIL ANSWER = "Y" OR "N"
       PERFORM A200-REWRITE-RECORD.

A200-REWRITE-RECORD.
    IF ANSWER = "Y" DISPLAY "Please enter new record contents"
                    ACCEPT TRANSACTION-RECORD
                    REWRITE TRANSACTION-RECORD.

A300-GET-ANSWER.
    DISPLAY "Do you want to replace this record? -- "
             TRANSACTION-RECORD
    DISPLAY "Please answer Y or N"
    ACCEPT ANSWER.
```

2.8 Extending Sequential Files

To position a file to its current end, and to allow the program to write new records beyond the last record in the file, use both the:

- EXTEND phrase of the OPEN statement
- WRITE statement

For new files, RMS-11 positions the current record pointer at the beginning of the file. Example 2-5 shows how to extend a sequential file.

Example 2-5: Extending a Sequential File

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SEQ04.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT   TRANS-FILE    ASSIGN   TO    "TRANS.DAT".
DATA DIVISION.
FILE SECTION.

FD  TRANS-FILE.
01  TRANSACTION-RECORD    PIC X(25).

PROCEDURE DIVISION.
A000-BEGIN.
    OPEN EXTEND TRANS-FILE.
    PERFORM A100-WRITE-RECORD UNTIL TRANSACTION-RECORD = "END".
    CLOSE TRANS-FILE.
    STOP RUN.
A100-WRITE-RECORD.
    DISPLAY "Enter next record  - X(25)"
    DISPLAY "Enter END to terminate the session"
    DISPLAY "-------------------------"
    ACCEPT TRANSACTION-RECORD
    IF TRANSACTION-RECORD NOT = "END" WRITE TRANSACTION-RECORD.
```

Without the EXTEND phrase, a COBOL-81 program would have to:

1. Open the input file
2. Copy it to an output file
3. Add records to the output file

2.9 Backing Up Your Sequential Files

If your sequential disk file becomes corrupt with bad data, or if your program abnormally terminates when the file is opened for OUTPUT, the file can become unusable. Proper backup procedures are the key to successful recovery.

You should back up your disk file at some reasonable point (daily, weekly, or monthly), depending on file activity, and save all transactions until you create a new backup. In this way, you can easily recreate your sequential disk file from your last backup sequential file and transaction file whenever the need arises.

Chapter 3
Processing Relative Files

A relative file consists of fixed-size record cells and uses a key to retrieve its records. The key, or record key, is an integer that specifies the record's storage cell within the file. It is analogous to the subscript of a table.

Unlike sequential files, where retrieving the twentieth record involves reading the previous nineteen records first, relative files can directly access the twentieth record with one read. In addition, relative files allow the program to read forward or backward depending upon the record key.

Another significant fact of relative file processing is that each cell does not have to contain a record. Although each cell occupies one record space, a field preceding the record on the storage medium indicates whether or not that cell contains a valid record. Thus, a file can contain fewer records than it has cells, and the empty cells can be anywhere in the file.

The numerical order of the cells remains the same during all operations on a relative file; however, accessing statements can move a record from one cell to another, delete a record from a cell, or insert new records into empty cells.

Relative file processing is available only on magnetic disks.

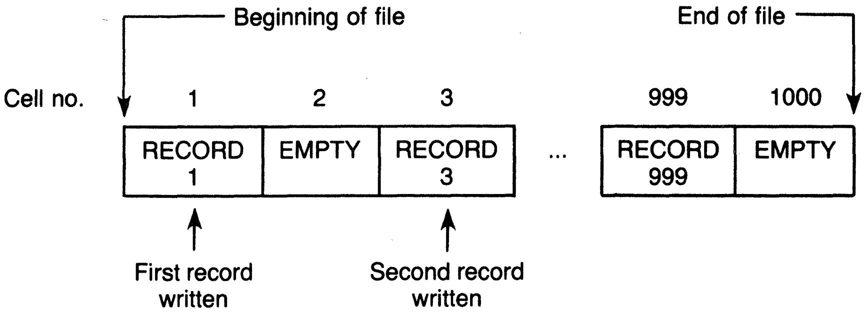
3.1 Relative File Organization

With relative file processing, RMS-11 structures a file as a series of fixed-sized record cells. Cell size is based on the size specified as the maximum permitted length for a record in the file. RMS-11 considers these cells as successively numbered from 1 (the first) to n (the last). A cell's relative record number, or RRN, represents its location relative to the beginning of the file.

Each cell in a relative file can contain a single record. There is no requirement, however, that every cell contain a record. Empty cells can be interspersed among cells containing records.

Since cell numbers in a relative file are unique, they can be used to identify both the cell and the record (if any) occupying that cell. Thus, record number 1 occupies the first cell in the file, record number 21 occupies the twenty-first cell, and so forth. When a cell number is used to identify a record, it is also known as a relative record number. Figure 3-1 depicts the structure of a relatively organized file.

Figure 3-1: Relative File Organization



C81ART-20220-20

Relative files have three capabilities not available with sequential files:

1. Random access by record key
2. Record deletion by record key
3. Record updating by record key

Relative files are used primarily when records must be accessed in random order and the records can easily be associated with a sequential number. When a program creates a relative file, RMS-11 allocates disk space for each cell. Additional space in the cell cannot be added thereafter unless you recreate the file. However, records can be replaced, so empty (dummy) records can be inserted to be replaced later with real records, giving the effect of adding records. After a program creates a relative file, it can be updated by replacing or deleting records. Records are replaced by rewriting the new record over or on top of the old one.

Relative files are used like tables. Their advantage over tables is that their size is limited to disk space rather than memory space. However, it takes much more time to retrieve an element from a relative file than from a table. Relative files are best for records that are easily associated with ascending, consecutive numbers, such as, but not limited to, years (the years 71 to 90 could be stored with record keys 1 to 20), months (record keys 1 to 12), or the 50 U.S. states (record keys 1 to 50).

3.2 Design Considerations

Before you create your relative file applications, you should design your file based on these design considerations:

1. Record format selection (See Chapter 1)

   - Fixed-length
   - Variable-length

   Relative files can contain only unblocked, fixed-length records. You can use variable-length records; however, RMS-11 calculates a cell size equal to the maximum record size plus overhead bytes, resulting in fixed-length records. Once created, relative records can be accessed sequentially, randomly, or dynamically.

2. Medium selection – Relative files can be accessed on disk only. Make sure the disk pack is large enough to meet your current and future needs.

3. Allocation – at time of file creation and file extension
4. Bucket size
5. Maximum record number
6. Compiler limitations

For more information on relative file design, see Chapter 7, File Optimization Techniques, and the *RMS-11 User's Guide.*

3.3 Statements for Relative File Processing

Processing a relative file involves:

1. Opening the file with the OPEN statement
2. Processing the file with valid I/O statements
3. Closing the file with the CLOSE statement

Table 3-1 lists the valid I/O statements and illustrates the following relationships:

- Organization determines valid access modes.
- Organization and access mode determine valid open modes.
- All three (organization, access, and open mode) enable or disable I/O statements.

Table 3-1: Valid I/O Statements for Relative Files

File Organization	Access Mode	Statement	Open Mode		
			INPUT	OUTPUT	I-O
RELATIVE	SEQUENTIAL	DELETE	No	No	Yes
		READ	Yes	No	Yes
		REWRITE	No	No	Yes
		START	Yes	No	Yes
		WRITE	No	Yes	No
	RANDOM	DELETE	No	No	Yes
		READ	Yes	No	Yes
		REWRITE	No	No	Yes
		WRITE	No	Yes	Yes
	DYNAMIC	DELETE	No	No	Yes
		READ	Yes	No	Yes
		REWRITE	No	No	Yes
		START	Yes	No	Yes
		WRITE	No	Yes	Yes

3.4 Defining a Relative File

Each relative file in a COBOL-81 program is given a name, or file name, in a SELECT clause in the Environment Division.

The ASSIGN clause associates the file name with a file specification. The file specification points the operating system to the file's physical and logical location on a specific hardware device (see HEINZ.DAT in Example 3-1). Each file is then further described in the program with a file description (FD) entry in the File Section of the Data Division (see FLAVORS in Example 3-1). The FD entry is then followed immediately by the file's record description (see KETCHUP-MASTER in Example 3-1).

You must specify the ORGANIZATION IS RELATIVE phrase in the SELECT clause; otherwise, COBOL-81 assumes sequential organization. You must also specify the RELATIVE KEY IS phrase and assign a relative key data name in random or dynamic access.

Example 3-1: Defining a Relative File

```
IDENTIFICATION DIVISION.
PROGRAM-ID. REL01.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT FLAVORS ASSIGN TO "HEINZ.DAT"
                   ORGANIZATION IS RELATIVE
                   ACCESS MODE IS RANDOM
                   RELATIVE KEY IS KETCHUP-MASTER-KEY.

DATA DIVISION.
FILE SECTION.
FD  FLAVORS.
01  KETCHUP-MASTER            PIC X(50).

WORKING-STORAGE SECTION.
01  KETCHUP-MASTER-KEY        PIC 99.
```

3.5 Creating Relative Files

A program creates a relative file by:

1. Specifying either of the following access modes in the SELECT clause:

   - Sequential access
   - Random access

   Each of these two access mode choices require different processing techniques. The next two sections discuss those techniques.

2. Opening the file as:

   - OUTPUT – The only function of a WRITE statement with output files is to place entirely new records into the file. If a file requires more space, RMS-11 automatically extends the file size, regardless of the access mode.
   - I-O – With input/output files, the WRITE statement places records into cells that already exist and contain no valid record.

3. Initializing the relative key data name for each record to be written

4. Executing the WRITE statement for each new relative record

5. Closing the file

3.5.1 Sequential Access Mode Creation

When a program creates a relative file in sequential access mode, RMS-11 does not use the relative key. RMS-11 writes the first record in the file at relative record number 1, the second record at relative record number 2, and so on, until the program closes the file. If you use the RELATIVE KEY IS clause, the compiler moves the relative record number of the record being written to the relative key data item. Example 3-2 writes 10 records with relative record numbers 1 to 10.

Example 3-2: Creating a Relative File in Sequential Access Mode

```
IDENTIFICATION DIVISION.
PROGRAM-ID. REL02.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT FLAVORS ASSIGN TO "HEINZ.DAT"
                   ORGANIZATION IS RELATIVE
                   ACCESS MODE IS SEQUENTIAL.

DATA DIVISION.
FILE SECTION.
FD  FLAVORS.
01  KETCHUP-MASTER           PIC X(50).

PROCEDURE DIVISION.
A000-BEGIN.
    OPEN OUTPUT FLAVORS.
    PERFORM A010-WRITE 10 TIMES.
    CLOSE FLAVORS.
A010-WRITE.
    WRITE KETCHUP-MASTER INVALID KEY DISPLAY "BAD WRITE"
                                     STOP RUN.
```

3.5.2 Random Access Mode Creation

When a program creates a relative file using random access mode, the program must place a value in the RELATIVE KEY data item before executing the WRITE statement. Example 3-3 shows how to supply the relative key. It writes 10 records in cells numbered: 2, 4, 6, 8, 10, 12, 14, 16, 18, and 20. Record cells 1, 3, 5, 7, 9, 11, 13, 15, 17, and 19 are also created but contain no valid record.

Example 3-3: Creating a Relative File in Random Access Mode

```
IDENTIFICATION DIVISION.
PROGRAM-ID. REL03.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT FLAVORS ASSIGN TO "HEINZ.DAT"
                   ORGANIZATION IS RELATIVE
                   ACCESS MODE IS RANDOM
                   RELATIVE KEY IS KETCHUP-MASTER-KEY.
```

(continued on next page)

Example 3-3: Creating a Relative File in Random Access Mode (Cont.)

```
DATA DIVISION.
FILE SECTION.
FD  FLAVORS.
01  KETCHUP-MASTER          PIC X(50).

WORKING-STORAGE SECTION.
01  KETCHUP-MASTER-KEY      PIC 99.

PROCEDURE DIVISION.
A000-BEGIN.
    OPEN OUTPUT FLAVORS.
    MOVE 0 TO KETCHUP-MASTER-KEY.
    PERFORM A010-CREATE-RELATIVE-FILE 10 TIMES.
    DISPLAY "END OF JOB".
    CLOSE FLAVORS.
    STOP RUN.
A010-CREATE-RELATIVE-FILE.
    ADD 2 TO KETCHUP-MASTER-KEY
    WRITE KETCHUP-MASTER INVALID KEY DISPLAY "BAD WRITE"
                                     STOP RUN.
```

3.6 Reading Relative Files

Your program can read a relative file three ways:

1. Sequentially
2. Randomly
3. Dynamically

3.6.1 Sequential Reading

To sequentially read relative records:

1. Specify the ACCESS MODE IS SEQUENTIAL clause.
2. Open the file as INPUT or I-O.
3. Read records as you would a sequential file, or use the START statement.

The READ statement makes the next logical record of an open file available to the program. The system sequentially reads the file from either: (1) cell 1 or (2) wherever you START the file, up to cell n. It skips the empty cells and retrieves only valid records. Each READ statement updates the contents of the file's RELATIVE KEY data item, if specified. The data item contains the relative number of the available record. When the at end condition occurs, execution of the READ statement is unsuccessful (see Chapter 5, Input/Output Exception Conditions Handling).

Sequential processing need not begin at the first record of a relative file. The START statement specifies the next record to be read sequentially. It positions the current record pointer for subsequent I/O operations.

Example 3-4 reads a relative file sequentially and displays its contents on the terminal.

Example 3-4: Sequentially Reading Relative Files

```
IDENTIFICATION DIVISION.
PROGRAM-ID. REL04.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT FLAVORS ASSIGN TO "HEINZ.DAT"
                   ORGANIZATION IS RELATIVE
                   ACCESS MODE IS SEQUENTIAL
                   RELATIVE KEY IS KETCHUP-MASTER-KEY.

DATA DIVISION.
FILE SECTION.
FD  FLAVORS.
01  KETCHUP-MASTER            PIC X(50).

WORKING-STORAGE SECTION.
01  KETCHUP-MASTER-KEY        PIC 99.

01  END-OF-FILE               PIC X.

PROCEDURE DIVISION.
A000-BEGIN.
    OPEN INPUT FLAVORS.
    PERFORM A010-DISPLAY-RECORDS UNTIL END-OF-FILE = "Y".
A005-EOJ.
    DISPLAY "END OF JOB".
    CLOSE FLAVORS.
    STOP RUN.
A010-DISPLAY-RECORDS.
    READ FLAVORS AT END MOVE "Y" TO END-OF-FILE.
    IF END-OF-FILE NOT = "Y" DISPLAY KETCHUP-MASTER.
```

3.6.2 Random Reading

To randomly read relative records:

1. Specify the ACCESS MODE IS RANDOM or DYNAMIC clause.
2. Open the file as INPUT or I-O.
3. Move the relative record number value to the RELATIVE KEY data name.
4. Read the record from the cell identified by the relative record number.

The READ statement selects a specific record from an open file and makes it available to the program. The value of the relative key identifies the specific record. The system randomly reads the record identified by the RELATIVE KEY data name clause. If the cell does not contain a valid record, the invalid key condition exists, and the READ fails (see Chapter 5, Input/Output Exception Conditions Handling).

Example 3-5 reads a relative file randomly and displays its contents on the terminal.

Example 3-5: Randomly Reading a Relative File

```
IDENTIFICATION DIVISION.
PROGRAM-ID. REL05.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT FLAVORS ASSIGN TO "HEINZ.DAT"
                   ORGANIZATION IS RELATIVE
                   ACCESS MODE IS RANDOM
                   RELATIVE KEY IS KETCHUP-MASTER-KEY.

DATA DIVISION.
FILE SECTION.
FD  FLAVORS.
01  KETCHUP-MASTER           PIC X(50).

WORKING-STORAGE SECTION.
01  KETCHUP-MASTER-KEY       PIC 99.

PROCEDURE DIVISION.
A000-BEGIN.
    OPEN INPUT FLAVORS.
    PERFORM A100-DISPLAY-RECORD UNTIL KETCHUP-MASTER-KEY = 00.
    DISPLAY "END OF JOB".
    CLOSE FLAVORS.
    STOP RUN.
A100-DISPLAY-RECORD.
    DISPLAY "TO DISPLAY A RECORD ENTER ITS RECORD NUMBER".
    ACCEPT KETCHUP-MASTER-KEY.
    READ FLAVORS INVALID KEY DISPLAY "BAD KEY"
                             CLOSE FLAVORS
                             STOP RUN.
    DISPLAY KETCHUP-MASTER.
```

3.6.3 Dynamic Reading

The READ statement has two formats so that it can select the next logical record (sequentially) or select a specific record (randomly) and make it available to the program. In dynamic mode, the program can switch from random access I/O statements to sequential access I/O statements and in any order without closing and reopening files. However, you must use the READ NEXT statement to sequentially read a relative file open in dynamic mode.

Sequential processing need not begin at the first record of a relative file. The START statement specifies the next record to be read sequentially. It positions the current record pointer for subsequent I/O operations.

A sequential read of a dynamic file is indicated by the NEXT phrase of the READ statement. A READ NEXT statement should follow the START statement since the READ NEXT statement reads the next record pointed to by the current record pointer. Subsequent READ NEXT statements sequentially retrieve records until another START statement or random READ statement executes.

Example processes a relative file containing 57 records. Each record has a unique number from 1 to 57 as its key. The program positions the file (START statement) to the cell corresponding to the value in INPUT-RECORD-KEY. The program's READ...NEXT statement retrieves the remaining valid records in the file for display on the terminal.

Example 3-6: Dynamically Reading a Relative File

```
IDENTIFICATION DIVISION.
PROGRAM-ID. RELOG.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT FLAVORS ASSIGN TO "HEINZ.DAT"
                   ORGANIZATION IS RELATIVE
                   ACCESS MODE IS DYNAMIC
                   RELATIVE KEY IS KETCHUP-MASTER-KEY.

DATA DIVISION.
FILE SECTION.
FD  FLAVORS.
01  KETCHUP-MASTER          PIC X(50).

WORKING-STORAGE SECTION.
01  KETCHUP-MASTER-KEY      PIC 99.
01  END-OF-FILE             PIC X   VALUE "N".

PROCEDURE DIVISION.
A000-BEGIN.
    OPEN I-O FLAVORS.
    DISPLAY "Enter number".
    ACCEPT KETCHUP-MASTER-KEY.
    START FLAVORS KEY = KETCHUP-MASTER-KEY
          INVALID KEY DISPLAY "Bad START statement"
          GO TO A005-END-OF-JOB.
    PERFORM A010-DISPLAY-RECORDS UNTIL END-OF-FILE = "Y".
A005-END-OF-JOB.
    DISPLAY "END OF JOB".
    CLOSE FLAVORS.
    STOP RUN.
A010-DISPLAY-RECORDS.
    READ FLAVORS NEXT RECORD AT END MOVE "Y" TO END-OF-FILE.
    IF END-OF-FILE NOT = "Y" DISPLAY KETCHUP-MASTER.
```

3.7 Updating Relative Files

A program updates a relative file with the DELETE, REWRITE, and WRITE statements. The WRITE statement adds a record to the file. Only the DELETE and REWRITE statements change the contents of records already existing in the file. In either case, adequate backup must be available in the event of error. The next two sections each discuss and present an example of how to rewrite and delete relative records.

3.7.1 Rewriting Relative Records

Two options available for rewriting relative records are:

1. Sequential access rewriting
2. Random access rewriting

The REWRITE statement logically replaces a record in a relative file. After successfully rewriting a record into the file, the program can access that record at any time. However, the program cannot access the record that occupied the cell previous to the rewrite operation.

3.7.1.1 Sequential Access Mode Rewriting — To rewrite relative records in sequential access mode:

1. Specify the ACCESS MODE IS SEQUENTIAL clause.
2. Open the file as I-O.
3. Use the START statement to position the record pointer and then READ the target record, or sequentially READ the file up to the target record.
4. Update the target record.
5. REWRITE the target record back into its cell.

The REWRITE statement places the successfully read record back into its cell in the file.

Example 3-7 sequentially reads a relative record, displays its contents on the terminal before it updates the record, updates the record, displays its contents on the terminal after it updates the record, and rewrites the record in the same cell.

Example 3-7: Rewriting Relative Records in Sequential Access Mode

```
IDENTIFICATION DIVISION.
PROGRAM-ID. REL07A.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT FLAVORS ASSIGN TO "HEINZ.DAT"
                   ORGANIZATION IS RELATIVE
                   ACCESS MODE IS SEQUENTIAL
                   RELATIVE KEY IS KETCHUP-MASTER-KEY.

DATA DIVISION.
FILE SECTION.
FD  FLAVORS.
01  KETCHUP-MASTER          PIC X(50).

WORKING-STORAGE SECTION.
01  KETCHUP-MASTER-KEY      PIC 99.

PROCEDURE DIVISION.
A000-BEGIN.
    OPEN I-O FLAVORS.
    PERFORM A100-UPDATE-RECORD UNTIL KETCHUP-MASTER-KEY = 00.
A005-EOJ.
    DISPLAY "END OF JOB".
    CLOSE FLAVORS.
    STOP RUN.
A100-UPDATE-RECORD.
    DISPLAY "TO UPDATE A RECORD ENTER ITS RECORD NUMBER"
    ACCEPT KETCHUP-MASTER-KEY
    START FLAVORS KEY IS EQUAL TO KETCHUP-MASTER-KEY
          INVALID KEY DISPLAY "BAD START"
                      STOP RUN.
    IF KETCHUP-MASTER-KEY IS NOT EQUAL TO 00
        PERFORM A200-READ-FLAVORS
        DISPLAY  "*********BEFORE UPDATE*********"
        DISPLAY KETCHUP-MASTER
****************************************************************************
*
*               Update routine
*
****************************************************************************
        DISPLAY  "*********AFTER UPDATE*********"
        DISPLAY KETCHUP-MASTER
        REWRITE KETCHUP-MASTER.
```

(continued on next page)

Example 3-7: Rewriting Relative Records in Sequential Access Mode (Cont.)

```
A200-READ-FLAVORS.
    READ FLAVORS AT END DISPLAY "END OF FILE"
                        GO TO A005-EOJ.
```

3.7.1.2 Random Access Mode Rewriting — To rewrite relative records in random access mode:

1. Specify the ACCESS MODE IS RANDOM or DYNAMIC clause.
2. Open the file as I-O.
3. Move the relative record number value of the record you want to read to the RELATIVE KEY data name.
4. Optionally read the record from the cell identified by the relative record number.
5. Update the record.
6. REWRITE the record into the cell identified by the relative record number.

The system randomly reads the record identified by the KEY IS clause. The REWRITE statement places the successfully read record back into its cell in the file.

If the cell does not contain a valid record, or if the rewrite operation is unsuccessful, the invalid key condition exists (see Chapter 5, Input/Output Exception Conditions Handling).

The next example randomly reads a relative record, displays its before contents on the terminal, updates the record, displays its after contents on the terminal, and rewrites the record in the same cell.

Example 3-8: Rewriting Relative Records in Random Access Mode

```
IDENTIFICATION DIVISION.
PROGRAM-ID. REL07B.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT FLAVORS ASSIGN TO "HEINZ.DAT"
                   ORGANIZATION IS RELATIVE
                   ACCESS MODE IS RANDOM
                   RELATIVE KEY IS KETCHUP-MASTER-KEY.

DATA DIVISION.
FILE SECTION.
FD  FLAVORS.
01  KETCHUP-MASTER            PIC X(50).

WORKING-STORAGE SECTION.
01  KETCHUP-MASTER-KEY        PIC 99.

PROCEDURE DIVISION.
A000-BEGIN.
    OPEN I-O FLAVORS.
    PERFORM A100-UPDATE-RECORD UNTIL KETCHUP-MASTER-KEY = 00.
A005-EOJ.
    DISPLAY "END OF JOB".
    CLOSE FLAVORS.
    STOP RUN.
```

(continued on next page)

Example 3-8: Rewriting Relative Records in Random Access Mode (Cont.)

```
A100-UPDATE-RECORD.
    DISPLAY "TO UPDATE A RECORD ENTER ITS RECORD NUMBER".
    ACCEPT KETCHUP-MASTER-KEY.
    READ FLAVORS INVALID KEY DISPLAY "BAD READ"
                        GO TO A005-EOJ.
    DISPLAY  "*********BEFORE UPDATE*********".
    DISPLAY KETCHUP-MASTER.
****************************************************************************
*
*               Update routine
*
****************************************************************************
    DISPLAY  "*********AFTER UPDATE*********".
    DISPLAY KETCHUP-MASTER.
    REWRITE KETCHUP-MASTER INVALID KEY DISPLAY "BAD REWRITE"
                                       GO TO A005-EOJ.
```

3.7.2 Deleting Relative Records

Two options are available for deleting relative records:

1. Sequential access mode deletion
2. Random access mode deletion

The DELETE statement logically removes an existing record from a relative file. After successfully removing a record from a file, the program cannot later access it.

3.7.2.1 Sequential Access Mode Deletion — To delete a relative record in sequential access mode you must:

1. Specify the ACCESS MODE IS SEQUENTIAL clause.
2. Open the file as INPUT-OUTPUT.
3. Either use the START statement to position the record pointer and then read the target record, or sequentially read the file up to the target record.
4. Delete the last read record.

Example 3-9: Deleting Relative Records in Sequential Access Mode

```
IDENTIFICATION DIVISION.
PROGRAM-ID. REL08.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT FLAVORS ASSIGN TO "HEINZ.DAT"
                   ORGANIZATION IS RELATIVE
                   ACCESS MODE IS SEQUENTIAL
                   RELATIVE KEY IS KETCHUP-MASTER-KEY.

DATA DIVISION.
FILE SECTION.
FD  FLAVORS.
01  KETCHUP-MASTER           PIC X(50).
```

(continued on next page)

Example 3-9: Deleting Relative Records in Sequential Access Mode (Cont.)

```
WORKING-STORAGE SECTION.
01  KETCHUP-MASTER-KEY       PIC 99.

PROCEDURE DIVISION.
A000-BEGIN.
    OPEN I-O FLAVORS.
    PERFORM A010-DELETE-RECORDS UNTIL KETCHUP-MASTER-KEY = 00.
A005-EOJ.
    DISPLAY "END OF JOB".
    CLOSE FLAVORS.
    STOP RUN.
A010-DELETE-RECORDS.
    DISPLAY "TO DELETE A RECORD ENTER ITS RECORD NUMBER".
    ACCEPT KETCHUP-MASTER-KEY.
    IF KETCHUP-MASTER-KEY NOT = 00 PERFORM A200-READ-FLAVORS
                                   DELETE FLAVORS RECORD.
A200-READ-FLAVORS.
    READ FLAVORS AT END DISPLAY "FILE AT END"
                        GO TO A005-EOJ.
```

3.7.2.2 Random Access Mode Deletion — To delete a relative record in random access mode you must:

- Specify the ACCESS MODE IS RANDOM clause.
- Open the file INPUT-OUTPUT.
- Move the relative record number value to the RELATIVE KEY data name.
- Delete the record identified by relative record number.

If the file does not contain a valid record, an invalid key condition exists.

Example 3-10: Deleting Relative Records in Random Access Mode

```
IDENTIFICATION DIVISION.
PROGRAM-ID. REL09.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT FLAVORS ASSIGN TO "HEINZ.DAT"
                   ORGANIZATION IS RELATIVE
                   ACCESS MODE IS RANDOM
                   RELATIVE KEY IS KETCHUP-MASTER-KEY.

DATA DIVISION.
FILE SECTION.
FD  FLAVORS.
01  KETCHUP-MASTER           PIC X(50).

WORKING-STORAGE SECTION.
01  KETCHUP-MASTER-KEY       PIC 99.

PROCEDURE DIVISION.
A000-BEGIN.
    OPEN I-O FLAVORS.
    PERFORM A010-DELETE-RECORDS UNTIL KETCHUP-MASTER-KEY = 00.
A005-EOJ.
    DISPLAY "END OF JOB".
    CLOSE FLAVORS.
    STOP RUN.
```

(continued on next page)

Example 3-10: Deleting Relative Records in Random Access Mode (Cont.)

```
A010-DELETE-RECORDS.
    DISPLAY "TO DELETE A RECORD ENTER ITS RECORD NUMBER".
    ACCEPT KETCHUP-MASTER-KEY.
    IF KETCHUP-MASTER-KEY NOT = 00
       DELETE FLAVORS RECORD
              INVALID KEY DISPLAY "INVALID DELETE"
                          STOP RUN.
```

3.8 Backing Up Your Relative Files

If your relative file becomes corrupt with bad data, or if your program abnormally terminates when the file is opened for OUTPUT or INPUT-OUTPUT, the file can become unusable. Proper backup procedures are the key to successful recovery.

You should backup your disk file at some reasonable point (daily, weekly, or monthly), depending on file activity, and save all transactions until you create a new backup. In this way, you can easily recreate your relative file from your last backup relative file and transaction file whenever the need arises.

Chapter 4
Processing Indexed Files

Unlike the sequential ordering of records in a sequential file or the relative positioning of records in a relative file, the location of records in indexed file organization is transparent to the program. It is possible to add new records to an indexed file and logically place them between physically adjacent records, without recreating the file. Not only can records be added, but they can also be deleted, making room for new records.

RMS-11 controls the placement of records in an indexed file based on user-specified primary and alternate keys in the record itself. The presence of keys in the records of the file governs this placement. This is the only file organization where RMS-11 uses the actual contents of the records for record placement within the file.

Indexed file processing is available only on disk.

4.1 Indexed File Organization

COBOL-81 allows sequential, random, and dynamic access to records. Each record is accessed by one of its primary or alternate keys.

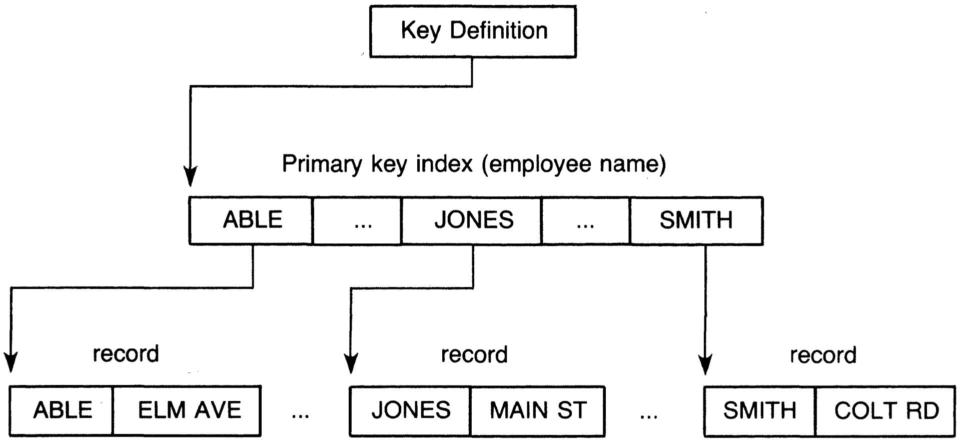
A major feature of indexed file organization is the use of a key to uniquely identify a record within the file. A key is a character string present in every record of an indexed file. Its location and length are identical in all records. When creating an indexed file, you must select the character string(s) to be the key(s). Selecting such a character string indicates to RMS-11 that the contents (key value) of that string in any record written to the file can be used by the program to identify that record for subsequent retrieval. For more information, see the RECORD KEY IS clause and the ALTERNATE KEY IS clause in the *COBOL-81 Language Reference Manual.*

You must define at least one main key, called the "primary key," for an indexed file. Primary key values must be unique and defined in the record description entry. For example, if an employee file uses Social Security numbers as a primary key, there can be no duplicate numbers in your file.

You can optionally define from 1 to 254 additional keys called "alternate keys." Alternate key values need not be unique if you specify the WITH DUPLICATES phrase in the file description entry. You must define each alternate key in the record description entry. Each alternate key represents an additional character string in each record of the file. The key value in any of these additional strings can also be used as a means of identifying the record for retrieval.

As your program writes records into an indexed file, RMS-11 locates the values contained in the primary and alternate keys. From the values in keys within the record, RMS-11 builds a tree-structured table known as an "index." An index consists of a series of entries. Each entry contains a key value copied from a record written by a program. With each key value is a pointer to the location in the file of the record from which the value was copied. Figure 4-1 shows the general structure of an indexed file defined with a primary key only.

Figure 4-1: Indexed File Organization



C81ART-20230-30

For a more detailed explanation of indexed file structure, see the *RMS-11 User's Guide*.

4.2 Design Considerations

Before you create your indexed file applications, you should design your file based on these design considerations:

1. Record format selection (See Chapter 1).
   - Fixed-length
   - Variable-length
2. Medium selection – Indexed files can be accessed on disk only.
3. Allocation – At time of file creation and file extension (See Chapter 7).
4. Speed – You want to maximize the speed with which the program processes data.
5. Space – You want to minimize file size, disk space, and memory requirements to run your program.

6. Shared access – You want your data to be exactly as accessible to the people using the computer system as necessary, no more, no less.

7. Ease of design – You do not want to spend more time than necessary writing the application.

8. Compiler limitations – The logical and physical limits imposed by the COBOL-81 compiler.

For more information on indexed file design optimization, see Chapter 7, File Optimization Techniques, and the *RMS-11 User's Guide*. If you do not carefully design your index file – that is, you take all the file defaults – your indexed file application could run more slowly than you expect.

4.3 Statements for Indexed File Processing

Processing an indexed file involves:

1. Opening the file with the OPEN statement
2. Processing the file with valid I/O statements
3. Closing the file with the CLOSE statement

Table 4-1 lists the valid I/O statements and illustrates the following relationships:

- Organization determines valid access modes.
- Organization and access mode determine valid open modes.
- All three (organization, access, and open mode) enable or disable I/O statements.

Table 4-1: Valid I/O Statements for Indexed Files

			Open Mode		
File Organization	**Access Mode**	**Statement**	**INPUT**	**OUTPUT**	**I-O**
INDEXED	SEQUENTIAL	DELETE	No	No	Yes
		READ	Yes	No	Yes
		REWRITE	No	No	Yes
		START	Yes	No	Yes
		WRITE	No	Yes	No
	RANDOM	DELETE	No	No	Yes
		READ	Yes	No	Yes
		REWRITE	No	No	Yes
		WRITE	No	Yes	Yes
	DYNAMIC	DELETE	No	No	Yes
		READ	Yes	No	Yes
		REWRITE	No	No	Yes
		START	Yes	No	Yes
		WRITE	No	Yes	Yes

4.4 Defining an Indexed File

Each indexed file in a COBOL-81 program is given a name, or file name, in a SELECT clause in the Environment Division. The ASSIGN clause associates the file name to a file specification. The file specification points the operating system to the file's physical and logical location on a specific hardware device (see Example 4-1, DAIRY.DAT). Each file is then further described in the program with a file description (FD) entry in the File Section of the Data Division (see Example 4-1, FLAVORS). The FD entry is then followed immediately by the file's record description (see Example 4-1, ICE-CREAM-MASTER). Refer to the *COBOL-81 Language Reference Manual* for information relating to the RECORD KEY and ALTERNATE RECORD KEY clauses.

Example 4-1 defines a dynamic access mode indexed file with one primary key (ICE-CREAM-MASTER-KEY) and two alternate record keys (ICE-CREAM-MASTER-CODE and ICE-CREAM-MASTER-STATE). Note that one alternate record key allows duplicates (ICE-CREAM-MASTER-STATE). Any program using the identical entries in the SELECT clause as shown in Example 4-1 can reference the DAIRY.DAT file sequentially and randomly.

Example 4-1: Defining an Indexed File

```
IDENTIFICATION DIVISION.
PROGRAM-ID. INDEX01.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT FLAVORS ASSIGN TO "DAIRY.DAT"
                   ORGANIZATION IS INDEXED
                   ACCESS MODE IS DYNAMIC
                   RECORD KEY IS ICE-CREAM-MASTER-KEY
                   ALTERNATE RECORD KEY IS ICE-CREAM-STORE-STATE
                                           WITH DUPLICATES
                   ALTERNATE RECORD KEY IS ICE-CREAM-STORE-CODE.

DATA DIVISION.
FILE SECTION.
FD  FLAVORS.
01  ICE-CREAM-MASTER.
    02 ICE-CREAM-MASTER-KEY          PIC XXXX.
    02 ICE-CREAM-MASTER-DATA.
       03  ICE-CREAM-STORE-CODE      PIC XXXXX.
       03  ICE-CREAM-STORE-ADDRESS   PIC X(20).
       03  ICE-CREAM-STORE-CITY      PIC X(20).
       03  ICE-CREAM-STORE-STATE     PIC XX.
PROCEDURE DIVISION.
A00-BEGIN.
```

You must specify the ORGANIZATION IS INDEXED phrase and the ACCESS MODE IS DYNAMIC phrase in the SELECT clause; otherwise, COBOL-81 assumes sequential organization and access mode.

4.5 Creating and Populating Indexed Files

A COBOL-81 program creates an indexed file by:

1. Opening the file for:
   - OUTPUT – to add records only
   - I-O – to add, change, or delete records

2. Initializing the key value/s

3. Executing the WRITE statement

4. Specifying either of the following access modes in the SELECT clause:

   - Sequential access – the program must write records in ascending order by primary key

   - Random or dynamic access – the program can write records in any order

The best way to initially populate an indexed file is to sequentially write the records in ascending order by primary key.

The program can add records to the file until it reaches the physical limitations of its storage device. When this occurs, you should: (1) delete unnecessary records, (2) back up the file, and (3) recreate the file by using either the RMSFIL utility to optimize file space or by using a COBOL-81 program. For more information on the RMSFIL utility, see the *RMS-11 Utilities Manual.*

Example 4-2 creates and populates an indexed file (DAIRY.DAT). The source file (DAIRYI.DAT) has been sorted in ascending sequence. Notice that the primary and alternate keys are initialized in ICE-CREAM-MASTER when the contents of the fields in INPUT-RECORD are read into ICE-CREAM-MASTER before the record is written.

Example 4-2: Creating and Populating an Indexed File

```
IDENTIFICATION DIVISION.
PROGRAM-ID. INDEX02.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT INPUT-FILE ASSIGN TO "DAIRYI.DAT".

    SELECT FLAVORS      ASSIGN TO "DAIRY.DAT"
                        ORGANIZATION IS INDEXED
                        ACCESS MODE IS SEQUENTIAL
                        RECORD KEY IS ICE-CREAM-MASTER-KEY
                        ALTERNATE RECORD KEY IS ICE-CREAM-STORE-STATE
                                                WITH DUPLICATES
                        ALTERNATE RECORD KEY IS ICE-CREAM-STORE-CODE.

DATA DIVISION.
FILE SECTION.
FD  INPUT-FILE.
01  INPUT-RECORD.
    02  INPUT-RECORD-KEY              PIC 9999.
    02  INPUT-RECORD-DATA             PIC X(47).

FD  FLAVORS.
01  ICE-CREAM-MASTER.
    02 ICE-CREAM-MASTER-KEY           PIC XXXX.
    02 ICE-CREAM-MASTER-DATA.
       03  ICE-CREAM-STORE-CODE       PIC XXXXX.
       03  ICE-CREAM-STORE-ADDRESS    PIC X(20).
       03  ICE-CREAM-STORE-CITY       PIC X(20).
       03  ICE-CREAM-STORE-STATE      PIC XX.
```

(continued on next page)

Example 4-2: Creating and Populating an Indexed File (Cont.)

```
WORKING-STORAGE SECTION.
01  END-OF-FILE                      PIC X.

PROCEDURE DIVISION.
A000-BEGIN.
    OPEN INPUT INPUT-FILE.
    OPEN OUTPUT FLAVORS.

A010-POPULATE.
    PERFORM A100-READ-INPUT UNTIL END-OF-FILE = "Y".

A020-EOJ.
    DISPLAY "END OF JOB".
    STOP RUN.

A100-READ-INPUT.
    READ INPUT-FILE INTO ICE-CREAM-MASTER
         AT END MOVE "Y" TO END-OF-FILE.
    IF END-OF-FILE NOT = "Y"
       WRITE ICE-CREAM-MASTER INVALID KEY DISPLAY "BAD WRITE"
                                          STOP RUN.
```

4.6 Reading Indexed Files

Your program can read an indexed file three ways:

1. Sequentially
2. Randomly
3. Dynamically

However, to randomly read the file, the program must: (1) initialize either the primary key data name or the alternate key data name before reading the target record, and (2) specify that data name in the KEY IS phrase of the READ statement.

Dynamic access permits switching back and forth from sequential access to random access any number of times during one OPEN of the file.

4.6.1 Sequential Reading

To read indexed records in a sequential mode:

1. Specify the ACCESS MODE IS SEQUENTIAL clause.
2. Open the file for INPUT or INPUT-OUTPUT.
3. Either:
   - Read records from the beginning of the file. Read records as you would a sequential file, that is, use READ...AT END... statement.
   - Read records after positioning the current record pointer somewhere in the file. Use the START statement to start the file at a specific record, then use the READ...NEXT RECORD AT END... statement to sequentially read subsequent records.

The READ statement makes the next logical record of an open file available to the program. It skips deleted records and sequentially reads and retrieves only valid records. When the at end condition occurs, execution of the READ statement is unsuccessful (see Chapter 5, Input/Output Exception Conditions Handling).

Example 4-3 sequentially reads the entire indexed file from the first record in the file and displays every record on the terminal:

Example 4-3: Sequentially Reading an Indexed File

```
IDENTIFICATION DIVISION.
PROGRAM-ID. INDEX03.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.

    SELECT FLAVORS     ASSIGN TO "DAIRY.DAT"
                       ORGANIZATION IS INDEXED
                       ACCESS MODE IS SEQUENTIAL
                       RECORD KEY IS ICE-CREAM-MASTER-KEY
                       ALTERNATE RECORD KEY IS ICE-CREAM-STORE-STATE
                                               WITH DUPLICATES
                       ALTERNATE RECORD KEY IS ICE-CREAM-STORE-CODE.

DATA DIVISION.
FILE SECTION.

FD  FLAVORS.
01  ICE-CREAM-MASTER.
    02 ICE-CREAM-MASTER-KEY           PIC XXXX.
    02 ICE-CREAM-MASTER-DATA.
       03  ICE-CREAM-STORE-CODE       PIC XXXXX.
       03  ICE-CREAM-STORE-ADDRESS    PIC X(20).
       03  ICE-CREAM-STORE-CITY       PIC X(20).
       03  ICE-CREAM-STORE-STATE      PIC XX.

WORKING-STORAGE SECTION.
01  END-OF-FILE                       PIC X.

PROCEDURE DIVISION.
A000-BEGIN.
    OPEN INPUT FLAVORS.

A010-SEQUENTIAL-READ.
    PERFORM A100-READ-INPUT UNTIL END-OF-FILE = "Y".

A020-EOJ.
    DISPLAY "END OF JOB".
    STOP RUN.

A100-READ-INPUT.
    READ  FLAVORS AT END MOVE "Y" TO END-OF-FILE.
    IF END-OF-FILE NOT = "Y"
       DISPLAY ICE-CREAM-MASTER
       STOP "Type CONTINUE to display next master".
```

4.6.2 Random Reading

To randomly read indexed records:

- Specify the ACCESS MODE IS RANDOM clause.
- Open the file for INPUT or INPUT-OUTPUT.
- Initialize the RECORD KEY or ALTERNATE RECORD KEY data name before reading the record.
- Read the record.

The READ statement selects a specific record from an open file and makes it available to the program. The value of the primary or alternate key identifies the specific record. The system randomly reads the record identified by the RECORD KEY or ALTERNATE RECORD KEY clause. If RMS-11 does not find a valid record, the invalid key condition exists, and the READ fails (see Chapter 5, Input/Output Exception Conditions Handling).

Example 4-4 randomly reads an indexed file and displays its contents on the terminal. It makes use of both the primary key (ICE-CREAM-MASTER-KEY) and the alternate key (ICE-CREAM-STORE-STATE).

Example 4-4: Randomly Reading an Indexed File

```
IDENTIFICATION DIVISION.
PROGRAM-ID. INDEX04.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT FLAVORS    ASSIGN TO "DAIRY.DAT"
                      ORGANIZATION IS INDEXED
                      ACCESS MODE IS RANDOM
                      RECORD KEY IS ICE-CREAM-MASTER-KEY
                      ALTERNATE RECORD KEY IS ICE-CREAM-STORE-STATE
                                           WITH DUPLICATES
                      ALTERNATE RECORD KEY IS ICE-CREAM-STORE-CODE.

DATA DIVISION.
FILE SECTION.
FD  FLAVORS.
01  ICE-CREAM-MASTER.
    02 ICE-CREAM-MASTER-KEY           PIC XXXX.
    02 ICE-CREAM-MASTER-DATA.
       03  ICE-CREAM-STORE-CODE       PIC XXXXX.
       03  ICE-CREAM-STORE-ADDRESS    PIC X(20).
       03  ICE-CREAM-STORE-CITY       PIC X(20).
       03  ICE-CREAM-STORE-STATE      PIC XX.

WORKING-STORAGE SECTION.
01  PROGRAM-STAT                PIC X.
    88  OPERATOR-STOPS-IT                     VALUE "1".
    88  LETS-SEE-NEXT-STORE                   VALUE "2".
    88  NO-MORE-DUPLICATES                    VALUE "3".
    88  STOP-THE-JOB                          VALUE "4".
01  ANSWER                      PIC X.
PROCEDURE DIVISION.
A000-BEGIN.
    OPEN I-O FLAVORS.
    PERFORM A030-RANDOM-READ UNTIL OPERATOR-STOPS-IT.
    DISPLAY "END OF JOB".
    STOP RUN.
```

(continued on next page)

Example 4-4: Randomly Reading an Indexed File (Cont.)

```
A030-RANDOM-READ.
    DISPLAY "Enter Key".
    ACCEPT ICE-CREAM-MASTER-KEY.
    PERFORM A100-READ-INPUT-BY-PRIMARY-KEY THROUGH
            A100-READ-INPUT-EXIT.
    DISPLAY " Do you wish to terminate the session?".
    PERFORM A040-GET-ANSWER UNTIL PROGRAM-STAT = "Y" OR "N".
    IF PROGRAM-STAT = "Y" MOVE "1" TO PROGRAM-STAT.

A040-GET-ANSWER.
    DISPLAY "Please answer Y or N"
    ACCEPT PROGRAM-STAT.

A100-READ-INPUT-BY-PRIMARY-KEY.
    READ FLAVORS KEY IS ICE-CREAM-MASTER-KEY
         INVALID KEY DISPLAY "Master does not exist - Try again"
         GO TO A100-READ-INPUT-EXIT.
    DISPLAY ICE-CREAM-MASTER.
    PERFORM A200-READ-BY-ALTERNATE-KEY UNTIL NO-MORE-DUPLICATES.

A100-READ-INPUT-EXIT.
    EXIT.

A200-READ-BY-ALTERNATE-KEY.
    DISPLAY "Do you want to see the next store in this state?".
    MOVE SPACE TO PROGRAM-STAT.
    PERFORM A040-GET-ANSWER UNTIL PROGRAM-STAT = "Y" OR "N".
    IF PROGRAM-STAT = "Y"
       MOVE "2" TO PROGRAM-STAT.
    IF LETS-SEE-NEXT-STORE
       READ FLAVORS KEY IS ICE-CREAM-STORE-STATE
                    INVALID KEY DISPLAY "No more stores in this state"
                                MOVE "3" TO PROGRAM-STAT.
    IF PROGRAM-STAT = "N"
       MOVE "3" TO PROGRAM-STAT.
```

4.6.3 Dynamic Reading

The READ statement has two formats, so it can select the next logical record (sequentially) or select a specific record (randomly) and make it available to the program. In dynamic mode, the program can switch from using random access I/O statements to sequential access I/O statements, in any order, without closing and reopening files. However, the program must use the READ NEXT statement to sequentially read an indexed file open in dynamic mode.

Sequential processing need not begin at the first record of an indexed file. The START statement specifies the next record to be read sequentially. It positions the current record pointer for subsequent I/O operations anywhere within the file.

A sequential read of a dynamic file is indicated by the NEXT phrase of the READ statement. A READ NEXT statement should follow the START statement since the READ NEXT statement reads the next record pointed to by the current record pointer. Subsequent READ NEXT statements sequentially retrieve records until another START statement or random READ statement executes.

Example 4-5 processes an indexed file containing 26 records. Each record has a unique letter of the alphabet as its primary key. The program positions the file to the first record whose INPUT-RECORD-KEY is equal to the specified letter of the alphabet. The program's READ NEXT statement sequentially retrieves the remaining valid records in the file for display on the terminal.

Example 4-5: Dynamically Reading an Indexed File

```
IDENTIFICATION DIVISION.
PROGRAM-ID. INDEX05.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.

    SELECT IND-ALPHA  ASSIGN TO "ALPHA.DAT"
                      ORGANIZATION IS INDEXED
                      ACCESS MODE IS DYNAMIC
                      RECORD KEY IS INPUT-RECORD-KEY.

DATA DIVISION.
FILE SECTION.
FD  IND-ALPHA.
01  INPUT-RECORD.
    02  INPUT-RECORD-KEY             PIC X.
    02  INPUT-RECORD-DATA            PIC X(50).

WORKING-STORAGE SECTION.
01  END-OF-FILE                      PIC X.
PROCEDURE DIVISION.
A000-BEGIN.
    OPEN I-O IND-ALPHA.
    DISPLAY "Enter letter"
    ACCEPT INPUT-RECORD-KEY.
    START IND-ALPHA KEY = INPUT-RECORD-KEY
          INVALID KEY DISPLAY "BAD START STATEMENT"
          GO TO A010-END-OF-JOB.
    PERFORM A100-GET-RECORDS THROUGH A100-GET-RECORDS-EXIT
            UNTIL END-OF-FILE = "Y".

A010-END-OF-JOB.
    DISPLAY "END OF JOB".
    CLOSE IND-ALPHA.
    STOP RUN.

A100-GET-RECORDS.
    READ IND-ALPHA NEXT RECORD AT END MOVE "Y" TO END-OF-FILE.
    IF END-OF-FILE NOT = "Y" DISPLAY INPUT-RECORD.
A100-GET-RECORDS-EXIT.
    EXIT.
```

4.7 Updating Indexed Files

To update a record in an indexed file your program must:

1. If you are using sequential access mode:
   - Read the target record.
   - Verify that this record is indeed the record you want to change (once it's gone, it's gone).
   - Change the record.
   - Rewrite or delete the record.
2. If you are using random access mode: rewrite or delete the record.

Your program can update an indexed file three ways:

1. Sequentially
2. Randomly
3. Dynamically

Note

A program cannot rewrite an existing record if it changes the contents of the primary key in that record. Instead, the program must delete the record, and write a new record. However, the program can change the value of any alternate key in any record at any time, if the file description entry includes the WITH DUPLICATES phrase in its ALTERNATE RECORD KEY IS clause.

4.7.1 Sequential Updating

To update indexed records in a sequential mode:

1. Specify the ACCESS MODE IS SEQUENTIAL clause.
2. Open the file for INPUT-OUTPUT.
3. Read records as you would a sequential file, that is, use the READ...AT END... statement.
4. Rewrite or delete records using the INVALID KEY phrase.

The READ statement makes the next logical record of an open file available to the program. It skips deleted records and sequentially reads and retrieves only valid records. When the at end condition occurs, execution of the READ statement is unsuccessful (see Chapter 5, Input-Output Exception Conditions Handling). The REWRITE statement replaces the record just read on the file, while the DELETE statement logically removes the same record from the file.

Example 4-6: Sequentially Updating an Indexed File

```
IDENTIFICATION DIVISION.
PROGRAM-ID. INDEX06.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.

    SELECT FLAVORS    ASSIGN TO "DAIRY.DAT"
                      ORGANIZATION IS INDEXED
                      ACCESS MODE IS SEQUENTIAL
                      RECORD KEY IS ICE-CREAM-MASTER-KEY
                      ALTERNATE RECORD KEY IS ICE-CREAM-STORE-STATE
                                           WITH DUPLICATES
                      ALTERNATE RECORD KEY IS ICE-CREAM-STORE-CODE.
```

(continued on next page)

Example 4-6: Sequentially Updating an Indexed File (Cont.)

```
DATA DIVISION.
FILE SECTION.

FD  FLAVORS.
01  ICE-CREAM-MASTER.
    02 ICE-CREAM-MASTER-KEY          PIC XXXX.
    02 ICE-CREAM-MASTER-DATA.
       03  ICE-CREAM-STORE-CODE      PIC XXXXX.
       03  ICE-CREAM-STORE-ADDRESS   PIC X(20).
       03  ICE-CREAM-STORE-CITY      PIC X(20).
       03  ICE-CREAM-STORE-STATE     PIC XX.

WORKING-STORAGE SECTION.
01  END-OF-FILE                      PIC X.
01  REWRITE-KEY                      PIC XXXXX.
01  DELETE-KEY                       PIC XX.
01  NEW-ADDRESS                      PIC X(20).

PROCEDURE DIVISION.
A000-BEGIN.
    OPEN I-O FLAVORS.
    DISPLAY "Which store code do you want to find?".
    ACCEPT REWRITE-KEY.
    DISPLAY "What is its new address?".
    ACCEPT NEW-ADDRESS.
    DISPLAY "Which state do you want to delete?".
    ACCEPT DELETE-KEY.
    PERFORM A100-READ-INPUT UNTIL END-OF-FILE = "Y".

A020-EOJ.
    DISPLAY "END OF JOB".
    STOP RUN.

A100-READ-INPUT.
    READ  FLAVORS AT END MOVE "Y" TO END-OF-FILE.
    IF END-OF-FILE NOT = "Y" AND
       REWRITE-KEY = ICE-CREAM-STORE-CODE
       PERFORM A200-REWRITE-MASTER.
    IF END-OF-FILE NOT = "Y" AND
       DELETE-KEY  = ICE-CREAM-STORE-STATE
       PERFORM A300-DELETE-MASTER.

A200-REWRITE-MASTER.
    MOVE NEW-ADDRESS TO ICE-CREAM-STORE-ADDRESS.
    REWRITE ICE-CREAM-MASTER
            INVALID KEY DISPLAY "Bad rewrite - ABORTED"
                        STOP RUN.

A300-DELETE-MASTER.
    DELETE FLAVORS.
```

4.7.2 Random Updating

To randomly update indexed records:

1. Specify the ACCESS MODE IS RANDOM clause.
2. Open the file for INPUT-OUTPUT.
3. Initialize the RECORD KEY or ALTERNATE RECORD KEY data name.
4. Write, rewrite, or delete records using the INVALID KEY phrase.

Example 4-7 shows one way to randomly update records in an indexed file.

Example 4-7: Randomly Updating an Indexed File

```
IDENTIFICATION DIVISION.
PROGRAM-ID. INDEX07.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT FLAVORS    ASSIGN TO "DAIRY.DAT"
                      ORGANIZATION IS INDEXED
                      ACCESS MODE IS RANDOM
                      RECORD KEY IS ICE-CREAM-MASTER-KEY
                      ALTERNATE RECORD KEY IS ICE-CREAM-STORE-STATE
                                           WITH DUPLICATES
                      ALTERNATE RECORD KEY IS ICE-CREAM-STORE-CODE.

DATA DIVISION.
FILE SECTION.
FD  FLAVORS.
01  ICE-CREAM-MASTER.
    02 ICE-CREAM-MASTER-KEY          PIC XXXX.
    02 ICE-CREAM-MASTER-DATA.
       03  ICE-CREAM-STORE-CODE      PIC XXXXX.
       03  ICE-CREAM-STORE-ADDRESS   PIC X(20).
       03  ICE-CREAM-STORE-CITY      PIC X(20).
       03  ICE-CREAM-STORE-STATE     PIC XX.

WORKING-STORAGE SECTION.
01  HOLD-ICE-CREAM-MASTER            PIC X(51).
01  PROGRAM-STAT                     PIC X.
    88  OPERATOR-STOPS-IT                    VALUE "1".
    88  LETS-SEE-NEXT-STORE                  VALUE "2".
    88  NO-MORE-DUPLICATES                   VALUE "3".

PROCEDURE DIVISION.
A000-BEGIN.
    OPEN I-O FLAVORS.
    PERFORM A030-RANDOM-READ UNTIL OPERATOR-STOPS-IT.

A020-EOJ.
    DISPLAY "END OF JOB".
    STOP RUN.

A030-RANDOM-READ.
    DISPLAY "Enter key".
    ACCEPT ICE-CREAM-MASTER-KEY.
    PERFORM A100-READ-INPUT-BY-PRIMARY-KEY
            THROUGH A100-READ-INPUT-EXIT.
    DISPLAY " Do you want to terminate the session?".
    PERFORM A040-GET-ANSWER UNTIL PROGRAM-STAT   = "Y" OR "N".
    IF PROGRAM-STAT   = "Y" MOVE "1" TO PROGRAM-STAT   .

A040-GET-ANSWER.
        DISPLAY "Please answer Y or N"
        ACCEPT PROGRAM-STAT  .

A100-READ-INPUT-BY-PRIMARY-KEY.
    READ FLAVORS KEY IS ICE-CREAM-MASTER-KEY
         INVALID KEY DISPLAY "Master does not exist - Try again"
         GO TO A100-READ-INPUT-EXIT.
    DISPLAY ICE-CREAM-MASTER.
    PERFORM A200-READ-BY-ALTERNATE-KEY UNTIL NO-MORE-DUPLICATES.

A100-READ-INPUT-EXIT.
    EXIT.
```

(continued on next page)

Example 4-7: Randomly Updating an Indexed File (Cont.)

```
A200-READ-BY-ALTERNATE-KEY.
    DISPLAY "Do you want to see the next store in this state?".
    PERFORM A040-GET-ANSWER UNTIL PROGRAM-STAT   = "Y" OR "N".
    IF PROGRAM-STAT   = "Y"
       MOVE "2" TO PROGRAM-STAT
       READ FLAVORS KEY IS ICE-CREAM-STORE-STATE
                    INVALID KEY DISPLAY "No more stores in this state"
                                MOVE "3" TO PROGRAM-STAT  .
    IF LETS-SEE-NEXT-STORE AND
       ICE-CREAM-STORE-STATE = "NY"
             PERFORM A500-DELETE-RANDOM-RECORD.
    IF LETS-SEE-NEXT-STORE AND
       ICE-CREAM-STORE-STATE = "NJ"
             MOVE "Monmouth" TO ICE-CREAM-STORE-CITY
             PERFORM A400-REWRITE-RANDOM-RECORD.
    IF LETS-SEE-NEXT-STORE AND
       ICE-CREAM-STORE-STATE = "CA"
             MOVE ICE-CREAM-MASTER TO HOLD-ICE-CREAM-MASTER
             PERFORM A500-DELETE-RANDOM-RECORD
             MOVE HOLD-ICE-CREAM-MASTER TO ICE-CREAM-MASTER
             MOVE "AZ" TO ICE-CREAM-STORE-STATE
             PERFORM A300-WRITE-RANDOM-RECORD.
    IF PROGRAM-STAT   = "N"
       MOVE "3" TO PROGRAM-STAT  .

A300-WRITE-RANDOM-RECORD.
    WRITE ICE-CREAM-MASTER
          INVALID KEY DISPLAY "Bad write - ABORTED"
                      STOP RUN.

A400-REWRITE-RANDOM-RECORD.
    REWRITE ICE-CREAM-MASTER
            INVALID KEY DISPLAY "Bad rewrite - ABORTED"
                        STOP RUN.

A500-DELETE-RANDOM-RECORD.
    DELETE FLAVORS
           INVALID KEY DISPLAY "Bad delete - ABORTED"
                       STOP RUN.
```

4.7.3 Dynamic Updating

In dynamic mode, the program can switch from using random access I/O statements to sequential access I/O statements in any order without closing and reopening files. To dynamically update indexed records:

1. Specify the ACCESS MODE IS DYNAMIC clause.
2. Open the file for INPUT-OUTPUT.
3. Read the records in one of two ways:
   - Sequentially:
     - Use the START statement to position the record pointer.
     - Use the READ...NEXT statement.
   - Randomly:
     - Initialize the RECORD KEY or ALTERNATE RECORD KEY data name.
     - Read records in any order you want using the INVALID KEY phrase.
4. Write, rewrite, or delete records using the INVALID KEY phrase.

4.8 Backing Up Your Indexed Files

As you update the records in your indexed files, RMS-11 updates its record pointers. If the file becomes corrupted with bad data, or if your program abnormally terminates when the file is opened for OUTPUT or I-O, the file might become unusable because the pointers might not get updated. This can be a serious problem. Proper planning is the key for a successful recovery.

You should back up your indexed file at some reasonable point (daily, weekly, or monthly), depending on file activity, and save all transactions until you create a new backup. In this way, you can easily recreate your current indexed file from your last backup indexed file and transaction file whenever the need arises.

Chapter 5
Input/Output Exception Conditions Handling

Many types of exception conditions can occur when a program processes a file, not all of which are errors. The three categories of conditions are:

1. At end condition – This is a normal condition when you access a file sequentially. However, if your program tries to read the file anytime after having read the last logical record in the file, and there is no applicable Declarative procedure or AT END phrase, the program abnormally terminates when the next READ statement executes.

2. Invalid key condition – When processing relative and indexed files, this can either be a normal condition (if you expect it to happen and plan for it) or an abnormal condition that causes your program to terminate if there is no applicable Declarative procedure or INVALID KEY phrase.

3. All other conditions – These can also either be normal conditions (if you expect them to happen and plan for them) or they can be abnormal conditions that cause your program to terminate.

Planning for exception conditions is an effective way to increase program and programmer productivity. A program with exception handling routines is more flexible than a program without them. They minimize operator intervention and often reduce or eliminate the time a programmer uses to debug and rerun the program.

This chapter introduces you to the tools you will need to execute sequential, relative, and indexed file exception handling routines as a normal part of your program. The tools you will need are:

- The AT END phrase
- The INVALID KEY phrase
- File Status Values
- Special registers – RMS-STS and RMS-STV
- Declarative Procedures

5.1 Planning for the At End Condition

COBOL-81 provides you with the option of testing for this condition with the AT END phrase of the READ statement for sequential, relative, and indexed files.

Programs often read sequential files from beginning to end. They can produce reports from the information in the file or even update it. However, the program must know when it reaches the end of the file, so that it can continue normal processing after experiencing the condition. If the program does not test for this condition when it occurs, and if no applicable Declarative procedure exists (see Section 5.4), the program terminates abnormally upon detecting it. The program must know when no more data is available from the file so that it can perform its normal end-of-job totaling, balancing, and closing of the file.

Example 5-1: Handling the At End Condition

```
READ SEQUENTIAL-FILE AT END PERFORM A600-TOTAL-ROUTINES
                            PERFORM A610-VERIFY-TOTALS-ROUTINES
                            MOVE "Y" TO END-OF-FILE.

READ RELATIVE-FILE NEXT RECORD AT END PERFORM A700-CLEAN-UP-ROUTINES
                                      CLOSE RELATIVE-FILE
                                      STOP RUN.

READ INDEXED-FILE NEXT RECORD AT END DISPLAY "End of file"
                                     DISPLAY "Do you want to continue?"
                                     ACCEPT REPLY
                                     PERFORM A700-CLEAN-UP-ROUTINES.
```

5.2 Planning for the Invalid Key Condition

An invalid key condition occurs whenever RMS-11 cannot complete a COBOL-81 DELETE, READ, REWRITE, START, or WRITE statement. When the condition occurs, execution of the statement that recognized it is unsuccessful, and the file is not affected.

For example, relative and indexed files use keys to retrieve records. The program specifying random access must initialize a key before executing a DELETE, READ, REWRITE, START, or WRITE statement. If the key does not result in the successful execution of any of these statements, the invalid key condition exists. This condition is fatal to the program, if the program does not check for the condition when it occurs and if no applicable Declarative procedure exists (see Section 5.4).

This condition, although fatal if not planned for, can be used to your advantage when properly used. You can, as in Example 5-2, read through an indexed file for all records with a specific duplicate key and produce a report from the information in those records. However, after you have read the last of the duplicate records, an invalid key condition exists for subsequent reads to indicate that there are no more records in the file with this key. Planning for the invalid key condition in this case allows the program to continue its normal processing.

Example 5-2: Handling the Invalid Key Condition

```
     .
     .
     .
     MOVE "SMITH" TO LAST-NAME.
     MOVE "Y" TO ANY-MORE-DUPLICATES.
     PERFORM A500-READ-DUPLICATES-ROUTINE
             UNTIL ANY-MORE-DUPLICATES = "N".
     .
     .
     .
```

(continued on next page)

Example 5-2: Handling the Invalid Key Condition (Cont.)

```
     STOP RUN.
A500-READ-DUPLICATES-ROUTINE.
     READ INDEXED-FILE RECORD INTO HOLD-RECORD
          KEY IS LAST-NAME
          INVALID KEY DISPLAY "Name not in file!" STOP RUN.
     PERFORM A510-READ-NEXT-DUPLICATES-ROUTINE
             UNTIL ANY-MORE-DUPLICATES = "N".

A510-READ-NEXT-DUPLICATES-ROUTINE.
     READ INDEXED-FILE NEXT RECORD
          AT END MOVE "N" TO ANY-MORE-DUPLICATES.
     IF ANY-MORE-DUPLICATES = "Y" PERFORM A700-PRINT-ROUTINES.

MOVE "N" TO ANY-MORE-DUPLICATES.
     .
     .
     .
A700-PRINT-ROUTINES.
     .
     .
     .
```

5.3 Using File Status Values

Your program can check for the specific cause of the failure of a file operation by checking for specific file status values in its exception handling routines. The values are provided by both:

- COBOL-81 – Use the FILE STATUS clause in a file description entry
- RMS-11 – Use the COBOL-81 special registers RMS-STS and RMS-STV

5.3.1 COBOL-81 File Status Values

The run-time execution of any COBOL-81 file processing statement results in an RMS-11 completion code value that reports the success or failure of the COBOL statement. To access this value you must specify the FILE STATUS clause in the file description entry, as shown in Example 5-3.

Example 5-3: Defining a File Status for a File

```
DATA DIVISION.
FILE SECTION.
FD  INDEXED-FILE

****************************************
*
    FILE STATUS IS INDEXED-FILE-STATUS.

*
****************************************

01  INDEXED-RECORD           PIC X(50).

WORKING-STORAGE SECTION.
01  INDEXED-FILE-STATUS      PIC XX.
```

The program can access this file status variable, INDEXED-FILE-STATUS, anywhere in the Procedure Division, and depending on its value, take a specific course of action without terminating the program. See how Example 5-4 uses the file status defined in Example 5-3. However, not all statements allow you to access the file status value as part of the statement. Your program has two options:

1. Examine the status value as part of an error recovery routine built into the statement itself. Only these relative and indexed file processing statements allow you to do this within the INVALID KEY clause: DELETE, READ, REWRITE, START, and WRITE. See Example 5-4.

2. Define a Declarative procedure to handle the condition (see Section 5.4). All file organizations and their I/O statements have this option available.

Example 5-4: Using the File Status Value in an Exception Handling Routine

```
PROCEDURE DIVISION.
A000-BEGIN.
    .
    .
    .
    DELETE INDEXED-RECORD
           INVALID KEY MOVE "Bad DELETE" to BAD-VERB-ID
                       PERFORM A900-EXCEPTION-HANDLING-ROUTINE.
    .
    .
    .
    READ INDEXED-FILE NEXT RECORD
         INVALID KEY MOVE "Bad READ" TO BAD-VERB-ID
                     PERFORM A900-EXCEPTION-HANDLING-ROUTINE.
    .
    .
    .
    REWRITE INDEXED-RECORD
            INVALID KEY MOVE "Bad REWRITE" TO BAD-VERB-ID
                        PERFORM A900-EXCEPTION-HANDLING-ROUTINE.
    .
    .
    .
    START INDEXED-FILE KEY IS EQUAL TO MASTER-KEY
          INVALID KEY MOVE "Bad START" TO BAD-VERB-ID
                      PERFORM A900-EXCEPTION-HANDLING-ROUTINE.
    .
    .
    .
    WRITE INDEXED-RECORD
          INVALID KEY MOVE "Bad WRITE" TO BAD-VERB-ID
                      PERFORM A900-EXCEPTION-HANDLING-ROUTINE.
    .
    .
    .
A900-EXCEPTION-HANDLING-ROUTINE.
    DISPLAY BAD-VERB-ID " - File Status Value = " INDEXED-FILE-STATUS.
    PERFORM A905-GET-ANSWER UNTIL ANSWER = "Y" OR "N".
    IF ANSWER = "N" STOP RUN.

A905-GET-ANSWER.
    DISPLAY "Do you want to continue?"
    DISPLAY "Please answer Y or N"
    ACCEPT ANSWER.
```

Every file processing statement in the Procedure Division of the *COBOL-81 Language Reference Manual* has a specific set of file status values in its Technical Notes section. This manual contains a complete list of all COBOL-81 file status values.

5.3.2 RMS-11 File Status Values

COBOL-81 checks for RMS-11 completion codes after each file and record operation. If the code indicates anything other than unconditional success, COBOL-81 maps the RMS-11 error code to a file status value. However, not all RMS-11 completion codes map to distinct file status values. Many RMS-11 completion codes map to a file status value of 30, a COBOL-81 code for permanent errors that have no corresponding file status value.

COBOL-81 provides two special registers, RMS-STS and RMS-STV, that supplement the file status values already available and allow the COBOL-81 program to directly access RMS-11 completion codes. Refer to the *RMS-11 Macro Reference Manual* for a complete list of RMS-11 completion codes.

You need not define these registers in your program. As special registers, they are available whenever and wherever you need to use them in the Procedure Division. However, if you define more than one file in the program, and intend to access these values, you must qualify your references to them. Notice the use of the WITH CONVERSION phrase of the DISPLAY statement in Example 5–5. This converts the PIC S9(4) COMP special registers from binary to decimal digits for terminal display.

Example 5-5: Referencing RMS-STS and RMS-STV Values

```
DATA DIVISION.
FILE SECTION.
FD  FILE-1.
01  RECORD-1         PIC X(50).

FD  FILE-2.
01  RECORD-2         PIC X(50).

WORKING-STORAGE SECTION.
01  ANSWER           PIC X.

PROCEDURE DIVISION.
A000-BEGIN.
    .
    .
    .
    WRITE RECORD-1 INVALID KEY PERFORM A901-REPORT-FILE1-STATUS.

    WRITE RECORD-2 INVALID KEY PERFORM A902-REPORT-FILE2-STATUS.
    .
    .
    .
A901-REPORT-FILE1-STATUS.
*******************************************
*
    DISPLAY "RMS-STS = " RMS-STS OF FILE-1 WITH CONVERSION.
    DISPLAY "RMS-STV = " RMS-STV OF FILE-1 WITH CONVERSION.
*
*******************************************
    PERFORM A999-GET-ANSWER UNTIL ANSWER = "Y" OR "N".
    IF ANSWER = "N" STOP RUN.

A902-REPORT-FILE2-STATUS.
*******************************************
*
    DISPLAY "RMS-STS = " RMS-STS OF FILE-2 WITH CONVERSION.
    DISPLAY "RMS-STV = " RMS-STV OF FILE-2 WITH CONVERSION.
*
*******************************************
    PERFORM A999-GET-ANSWER UNTIL ANSWER = "Y" OR "N".
    IF ANSWER = "N" STOP RUN.
```

(continued on next page)

Example 5-5: Referencing RMS-STS and RMS-STV Values (Cont.)

```
A999-GET-ANSWER.
    DISPLAY "Do you want to continue?"
    DISPLAY "Please answer Y or N"
    ACCEPT ANSWER.
```

5.4 Using Declarative Procedures to Handle Exception Conditions

A Declarative procedure executes whenever an I/O statement results in an exception condition (a file status value other than "00") and the I/O statement does not contain an AT END or INVALID KEY phrase. The AT END and INVALID KEY phrases take precedence over a Declarative procedure but only for the I/O statement that includes the clause. Therefore you can have specific I/O statement exception condition handling for a file and also include a Declarative procedure for general exception handling.

A Declarative is a set of one or more special-purpose sections (called Declarative procedures) at the beginning of the Procedure Division. As shown in Example 5-6, the key word DECLARATIVES precedes the first of these sections, and the key words END DECLARATIVES follow the last.

Example 5-6: The Declarative Skeleton

```
PROCEDURE DIVISION.
DECLARATIVES.
        .
        .
        .
END DECLARATIVES.
```

As shown in Example 5-7, a Declarative procedure consists of a section header, followed, in order, by a USE sentence and zero, one, or more paragraphs.

Example 5-7: A Declarative Procedure Skeleton

```
DO-00-FILE-A-PROBLEM SECTION.
    USE AFTER STANDARD ERROR PROCEDURE ON FILE-A.

DO-01-FILE-A-PROBLEM.
        .
        .
        .
DO-02-FILE-A-PROBLEM.
        .
        .
        .
DO-03-FILE-A-PROBLEM.
        .
        .
        .
```

COBOL-81 Declarative procedures execute based on these five types of conditions in the USE statement:

1. File name – You can define a file name Declarative procedure for each file name. This procedure overrides the next four procedures. It executes for any unsuccessful exception condition.

2. INPUT – You can define only one INPUT Declarative procedure for each program. This procedure executes for any unsuccessful exception condition if: (1) the file is open for INPUT and, (2) a file name Declarative does not exist for that file.

3. OUTPUT – You can define only one OUTPUT Declarative procedure for each program. This procedure executes for any unsuccessful exception condition if: (1) the file is open for OUTPUT and, (2) a file name Declarative does not exist for that file.

4. I-O – You can define only one I-O Declarative procedure for each program. This procedure executes for any unsuccessful exception condition if: (1) the file is open for I-O, and (2) a file name Declarative does not exist for that file.

5. EXTEND – You can define only one EXTEND Declarative procedure for each program. This procedure executes for any unsuccessful exception condition if: (1) the file is open for EXTEND, and (2) a file name Declarative does not exist for that file.

The USE statement itself does not execute. It defines the condition that caused execution of the Declarative procedure. For more information about Declarative procedures, refer to the USE statement in the *COBOL-81 Language Reference Manual.*

Example 5-8 gives you a sample of how to include each condition in your program and contains explanatory comments for each.

Example 5-8: Five Types of Declarative Procedures

```
PROCEDURE DIVISION.
DECLARATIVES.
*******************************************************
D1-00-FILE-A-PROBLEM SECTION.
    USE AFTER STANDARD ERROR PROCEDURE ON FILE-A.
*
*
* If any I/O statement for FILE-A results in an
* error, D1-00-FILE-A-PROBLEM executes.
*
*
D1-01-FILE-A-PROBLEM.
    PERFORM D9-00-REPORT-FILE-STATUS.
    .
    .
    .

*******************************************************
D2-00-FILE-INPUT-PROBLEM SECTION.
    USE AFTER STANDARD EXCEPTION PROCEDURE ON INPUT.
*
*
* If an error occurs because of an I/O statement
* for any file open in the input mode except FILE-A,
* D2-00-FILE-INPUT-PROBLEM executes.
*
*
D2-01-FILE-INPUT-PROBLEM.
    PERFORM D9-00-REPORT-FILE-STATUS.
    .
    .
    .
```

(continued on next page)

Example 5-8: Five Types of Declarative Procedures (Cont.)

```
******************************************************
D3-00-FILE-OUTPUT-PROBLEM SECTION.
    USE AFTER STANDARD EXCEPTION PROCEDURE ON OUTPUT.
*
*
* If an error occurs because of an I/O statement
* for any file open in the output mode except FILE-A,
* D3-00-FILE-OUTPUT-PROBLEM executes.
*
*
D3-01-FILE-OUTPUT-PROBLEM.
    PERFORM D9-00-REPORT-FILE-STATUS.
    .
    .
    .

******************************************************
D4-00-FILE-I-O-PROBLEM SECTION.
    USE AFTER STANDARD EXCEPTION PROCEDURE ON I-O.
*
*
* If an error occurs because of an I/O statement
* for any file open in the input-output mode except FILE-A,
* D4-00-FILE-I-O-PROBLEM executes.
*
*
*
D4-01-FILE-I-O-PROBLEM.
    PERFORM D9-00-REPORT-FILE-STATUS.
    .
    .
    .

******************************************************
D5-00-FILE-EXTEND-PROBLEM SECTION.
    USE AFTER STANDARD EXCEPTION PROCEDURE ON EXTEND.
*
*
* If an error occurs because of an I/O statement
* for any file open in the extend mode except FILE-A,
* D5-00-FILE-EXTEND-PROBLEM executes.
*
*
D5-01-FILE-EXTEND-PROBLEM.
    PERFORM D9-00-REPORT-FILE-STATUS.
    .
    .
    .

******************************************************
D9-00-REPORT-FILE-STATUS SECTION.
    .
    .
    .
END DECLARATIVES

******************************************************
A000-BEGIN SECTION.
    .
    .
    .
```

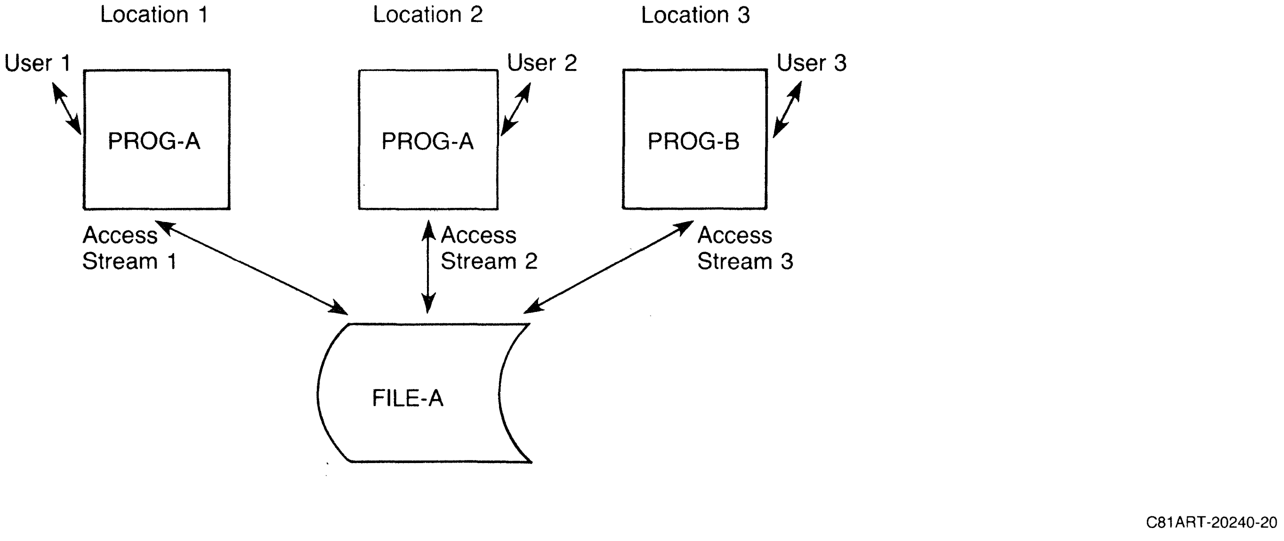
Chapter 6
Sharing Files and Protecting Records

This chapter introduces and discusses COBOL-81 file sharing and record locking for sequential, relative, and indexed files. For system-specific information, refer to the *RMS-11 User's Guide*.

6.1 File Sharing and Record Locking Concepts

In a data manipulation environment where many users and programs access the same data, file control must be used to protect files from nonprivileged users, to permit the right degree of file sharing for other users, and to promote data integrity in the files. For example, in Figure 6-1, many users and programs want to access data found in FILE-A.

Figure 6-1: Multiple Access to a File



File sharing and record locking allow you to control file and record operations when more than one access stream is concurrently accessing a file to perform file and record operations. See Figure 6-1.

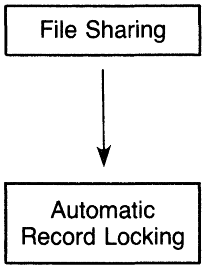
You create one RMS-11 access stream with each OPEN statement. The OPEN statement readies the file for subsequent record operations. The access stream remains active until you either terminate the access stream with the CLOSE statement or when your program terminates. A COBOL-81 program can define one or more access streams. For more information on access streams, see your system's RMS-11 documentation.

File sharing allows readers and writers to access a particular file concurrently. The protection level of the file, set by the file owner, determines whether or not a file can be shared by a particular type of user.

Record locking controls simultaneous record operations in files that are being accessed concurrently. Record locking ensures that when a program is writing, deleting, or rewriting a record on a given access stream, another access stream is allowed to access the same record in a specified manner.

Figure 6-2 illustrates the relationship of record locking to file sharing. A program must successfully meet all file-sharing requirements before it can share the records within that file with other concurrent programs and their access streams.

Figure 6-2: File Sharing and Record Locking Relationship



C81ART-20250-20

File sharing is a function of the File System, while record locking is a function of RMS-11. The File System controls the placement of and access to files. It manages the file-sharing process where multiple access streams simultaneously access a file. RMS-11 provides access methods to records within a file. This includes managing the record-locking process where multiple access streams simultaneously access a record.

You must have successful file sharing before you can consider record locking.

With COBOL-81, the file operations begin with an OPEN statement and end with a CLOSE statement. The OPEN statement initializes an access stream. The CLOSE statement terminates an access stream and can be either explicit (by the program) or implicit (on the program termination).

With COBOL-81, you use the ALLOWING clause in the OPEN statement to provide the file-sharing and record-locking specification. This clause describes what operations other access streams can perform. It provides the specification for the read/write intentions of your access stream.

Note

The first program to open a file determines how other programs can access the file simultaneously.

The record operations for COBOL-81 are:

- READ
- START
- WRITE
- REWRITE
- DELETE

6.2 Ensuring Successful File Sharing

File sharing requires that you:

- Provide disk residency for the file
- Use the PDP-11 system file protection facility
- Determine the intended access mode to the file (COBOL-81 open modes)
- Indicate the access allowed by other streams (COBOL-81 ALLOWING clause)

The remainder of this section discusses these four requirements for file sharing.

6.2.1 Providing Disk Residency

Only files that reside on disk can be shared. With COBOL-81, you can share relative and indexed files. Also, you can share sequential files with any number of users but in read-only access mode (RSTS/E allows multiple readers and only one writer).

6.2.2 Using File Protection

By using the appropriate file protection, the owner of a file determines how other users can access the file. An owner can permit up to four types of file access for each of four user categories. The level of file protection the file owner specifies determines the types of successful opens that a COBOL-81 program can specify. Those four types of file access for each user category are:

- Read – Permits the reading of the records in the file
- Write – Permits updating or extending the records in the file
- Extend – Permits extending records in the file (RSX-11M/M-PLUS)
- Delete – Permits deletion of the file and is, therefore, not applicable to a COBOL-81 program (since COBOL-81 has no delete file facility)

In the PDP-11 file protection facility, four different categories of users exist with respect to data structures and devices. A file owner decides which of these users can share the file:

- SYSTEM – Users of the system with certain I/O-related privileges. Project Programmer Numbers (PPN) or User Identification Codes (UIC) such as project 1, programmer 20 – [1,20]
- OWNER – Users whose PPN or UIC is identical to the PPN or UIC of the file owner such as [52,20]

- GROUP – Users of the system whose project number is identical to the project number of the file owner such as 52 in [52,20]
- WORLD – All users of the system

The owner of the file has a default protection that the system applies to each newly created file unless the owner specifically requests modified protection.

See your system's documentation for more information on its file protection.

6.2.3 Determining the Intended Access Mode to a File

Once you establish disk residency and privileges for a file, you can consider the third file sharing criterion – how the stream accesses the file. You specify this access by using the COBOL-81 OPEN and ACCESS modes.

The COBOL-81 open modes are INPUT, OUTPUT, EXTEND, and I-O. The COBOL-81 access mechanisms are: SEQUENTIAL, RANDOM, and DYNAMIC. The combination of OPEN and ACCESS modes determines the operations intended to be performed on the file.

You must validate your COBOL-81 file processing intention against the file protection that was assigned by the file owner. For example, to use an OPEN INPUT statement requires that the owner of the file has granted read access privileges to the file; to use an OPEN OUTPUT or EXTEND requires that the owner has granted write access privileges to the file; while to use an OPEN I-O means the owner has granted both read and write access privileges to the file.

Table 6-1 shows the relationship between open mode, access mode, and intended COBOL-81 operations. The DYNAMIC access mode is omitted because the result of its intention is always the same as RANDOM. The key word, ANY, indicates that all access methods result in the same intentions.

Table 6-1: File Sharing and Intended COBOL-81 Operations

Open Mode	Access Mode	Intended COBOL-81 Operations
INPUT	ANY	READ,START
OUTPUT	ANY	WRITE
I-O	SEQUENTIAL	READ,START,REWRITE,DELETE
	RANDOM	READ,START,REWRITE,DELETE,WRITE
EXTEND	SEQUENTIAL	WRITE

Note that if the file protection does not permit the intended operations, file access is not granted on the file, even if open and access modes are compatible.

File protection and open mode access apply to both nonshared and shared (includes more than one access stream) files. A file protection and intent check are made when the first access stream opens a file (nonshared file environment), and again when the second and subsequent access streams open the file (shared file environment).

After these file sharing checks pass, you can apply the fourth file sharing criterion – access allowed to other streams.

6.2.4 Indicating the Access Allowed to Other Streams

You use the COBOL-81 ALLOWING clause of the OPEN statement to specify what other access streams are allowed to that file.

The OPEN ALLOWING options are as follows:

1. OPEN ALLOWING READERS – Locks the file against operations that indicate intended write access (OPEN I-O and OPEN EXTEND). Other streams can open the file for INPUT.
2. OPEN ALLOWING ALL – Allows read and write access by other streams. Other access streams can open the file for INPUT, EXTEND, and I-O modes.

The first access stream uses the ALLOWING clause to specify what other access streams can do. When the second and subsequent access streams attempt to open the file, the following checks occur:

1. The allowed options of this access stream are checked against the intended access of the previous stream.
2. The intended access of this access stream are checked against the allowed access of the previous stream.

For example, if the first access stream permits ALLOWING READERS, then a subsequent access stream that opens the file ALLOWING ALL would fail because the clause option and the I-O mode declares write intent to the file.

6.3 Describing Types of Access Streams

You can establish several types of access streams. For example, two programs opening the same file represent two access streams to that file. Both programs begin with the file open, perform record operations, and then close the file.

In addition, a single program can establish multiple access streams to a file. In this case, you use multiple SELECT clauses to choose the file, while the FD and all other clauses and statements treat the file independently. Example 6-1 shows two access streams to the same file.

Example 6-1: Creating Two Access Streams to a File from the Same Program

```
IDENTIFICATION DIVISION.
PROGRAM-ID. ACCESSTRM.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT FILE-1
           ORGANIZATION IS SEQUENTIAL
           ASSIGN TO "SHAREDAT.DAT".
    SELECT FILE-2
           ORGANIZATION IS SEQUENTIAL
           ASSIGN TO "SHAREDAT.DAT".
DATA DIVISION.
FILE SECTION.
FD FILE-1 RECORD CONTAINS 512 CHARACTERS.
01 RECORD-1                PIC X(512).
FD FILE-2 RECORD CONTAINS 512 CHARACTERS.
01 RECORD-2                PIC X(512).
PROCEDURE DIVISION.
```

(continued on next page)

Example 6-1: Creating Two Access Streams to a File from the Same Program (Cont.)

```
A00-BEGIN.
    OPEN INPUT FILE-1 ALLOWING READERS.
    OPEN INPUT FILE-2 ALLOWING READERS.
*    .
*    .
*    .
    READ FILE-1  AT END DISPLAY "File-1 is AT END".
*    .
*    .
*    .
    READ FILE-2  AT END DISPLAY "File-2 is AT END".
*    .
*    .
*    .
A99-END.
    CLOSE FILE-1.
    CLOSE FILE-2.
    STOP RUN.
```

6.4 Summarizing Related File-Sharing Criteria

This section uses two tables to summarize the relationships among three of the four file-sharing criteria (disk residency, the first file-sharing requirement, is not included).

Table 6-2 summarizes the file protection and open mode requirements. For example, the file protection privilege READ (R) permits OPEN INPUT.

Table 6-2: File Protection and Open Mode Requirements for File Sharing

File Protection	Open Mode
R	INPUT
W	OUTPUT, EXTEND
RW	I-O, INPUT, OUTPUT, EXTEND

Table 6-3 shows the legal and illegal OPEN ALLOWING combinations between first and subsequent access streams.

Remember, you specify intended operations through the first access stream. For the second and subsequent shared access to a file, you use the access (OPEN modes) intentions and the ALLOWING clause to determine if and how a file is shared.

The abbreviations used in Table 6-3 are as follows:

1. OPEN ABBREVIATIONS
   - E,IO – OPEN EXTEND, OPEN I-O
   - I – OPEN INPUT
   - O – OPEN OUTPUT
2. ALLOWING ABBREVIATIONS
   - A – OPEN ALLOWING ALL
   - R – OPEN ALLOWING READERS

Table 6-3: File Sharing Options

Subsequent Stream *First Stream	E,IO A	E,IO R	I A	I R	O A,R
E,IO A	G	1	G	1	3
E,IO R	2	1,2	G	1	3
I A	G	G	G	G	3
I R	2	2	G	G	3
O A	G	G	G	1	3
O R	G	G	G	1	3

* Assumes no file protection violations on first stream.

Legend: **G** Second stream successfully opens and file sharing is granted.

1 Second stream denied access to the file because first intends write, while second specifies read-only sharing.

2 Second stream denied access to the file because second intends write, while first specifies read-only sharing.

3 No sharing. Second creates new file with OPEN OUTPUT.

C81ART-20260-30

In the following example, four streams illustrate some of the file sharing rules.

```
OPEN I-O    ALLOWING ALL        -   STREAM 1
OPEN INPUT  ALLOWING READERS    -   STREAM 2
OPEN I-O    ALLOWING ALL        -   STREAM 3
OPEN INPUT  ALLOWING ALL        -   STREAM 4
```

In this example:

- Stream 1 initiates file sharing by specifying ALLOWING ALL. This stream has read and write privileges.
- Stream 2's ALLOWING READERS file sharing option is compatible with stream 1. Stream 2 can file share and read the file.
- Stream 3's ALLOWING ALL – file sharing option violates the intent of stream 2 (readers only). Stream 3's OPEN I-O implies a write intention that stream 2 disallows. Consequently, stream 3 gets a file-locked error and becomes inactive.
- Stream 4's ALLOWING ALL file sharing option is compatible with stream 2. STREAM 4's OPEN INPUT implies a read intention only.

6.5 Checking File Operations

You frequently encounter the COBOL-81 file status values and RMS-11 completion codes in Table 6-4 when file sharing. You can check the success or failure of a file open operation by examining either the COBOL-81 file status values provided through the FILE STATUS IS clause in a file description entry or the RMS-11 completion/error codes provided by the COBOL-81 special registers RMS-STS and RMS-STV.

Table 6-4: COBOL-81 File Status Values and RMS-11 Completion/Error Codes

COBOL-81 File Status Values	RMS-STS Codes Decimal Value	Description and Symbolic Value
00	1	Successful Operation – SU$SUC
30	-1296	File Protection Violation – ER$PRV
91	-704	File is Locked – ER$FLK
92	-1440	Bucket Containing Record is Locked – ER$RLK
94	-1784	File is Closed WITH LOCK – ER$WLK

File status 30, when it corresponds to the RMS-11 STS symbol ER$PRV, is a result of a violation of the file protection codes as described in Section 6.2.2. To correct the condition, the file owner must reset the protection on the file or the directory that contains the file.

File status 94, which corresponds to the symbol ER$WLK in the RMS-11 STS special register, indicates that a previous accessor of the file has denied access by executing a CLOSE WITH LOCK statement.

For a complete list of COBOL-81 file status values, see Appendix C of the *COBOL-81 Language Reference Manual*. For a complete list of RMS-11 completion codes, see your system's *RMS-11 Macro Programmer's Guide*.

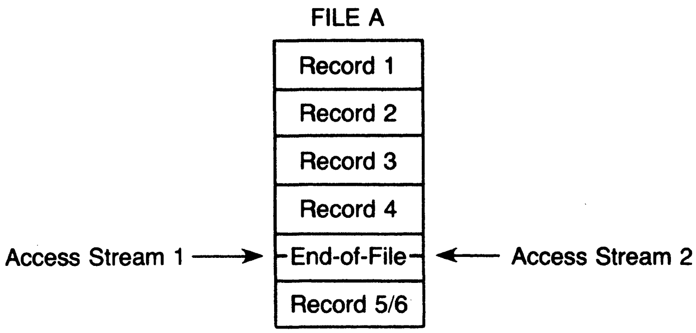
6.6 Specifying the OPEN EXTEND with a Sequential File

In a shared sequential file environment, when two concurrent access streams use EXTEND ALLOWING ALL, and both streams issue a write (to the end of the file – EOF), the additional data in the file comes from the stream that issued the last write to the file. Figure 6-3 illustrates why this data overwrite occurs.

As the file operations begin, both access streams point to the EOF. Next, note that the record to be written (immediately after the EOF) is never in a locked state. When access stream 1 writes to the file, record 5 is created. When access stream 1 executes a CLOSE statement, access stream 2 then writes to the file. Since access stream 2 still points to the EOF, the stream writes over the data written by stream 1 (record 5). Consequently, record 6 erases record 5 and supplies the new data in FILE A.

You can avoid this overwrite condition by better control of the allowing options in the system design.

Figure 6-3: The Overwrite Condition



C81ART-20270-15

6.7 Using Record Locking

Automatic record locking is the default. In automatic record locking, if you do not specify an ALLOWING clause on the OPEN statement, the default is ALLOWING READERS.

Automatic record locking applies the lock when you access the record and releases the lock when you deaccess the record. In automatic record locking, the access stream can have only one record locked at a time and can apply only one type of lock to the records of the file.

You deaccess a record by using the next READ operation, a REWRITE or a DELETE operation of the record, or by closing the file. When you close a file, any record lock that remains is released automatically.

In Example 6-2, the program uses automatic record locking. The program opens the file with I-O ALLOWING READERS clause. Another access stream in another program opens the file with INPUT ALLOWING ALL clause.

If the first access stream is updating records in random order with a REWRITE statement, a record lock can occur in the second stream from a READ statement until the REWRITE statement of the first stream terminates. Record locks can also occur in the first stream when the second stream reads a record and the first stream tries to read the same record.

Example 6-2: Automatic Record Locking

```
IDENTIFICATION DIVISION.
PROGRAM-ID. AUTOLOCK.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT FILE-1
           ASSIGN TO "SHAREDAT.DAT".
DATA DIVISION.
FILE SECTION.
FD FILE-1
   RECORD CONTAINS 100 CHARACTERS.
01 FILE-1-REC       PIC X(100).
PROCEDURE DIVISION.
A00-BEGIN.
    OPEN I-O FILE-1 ALLOWING READERS.
    READ FILE-1 AT END STOP RUN.
    REWRITE FILE-1-REC.
    CLOSE FILE-1.
    STOP RUN.
```

Chapter 7
File Optimization Techniques

COBOL-81 provides methods of controlling RMS-11 actions during I/O operations. You have the choice of accepting the defaults RMS-11 provides or using these optional methods to make your program more efficient.

The COBOL-81 language elements that can specify alternatives to the RMS-11 defaults are:

- The APPLY clause in the I-O-CONTROL paragraph
- The RESERVE n AREAS clause in the FILE-CONTROL paragraph
- The SAME [RECORD] AREA clause in the I-O-CONTROL paragraph
- The BLOCK CONTAINS clause in the FD entry

To use the optional I/O techniques to your advantage, you must understand how RMS-11 puts them into effect. This section explains COBOL-81 I/O optimization techniques in RMS-11 terms. It also presents the advantages and disadvantages of using each technique.

For a better understanding of the RMS-11 terms and concepts included in these explanations, see the *RMS-11 User's Guide*.

7.1 Using the APPLY Clause

You can specify eight different I/O techniques with the APPLY clause: DEFERRED WRITE, EXTENSION, FILL-SIZE, MASS-INSERT, PREALLOCATION, CONTIGUOUS PREALLOCATION, WINDOW, and PRINT-CONTROL. Except for PRINT-CONTROL, each APPLY technique can save time during program execution. However, there are disadvantages in using each one. This section explains how each technique saves execution time, and what you exchange for that saving.

APPLY DEFERRED-WRITE (relative and indexed files only)

> This technique causes RMS-11 to write several records at once, instead of writing each one separately. Normally, each WRITE or REWRITE operation performed by your program requires an I/O operation from RMS. With DEFERRED-WRITE, however, RMS-11 delays the operation until the file's I/O buffer is full. Using DEFERRED-WRITE saves execution time by reducing the number of write operations RMS-11 performs.
>
> If the system crashes during the program's execution, you might lose records currently in the file's I/O buffer. If RMS-11 has been delaying its write operations, the loss can be more than just one record.

APPLY EXTENSION integer (disk files only)

The integer you specify here overrides the default extension quantity (DEQ). The DEQ for sequential files is 5 blocks, and for relative and indexed files is four times the bucket size in blocks. The integer specifies the number of blocks RMS-11 uses to extend a file when the allocated space cannot hold another record. Using a large extension amount results in fewer extension operations by RMS-11 and, therefore, saves execution time.

A large extension amount can waste space if, for example, RMS-11 extends the file for the sake of one record and no other records are written to the file.

APPLY FILL-SIZE (indexed files only)

Fill size in COBOL is the same as an RMS-11 bucket fill size. Both indicate the number of bytes per bucket that RMS-11 fills when inserting records. RMS-11 normally fills the entire bucket; the only way to change this is by specifying a fill number when you create the file using the RMSDES or RMSDEF utility (see the *RMS-11 User's Guide*).

To keep this fill number in effect as you add records to the file with your COBOL-81 program, you must use APPLY FILL-SIZE. If you subsequently access the file without specifying FILL-SIZE, RMS-11 can use the extra bucket space to insert records, thereby avoiding bucket splitting and saving execution time. This technique has no effect unless you created the file with a bucket fill size using the RMS-11 RMSDES or RMSDEF utility.

This technique can waste bucket space unless you open the file without specifying FILL-SIZE and randomly insert records.

APPLY MASS-INSERT (indexed files only)

Use this technique when you have a series of records to add and:

- You have sorted the records into ascending order by the file's primary key
- The lowest key value in the records is greater than the highest key value in the file; that is, RMS-11 inserts the records at the logical end of file

Inserting all these records at once saves execution time by eliminating the index search for all but the first write operation.

The disadvantage in using MASS-INSERT is that you cannot perform random insertions while the technique is in effect.

APPLY PREALLOCATION (disk files only)

The integer you specify here overrides the default preallocation that RMS-11 uses when it creates a file. The default value for sequential files is four 512-byte blocks; for indexed files, the default is four times the bucket size. You can save execution time by specifying a value larger than the default, thereby reducing the number of times RMS-11 must extend the file before writing more records.

If the eventual size of the file is known, preallocation eliminates the RMS-11 overhead needed to extend the file. If the file grows beyond the initial preallocation, you can specify the APPLY EXTENSION clause to override the default extension value used by RMS-11.

Do not use this technique unless you plan to use the entire preallocated amount; otherwise, you waste disk space.

APPLY CONTIGUOUS PREALLOCATION (disk files only)

This technique requires that the blocks RMS-11 preallocates for the file be in a continuous logical series on disk. Your program runs faster if the file is contiguous because RMS-11 takes less time to locate a record.

You cannot extend contiguous files; a protection violation error occurs if you attempt to write a record and the allocated space is exhausted. Therefore, do not use contiguous preallocation unless you are sure of the file's eventual size. Then, if you must extend it, make the file noncontiguous.

APPLY WINDOW

The APPLY WINDOW value changes the value of the RTV field in the File Access Block (FAB) for the file named in the APPLY WINDOW clause. The meaning of the RTV field depends on your operating system. See the *RMS-11 User's Guide* for more information.

On an RSX-11M/M-PLUS system, you can override your default window size with the APPLY WINDOW clause. The one-byte Retrieval Window Size (RTV) field contains the number of retrieval pointers kept in memory for the file represented by the FAB. This set of pointers is called a "window." Retrieval pointers map virtual block numbers to logical block numbers. Table 7-1 lists the permissible window values.

On a RSTS/E system, you override the clustersize. The one-byte clustersize (RTV) field contains the number of blocks in each cluster of the file represented by the FAB. Table 7-1 lists the permissible clustersize values.

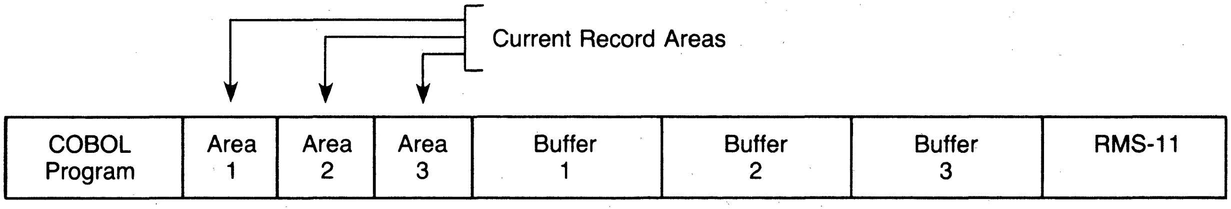
Table 7-1: Permissible APPLY WINDOW Values

Permissible Clustersize/Window Values		**Result**
RSTS/E Clustersize	**RSX-11M RSX-11M/PLUS Window Size**	
0	0	Causes RMS-11 to use the current default value
A power of two	1-127	Overrides the current default value
	255	As much of the file as possible is mapped with one window
255		RMS-11 sets the clustersize to 256 blocks

7.2 Current Record Area

Your program defines the current record area by the record descriptions that follow the file descriptions. The system creates one current record area for each file you open. Figure 7-1 shows a program with three record areas for its three files. All READ and WRITE operations use this area during record transfers to and from the I/O buffers (see Section 7.4).

Figure 7-1: Current Record Areas for Three Files



C81ART-20280-20

A file's current record area size is defined by the size of its largest record. The number of bytes transferred depends on the record description named in the WRITE or REWRITE statement or the RECORD VARYING phrase of a variable-length record file.

The system does not clear the current record area before executing a READ operation. Consider an example where the current record area contains 20 characters from the first record read from a file. If the next READ returns a 12-character record, the remaining 8 character positions do not change:

1. Current record area after first READ: `0239394CABINET, FILE`
2. Contents of second record: `6627402CHAIR`
3. Current record area after second READ: `6627402CHAIRET, FILE`

7.3 Sharing Record Areas

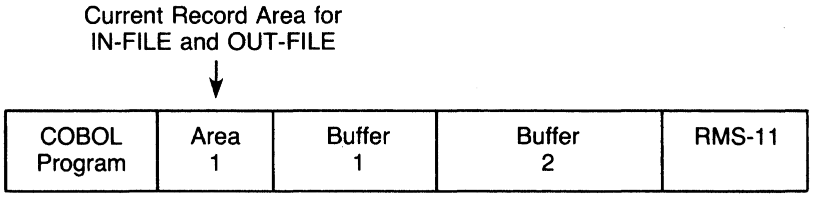
You can reduce your task size by specifying that two or more files share record areas. Normally, the COBOL-81 OTS allocates a current record area, which contains the most recently accessed record, for each file your program opens. The SAME RECORD AREA overrides this default. In addition to saving space, sharing record areas can improve execution by eliminating an extra MOVE operation when you are copying from an input file to an output file, as Example 7-1 and Figure 7-2 illustrate.

Example 7-1 Sharing Record Areas

Program Without Sharing   Program With Sharing

```
                                     I-O-CONTROL.
                                           SAME RECORD AREA FOR
                                               IN-FILE, OUT-FILE.
                                           .
                                           .
                                           .
PROCEDURE DIVISION.                  PROCEDURE DIVISION.
      .                                    .
      .                                    .
      .                                    .
      READ IN-FILE ...                     READ IN-FILE ...
      .                                    .
      .                                    .
      .                                    .
      MOVE IN-REC TO OUT-REC.
      WRITE OUT-REC ...                    WRITE OUT-REC ...
```

Figure 7-2: Two Files Sharing a Record Area



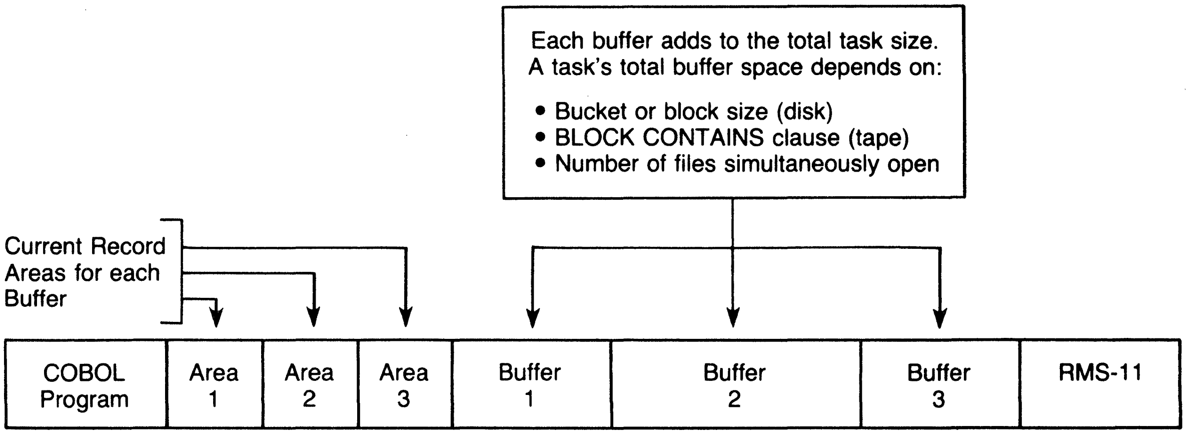
C81ART-20290-20

The example also shows the disadvantage of sharing record areas: the records from each file do not exist separately. Therefore, if your program changes a record in the record area defined for the output file, the record in the record area for the input file is also changed.

7.4 I/O Buffers

A buffer is a part of memory dedicated to a task (see Figure 7-3). It is a holding area for data moved to and from other storage areas.

Figure 7-3: A Program's Buffer Area in a Task Structure



Each buffer adds to the total task size.
A task's total buffer space depends on:
• Bucket or block size (disk)
• BLOCK CONTAINS clause (tape)
• Number of files simultaneously open
Current Record Areas for each Buffer
COBOL Program
Area 1
Area 2
Area 3
Buffer 1
Buffer 2
Buffer 3
RMS-11

C81ART-20300-35

Using buffers, the system can perform I/O operations independent of record and file descriptions. It can read and write more data (or less) in each operation than the program requests. For example, when your program reads a record that is part of a block, only that record is made available in the current record area (see Section 7.2). The remaining records in the block are still available in the file's I/O buffer. The system can make them available to your program without accessing the file for each COBOL-81 I/O operation.

For every file your program opens, COBOL-81 allocates an I/O buffer (unless you specify shared buffer areas; see Section 7.6). The size of each I/O buffer adds to your total task size.

By default, the compiler allocates a buffer size of one block, or 512 bytes, for sequential disk files. For relative files, it allocates a buffer size in bucket increments. For indexed files, it allocates a buffer size equal to twice the size of one bucket, which is a unit storage structure for indexed files.

You can override these defaults by using the BLOCK CONTAINS clause (see Section 7.7) For sequential files, COBOL-81 uses this clause to calculate the buffer size. For relative and indexed files, this clause is used to calculate the bucket size, which in turn determines the buffer size.

COBOL-81 also uses the BLOCK CONTAINS clause to determine each file's unit of transfer. This quantity is the amount of the file's data RMS-11 moves to and from the I/O buffer at a time.

7.5 Reserving Additional I/O Buffer Space for Your Files

You can override the system defaults in Table 7-1 with the RESERVE n AREAS clause. Reserving more buffers increases your task size, but it also increases execution speed because RMS-11 performs fewer disk accesses.

Table 7-2: Default Buffer Areas Reserved by the COBOL-81 OTS

File Organization	Reserved Buffer Areas
Sequential	1
Relative	1
Indexed	2

For sequential files, the RESERVE n AREAS clause does not actually create additional buffers. Instead, the OTS multiplies the number you specify by the block size (which is determined by the BLOCK CONTAINS clause; see Section 7.7) to establish the size of one buffer. Consider this example:

```
FILE-CONTROL.
    SELECT SEQ-FILE ASSIGN TO "TEST"
        ORGANIZATION IS SEQUENTIAL
        RESERVE 2 AREAS.
          .
          .
          .
FD SEQ-FILE
    BLOCK CONTAINS 512 CHARACTERS
          .
          .
          .
```

Rather than reserving two 512-byte buffer areas for SEQ-FILE, the OTS reserves one 1024-byte buffer.

Note

Using this clause on sequential magnetic tape files does not create a more efficient task. See Section 7.8.1 for an explanation.

Reserving additional buffer areas benefits sequential-access relative files when there are multiple records for each buffer. Locating the next record takes less time if it is in the same buffer. However, there is a trade-off: the larger the buffer, the larger the task, but the faster the program reads.

Reserving additional areas for indexed files gives RMS-11 more space to store the file's index structure. Locating a record takes less time if the record's index is stored in one of the buffer areas. As a general rule, allocate one buffer for each key that your program uses to access records, in addition to the two default buffer areas. For example, if the file contains a primary key and two alternate keys, and you use all these keys to access records, reserve a total of five buffer areas.

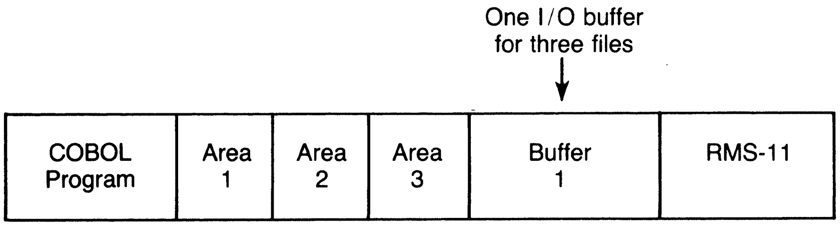
7.6 Sharing Buffer Areas

You can decrease your task size by specifying that two or more files are to share the same I/O buffer area. Normally, the COBOL-81 OTS establishes one I/O buffer for every file your program opens. The SAME AREA CLAUSE overrides that default. The code in Example 7-2 causes FILE1, FILE2, and FILE3 to share one I/O buffer, as shown in Figure 7-4.

Example 7-2: Statements Causing Three Files to Share One I/O Buffer

```
I-O-CONTROL.
        SAME AREA FOR FILE1 FILE2 FILE3.
```

Figure 7-4: Using One I/O Buffer to Process Three Files



C81ART-20310-20

Without the SAME AREA clause, the OTS would have allocated three separate buffers for these files. The amount of space you save depends on the buffer sizes defined for each file.

The only disadvantage in forcing files to share an I/O buffer is that only one of those files can be opened at a time during execution.

7.7 Tailoring I/O Buffers to Increase Speed of I/O Operations

Record blocking can increase the speed of I/O operations. The block size determines the number of records the system transfers in its buffer(s) during each I/O operation.

I/O-bound programs should contain a large BLOCK CONTAINS value, especially when using sequential operations. In fact, segmenting the program to allow for a larger buffer size can make a program execute faster depending on the application; for example, a sequential access of a heavily I/O bound program.

In general, a larger buffer size reduces the number of I/O transfers RMS-11 must perform, thereby improving execution speed. However, a larger buffer size also increases the size of your task image. Additionally, COBOL-81 performs rounding as it calculates buffer size, and the unit of transfer can differ from the buffer size. The rounding that occurs, and the difference between buffer size and unit of transfer, can create a large task image that executes inefficiently.

To make efficient use of the BLOCK CONTAINS clause, you must understand how COBOL-81 calculates:

- Buffer size and unit of transfer for sequential files
- Bucket size for relative and indexed files

The next sections explain these calculations and use these terms to refer to record blocking and I/O processing:

Logical Block

A group of consecutive bytes in memory.

Physical Block

A group of consecutive tape-resident data bytes treated as a unit; it is the number of bytes between interrecord gaps.

Block

A group of consecutive disk-resident data bytes treated as a unit. A block's size is 512 bytes.

Bucket

For relative and indexed files, the unit of transfer between storage devices and I/O buffers in memory. A bucket can contain one or more records; however, records cannot span buckets.

Bucket size

Consists of either one to thirty-two 512-byte blocks for an RSX system, or one to fifteen 512-byte blocks for a RSTS/E system, and determines the unit of transfer from disk to memory of I/O operations.

Record Unit Size

The storage medium space (in bytes) needed to store a record in a file:

- For fixed-length records, record unit size is the record length specified by either the record description or the RECORD CONTAINS clause, whichever is larger.
- For variable-length records, record unit size is the maximum record length plus the overhead bytes used to specify record length.

Note

Do not confuse "record size" with "record unit size." "Record size" refers to the number of character positions in a record – the number you define in the record description.

7.8 Optimizing File Design

This section introduces you to sequential, relative, and indexed file design optimization.

7.8.1 Sequential Files

Sequential files have the simplest structure and the fewest options for definition, population, and handling. You can reduce the number of disk accesses by keeping record length to a minimum.

7.8.1.1 Buffer Size Calculations for Sequential Files — COBOL-81 determines the file's buffer size (in bytes) using two different formulas, depending on which format of the BLOCK CONTAINS clause you use. The following items refer to the formulas used in this section:

1. AR = number of Areas Reserved

   This quantity is determined from the RESERVE n AREAS clause. If you do not specify a RESERVE n AREAS clause, the default is one area for sequential files (that is, AR = 1).

2. URS = Unit Record Size

   For fixed-length records, this equals the record length specified by the record description or the RECORD CONTAINS clause, whichever is larger.

   For variable-length records, it equals the maximum record length plus 4 overhead bytes.

The buffer size formula when using the BLOCK CONTAINS n CHARACTERS clause is:

```
buffer size = n * AR
```

Buffer size is in bytes and is rounded up to the next multiple of four.

The buffer size formula when using the BLOCK CONTAINS n RECORDS clause is:

```
buffer size = n * URS * AR
```

Buffer size is in bytes and is rounded up to the next multiple of four.

COBOL-81 allows buffer sizes that are not multiples of 512 because magnetic tape files do not have the 512-byte restriction (that is, they are not block-structured). However, because disk files are block-structured, the unit of transfer must be a multiple of 512. If you create a buffer size of 600 bytes, for example, the unit of transfer for a magnetic tape file is 600 bytes. For a disk file, however, it is 512 bytes (1 block), and the remaining 88 bytes are not used by RMS-11.

With a sequential disk file, you can use multiblocking to access a buffer area larger than the default. Because the system transfers disk data in 512-byte blocks, a blocking factor with a multiple of 512-bytes improves I/O access time. In the following example, the multiblock count of four (2048 divided by 512) causes reads and writes to FILE-A to access a buffer area of 4 physical blocks:

```
FILE SECTION.
FD   FILE-A
     BLOCK CONTAINS 2048 CHARACTERS
                          .
                          .
                          .
```

If you do not want to calculate the buffer size, but you want to specify the number of records in each buffer, use the BLOCK CONTAINS n RECORDS clause. This example specifies a buffer large enough to hold 15 records:

```
BLOCK CONTAINS 15 RECORDS
```

When using the BLOCK CONTAINS n RECORDS clause for sequential files on disk, RMS-11 calculates the buffer size by using the maximum record unit size and rounding down to a multiple of 512 bytes.

In the next example, the BLOCK CONTAINS clause specifies five records. RMS-11 calculates the block size as eight records, or 512 bytes.

```
FILE SECTION.
FD  FILE-A
    BLOCK CONTAINS 5 RECORDS.
01  FILE-A-REC      PIC X(64).
                    .
                    .
                    .
```

In short, these calculations show that you make most efficient use of your task's space if you:

- Do not reserve more than one area for a magnetic tape file
- Create a buffer size that is a multiple of 512 for a disk file

7.8.1.2 Unit of Transfer for Sequential Files on Magnetic Tape — For magnetic tape files, COBOL-81 uses the BLOCK CONTAINS clause to determine the unit of transfer in bytes. The system stores records in the block in the order they are written (first-to-last). Records cannot span physical blocks; therefore, a physical record can contain only complete records (regardless of record format). The block size on magnetic tape must be between 18 and 8192 bytes long.

If the block size on magnetic tape is not evenly divisible by 4, RMS-11 rounds up the block size to the nearest multiple of 4. The following items refer to the formulas used in this section:

1. AR = number of Areas Reserved

   This quantity is determined from the RESERVE n AREAS clause. If you do not specify a RESERVE n AREAS clause, the default is one area for sequential files (that is, AR = 1).

2. URS = Unit Record Size

   For fixed-length records, this equals the record length specified by the Record Description or the RECORD CONTAINS clause, whichever is larger.

   For variable-length records, it equals the maximum record length plus 4 overhead bytes.

If you do not specify the BLOCK CONTAINS clause, the default unit of transfer formula is:

Unit of transfer = A block size set by the size of the largest record

If you specify the BLOCK CONTAINS n CHARACTERS clause, the unit of transfer formula is:

Unit of transfer = n

Unit of transfer is in bytes and is rounded up to the next multiple of four.

Samples

```
FD  TEST-FILE
    BLOCK CONTAINS 512 CHARACTERS
    LABEL RECORDS ARE STANDARD.
01  REC-1    PIC  X(494).
```

Unit of transfer = 512 bytes

Buffer size = 512 * 1 = 512 bytes

```
FD  TEST-FILE
BLOCK CONTAINS 512 CHARACTERS
    RECORD IS VARYING
    FROM 400 TO 494 CHARACTERS
    LABEL RECORDS ARE STANDARD.
01  REC-1    PIC  X(494).
01  REC-2    PIC  X(400).
```

Unit of transfer = 512 bytes

Buffer size = 512 * 1 = 512 bytes

```
SELECT SEQ-FILE
       ASSIGN TO "MT1:REPORT.DAT"
       RESERVE 2 AREAS.
         .
         .
         .
FD SEQ-FILE
       BLOCK CONTAINS 500 CHARACTERS
         .
         .
         .
```

Unit of transfer= 500 bytes
Buffer size = 500 * 2 = 1000 bytes

If you specify the BLOCK CONTAINS n RECORDS clause, the unit of transfer formula is:

Unit of transfer = n * URS

Unit of transfer is in bytes and is rounded up to the next multiple of four.

Samples

```
FD  TEST-FILE
    BLOCK CONTAINS 50 RECORDS
    LABEL RECORDS ARE STANDARD.
01  REC-1    PIC  X(20).
```

Unit of transfer = 50 * 20 = 1000 bytes

Buffer size = 50 * 20 * 1 = 1000 bytes

```
FD  TEST-FILE
    BLOCK CONTAINS 8 RECORDS
    RECORD IS VARYING
    FROM 40 TO 88 CHARACTERS
    LABEL RECORDS ARE STANDARD.
01  REC-1    PIC  X(40).
01  REC-2    PIC  X(88).
```

Unit of transfer 8 * (88 + 4) = 736 bytes

Buffer size = 8 * (88 + 4) * 1 = 736 bytes

```
SELECT SEQ-FILE
    ASSIGN TO "M10:REPORT.DAT"
    RESERVE 3 AREAS.
         .
         .
         .
FD SEQ-FILE
    BLOCK CONTAINS 2 RECORDS
    RECORD CONTAINS 200 CHARACTERS
         .
         .
         .
```

Unit of transfer = 2 * 200 = 400 bytes

Buffer size = 2 * 200 * 3 = 1200 bytes

7.8.1.3 Unit of Transfer for Sequential Files on Disk — For disk files, COBOL-81 uses the buffer size to determine the unit of transfer in blocks. Records are packed together in each block, and the records can span block boundaries. The unit of transfer is simply the greatest number of 512-byte blocks contained in the buffer size.

The following items refer to the formulas used in this section:

1. AR = number of Areas Reserved

   This quantity is determined from the RESERVE n AREAS clause. If you do not specify a RESERVE n AREAS clause, the default is one area for sequential files (that is, AR = 1).

2. URS = Unit Record Size

   For fixed-length records, this equals the record length specified by the record description or the RECORD CONTAINS clause, whichever is larger.

   For variable-length records, it equals the maximum record length plus 4 overhead bytes.

If you do not specify the BLOCK CONTAINS clause, the default unit of transfer formula is:

Unit of transfer = One block, or 512 bytes.

If you specify the BLOCK CONTAINS n CHARACTERS clause, the unit of transfer formula is:

Unit of transfer = (n * AR) / 512

Unit of transfer is in blocks and is rounded down to the next integer.

Samples

```
FD  TEST-FILE
    BLOCK CONTAINS 512 CHARACTERS
    LABEL RECORDS ARE STANDARD.
01  REC-1     PIC X(494).
```

Unit of transfer =(512 * 1)/512=1 block

Buffer size = (512 * 1) / 512 = 1 block

```
FD  TEST-FILE
    BLOCK CONTAINS 512 CHARACTERS
    RECORD IS VARYING
    FROM 400 TO 494 CHARACTERS
    LABEL RECORDS ARE STANDARD.
01  REC-1     PIC  X(494).
01  REC-2     PIC  X(400).
```

Unit of transfer =(512 * 1)/512=1 block

Buffer size = (512 * 1) / 512 = 1 block

```
SELECT SEQ-FILE
    ASSIGN TO "REPORT.DAT"
    RESERVE 2 AREAS.

         .
         .
         .
FD SEQ-FILE
    BLOCK CONTAINS 500 CHARACTERS
    LABEL RECORDS STANDARD.
01  REC-1     PIC X(100).
```

Unit of transfer = (500 * 2)/512= 1.9...
rounded down = 1 block

Buffer size = 500 * 2 = 1000 bytes

If you specify the BLOCK CONTAINS n RECORDS clause, the unit of transfer formula is:

Unit of transfer = (n * URS * AR) / 512

Unit of transfer is in blocks and is rounded up to the next integer.

Samples

```
FD  TEST-FILE
    BLOCK CONTAINS 50 RECORDS
    LABEL RECORDS ARE STANDARD.
01  REC-1     PIC X(20).
```

Unit of transfer = (50 * 20 * 1) / 512
rounded down = 1 blocks

Buffer size = (50 * 20) * 1 = 1000 bytes

```
FD  TEST-FILE
    BLOCK CONTAINS 100 RECORDS
    RECORD VARYING
    FROM 10 TO 30 CHARACTERS
    LABEL RECORDS ARE STANDARD.
01  REC-1     PIC X(10).
01  REC-2     PIC X(20).
01  REC-3     PIC X(30).
```

Unit of transfer = (100 * (30+4) * 1)/512
rounded down = 6 blocks

Buffer size = (100 * (30+4) * 1) = 3400 bytes

```
SELECT SEQ-FILE
    ASSIGN TO "REPORT.DAT"
    RESERVE 3 AREAS.
        .
        .
        .
FD SEQ-FILE
    BLOCK CONTAINS 2 RECORDS
    RECORD CONTAINS 200 CHARACTERS
        .
        .
        .
```

Unit of transfer = (2 * 200 * 3)/512= 2.3...
rounded down = 2 blocks

Buffer size = 2 * 200 * 3 = 1200 bytes

7.8.2 Relative Files

I/O optimization of a relative file depends on four items:

1. Maximum record number – the highest numbered record written to a relative file.
2. Cell size – the unit of disk space needed to store a record unit size (record unit size = record + record overhead).
3. Bucket size – the number of blocks read or written in one I/O operation (equivalent to buffer size). To determine how many physical blocks a bucket can contain, refer to the *RMS-11 User's Guide*.
4. File size – the number of blocks used to preallocate the file.

7.8.2.1 Maximum Record Number (MRN) — If you create a relative file with a COBOL-81 program, the system sets the MRN to 0, allowing the file to expand to any size.

If you create a relative file with the RMSDEF utility, select a realistic MRN. An attempt to insert a record with a number higher than the MRN will fail.

7.8.2.2 Cell Size — The system calculates cell size. However, you can specify a different cell size when you create the file by using the RECORD CONTAINS clause in the file description. You cannot write records larger than the specified cell size.

The system calculates cell size using these formulas:

Fixed-length records: cell size = 1 + record size

Variable-length records: cell size = 3 + record size

For fixed-length records, the overhead byte is a record deletion indicator. Variable-length records use two additional overhead bytes to indicate record length. The following example calculates a cell size of 101 BYTES for fixed-length records:

```
FD   A-FILE
     RECORD CONTAINS 100 CHARACTERS
                     .
                     .
                     .
```

The next example calculates a cell size of 153 for variable-length records:

```
FD  B-FILE
    RECORD IS VARYING IN SIZE FROM 50 TO 150 CHARACTERS
                     .
                     .
                     .
```

Avoid selecting a cell size larger than necessary: this wastes disk space. To optimize the packing of cells into buckets, cell size should be evenly divisible into bucket size.

7.8.2.3 Bucket Size — A bucket's size is from one to thirty-two 512-byte blocks. A large bucket improves sequential access to a relative file. You can prevent wasted space between the last cell and the end of a bucket by specifying a bucket size that is a multiple of cell size. (See Section 7.8.2.5.)

If you omit the BLOCK CONTAINS clause, the system calculates a bucket size large enough to hold at least one cell (or 512 bytes, whichever is larger); that is, large enough to hold a record and its overhead byte(s). Records cannot cross bucket boundaries, although they can cross block boundaries.

To set your own bucket size (in bytes for each bucket), use the BLOCK CONTAINS n CHARACTERS clause of the file description. Consider this example:

```
FILE-CONTROL.
    SELECT A-FILE
      ORGANIZATION IS RELATIVE.
        .
        .
        .
DATA DIVISION.
FILE SECTION.
FD   A-FILE
     RECORD CONTAINS 60 CHARACTERS
     BLOCK CONTAINS 1536 CHARACTERS
        .
        .
        .
```

In the example, the bucket size is three 512-byte blocks. Each bucket contains:

25 records (25 X 60)	=	1500 bytes
1 overhead byte for each record (1 X 25)	=	25 bytes
11 bytes of wasted space	=	11 bytes
TOTAL	=	1536 bytes

If you use the BLOCK CONTAINS CHARACTERS clause and specify a value that is not a multiple of 512, then the compiler issues a warning diagnostic, and RMS-11 rounds the value to the next higher multiple of 512.

To improve I/O access time:

- Use the BLOCK CONTAINS n RECORDS clause to specify the number of records (cells) in each bucket
- Specify a small bucket size for random access
- Specify a large bucket size for sequential access

The following example creates buckets that contain eight records:

```
FD   A-FILE
     RECORD CONTAINS 60 CHARACTERS
     BLOCK CONTAINS 8 RECORDS.
         .
         .
         .
```

In the example, the bucket size is one 512-byte block. Each bucket contains:

8 records (8 x 60)	=	480 bytes
1 overhead byte for each record (1 x 8)	=	8 bytes
24 bytes of wasted space	=	24 bytes
TOTAL	=	512 bytes

7.8.2.4 File Size — Calculating a file's size helps you determine its space requirements. A file's size is a function of its bucket size. When you create a relative file, use these calculations to determine the number of blocks that you need:

$$\text{file size (in blocks)} = \frac{(511 + (\text{number of buckets} * \text{bytes per bucket}))}{512}$$

$$\text{number of buckets} = \frac{(\text{number of records in the file})}{(\text{number of cells for each bucket})}$$

Assume that you want to create a relative file able to hold 3,000 records. The records are 255 bytes long (plus one byte for each record for overhead), four cells to a bucket (BLOCK CONTAINS 4 RECORDS). To determine file size:

1. Calculate the number of buckets:

   $$750 = \frac{3000}{4}$$

2. Calculate bucket size (see Section 7.8.2.5):

   $$2 = \frac{(4 * (1 + 255))}{512}$$

3. Calculate bytes for each bucket (bucket size * number of bytes in a block):

   $$1024 = 2 * 512$$

4. Calculate file size:

   $$1500 \text{ physical blocks} = \frac{511 + (750 * 1024)}{512}$$

To populate the entire file, use the APPLY CONTIGUOUS PREALLOCATION clause, if possible, to allocate the 1500 calculated blocks for best performance.

Before writing a record to a relative file, RMS-11 must format all buckets up to and including the bucket to contain the record. Each time bucket reformatting occurs, response time suffers. Therefore, writing the highest-numbered record first forces formatting of the entire file only once. However, this technique can waste disk space if the file is only partially loaded and not preallocated.

7.8.2.5 Bucket Size and Buffer Size Calculations for Relative Files — COBOL-81 determines the file's bucket size (in 512-byte blocks) using two different formulas, depending on which format of the BLOCK CONTAINS clause you use. COBOL-81 then determines the file's buffer size (in blocks) using the bucket size:

$$\text{buffer size (in blocks)} = \text{(bucket size)} * \text{AR}$$

The unit of transfer for a relative file is equal to the bucket size.

The following items refer to the formulas used in this section:

1. AR = number of Areas Reserved

   This quantity is determined from the RESERVE n AREAS clause. If you do not specify a RESERVE n AREAS clause, the default is 1 area for relative files (that is, AR = 1).

2. URS = Unit Record Size (Cell size)

   For fixed-length records, this equals the sum of:

   - 1 (for a delete code controlled by RMS)
   - The record length specified by the Record Description or the RECORD CONTAINS clause, whichever is larger.

   For variable-length records, it equals the maximum record length plus three overhead bytes.

If you do not specify a BLOCK CONTAINS clause, the default bucket size formula is:

Default bucket size = The cell size rounded up to a multiple of 512 bytes

Samples

```
FD  TEST-FILE
    LABEL RECORDS ARE STANDARD.
01  REC-1    PIC   X(500).
```

Bucket size = 512 bytes

Buffer size = 512 * 1 = 512 bytes

```
FD  TEST-FILE
    LABEL RECORDS ARE STANDARD.
01  REC-1    PIC   X(900).
```

Bucket size = 900 bytes
rounded up = 1024 bytes

Buffer size = 1024 * 1 = 1024 bytes

If you specify th BLOCK CONTAINS n CHARACTERS clause, the bucket size formula is:

bucket size = n / 512, rounded up

- If n is less than the unit record size, the compiler issues a warning diagnostic and uses the default method to compute the bucket size.
- The variable n must be a multiple of 512. If not, the compiler issues a warning diagnostic and rounds n up to the next multiple of 512.

Samples

```
FD  TEST-FILE
    BLOCK CONTAINS 2500 CHARACTERS
    LABEL RECORDS ARE STANDARD.
01  REC-1    PIC   X(500).
```

Bucket size = 2500 / 512 = 4.88 blocks
rounded up = 5 blocks

Buffer size = 5 * 512 * 1 = 2560 bytes

```
FD  TEST-FILE
    BLOCK CONTAINS 512 CHARACTERS
    LABEL RECORDS ARE STANDARD.
01  REC-1    PIC   X(494).
```

Bucket size = 512 / 512 = 1 block

Buffer size = 1 * 512 * 1 = 512 bytes

If you specify the BLOCK CONTAINS n RECORDS clause, the bucket size formula is:

bucket size = (n * URS) / 512, rounded up

Samples

```
FD  TEST-FILE
    BLOCK CONTAINS 50 RECORDS
    LABEL RECORDS ARE STANDARD.
01  REC-1    PIC  X(20).
```

Bucket size = ((50 * (1+20))/512 = 2.05
rounded up = 3 blocks

Buffer size = 3 * 512 = 1536 bytes

```
FD  TEST-FILE
    BLOCK CONTAINS 100 RECORDS
    RECORD CONTAINS 25 CHARACTERS
    LABEL RECORDS ARE STANDARD.
01  REC-1    PIC  X(20).
```

Bucket size = ((100 * (1+25))/512 = 5.07
rounded up = 6 blocks

Buffer size = 6 * 1 * 512 = 3072 bytes

```
FD  TEST-FILE
    BLOCK CONTAINS 5 RECORDS
    RECORD VARYING
           FROM 90 TO 99 CHARACTERS
    LABEL RECORDS ARE STANDARD.
01  REC-1  PIC X(90).
01  REC-2  PIC X(99).
```

Bucket size = ((5 * (99+3))/512 = .99
rounded up = 1 block

Buffer size = 1 * 512 = 512 bytes

```
SELECT TEST-FILE
    ASSIGN TO "REPORT.DAT"
    RESERVE 4 AREAS.
      .
      .
      .
FD  TEST-FILE
    BLOCK CONTAINS 50 RECORDS
    LABEL RECORDS ARE STANDARD.
01  REC-1    PIC  X(20)
```

Bucket size = ((50 * (1+20))/512 = 2.05
rounded up = 3 blocks

Buffer size = 4 * 3 = 12 blocks = 6144 bytes

7.8.3 Indexed Files

I/O optimization of an indexed file depends on five items:

1. Records – The size and format of the data records can affect the disk space used by the file.
2. Keys – The number of keys and existence of duplicate key values can affect disk space and processing time.
3. Buckets – Bucket size can affect disk space and processing time. Index depth and file activity can affect bucket size.
4. Index Depth – The depth of the index can affect processing time.
5. File size – The length of files affects space and access time.

7.8.3.1 Records — Variable-length records can save file space. You need to write only the primary record key data item (plus alternate keys, if any) for each record. In contrast, fixed-length records require that all records be equal in length.

For example, assume that you are designing an employee master file. A variable-length record file lets you write a long record for a senior employee with a large amount of historical data, and a short record for a new employee with less historical data.

In the following example of a variable-length record description, integer 10 of the RECORD VARYING clause represents the length of the primary record key, while integer 80 describes the length of the longest record in A-FILE:

```
FILE-CONTROL.
    SELECT A-FILE ASSIGN TO "AMAST"
           ORGANIZATION IS INDEXED.
DATA DIVISION.
FILE SECTION.
FD  A-FILE
    ACCESS MODE IS DYNAMIC
    RECORD KEY IS A-KEY
    RECORD VARYING FROM 10 TO 80 CHARACTERS.
01  A-REC.
    03  A-KEY           PIC X(10).
    03  A-REST-OF-REC   PIC X(70).
        .
        .
        .
```

Buckets must contain enough room for record insertion, or bucket splitting occurs (see the *RMS-11 User's Guide* for more information on bucket splitting). For each record moved, a 7-byte pointer to the new record location remains in the original bucket. Thus, bucket splits can accumulate overhead, possibly reducing usable space so much that the original bucket can no longer receive records.

Record deletions can also accumulate storage overhead. However, most of the space is available for reuse. Because there can be no duplicate primary keys, RMS-11 can reclaim all but 2 bytes of the deleted record space. This 2-byte field is a record deletion flag.

There are several ways to minimize overhead accumulation. First, determine or estimate the frequency of certain operations. For example, if you expect to add or delete 100 records in a 100,000-record file, your database is stable enough to allow some wasted space for record additions and deletions. However, if you expect frequent additions and deletions, try to:

- Choose a bucket size that allows for overhead accumulation, if possible. Avoid bucket sizes that are an exact or near multiple of your record size.
- Optimize record insertion performance by using the RMSDEF utility to define the file with fill numbers; use the APPLY FILL-SIZE clause when loading the file.

7.8.3.2 Alternate Keys — Each alternate key requires the creation and maintenance of a separate index structure. The more keys you define, the longer each WRITE, REWRITE, and DELETE operation takes. (READ operations are not affected by multiple keys.)

If your application requires alternate keys, you can minimize I/O processing time if you avoid duplicate alternate keys. Duplicate keys can create long record pointer arrays, which fill bucket space and increase access time.

7.8.3.3 Bucket Size — Bucket size selection can influence indexed file performance.

To the system, bucket size is an integral number of physical blocks, each 512 bytes long. Thus, a bucket size of one specifies a 512-byte bucket, while a bucket size of two specifies a 1024-byte bucket, and so on.

The COBOL-81 compiler passes bucket size values to RMS-11 based on what you specify in the BLOCK CONTAINS clause. In this case, you express bucket size in terms of records or characters.

If you specify block size in records, the bucket can contain more records than you specify but never fewer. For example, assume that your file contains fixed-length, 100-byte records and you want each bucket to contain five records, as follows:

```
BLOCK CONTAINS 5 RECORDS
```

This appears to define a bucket as a 512-byte block, containing five records of 100 bytes each. However, the compiler adds RMS-11 record and bucket overhead to each bucket, as follows:

Bucket overhead	15 bytes per bucket
Record overhead	7 bytes per record (fixed-length) 9 bytes per record (variable-length)

Thus, in the previous example, the bucket size calculation is:

Bucket overhead	15 bytes
Record size is 100 bytes + 7 bytes Record Overhead for each of 5 records	
Total Record Space is (100 + 7) * 5, or	535 bytes
TOTAL	550 bytes

Because blocks are 512 bytes long, and buckets are always an integral number of blocks, the smallest bucket size possible (the system default) in this case is two blocks. However, the system puts in as many records as fit into each bucket. Thus, the bucket actually contains nine records, not five.

The CHARACTERS option of the BLOCK CONTAINS clause lets you specify bucket size more directly. For example:

```
BLOCK CONTAINS 2048 CHARACTERS
```

This specifies a bucket size of four 512-byte blocks. The number of characters in a bucket is always a multiple of 512. If it is not, the compiler rounds it to the next higher multiple of 512.

To improve I/O access time:

- Use the BLOCK CONTAINS n RECORDS clause to specify the number of records (cells) in each bucket
- Specify a small bucket size for random access applications
- Specify a large bucket size for sequential access applications

7.8.3.4 Index Depth — The length of data records, key fields, and buckets in the file determines the depth of the index. Index depth, in turn, determines the number of disk accesses needed to retrieve a record. The smaller the index depth, the better the performance. In general, an index depth of three or four gives satisfactory performance. If your calculated index depth is greater than four, you should consider redesigning the file.

You can optimize your file's index depth after you have determined file, record, and key size. Calculating index depth is an iterative process, with bucket size as the variable. Keep in mind that the highest level (root level) can contain only one bucket. For an example of index depth calculation, see Section 7.8.3.5.

7.8.3.5 File Size — When you calculate file size:

- Every bucket in an indexed file contains 15 bytes of overhead.
- Every bucket in an indexed file contains records. Only record type and size differ.
- Data records are only in level 0 buckets of the primary index.
- Index records are in level 1 and higher numbered buckets.
- If you use alternate keys, SIDRs (Secondary Index Data Records) are only in level 0 buckets of alternate indexes.

Use these calculations to determine data and index record size:

1. Data records:

```
Fixed-length record size = actual record size + 7
Variable-length record size = actual record size + 9
```

2. Index records:

```
Record size = key size + 3
```

If a file has more than 65,536 blocks, the 3-byte index record overhead could increase to 5 bytes.

Use these calculations to determine SIDR record length:

1. No duplicates allowed:

```
Record size = key size + 9
```

2. Duplicates allowed:

```
Record size = key size + 8 + 5 * (no. of duplicate records)
```

Note

Bucket packing efficiency determines how well bucket space is used. A packing efficiency of 1 means the buckets of an index are full. A packing efficiency of .5 means that the buckets, on the average, are half full.

Consider an indexed file with these attributes:

- 100,000 fixed-length records of 200 characters each
- Primary key = 20 characters
- Alternate key = 8 characters, no duplicates allowed
- Bucket size = 3 (an arbitrary value)
- No fill number

Primary key index level calculations:

Level 0 (data level buckets):

$$\text{data records per bucket} = \frac{\text{bytes per bucket} - 15}{\text{rec. size} + 7}$$

$$= \frac{3 * 512 - 15}{200 + 7} = 7$$

$$\text{number of data buckets} = \frac{\text{number of data records}}{\text{recs. per bucket}}$$

$$= \frac{100{,}000}{7} = 14{,}286$$

Level 1 (index buckets):

$$\text{index records per bucket} = \frac{\text{bytes per bucket} - 15}{\text{key size} + 3}$$

$$= \frac{3 * 512 - 15}{20 + 3} = 66$$

$$\text{number of index buckets} = \frac{\text{no. of buckets from level n–1}}{\text{index recs. per bucket}}$$

$$= \frac{14{,}286}{66} = \text{216 level 1 buckets to address all data buckets at level 0}$$

Continue calculating index depth until you reach the root level; that is, when the number of buckets needed to address the buckets from the previous level equals one.

Level 2 (index buckets):

$$\text{number of buckets} = \frac{216}{66} = \text{4 level 2 buckets to address all level 1 buckets}$$

Level 3 (index buckets):

$$\text{number of buckets} = \frac{4}{66} = 1 \quad \begin{array}{l}\text{level 3 bucket to address all level 2 buckets}\\ \text{(level 3 is the root bucket for the primary index)}\end{array}$$

7.8.3.6 Alternate Key Index Level Calculations — If you allow duplicate keys in alternate indexes, the number and size of SIDRs depend on the number of duplicate key values in the file. (For duplicate key alternate index calculations, see the *RMS-11 User's Guide*.) Since alternate index records are usually inserted in random order, the bucket packing efficiency ranges from about .5 to about .65. The following example uses an average efficiency of .55.

Data level buckets (no duplicate alternate keys):

$$\text{SIDRs per bucket} = \frac{\text{bytes per bucket} - 15}{\text{key size} + 9} = \frac{1536 - 15}{8 + 9} = 89$$

$$\text{number of buckets} = \frac{\text{number of records}}{\text{recs. per bucket}} = \frac{100{,}000}{89} = 1123 \quad \text{level 0 alternate index buckets}$$

Level 1 (index buckets):

$$\text{records per bucket} = \frac{1536 - 15}{8 + 3} = 138$$

$$\text{number of buckets} = \frac{1123}{138} = 9 \quad \text{level 1 buckets to address data buckets (SIDRs) at level 0}$$

Level 2 (index buckets):

$$\text{number of buckets} = \frac{9}{138} = 1 \quad \begin{array}{l}\text{level 2 bucket to address data buckets at level 1}\\ \text{(Level 2 is the root level)}\end{array}$$

7.8.3.7 Caching Index Roots — The system requires at least two buffers to process an indexed file: one for a data bucket, the other for an index bucket. Each buffer is large enough to contain a single bucket. If your program does not contain a RESERVE n AREAS clause, your system sets the default.

A RESERVE n AREAS clause creates additional buffers for processing an indexed file. At run time, the system retains (caches) in memory the roots of one or more indexes of the file. Random access to any record through that index requires one less I/O operation.

The following rules apply for caching index roots:

- Allocate one buffer for each key that your program uses to access file records, in addition to the two required buffers. For example, if the file contains a primary key and two alternate keys, and you use all of these keys to access records, allocate a total of five buffers. If you use only one key, you need only one additional buffer area, for a total of three.
- Use the RESERVE n AREAS clause to obtain allocation, where n is two more than the number of distinct keys used for access. For example, the RESERVE 5 AREAS clause allocates two required buffers, plus three buffer areas for caching the roots of three distinct file access keys.

7.8.3.8 Bucket Size and Buffer Size Calculations for Indexed Files — In its calculations for indexed files, COBOL-81 adds 15 bytes to each bucket and 7 bytes to each record, for RMS-11 overhead. COBOL-81 then determines the file's buffer size (in blocks) using the bucket size:

```
buffer size (in blocks) = (bucket size) * AR
```

The unit of transfer for an indexed file is equal to the bucket size.

COBOL-81 determines the file's bucket size (in 512-byte blocks) using two different formulas, depending on which format of the BLOCK CONTAINS clause you use. The following items refer to the formulas used in this section:

1. AR = number of Areas Reserved

   This quantity is determined from the RESERVE n AREAS clause. If you do not specify a RESERVE n AREAS clause, the default is 2 areas for indexed files (that is, AR = 2).

2. URS = Unit Record Size

   For fixed-length records, this equals the record length specified by the record description or the RECORD CONTAINS clause, whichever is larger.

   For variable-length records, it equals the maximum record length plus 2 overhead bytes.

If you do not specify the BLOCK CONTAINS clause, the default bucket size formula is:

bucket size = (15 + (7 + URS)) / 512, rounded up

Sample

```
FD  TEST-FILE
    RECORD IS VARYING FROM 100
              TO 511 CHARACTERS
    LABEL RECORDS ARE STANDARD.
01  REC-1       PIC X(100).
01  REC-2       PIC X(511).
```

Bucket size = 2 rounded up = ((15+(7+(511+2))))/512

Buffer size = 2 * 2 = 4 blocks = 2048 bytes

If you specify the BLOCK CONTAINS n CHARACTERS clause, the bucket size formula is:

bucket size = n / 512

- The variable n must be a multiple of 512. If not, the compiler issues a warning diagnostic and rounds n up to the next multiple of 512.
- The variable n must be equal to or greater than (15 + (7 + URS)). If it is less, the compiler issues a warning diagnostic and uses the default method to compute bucket size.

Sample

```
SELECT IDX-FILE
    ASSIGN TO "ADDRES.DAT"
    ORGANIZATION IS INDEXED
         .
         .
         .
FD IDX-FILE
    BLOCK CONTAINS 512 CHARACTERS
    LABEL RECORDS STANDARD.
01  REC-1       PIC X(100).
```

Bucket size = 512 / 512 = 1 block

Buffer size = 1 * 2 = 2 blocks = 1024 bytes

If you specify the BLOCK CONTAINS n RECORDS clause, the bucket size formula is:

bucket size = [15 + (n * (7 + URS))] / 512

Bucket size is in 512-byte blocks and is rounded up to the next integer.

Samples

```
SELECT IDX-FILE
    ASSIGN TO "RFCRA"
    ORGANIZATION IS INDEXED
        .
        .
        .
```

Bucket size = [15 + (2 * (7 + 242))]/512 = 513/512

rounded up = 2 blocks

```
FD  IDX-FILE
    BLOCK CONTAINS 2 RECORDS
    RECORD CONTAINS 242 CHARACTERS
        .
        .
        .
```

Buffer size = 2 * 2 = 4 blocks = 2048 bytes

Note

The bucket in this example is rounded to 2 blocks or 1024 bytes. It actually contains 4 records, not 2.

```
SELECT IDX-FILE
    ASSIGN TO "RFCRA"
    ORGANIZATION IS INDEXED
    RESERVE 3 AREAS.
        .
        .
        .
```

Bucket size = [15 + (1 * (7 + 500))]/512 = 522/512

rounded up = 2 blocks

```
FD  IDX-FILE
    BLOCK CONTAINS 1 RECORDS
    RECORD CONTAINS 500 CHARACTERS
        .
        .
        .
```

Buffer size = 2 * 3 = 6 blocks = 3072 bytes

Even though the bucket size here is 2 blocks or 1024 bytes, it can contain only 1 record. The addition of another record would require 507 more bytes but this bucket has only 502 free bytes (1024 – 522 = 502). Therefore, the code wastes 502 bytes in each bucket.

Chapter 8
Producing Printed Reports with COBOL-81

This chapter introduces you to one of the most valuable and most looked-at products produced by your COBOL-81 program: the report. It discusses the report's design, components, and modes of printing. It also discusses the two methods of producing your COBOL-81 report: programming the conventional report and programming the linage-file report. The chapter also discusses the accumulation and reporting of control totals and gives practical solutions to everyday problems.

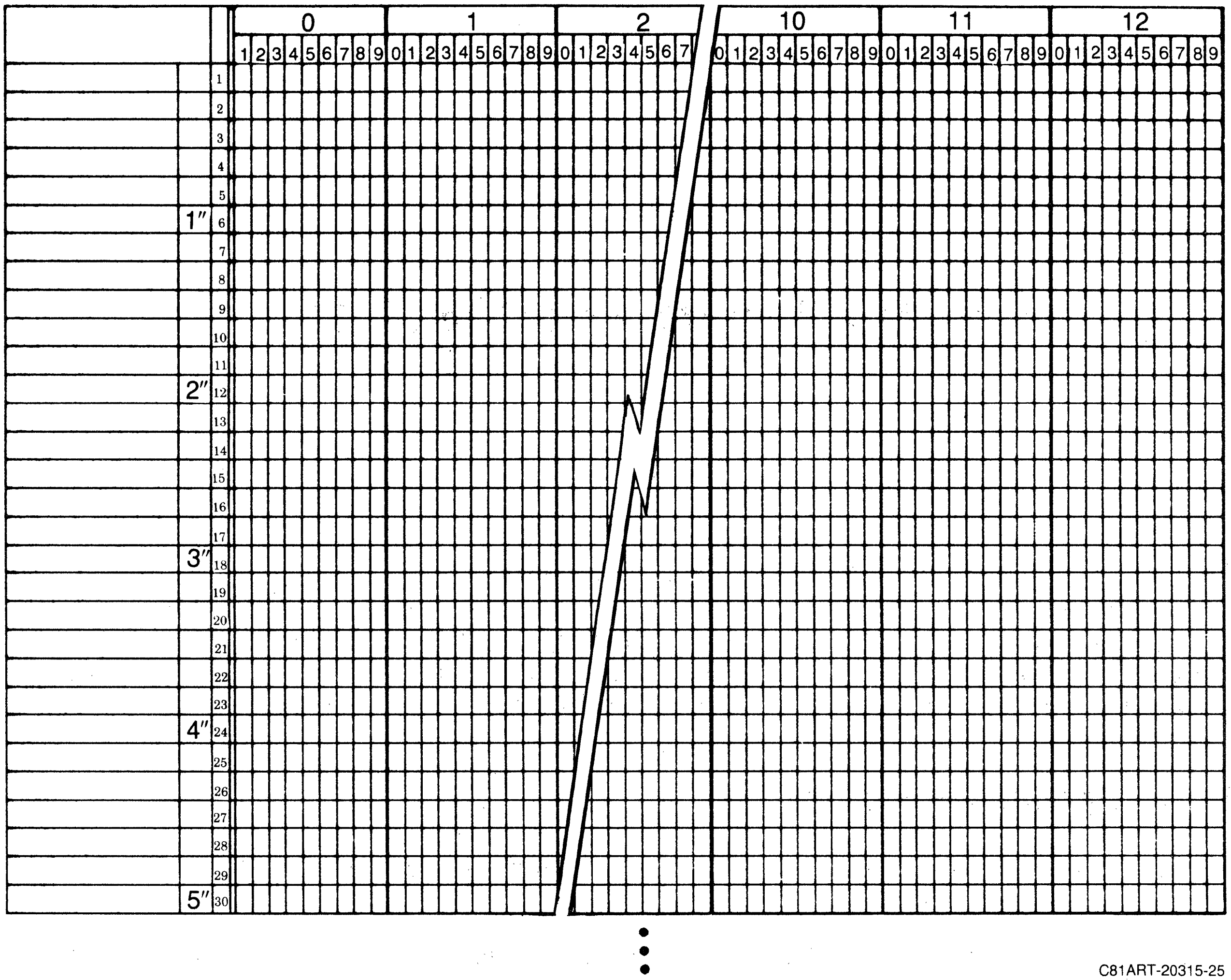
8.1 Designing the Report

A report is closely tied to the information in a file. It is limited by the data in the file, and, in turn, the data in the file is often dictated by the requirements of a report.

Since the needs of a report are dictated by the data in the file, the design of the report should come early in the design of the application system. In fact, the reporting requirements generally dictate the need for the system. Your job is to determine the data needed in the file from the information needed in the report. You should provide your customers with sample reports or layouts as soon as possible so that they can make any necessary changes early in the design.

The design of a report usually begins with a rough description of the data to be reported. Given a general idea of what a report is to contain, you must fill in the details, including page headings, rows and columns, column sizes, and so on. The usual tool for laying out the report is the report layout worksheet shown in Figure 8-1. This form is marked off with 132 characters on a line and 60 lines on a page. When you complete the report layout worksheet, you can use it as program documentation. The report layout worksheet is helpful in actually formatting the report for programming, if there is a need to position each line and column.

Figure 8-1: A Sample Report Layout Worksheet

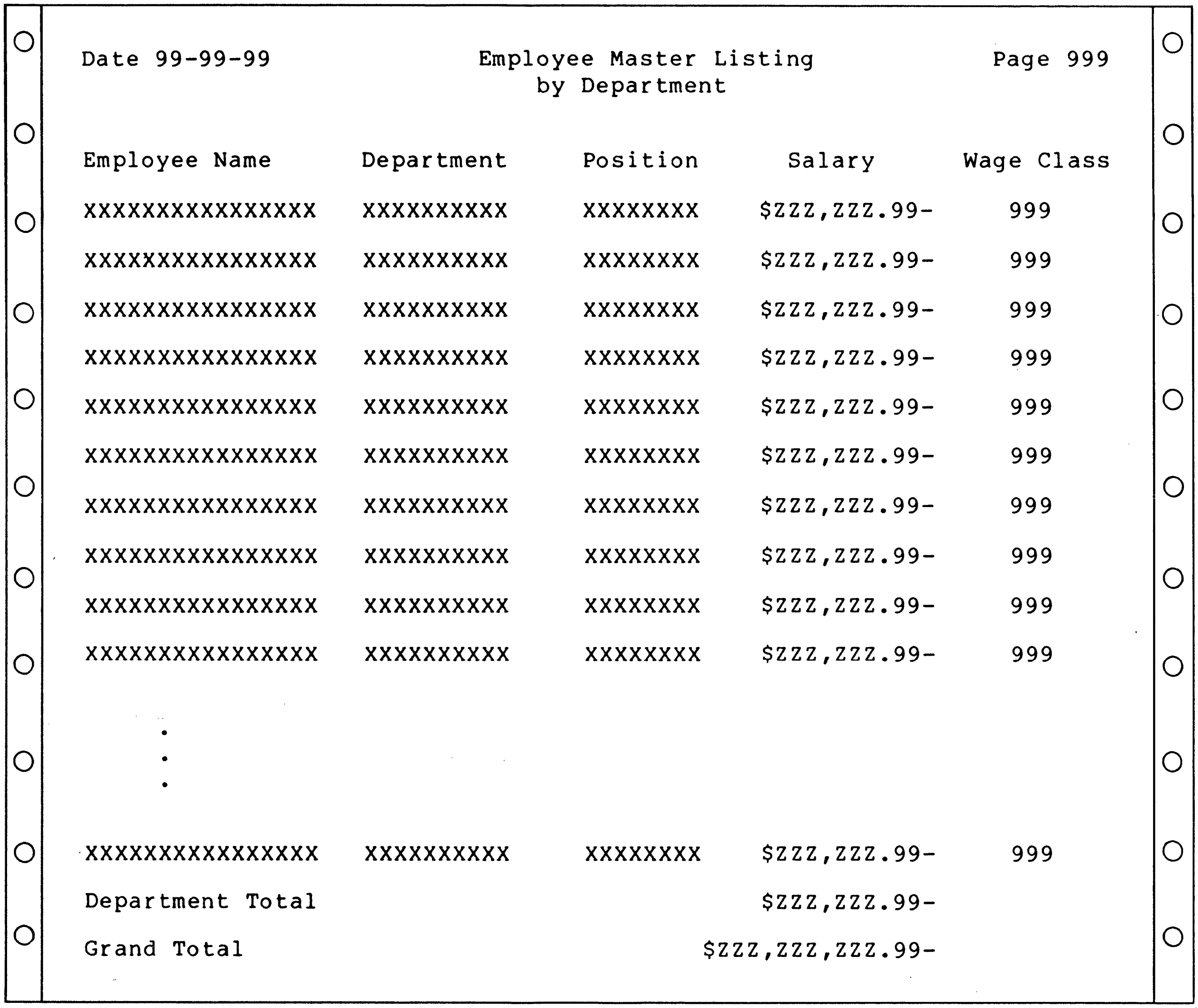


C81ART-20315-25

However, in practice the report layout worksheet is an acceptable design tool but it is difficult to change. As soon as you show the report layout worksheet to your users, they will change it. After a few iterations of this, you will be happy never to see a report layout worksheet again.

Another way to lay out a report is to create it in a file using an editor and a video terminal. Users with terminals can then edit the report, make changes, and move text around easily. Users who do not have a terminal can simply print this file. Figure 8-2 is such an example.

Figure 8-2: Typical Report Layout



```
Date 99-99-99              Employee Master Listing            Page 999
                               by Department

Employee Name        Department      Position        Salary      Wage Class

XXXXXXXXXXXXXXXX     XXXXXXXXXX      XXXXXXXX    $ZZZ,ZZZ.99-       999
XXXXXXXXXXXXXXXX     XXXXXXXXXX      XXXXXXXX    $ZZZ,ZZZ.99-       999
XXXXXXXXXXXXXXXX     XXXXXXXXXX      XXXXXXXX    $ZZZ,ZZZ.99-       999
XXXXXXXXXXXXXXXX     XXXXXXXXXX      XXXXXXXX    $ZZZ,ZZZ.99-       999
XXXXXXXXXXXXXXXX     XXXXXXXXXX      XXXXXXXX    $ZZZ,ZZZ.99-       999
XXXXXXXXXXXXXXXX     XXXXXXXXXX      XXXXXXXX    $ZZZ,ZZZ.99-       999
XXXXXXXXXXXXXXXX     XXXXXXXXXX      XXXXXXXX    $ZZZ,ZZZ.99-       999
XXXXXXXXXXXXXXXX     XXXXXXXXXX      XXXXXXXX    $ZZZ,ZZZ.99-       999
XXXXXXXXXXXXXXXX     XXXXXXXXXX      XXXXXXXX    $ZZZ,ZZZ.99-       999
XXXXXXXXXXXXXXXX     XXXXXXXXXX      XXXXXXXX    $ZZZ,ZZZ.99-       999

      .
      .
      .

XXXXXXXXXXXXXXXX     XXXXXXXXXX      XXXXXXXX    $ZZZ,ZZZ.99-       999
Department Total                                 $ZZZ,ZZZ.99-
Grand Total                                  $ZZZ,ZZZ,ZZZ.99-
```

C81ART-20320-25

This layout sheet is easy to duplicate and distribute in memos. It is also easy to change, using the cut and paste technique. The report does not show the exact column positions, but these are not needed. In the design stage, the customer should be made to focus on the information contained in the report and the general format.

8.2 Components of a Report

The seven components of a report, as shown in Figure 8-3, include:

1. Report Heading – The report heading consists of information printed before the main body of a report. It can be printed on a separate page by itself or it can be printed as the first page heading, with the remaining page headings abbreviated to save paper. The report heading can contain such information as handling and distribution instructions. It might also contain the selection criteria, the sort order, and other assumptions that went into the creation of the report.

2. Page Heading – The page heading consists of information printed on one or more lines of every page in the report. It usually names and dates the report, identifies the number of the page in the report, and titles each column of information in the detail line.

3. Control Heading – The control heading consists of one or more lines of information identifying the beginning of a new logical area on a page.

4. Detail Line(s) – The detail consists of one or more lines of the primary data of the report.

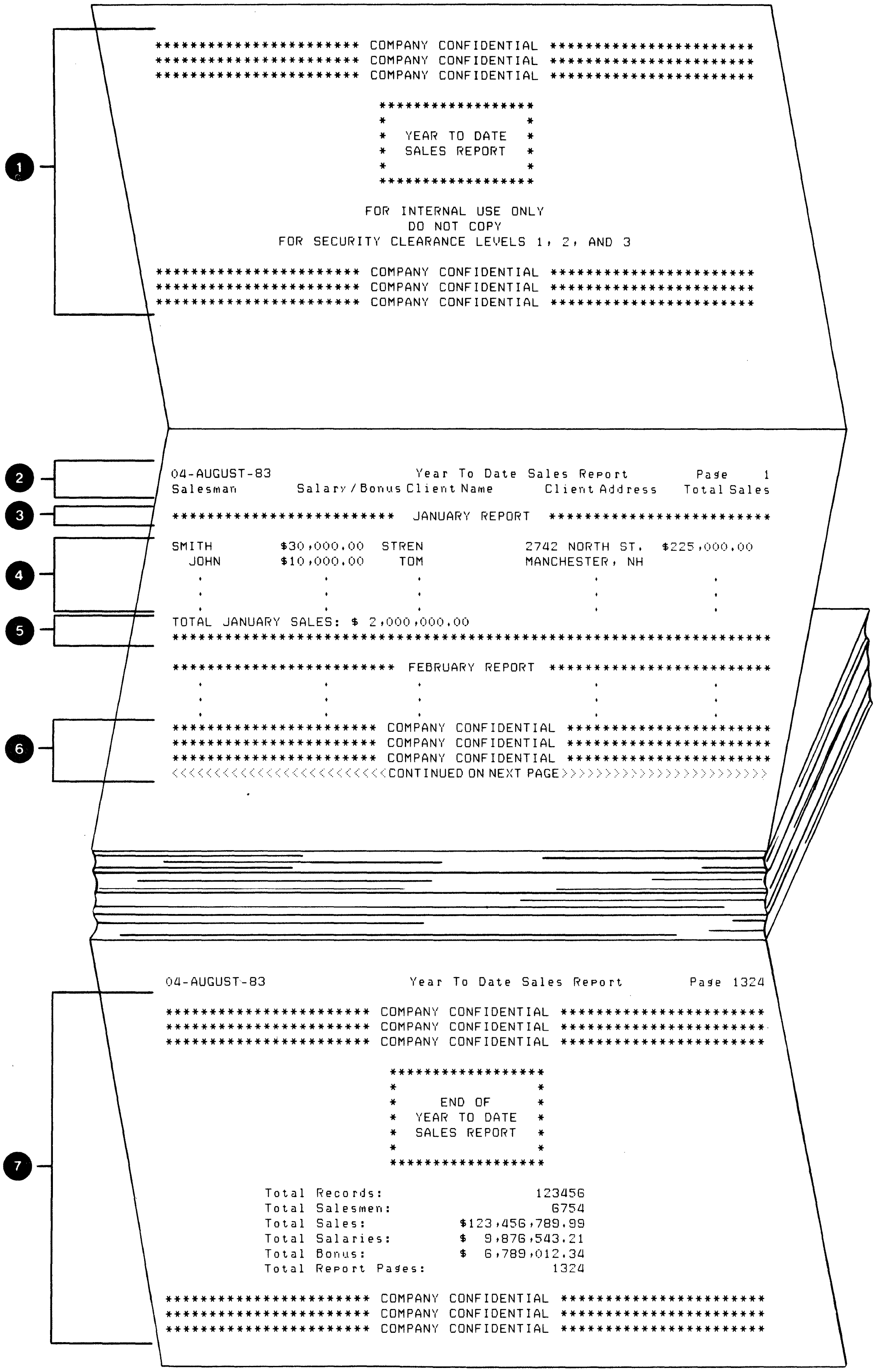
5. Control Footing – The control footing consists of one or more lines of information identifying the end of a logical area. The control footing can contain one or more totals and an accompanying message.

6. Page Footing – The page footing consists of one or more lines of information at the bottom of each page.

7. Report Footing – The report footing consists of information printed after the main body of the report. It can be continued on the same page of the report body or it can be a separate page. It might contain information such as hash or control totals. A report footing is also a convenient place to print run-time statistics, such as the number of records read and written for each file. It can further provide warning messages, such as noting when a table is close to overflowing.

   All reports should have something to indicate the end of the report, such as an "END OF REPORT" message, so that you can tell at a glance that you have all of the pages. (The consecutive page numbers tell if a page is missing, but they do not indicate which page is the last.)

Figure 8-3: Components of a Report



```
************************ COMPANY CONFIDENTIAL ************************
************************ COMPANY CONFIDENTIAL ************************
************************ COMPANY CONFIDENTIAL ************************

                         ******************
                         *                *
                         *  YEAR TO DATE  *
                         *  SALES REPORT  *
                         *                *
                         ******************

                        FOR INTERNAL USE ONLY
                             DO NOT COPY
               FOR SECURITY CLEARANCE LEVELS 1, 2, AND 3

************************ COMPANY CONFIDENTIAL ************************
************************ COMPANY CONFIDENTIAL ************************
************************ COMPANY CONFIDENTIAL ************************
```

```
04-AUGUST-83                   Year To Date Sales Report           Page     1
Salesman       Salary/Bonus Client Name      Client Address   Total Sales
************************* JANUARY REPORT *************************

SMITH         $30,000.00  STREN              2742 NORTH ST.  $225,000.00
  JOHN        $10,000.00   TOM               MANCHESTER, NH
    .             .         .                     .              .
    .             .         .                     .              .
    .             .         .                     .              .
TOTAL JANUARY SALES: $ 2,000,000.00
*******************************************************************

************************* FEBRUARY REPORT *************************
    .             .         .                     .              .
    .             .         .                     .              .
    .             .         .                     .              .
************************ COMPANY CONFIDENTIAL ************************
************************ COMPANY CONFIDENTIAL ************************
************************ COMPANY CONFIDENTIAL ************************
<<<<<<<<<<<<<<<<<<<<<<<<<<CONTINUED ON NEXT PAGE>>>>>>>>>>>>>>>>>>>>>>>>>>
```

```
04-AUGUST-83                   Year To Date Sales Report           Page 1324

************************ COMPANY CONFIDENTIAL ************************
************************ COMPANY CONFIDENTIAL ************************
************************ COMPANY CONFIDENTIAL ************************

                         ******************
                         *                *
                         *     END OF     *
                         *  YEAR TO DATE  *
                         *  SALES REPORT  *
                         *                *
                         ******************

            Total Records:                  123456
            Total Salesmen:                   6754
            Total Sales:             $123,456,789.99
            Total Salaries:          $   9,876,543.21
            Total Bonus:             $   6,789,012.34
            Total Report Pages:               1324

************************ COMPANY CONFIDENTIAL ************************
************************ COMPANY CONFIDENTIAL ************************
************************ COMPANY CONFIDENTIAL ************************
```

C81ART-20330-100

8.3 The Logical Page and the Physical Page

A physical page is the actual piece of paper printed by your printer.

A logical page is conceptual, consisting of a page body (required), top margin (optional), footing (optional), and bottom margin (optional). Figure 8-5 in Section 8.6.1 and Figure 8-8 in Section 8.7.1 illustrate the logical page structure for the conventional-file report and linage-file report, respectively.

You define the number of lines on the logical page based on the number of lines on the target physical page. The number of lines determines the size of the logical page. You must then choose those lines within the logical page that are to be page headers, control headers, detail lines, control footings, and page footings. Having defined the framework of the logical page, your program must stay within those bounds; otherwise, when you print your report you will get unpredictable results.

The report your program creates consists of one or more logical pages. Each page consists of one or more records. Each record contains two types of data:

1. The actual characters on the physical (printed) page
2. The special characters used by the system spooler and printer that control line and page advancement (these characters never print on the physical page).

Each logical page follows the preceding logical page with no spacing between them.

8.3.1 Horizontal Spacing for the Logical Page

Your program must provide all horizontal spacing for your logical page. You do this by defining every report item on your report layout worksheet (See Section 8.1) and every space character between each report item.

8.3.2 Vertical Spacing for the Logical Page

COBOL-81 lets you skip one or more lines either before or after your program writes a line of the report. The compiler automatically provides the necessary line-feed characters for you. For example, to print a line before advancing 5 lines, you would use a statement like this:

```
WRITE... BEFORE ADVANCING 5 LINES.
```

To print a line after advancing 2 lines, you would use this statement:

```
WRITE... AFTER ADVANCING 2 LINES.
```

8.4 Modes for Printing Reports

A program can either spool a report to a mass storage device for later printing or it can allocate the printer directly and immediately produce the report. The next sections explain how to print a report on line or from the system spooler.

8.4.1 Online Printing

To directly allocate a printer, you must specify the printer's device name in the file specification for the report file. Example 8-1 shows how you allocate the system's line-printer at run time.

Example 8-1: Directly Allocating a Printer for Immediate Use

```
SELECT REPORT-FILE ASSIGN TO "LP:".
```

The advantages of directly allocating the printer are:

1. Results are immediate.
2. It is an acceptable method of associating a number from the preprinted form (as in the case of payroll checks) to a record in a file. As the operator opens each box of forms and mounts them (or remounts them if a paper jam occurs) in the printer, your program can request and ACCEPT the starting number from each new box of forms. If the program then outputs a record for each printed form (and includes the form number in the record) you establish an immediate audit trail.

The disadvantages of directly allocating the printer are:

1. You must either wait until all printer requests from the system spooler are complete, or you must change job priorities.
2. You tie up the printer for as long as your job runs. If your program is heavily compute-bound and runs for a long time, you could significantly reduce your installation's pages-printed-per-day production schedule.
3. You do not have a backup report file in the event of power failure or other unforeseeable accident. Therefore, you must start the job over again from the beginning.
4. You cannot use the system network.

8.4.2 Spooling to a Mass Storage Device

To spool your report to a mass storage device (such as a disk or magnetic tape) for later printing, you must provide a file specification. Example 8-2 shows you how to spool the "JAN28P.DAT" report to disk pack "DB1:".

Example 8-2: Spooling a Report to a Disk Pack for Later Printing

```
SELECT REPORT-FILE ASSIGN TO "DB1:JAN28P".
```

The advantages of spooling to a mass storage device are:

1. You can run your job at any time regardless of other printer activity and regardless of printer status and current mounted form.
2. Your application program does not make immediate resource demands on the target printer.

3. You can schedule the printing at your leisure and print the file according to your priority scheme.
4. You get optimum use of the printer. This method results in printing maximum lines/minute.
5. You have a backup of the file.
6. You can use the system network.

The disadvantages of this method are:

1. You do not see immediate results.
2. You would find it difficult and expensive to input the preprinted form numbers (for example, check numbers) from your forms into your report file.

8.5 Accumulating and Reporting Totals

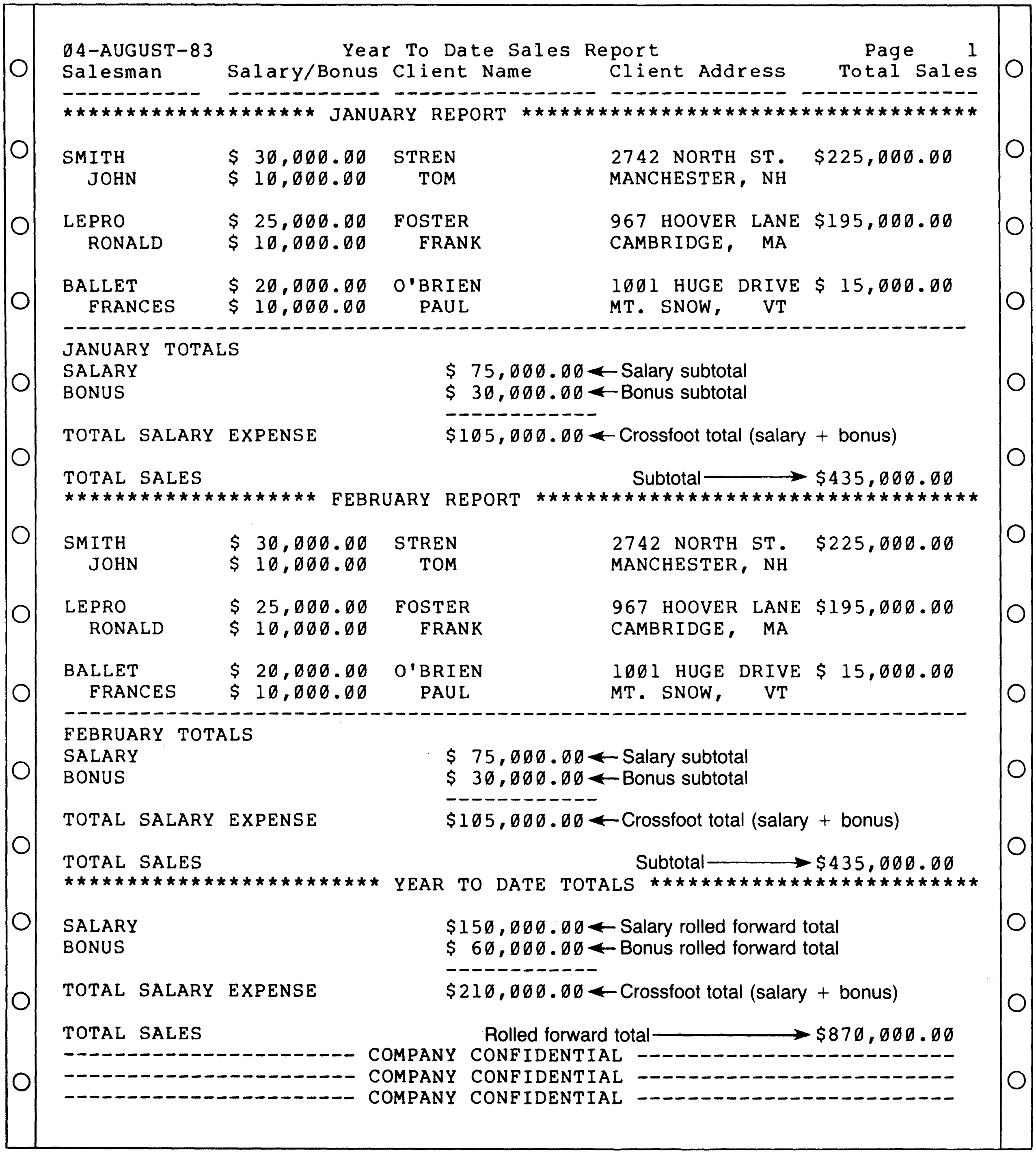
Your program can report three catagories of totals in the control footing and report footing locations of your report:

1. Subtotals – Subtotaling is the process of summing a detail item from each detail line. In Figure 8-4, "Salary," "Bonus," and "Total Sales" are subtotaled. For example, to get the first salary subtotal for January on page 1 ($75,000.00), the program must add each salesman's salary ($30,000 + $25,000 + $20,000). After printing the salary total the program must zero the total to begin subtotaling for the February report, and so forth.

2. Crossfoot Totals – Crossfooting is the process of summing subtotals from a common group of totals. In Figure 8-4, "TOTAL SALARY EXPENSE" is crossfooted by adding "TOTAL SALARY" and "TOTAL BONUS." For example, to get the first "TOTAL SALARY EXPENSE" crossfoot total for the January report, the program must add the salary subtotal and the bonus subtotal before the program clears the subtotals.

3. Rolled Forward Totals – Rolling forward is the process of summing either subtotals or crossfoot totals. In Figure 8-4, the "YEAR TO DATE TOTALS" at the bottom of page 1 are rolled forward from both "JANUARY" and "FEBRUARY" totals. The program computes the salary and bonus "YEAR TO DATE TOTALS" from the previous salary and bonus subtotals. It computes the total salary expense figure from the previous total salary expense crossfoot totals.

Figure 8-4: Subtotals, Crossfoot Totals, and Rolled Forward Totals



```
Ø4-AUGUST-83          Year To Date Sales Report              Page    1
Salesman     Salary/Bonus Client Name     Client Address    Total Sales
----------- ------------ ---------------- ------------- --------------
******************** JANUARY REPORT ***********************************

SMITH         $ 3Ø,ØØØ.ØØ  STREN            2742 NORTH ST.  $225,ØØØ.ØØ
  JOHN        $ 1Ø,ØØØ.ØØ    TOM            MANCHESTER, NH

LEPRO         $ 25,ØØØ.ØØ  FOSTER           967 HOOVER LANE $195,ØØØ.ØØ
  RONALD      $ 1Ø,ØØØ.ØØ    FRANK          CAMBRIDGE,  MA

BALLET        $ 2Ø,ØØØ.ØØ  O'BRIEN          1ØØ1 HUGE DRIVE $ 15,ØØØ.ØØ
  FRANCES     $ 1Ø,ØØØ.ØØ    PAUL           MT. SNOW,    VT
---------------------------------------------------------------------
JANUARY TOTALS
SALARY                         $ 75,ØØØ.ØØ←Salary subtotal
BONUS                          $ 3Ø,ØØØ.ØØ←Bonus subtotal
                               ------------
TOTAL SALARY EXPENSE           $1Ø5,ØØØ.ØØ←Crossfoot total (salary + bonus)

TOTAL SALES                                  Subtotal————→ $435,ØØØ.ØØ
******************** FEBRUARY REPORT **********************************

SMITH         $ 3Ø,ØØØ.ØØ  STREN            2742 NORTH ST.  $225,ØØØ.ØØ
  JOHN        $ 1Ø,ØØØ.ØØ    TOM            MANCHESTER, NH

LEPRO         $ 25,ØØØ.ØØ  FOSTER           967 HOOVER LANE $195,ØØØ.ØØ
  RONALD      $ 1Ø,ØØØ.ØØ    FRANK          CAMBRIDGE,  MA

BALLET        $ 2Ø,ØØØ.ØØ  O'BRIEN          1ØØ1 HUGE DRIVE $ 15,ØØØ.ØØ
  FRANCES     $ 1Ø,ØØØ.ØØ    PAUL           MT. SNOW,    VT
---------------------------------------------------------------------
FEBRUARY TOTALS
SALARY                         $ 75,ØØØ.ØØ←Salary subtotal
BONUS                          $ 3Ø,ØØØ.ØØ←Bonus subtotal
                               ------------
TOTAL SALARY EXPENSE           $1Ø5,ØØØ.ØØ←Crossfoot total (salary + bonus)

TOTAL SALES                                  Subtotal————→$435,ØØØ.ØØ
************************* YEAR TO DATE TOTALS **************************

SALARY                         $15Ø,ØØØ.ØØ←Salary rolled forward total
BONUS                          $ 6Ø,ØØØ.ØØ←Bonus rolled forward total
                               ------------
TOTAL SALARY EXPENSE           $21Ø,ØØØ.ØØ←Crossfoot total (salary + bonus)

TOTAL SALES                         Rolled forward total————→$87Ø,ØØØ.ØØ
----------------------- COMPANY CONFIDENTIAL -------------------------
----------------------- COMPANY CONFIDENTIAL -------------------------
----------------------- COMPANY CONFIDENTIAL -------------------------
```

C81ART-20340-60

8.6 Programming the Conventional COBOL-81 Report

Conventional-file reports:

- Have sequential organization and access mode
- Contain variable-length records
- Consist of one or more logical pages
- Contain compiler-generated form-feed and line-feed characters that control the vertical position of the physical page in the line printer

The next sections discuss:

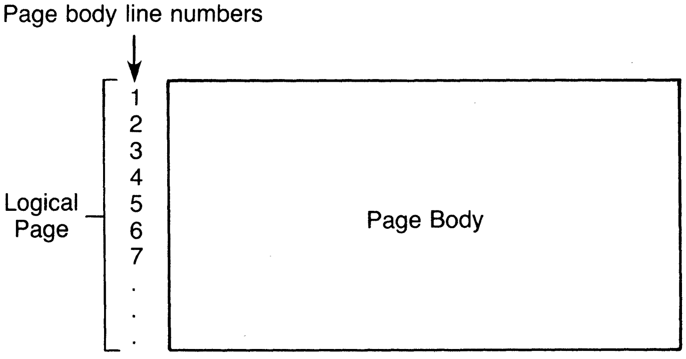
1. Defining the logical page
2. Advancing to the next logical page
3. Programming for the page-overflow condition
4. Using a line counter
5. Programming for a 20-line logical page, a special forms example

8.6.1 Defining the Logical Page

After you have defined your logical page (see Section 8.3), then you must include routines that keep track of how many lines your program writes on the page so that it can handle the page-overflow condition and advance to the next logical page. The next two sections discuss these two subjects in more detail.

Figure 8-5 shows the logical page area for a conventional report. The conventional report logical page area can consist of the page areas discussed in Section 8.2.

Figure 8-5: Logical Page Area for a Conventional Report



C81ART-20350-25

8.6.2 How to Advance to the Next Logical Page

COBOL-81 lets your program control automatic logical page advancement with the WRITE statement. To advance to the next logical page and position the printer on the first print line, your program must specify the AFTER/BEFORE ADVANCING PAGE clause. The next two sections discuss the routines you should include in your report programs and presents a sample program that contains them.

8.6.3 Programming for the Page-Overflow Condition

If your program writes more lines than the logical page can accomodate, a page-overflow condition exists. This is a normal condition. It lets your program know when it should execute its top-of-page routines. These routines should contain a WRITE...AFTER/BEFORE ADVANCING PAGE statement. This statement positions the line printer at the top of the next logical page.

You must include routines in your program to determine when a report's logical page is full and when the program prints the last line on the logical page (in the case where you do not use all lines on the page). Example 8-3 shows two methods to check for this condition:

1. Paragraph A100-FIRST-REPORT-ROUTINES checks for a full page after it writes a report line. If the page-overflow condition exists, A901-HEADER-ROUTINE executes.
2. Paragraph A500-SECOND-REPORT-ROUTINES checks if more than 50 lines exist on the current logical page. If more than 50 lines exist, A902-HEADER-ROUTINE executes.

In either case, the AFTER ADVANCING PAGE clause in the A901-HEADER-ROUTINE and A902-HEADER-ROUTINE paragraphs generate the needed form-feed characters for the printer to position itself at the top of the next physical page.

Example 8-3: Checking for the Page-Overflow Condition

```
PROCEDURE DIVISION.
A000-BEGIN.
     .
     .
     .
A100-FIRST-REPORT-ROUTINES.
*
* A901-HEADER-ROUTINE executes whenever the number of lines written exceed
* the number of lines on the 66-line default logical page
*
    WRITE A-LINE1 AFTER ADVANCING 2 LINES.
    ADD 2 TO REPORT1-LINE-COUNT.
    IF REPORT1-LINE-COUNT > 65 PERFORM A901-HEADER-ROUTINE.
     .
     .
     .
A500-SECOND-REPORT-ROUTINES.
*
* This routine uses only the first 50 lines of the 66 line report.
*
    WRITE A-LINE2 AFTER ADVANCING 2 LINES.
    ADD 2 TO REPORT2-LINE-COUNT.
    IF REPORT2-LINE-COUNT IS GREATER THAN 50
                               PERFORM A902-HEADER-ROUTINE.
     .
     .
     .
A901-HEADER-ROUTINE.
    WRITE A-LINE1 FROM REPORT1-HEADER-LINE-1 AFTER ADVANCING PAGE.
    MOVE 0 TO REPORT1-LINE-COUNT.
    ADD 1 TO REPORT1-LINE-COUNT.
     .
     .
     .
A902-HEADER-ROUTINE.
    WRITE A-LINE2 FROM REPORT2-HEADER-LINE-1 AFTER ADVANCING PAGE.
    MOVE 0 TO REPORT2-LINE-COUNT.
    ADD 1 TO REPORT2-LINE-COUNT.
     .
     .
     .
```

8.6.4 Using a Line Counter

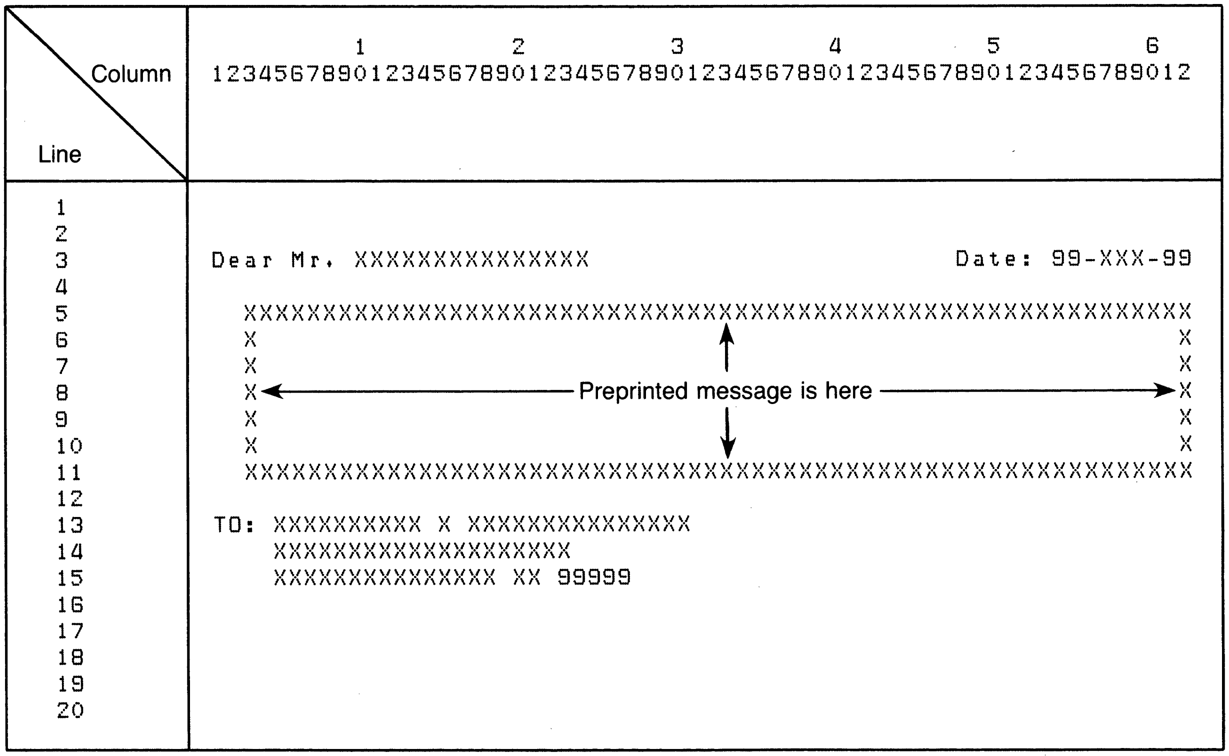
One method of keeping track of how many lines your program writes on a logical page is to use a line counter. Define this counter in the Working-Storage Section, and reset it every time the program begins a new logical page. Add 1 to this counter for each line the program writes and for each line it skips. Then, before the program writes a new line, have it check the line counter value to see if the current logical page can accept the new line. If the line counter value equals or exceeds the maximum number of lines for the logical page, have the program execute its header routines to position the printer to the top of the next logical page.

8.6.5 A Special Forms Example

Example 8-4 writes two reports from the same input file. The first report, in Figure 8-6, is a statement report with a logical page length of 20 lines and a width of 80 characters. However, it only uses the first 15 lines on the page. This report is a preprinted form letter to be inserted in a business envelope. The customer's name and address on lines 13, 14, and 15 are to appear through the envelope's glassine window. The only information the program must supply is the date for line 3, the customer's name for lines 3 and 13, and the customer's address for lines 14 and 15.

The second report is a double-spaced master listing of all input records (see Figure 8-7). Its logical page is identical to the system default logical page (in this case, 66 vertical lines and 132 horizontal characters). However, the program uses only the first 55 lines on the page. Both reports are output to a disk for later printing.

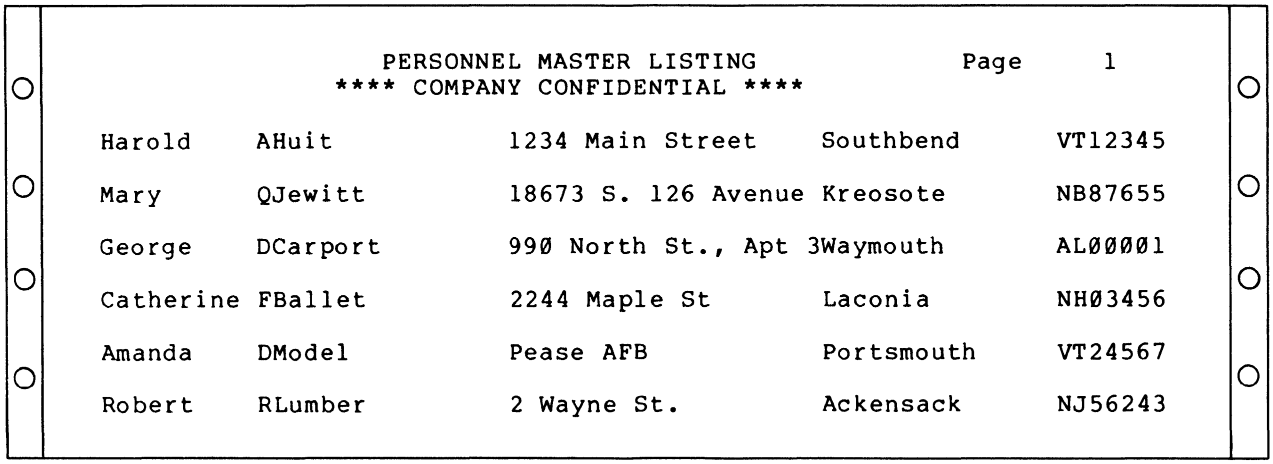
Figure 8-6: A 20-Line Logical Page



```
Column          1         2         3         4         5         6
Line   12345678901234567890123456789012345678901234567890123456789012

 1
 2
 3     Dear Mr. XXXXXXXXXXXXXXX                      Date: 99-XXX-99
 4
 5       XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 6       X                                                          X
 7       X                                                          X
 8       X              Preprinted message is here                  X
 9       X                                                          X
10       X                                                          X
11       XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
12
13     TO: XXXXXXXXXX X XXXXXXXXXXXXXXX
14         XXXXXXXXXXXXXXXXXXXX
15         XXXXXXXXXXXXXXX XX 99999
16
17
18
19
20
```

C81ART-20360-35

Figure 8-7: A Double-Spaced Master Listing



```
                     PERSONNEL MASTER LISTING            Page      1
                  **** COMPANY CONFIDENTIAL ****

   Harold     AHuit             1234 Main Street    Southbend      VT12345

   Mary       QJewitt           18673 S. 126 Avenue Kreosote       NB87655

   George     DCarport          99Ø North St., Apt 3Waymouth       ALØØØØ1

   Catherine  FBallet           2244 Maple St       Laconia        NHØ3456

   Amanda     DModel            Pease AFB           Portsmouth     VT24567

   Robert     RLumber           2 Wayne St.         Ackensack      NJ56243
```

C81ART-20370-25

Example 8-4: Page Advancing and Line Skipping

```
IDENTIFICATION DIVISION.
PROGRAM-ID. REPO1.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT INPUT-FILE   ASSIGN TO "REPIN.DAT".
    SELECT FORM1-REPORT ASSIGN TO "FORM1.DAT".
    SELECT FORM2-REPORT ASSIGN TO "FORM2.DAT".

DATA DIVISION.
FILE SECTION.
FD  INPUT-FILE.
01  INPUT-RECORD.
    02  I-NAME.
        03  I-FIRST                     PIC X(10).
        03  I-MID                       PIC X.
        03  I-LAST                      PIC X(15).
    02  I-ADDRESS.
        03  I-STREET                    PIC X(20).
        03  I-CITY                      PIC X(15).
        03  I-STATE                     PIC XX.
        03  I-ZIP                       PIC 99999.
FD  FORM1-REPORT.
01  FORM1-PRINT-LINE                    PIC X(80).
FD  FORM2-REPORT.
01  FORM2-PRINT-LINE                    PIC X(80).

WORKING-STORAGE SECTION.
01  END-OF-FILE                         PIC  X      VALUE SPACE.
01  MAX-LINES-ON-FORM2                  PIC  99     VALUE 55.
01  FORM2-LINE-COUNTER                  PIC  99     VALUE 00.
01  PAGE-NO                             PIC  99999  VALUE 0.
01  FORM1-LINE-3.
    02                                  PIC X(9)    VALUE SPACES.
    02  FORM1-LAST                      PIC X(15).
```

(continued on next page)

Example 8-4: Page Advancing and Line Skipping (Cont.)

```
01  FORM1-LINE-13.
    02                                   PIC X(4)   VALUE SPACES.
    02  FORM1-NAME                       PIC X(26).
01  FORM1-LINE-14.
    02                                   PIC X(4)   VALUE SPACES.
    02  FORM1-STREET                     PIC X(20).
01  FORM1-LINE-15.
    02                                   PIC X(4)   VALUE SPACES.
    02  FORM1-CITY                       PIC X(15).
    02                                   PIC X      VALUE SPACE.
    02  FORM1-STATE                      PIC XX.
    02                                   PIC X      VALUE SPACE.
    02  FORM1-ZIP                        PIC 99999.
01  FORM2-HEADER-1.
    02              PIC X(15) VALUE SPACES.
    02              PIC X(30) VALUE "   PERSONNEL MASTER LISTING   ".
    02              PIC X(10) VALUE SPACES.
    02              PIC XXXXX VALUE "Page ".
    02  F2H-PAGE    PIC ZZZZZ.
01  FORM2-HEADER-2.
    02              PIC X(15) VALUE SPACES.
    02              PIC X(30) VALUE "**** COMPANY CONFIDENTIAL ****".
PROCEDURE DIVISION.
A000-BEGIN.
    OPEN INPUT  INPUT-FILE
         OUTPUT FORM1-REPORT
                FORM2-REPORT.
    PERFORM A900-PRINT-HEADERS-ROUTINE.
    PERFORM A100-PRINT-REPORTS UNTIL END-OF-FILE = "Y".
    CLOSE INPUT-FILE
          FORM1-REPORT
          FORM2-REPORT.
    DISPLAY "END OF JOB".
    STOP RUN.

A100-PRINT-REPORTS.
    READ INPUT-FILE AT END MOVE "Y" TO END-OF-FILE.
    IF END-OF-FILE NOT = "Y"
       PERFORM A200-PRINT-REPORTS.

A200-PRINT-REPORTS.
    IF FORM2-LINE-COUNTER IS GREATER THAN MAX-LINES-ON-FORM2
       PERFORM A900-PRINT-HEADERS-ROUTINE.

    WRITE FORM2-PRINT-LINE FROM INPUT-RECORD
                           AFTER ADVANCING 2 LINES.
    ADD 2 TO FORM2-LINE-COUNTER.

    MOVE I-LAST      TO FORM1-LAST.
    WRITE FORM1-PRINT-LINE FROM FORM1-LINE-3
                           AFTER ADVANCING 3 LINES.
    MOVE I-NAME      TO FORM1-NAME.
    WRITE FORM1-PRINT-LINE FROM FORM1-LINE-13
                           AFTER ADVANCING 10 LINES.
    MOVE I-STREET    TO FORM1-STREET.
    WRITE FORM1-PRINT-LINE FROM FORM1-LINE-14.
    MOVE I-CITY      TO FORM1-CITY.
    MOVE I-STATE     TO FORM1-STATE.
    MOVE I-ZIP       TO FORM1-ZIP.
    WRITE FORM1-PRINT-LINE FROM FORM1-LINE-15.
```

(continued on next page)

Example 8-4: Page Advancing and Line Skipping (Cont.)

```
A900-PRINT-HEADERS-ROUTINE.

*
* This routine generates a form feed, writes two lines,
* skips 2 lines, then resets the line counter to 4 to
* indicate used lines on the current logical page.
* Line 5 on this page is the next print line.
*
     ADD 1 TO PAGE-NO.
     MOVE PAGE-NO TO F2H-PAGE.
     WRITE FORM2-PRINT-LINE FROM FORM2-HEADER-1
                            AFTER ADVANCING PAGE.
     WRITE FORM2-PRINT-LINE FROM FORM2-HEADER-2
                            BEFORE ADVANCING 2.
     MOVE 4 TO FORM2-LINE-COUNTER.
```

8.7 Programming the Linage-File Report

A linage-file report:

- Has sequential organization and access mode
- Contains variable-length records
- Consists of one or more logical pages
- Uses the LINAGE clause to define the number of lines on the logical page and to divide it into logical page sections
- Has a LINAGE-COUNTER special register assigned to it to monitor the number of lines written to the current logical page
- Contains compiler-generated line-feed characters that control the vertical position of the physical page in the line printer

The next sections discuss:

1. Defining the logical page with the LINAGE clause
2. Advancing to the next logical page
3. Programming for the page-overflow condition
4. Using the LINAGE-COUNTER special register
5. Programming for a 20-line logical page

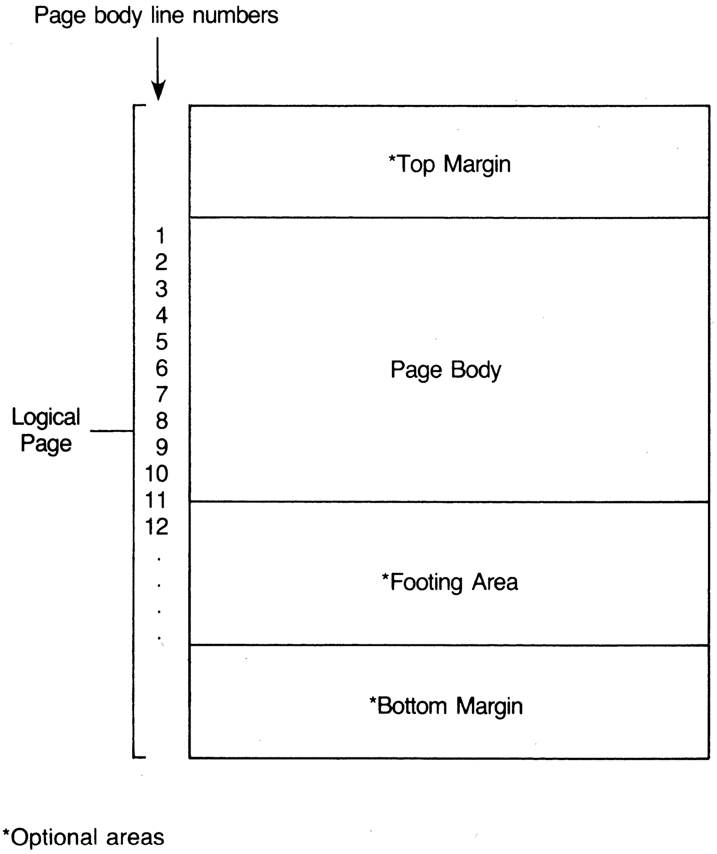
8.7.1 Defining the Logical Page with the LINAGE Clause

Use the LINAGE clause in the file description entry to define the number of lines on a logical page and to divide it into logical page sections. The page sections are named:

1. Top margin
2. Page body
3. Footing area
4. Bottom margin

Figure 8-8 shows all four sections of a linage-file logical page.

Figure 8-8: Logical Page Areas for a Linage-File Report



C81ART-20380-40

To define how many lines you want your program to skip at the top or the bottom of the logical page, you use these phrases of the LINAGE clause:

1. LINES AT TOP – to position the printer on the first print line in the page body
2. LINES AT BOTTOM – to position the printer at the top of the next logical page after the current page body is complete

Use the WITH FOOTING phrase to define a footing area in the logical page. This area controls your page-overflow conditions and lets you insert specific text on the bottom lines of your logical page, such as footnotes or page numbers. Section 8.7.3 and Example 8-5 explain this topic in more detail.

8.7.2 Advancing to the Next Logical Page

Linage-files automatically control the advancement to the next logical page whenever the LINAGE-COUNTER value equals the number of lines on the logical page. However, COBOL-81 also lets your program control logical page advancement with the WRITE statement.

To manually advance to the next logical page from any line in the current page body and position the printer on the first print line of the next page body, your program must include either the BEFORE ADVANCING PAGE clause or the AFTER ADVANCING PAGE clause of the WRITE statement.

8.7.3 Programming for the Page-Overflow Condition

If your program writes more lines than the logical page can accomodate, a page-overflow condition exists. When a page-overflow condition occurs, the compiler automatically advances the linage-file report to the top of the next logical page.

Note

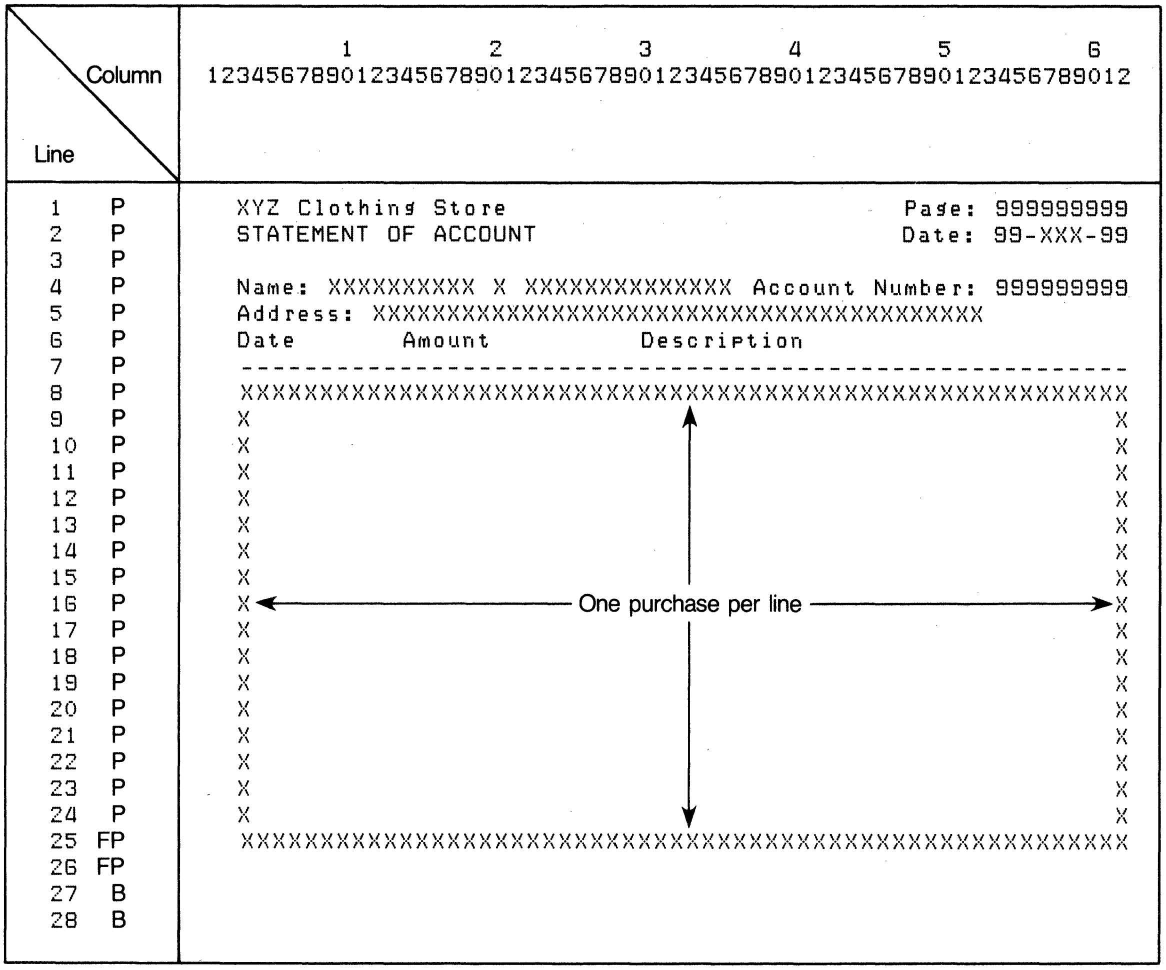
Do not write more lines than your logical page can accomodate. If you do overflow the page, the first overflow line prints on the first line of the next logical page. Your logical page is then out of synchronization with your physical page.

You must include routines in your program so it can determine when:

- Its logical page is full. This allows your program to print header information on the top of each form.
- It prints the last logical line on the logical page (whenever you do not want to use all the lines on the current logical page.) This allows your program to stop detail-line processing and provide summary totals in the current logical page before advancing to the next logical page.

Example 8-5 shows you ways to include these routines in your program. It uses the logical page shown in Figure 8-9. Each detail line of the report represents a separate purchase at the XYZ Clothing Store. Each page can contain from 1 to 18 purchase lines. Each customer can have an unlimited number of purchases. A total of purchases for each customer is to appear on line 25 of that customer's last statement page. Headers appear on the top of each page. The input file, INPUT.DAT, consists of individual purchase records sorted in ascending order by customer account number and purchase date. In Example 8-5, the LINAGE clause defines a footing area so the program can check for a page-overflow condition. When the condition is detected, the program executes its header routine to print lines 1 through 7.

Figure 8-9: A 28-Line Logical Page



```
                  1         2         3         4         5         6
Column   12345678901234567890123456789012345678901234567890123456789012
Line
 1 P       XYZ Clothing Store                           Page: 999999999
 2 P       STATEMENT OF ACCOUNT                         Date: 99-XXX-99
 3 P
 4 P       Name: XXXXXXXXXX X XXXXXXXXXXXXXX Account Number: 999999999
 5 P       Address: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 6 P       Date       Amount          Description
 7 P       ------------------------------------------------------------
 8 P       XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 9 P       X                                                          X
10 P       X                                                          X
11 P       X                                                          X
12 P       X                                                          X
13 P       X                                                          X
14 P       X                                                          X
15 P       X                                                          X
16 P       X                                                          X
17 P       X                                                          X
18 P       X                                                          X
19 P       X                                                          X
20 P       X                                                          X
21 P       X                                                          X
22 P       X                                                          X
23 P       X                                                          X
24 P       X                                                          X
25 FP      XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
26 FP
27 B
28 B
```

Legend: T = Top margin = line 00
P = Page body = lines 01-26
F = Footing area = lines 25-26
B = Bottom margin = lines 27-28

C81ART-20390-50

Example 8-5: Checking for Page-Overflow on a 28-Line Logical Page

```
IDENTIFICATION DIVISION.
PROGRAM-ID. REPOVF.
*********************************************************
*
* For RSTS/E - Print this report: PRINT REPORT.DAT/NOFEED
*
* For RSX - Print this report: PRINT/LENGTH=0  REPORT.DAT
*
*********************************************************
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT INPUT-FILE  ASSIGN TO "INPUT.DAT".
    SELECT REPORT-FILE ASSIGN TO "REPORT.DAT".

DATA DIVISION.
FILE SECTION.
FD  INPUT-FILE.
01  INPUT-RECORD.
    02  I-NAME.
        03  I-FIRST           PIC X(10).
        03  I-MID             PIC X.
        03  I-LAST            PIC X(15).
    02  I-ADDRESS.
        03  I-STREET          PIC X(20).
        03  I-CITY            PIC X(15).
        03  I-STATE           PIC XX.
        03  I-ZIP             PIC 99999.
    02  I-ACCOUNT-NUMBER      PIC X(9).
    02  I-PURCHASE-DATE       PIC XXXXXX.
    02  I-PURCHASE-AMOUNT     PIC S9(6)V99.
    02  I-PURCHASE-DESCRIP    PIC X(20).
FD  REPORT-FILE
    LINAGE IS 26 LINES
           WITH FOOTING AT 25
           LINES AT BOTTOM  2.
01  PRINT-LINE                PIC X(80).

WORKING-STORAGE SECTION.
01  HEAD-1.
    02  H1-LC     PIC 99.
    02  FILLER    PIC X(20) VALUE "XYZ Clothing Store  ".
    02  FILLER    PIC X(25) VALUE SPACES.
    02  FILLER    PIC X(6)  VALUE "Page: ".
    02  H1-PAGE   PIC Z(9).
01  HEAD-2.
    02  H2-LC     PIC 99.
    02  FILLER    PIC X(20) VALUE "STATEMENT OF ACCOUNT".
    02  FILLER    PIC X(25) VALUE SPACES.
    02  FILLER    PIC X(6)  VALUE "Date: ".
    02  H2-DATE   PIC X(9).
01  HEAD-3.
    02  H3-LC     PIC 99.
    02  FILLER    PIC X(6)  VALUE "Name: ".
    02  H3-FNAME  PIC X(10).
    02  FILLER    PIC X     VALUE SPACE.
    02  H3-MNAME  PIC X.
    02  FILLER    PIC X     VALUE SPACE.
    02  H3-LNAME  PIC X(15).
    02  FILLER    PIC X(17) VALUE " Account Number: ".
    02  H3-NUM    PIC Z(9).
```

(continued on next page)

Example 8-5: Checking for Page-Overflow on a 28-Line Logical Page (Cont.)

```
01  HEAD-4.
    02  H4-LC     PIC 99.
    02  FILLER    PIC X(9)   VALUE "Address: ".
    02  H4-STRT   PIC X(20).
    02  FILLER    PIC X      VALUE SPACE.
    02  H4-CITY   PIC X(15).
    02  FILLER    PIC X      VALUE SPACE.
    02  H4-STATE  PIC XX.
    02  FILLER    PIC X      VALUE SPACE.
    02  H4-ZIP    PIC 99999.
01  HEAD-5.
    02  H5-LC     PIC 99.
    02  FILLER    PIC X(4)   VALUE "Date".
    02  FILLER    PIC X(7)   VALUE SPACES.
    02  FILLER    PIC X(6)   VALUE "Amount".
    02  FILLER    PIC X(10)  VALUE SPACES.
    02  FILLER    PIC X(11)  VALUE "Description".
01  HEAD-6        PIC X(61)  VALUE ALL "-".
01  DETAIL-LINE.
    02  DET-LC    PIC 99.
    02  DL-DATE   PIC X(9).
    02  FILLER    PIC X      VALUE SPACE.
    02  DL-AMT    PIC $ZZZ,ZZZ.99-.
    02  FILLER    PIC X      VALUE SPACE.
    02  DL-DESC   PIC X(20).
01  TOTAL-LINE.
    02  TOT-LC    PIC 99.
    02  FILLER    PIC X(25)  VALUE "Total purchases to date: ".
    02  TL        PIC $ZZZ,ZZZ,ZZZ.99-.

01  TOTAL-PURCHASES PIC S9(9)V99.
01  PAGE-NUMBER     PIC S9(9).
01  HOLD-I-ACCOUNT-NUMBER  PIC X(9)    VALUE IS LOW-VALUES.
01  END-OF-FILE     PIC X      VALUE IS "N".
01  THESE-MANY      PIC 99     VALUE IS 1.
PROCEDURE DIVISION.
A000-BEGIN.
    OPEN INPUT  INPUT-FILE
         OUTPUT REPORT-FILE.
    DISPLAY " Enter date--DD-MMM-YY:".
    ACCEPT H2-DATE.
    PERFORM A100-READ-INPUT UNTIL END-OF-FILE = "Y".
A050-WRAP-UP.
    CLOSE INPUT-FILE
          REPORT-FILE.
    DISPLAY "END-OF-JOB".
    STOP RUN.
A100-READ-INPUT.
    READ INPUT-FILE AT END MOVE "Y" TO END-OF-FILE
                           PERFORM A400-PRINT-TOTALS
                           MOVE HIGH-VALUES TO I-ACCOUNT-NUMBER.
    DISPLAY INPUT-RECORD.
    IF END-OF-FILE NOT = "Y"
       AND I-ACCOUNT-NUMBER NOT = HOLD-I-ACCOUNT-NUMBER
              PERFORM A200-NEW-CUSTOMER.
    IF END-OF-FILE NOT = "Y"
       AND I-ACCOUNT-NUMBER = HOLD-I-ACCOUNT-NUMBER
              PERFORM A300-PRINT-DETAIL-LINE.
    MOVE I-ACCOUNT-NUMBER TO HOLD-I-ACCOUNT-NUMBER.
```

(continued on next page)

Example 8-5: Checking for Page-Overflow on a 28-Line Logical Page (Cont.)

```
A200-NEW-CUSTOMER.
    IF HOLD-I-ACCOUNT-NUMBER = LOW-VALUES
            PERFORM A600-SET-UP-HEADERS
            PERFORM A500-PRINT-HEADERS
            PERFORM A300-PRINT-DETAIL-LINE
        ELSE
            PERFORM A400-PRINT-TOTALS
            PERFORM A600-SET-UP-HEADERS
            PERFORM A500-PRINT-HEADERS
            PERFORM A300-PRINT-DETAIL-LINE.

A300-PRINT-DETAIL-LINE.
    MOVE I-PURCHASE-DATE     TO DL-DATE.
    MOVE I-PURCHASE-AMOUNT   TO DL-AMT.
    MOVE I-PURCHASE-DESCRIP TO DL-DESC.
    WRITE PRINT-LINE FROM DETAIL-LINE
                     AT END-OF-PAGE PERFORM A500-PRINT-HEADERS.
    ADD I-PURCHASE-AMOUNT TO TOTAL-PURCHASES.

A400-PRINT-TOTALS.
    MOVE TOTAL-PURCHASES TO TL.
    COMPUTE THESE-MANY = 25 - LINAGE-COUNTER.
    WRITE PRINT-LINE FROM TOTAL-LINE AFTER ADVANCING THESE-MANY LINES.
    MOVE 0 TO TOTAL-PURCHASES.

A500-PRINT-HEADERS.
    ADD 1 TO PAGE-NUMBER.
    MOVE PAGE-NUMBER TO H1-PAGE.
    WRITE PRINT-LINE FROM HEAD-1 AFTER ADVANCING PAGE.
    WRITE PRINT-LINE FROM HEAD-2.
    MOVE SPACES TO PRINT-LINE.
    WRITE PRINT-LINE.
    WRITE PRINT-LINE FROM HEAD-3.
    WRITE PRINT-LINE FROM HEAD-4.
    WRITE PRINT-LINE FROM HEAD-5.
    WRITE PRINT-LINE FROM HEAD-6.

A600-SET-UP-HEADERS.
    MOVE I-FIRST          TO H3-FNAME.
    MOVE I-MID            TO H3-MNAME.
    MOVE I-LAST           TO H3-LNAME.
    MOVE I-ACCOUNT-NUMBER TO H3-NUM.
    MOVE I-STREET         TO H4-STRT.
    MOVE I-CITY           TO H4-CITY.
    MOVE I-STATE          TO H4-STATE.
    MOVE I-ZIP            TO H4-ZIP.
```

8.7.4 Using the LINAGE-COUNTER

One way to keep track of how many lines your program writes on a logical page is to use the LINAGE-COUNTER special register. The compiler resets this register to 1 every time your program begins a new logical page, and adds 1 to this register for each line the program writes.

Before the program writes a new line, it checks the LINAGE-COUNTER value to see if the current logical page can accept the new line. If the value equals the maximum number of lines for the page body, the compiler generates the appropriate number of line-feed characters to position the device on the first print line of the next page body.

8.7.5 A Special Forms Example

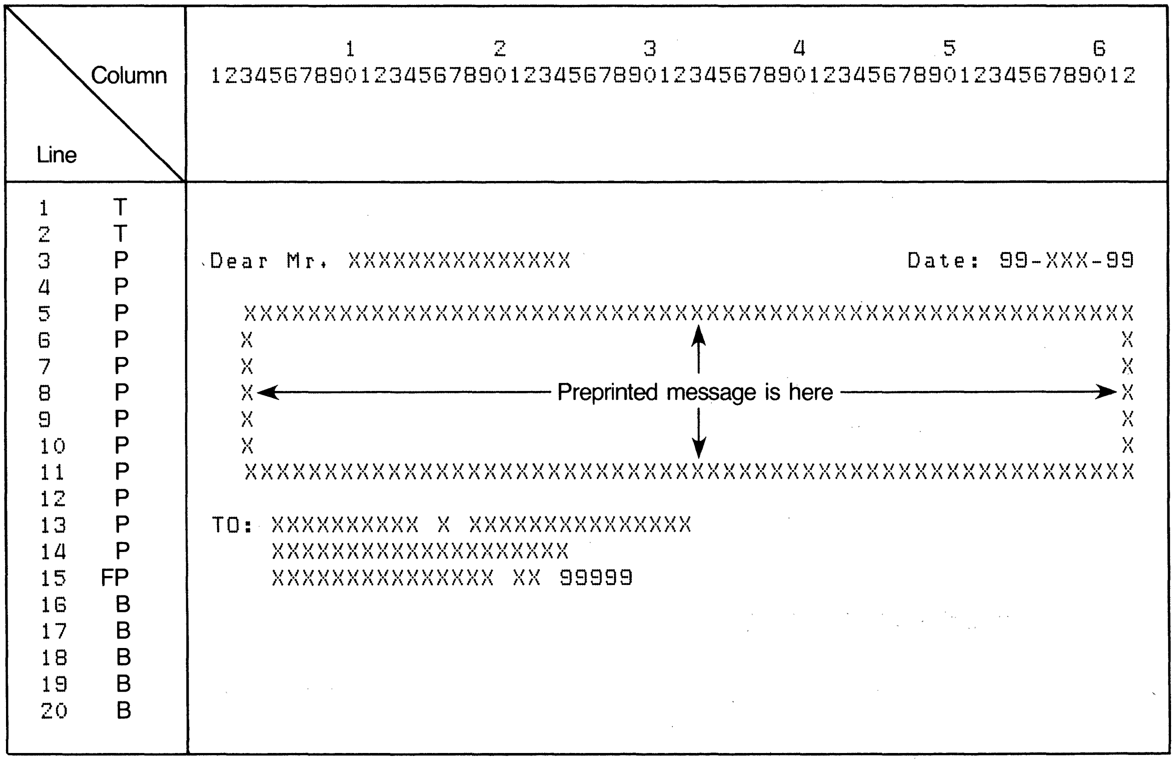
The file description entry in Example 8-6 uses the LINAGE clause to define the logical page areas shown in Figure 8-10. Figure 8-10 shows a 20-line logical page which includes a top margin (T), a page body (P), a footing area (F), and a bottom margin (B). Example 8-7 shows a complete program that generates the logical page shown in Figure 8-10 and uses the LINAGE clause in Example 8-6.

Example 8-6: A Sample LINAGE Clause for a 20-Line Logical Page

```
FD  MINIF1-REPORT
    LINAGE IS 13 LINES
                       LINES AT TOP       2
                       LINES AT BOTTOM    5.
```

The first line to which the logical page can be positioned is the third line. This is the first print line of the page. The page-overflow condition occurs when a WRITE statement causes the LINAGE-COUNTER value to equal 15. Line 15 is the last line on the page on which text can be written. Automatic page-advancement occurs when a WRITE statement causes the LINAGE-COUNTER value to exceed 15. When this condition occurs, the OTS automatically positions you on the first print line in the page body of the next logical page.

Figure 8-10: A 20-Line Logical Page



```
        Column           1         2         3         4         5         6
                12345678901234567890123456789012345678901234567890123456789012
Line

 1   T
 2   T
 3   P          Dear Mr. XXXXXXXXXXXXXXX                       Date: 99-XXX-99
 4   P
 5   P             XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 6   P             X                           ↑                             X
 7   P             X                           |                             X
 8   P             X<-------------- Preprinted message is here ------------->X
 9   P             X                           |                             X
10   P             X                           ↓                             X
11   P             XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
12   P
13   P          TO: XXXXXXXXXX X XXXXXXXXXXXXXXX
14   P              XXXXXXXXXXXXXXXXXXXX
15  FP              XXXXXXXXXXXXXXX XX 99999
16   B
17   B
18   B
19   B
20   B
```

Legend: T = Top margin = lines 1 and 2
P = Page body = lines 3 through 15
F = Footing area = line 15
B = Bottom margin = lines 16 through 20

C81ART-20400-50

Example 8-7: Programming a 20-Line Logical Page Defined by the LINAGE Clause

```
IDENTIFICATION DIVISION.
PROGRAM-ID. REPLINAG.
*********************************************************
*
* For RSTS/E - Print the report - PRINT MINIF1.DAT/NOFEED
*
*
* For RSX - Print the report - PRINT/LENGTH=0 MINIF1.DAT
*
*********************************************************
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT INPUT-FILE   ASSIGN TO "REPIN.DAT".
    SELECT MINIF1-REPORT ASSIGN TO "MINIF1.DAT".

DATA DIVISION.
FILE SECTION.
FD  INPUT-FILE.
01  INPUT-RECORD.
    02  I-NAME.
        03  I-FIRST                          PIC X(10).
        03  I-MID                            PIC X.
        03  I-LAST                           PIC X(15).
    02  I-ADDRESS.
        03  I-STREET                         PIC X(20).
        03  I-CITY                           PIC X(15).
        03  I-STATE                          PIC XX.
        03  I-ZIP                            PIC 99999.
FD  MINIF1-REPORT
    LINAGE IS 13 LINES
           LINES AT TOP    2
           LINES AT BOTTOM 5.
01  MINIF1-PRINT-LINE                         PIC X(80).
WORKING-STORAGE SECTION.
01  END-OF-FILE                          PIC  X      VALUE SPACE.
01  LINE-UP-OK                           PIC  X      VALUE SPACE.
01  MINIF1-LINE-3.
    02  FILLER                           PIC X(9)   VALUE SPACES.
    02  MINIF1-LAST                      PIC X(15).
    02  FILLER                           PIC X(23)  VALUE SPACES.
    02  FILLER                           PIC X(6)   VALUE "Date: ".
    02  MINIF1-DATE                      PIC 99/99/99.
01  MINIF1-LINE-13.
    02  FILLER                           PIC X(4)   VALUE SPACES.
    02  MINIF1-NAME                      PIC X(26).
01  MINIF1-LINE-14.
    02  FILLER                           PIC X(4)   VALUE SPACES.
    02  MINIF1-STREET                    PIC X(20).
01  MINIF1-LINE-15.
    02  FILLER                           PIC X(4)   VALUE SPACES.
    02  MINIF1-CITY                      PIC X(15).
    02  FILLER                           PIC X      VALUE SPACE.
    02  MINIF1-STATE                     PIC XX.
    02  FILLER                           PIC X      VALUE SPACE.
    02  MINIF1-ZIP                       PIC 99999.
PROCEDURE DIVISION.
A000-BEGIN.
    OPEN OUTPUT MINIF1-REPORT.
    ACCEPT MINIF1-DATE FROM DATE.
    PERFORM A300-FORM-LINE-UP UNTIL LINE-UP-OK = "Y".
    OPEN INPUT  INPUT-FILE.
    PERFORM A100-READ-INPUT UNTIL END-OF-FILE = "Y".
```

(continued on next page)

Example 8-7: Programming a 20-Line Logical Page Defined by the LINAGE Clause (Cont.)

```
A010-WRAP-UP.
    CLOSE INPUT-FILE
          MINIF1-REPORT.
    DISPLAY "END OF JOB".
    STOP RUN.
A100-READ-INPUT.
    READ INPUT-FILE AT END MOVE "Y" TO END-OF-FILE.
    IF END-OF-FILE NOT = "Y"
       PERFORM A200-PRINT-REPORT.
A200-PRINT-REPORT.
    MOVE I-LAST          TO MINIF1-LAST.
    WRITE MINIF1-PRINT-LINE FROM MINIF1-LINE-3 BEFORE ADVANCING 1 LINE.
    MOVE SPACES TO MINIF1-PRINT-LINE.
    WRITE MINIF1-PRINT-LINE AFTER ADVANCING 9 LINES.
    MOVE I-NAME          TO MINIF1-NAME.
    WRITE MINIF1-PRINT-LINE FROM MINIF1-LINE-13 BEFORE ADVANCING 1 LINE.
    MOVE I-STREET        TO MINIF1-STREET.
    WRITE MINIF1-PRINT-LINE FROM MINIF1-LINE-14 BE  TO MINIF1-CITY.
    MOVE I-STATE     TO MINIF1-STATE.
    MOVE I-ZIP       TO MINIF1-ZIP.
    WRITE MINIF1-PRINT-LINE FROM MINIF1-LINE-15 BEFORE ADVANCING 1 LINE.
A300-FORM-LINE-UP.
    MOVE ALL "X" TO INPUT-RECORD.
    PERFORM A200-PRINT-REPORT 3 TIMES.
    DISPLAY "Is line up OK? (Y/N): " WITH NO ADVANCING.
    ACCEPT LINE-UP-OK.
```

8.8 How to Print Your Report

The next two sections discuss how to print reports on your operating system. The first section explains how to print a conventional-file report. The second section explains how to print a linage-file report.

8.8.1 Printing the Conventional Report

It is the information in the system spooler that controls your line printer and defines its current default page size and form identification. You must provide special form handling information to your system spooler whenever you want to print a report on a different size form. The method you use to signal a form change to the system spooler depends on your mode of printing (see Section 8.4).

If your report *does not* conform to the printer's current default page size, you – or the system manager – must change your system spooler's form specifications from a privileged account. You must do this each time you mount a form whose dimensions differ from the previous form.

If you are printing your special-forms report from a storage device, you must identify the form in your PRINT command. See your system's PRINT command documentation.

If you are printing your special-forms report on line, you must change the system spooler's current default page size *before* you run your program.

Once the current default page size conforms to your report's dimensions you can then print the report.

Refer to your operating system's spooler documentation for more specific information. On a RSTS/E system, refer to the *RSTS/E System Manager's Guide*. On an RSX-11M or RSX-11M-PLUS system, refer to your queue manager documentation.

8.8.2 Printing a Linage-File Report

To print a linage-file report on a RSTS/E system, use the /NOFEED file qualifier of the PRINT command to suppress the insertion of form-feed characters. For example:

```
PRINT report-file-specification/NOFEED
```

To print a linage-file report on an RSX-11M/M-PLUS system, use the /LENGTH=0 file qualifier of the PRINT command to suppress the insertion of form-feed characters. For example:

```
PRINT/LENGTH=0 report-file-specification
```

The LINAGE clause causes a COBOL-81 report file to be in print-file format (see Chapter 1, Section 1.4.2, Print-Controlled Record Format). When a WRITE statement positions the file to the top of the next logical page, device positioning occurs by line spacing rather than by page ejection or form feed.

The default PRINT command causes the insertion of a form-feed character when a form nears the end of a page. Therefore, when the default PRINT command refers to a linage-file report, it can change the report's page spacing.

8.9 Solving Report Problems

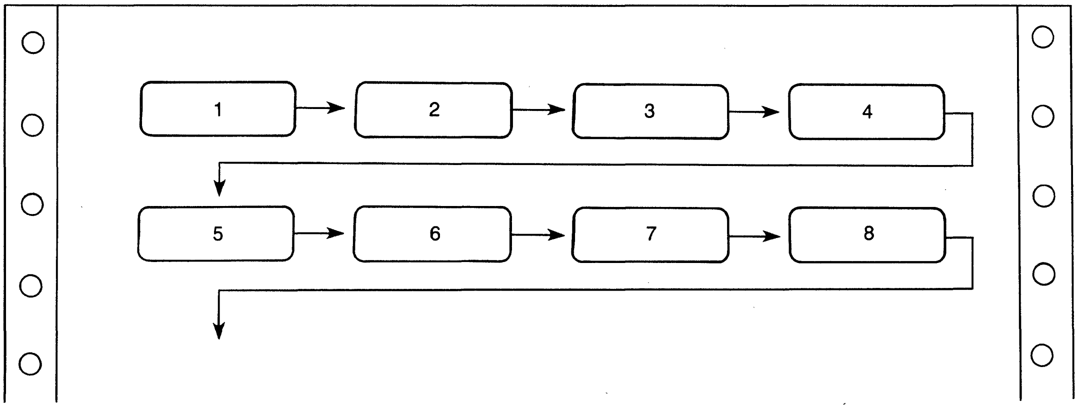
There are several variations to the basic report format. The next sections present, explain, and propose solutions to them.

8.9.1 Printing More Than One Logical Line on a Single Physical Line

When there are few columns in your report, you can print several logical lines on one physical line. If you were to print names and addresses on four-up self-sticking multi-label forms, you would print the form left-to-right, top-to-bottom, as shown in Figure 8-11 and Example 8-8. To print four-up self sticking labels you must format each logical line with four input records.

However, if the columns must be in sorted order by column, the task becomes more difficult. The last line at the end of the first column is continued at the top of the second column of the same page, indented to the right, and so forth, as shown in Figure 8-12 and Example 8-9. Example 8-9 defines a table containing all data to appear on the page. It fills the table by reading the input records and storing the data in the table as it is to appear on the page. It space-fills the table after printing its contents. Then, when it reaches the end of file, the remaining entries in the table are automatically blank. You can extend this technique to print any number of logical lines on a single physical line.

Figure 8-11: Printing Labels Four-Up



C81ART-20410-20

Example 8-8: Printing Labels Four-Up

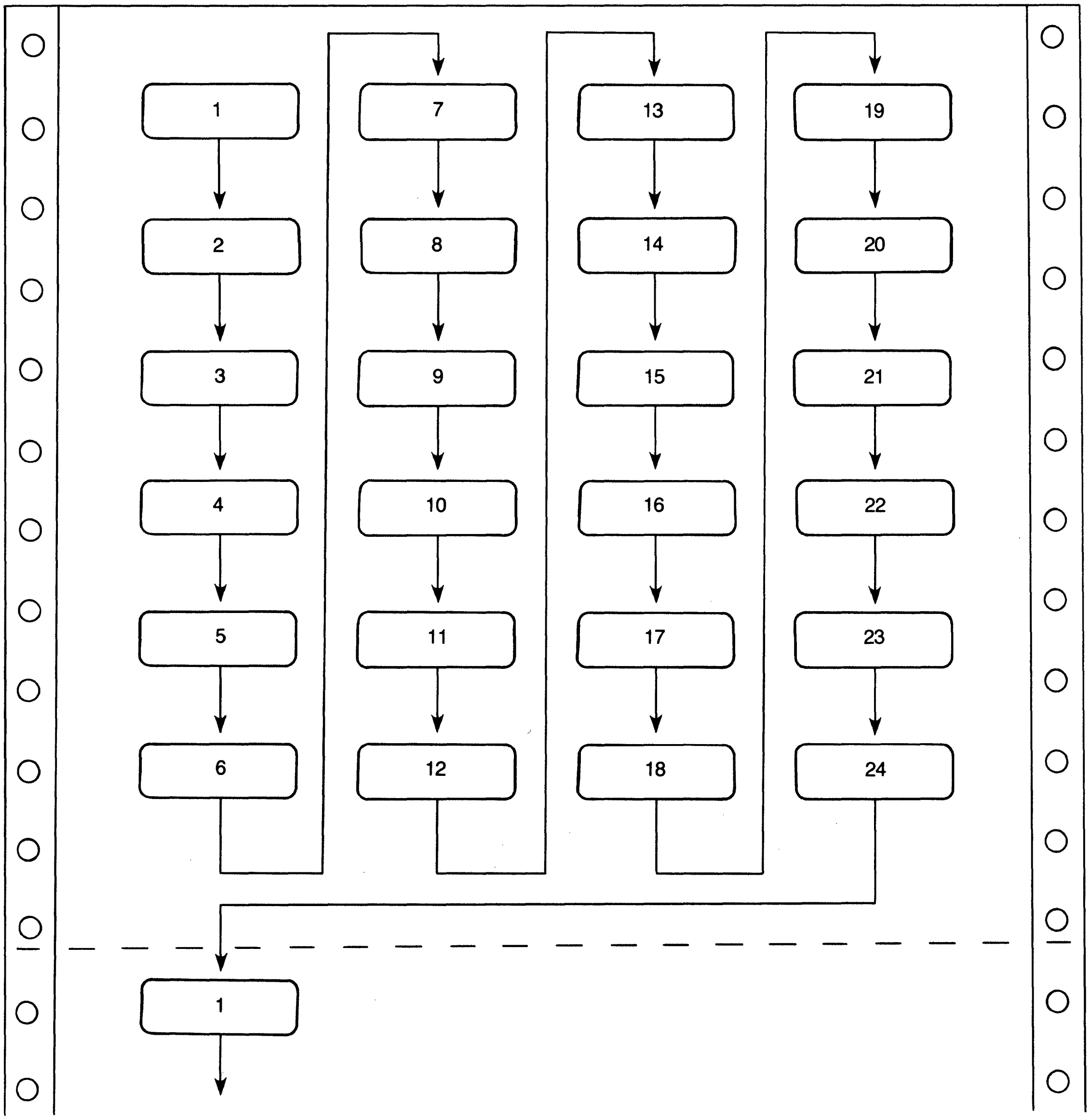
```
IDENTIFICATION DIVISION.
PROGRAM-ID. REP02.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT INPUT-FILE  ASSIGN TO "LABELS.DAT".
    SELECT REPORT-FILE ASSIGN TO "LABELS.REP".
DATA DIVISION.
FILE SECTION.
FD  INPUT-FILE.
01  INPUT-RECORD.
    02  INPUT-NAME      PIC X(20).
    02  INPUT-ADDRESS   PIC X(15).
    02  INPUT-CITY      PIC X(10).
    02  INPUT-STATE     PIC XX.
    02  INPUT-ZIP       PIC 99999.
FD  REPORT-FILE.
01  REPORT-RECORD       PIC X(132).
WORKING-STORAGE SECTION.
01  LABELS-TABLE.
        03  NAME-LINE.
            05  LINE-1 OCCURS 4 TIMES INDEXED BY INDEX-1.
                07  LABEL-NAME       PIC X(20).
                07  FILLER           PIC X(10).
        03  ADDRESS-LINE.
            05  LINE-2 OCCURS 4 TIMES INDEXED BY INDEX-2.
                07  LABEL-ADDRESS    PIC X(15).
                07  FILLER           PIC X(15).
        03  CSZ-LINE.
            05  LINE-3 OCCURS 4 TIMES INDEXED BY INDEX-3.
                07  LABEL-CITY       PIC X(10).
                07  FILLER           PIC XXXX.
                07  LABEL-STATE      PIC XX.
                07  FILLER           PIC XXXX.
                07  LABEL-ZIP        PIC 99999.
                07  FILLER           PIC XXXXX.


01  END-OF-FILE             PIC X.

PROCEDURE DIVISION.
A000-BEGIN.
    OPEN INPUT  INPUT-FILE
         OUTPUT REPORT-FILE.
    MOVE SPACES TO LABELS-TABLE.
    SET INDEX-1, INDEX-2, INDEX-3 TO 1.
    PERFORM A100-READ-INPUT UNTIL END-OF-FILE = "Y".
A050-WRAP-UP.
    IF LABEL-NAME(1) IS NOT EQUAL TO SPACES
       PERFORM A300-PRINT-FOUR-LABELS.
A050-END-OF-JOB.
    CLOSE INPUT-FILE
          REPORT-FILE.
    DISPLAY "END OF JOB".
    STOP RUN.

A100-READ-INPUT.
    READ INPUT-FILE AT END MOVE "Y" TO END-OF-FILE.
    IF END-OF-FILE = "Y" NEXT SENTENCE
       ELSE PERFORM A200-GENERATE-TABLE.
```

(continued on next page)

Example 8-8: Printing Labels Four-Up (Cont.)

```
A200-GENERATE-TABLE.
    MOVE INPUT-NAME      TO LABEL-NAME(INDEX-1)
    MOVE INPUT-ADDRESS   TO LABEL-ADDRESS(INDEX-2)
    MOVE INPUT-CITY      TO LABEL-CITY(INDEX-3)
    MOVE INPUT-STATE     TO LABEL-STATE(INDEX-3)
    MOVE INPUT-ZIP       TO LABEL-ZIP(INDEX-3)
    IF INDEX-1 = 4 PERFORM A300-PRINT-FOUR-LABELS
       ELSE        SET INDEX-1, INDEX-2, INDEX-3 UP BY 1.

A300-PRINT-FOUR-LABELS.
    WRITE REPORT-RECORD FROM NAME-LINE AFTER ADVANCING 3.
    WRITE REPORT-RECORD FROM ADDRESS-LINE AFTER ADVANCING 1.
    WRITE REPORT-RECORD FROM CSZ-LINE AFTER ADVANCING 1.
    MOVE SPACES TO LABELS-TABLE.
    SET INDEX-1, INDEX-2, INDEX-3 TO 1.
```

Figure 8-12: Printing Labels Four-Up in Sort Order



C82ART-20420-50

Example 8-9: Printing Labels Four-Up in Sort Order

```
IDENTIFICATION DIVISION.
PROGRAM-ID. REP03.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT INPUT-FILE  ASSIGN TO "LABELS.DAT".
    SELECT REPORT-FILE ASSIGN TO "LABELS.REP".
DATA DIVISION.
FILE SECTION.
FD  INPUT-FILE.
01  INPUT-RECORD.
    02  INPUT-NAME      PIC X(20).
    02  INPUT-ADDRESS   PIC X(15).
    02  INPUT-CITY      PIC X(10).
    02  INPUT-STATE     PIC XX.
    02  INPUT-ZIP       PIC 99999.
FD  REPORT-FILE.
01  REPORT-RECORD       PIC X(132).
WORKING-STORAGE SECTION.
01  LABELS-TABLE.
    03  FOUR-UP OCCURS 6 TIMES INDEXED BY ROW-INDEX.
        04  NAME-LINE.
            05  LINE-1 OCCURS 4 TIMES INDEXED BY NAME-INDEX.
                07  LABEL-NAME       PIC X(20).
                07  FILLER           PIC X(10).
        04  ADDRESS-LINE.
            05  LINE-2 OCCURS 4 TIMES INDEXED BY ADDRESS-INDEX.
                07  LABEL-ADDRESS    PIC X(15).
                07  FILLER           PIC X(15).
        04  CSZ-LINE.
            05  LINE-3 OCCURS 4 TIMES INDEXED BY CSZ-INDEX.
                07  LABEL-CITY       PIC X(10).
                07  FILLER           PIC XXXX.
                07  LABEL-STATE      PIC XX.
                07  FILLER           PIC XXXX.
                07  LABEL-ZIP        PIC 99999.
                07  FILLER           PIC XXXXX.

01  END-OF-FILE            PIC X.

PROCEDURE DIVISION.
A000-BEGIN.
    OPEN INPUT  INPUT-FILE
         OUTPUT REPORT-FILE.
    MOVE SPACES TO LABELS-TABLE.
    SET ROW-INDEX, NAME-INDEX, ADDRESS-INDEX, CSZ-INDEX TO 1.
    PERFORM A100-READ-INPUT UNTIL END-OF-FILE = "Y".

A050-WRAP-UP.
    IF LABEL-NAME(1, 1) IS NOT EQUAL TO SPACES
       PERFORM A300-PRINT-PAGE-OF-LABELS VARYING ROW-INDEX
               FROM 1 BY 1 UNTIL ROW-INDEX IS GREATER THAN 6.
```

(continued on next page)

Example 8-9: Printing Labels Four-Up in Sort Order (Cont.)

```
A050-END-OF-JOB.
    CLOSE INPUT-FILE
          REPORT-FILE.
    DISPLAY "END OF JOB".
    STOP RUN.

A100-READ-INPUT.
    READ INPUT-FILE AT END MOVE "Y" TO END-OF-FILE.
    IF END-OF-FILE = "Y" NEXT SENTENCE
       ELSE PERFORM A200-GENERATE-LABELS.

A200-GENERATE-LABELS.
    MOVE INPUT-NAME     TO LABEL-NAME(ROW-INDEX, NAME-INDEX)
    MOVE INPUT-ADDRESS  TO LABEL-ADDRESS(ROW-INDEX, ADDRESS-INDEX)
    MOVE INPUT-CITY     TO LABEL-CITY(ROW-INDEX, CSZ-INDEX)
    MOVE INPUT-STATE    TO LABEL-STATE(ROW-INDEX, CSZ-INDEX)
    MOVE INPUT-ZIP      TO LABEL-ZIP(ROW-INDEX, CSZ-INDEX)
    IF ROW-INDEX = 6 AND NAME-INDEX = 4
       PERFORM A300-PRINT-PAGE-OF-LABELS VARYING ROW-INDEX
               FROM 1 BY 1 UNTIL ROW-INDEX IS GREATER THAN 6
       MOVE SPACES TO LABELS-TABLE
       SET ROW-INDEX, NAME-INDEX, ADDRESS-INDEX, CSZ-INDEX TO 1
     ELSE
       PERFORM A210-UPDATE-INDEXES.

A210-UPDATE-INDEXES.
    IF ROW-INDEX =  6 SET ROW-INDEX      TO 1
                      SET NAME-INDEX
                          ADDRESS-INDEX
                          CSZ-INDEX   UP BY 1
       ELSE
               SET ROW-INDEX UP BY 1.

A300-PRINT-PAGE-OF-LABELS.
    WRITE REPORT-RECORD FROM NAME-LINE(ROW-INDEX)
          AFTER ADVANCING 3.
    WRITE REPORT-RECORD FROM ADDRESS-LINE(ROW-INDEX)
          AFTER ADVANCING 1.
    WRITE REPORT-RECORD FROM CSZ-LINE(ROW-INDEX)
          AFTER ADVANCING 1.
```

8.9.2 Group Indicating

Group indicating is a process that greatly improves a report's readability where there are long sequences of entries which have some element in common. You print the element once, then leave it blank for subsequent lines so long as there is no change in that element. For example, if your sample files's sort sequence is State (major key) and City (minor key), you could get sequences like those in Table 8-1.

Table 8-1: Results of Group Indicating

Without Group Indicating			With Group Indicating		
STATE	CITY	STORE NUMBER	STATE	CITY	STORE NUMBER
Arizona	Grand Canyon	111111	Arizona	Grand Canyon	111111
Arizona	Grand Canyon	123456			123456
Arizona	Grand Canyon	222222			222222
Arizona	Tucson	333333		Tucson	333333
Arizona	Tucson	444444			444444
Arizona	Tucson	555555			555555
Massachusetts	Maynard	111111	Massachusetts	Maynard	111111
Massachusetts	Maynard	222222			222222
Massachusetts	Maynard	333333			333333
Massachusetts	Maynard	444444			444444
Massachusetts	Tewksbury	111111		Tewksbury	111111
Massachusetts	Tewksbury	222222			222222
New Hampshire	Manchester	111111	New Hampshire	Manchester	111111
New Hampshire	Manchester	222222			222222
New Hampshire	Merrimack	333333		Merrimack	333333
New Hampshire	Merrimack	444444			444444
New Hampshire	Merrimack	555555			555555
New Hampshire	Nashua	666666		Nashua	666666

8.9.3 Fitting Reports on the Page

If you need 160 columns and the print page is limited to 132 you can:

- Eliminate as many unused spaces as possible between columns. Columns should not be run together; however, you can use one blank space instead of several.
- Eliminate the nonessential.
- Print two or more lines, staggering the headers and columns.
- Print two reports.

8.9.4 Printing Totals Before Detail Lines

If a report must include totals at the top of the page before the detail lines, there are three solutions:

1. Store the logical print lines in a table, total the table, and then print from the table.
2. Pass through the file twice. The first time through, compute the totals. The second time through, print the report. This method is slow and is complicated if there are many subtotals.
3. Write the lines into a file with a sort key containing the report, page, and line number. When your program writes the last line and computes the total, have it assign a page and line number to the total line's sort key. Use an appropriate page and line number to cause the total line to sort in front of its detail lines. After the program completes, sort the file, read it, drop the sort key, and produce the report.

8.9.5 Underlining Items in Your Reports

Sometimes you must underline a column of numbers to denote a total and also underline the total to highlight it:

```
1234
1122
2356
====
```

To print a single underline use the underscore character and suppress line spacing. For example:

```
WRITE PRINT-LINE FROM SINGLE-UNDERLINE-TOTAL
                 BEFORE ADVANCING 0 LINES.
```

This overprints the underscore on the previous line, underlining the item: 1122. Use the equal sign (=) to simulate double underlines. However, write the equal signs on the next line. For example:

```
WRITE PRINT-LINE FROM DOUBLE-UNDERLINE-TOTAL AFTER ADVANCING 1 LINE.
```

8.9.6 Bolding Items in Your Reports

To bold an entire line in a report:

1. Write the line as many times as you want, specifying the BEFORE ADVANCING 0 LINES phrase (three times is sufficient). This darkens the line but does not advance to the next line.
2. Write the line one last time without the BEFORE ADVANCING phrase. This overprints the line again and advances to the next print line.

For example:

```
WRITE PRINT-LINE FROM TOTAL-LINE BEFORE ADVANCING 0 LINES.
WRITE PRINT-LINE FROM TOTAL-LINE BEFORE ADVANCING 0 LINES.
WRITE PRINT-LINE FROM TOTAL-LINE BEFORE ADVANCING 0 LINES.
WRITE PRINT-LINE FROM TOTAL-LINE.
```

This example produces a darker image in the report. You can use similar statements for characters and words, as well as complete lines. To bold only a word or only a character within a line, you must:

1. Write the print line and specify the BEFORE ADVANCING 0 LINES phrase.
2. Create a skeleton line by removing all other items in the print line that are not to be bolded.
3. Write the skeleton line as many times as you want and specify the BEFORE ADVANCING 0 LINES phrase. This darkens the items in the skeleton line but does not advance to the next line.
4. Write the skeleton line one last time without the BEFORE ADVANCING phrase. This overprints the line again and advances to the next print line.

For example:

```
    WRITE PRINT-LINE FROM TOTAL-LINE BEFORE ADVANCING 0 LINES.
*
* Move spaces over the items in the source print line (TOTAL-LINE)
* that are not to be bolded
*
    MOVE SPACES TO ...
    WRITE PRINT-LINE FROM TOTAL-LINE BEFORE ADVANCING 0 LINES.
    WRITE PRINT-LINE FROM TOTAL-LINE BEFORE ADVANCING 0 LINES.
    WRITE PRINT-LINE FROM TOTAL-LINE.
```

Chapter 9
Forms for Video Terminals

This chapter explains how you can design an online video form similar to a printed form using COBOL-81 ACCEPT and DISPLAY statements. These statements provide you with options for developing video forms on either VT52, VT100, or Professional terminals, and let you write your application without regard to the type of terminal the application will eventually run on. You can run your forms application on any of these three terminals. However, not all options are available for the VT52 terminal. Those options that are not available for the VT52 terminal have no effect on the form.

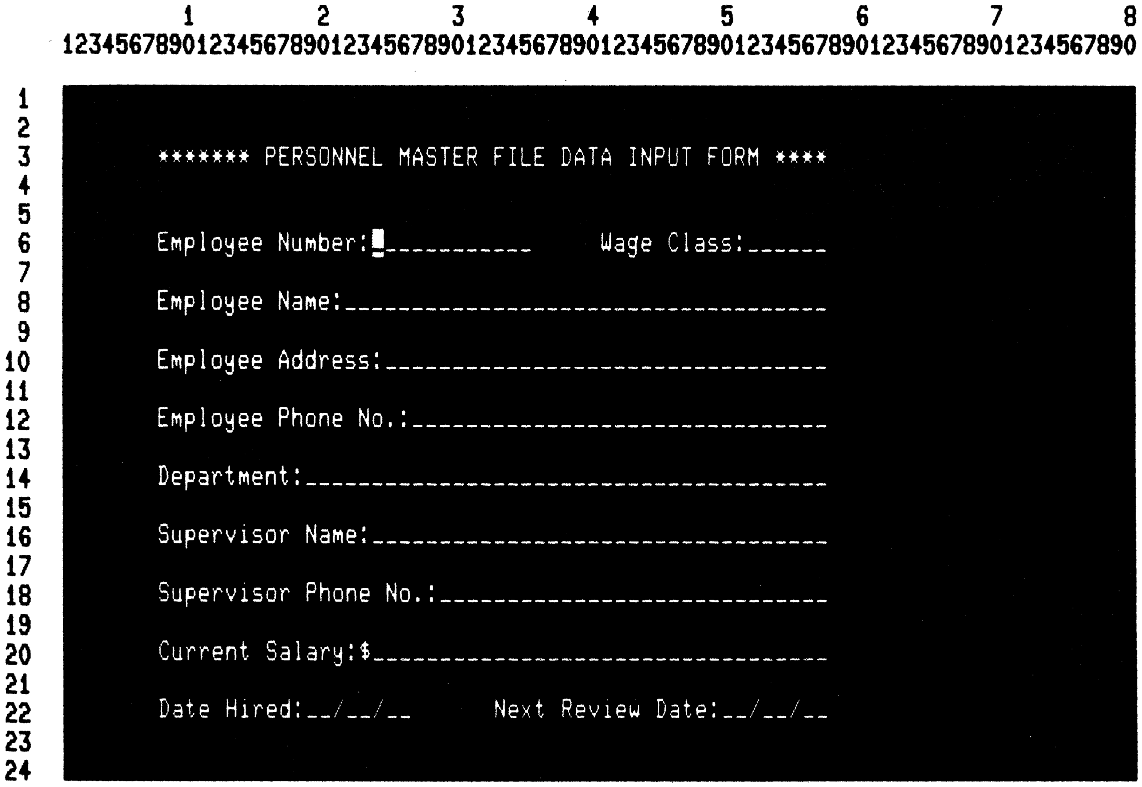
For simple screen applications, or applications requiring specialized screen displays, such as scrolling regions or double-width/double-height displays, refer to Appendix A, Designing Your Form with Escape Sequences. Using escape sequences does not cause the compiler to include special screen handling routines as does the ACCEPT/DISPLAY options. Escape sequences can reduce the memory space used by your program. However, most screen applications make use of both the ACCEPT and DISPLAY options and escape sequences.

A video form allows you to:

- Improve the appearance of an application's terminal dialog
- Make data entry applications, menu selections, and special control keys easier to use
- Clarify the type of input expected from an operator

For example, Figure 9-1 is a sample form created by a COBOL-81 program that lets you enter employee information into a master file. This program prompts the entry clerk for input data to the form. The program then moves the cursor to another part of the form and prompts the clerk for more data. Once all data is entered, the program writes it to the master file and paints a new form.

Figure 9-1: Adding Information to a Master File with a Video Form



```


       ******* PERSONNEL MASTER FILE DATA INPUT FORM ****


       Employee Number:___________      Wage Class:______

       Employee Name:___________________________________

       Employee Address:________________________________

       Employee Phone No.:______________________________

       Department:______________________________________

       Supervisor Name:_________________________________

       Supervisor Phone No.:____________________________

       Current Salary:$_________________________________

       Date Hired:__/__/__       Next Review Date:__/__/__
```

C81ART-20430-30

9.1 Designing Your Form with ACCEPT/DISPLAY Options

To help you design a video form, the ACCEPT/DISPLAY options allow you to do the following:

- Erase parts or all of the screen
- Use relative and absolute cursor positioning
- Specify video attributes on data to be displayed and accepted
- Convert data to appropriate usage when accepting data
- Handle error conditions when accepting and displaying data
- Provide screen protection by limiting the number of characters typed on the terminal
- Accept data without echoing
- Specify default values for ACCEPT statements
- Define and handle special control keys

The remainder of this chapter discusses these topics.

9.1.1 Clearing a Screen Area

To clear part or all of your screen before you accept or display data, you can use one of the following ERASE options of the ACCEPT and DISPLAY statements:

- ERASE SCREEN – Erases the entire screen before accepting or displaying data at the specified or implied cursor position.
- ERASE LINE – Erases the entire specified line before accepting or displaying data at the specified or implied cursor position.
- ERASE TO END OF SCREEN – Erases from the specified or implied cursor position to the end of the screen before accepting or displaying data at the specified cursor position.
- ERASE TO END OF LINE – Erases from the specified or implied cursor position to the end of the line before accepting or displaying data at the specified cursor position.

Table 9-1 lists the ERASE options and indicates if the option requires relative or absolute cursor positioning for your terminal type. (See Section 9.1.2.)

Table 9-1: Cursor Positioning Requirements for ERASE Options

	Cursor Positioning for Your Terminal Type	
ERASE Option	**VT52**	**VT100 and Professional**
ERASE SCREEN	Absolute only	Absolute or Relative
ERASE LINE	Absolute only	Absolute or Relative
ERASE TO END OF SCREEN	Absolute or Relative	Absolute or Relative
ERASE TO END OF LINE	Absolute or Relative	Absolute or Relative

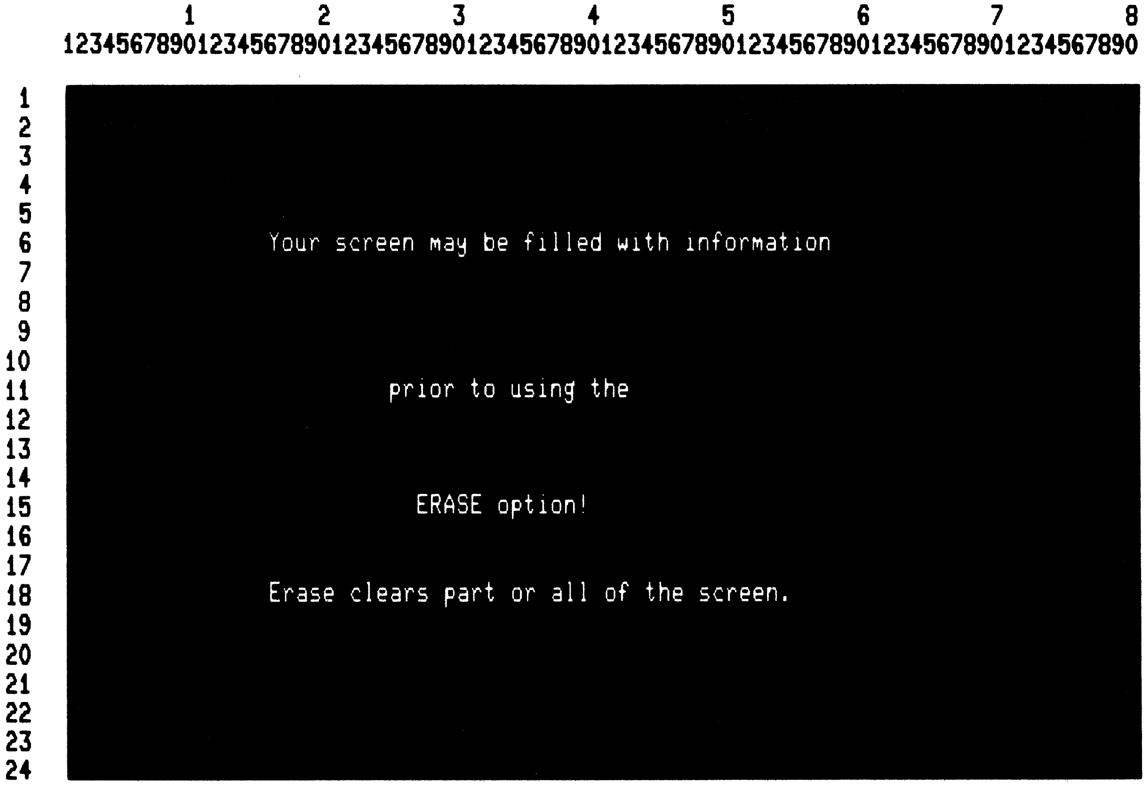
In Example 9-1, and as shown in Figure 9-2, the entire screen is erased before "Employee number:" is displayed.

Example 9-1: Erasing a Screen

```
IDENTIFICATION DIVISION.
PROGRAM-ID. ERASEIT.
DATA DIVISION.
PROCEDURE DIVISION.
A00-BEGIN.
    DISPLAY "Employee number:" LINE 4 COLUMN 5 ERASE SCREEN.
    DISPLAY " " LINE 23 COLUMN 1.
    STOP RUN.
```

Figure 9-2: Effects of the ERASE Option

Screen Before ERASE executes

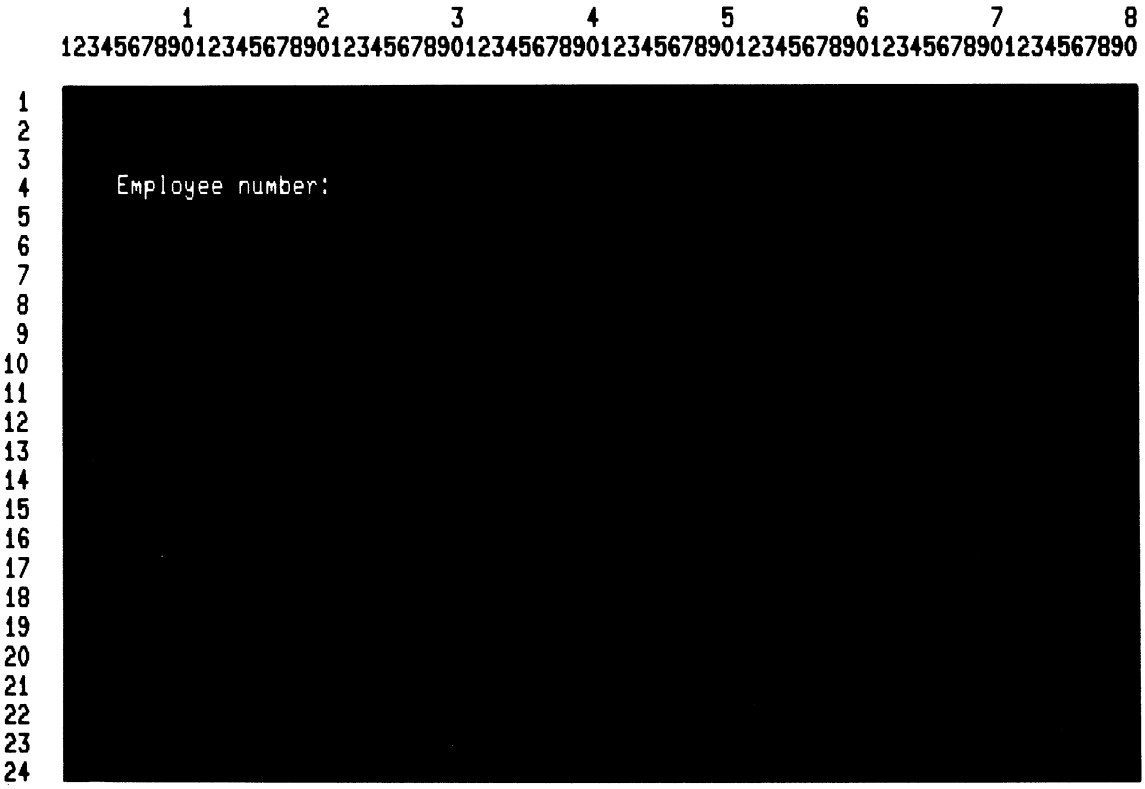


```





               Your screen may be filled with information




                        prior to using the



                          ERASE option!


               Erase clears part or all of the screen.
```

C81ART-20440-30

Figure 9-2: Effects of the ERASE Option (Cont.)

Screen After ERASE executes



```
    Employee number:
```

C81ART-20450-30

9.1.2 Horizontal and Vertical Positioning of the Cursor

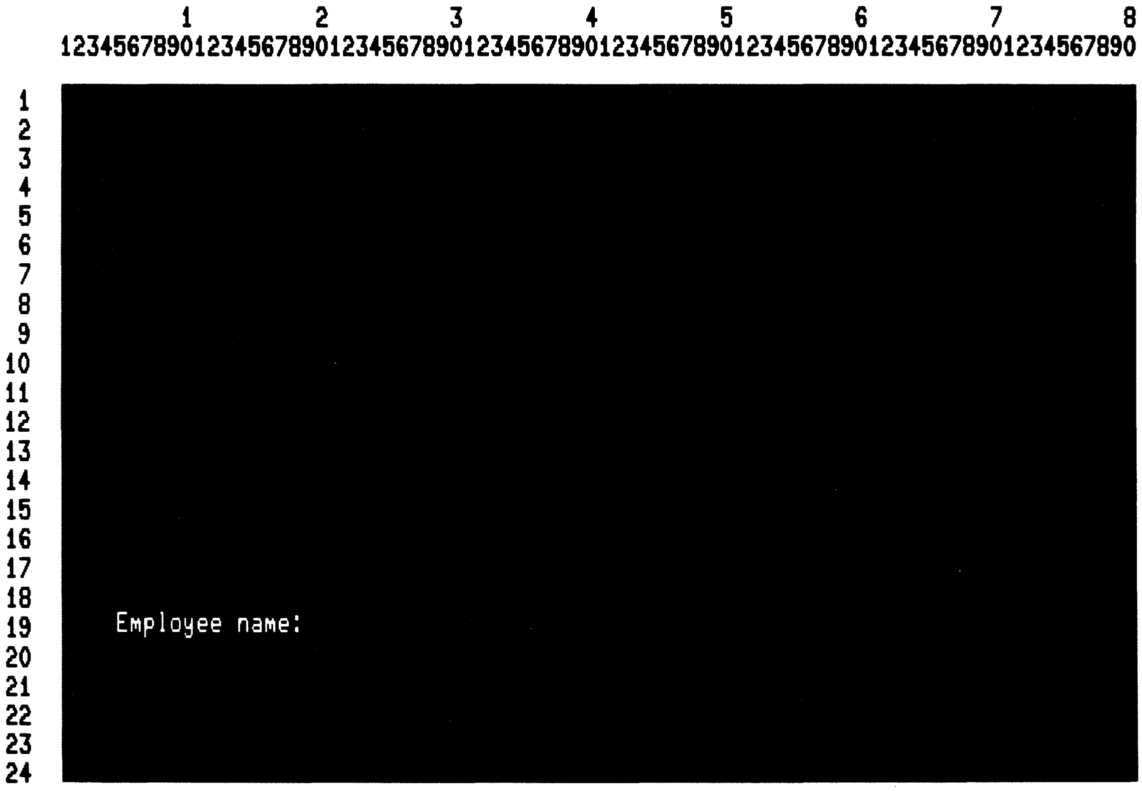
To position data items at a specified line and column, you use the LINE NUMBER (or LINE) and COLUMN NUMBER (or COLUMN) clauses, respectively. You can use these clauses with both the ACCEPT and DISPLAY statements. You can also use literals or numeric data items to specify line and column numbers.

In Example 9-2, and as shown in Figure 9-3, "Employee name:" is displayed on line 19 in column 5.

Example 9-2: Cursor Positioning

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  LOCATE.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 COL-NUM              PIC 99        VALUE 5.
PROCEDURE DIVISION.
A00-OUT-PARA.
    DISPLAY "Employee name:"     LINE 19
                                 COLUMN COL-NUM
                                 ERASE SCREEN.
    STOP RUN.
```

Figure 9-3: Positioning the Data on Line 19 Column 5



Employee name:

C81ART-20460-30

If you use LINE, but not COLUMN, data is accepted or displayed at column 1 and the specified line position.

If you use COLUMN, but not LINE, data is accepted or displayed at the current line and specified column position.

If you do not use either clause, data is accepted or displayed at the position specified by the current ACCEPT/DISPLAY rules in the *COBOL-81 Language Reference Manual.*

Note

The presence of either or both the LINE and COLUMN clauses implies NO ADVANCING.

You can use the PLUS phrase with LINE or COLUMN for relative cursor positioning. PLUS eliminates the need for counting lines or columns. Once you specify an initial LINE or COLUMN number, you can position items by using LINE PLUS or COLUMN PLUS. If you use PLUS without an integer, PLUS 1 is implied.

To get predictable results from your relative cursor positioning statements do not:

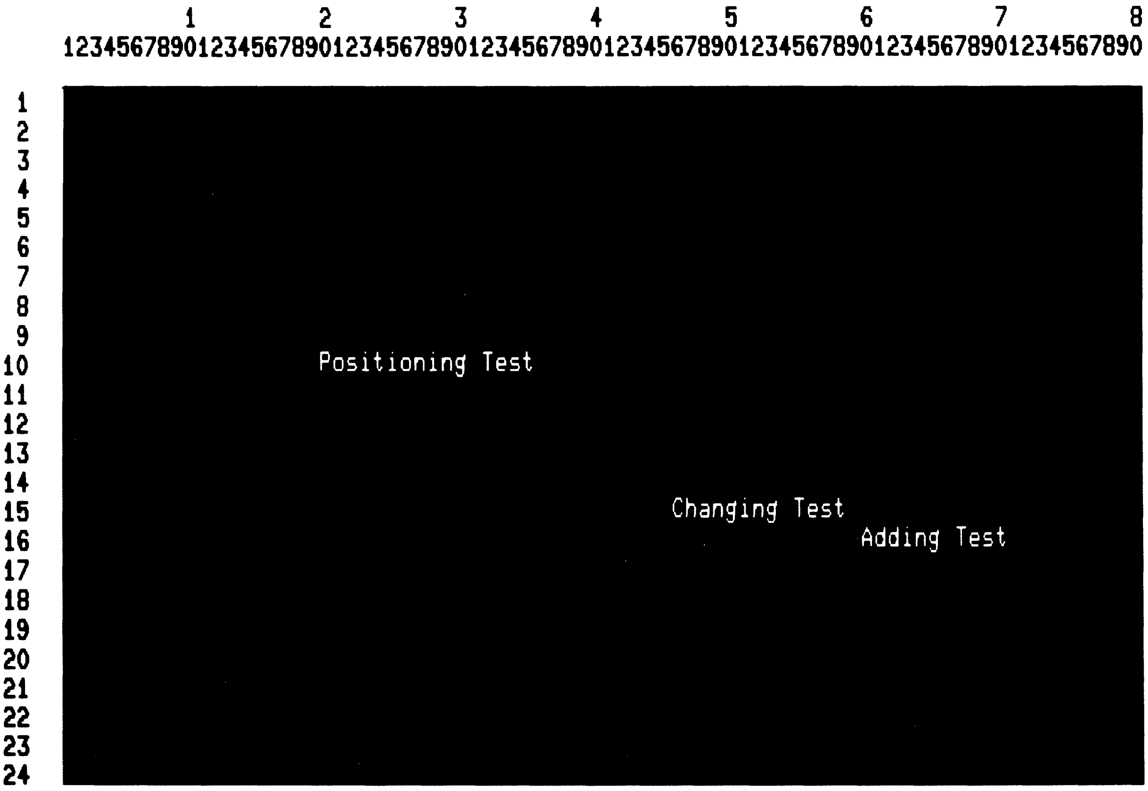
- Cause a display line to wraparound to the next line
- Accept data into unprotected fields

In Example 9-3,and as shown in Figure 9-4, PLUS is used twice to show relative positioning, once with an integer, once without.

Example 9-3: Use of PLUS for Cursor Positioning

```
IDENTIFICATION DIVISION.
PROGRAM-ID. LINEPLUS.
PROCEDURE DIVISION.
A00-BEGIN.
    DISPLAY "Positioning Test" LINE 10      COLUMN 20 ERASE SCREEN
            "Changing Test"    LINE PLUS 5  COLUMN PLUS 10
            "Adding Test"      LINE PLUS    COLUMN PLUS.
    DISPLAY " " LINE 23 COLUMN 1.
    STOP RUN.
```

Figure 9-4: Cursor Positioning Using the PLUS Option



```
Positioning Test
Changing Test
Adding Test
```

C81ART-20470-30

"Positioning Test" displays on line 10, column 20; "Changing Test" on line 15, column 46 of the form; and "Adding Test" on line 16, column 60 of the form.

Note

If you use LINE PLUS so that relative positioning goes beyond the bottom of the screen, your form scrolls with each such display.

9.1.3 Assigning Character Attributes to Your Format Entries

You can use one or more of the character attributes in Table 9-2 to highlight your screen data depending on your terminal type. Example 9-4 shows the use of these attributes in a program segment. Figure 9-5 shows the results of Example 9-4.

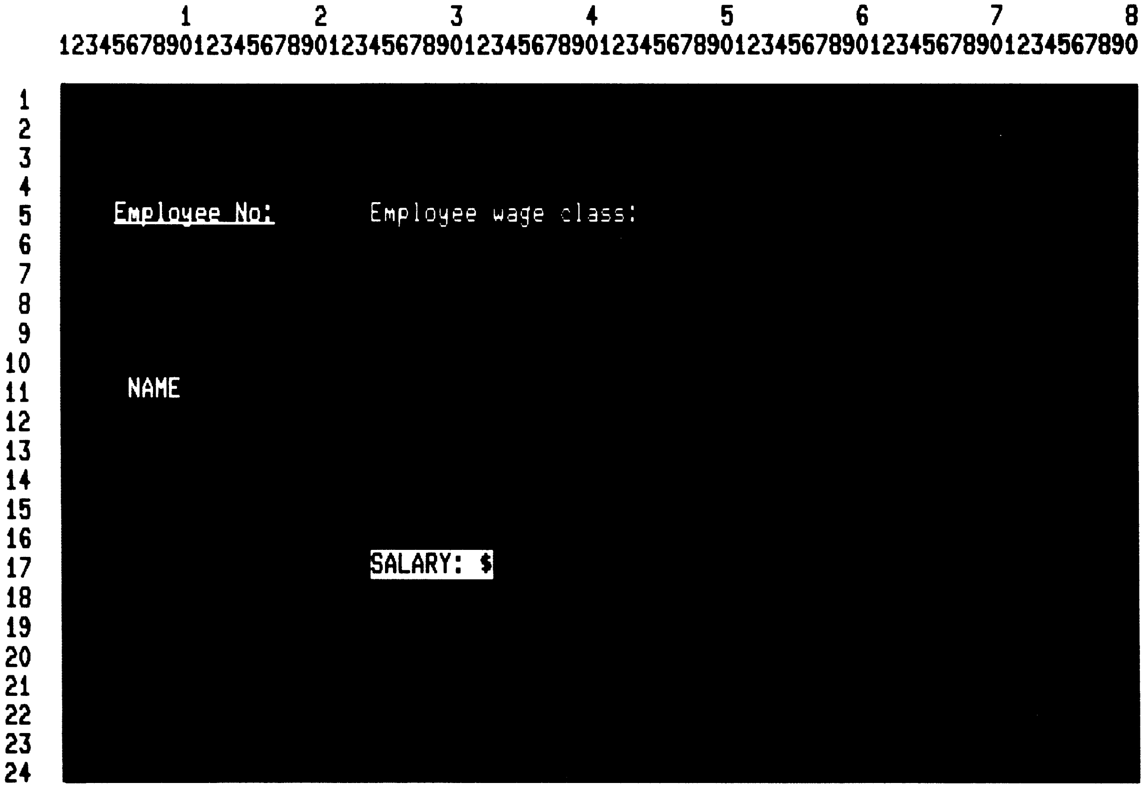
Table 9-2: Available Character Attributes by Terminal Type

Character Attribute	VT100 Family Terminals with the Advanced Video Option and the Professional Terminal	VT52 and the VT100 without the Advanced Video Option
BELL sounds your terminal bell	Available	Available
UNDERLINED underlines your text	Available	Not Available
BOLD intensifies your text	Available	Not Available
BLINKING blinks your text	Available	Not Available
REVERSED changes your screen's background	Available	Not Available

Example 9-4: Character Attributes

```
IDENTIFICATION DIVISION.
PROGRAM-ID. CHARATTR.
PROCEDURE DIVISION.
A00-BEGIN.

    DISPLAY "Employee No:" UNDERLINED LINE 5 COLUMN 5 ERASE SCREEN.

    DISPLAY "Employee wage class:" BOLD LINE 5 COLUMN 24.

    DISPLAY "NAME" BLINKING LINE PLUS 6  COLUMN 6.

    DISPLAY "SALARY: $" REVERSED LINE PLUS 6 COLUMN 24.

    DISPLAY " " LINE 23 COLUMN 1.
```

Figure 9-5: Screen Display with Character Attributes



Employee No: Employee wage class:

NAME

SALARY: $

C81ART-20480-30

9.1.4 Handling Data with ACCEPT Options

Several ACCEPT clauses help you handle data. These include the CONVERSION, ON EXCEPTION, PROTECTED, SIZE, NO ECHO, and DEFAULT clauses.

9.1.4.1 Using CONVERSION with ACCEPT Data — When you use the CONVERSION clause with an ACCEPT numeric operand, COBOL-81 converts the data entered on the form to a trailing-signed decimal field. It then moves the data from the screen to your program using standard MOVE statement rules.

When an ACCEPT operand is not numeric, CONVERSION moves the input characters as an alphanumeric string, using standard MOVE statement rules. The clause lets you accept data into an alphanumeric-edited field and permits use of the JUSTIFIED clause, if you specify it on the destination item.

If you use CONVERSION with ACCEPT numeric data, you can also use the ON EXCEPTION clause. This clause lets you control data entry errors that can occur in a numeric field with CONVERSION.

9.1.4.2 Using ON EXCEPTION When Accepting Data with CONVERSION — If you enter illegal numeric data or exceed the PICTURE description of the ACCEPT data (with an overflow to either the left or right of the decimal point), the imperative statement associated with ON EXCEPTION executes, and the destination field does not change.

Note

ON EXCEPTION has no effect when accepting data from PROTECTED fields.

In Example 9-5, and as shown in Figure 9-6, ON EXCEPTION executes if you enter an alphanumeric or a numeric item out of the valid range. The valid range prompts you to try again.

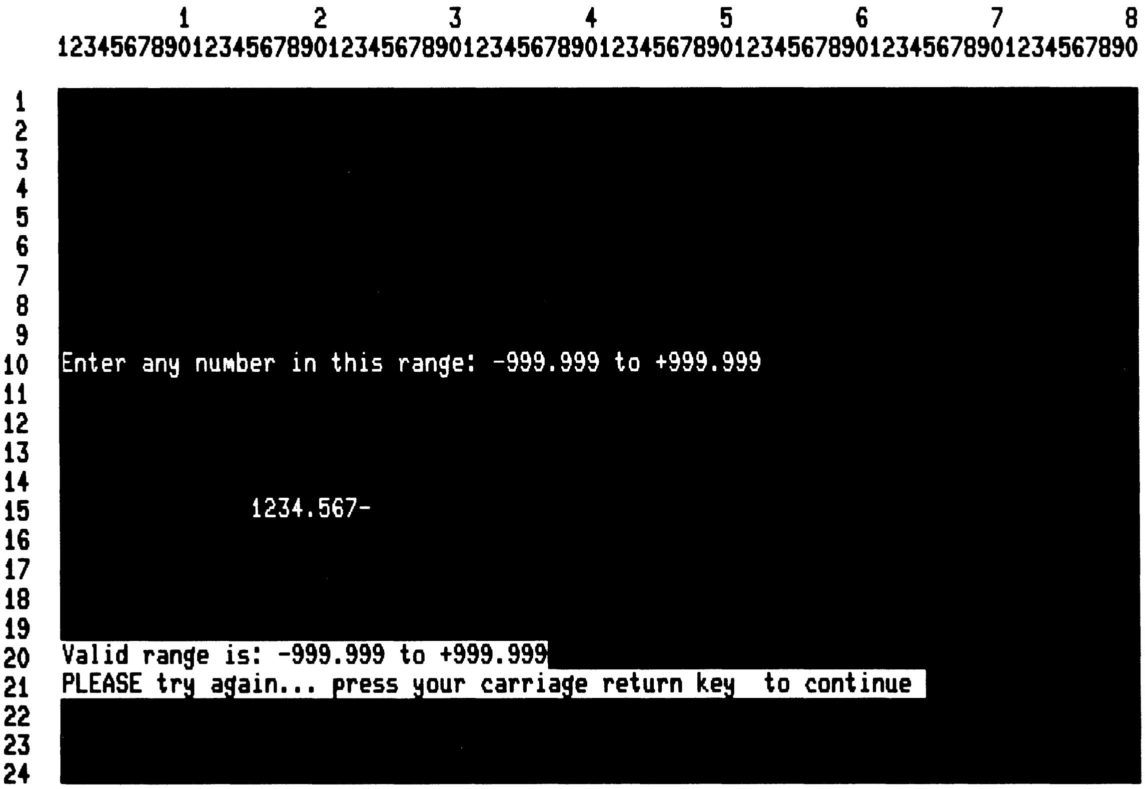
If you do not use ON EXCEPTION and an error occurs:

- The field on the screen is filled with spaces
- Automatic reprompting for the data results
- The destination field does not change

Example 9-5: Use of ON EXCEPTION

```
IDENTIFICATION DIVISION.
PROGRAM-ID. ONEXC.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  NUM-DATA   PIC S9(3)V9(3) COMP-3.
PROCEDURE DIVISION.
A00-BEGIN.
    DISPLAY "Enter any number in this range: -999.999 to +999.999"
            LINE 10 COLUMN 1
            ERASE SCREEN.
    ACCEPT NUM-DATA WITH CONVERSION LINE 15 COLUMN 15
        ON EXCEPTION
           DISPLAY "Valid range is: -999.999 to +999.999"
              LINE 20 REVERSED WITH BELL ERASE TO END OF SCREEN
           DISPLAY "PLEASE try again... press your carriage return key
-             " to continue" LINE PLUS REVERSED
           ACCEPT NUM-DATA.
    GO TO A00-BEGIN.
```

Figure 9-6: Accepting Data with the ON EXCEPTION Option



```
Enter any number in this range: -999.999 to +999.999




                1234.567-





Valid range is: -999.999 to +999.999
PLEASE try again... press your carriage return key  to continue
```

C81ART-20490-30

9.1.4.3 Protecting Your Screen — You can use the PROTECTED clause in an ACCEPT statement to put a limit on the characters to be input. This clause prevents writing or deleting parts of the screen.

If you use this clause, and you try to type past the right-most position of the input field or delete past the left edge of the input field, the terminal bell sounds and the screen cursor does not move. You can accept the data on the screen by pressing a legal terminator key or you can delete the data by pressing your delete key. For more information on legal terminator keys, refer to the CONTROL KEY phrase of the ACCEPT statement in the *COBOL-81 Language Reference Manual.*

You can also use either REVERSED, BOLD, BLINKING, or UNDERLINED with the PROTECTED clause. When you specify one of these attributes, the input field fills with spaces. This lets you see the size of the input field on the screen before you enter data. The characters you enter also echo the specified attribute.

The PROTECTED SIZE clause sets the size of the input field on the screen and allows you to change the size from the size indicated by the PICTURE clause of the destination item. Example 9-6 and Figure 9-7 show how to use the SIZE clause with the PROTECTED clause. When the example uses the SIZE 3 clause, any attempt to enter more than three characters results in ringing the terminal bell. When the example uses the SIZE 10 clause, the ACCEPT statement includes the ON EXCEPTION clause to warn you whenever you enter a number that would result in truncation at either end of the assumed decimal point. Figure 9-7 shows such an example. The operator entered a 10-digit number and exceeded the storage capacity of the data item NUM-DAT on the left side of the assumed decimal point.

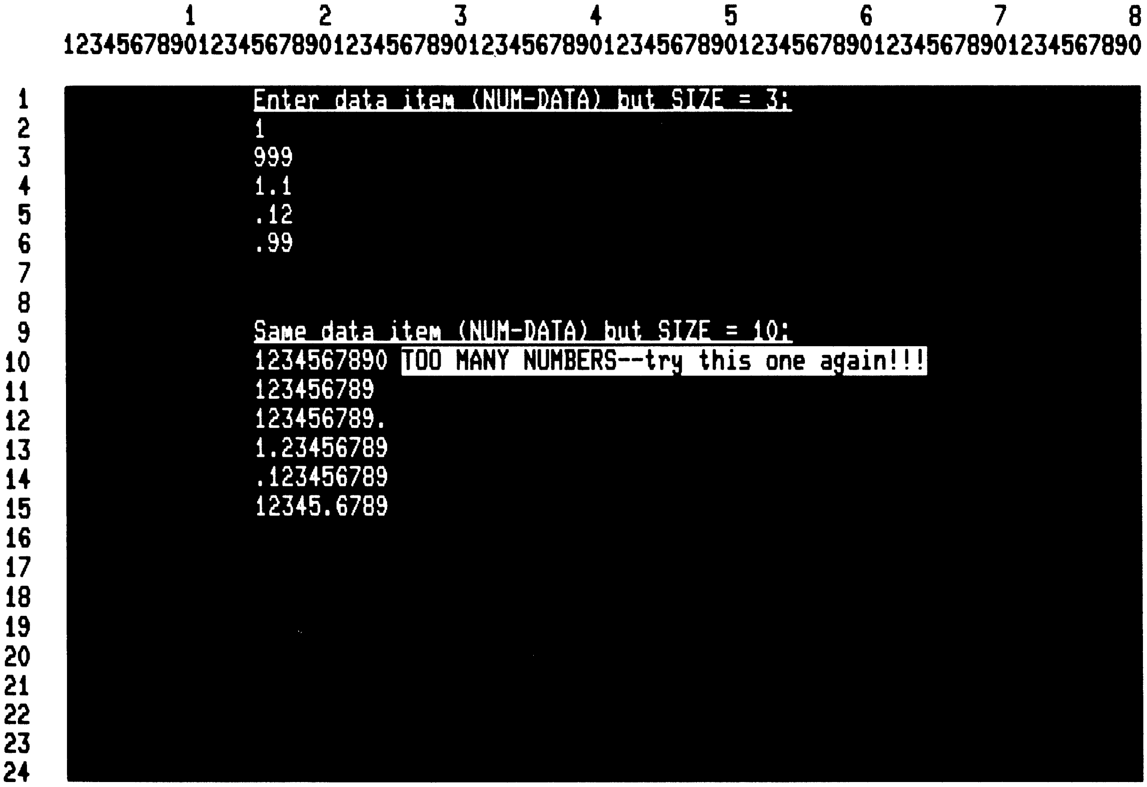
Note

The SIZE clause only controls the number of characters you can enter; it does not alter any other PICTURE clause requirements. Truncation, space or zero filling, and decimal point alignment occur according to MOVE statement rules.

Example 9-6: Use of SIZE

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  PROTECT.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  NUM-DATA  PIC S9(9)V9(9) COMP-3.

PROCEDURE DIVISION.
A00-BEGIN.

    DISPLAY "Enter data item (NUM-DATA) but SIZE = 3:"
             LINE 1 COLUMN 15
             UNDERLINED
             ERASE SCREEN.
    PERFORM ACCEPT-THREE 5 TIMES.

    DISPLAY "Same data item (NUM-DATA) but SIZE = 10:" LINE PLUS 3
             COLUMN 15
             UNDERLINED.
    PERFORM ACCEPT-TEN 5 TIMES.
    STOP RUN.

ACCEPT-THREE.
    ACCEPT NUM-DATA WITH CONVERSION PROTECTED SIZE 3
           LINE PLUS COLUMN 15.

ACCEPT-TEN.
    ACCEPT NUM-DATA WITH CONVERSION PROTECTED SIZE 10
           LINE PLUS COLUMN 15
           ON EXCEPTION
               DISPLAY "TOO MANY NUMBERS--try this one again!!!"
                        COLUMN PLUS
                        REVERSED
                        GO TO ACCEPT-TEN.
```

Figure 9-7: Screen Display of NUM-DATA Using the PROTECTED Option



```
Enter data item (NUM-DATA) but SIZE = 3:
1
999
1.1
.12
.99


Same data item (NUM-DATA) but SIZE = 10:
1234567890 TOO MANY NUMBERS--try this one again!!!
123456789
123456789.
1.23456789
.123456789
12345.6789
```

C81ART-20500-30

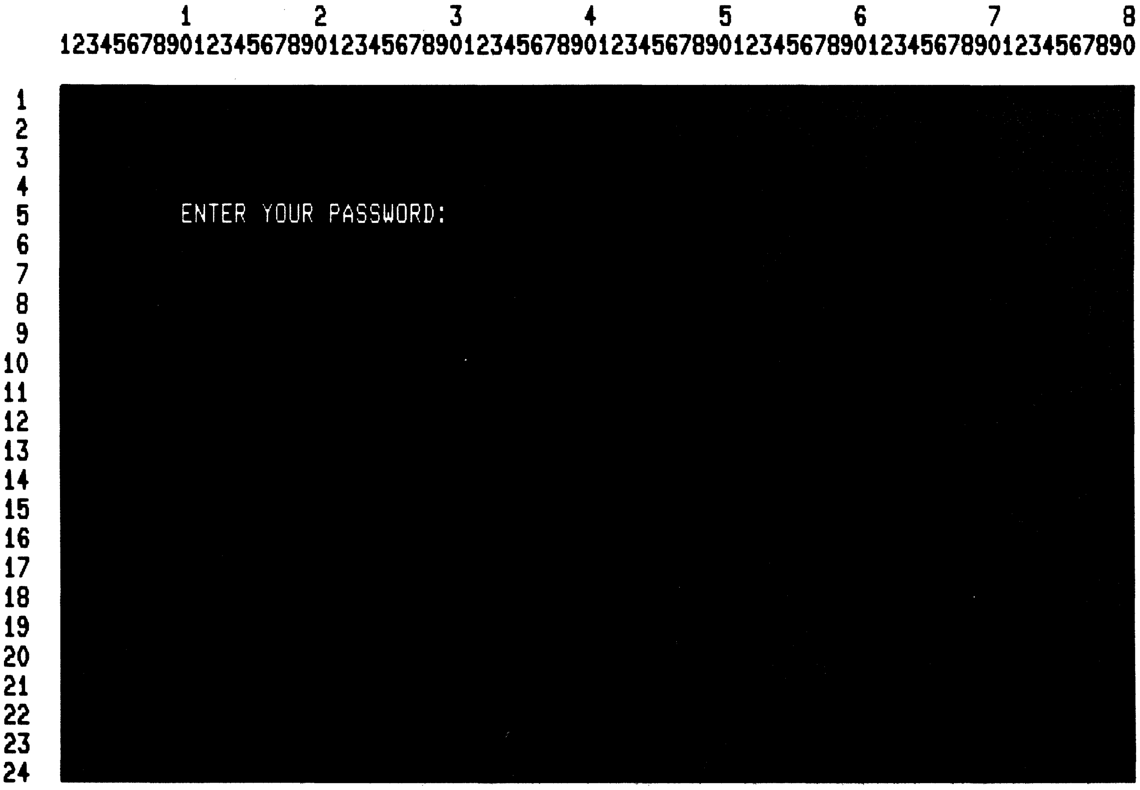
When you do not use the PROTECTED clause, the amount of data transferred is determined according to the ACCEPT statement rules in the *COBOL-81 Language Reference Manual.*

9.1.4.4 Using NO ECHO with ACCEPT Data — By default, the characters you type at the terminal are displayed on the screen. Example 9-7 and Figure 9-8 show how the NO ECHO clause prevents the input field from being displayed. The NO ECHO clause allows you to keep passwords and other information confidential.

Example 9-7: Use of NO ECHO

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  NOSHOW.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  PASSWORD PIC X(25).

PROCEDURE DIVISION.
A00-BEGIN.
    DISPLAY "ENTER YOUR PASSWORD: " LINE 5 COLUMN 10
            ERASE SCREEN.
    ACCEPT PASSWORD WITH NO ECHO.
    STOP RUN.
```

Figure 9-8: Accepting Data with the NO ECHO Option



```
ENTER YOUR PASSWORD:
```

C81ART-20510-30

9.1.4.5 Assigning Default Values to Data Fields — Use the DEFAULT clause to assign a value to an ACCEPT data item whenever:

- The program requires a value, in the cases where an operator does not have a value for the data item.

- There is a high probability that the default value is identical in most records – for example, using USA or a state's abbreviation in a mailing list.

When you use the DEFAULT clause, execution of the program proceeds as if the default value had been typed in. However, the value does not automatically display on the screen.

Example 9-8 and Figure 9-9 show you how to use the DEFAULT clause to specify default input values (the value must be an alphanumeric data name, a nonnumeric literal, or figurative constant). The example uses the TO-BE-SUPPLIED abbreviations "[TBS]", "***[TBS]****", and "+00.00" as the default values for three data items in the program.

Example 9-8: Use of the DEFAULT Clause

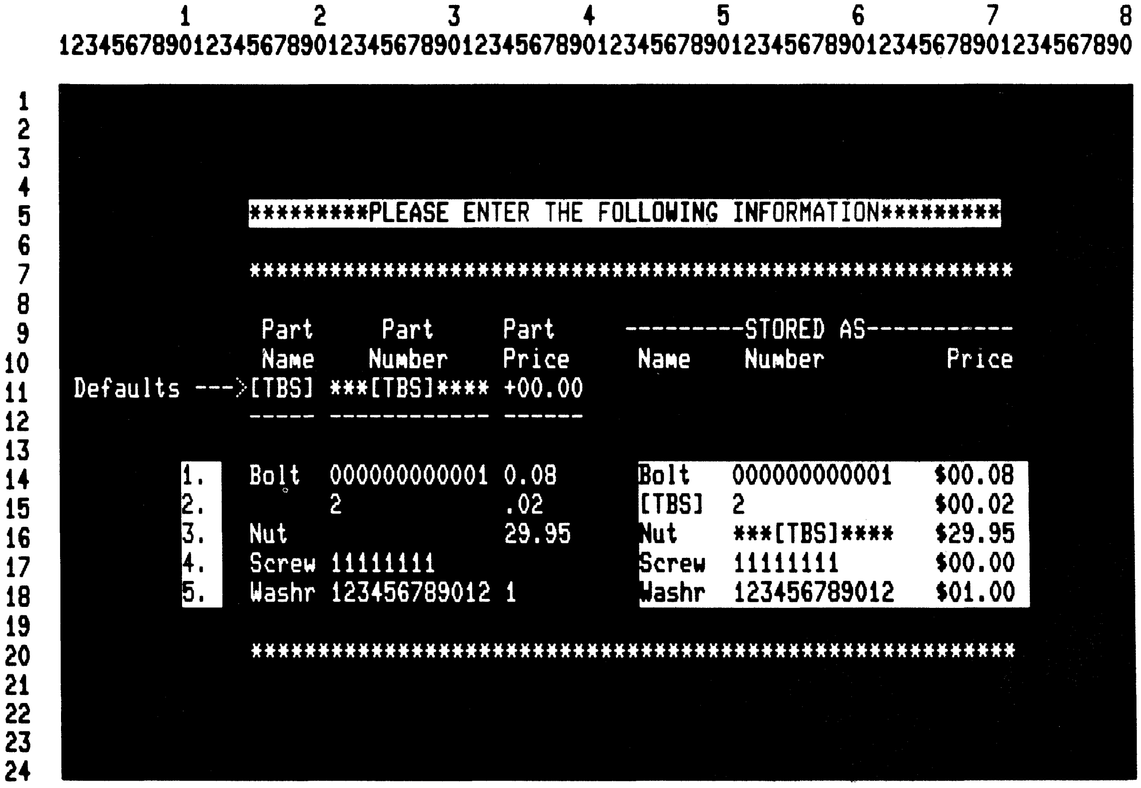
```
IDENTIFICATION DIVISION.
PROGRAM-ID. TRYDEF.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  DATA1A      PIC 9(12).
01  NAME1A      PIC XXXXX.
01  PRICEA      PIC S99V99.
01  DATA123.
    02  NAME1B  PIC XXXXX.
    02          PIC XX VALUE SPACES.
    02  DATA1B  PIC XXXXXXXXXXXX.
    02          PIC XXX VALUE SPACES.
    02  PRICEB  PIC $99.99-.
01  NAME-DEFAULT PIC XXXXX VALUE "[TBS]".
01  COL-NUM     PIC 99    VALUE 10.
PROCEDURE DIVISION.
A00-DEFAULT-TEST.

    DISPLAY "*********PLEASE ENTER THE FOLLOWING INFORMATION*********"
             LINE 5 COLUMN 15
             REVERSED BLINKING
             ERASE SCREEN.

    DISPLAY "********************************************************"
             LINE 7 COLUMN 15.

    DISPLAY " Part     Part      Part
-           "   ---------STORED AS-----------"
             LINE 9 COLUMN 15.

    DISPLAY " Name    Number    Price
-           "    Name    Number         Price "
             LINE 10 COLUMN 15.

    DISPLAY "Defaults --->[TBS] ***[TBS]**** +00.00"
             LINE 11 COLUMN 2.

    DISPLAY "----- ------------ ------"
             LINE 12 COLUMN 15.

    DISPLAY "********************************************************"
             LINE 20 COLUMN 15.

    DISPLAY "5. " REVERSED BLINKING LINE 18 COLUMN COL-NUM.
    DISPLAY "4. " REVERSED BLINKING LINE 17 COLUMN COL-NUM.
    DISPLAY "3. " REVERSED BLINKING LINE 16 COLUMN COL-NUM.
    DISPLAY "2. " REVERSED BLINKING LINE 15 COLUMN COL-NUM.
    DISPLAY "1. " REVERSED BLINKING LINE 14 COLUMN COL-NUM.
    DISPLAY " " LINE 13 COLUMN 15.

    PERFORM A05-GET-DATA 5 TIMES.

    DISPLAY " " LINE 22 COLUMN 1.

    STOP RUN.
```

(continued on next page)

Example 9-8: Use of the DEFAULT Clause (Cont.)

```
A05-GET-DATA.

    ACCEPT NAME1A
           PROTECTED
           DEFAULT NAME-DEFAULT
           LINE PLUS COLUMN 15 ERASE TO END OF LINE.

    ACCEPT DATA1A
           PROTECTED
           DEFAULT "***[TBS]****"
           COLUMN 21.

    ACCEPT PRICEA
           PROTECTED
           WITH CONVERSION
           DEFAULT ZERO
           COLUMN 34.

    MOVE NAME1A TO NAME1B.
    MOVE DATA1A TO DATA1B.
    MOVE PRICEA TO PRICEB.
    DISPLAY DATA123 REVERSED COLUMN 44.
```

Figure 9-9: Accepting Data with the DEFAULT Option



```




              *********PLEASE ENTER THE FOLLOWING INFORMATION*********

              *********************************************************

               Part     Part     Part   ---------STORED AS-----------
               Name    Number    Price     Name   Number         Price
 Defaults --->[TBS] ***[TBS]**** +00.00
              ----- ------------ ------

         1.   Bolt  000000000001 0.08      Bolt   000000000001   $00.08
         2.         2            .02       [TBS]  2              $00.02
         3.   Nut                29.95     Nut    ***[TBS]****   $29.95
         4.   Screw 11111111               Screw  11111111       $00.00
         5.   Washr 123456789012 1         Washr  123456789012   $01.00

              *********************************************************
```

C81ART-20520-30

9.1.5 Using Keys on Your Terminal to Define Special Program Functions

Use the CONTROL KEY IN clause of the ACCEPT statement to tailor your screen handling programs to give special meanings to any or all of these keys on your terminal:

- Cursor positioning keys (cursor up, down, left, right)
- Program function keys (PF1, PF2, PF3, and PF4)
- Auxiliary keypad keys (if in application mode) 0 through 9, "–" (minus), "," (comma), "." (period), and ENTER
- Top row function keys and editing keys – for Professional terminals

The CONTROL KEY IN clause gives your program a way to recognize these keys whenever the program allows you to use one. Tables 9-3 and 9-4 list the characters returned by the OTS to the data-name specified in the CONTROL KEY IN clause. Table 9-3 is for VT52, VT100, and Professional terminals. Table 9-4 shows additional keys available on Professional terminals only. Example 9-9 shows you how to define and use the data-name in the CONTROL KEY IN clause. It also shows you how to access your alternate keypad keys (see paragraph P0).

Table 9-3: COBOL-81 Characters Returned by the OTS for Cursor Positioning, Program Function, and Auxiliary Keypad Keys

	Terminal Types			Characters Returned by the OTS in the Data-Name Specified by CONTROL KEY IN	
Key Name – "Keypad Name"	VT52	Professional 7-bit and VT100	Professional 8-bit	First	Remaining
Cursor up – "UP"	ESC A	ESC [A	CSI A	CSI	A
Cursor down – "DOWN"	ESC B	ESC [B	CSI B	CSI	B
Cursor right – "RIGHT"	ESC C	ESC [C	CSI C	CSI	C
Cursor left – "LEFT"	ESC D	ESC [D	CSI D	CSI	D
Program function – "PF1"		ESC OP	SS3 P	SS3	P
Program function – "PF2"		ESC OQ	SS3 Q	SS3	Q
Program function – "PF3"		ESC OR	SS3 R	SS3	R
Program function – "PF4"		ESC OS	SS3 S	SS3	S
Auxiliary keypad – left blank	ESC P			SS3	P
Auxiliary keypad – center blank	ESC Q			SS3	Q
Auxiliary keypad – right blank	ESC R			SS3	R
Auxiliary keypad – "0"	ESC ?p	ESC Op	SS3 p	SS3	p
Auxiliary keypad – "1"	ESC ?q	ESC Oq	SS3 q	SS3	q
Auxiliary keypad – "2"	ESC ?r	ESC Or	SS3 r	SS3	r
Auxiliary keypad – "3"	ESC ?s	ESC Os	SS3 s	SS3	s
Auxiliary keypad – "4"	ESC ?t	ESC Ot	SS3 t	SS3	t
Auxiliary keypad – "5"	ESC ?u	ESC Ou	SS3 u	SS3	u
Auxiliary keypad – "6"	ESC ?v	ESC Ov	SS3 v	SS3	v
Auxiliary keypad – "7"	ESC ?w	ESC Ow	SS3 w	SS3	w
Auxiliary keypad – "8"	ESC ?x	ESC Ox	SS3 x	SS3	x
Auxiliary keypad – "9"	ESC ?y	ESC Oy	SS3 y	SS3	y

(continued on next page)

Table 9-3: COBOL-81 Characters Returned by the OTS for Cursor Positioning, Program Function, and Auxiliary Keypad Keys (Cont.)

	Terminal Types			Characters Returned by the OTS in the Data-Name Specified by CONTROL KEY IN	
Key Name – "Keypad Name"	VT52	Professional 7-bit and VT100	Professional 8-bit	First	Remaining
Auxiliary keypad – "–"		ESC Om	SS3 m	SS3	m
Auxiliary keypad – ","		ESC Ol	SS3 l	SS3	l
Auxiliary keypad – "."	ESC ?n	ESC On	SS3 n	SS3	n
Auxiliary keypad – "ENTER"	ESC ?M	ESC OM	SS3 M	SS3	M
CTRL/Z – "CTRL/Z"	26	26	26	26	
TAB – "TAB"	9	9	9	9	
RETURN – "RETURN"	13	13	13	13	

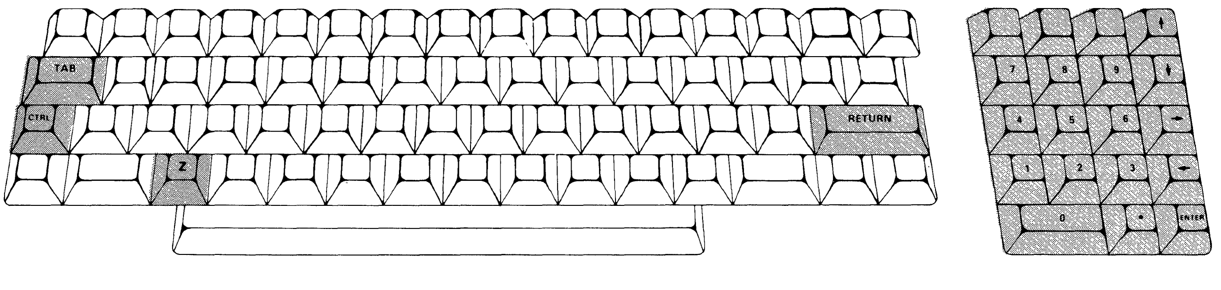
Note: At the present time the CSI and SS3 character is shown for your information only. You need not check for their presence because the remaining characters are unique and need no qualification.

For your information, the definition and value of the CSI and SS3 character used in Table 9-3 and Table 9-4 are shown below:

```
01  SS3X                  PIC 9999 COMP VALUE 143.
01  SS3 REDEFINES SS3X    PIC X.

01  CSIX                  PIC 9999 COMP VALUE 155.
01  CSI REDEFINES CSIX    PIC X.
```

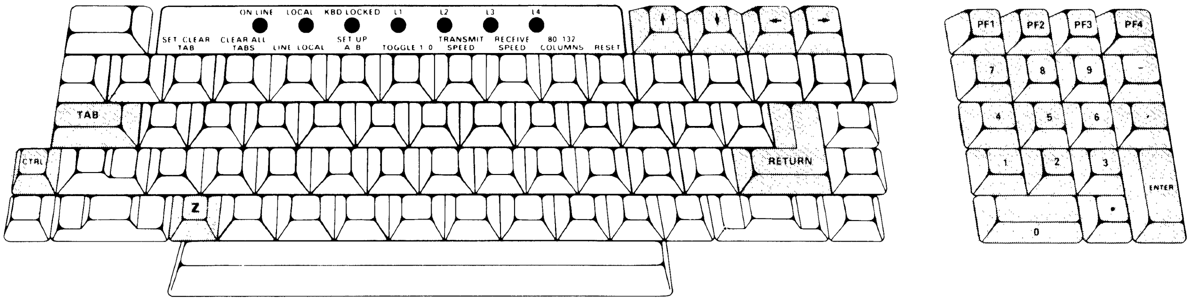
Figures 9-10, 9-11, and 9-12 are the standard keypads for the VT52, VT100, and Professional terminals, respectively. The nonblank keys correspond to the keypad names in Tables 9-3 and 9-4, which list the characters returned to the application program by the OTS when pressed.

Figure 9-10: COBOL-81's Control Keys on the Standard VT52 Keypad



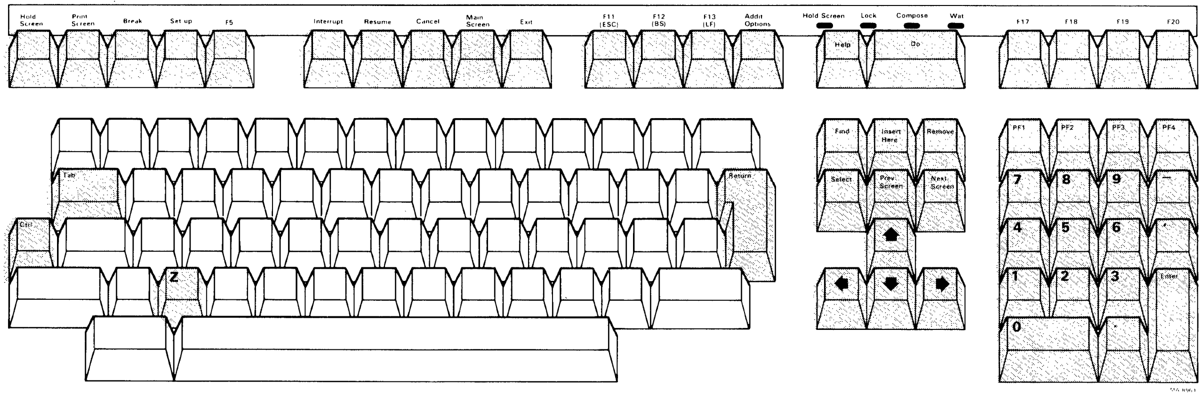
C81ART-20530-15

Figure 9-11: COBOL-81's Control Keys on the Standard VT100 Keypad



C81ART-20540-15

Figure 9-12: COBOL-81's Control Keys on the Standard Professional Keypad



C81ART-20550-15

Table 9-4: Characters Returned by the OTS for the Professional's Top Row Function and Editing Keys

	Terminal Types			***Characters Returned by the OTS in the Data-Name Specified by CONTROL KEY IN	
Generic Key Name	VT52 and VT100	Professional 7-bit	Professional 8-bit	First	Remaining
F1					
F2					
F3		ESC [13~	CSI 13~	CSI	13~
F4		ESC [14~	CSI 14~	CSI	14~
F5		ESC [15~	CSI 15~	CSI	15~
F6					
F7		ESC [18~	CSI 18~	CSI	18~
F8		ESC [19~	CSI 19~	CSI	19~
F9		ESC [20~	CSI 20~	CSI	20~
F10		ESC [21~	CSI 21~	CSI	21~

(continued on next page)

Table 9-4: Characters Returned by the OTS for the Professional's Top Row Function and Editing Keys (Cont.)

	Terminal Types			***Characters Returned by the OTS in the Data-Name Specified by CONTROL KEY IN	
Generic Key Name	VT52 and VT100	Professional 7-bit	Professional 8-bit	First	Remaining
F11		ESC [23~	CSI 23~	CSI	23~
F12		ESC [24~	CSI 24~	CSI	24~
F13		ESC [25~	CSI 25~	CSI	25~
F14		ESC [26~	CSI 26~	CSI	26~
F15		ESC [28~	CSI 28~	CSI	28~
F16		ESC [29~	CSI 29~	CSI	29~
F17		ESC [31~	CSI 31~	CSI	31~
F18		ESC [32~	CSI 32~	CSI	32~
F19		ESC [33~	CSI 33~	CSI	33~
F20		ESC [34~	CSI 34~	CSI	34~
Editing key – Find		ESC [1~	CSI 1~	CSI	1~
Editing key – Insert Here		ESC [2~	CSI 2~	CSI	2~
Editing key – Remove		ESC [3~	CSI 3~	CSI	3~
Editing key – Select		ESC [4~	CSI 4~	CSI	4~
Editing key – Prev Screen		ESC [5~	CSI 5~	CSI	5~
Editing key – Next Screen		ESC [6~	CSI 6~	CSI	6~

*** If your system allows your application to reference this key

Note: At the present time the CSI character is shown for your information only. You need not check for their presence because the remaining characters are unique and need no qualification.

Example 9-9 and Figure 9-13 show you how to use the CONTROL KEY clause to handle arrow keys, program function keys, auxiliary keypad keys, CTRL/Z, TAB, and RETURN using a VT100 terminal on an RSX-11M/M-PLUS system. The example also includes information for changes needed for other terminals and operating systems.

When you use this clause, you allow PF keys or arrow keys, as well as RETURN and TAB to terminate input. Also, this clause permits you to use those keys to move the cursor around the screen and make menu selections without typing any data on the screen.

Note

To activate the auxiliary keypad, your program must execute DISPLAY ESC "=". You must additionally define ESC as the escape character. Refer to the following examples.

In Example 9-9, the terminator key codes display on the screen. Figure 9-13 shows a sample run using the right arrow terminal key.

Example 9-9: Use of the CONTROL KEY Clause

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  SPECIAL.
DATA DIVISION.
WORKING-STORAGE SECTION.

*
*      The code returned will be the same on VT52,
*      VT100, or CT terminals.
*
01 CONTROL-KEY.
   02  FIRST-CHAR-CONTROL-KEY  PIC X.
   02  REMAINING-CHAR-CONTROL-KEY PIC XXXX.
        88 UP-ARROW        VALUE "A".
        88 DOWN-ARROW      VALUE "B".
        88 RIGHT-ARROW     VALUE "C".
        88 LEFT-ARROW      VALUE "D".
        88 PF1             VALUE "P".
        88 PF2             VALUE "Q".
        88 PF3             VALUE "R".
        88 PF4             VALUE "S".
        88 AUX0            VALUE "p".
        88 AUX1            VALUE "q".
        88 AUX2            VALUE "r".
        88 AUX3            VALUE "s".
        88 AUX4            VALUE "t".
        88 AUX5            VALUE "u".
        88 AUX6            VALUE "v".
        88 AUX7            VALUE "w".
        88 AUX8            VALUE "x".
        88 AUX9            VALUE "y".
        88 AUXMINUS        VALUE "m".
        88 AUXCOMMA        VALUE "l".
        88 AUXPERIOD       VALUE "n".
        88 AUXENTER        VALUE "M".
*
*
* For the Professional:
*
*       88 F1              VALUE "13~".
*       88 F2              VALUE "14~".
*                .
*                .
*                .
*       88 F20             VALUE "34~".
*       88 EK-FIND         VALUE "1~".
*                .
*                .
*                .
*       88 EK-NEXT         VALUE "6~".
*
*
* For RSX      ESCAPE value is 27.
* For RSTS/E   ESCAPE value is 155.
*
01      TAB-KEY                  PIC 9999 COMP VALUE 9.
01  TAB REDEFINES TAB-KEY        PIC X.
01      CARRIAGE-RETURN          PIC 9999 COMP VALUE 13.
01  CR REDEFINES CARRIAGE-RETURN PIC X.
01      CZ                       PIC 9999 COMP VALUE 26.
01  CTRL-Z REDEFINES CZ          PIC X.
01      ESCAPE                   PIC 9999 COMP VALUE 27.
01  ESC REDEFINES ESCAPE         PIC X.
01      SS3X                     PIC 9999 COMP VALUE 143.
```

(continued on next page)

Example 9-9: Use of the CONTROL KEY Clause (Cont.)

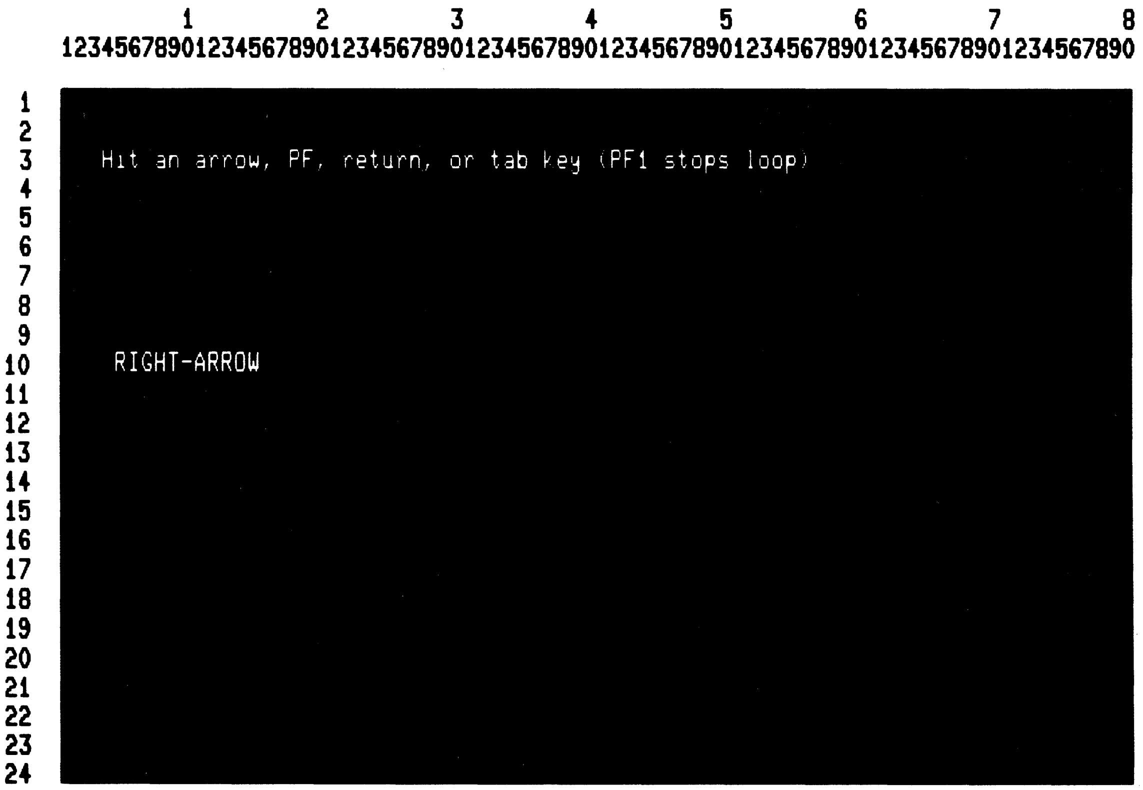
```
01   SS3 REDEFINES SS3X           PIC X.
01       CSIX                     PIC 9999 COMP VALUE 155.
01   CSI REDEFINES CSIX           PIC X.

PROCEDURE DIVISION.
P0.
*
* DISPLAY ESC "=" puts you in alternate keypad mode
*

    DISPLAY ESC "=".
    DISPLAY " "  ERASE SCREEN.

P1.
    DISPLAY "Hit an arrow, PF, return, or tab key (PF1 stops loop)"
             LINE 3 COLUMN 4.

    ACCEPT CONTROL KEY IN CONTROL-KEY.

    IF CR = FIRST-CHAR-CONTROL-KEY
       DISPLAY "RETURN" LINE 10 COLUMN 5 ERASE LINE GO TO P1.

    IF TAB = FIRST-CHAR-CONTROL-KEY
       DISPLAY "TAB" LINE 10 COLUMN 5 ERASE LINE GO TO P1.

    IF CTRL-Z = FIRST-CHAR-CONTROL-KEY
       DISPLAY "CTRL/Z" LINE 10 COLUMN 5 ERASE LINE GO TO P1.

    IF PF1 DISPLAY "PF1" LINE 10 COLUMN 5 ERASE LINE GO TO P2.

    IF PF2 DISPLAY "PF2" LINE 10 COLUMN 5 ERASE LINE GO TO P1.

    IF PF3 DISPLAY "PF3" LINE 10 COLUMN 5 ERASE LINE GO TO P1.

    IF PF4 DISPLAY "PF4" LINE 10 COLUMN 5 ERASE LINE GO TO P1.

    IF UP-ARROW DISPLAY "UP-ARROW" LINE 10 COLUMN 5 ERASE LINE
       GO TO P1.

    IF DOWN-ARROW DISPLAY "DOWN-ARROW" LINE 10 COLUMN 5
       ERASE LINE GO TO P1.

    IF LEFT-ARROW DISPLAY "LEFT-ARROW" LINE 10 COLUMN 5
       ERASE LINE GO TO P1.

    IF RIGHT-ARROW DISPLAY "RIGHT-ARROW" LINE 10 COLUMN 5
       ERASE LINE GO TO P1.

    IF AUX0 DISPLAY "AUXILIARY KEYPAD 0" LINE 10 COLUMN 5
       ERASE LINE GO TO P1.

    IF AUX1 DISPLAY "AUXILIARY KEYPAD 1" LINE 10 COLUMN 5
       ERASE LINE GO TO P1.

    IF AUX2 DISPLAY "AUXILIARY KEYPAD 2" LINE 10 COLUMN 5
       ERASE LINE GO TO P1.

    IF AUX3 DISPLAY "AUXILIARY KEYPAD 3" LINE 10 COLUMN 5
       ERASE LINE GO TO P1.

    IF AUX4 DISPLAY "AUXILIARY KEYPAD 4" LINE 10 COLUMN 5
       ERASE LINE GO TO P1.
```

(continued on next page)

Example 9-9: Use of the CONTROL KEY Clause (Cont.)

```
    IF AUX5 DISPLAY "AUXILIARY KEYPAD 5" LINE 10 COLUMN 5
       ERASE LINE GO TO P1.

    IF AUX6 DISPLAY "AUXILIARY KEYPAD 6" LINE 10 COLUMN 5
       ERASE LINE GO TO P1.

    IF AUX7 DISPLAY "AUXILIARY KEYPAD 7" LINE 10 COLUMN 5
       ERASE LINE GO TO P1.

    IF AUX8 DISPLAY "AUXILIARY KEYPAD 8" LINE 10 COLUMN 5
       ERASE LINE GO TO P1.

    IF AUX9 DISPLAY "AUXILIARY KEYPAD 9" LINE 10 COLUMN 5
       ERASE LINE GO TO P1.

    IF AUXMINUS DISPLAY "AUXILIARY KEYPAD -" LINE 10 COLUMN 5
       ERASE LINE GO TO P1.

    IF AUXCOMMA DISPLAY "AUXILIARY KEYPAD ," LINE 10 COLUMN 5
       ERASE LINE GO TO P1.

    IF AUXPERIOD DISPLAY "AUXILIARY KEYPAD ." LINE 10 COLUMN 5
       ERASE LINE GO TO P1.

    IF AUXENTER DISPLAY "AUXILIARY KEYPAD ENTER" LINE 10 COLUMN 5
       ERASE LINE GO TO P1.

    DISPLAY "Not an allowable control key -
-           "press your return key to continue"
            LINE 10 COLUMN 5
            WITH BELL ERASE LINE.
    ACCEPT CONTROL-KEY.
    GO TO P1.
P2.
    DISPLAY "Press your carriage return key to end this job"
            LINE 11 COLUMN 5 ERASE LINE.
    ACCEPT CONTROL KEY IN CONTROL-KEY LINE 12 COLUMN 5 ERASE LINE.
    IF CR NOT = FIRST-CHAR-CONTROL-KEY GO TO P0
       ELSE
          DISPLAY "END OF JOB" LINE 13 COLUMN 35
                   BOLD BLINKING REVERSED BELL
                   ERASE SCREEN
                   STOP RUN.
```

Figure 9-13: Screen Display of Program SPECIAL



```


   Hit an arrow, PF, return, or tab key (PF1 stops loop)






    RIGHT-ARROW
```

C81ART-20560-30

To expand upon Example 9-9, you can, for example, accept data in addition to specifying the CONTROL KEY clause. What you get in this case is the ability to accept data and the ability to determine what to do next. You can use the CONTROL KEY clause to move the cursor around on the screen or take a specific course of action.

9.1.6 Using the CONVERSION Clause to Display Data

Use the CONVERSION clause to display the contents of numeric fields. When you use the CONVERSION clause with a DISPLAY statement, the numeric item appears on the screen:

- In DISPLAY usage
- With a decimal point (if needed)
- With a sign (if needed)

This lets you see the values of COMP and COMP-3 data items in human-readable form. The size of the displayed field is determined from the PICTURE clause of the displayed item. Example 9-10 and Figure 9-14 shows how to display the different types of data with the CONVERSION clause.

Example 9-10: Use of CONVERSION

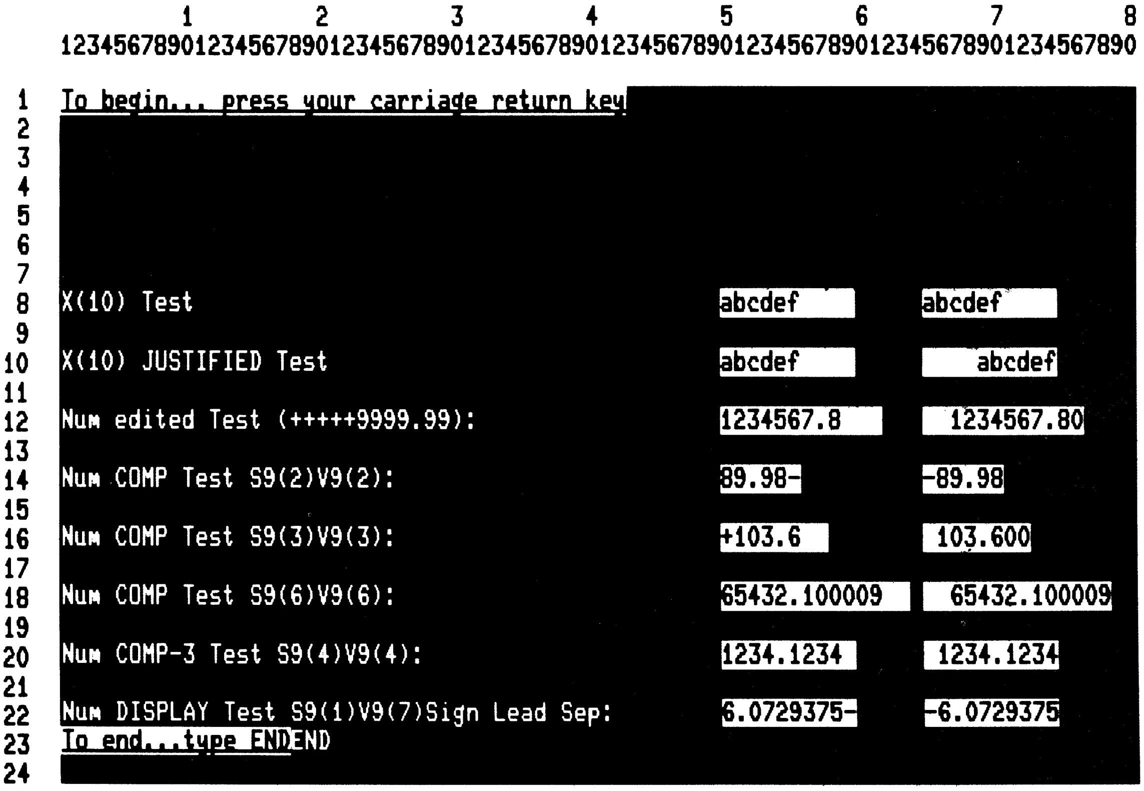
```
IDENTIFICATION DIVISION.
PROGRAM-ID.  CONVERT.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  DATA1A     PIC X(10).
01  DATA1B     PIC X(10) JUST.
01  DATA2      PIC +++++9999.99.
01  DATA3      PIC S9(2)V9(2) COMP.
01  DATA4      PIC S9(3)V9(3) COMP.
01  DATA5      PIC S9(6)V9(6) COMP.
01  DATA6      PIC S9(4)V9(4) COMP-3.
01  DATA7      PIC S9(1)V9(7) SIGN LEADING SEPARATE.

PROCEDURE DIVISION.
CONVERT-CHECK SECTION.
P1.
    DISPLAY "To begin... press your carriage return key"
            LINE 1 COLUMN 1 ERASE SCREEN
            BELL UNDERLINED REVERSED.
    ACCEPT DATA1A.

    DISPLAY "X(10) Test" LINE 8 ERASE LINE.
    ACCEPT DATA1A WITH CONVERSION PROTECTED REVERSED
           LINE 8 COLUMN 50.
    DISPLAY DATA1A REVERSED WITH CONVERSION
            LINE 8 COLUMN 65.

    DISPLAY "X(10) JUSTIFIED Test" LINE 10 ERASE LINE.
    ACCEPT DATA1B WITH CONVERSION PROTECTED REVERSED
           LINE 10 COLUMN 50.
    DISPLAY DATA1B REVERSED WITH CONVERSION
            LINE 10 COLUMN 65.
P2.
    DISPLAY "Num edited Test (+++++9999.99):" LINE 12 ERASE LINE.
    ACCEPT DATA2 PROTECTED REVERSED WITH CONVERSION
           LINE 12 COLUMN 50.
    DISPLAY DATA2 REVERSED WITH CONVERSION
            LINE 12 COLUMN 65.
P3.
    DISPLAY "Num COMP Test S9(2)V9(2):" LINE 14 ERASE LINE.
    ACCEPT DATA3 PROTECTED REVERSED WITH CONVERSION
           LINE 14 COLUMN 50.
    DISPLAY DATA3 REVERSED WITH CONVERSION LINE 14 COLUMN 65.
P4.
    DISPLAY "Num COMP Test S9(3)V9(3):" LINE 16 ERASE LINE.
    ACCEPT DATA4 PROTECTED REVERSED WITH CONVERSION
           LINE 16 COLUMN 50.
    DISPLAY DATA4 REVERSED WITH CONVERSION
            LINE 16 COLUMN 65.
P5.
    DISPLAY "Num COMP Test S9(6)V9(6):" LINE 18 ERASE LINE.
    ACCEPT DATA5 PROTECTED REVERSED WITH CONVERSION
           LINE 18 COLUMN 50.
    DISPLAY DATA5 REVERSED WITH CONVERSION
            LINE 18  COLUMN 65.
P6.
    DISPLAY "Num COMP-3 Test S9(4)V9(4):" LINE 20 ERASE LINE.
    ACCEPT DATA6 PROTECTED REVERSED WITH CONVERSION
           LINE 20 COLUMN 50.
    DISPLAY DATA6 REVERSED WITH CONVERSION
            LINE 20 COLUMN 65.
```

(continued on next page)

Example 9-10: Use of CONVERSION (Cont.)

```
P7.
    DISPLAY "Num DISPLAY Test S9(1)V9(7)Sign Lead Sep:"
           LINE 22 ERASE LINE.
    ACCEPT DATA7 PROTECTED REVERSED WITH CONVERSION
           LINE 22 COLUMN 50.
    DISPLAY DATA7 REVERSED WITH CONVERSION
            LINE 22 COLUMN 65.
P8.
    DISPLAY "To end...type END"
            LINE PLUS COLUMN 1 ERASE LINE
            BELL UNDERLINED REVERSED.

    ACCEPT DATA1A.
    IF DATA1A = "END" STOP RUN.
    GO TO P1.
```

Figure 9-14: Sample Run of Program CONVERT



```
            1         2         3         4         5         6         7         8
   12345678901234567890123456789012345678901234567890123456789012345678901234567890

 1 To begin... press your carriage return key
 2
 3
 4
 5
 6
 7
 8 X(10) Test                                       abcdef         abcdef
 9
10 X(10) JUSTIFIED Test                             abcdef             abcdef
11
12 Num edited Test (+++++9999.99):                  1234567.8       1234567.80
13
14 Num COMP Test S9(2)V9(2):                        89.98-         -89.98
15
16 Num COMP Test S9(3)V9(3):                        +103.6          103.600
17
18 Num COMP Test S9(6)V9(6):                        65432.100009    65432.100009
19
20 Num COMP-3 Test S9(4)V9(4):                      1234.1234       1234.1234
21
22 Num DISPLAY Test S9(1)V9(7)Sign Lead Sep:        6.0729375-     -6.0729375
23 To end...type ENDEND
24
```

C81ART-20570-30

Chapter 10
Sorting Records and Merging Files

The SORT and MERGE statements provide a wide range of sorting capabilities and options. You declare the sort file or merge file with a SELECT statement in the Environment Division. Use an SD entry in the Data Division to define the characteristics of these files.

This chapter presents and explains examples showing how to use the phrases of the SORT and MERGE statements.

Note

Use MERGE statement phrases the same way you use equivalent SORT phrases.

10.1 ASCENDING and DESCENDING KEY Phrases

Use these phrases to specify the parameters for your sort keys. Establish your sort hierarchy by specifying your major sort key first, your intermediate sort key(s) next, and your minor sort key last.

In Example 10-1, SORT-KEY-1 is specified first. It is the major key and will be sorted in ascending order. SORT-KEY-2 is specified second. It is the intermediate key and will also be sorted in ascending order. The minor key, SORT-KEY-3, is specified last and will be sorted in descending order.

Example 10-1: Using the ASCENDING KEY and DESCENDING KEY Phrases

```
ASCENDING  KEY SORT-KEY-1 SORT-KEY-2
DESCENDING KEY SORT-KEY-3
```

10.2 USING and GIVING Phrases

If you only need to resequence a file – that is, you do not need to manipulate data before and after a sort – use the USING and GIVING phrases of the SORT statement. The USING phrase opens the input file and then reads and releases its records to the sort. The GIVING phrase opens and writes sorted records to the output file. In Example 10-2, the SORT phrases:

1. Open INPUT-FILE
2. Read all records in INPUT-FILE and release them to the sort
3. Sort the records in ascending sequence using the data in SORT-KEY-1
4. Open the output file and write the sorted records to OUTPUT-FILE
5. Close all the files used in the SORT statement

Example 10-2: Using the USING and GIVING Phrases

```
SORT SORT-FILE ON ASCENDING KEY SORT-KEY-1
               USING INPUT-FILE
               GIVING OUTPUT-FILE.
```

10.3 INPUT PROCEDURE and OUTPUT PROCEDURE Phrases

You can manipulate data before and after a sort with the INPUT PROCEDURE and OUTPUT PROCEDURE phrases.

INPUT PROCEDURE replaces the USING phrase when you want to manipulate data entering the sort. The SORT statement transfers control to the sections named in the INPUT PROCEDURE phrase. You then use COBOL-81 statements to open and read files and to manipulate data. Use the RELEASE statement to tranfer records to the sort. After the last statement of the input procedure is executed, the records are sorted.

The SORT statement then transfers control to the sections named in the OUTPUT PROCEDURE phrase. This phrase replaces the GIVING phrase when you want to manipulate data in the sort. You can use COBOL-81 statements to open files and manipulate data. Use the RETURN statement to transfer records from the sort.

Note

You cannot access records released to the sort file after the SORT statement ends.

You cannot transfer control directly to the INPUT PROCEDURE or OUTPUT PROCEDURE sections from a GO TO or PERFORM statement executed in another part of your program. Example 10-3 shows the use of the INPUT PROCEDURE and OUTPUT PROCEDURE phrases.

Example 10-3: Using the INPUT PROCEDURE and OUTPUT PROCEDURE Phrases

```
PROCEDURE DIVISION.
000-SORT SECTION.
010-DO-THE-SORT.
    SORT SORT-FILE ON ASCENDING KEY SORT-KEY-1
                   DESCENDING KEY SORT-KEY-2
                   INPUT PROCEDURE IS 050-RETRIEVE-INPUT
                                 THRU 100-DONE-INPUT
                   OUTPUT PROCEDURE IS 200-WRITE-OUTPUT
                                  THRU 230-DONE-OUTPUT.
    DISPLAY "END OF SORT".
    STOP RUN.
050-RETRIEVE-INPUT SECTION.
060-OPEN-INPUT.
    OPEN INPUT IN-FILE.
070-READ-INPUT.
    READ IN-FILE AT END
        CLOSE IN-FILE
        GO TO 100-DONE-INPUT.
************************************************************
*You can add, change, or delete records before sorting     *
*using COBOL data manipulation techniques.                 *
************************************************************
    IF INPUT-RECORD-BALANCE =  0 GO TO 070-READ-INPUT.
    MOVE INPUT-RECORD TO SORT-RECORD.
    RELEASE SORT-RECORD.
    GO TO 070-READ-INPUT.
100-DONE-INPUT SECTION.
110-EXIT-INPUT.
    EXIT.
200-WRITE-OUTPUT SECTION.
210-OPEN-OUTPUT.
    OPEN OUTPUT OUT-FILE.
220-GET-SORTED-RECORDS.
    RETURN SORT-FILE AT END
        CLOSE OUT-FILE
        GO TO 230-DONE-OUTPUT.
****************************************************
*You can add, change, or delete sorted records     *
*using COBOL data manipulation techniques.         *
****************************************************
    IF SORT-RECORD-STATE = "CA" GO TO 220-GET-SORTED-RECORDS.
    MOVE SORT-RECORD TO OUTPUT-RECORD.
    WRITE OUTPUT-RECORD.
    GO TO 220-GET-SORTED-RECORDS.
230-DONE-OUTPUT SECTION.
240-EXIT-OUTPUT.
    EXIT.
```

You can combine the INPUT PROCEDURE and GIVING phrases, or the USING and OUTPUT PROCEDURE phrases. In Example 10-4, there is no need to use the INPUT PROCEDURE phrase because the application has no need to manipulate its input data. In this example, all input records are sorted. Example 10-4 is identical to Example 10-3; however, in Example 10-4, the USING phrase replaces INPUT PROCEDURE.

Example 10-4: Replacing the INPUT PROCEDURE Phrase with the USING Phrase

```
PROCEDURE DIVISION.
000-SORT SECTION.
010-DO-THE-SORT.
    SORT SORT-FILE ON ASCENDING KEY SORT-KEY-1
                   DESCENDING KEY SORT-KEY-2
                   USING IN-FILE
                   OUTPUT PROCEDURE IS 200-WRITE-OUTPUT
                                  THRU 230-DONE-OUTPUT.
    DISPLAY "END OF SORT".
    STOP RUN.
200-WRITE-OUTPUT SECTION.
210-OPEN-OUTPUT.
    OPEN OUTPUT OUT-FILE.
220-GET-SORTED-RECORDS.
    RETURN SORT-FILE AT END
        CLOSE OUT-FILE
        GO TO 230-DONE-OUTPUT.
*
* This routine drops duplicate records
*
    IF THIS-IS-A-DUPLICATE-RECORD GO TO 220-GET-SORTED-RECORDS.
    MOVE SORTED-RECORD TO OUTPUT-RECORD.
    WRITE OUTPUT-RECORD.
    GO TO 220-GET-SORTED-RECORDS.
230-DONE-OUTPUT SECTION.
240-EXIT-OUTPUT.
    EXIT.
```

10.4 WITH DUPLICATES IN ORDER Phrase

Use the WITH DUPLICATES IN ORDER phrase to request that records with duplicate keys be written into the output file in the same order in which they were read into it. Without this phrase, the order of records with duplicate keys will be unpredictable.

Example 10-5 shows an input file that is to be sorted by the name field. There are two sets of records with duplicate keys. Note what can happen when you do not specify the WITH DUPLICATES IN ORDER phrase.

Example 10-5: Sorting with and Without the DUPLICATES IN ORDER Phrase

```
SORT SORT-FILE ON ASCENDING KEY NAME
               WITH DUPLICATES IN ORDER
               USING INPUT-FILE
               GIVING OUTPUT-FILE.
```

Input file	**Sorted without Duplicates in Order**	**Sorted with Duplicates in Order**
Record Name Data	Record Name Data	Record Name Data
JONES ABCD	DAVIS LMNO	DAVIS LMNO
DAVIS LMNO	JONES EFGH	JONES ABCD
WHITE STUV	JONES ABCD	JONES EFGH
JONES EFGH	SMITH 1234	SMITH 1234
SMITH 1234	WHITE STUV	WHITE STUV
WHITE WXYZ	WHITE WXYZ	WHITE WXYZ

If you omit the WITH DUPLICATES IN ORDER phrase, you cannot predict the order of records with duplicate sort keys. The JONES records are not in the same sequence as they were in the input file, but the WHITE records are.

In contrast, the WITH DUPLICATES IN ORDER phrase guarantees that records with duplicate sort keys remain in the same sequence as they were in the input file.

10.5 File Organization

You can sort any file regardless of its organization. The organization of the output file can differ from that of the input file. For example, a sort can have a sequential input file and a relative output file. In this case, the relative key for the first record returned from the sort is 1; the second record's relative key is 2; and so forth.

If an indexed file is described as output in the GIVING or OUTPUT PROCEDURE phrases, the first sort key associated with the ASCENDING phrase must specify the same character positions specified by the RECORD KEY phrase (primary key) for that file.

10.6 Multiple Sorts

A program can contain more than one sort file and/or more than one SORT statement. Example 10-6 uses two sort files to produce two differently sequenced reports.

Example 10-6: Multiple Sorts in the Same Program

```
DATA DIVISION.
FILE SECTION.
SD  SORT-FILE1.
01  SORT-REC-1.
    03  S1-KEY-1      PIC X(5).
    03  FILLER        PIC X(40).
    03  S1-KEY-2      PIC X(5).
    03  FILLER        PIC X(50).
SD  SORT-FILE2.
01  SORT-REC-2.
    03  FILLER        PIC X(20).
    03  S2-KEY-1      PIC X(10).
    03  FILLER        PIC X(10).
    03  S2-KEY-2      PIC X(10).
    03  FILLER        PIC X(50).
            .
            .
            .
PROCEDURE DIVISION.
000-SORT SECTION.
010-DO-FIRST-SORT.
    SORT SORT-FILE1 ON ASCENDING KEY
                   S1-KEY-1
                   S1-KEY-2
                   WITH DUPLICATES IN ORDER
                   USING INPUT-FILE
                   OUTPUT PROCEDURE IS 050-CREATE-REPORT-1
                                  THRU 300-DONE-REPORT-1.
```

(continued on next page)

Example 10-6: Multiple Sorts in the Same Program (Cont.)

```
020-DO-SECOND-REPORT.
    SORT SORT-FILE2 ON ASCENDING KEY
                   S2-KEY-1
                     DESCENDING KEY
                   S2-KEY-2
                   USING INPUT-FILE
                   OUTPUT PROCEDURE IS 400-CREATE-REPORT-2
                                  THRU 700-DONE-REPORT-2.
030-END-JOB.
    DISPLAY "PROGRAM ENDED".
    STOP RUN.
050-CREATE-REPORT-1 SECTION.
**********************************************************
*                                                        *
*                                                        *
*   Use the RETURN statement to read the sorted records. *
*                                                        *
*                                                        *
**********************************************************
300-DONE-REPORT-1 SECTION.
310-EXIT-REPORT-1.
    EXIT.

400-CREATE-REPORT-2 SECTION.
**********************************************************
*                                                        *
*                                                        *
*   Use the RETURN statement to read the sorted records. *
*                                                        *
*                                                        *
**********************************************************
700-DONE-REPORT-2 SECTION.
710-EXIT-REPORT.
    EXIT.
```

10.7 Sorting Variable-Length Records

If you specify the USING phrase and the input file contains variable-length records, the sort file record must not be smaller than the smallest record – nor larger than the largest record – described in the input file.

If you specify the GIVING phrase and the output file contains variable-length records, the sort file record must not be smaller than the smallest record – nor larger than the largest record – described in the output file.

10.8 Preventing I/O Aborts

All I/O errors detected during a sort can cause abnormal program termination. The USE AFTER STANDARD ERROR PROCEDURE declarative specifies error handling procedures as shown in Example 10-7.

Example 10-7: A Declarative Procedure for a Sort

```
PROCEDURE DIVISION.
DECLARATIVES.
SORT-FILE SECTION.
    USE AFTER STANDARD ERROR PROCEDURE ON INPUT-FILE.
SORT-ERROR.
    DISPLAY "I-O TYPE ERROR WHILE SORTING".
    DISPLAY "INPUT-FILE STATUS IS " INPUT-STATUS.
    STOP RUN.
END DECLARATIVES.
000-SORT SECTION.
010-DO-THE-SORT.
    SORT SORT-FILE ON DESCENDING KEY
                      S-KEY-1
                   WITH DUPLICATES IN ORDER
                   USING INPUT-FILE
                   GIVING OUTPUT-FILE.
    DISPLAY "END OF SORT".
    STOP RUN.
```

Note

The USE statement does not apply to SD (Sort Description) files.

10.9 The MERGE Statement

The MERGE statement combines two or more identically sequenced input files and makes their records available, in merged order, to an output procedure or directly to one or more output files. Use MERGE statement phrases the same way you use their SORT phrase equivalents.

In Example 10-8, two district sales input files are merged into one regional sales output file.

Example 10-8: Merge Two Files into One File

```
DATA DIVISION.
FILE SECTION.
SD  MERGE-FILE.
01  MERGE-REC.
    03  FILLER            PIC XX.
    03  M-PRODUCT-CODE    PIC X(10).
    03  FILLER            PIC X(88).
FD  DISTRICT1-SALES.
01  DISTRICT1-REC         PIC X(100).
FD  DISTRICT2-SALES.
01  DISTRICT2-REC         PIC X(100).
FD  REGION1-SALES
01  REGION1-REC           PIC X(100).
PROCEDURE DIVISION.
000-MERGE-FILES.
    MERGE MERGE-FILE ON ASCENDING KEY M-PRODUCT-CODE
          USING DISTRICT1-SALES DISTRICT2-SALES
          GIVING REGION1-SALES.
    STOP RUN.
```

10.10 Sample Programs

The following sample programs show how to use the SORT and MERGE statements.

Example 10-9: SORTA – Sorting with the USING and GIVING Phrase

```
IDENTIFICATION DIVISION.
PROGRAM-ID.            SORTA.
*************************************************
*   This program shows how to sort              *
*   a file with the USING and GIVING phrases    *
*   of the SORT statement. The fields to be     *
*   sorted are S-KEY-1 and S-KEY-2; they        *
*   contain account numbers and amounts. The    *
*   sort sequence is amount within account      *
*   number.                                     *
*   Notice that OUTPUT-FILE is a relative file. *
*************************************************
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.       PDP-11.
OBJECT-COMPUTER.       PDP-11.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT INPUT-FILE ASSIGN TO "INPFIL".
    SELECT OUTPUT-FILE ASSIGN TO "OUTFIL"
           ORGANIZATION IS RELATIVE.
    SELECT SORT-FILE ASSIGN TO "SRTFIL".
DATA DIVISION.
FILE SECTION.
SD  SORT-FILE.
01  SORT-REC.
    03  S-KEY-1.
        05  S-ACCOUNT-NUM      PIC X(8).
    03  FILLER                 PIC X(32).
    03  S-KEY-2.
        05  S-AMOUNT           PIC S9(5)V99.
    03  FILLER                 PIC X(53).
FD  INPUT-FILE
    LABEL RECORDS ARE STANDARD.
01  IN-REC                     PIC X(100).
FD  OUTPUT-FILE
    LABEL RECORDS ARE STANDARD.
01  OUT-REC                    PIC X(100).
PROCEDURE DIVISION.
000-DO-THE-SORT.
    SORT SORT-FILE ON ASCENDING KEY
                      S-KEY-1
                      S-KEY-2
         WITH DUPLICATES IN ORDER
         USING INPUT-FILE GIVING OUTPUT-FILE.
***********************************************************
*   At this point, you could transfer control to another  *
*   section of your program and continue processing.      *
***********************************************************
    DISPLAY "END OF PROGRAM SORTA".
    STOP RUN.
```

Example 10-10: SORTB – Sorting with the USING and OUTPUT PROCEDURE Phrases

```
IDENTIFICATION DIVISION.
PROGRAM-ID.            SORTB.
**************************************************************
*   This program shows how to sort a file                   *
*   with the USING and OUTPUT PROCEDURE phrases              *
*   of the SORT statement. The program eliminates            *
*   duplicate records by adding their amounts to the         *
*   amount in the first record with the same account         *
*   number. Only records with unique account numbers         *
*   are written to the output file. The fields to be         *
*   sorted are S-KEY-1 and S-KEY-2; they contain account     *
*   numbers and amounts. The sort sequence is amount         *
*   within account number.                                   *
*   Notice that the organization of OUTPUT-FILE is indexed.  *
**************************************************************
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.       PDP-11.
OBJECT-COMPUTER.       PDP-11.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT INPUT-FILE ASSIGN TO "INPFIL".
    SELECT OUTPUT-FILE ASSIGN TO "OUTFIL"
           ORGANIZATION IS INDEXED
           RECORD KEY IS OUT-KEY.
    SELECT SORT-FILE ASSIGN TO "SRTFIL".
DATA DIVISION.
FILE SECTION.
SD  SORT-FILE.
01  SORT-REC.
    03  S-KEY-1.
        05  S-ACCOUNT-NUM       PIC X(8).
    03  FILLER                  PIC X(32).
    03  S-KEY-2.
        05  S-AMOUNT            PIC S9(5)V99.
    03  FILLER                  PIC X(53).
FD  INPUT-FILE
    LABEL RECORDS ARE STANDARD.
01  IN-REC                      PIC X(100).
FD  OUTPUT-FILE
    LABEL RECORDS ARE STANDARD.
01  OUT-REC.
    03  OUT-KEY                 PIC X(8).
    03  FILLER                  PIC X(92).
WORKING-STORAGE SECTION.
01  INITIAL-SORT-READ           PIC X   VALUE "Y".
01  SAVE-SORT-REC.
    03  SR-ACCOUNT-NUM          PIC X(8).
    03  FILLER                  PIC X(32).
    03  SR-AMOUNT               PIC S9(5)V99.
    03  FILLER                  PIC X(53).
PROCEDURE DIVISION.
000-START SECTION.
005-DO-THE-SORT.
    SORT SORT-FILE ON ASCENDING KEY
                      S-KEY-1
                      S-KEY-2
         USING INPUT-FILE
         OUTPUT PROCEDURE IS 300-CREATE-OUTPUT-FILE
                        THRU 600-DONE-CREATE.
*************************************************************
*    At this point, you could transfer control to another   *
*    section of the program and continue processing.        *
*************************************************************
```

(continued on next page)

Example 10-10: SORTB – Sorting with the USING and OUTPUT PROCEDURE Phrases (Cont.)

```
    DISPLAY "END OF PROGRAM SORTB".
    STOP RUN.
300-CREATE-OUTPUT-FILE SECTION.
350-OPEN-OUTPUT.
    OPEN OUTPUT OUTPUT-FILE.
400-READ-SORT-FILE.
    RETURN SORT-FILE AT END
        PERFORM 500-WRITE-THE-OUTPUT
        CLOSE OUTPUT-FILE
        GO TO 600-DONE-CREATE.
    IF INITIAL-SORT-READ = "Y"
        MOVE SORT-REC TO SAVE-SORT-REC
        MOVE "N" TO INITIAL-SORT-READ
        GO TO 400-READ-SORT-FILE.
450-COMPARE-ACCOUNT-NUM.
    IF S-ACCOUNT-NUM = SR-ACCOUNT-NUM
        ADD S-AMOUNT TO SR-AMOUNT
        GO TO 400-READ-SORT-FILE.
500-WRITE-THE-OUTPUT.
    MOVE SAVE-SORT-REC TO OUT-REC.
    WRITE OUT-REC INVALID KEY
        DISPLAY "INVALID KEY " SR-ACCOUNT-NUM " SORTB ABORTED"
        CLOSE OUTPUT-FILE STOP RUN.
550-GET-A-REC.
    MOVE SORT-REC TO SAVE-SORT-REC.
    GO TO 400-READ-SORT-FILE.
600-DONE-CREATE SECTION.
650-EXIT-PARAGRAPH.
    EXIT.
```

Example 10-11: SORTC – Sorting with the INPUT PROCEDURE and OUTPUT PROCEDURE Phrases

```
IDENTIFICATION DIVISION.
PROGRAM-ID.              SORTC.
**********************************************************
*   This program shows how to use the INPUT              *
*   PROCEDURE and OUTPUT PROCEDURE phrases of the        *
*   SORT statement. Input to the sort is two files       *
*   containing the same type of data. Records with       *
*   a "D" status-code are not released to the sort.      *
*   The program eliminates duplicate records by          *
*   adding their amounts to the amount in the first      *
*   record with the same account number. Only records    *
*   with unique account numbers are written to           *
*   the output file. The fields to be sorted are         *
*   S-KEY-1 AND S-KEY-2. The sort sequence is amount     *
*   within account number.                               *
**********************************************************
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.         PDP-11.
OBJECT-COMPUTER.         PDP-11.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT FIRST-FILE ASSIGN TO "FILE01".
    SELECT SECOND-FILE ASSIGN TO "FILE02".
    SELECT OUTPUT-FILE ASSIGN TO "OUTFIL".
    SELECT SORT-FILE ASSIGN TO "SRTFIL".
DATA DIVISION.
FILE SECTION.
SD  SORT-FILE.
```

(continued on next page)

Example 10-11: SORTC – Sorting with the INPUT PROCEDURE and OUTPUT PROCEDURE Phrases (Cont.)

```
01  SORT-REC.
    03  S-KEY-1.
        05  S-ACCOUNT-NUM      PIC X(8).
    03  FILLER                 PIC X(32).
    03  S-KEY-2.
        05  S-AMOUNT           PIC S9(5)V99.
    03  FILLER                 PIC X(53).
FD  FIRST-FILE
    LABEL RECORDS ARE STANDARD.
01  RECORD1.
    03  FILLER                 PIC X(99).
    03  R1-STATUS-CODE         PIC X.
FD  SECOND-FILE
    LABEL RECORDS ARE STANDARD.
01  RECORD2.
    03  FILLER                 PIC X(99).
    03  R2-STATUS-CODE         PIC X.
FD  OUTPUT-FILE
    LABEL RECORDS ARE STANDARD.
01  OUT-REC                    PIC X(100).
WORKING-STORAGE SECTION.
01  INITIAL-SORT-READ          PIC X    VALUE "Y".
01  FILE01-COUNT               PIC 9(5) VALUE ZEROES.
01  FILE02-COUNT               PIC 9(5) VALUE ZEROES.
01  SORT-COUNT                 PIC 9(5) VALUE ZEROES.
01  OUTPUT-COUNT               PIC 9(5) VALUE ZEROES.
01  SAVE-SORT-REC.
    03  SR-ACCOUNT-NUM         PIC X(8).
    03  FILLER                 PIC X(32).
    03  SR-AMOUNT              PIC S9(5)V99.
    03  FILLER                 PIC X(53).
PROCEDURE DIVISION.
000-START SECTION.
005-DO-THE-SORT.
    SORT SORT-FILE ON ASCENDING KEY
                      S-KEY-1
                      S-KEY-2
         INPUT PROCEDURE IS 010-GET-INPUT
                       THRU 200-DONE-INPUT-GET
         OUTPUT PROCEDURE IS 300-CREATE-OUTPUT-FILE
                        THRU 600-DONE-CREATE.
********************************************************
*   Notice the use of DISPLAY and record counters to   *
*   produce sort statistics.                           *
********************************************************
    DISPLAY "TOTAL FIRST-FILE RECORDS IS       " FILE01-COUNT.
    DISPLAY "TOTAL SECOND-FILE RECORDS IS      " FILE02-COUNT.
    DISPLAY "TOTAL NUMBER OF SORTED RECORDS IS " SORT-COUNT.
    DISPLAY "TOTAL NUMBER OF OUTPUT RECORDS IS " OUTPUT-COUNT.
************************************************************
*   At this point, you could transfer control to another   *
*   section of the program  and continue processing.       *
************************************************************
    DISPLAY "END OF PROGRAM SORTC".
    STOP RUN.
010-GET-INPUT SECTION.
050-OPEN-FILES.
    OPEN INPUT FIRST-FILE.
```

(continued on next page)

Example 10-11: SORTC – Sorting with the INPUT PROCEDURE and OUTPUT PROCEDURE Phrases (Cont.)

```
100-READ-FIRST-FILE.
    READ FIRST-FILE AT END
        CLOSE FIRST-FILE
        OPEN INPUT SECOND-FILE
        GO TO 150-READ-SECOND-FILE.
    ADD 1 TO FILE01-COUNT.
    IF R1-STATUS-CODE = "D"
        GO TO 100-READ-FIRST-FILE.
    RELEASE SORT-REC FROM RECORD1.
    GO TO 100-READ-FIRST-FILE.
150-READ-SECOND-FILE.
    READ SECOND-FILE AT END
        CLOSE SECOND-FILE
        GO TO 200-DONE-INPUT-GET.
    ADD 1 TO FILE02-COUNT.
    IF R2-STATUS-CODE = "D"
        GO TO 150-READ-SECOND-FILE.
    RELEASE SORT-REC FROM RECORD2.
    GO TO 150-READ-SECOND-FILE.
200-DONE-INPUT-GET SECTION.
250-EXIT-PARAGRAPH.
    EXIT.
300-CREATE-OUTPUT-FILE SECTION.
350-OPEN-OUTPUT.
    OPEN OUTPUT OUTPUT-FILE.
400-READ-SORT-FILE.
    RETURN SORT-FILE AT END
        PERFORM 500-WRITE-THE-OUTPUT
        CLOSE OUTPUT-FILE
        GO TO 600-DONE-CREATE.
    ADD 1 TO SORT-COUNT.
    IF INITIAL-SORT-READ = "Y"
        MOVE SORT-REC TO SAVE-SORT-REC
        MOVE "N" TO INITIAL-SORT-READ
        GO TO 400-READ-SORT-FILE.
450-COMPARE-ACCOUNT-NUM.
    IF S-ACCOUNT-NUM = SR-ACCOUNT-NUM
        ADD S-AMOUNT TO SR-AMOUNT
        GO TO 400-READ-SORT-FILE.
500-WRITE-THE-OUTPUT.
    MOVE SAVE-SORT-REC TO OUT-REC.
    WRITE OUT-REC.
    ADD 1 TO OUTPUT-COUNT.
550-GET-A-REC.
    MOVE SORT-REC TO SAVE-SORT-REC.
    GO TO 400-READ-SORT-FILE.
600-DONE-CREATE SECTION.
650-EXIT-PARAGRAPH.
    EXIT.
```

Example 10-12: SORTE – Sorting a File and Expanding Its Output Records

```
IDENTIFICATION DIVISION.
PROGRAM-ID.   SORTE.
************************************************
*   This program increases the size of the     *
*   variable input records by a new six-       *
*   character field and uses this field        *
*   as the sort key.                           *
************************************************
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
```

(continued on next page)

Example 10-12: SORTE – Sorting a File and Expanding Its Output Records (Cont.)

```
SOURCE-COMPUTER.    PDP-11.
OBJECT-COMPUTER.    PDP-11.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT INFILE ASSIGN TO "INFILE".
    SELECT SORT-FILE ASSIGN TO "SRTFIL".
    SELECT OUT-FILE ASSIGN TO "OUTFILE".
DATA DIVISION.
FILE SECTION.
FD  INFILE
      RECORD VARYING FROM 100 TO 490 CHARACTERS
                     DEPENDING ON IN-LENGTH.
01  INREC.
    03 ACCOUNT                 PIC 9(5).
    03 INCOME-FIRST-QUARTER    PIC 9(5)V99.
    03 INCOME-SECOND-QUARTER   PIC 9(5)V99.
    03 INCOME-THIRD-QUARTER    PIC 9(5)V99.
    03 INCOME-FOURTH-QUARTER   PIC 9(5)V99.
    03 ORDER-COUNT             PIC 9(2).
    03 ORDERS OCCURS 1 TO 7 TIMES
              DEPENDING ON ORDER-COUNT.
       05  ORDER-DATE         PIC 9(6).
       05  FILLER             PIC X(59).
SD  SORT-FILE
    RECORD VARYING FROM 106 TO 496 CHARACTERS
    DEPENDING ON SORT-LENGTH.
01  SORT-REC.
    03  SORT-ANNUAL-INCOME     PIC 9(6).
    03  SORT-REST-OF-RECORD    PIC X(490).
FD      OUT-FILE
        RECORD VARYING FROM 106 TO 496 CHARACTERS
        DEPENDING ON OUT-LENGTH.
01      OUT-REC                    PIC X(496).
WORKING-STORAGE SECTION.
01  IN-LENGTH                      PIC 9(3) COMP.
01  SORT-LENGTH                    PIC 9(3) COMP.
01  OUT-LENGTH                     PIC 9(3) COMP.
PROCEDURE DIVISION.
000-START SECTION.
005-SORT-HERE.
    SORT SORT-FILE
         ON DESCENDING SORT-ANNUAL-INCOME
         INPUT PROCEDURE 010-GET-INPUT
                    THRU 070-DONE-INPUT
         OUTPUT PROCEDURE 100-WRITE-OUTPUT.
    DISPLAY "END OF PROGRAM SORTE".
    STOP RUN.
010-GET-INPUT SECTION.
020-OPEN-INPUT.
    OPEN INPUT INFILE.
030-READ-INPUT.
    READ INFILE AT END
        CLOSE INFILE
        GO TO 070-DONE-INPUT.
040-ADD-INCOME.
    ADD INCOME-FIRST-QUARTER
        INCOME-SECOND-QUARTER
        INCOME-THIRD-QUARTER
        INCOME-FOURTH-QUARTER
        GIVING SORT-ANNUAL-INCOME.
```

(continued on next page)

Example 10-12: SORTE – Sorting a File and Expanding Its Output Records (Cont.)

```
050-CREATE-SORT-REC.
    ADD 6 IN-LENGTH GIVING SORT-LENGTH.
    MOVE INREC TO SORT-REST-OF-RECORD.
    RELEASE SORT-REC.
    GO TO 030-READ-INPUT.
070-DONE-INPUT SECTION.
080-EXIT.
    EXIT.
100-WRITE-OUTPUT SECTION.
110-OPEN.
    OPEN OUTPUT OUT-FILE.
120-WRITE.
    RETURN SORT-FILE AT END
        CLOSE OUT-FILE
        GO TO 130-DONE.
    MOVE SORT-LENGTH TO OUT-LENGTH.
    WRITE OUT-REC.
    GO TO 120-WRITE.
130-DONE.
    EXIT.
```

Example 10-13: MERGE01 – Merging Three Files

```
IDENTIFICATION DIVISION.
PROGRAM-ID.    MERGE01.
******************************************************
*   This program MERGEs three identically sequenced  *
*   regional sales files into one total sales file.  *
*   The program adds sales amounts and writes one    *
*   record for each product-code.                    *
******************************************************
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.     PDP-11.
OBJECT-COMPUTER.     PDP-11.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT REGION1-SALES ASSIGN TO "REG1SLS".
    SELECT REGION2-SALES ASSIGN TO "REG2SLS".
    SELECT REGION3-SALES ASSIGN TO "REG3SLS".
    SELECT MERGE-FILE    ASSIGN TO "MRGFILE".
    SELECT TOTAL-SALES   ASSIGN TO "TOTLSLS".
DATA DIVISION.
FILE SECTION.
FD  REGION1-SALES
    LABEL RECORDS ARE STANDARD.
01  REGION1-RECORD            PIC X(100).
FD  REGION2-SALES
    LABEL RECORDS ARE STANDARD.
01  REGION2-RECORD            PIC X(100).
FD  REGION3-SALES
    LABEL RECORDS ARE STANDARD.
01  REGION3-RECORD            PIC X(100).
SD  MERGE-FILE.
01  MERGE-REC.
    03  M-REGION-CODE         PIC XX.
    03  M-PRODUCT-CODE        PIC X(10).
    03  M-SALES-AMT           PIC S9(7)V99.
    03  FILLER                PIC X(79).
FD  TOTAL-SALES
    LABEL RECORDS ARE STANDARD.
```

(continued on next page)

Example 10-13: MERGE01 – Merging Three Files (Cont.)

```
01  TOTAL-RECORD              PIC X(100).
WORKING-STORAGE SECTION.
01  INITIAL-READ              PIC X  VALUE "Y".
01  THE-COUNTERS.
    03  PRODUCT-AMT           PIC S9(7)V99.
    03  REGION1-AMT           PIC S9(9)V99.
    03  REGION2-AMT           PIC S9(9)V99.
    03  REGION3-AMT           PIC S9(9)V99.
    03  TOTAL-AMT             PIC S9(11)V99.
01  SAVE-MERGE-REC.
    03  S-REGION-CODE         PIC XX.
    03  S-PRODUCT-CODE        PIC X(10).
    03  S-SALES-AMT           PIC S9(7)V99.
    03  FILLER                PIC X(79).
PROCEDURE DIVISION.
000-START SECTION.
010-MERGE-FILES.
    OPEN OUTPUT TOTAL-SALES.
    MERGE MERGE-FILE ON ASCENDING KEY M-PRODUCT-CODE
          USING REGION1-SALES REGION2-SALES REGION3-SALES
          OUTPUT PROCEDURE IS 020-BUILD-TOTAL-SALES
                         THRU 100-DONE-TOTAL-SALES.
    DISPLAY "TOTAL SALES FOR REGION 1 " REGION1-AMT.
    DISPLAY "TOTAL SALES FOR REGION 2 " REGION2-AMT.
    DISPLAY "TOTAL SALES FOR REGION 3 " REGION3-AMT.
    DISPLAY "TOTAL ALL SALES          " TOTAL-AMT.
    CLOSE TOTAL-SALES.
    DISPLAY "END OF PROGRAM MERGE01".
    STOP RUN.
020-BUILD-TOTAL-SALES SECTION.
030-GET-MERGE-RECORDS.
    RETURN MERGE-FILE AT END
           MOVE PRODUCT-AMT TO S-SALES-AMT
           WRITE TOTAL-RECORD FROM SAVE-MERGE-REC
           GO TO 100-DONE-TOTAL-SALES.
    IF INITIAL-READ = "Y"
           MOVE "N" TO INITIAL-READ
           MOVE MERGE-REC TO SAVE-MERGE-REC
           PERFORM 050-TALLY-AMOUNTS
           GO TO 030-GET-MERGE-RECORDS.
040-COMPARE-PRODUCT-CODE.
    IF M-PRODUCT-CODE = S-PRODUCT-CODE
           PERFORM 050-TALLY-AMOUNTS
           GO TO 030-GET-MERGE-RECORDS.
    MOVE PRODUCT-AMT TO S-SALES-AMT.
    MOVE ZEROES TO PRODUCT-AMT.
    WRITE TOTAL-RECORD FROM SAVE-MERGE-REC.
    MOVE MERGE-REC TO SAVE-MERGE-REC.
    GO TO 040-COMPARE-PRODUCT-CODE.
050-TALLY-AMOUNTS.
    ADD M-SALES-AMT TO PRODUCT-AMT TOTAL-AMT.
    IF M-REGION-CODE = "01"
           ADD M-SALES-AMT TO REGION1-AMT.
    IF M-REGION-CODE = "02"
           ADD M-SALES-AMT TO REGION2-AMT.
    IF M-REGION-CODE = "03"
           ADD M-SALES-AMT TO REGION3-AMT.
100-DONE-TOTAL-SALES SECTION.
120-DONE.
    EXIT.
```

Appendix A
Designing Your Form with Escape Sequences

Before you can use escape sequences in DISPLAY statements to control output to a screen, you must define data items in the Working-Storage Section to represent the following:

- The ESCAPE (or ALTMODE) character
- The current line number
- The current column number

This section explains how to define line and column numbers on VT100, VT52, and Professional terminals. The definition of the ESCAPE character is the same for each terminal.

A.1 Defining the ESCAPE

To store an ESCAPE character in a data item, you must:

1. Define an item with COMP usage
2. Set its value to 155 for RSTS/E and 27 for RSX-11M/M-PLUS/Professional (the ASCII value for the ESCAPE key)
3. Redefine the least-significant byte as alphanumeric

The following code shows you how to define the ESCAPE character.

```
WORKING-STORAGE SECTION.
*
*  For RSX:
*
*          01  ESCAPE-VAL PIC 999 COMP VALUE 27.
*
*
*  For RSTS/E:
01  ESCAPE-VAL  PIC 999 COMP VALUE 155.
01  ESCAPE-RED  REDEFINES ESCAPE-VAL.
    03  ESCAPE  PIC X.
    03  FILLER  PIC X.
```

The compiler allocates two bytes of storage for ESCAPE-VAL because it is a COMP item. The value 155 is stored in the least significant byte. The data item ESCAPE redefines that byte as alphanumeric. You can now use the data item ESCAPE in DISPLAY statements.

A.2 Defining Items for Line and Column Numbers

If your program will be used on a VT100 terminal, define two items with DISPLAY usage to contain the current values for line number and column number. Since (line 1, column 1) represents the upper left position on the screen, initializing the value of each item to 1 prepares the program to begin displaying in the upper left corner. For example:

```
01   LINE-NO PIC 99 VALUE 1.
01   COL-NO PIC 99 VALUE 1.
```

If you are programming on a VT52 terminal, the values for cursor position must be represented by one-byte binary items and displayed as alphanumeric items, as is true of the ESCAPE. Therefore, each of the two DISPLAY items must redefine the least significant byte of a COMP item. Line and column numbers begin with 32 on a VT52, so (line 32, column 32) represents the upper left position on the screen. The following lines define the items you will need on a VT52 terminal, and initialize their values to 32:

```
01   LINE-VAL PIC 999 COMP VALUE 32.
01   LINE-RED REDEFINES LINE-VAL.
          03  LINE-NO PIC X.
          03  FILLER PIC X.
01   COL-VAL PIC 999 COMP VALUE 32.
01   COL-RED REDEFINES COL-VAL.
          03  COL-NO PIC X.
          03  FILLER PIC X.
```

LINE-NO and COL-NO name the items you will use in cursor positioning sequences. To change cursor position, you must move the new values to LINE-VAL and COL-VAL, rather than moving them directly to LINE-NO and COL-NO.

A.3 Sending Control Character Sequences to the Terminal

The sequence of characters you must use for each screen control operation depends on the terminal you are using. This section shows you the proper sequences to control the most frequently used operations on VT100, VT52, and Professional terminals. For information on all the screen operations available, consult the documentation for your terminal.

If you are developing a program for use on both VT100 and VT52 terminals, it is inconvenient to code two different sequences for each DISPLAY statement. You can avoid this extra effort by forcing a VT100 into "VT52-Compatible Mode" at the beginning of the program. This section explains the sequence that puts a VT100 into that mode once it is displayed. (VT52 terminals do not have a comparable "VT100-Compatible Mode.")

Note

Sending VT100 escape sequences to a VT52 terminal can lock the terminal. The terminal can be reset by turning the unit off and on again.

A.3.1 Sending to VT100 Terminals

The following sequence controls cursor positioning:

escape "[" line-number ";" column-number "f"

Escape, line-number, and column-number must be data names for the items discussed in Sections A.1 and A.2.

Substituting an "H" for the "f" in this sequence gives the same result.

This example shows how to use the sequence in DISPLAY statements:

```
WORKING-STORAGE SECTION.
*
*  For RSX:
*
*         01  ESCAPE-VAL PIC 999 COMP VALUE 27.
*
*
*  For RSTS/E:
01  ESCAPE-VAL PIC 999 COMP VALUE 155.
01  ESCAPE-RED REDEFINES ESCAPE-VAL.
    03 ESCAPE PIC X.
    03 FILLER PIC X.
01  LINE-NO PIC 99 VALUE 1.
01  COL-NO PIC 99 VALUE 1.
            .
            .
            .
    DISPLAY ESCAPE "[" LINE-NO ";" COL-NO "f" WITH NO ADVANCING.
    DISPLAY MES-SAGE1.
    MOVE 4 TO LINE-NO.
    MOVE 20 TO COL-NO.
    DISPLAY ESCAPE "[" LINE-NO ";" COL-NO "f" MES-SAGE2.
```

The first two statements display the contents of MES-SAGE1 in the top left corner of the screen. Then the program changes the values of the cursor coordinates, and MES-SAGE2 begins in the 20th column of the 4th line.

Note the WITH NO ADVANCING phrase in the first statement. If you did not specify this phrase, COBOL-81 would position the cursor at the beginning of the second line after executing the statement, thereby defeating the purpose of positioning the cursor.

The example shows another way of preventing the cursor from being repositioned: specifying the prompt in the same statement that positions the cursor. MES-SAGE2 is displayed in this way.

The following sequences erase the screen:

escape "[J"   erases from the cursor to the end of the screen

escape "[1J"   erases from the beginning of the screen to the cursor

escape "[2J"   erases the entire screen

The position of the cursor does not change when you display these sequences unless you do not specify WITH NO ADVANCING.

The next example erases the entire screen and positions the cursor at the top left corner in preparation for displaying output:

```
WORKING-STORAGE SECTION.
*
*  For RSX:
*
*          01  ESCAPE PIC 999 COMP VALUE 27.
*
*
*  For RSTS/E:
01  ESCAPE PIC 999 COMP VALUE 155.
01  ESC-R REDEFINES ESCAPE.
    03  E PIC X.
    03  FILLER PIC X.
01  X PIC 99 VALUE 1.
01  Y PIC 99 VALUE 1.

PROCEDURE DIVISION.
A00-BEGIN.
    DISPLAY E  "[2J"   E "[" X ";" Y "f" WITH NO ADVANCING.
           └────┬────┘  └────────┬────────┘
              Erases         Moves cursor
              screen          to top left
```

A.3.2 Sending to VT52 Terminals

The following sequence controls cursor positioning:

escape "Y" line-number column-number

Escape, line-number, and column-number must be data-names for the items discussed in Sections A.1 and A.2.

This example shows how to use the sequence in DISPLAY statements:

```
WORKING-STORAGE SECTION.
*
*  For RSX:
*
*          01  ESCAPE-VAL PIC 999 COMP VALUE 27.
*
*
* For RSTS/E:
01  ESCAPE-VAL PIC 999 COMP VALUE 155.
01  ESCAPE-RED REDEFINES ESCAPE-VAL.
    03 ESCAPE PIC X.
    03 FILLER PIC X.
01  LINE-VAL PIC 999 COMP VALUE 32.
```

(continued on next page)

```
01   LINE-RED REDEFINES LINE-VAL.
     03  LINE-NO PIC X.
     03  FILLER PIC X.
01   COL-VAL PIC 999 COMP VALUE 32.
01   COL-RED REDEFINES COL-VAL.
     03  COL-NO PIC X.
     03  FILLER PIC X.
PROCEDURE DIVISION.
A00-BEGIN.
      .
      .
      .
     DISPLAY ESCAPE "Y" LINE-NO COL-NO WITH NO ADVANCING.
     DISPLAY MES-SAGE1.
     MOVE 36 TO LINE-VAL.
     MOVE 52 TO COL-VAL.
     DISPLAY ESCAPE "Y" LINE-NO COL-NO MES-SAGE2.
```

This example produces the same output as its corresponding VT100 example. The first two statements display the contents of MES-SAGE1 in the top left corner of the screen. Then the program changes the values of the cursor coordinates. Note that the new values for cursor position are moved to LIN-VAL and COL-VAL, rather than LIN-NO and COL-NO. The current line position becomes (36 – 32) = 4, and the current column position becomes (52 – 32) = 20. Therefore, MES-SAGE2 begins in the 20th column of the 4th line.

Note the WITH NO ADVANCING phrase in the first statement. If you did not specify this phrase, COBOL-81 would position the cursor at the beginning of the second line after executing the statement, thereby defeating the purpose of positioning the cursor.

The example shows another way of preventing the cursor from being repositioned: specifying the prompt in the same statement that positions the cursor. MES-SAGE2 is displayed in this way.

Use the following sequence to position the cursor at the top left of the screen without having to specify a line-number or column-number:

escape ''H''

These two examples have the same effect:

```
                 (1)                               (2)

MOVE 32 TO LINE-VAL COL-VAL.            DISPLAY ESCAPE "H".
DISPLAY ESCAPE "Y" LINE-NO COL-NO.
```

Use the following sequence to erase the screen:

escape ''J''

This sequence erases from the current cursor position to the end of the screen. Therefore, if you want to erase the entire screen, you must first move the cursor to the top left position, and then display this sequence. Here is an example:

```
WORKING-STORAGE SECTION.
*
*  For RSX:
*
*         01  ESCAPE PIC 999 COMP VALUE 27.
*
*
*  For RSTS/E:
01  ESCAPE PIC 999 COMP VALUE 155.
01  ESC-R REDEFINES ESCAPE.
    03  E PIC X.
    03  FILLER PIC X.
01  LINE-NO PIC 999 COMP VALUE 32.
01  LINE-R REDEFINES LINE-NO.
    03  X PIC X.
    03  FILLER PIC X.
01  COL-NO PIC 999 COMP VALUE 32.
01  COL-R REDEFINES COL-NO.
    03  Y PIC X.
    03  FILLER PIC X.
    .
    .
    .
    MOVE 47 TO LINE-NO.
    MOVE 72 TO COL-NO.
    DISPLAY E "Y" X Y MES-SAGE.
    DISPLAY E  "H"          E  "J".
          Moves cursor      Erases
           to top left      screen
```

A.3.3 Switching from VT100 Mode to VT52 Mode

VT100 terminals can operate in "VT52-Compatible Mode." To develop a program that will run on both VT100 and VT52 terminals, use the VT52 control sequences. At the beginning of the Procedure Division, prompt the user to identify the terminal type. If the terminal is a VT100, use the following sequence to put it in VT52 mode:

escape "[?2l"

At the end of the Procedure Division, use this sequence to return the terminal to VT100 mode:

escape "<"

This example shows how to use both sequences in DISPLAY statements:

```
PROCEDURE DIVISION.
VT52-MODE.
    DISPLAY ESCAPE "[ ?21".
PARA1.
    DISPLAY ESCAPE "Y" LINE-NO COL-NO ESC "J".
```

(continued on next page)

```
PARA2.
    .
    .
    .
VT100-MODE.
    DISPLAY ESCAPE "<".
FINI.
    STOP RUN.
```

Section A.4.2. contains a VT52 programming example that optionally forces a VT100 into (and out of) VT52 Compatible Mode.

A.4 Examples

This section includes two VT100 examples and one VT52 example. The first two examples show cursor positioning. The last example and its screen display in Figure A–1 shows you how to display double-height and double-width text on your screen.

A.4.1 VT100 Example

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  VT100.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
*
*  For RSTS/E:
*            01  ESC-VAL      PIC 9(4) COMP VALUE 155.
*
*  For RSX:
*            01  ESC-VAL      PIC 9(4) COMP VALUE 27.
*
01  ESC-VAL                   PIC 9(4) COMP VALUE 27.
01  ESC-RDF REDEFINES ESC-VAL.
    03  ESC                   PIC X.
    03  FILLER                PIC X.
01  LIN                       PIC 99 VALUE 1.
01  COL                       PIC 99 VALUE 1.
01  ANS                       PIC X.
01  CLEAN-SCREEN              PIC X(4).
01  HOME                      PIC X(6).
01  FIRST-LINE                PIC 9(4) COMP VALUE 48.
01  FLINE REDEFINES FIRST-LINE.
    03  X                     PIC X.
    03  FILLER                PIC X.
01  FIRST-COL                 PIC 9(4) COMP VALUE 48.
01  FCOL REDEFINES FIRST-COL.
    03  Y                     PIC X.
    03  FILLER                PIC X.
PROCEDURE DIVISION.
SET-UP.
    STRING ESC "[2J" DELIMITED SIZE INTO CLEAN-SCREEN.
    STRING ESC "[" X ";" Y "f" DELIMITED SIZE INTO HOME.
```

(continued on next page)

```
NUTSHELL.
    DISPLAY CLEAN-SCREEN HOME WITH NO ADVANCING.
    PERFORM MESSAGE1.
    MOVE 10 TO LIN.
    MOVE 25 TO COL.
    DISPLAY ESC "[" LIN ";" COL "f" WITH NO ADVANCING.
    PERFORM MESSAGE2.
    ADD 1 TO LIN.
    DISPLAY ESC "[" LIN ";" COL "f" WITH NO ADVANCING.
    PERFORM MESSAGE3.
    DISPLAY ESC "[1J" ESC "[" LIN ";" COL "f" WITH NO ADVANCING.
    PERFORM MESSAGE4.
    DISPLAY HOME WITH NO ADVANCING.
    PERFORM MESSAGE5.
    DISPLAY CLEAN-SCREEN WITH NO ADVANCING.
    DISPLAY ESC "[7m" WITH NO ADVANCING.
    PERFORM MESSAGE6.
    DISPLAY CLEAN-SCREEN HOME WITH NO ADVANCING.
    DISPLAY ESC "[0m" WITH NO ADVANCING.
FINI.
    STOP RUN.
MESSAGE1.
    DISPLAY "Screen cleared courtesy of the sequence".
    DISPLAY "in the data item CLEAN-SCREEN.".
    DISPLAY "Top-left positioning courtesy of the sequence in HOME.".
    DISPLAY "Press RETURN for more . . ." WITH NO ADVANCING.
    ACCEPT ANS.
MESSAGE2.
    DISPLAY "This message begins in column 25 of line 10.".
MESSAGE3.
    DISPLAY "To erase the screen, press RETURN." WITH NO ADVANCING.
    ACCEPT ANS.
MESSAGE4.
    DISPLAY "Now, press RETURN to get the cursor to top-left."
        WITH NO ADVANCING.
    ACCEPT ANS.
MESSAGE5.
    DISPLAY "Notice that moving the cursor to top-left".
    DISPLAY "doesn't clear the screen.".
    DISPLAY "Now press RETURN to clear." WITH NO ADVANCING.
    ACCEPT ANS.
MESSAGE6.
    DISPLAY "Notice that clearing the screen".
    DISPLAY "doesn't affect cursor position.".
    DISPLAY "Now, press RETURN to end." WITH NO ADVANCING.
    ACCEPT ANS.
```

A.4.2 VT52 Example

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  VT52.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
*
*  For RSTS/E:
*              01  ESC-VAL    PIC 9(4) COMP VALUE 155.
*
*  For RSX:
01  ESC-VAL                   PIC 9(4) COMP VALUE 27.
01  ESC-RDF REDEFINES ESC-VAL.
    03  ESC                   PIC X.
    03  FILLER                PIC X.
```

(continued on next page)

```
01  LIN-VAL                   PIC 9(4) COMP VALUE 32.
01  LIN-RDF REDEFINES LIN-VAL.
    03  LIN                   PIC X.
    03  FILLER                PIC X.
01  COL-VAL                   PIC 9(4) COMP VALUE 32.
01  COL-RDF REDEFINES COL-VAL.
    03  COL                   PIC X.
    03  FILLER                PIC X.
01  TERM-ANS                  PIC X VALUE SPACE.
01  ANS                       PIC X.
01  HOME                      PIC XX.
01  CLEAR                     PIC XX.
PROCEDURE DIVISION.
PROPER-MODE.
    DISPLAY "Are you on a VT52 terminal?".
    DISPLAY "(Type 'Y' or 'N')" WITH NO ADVANCING.
    ACCEPT TERM-ANS.
    IF (TERM-ANS = "N") OR (TERM-ANS = "n")
      THEN DISPLAY ESC "[?2l".
SET-UP.
    STRING ESC "H" DELIMITED SIZE INTO HOME.
    STRING ESC "J" DELIMITED SIZE INTO CLEAR.
NUTSHELL.
    DISPLAY HOME CLEAR WITH NO ADVANCING.
    PERFORM MESSAGE1.
    MOVE 42 TO LIN-VAL.
    MOVE 57 TO COL-VAL.
    PERFORM MESSAGE2.
    PERFORM ERASE-MESSAGE2.
    MOVE 47 TO LIN-VAL.
    PERFORM MESSAGE3.
    DISPLAY ESC "H" WITH NO ADVANCING.
    PERFORM MESSAGE4.
    DISPLAY HOME CLEAR.
FINI.
    STOP RUN.
MESSAGE1.
    DISPLAY "When you press RETURN, the".
    DISPLAY "next message will begin in".
    DISPLAY "col. 25 of line 10, using".
    DISPLAY "the sequence:".
    DISPLAY "   ESC ""Y"" (line) (col.)" WITH NO ADVANCING.
    ACCEPT ANS.
MESSAGE2.
    DISPLAY ESC "Y" LIN COL WITH NO ADVANCING.
    DISPLAY "Notice that the source program had".
    PERFORM NEXT-LINE.
    DISPLAY "to move the new values for posi-".
    PERFORM NEXT-LINE.
    DISPLAY "tion to LIN-VAL and COL-VAL,".
    PERFORM NEXT-LINE.
    DISPLAY "rather than LIN and COL.".
    PERFORM NEXT-LINE.
    DISPLAY "  To erase this message, press".
    PERFORM NEXT-LINE.
    DISPLAY "RETURN." WITH NO ADVANCING.
    ACCEPT ANS.
MESSAGE3.
    DISPLAY ESC "Y" LIN COL WITH NO ADVANCING.
    DISPLAY "That erasure was accomplished by repositioning".
    DISPLAY "the cursor to (10,1), and DISPLAYing the sequence:".
    DISPLAY "            ESC ""J""  ".
    DISPLAY "Now, press RETURN and the sequence".
    DISPLAY "            ESC ""H""  ".
    DISPLAY "will move the cursor to top-left." WITH NO ADVANCING.
    ACCEPT ANS.
```

(continued on next page)

```
MESSAGE4.
    DISPLAY "Next, the program will erase         ".
    DISPLAY "the screen by DISPLAYing the         ".
    DISPLAY "contents of the data items           ".
    DISPLAY "HOME and CLEAR.  HOME contains       ".
    DISPLAY "     ESC ""H""                       ".
    DISPLAY "and CLEAR contains                   ".
    DISPLAY "     ESC ""J""                       ".
    DISPLAY "Press RETURN when ready." WITH NO ADVANCING.
    ACCEPT ANS.
NEXT-LINE.
    ADD 1 TO LIN-VAL.
    DISPLAY ESC "Y" LIN COL WITH NO ADVANCING.
ERASE-MESSAGE2.
    MOVE 42 TO LIN-VAL.
    MOVE 32 TO COL-VAL.
    DISPLAY ESC "Y" LIN COL WITH NO ADVANCING.
    DISPLAY ESC "J".
```

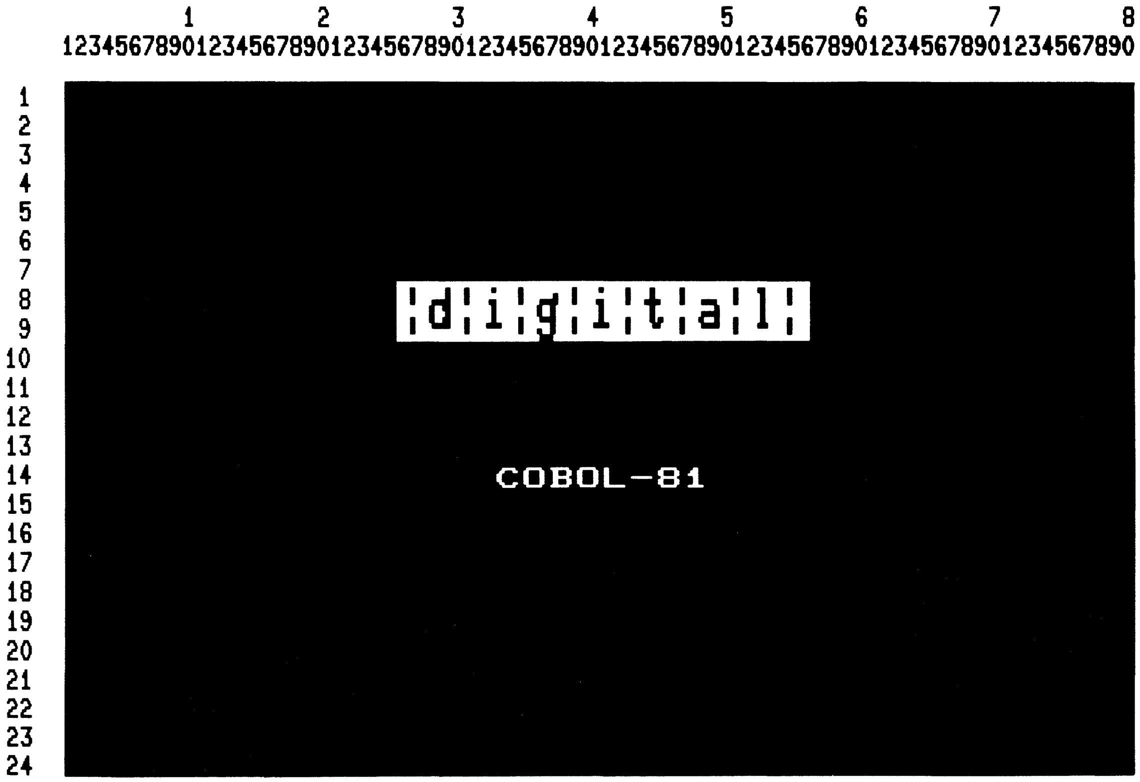
A.4.3 Example of Double-Height and Double-Width Lines on a VT100

```
IDENTIFICATION DIVISION.

PROGRAM-ID.
  ADVANCED-VIDEO-DEMO.

ENVIRONMENT DIVISION.

DATA DIVISION.

FILE SECTION.

WORKING-STORAGE SECTION.
01 DOUBLE-HEIGTH-LINE.
   02 TOP-LINE.
      05 ESCAPE-CODE   PIC 9(4) COMP VALUE 27.
      05 ESCAPE REDEFINES ESCAPE-CODE.
         10 ESCAPE-IDENT    PIC X.
         10 FILLER    PIC X.
      05 TOP-LINE-CODE                PIC XX  VALUE "#3".
      05 MESSAGE-TEXT      PIC X(15) VALUE "¦d¦i¦g¦i¦t¦a¦l¦" .
   02 bottom-line.
      05 ESCAPE-CODE   PIC 9(4) COMP VALUE 27.
      05 ESCAPE REDEFINES ESCAPE-CODE.
         10 ESCAPE-IDENT    PIC X.
         10 FILLER    PIC X.
      05 BOTTOM-LINE-CODE             PIC XX  VALUE "#4".
      05 MESSAGE-TEXT      PIC X(15) VALUE "¦d¦i¦g¦i¦t¦a¦l¦" .

01 DOUBLE-WIDTH-LINE.
      05 ESCAPE-CODE   PIC 9(4) COMP VALUE 27.
      05 ESCAPE REDEFINES ESCAPE-CODE.
        10 ESCAPE-IDENT    PIC X.
        10 FILLER    PIC X.
      05 DOUBLE-WIDTH-LINE-CODE       PIC XX  VALUE "#6".
      05 MESSAGE-TEXT      PIC X(8) VALUE "COBOL-81".

PROCEDURE DIVISION.
```

(continued on next page)

```
PROCESS-LOGO.
  DISPLAY " "   LINE 1
                COLUMN 1
                ERASE TO END OF SCREEN.
  DISPLAY TOP-LINE OF DOUBLE-HEIGTH-LINE
          LINE 8
          COLUMN 13
          REVERSED
          BOLD.
  DISPLAY BOTTOM-LINE OF DOUBLE-HEIGTH-LINE
          LINE 9
          COLUMN 13
          REVERSED
          BOLD.
  DISPLAY DOUBLE-WIDTH-LINE
          LINE 14
          COLUMN 17
          BOLD.
```

Figure A-1: Double-Height and Double-Width Lines on a VT100



```
digital

COBOL-81
```

C81ART-20595-30

Appendix B
Logical Unit Number (LUN) Assignments

The COBOL-81 Object-Time System (OTS) assigns no more than 14 logical units (or channels) per task image. The logical unit numbers (LUNs) for a given task range from 1 to 14. The first logical unit (LUN 1) is always assigned to the terminal from which the program is executed. The OTS assigns the remaining LUNs as needed based on the number of files you open in your program and the COBOL-81 features you use.

Table B-1 shows the maximum number of files that your task can open simultaneously based on the COBOL-81 features you include in the task image. As you can see from Table B-1, if your task includes *both* ACCEPT FROM device-name and DIGITAL extensions to the ACCEPT or DISPLAY statements, your task can simultaneously open only 12 files. If your task includes one (or more than one) SORT or MERGE statement, the OTS assigns three logical units to the sort or merge operation(s), leaving you a maximum of only 10 files to be open simultaneously. If you include the COBOL-81 Symbolic Debugger, the OTS assigns four logical units to the Debugger, leaving you a maximum of only 9 files to be open simultaneously.

By default, the COBOL-81 OTS dynamically assigns one logical unit for each file you open during task execution. If you receive the following BLDODL utility error message, your task attempted to use more than the 14 logical unit assignments:

```
? Maximum number of LUNs exceeded.
? Your SKL file(s) require n LUNs for file I/O.
? That value replaced by system maximum.
```

You can ignore this error message, however, if your task does not violate the LUN assignments shown in Table B-1. Consider, for example, a task that defines 25 files, uses the Symbolic Debugger, and uses ACCEPT/DISPLAY extensions. As long as this task does not simultaneously open more than 9 of its 25 files there will be no problem. Here is how the OTS makes its LUN assignments for this task:

1 – ACCEPT/DISPLAY extensions (they share LUN 1 with your terminal)
9 – maximum files that can be open at the same time in this task
4 – Symbolic Debugger

14 Total LUNs

If your task opens more files than are allowed to be open at the same time you must either:

1. Reduce the number of files your task opens at the same time – to be compatible with the requirements in Table B-1

2. Eliminate one or more COBOL-81 features shown in Table B-1

RSX-11M/M-PLUS systems support more than 14 LUNs. For more information refer to the *RSX-11M Task Builder Reference Manual.*

Table B-1: Logical Unit Assignments for COBOL-81 Features and Files Open at the Same Time

Maximum Number of Files Open at the Same Time	COBOL-81 Features Included in Task Image			
	ACCEPT/DISPLAY Extensions	ACCEPT from device-name	SORT/MERGE Statement(s)	COBOL-81 Symbolic Debugger
13	1			
13		1		
12	1	1		
10			3	
10	1		3	
10		1	3	
9				4
9	1	1	3	
9	1			4
9		1		4
6			3	4
5	1	1	3	4

Master Index

This Master Index contains a complete list of the references to subjects in the COBOL-81 Language Reference Manual and the four parts of the COBOL-81 User's Guide.

The index uses the following conventions:

Example	Explanation
1-8t	A page number followed by a t indicates a table.
4-6f	A page number followed by an f indicates a figure.

Entries in the Master Index are also preceded by an acronym indicating which manual, and part to a manual, the page number refers to:

Acronym	Title
LRM	COBOL-81 Language Reference Manual
RSTS/E UG I	COBOL-81 User's Guide, Part I for RSTS/E
RSX UG I	COBOL-81 User's Guide, Part I for RSX-11M/M-PLUS
UG II	COBOL-81 User's Guide, Part II
UG III	COBOL-81 User's Guide, Part III
UG IV	COBOL-81 User's Guide, Part IV

Where a subject references more than one manual and/or parts, references to the COBOL-81 Language Reference Manual appear first, followed in order by the COBOL-81 User's Guide Part I, then Part II, Part III, and Part IV.

A

Abbreviated combined relation conditions, *LRM* 5-20 to 5-21
Abbreviating DCL commands, *RSTS/E UG I* 1-2, *RSX UG I* 1-2
ACCEPT statement, *LRM* 5-34 to 5-45
  reference to devices, *LRM* 3-6
Access mode
  changing, *UG IV* 1-12
  default, *UG IV* 1-12
  dynamic, *UG IV* 1-12
  random, *UG IV* 1-12
  sequential, *UG IV* 1-12
ACCESS MODE clause, *LRM* 3-13 to 3-14
Access stream, *UG IV* 6-2
  initializing, *UG IV* 6-2
  terminating, *UG IV* 6-2
  types, *UG IV* 6-5
Accessing a table with SEARCH, *UG III* 3-17f
Accounts, *RSTS/E UG I* 1-3, *RSX UG I* 1-3
Active/inactive arguments
  inspecting data, *UG III* 2-35
ADD statement, *LRM* 5-46 to 5-47
Alignment, effect of SYNC clause, *LRM* 4-64 to 4-65
ALL literal figurative constant, *LRM* 1-8
ALLOWING clause, *UG IV* 6-2
ALPHABET clause, *LRM* 3-6
Alphabet-name, defined, *LRM* 1-5
ALPHABETIC test, *LRM* 5-16
ALTERNATE RECORD KEY clause, *LRM* 3-15
ANSI format, *LRM* 1-19 to 1-22, *RSTS/E UG I* 2-2, *RSX UG I* 2-2, *UG II* 5-3
/ANSI_FORMAT compiler qualifier, *RSTS/E UG I* 2-2, 3-2t, 3-3, *RSX UG I* 2-2, 3-2t, 3-3
APPEND, DCL command, *RSTS/E UG I* 1-6t, *RSX UG I* 1-6t
APPLY clause, *UG IV* 7-1
  general rules for, *LRM* 3-22 to 3-23
  syntax rules for, *LRM* 3-22
Area A
  in ANSI format, *LRM* 1-20

Area A (Cont.)
in terminal format, *LRM* 1-17
Area B
in ANSI format, *LRM* 1-20
in terminal format, *LRM* 1-17
Argument address list
function, *UG II* 6-16
general format, *UG II* 6-15f
using, *UG II* 6-15
Arithmetic expressions, *LRM* 5-12 to 5-14
composition of, *LRM* 5-12
data items in, *LRM* 5-12
evaluation of, *LRM* 5-12
literals in, *LRM* 5-12
operators in, *LRM* 5-12
processing, *UG III* 1-20
using parentheses in, *LRM* 5-12
using signs in, *LRM* 5-12
Arithmetic operations
multiple receiving fields, *LRM* 5-22
restrictions for operands, *LRM* 5-22
rounding off results in, *LRM* 5-22, *UG III* 1-16
storing partial results, *LRM* 5-22
Arithmetic operators, *LRM* 5-12
Arithmetic statements, *UG III* 1-15 to 1-20
advantages over COMPUTE, *UG III* 4-4
binary truncation of, *UG III* 1-15
common errors in, *UG III* 1-19
defined, *LRM* 5-22
instead of COMPUTE, *UG III* 4-4
intermediate results, *UG III* 1-15
with GIVING phrase, *UG III* 1-18
with SIZE ERROR phrase, *UG III* 1-17
ASCENDING phrase, *UG IV* 10-1
ASCII character set, *LRM* B-1 to B-2
octal and decimal equivalents, *LRM* B-1 to B-2
ASSIGN clause, *LRM* 3-16
ASSIGN, DCL command, *RSTS/E UG I* 1-5, *RSX UG I* 1-5
Asterisk delimiter
to unstring data, *UG III* 2-20t
Asterisk indicator character (*)
See also Comment character (*)
in ANSI format, *LRM* 1-19
in terminal format, *LRM* 1-16
At end condition, *LRM* 5-28
planning for, *UG IV* 5-2
AUTHOR paragraph, *LRM* 2-3
Auxiliary keypad keys, *UG IV* 9-17

B

Binary search
of a table, *LRM* 5-119
requirements for, *UG III* 3-15
results of using, *UG III* 3-16
with AT END statement, *UG III* 3-16
with keys, *UG III* 3-15
with multiple keys, *UG III* 3-16, 3-20f
Binary truncation, *UG III* 1-15
Blank lines
in ANSI format, *LRM* 1-21
in terminal format, *LRM* 1-17
BLANK WHEN ZERO clause, *LRM* 4-25
/BLD compiler switch, *RSTS/E UG I* D-3t, D-4, *RSX UG I* D-3t, D-4
BLDODL utility
command line format, *RSTS/E UG I* D-8, *RSX UG I* D-8
BLDODL utility switches
/CLU:, *RSTS/E UG I* D-8, *RSX UG I* D-8
/DEB, *RSTS/E UG I* D-8, *RSX UG I* D-8
/DIA, *RSTS/E UG I* D-10, *RSX UG I* D-10
improving program performance with, *UG II* 5-2
/IO:, *RSTS/E UG I* D-9, *RSX UG I* D-9
/IO:DECOV, *RSTS/E UG I* D-9, *RSX UG I* D-9
/IO:MEMRES, *RSTS/E UG I* D-9, *RSX UG I* D-9
/IO:NONOV, *RSTS/E UG I* D-9, *RSX UG I* D-9, *UG II* 4-5
/IO:USEROV, *RSTS/E UG I* D-9, *RSX UG I* D-9
/LRG, *RSTS/E UG I* D-9, *RSX UG I* D-9, *UG II* 4-5
/MAP, *RSTS/E UG I* D-8, *RSX UG I* D-8
/MER, *RSTS/E UG I* D-9, *RSX UG I* D-9
/OBJ, *RSTS/E UG I* D-9, *RSX UG I* D-9
/RES, *RSTS/E UG I* D-9, *RSX UG I* D-9
/ULIB, *RSTS/E UG I* D-8, *RSX UG I* D-8
Block
definition of, *UG IV* 7-8
logical, *UG IV* 7-8
physical, *UG IV* 7-8
related to a record, *LRM* 4-5
BLOCK CONTAINS clause, *LRM* 4-26 to 4-27
Block size limit, magnetic tape, *UG IV* 7-10
Bottom margin, *UG IV* 8-16

/-BOU compiler switch, *RSTS/E UG I* D-3t, D-5, *RSX UG I* D-3t, D-5
Boundary equivalence, *LRM* 4-10 to 4-14
Braces, use in general formats, *LRM* 1-13
Brackets, use in general formats, *LRM* 1-13
Bucket, *UG IV* 7-8
  related to physical record, *LRM* 4-5
  size, *UG IV* 7-8
Buffer areas, sharing, *UG IV* 7-7

C

C81, CCL/MCR command, *RSTS/E UG I* D-1, *RSX UG I* D-1
CALL statement, *LRM* 5-48 to 5-50
  effect on program logic, *UG II* 6-3
  nesting, *UG II* 6-4
  transferring program control, *UG II* 6-3
Called programs
  defined, *UG II* 6-1
  exiting from, *UG II* 6-3
  Linkage Section of, *LRM* 4-17, *UG II* 6-8
  Procedure Division header of, *LRM* 4-17
Calling programs
  accessing data items in, *UG II* 6-7
  COBOL-81 from MACRO, *UG II* 6-14
  MACRO from COBOL-81, *UG II* 6-13
CANCEL BREAKPOINT, Debugger command, *UG II* 3-2t, 3-8
Categories of data items, *LRM* 4-6
CCL commands
  C81, *RSTS/E UG I* D-2
  commas, use in compiler command line, *RSTS/E UG I* D-2
  compiler command line format, *RSTS/E UG I* D-2
  default file types, *RSTS/E UG I* D-2
  examples, *RSTS/E UG I* D-2
Cell
  contents, *UG IV* 3-1
  location in the file, *UG IV* 3-1
  numbering, *UG IV* 3-1
  relative record number, *UG IV* 3-1
  size, *UG IV* 3-1
Channel
  See Logical Unit Number (LUN)
Character attributes for terminal screen, *UG IV* 9-8
Character sets
  and collating sequence, *LRM* 3-7
  ASCII, *LRM* 3-7
  COBOL-81, *LRM* 1-2
Character sets (Cont.)
  computer, *LRM* 1-2
  in ALPHABET clause, *LRM* 3-6
  in CODE-SET clause, *LRM* 4-28
Character transfer
  using the STRING statement, *LRM* 5-135 to 5-139
  using the UNSTRING statement, *LRM* 5-143 to 5-148
Character-string, *LRM* 1-1
/CHECK compiler qualifier, *RSTS/E UG I* 3-2t, 3-3, *RSX UG I* 3-2t, 3-3
  for improving program performance, *UG II* 5-2
/CHECK:BOUNDS compiler qualifier, *RSTS/E UG I* 3-2t, 3-3, *RSX UG I* 3-2t, 3-3
/CHECK:NOBOUNDS compiler qualifier, *RSTS/E UG I* 3-2t, 3-3, *RSX UG I* 3-2t, 3-3
/CHECK:NOPERFORM compiler qualifier, *RSTS/E UG I* 3-2t, 3-3, *RSX UG I* 3-2t, 3-3
/CHECK:PERFORM compiler qualifier, *RSTS/E UG I* 3-2t, 3-3, *RSX UG I* 3-2t, 3-3
Choice indicators, use in general formats, *LRM* 1-14
/-CIS compiler switch, *RSTS/E UG I* D-3t, D-6, *RSX UG I* D-3t, D-6
/CIS compiler switch, *RSTS/E UG I* D-3t, D-4, *RSX UG I* D-3t, D-4
Class condition, *LRM* 5-16
Class tests, *UG III* 2-5
  numeric, *UG III* 1-10
Classes
  for nonnumeric data, *UG III* 2-4
  of data items, *LRM* 4-6
CLOSE statement, *LRM* 5-51 to 5-54
/CLU: BLDODL switch, *RSTS/E UG I* D-8, *RSX UG I* D-8
COBOL language elements, *LRM* 1-1
COBOL word, *LRM* 1-3
COBOL, DCL command, *RSTS/E UG I* 1-1, *RSX UG I* 1-1
COBOL-81 Symbolic Debugger
  See Debugger
CODE-SET clause, *LRM* 4-28
/CODE:[NO]CIS compiler qualifier, *RSTS/E UG I* 3-2t, 3-4, *RSX UG I* 3-2t, 3-4
Collating sequence
  as related to alphabet-name, *LRM* 3-5
  in ALPHABET clause, *LRM* 3-6

- Collating sequence (Cont.)
  - specifying in a COBOL program, *LRM* 3-3
  - when merging files, *LRM* 5-84
- Combining files
  - *See Merging files*
- Commas
  - as separators, *LRM* 1-11
  - using in C81 command line, *RSTS/E UG I* D-2, *RSX UG I* D-2
- Comment character (*), *RSTS/E UG I* 2-2, *RSX UG I* 2-2
- Comment lines
  - in ANSI format, *LRM* 1-21
  - in terminal format, *LRM* 1-17
- Common errors
  - in nonnumeric MOVE statements, *UG III* 2-10
  - in STRING statements, *UG III* 2-17
  - when inspecting data, *UG III* 2-47
  - when unstringing data, *UG III* 2-31
- COMP data items
  - as VAX-11 COBOL incompatibility, *LRM* D-1
- COMP SYNC data items, *LRM* 4-64 to 4-65
- Comparing operands, *LRM* 5-14, *UG III* 2-4
  - when alphabetic, *LRM* 5-16
  - when nonnumeric, *LRM* 5-15
  - when numeric, *LRM* 5-15, 5-16
- Compile-time environment, documenting, *LRM* 3-2
- Compiler
  - command line format, *RSTS/E UG I* 3-2, *RSX UG I* 3-2
  - diagnostics, *RSTS/E UG I* 3-7, *RSX UG I* 3-7
  - functions, *RSTS/E UG I* 3-1, *RSX UG I* 3-1
  - output files
    - OBJ, *RSTS/E UG I* 3-1, *RSX UG I* 3-1
    - SKL, *RSTS/E UG I* 3-1, *RSX UG I* 3-1
- Compiler implementation limitations, *RSTS/E UG I* A-1, *RSX UG I* A-1
- Compiler qualifiers
  - /ANSI_FORMAT, *RSTS/E UG I* 2-2, 3-2t, 3-3, *RSX UG I* 2-2, 3-2t, 3-3
  - /CHECK, *RSTS/E UG I* 3-2t, 3-3, *RSX UG I* 3-2t, 3-3
  - /CHECK:BOUNDS, *RSTS/E UG I* 3-2t, 3-3, *RSX UG I* 3-2t, 3-3
- Compiler qualifiers (Cont.)
  - /CHECK:NOBOUNDS, *RSTS/E UG I* 3-2t, 3-3, *RSX UG I* 3-2t, 3-3
  - /CHECK:NOPERFORM, *RSTS/E UG I* 3-2t, 3-3, *RSX UG I* 3-2t, 3-3
  - /CHECK:PERFORM, *RSTS/E UG I* 3-2t, 3-3, *RSX UG I* 3-2t, 3-3
  - /CODE:[NO]CIS, *RSTS/E UG I* 3-2t, 3-4, *RSX UG I* 3-2t, 3-4
  - /CROSS_REFERENCE, *RSTS/E UG I* 3-2t, 3-4, *RSX UG I* 3-2t, 3-4
  - /DEBUG, *RSTS/E UG I* 3-2t, 3-4, *RSX UG I* 3-2t, 3-4, *UG II* 3-2
  - /DIAGNOSTICS, *RSTS/E UG I* 3-2t, 3-5, *RSX UG I* 3-2t, 3-5
  - examples, *RSTS/E UG I* 3-6, *RSX UG I* 3-6
  - /LIST, *RSTS/E UG I* 3-2t, 3-5, 3-5t, *RSX UG I* 3-2t, 3-5, 3-5t
  - /NAMES, *RSTS/E UG I* 3-2t, 3-5, *RSX UG I* 3-2t, 3-5
  - /NOANSI_FORMAT, *RSTS/E UG I* 3-2t, 3-3t, *RSX UG I* 3-2t, 3-3t
  - /NOCHECK, *RSTS/E UG I* 3-2t, *RSX UG I* 3-2t
  - /NOCROSS_REFERENCE, *RSTS/E UG I* 3-2t, 3-4t, *RSX UG I* 3-2t, 3-4t
  - /NODEBUG, *RSTS/E UG I* 3-2t, 3-4t, *RSX UG I* 3-2t, 3-4t
  - /NODIAGNOSTICS, *RSTS/E UG I* 3-2t, 3-5t, *RSX UG I* 3-2t, 3-5t
  - /NOLIST, *RSTS/E UG I* 3-2t, *RSX UG I* 3-2t
  - /NOOBJECT, *RSTS/E UG I* 3-3t, 3-5, *RSX UG I* 3-3t, 3-5
  - /NOSHOW, *RSTS/E UG I* 3-3t, 3-5t, *RSX UG I* 3-3t, 3-5t
  - /NOSUBPROGRAM, *RSTS/E UG I* 3-3t, 3-5t, *RSX UG I* 3-3t, 3-5t
  - /NOTRUNCATE, *RSTS/E UG I* 3-3t, 3-6t, *RSX UG I* 3-3t, 3-6t
  - /NOWARNINGS, *RSTS/E UG I* 3-3t, 3-6, *RSX UG I* 3-3t, 3-6
  - /OBJECT, *RSTS/E UG I* 3-3t, 3-5, *RSX UG I* 3-3t, 3-5
  - /SHOW, *RSTS/E UG I* 3-3t, 3-5, *RSX UG I* 3-3t, 3-5
  - /SHOW:MAP, *RSTS/E UG I* 3-3t, 3-5, 3-5t, *RSX UG I* 3-3t, 3-5, 3-5t
  - /SUBPROGRAM, *RSTS/E UG I* 3-3t, 3-5, *RSX UG I* 3-3t, 3-5
  - /TEMPORARY, *RSTS/E UG I* 3-3t, 3-6, *RSX UG I* 3-3t, 3-6

- Compiler qualifiers (Cont.)
  - /TRUNCATE, *RSTS/E UG I* 3-3t, 3-6, *RSX UG I* 3-3t, 3-6
  - using to improve performance, *UG II* 5-1
  - /WARNINGS, *RSTS/E UG I* 3-6, *RSX UG I* 3-6
  - /WARNINGS:INFORMATIONAL, *RSTS/E UG I* 3-3t, 3-6, *RSX UG I* 3-3t, 3-6
  - /WARNINGS:NOINFORMATIONAL, *RSTS/E UG I* 3-3t, 3-6, *RSX UG I* 3-3t, 3-6
- Compiler switches
  - /BLD, *RSTS/E UG I* D-3t, D-4, *RSX UG I* D-3t, D-4
  - /-BOU, *RSTS/E UG I* D-3t, D-5, *RSX UG I* D-3t, D-5
  - /-CIS, *RSTS/E UG I* D-3t, D-6, *RSX UG I* D-3t, D-6
  - /CIS, *RSTS/E UG I* D-3t, D-4, *RSX UG I* D-3t, D-4
  - /CRF, *RSTS/E UG I* D-3t, D-4, *RSX UG I* D-3t, D-4
  - /CVF, *RSTS/E UG I* D-3t, D-4, *RSX UG I* D-3t, D-4
  - /DEB, *RSTS/E UG I* D-3t, D-5, *RSX UG I* D-3t, D-5
  - examples, *RSTS/E UG I* D-7, *RSX UG I* D-7
  - /FIPS:74, *RSTS/E UG I* D-3t, D-6, *RSX UG I* D-3t, D-6
  - /-INF, *RSTS/E UG I* D-3t, D-6, *RSX UG I* D-3t, D-6
  - /KER, *RSTS/E UG I* D-3t, D-6, *RSX UG I* D-3t, D-6
  - /MAP, *RSTS/E UG I* D-3t, D-5, *RSX UG I* D-3t, D-5
  - /-PER, *RSTS/E UG I* D-3t, D-6, *RSX UG I* D-3t, D-6
  - /-SKL, *RSTS/E UG I* D-3t, D-6, *RSX UG I* D-3t, D-6
  - /STA:VAX, *RSTS/E UG I* D-3t, D-5, *RSX UG I* D-3t, D-5
  - /SUB, *RSTS/E UG I* D-3t, D-5, *RSX UG I* D-3t, D-5
  - /TMP, *RSTS/E UG I* D-3t, D-7, *RSX UG I* D-3t, D-7
  - /TRU, *RSTS/E UG I* D-3t, D-5, *RSX UG I* D-3t, D-5
- Compiler-directing sentence, *LRM* 5-3
- Compiler-directing statement, *LRM* 5-3
- Compiling
  - main and subprograms, *UG II* 6-2
- Compiling (Cont.)
  - source programs, *RSTS/E UG I* 3-1, *RSX UG I* 3-1
- Complex conditions, *LRM* 5-18
- COMPUTE statement, *LRM* 5-55 to 5-56
- Computer character set, *LRM* B-1 to B-2
  - octal and decimal equivalents, *LRM* B-1 to B-2
- Concatenating items, *UG III* 2-11
- Concise Command Language (CCL)
  - *See CCL commands*
- Condition-name condition, *LRM* 5-17
- Condition-names
  - associating values with, *LRM* 4-71 to 4-73
  - defined, *LRM* 1-5
  - in general formats and rules, *LRM* 5-11
  - in SWITCH clause, *LRM* 3-6
  - qualifying, *LRM* 5-11
- Conditional expressions, *LRM* 5-14 to 5-18
  - class condition, *LRM* 5-16
  - combining, *LRM* 5-19
  - complex conditions, *LRM* 5-18
  - condition-name, *LRM* 5-17
  - evaluation of, *LRM* 5-21
  - negating, *LRM* 5-19
  - relation condition, *LRM* 5-14 to 5-16
  - sign condition, *LRM* 5-18
  - switch-status condition, *LRM* 5-18
- Conditional sentence, *LRM* 5-4
- Conditional statement, *LRM* 5-4
- Conditional variables
  - relation to condition-names, *LRM* 1-5
- Configuration Section, *LRM* 3-2
- Continuation character (-), *RSTS/E UG I* 2-2, *RSX UG I* 2-2
  - in Debugger commands, *UG II* 3-3
  - to continue DCL commands, *RSTS/E UG I* 1-2, *RSX UG I* 1-2
- CONTINUE, DCL command, *RSTS/E UG I* 1-2, *RSX UG I* 1-2
- Control footing, *UG IV* 8-4
- Control heading, *UG IV* 8-4
- CONTROL KEY IN clause, *UG IV* 9-17
- Controlling index, *UG III* 3-15
- Conventional report
  - line counter usage, *UG IV* 8-12
  - logical page, *UG IV* 8-10
  - makeup, *UG IV* 8-10
  - page advancing, *UG IV* 8-10
  - page-overflow condition, *UG IV* 8-11

Conventional report (Cont.)
printing the, *UG IV* 8-24
CONVERSION clause, *UG IV* 9-9, 9-24
COPY statement, *LRM* 6-1 to 6-5, *RSTS/E UG I* 2-3, *RSX UG I* 2-3
COPY, DCL command, *RSTS/E UG I* 1-6t, *RSX UG I* 1-6t
CORRESPONDING phrase, *LRM* 5-23
Counting characters in a data item, *LRM* 5-76 to 5-81
CREATE, DCL command, *RSTS/E UG I* 1-6t, *RSX UG I* 1-6t
/CRF compiler switch, *RSTS/E UG I* D-3t, D-4, *RSX UG I* D-3t, D-4
/CROSS_REFERENCE compiler qualifier, *RSTS/E UG I* 3-2t, 3-4, *RSX UG I* 3-2t, 3-4
CTRL/C
in DCL commands, *RSTS/E UG I* 1-2
in Debugger commands, *UG II* 3-11
CTRL/U
in DCL commands, *RSX UG I* 1-2
CURRENCY SIGN clause, *LRM* 3-7
Currency symbol
in PICTURE clause, *LRM* 4-47
in SPECIAL-NAMES paragraph, *LRM* 3-5
Cursor positioning keys, *UG IV* 9-17
/CVF compiler switch, *RSTS/E UG I* D-3t, D-4, *RSX UG I* D-3t, D-4

D

Data description
complete entry skeleton, *LRM* 4-22 to 4-24
elements of, *LRM* 1-25, 4-1
Data Division
entries, elements of, *LRM* 1-25
general format and rules, *LRM* 4-15
Data handling techniques
for improving program performance, *UG II* 5-3
Data items
assigning initial values to, *LRM* 4-71 to 4-73
categories of, *LRM* 4-6, 4-44
classes of, *LRM* 4-6
COMP-3, *UG III* 1-7
contents and class incompatibility, *LRM* 5-24
default initial values, *LRM* 4-16
DISPLAY, *UG III* 1-7
in arithmetic expressions, *LRM* 5-12
index, *LRM* 5-10, *UG III* 3-13

Data items (Cont.)
maximum size of PICTURE clause for a, *LRM* 4-43
naming, *LRM* 4-29
specifying characteristics of, *LRM* 4-22 to 4-24, 4-43 to 4-52
specifying nonstandard data positioning in, *LRM* 4-31
specifying storage format for, *LRM* 4-66 to 4-70
with DISPLAY usage, *UG III* 1-6
Data movement, *UG III* 2-5 to 2-47
with editing symbols, *UG III* 2-8f
with no editing, *UG III* 2-9f
Data organization, *UG III* 2-2
DATA RECORDS clause, *LRM* 4-30
Data storage
representation on media, *LRM* 4-28
word and byte representation, *UG III* 1-2f
Data testing, *UG III* 2-3 to 2-5
Data transfer
positioning rules for, *LRM* 4-6
using the MOVE statement, *LRM* 5-87 to 5-90
Data types
COMP, *UG III* 4-1
COMP compared to COMP SYNC, *UG III* 1-2, 1-7
COMP-3, *UG III* 1-5, 1-7, 4-2
scaling and mixing, *UG III* 4-2
Data-handling operations
undefined results
from incompatible data, *LRM* 5-24
from operand overlap, *LRM* 5-24
Data-name clause, *LRM* 4-29
Data-names
defined, *LRM* 1-5
in an identifier, *LRM* 5-11
using as subscripts, *UG III* 3-12
DCL commands
abbreviating, *RSTS/E UG I* 1-2, *RSX UG I* 1-2
APPEND, *RSTS/E UG I* 1-6t, *RSX UG I* 1-6t
ASSIGN, *RSTS/E UG I* 1-5, *RSX UG I* 1-5
COBOL, *RSTS/E UG I* 1-1, *RSX UG I* 1-1
CONTINUE, *RSTS/E UG I* 1-2
COPY, *RSTS/E UG I* 1-6t, *RSX UG I* 1-6t
CREATE, *RSTS/E UG I* 1-6t, *RSX UG I* 1-6t

DCL commands (Cont.)
DELETE, *RSTS/E UG I* 1-6t, *RSX UG I* 1-6t
DIRECTORY, *RSTS/E UG I* 1-6t, *RSX UG I* 1-6t
EDIT, *RSTS/E UG I* 1-1, 1-6t, *RSX UG I* 1-1, 1-6t
HELP, *RSTS/E UG I* 1-6, *RSX UG I* 1-6
LINK/C81, *RSTS/E UG I* 1-1, 4-1, *RSX UG I* 1-1, 4-1
RENAME, *RSTS/E UG I* 1-6t, *RSX UG I* 1-6t
RUN, *RSTS/E UG I* 1-1, *RSX UG I* 1-1
STOP, *RSTS/E UG I* 1-2
TYPE, *RSTS/E UG I* 1-6t, *RSX UG I* 1-6t
using continuation character (-) with, *RSTS/E UG I* 1-2, *RSX UG I* 1-2
/DEB BLDODL switch, *RSTS/E UG I* D-8, *RSX UG I* D-8
/DEB compiler switch, *RSTS/E UG I* D-3t, D-5, *RSX UG I* D-3t, D-5
/DEBUG compiler qualifier, *UG II* 3-2
with COBOL command, *RSTS/E UG I* 3-2t, 3-4, *RSX UG I* 3-2t, 3-4
/DEBUG qualifier
with LINK/C81 command, *RSTS/E UG I* 4-3, *RSX UG I* 4-3
Debugger
command line format, *UG II* 3-3
commands for position, *UG II* 3-4
I/O requirements, *UG II* 3-2
invoking, *UG II* 3-3
limitations, *UG II* 3-3
memory requirements, *UG II* 3-2
symbols file, *UG II* 3-2
using CTRL/C, *UG II* 3-11
Debugger commands
CANCEL BREAKPOINT command, *UG II* 3-2t, 3-8
DEFINE command, *UG II* 3-2t, 3-9
DISPLAY command, *UG II* 3-2t, 3-5
ASCII option, *UG II* 3-5
BYTE option, *UG II* 3-5
HELP command, *UG II* 3-2t, 3-5
MOVE command, *UG II* 3-2t, 3-6
PROCEED command, *UG II* 3-2t, 3-10
SET BREAKPOINT command, *UG II* 3-2t, 3-7
SHOW BREAKPOINTS command, *UG II* 3-2t, 3-8
SHOW SYNONYMS command, *UG II* 3-2t, 3-10
STOP command, *UG II* 3-2t, 3-11
UNDEFINE command, *UG II* 3-2t, 3-10
Decimal point
selecting for a program, *LRM* 3-5
specifying as comma, *LRM* 3-7
Decimal scaling position, *UG III* 1-6
Decimal truncation
reasons for avoiding, *UG III* 4-3
/TRUNCATE compiler qualifier, *UG III* 4-3
DECIMAL-POINT IS COMMA clause, *LRM* 3-7
Declarative procedures
examples, *UG IV* 5-7
EXTEND, *UG IV* 5-7
file name, *UG IV* 5-6
I-O, *UG IV* 5-7
INPUT, *UG IV* 5-7
OUTPUT, *UG IV* 5-7
referencing with the USE statement, *LRM* 5-149 to 5-150
sort, *UG IV* 10-6
using, *UG IV* 5-6
Declaratives
structure of, *LRM* 1-25
DEFAULT clause, *UG IV* 9-14
Default file types
See File types, default
DEFERRED-WRITE phrase of the APPLY clause, *LRM* 3-22
DEFINE, Debugger command, *UG II* 3-2t, 3-9
Defining tables, *UG III* 3-1 to 3-8
DELETE statement, *LRM* 5-57 to 5-58
DELETE, DCL command, *RSTS/E UG I* 1-6, *RSX UG I* 1-6t
Delimiters
as subscripts
sample results, *UG III* 2-17t
Delimiting
multiple receiving items, *UG III* 2-21t
with all asterisks, *UG III* 2-22t
with all double asterisks, *UG III* 2-23t
with two asterisks, *UG III* 2-22t
DESCENDING phrase, *UG IV* 10-1
Descriptions of relational operators, *UG III* 2-3f
Detail lines, *UG IV* 8-4
Devices
program references to, *LRM* 3-6
/DIA BLDODL switch, *RSTS/E UG I* D-10, *RSX UG I* D-10
Diagnostics
See Error messages

/DIAGNOSTICS compiler qualifier, *RSTS/E UG I* 3-2t, 3-5, *RSX UG I* 3-2t, 3-5
Directory
file specification for, *RSTS/E UG I* 1-3, *RSX UG I* 1-3
DIRECTORY, DCL command, *RSTS/E UG I* 1-6t, *RSX UG I* 1-6t
Disk libraries
See Libraries
DISPLAY option with SET BREAKPOINT Debugger command, *UG II* 3-7
DISPLAY statement, *LRM* 5-59 to 5-65
reference to devices, *LRM* 3-6
DISPLAY, Debugger command, *UG II* 3-2t, 3-5
ASCII option, *UG II* 3-5
BYTE option, *UG II* 3-5
DIVIDE statement, *LRM* 5-66 to 5-68
Division by zero, *LRM* 5-23
Divisions, in a COBOL program, *LRM* 1-23
Duplicate keys, not allowing, *LRM* D-9
DUPLICATES IN ORDER phrase, *UG IV* 10-4

E

EDIT, DCL command, *RSTS/E UG I* 1-1, 1-6t, *RSX UG I* 1-1, 1-6t
Edited moves
nonnumeric data, *UG III* 2-8
Editing rules
for numeric data, *UG III* 1-13
for PICTURE clause, *LRM* 4-47
Editing symbols
for numeric data, *UG III* 1-13
in PICTURE clause, *LRM* 4-44 to 4-47
Efficiency of indexing, *UG III* 4-3
Elementary data items
defined, *LRM* 4-2
nonnumeric, *UG III* 2-2
specifying alternative groupings of, *LRM* 4-60 to 4-61
Elementary moves, *LRM* 5-88, *UG III* 1-11
legal, *UG III* 2-7t
nonnumeric, *UG III* 2-7
numeric, *UG III* 1-11
numeric edited, *UG III* 1-13
Ellipsis, in general formats, *LRM* 1-14
Environment Division
syntax and general rules, *LRM* 3-1
Erasing
a line on the terminal screen, *UG IV* 9-3
entire terminal screen, *UG IV* 9-3
to end of line on terminal screen, *UG IV* 9-3
to end of terminal screen, *UG IV* 9-3
Error handling
with the USE statement, *LRM* 5-149 to 5-150
Error messages
limitations, *RSTS/E UG I* 3-7, *RSX UG I* 3-7
link-time, *RSTS/E UG I* 4-3, *RSX UG I* 4-3
run-time, *RSTS/E UG I* 5-2, C-1, *RSX UG I* 5-2, C-1
types of, *RSTS/E UG I* 3-7, *RSX UG I* 3-7
Errors
in arithmetic statements, *UG III* 1-19
in MOVE statements, *UG III* 1-14
in size, *UG III* 1-17
Escape sequences, *UG IV* A-1
as incompatibility with VAX-11 COBOL, *LRM* D-9
Execution control, transferring with CALL statement, *UG II* 6-3
EXIT PROGRAM statement, *LRM* 5-70
effect on program logic, *UG II* 6-3
format, *UG II* 6-3
returning control, *UG II* 6-3
EXIT statement, *LRM* 5-69
Exponentiation, *LRM* 5-13
results when invalid, *LRM* 5-23
Expression processing
arithmetic, *UG III* 1-20
EXTENSION phrase, *LRM* 3-23

F

FAB
See File Access Block
Fatal diagnostics, *RSTS/E UG I* 3-7, *RSX UG I* 3-7
FD
See File description
Figurative constants, *LRM* 1-8 to 1-9
ALL literal, *LRM* 1-8
HIGH-VALUE, *LRM* 1-8

Figurative constants (Cont.)
  LOW-VALUE, *LRM* 1-8
  QUOTE, QUOTES, *LRM* 1-8
  SPACE, *LRM* 1-8
  ZERO, *LRM* 1-8
File
  connector, *UG IV* 1-9
  defining a disk, *UG IV* 1-9
  defining a magnetic tape, *UG IV* 1-9
  handling, *UG IV* 1-8
  identifying, *UG IV* 1-9
  multiple openings in same program, *UG IV* 1-13
  opening and closing a, *UG IV* 1-13
  optimization, *UG IV* 7-1
  protection level, *UG IV* 6-2
  system, *UG IV* 6-2
File access (OPEN statement), *LRM* 5-93 to 5-97
File Access Block (FAB), *LRM* 3-24 to 3-25
File attributes
  defining, *UG IV* 1-3
File description
  clauses of, *LRM* 4-15
  complete entry skeleton, *LRM* 4-18 to 4-20
  purpose of, *LRM* 4-1
  structure of, *LRM* 4-15
File mapping, *LRM* 3-23
File name, specifying, *RSTS/E UG I* 1-3, *RSX UG I* 1-3
File optimization
  for improving I/O performance, *UG II* 5-3
  using I-O-CONTROL paragraph, *LRM* 3-21 to 3-25
File organization, *UG IV* 1-1
  advantages and disadvantages, *UG IV* 1-2
  default, *UG IV* 1-11
  indexed, *UG IV* 1-11
  relative, *UG IV* 1-11
  sequential, *UG IV* 1-11
  specifying, *UG IV* 1-3
File Section, *LRM* 4-15
File sharing, *UG IV* 6-1
  common file status values, *UG IV* 6-8
  common RMS-11 completion codes, *UG IV* 6-8
  requirements, *UG IV* 6-3
File specification, *RSTS/E UG I* 1-3, *RSX UG I* 1-3
File specification
  assigning
    with ASSIGN clause, *LRM* 3-16
    with VALUE OF ID clause, *LRM* 4-74
  examples, *RSTS/E UG I* 1-3, *RSX UG I* 1-3
  format, *RSTS/E UG I* 1-3, *RSX UG I* 1-3
  how RMS-11 builds a COBOL, *UG IV* 1-13
  keeping as a variable, *UG IV* 1-10
  overriding at run-time, *UG IV* 1-10
  variable, *UG IV* 1-10
File status
  data item, *LRM* 5-24
  values
    complete list of, *LRM* C-1 to C-2
    for COBOL-81, *UG IV* 5-3
    for RMS-11, *UG IV* 5-5
FILE STATUS clause, *LRM* 3-17
File structure
  specifying in a COBOL program, *LRM* 3-18
File types, default
  examples, *RSTS/E UG I* 1-5, *RSX UG I* 1-5
  for object file, *RSTS/E UG I* 3-1, *RSX UG I* 3-1
  for skeleton descriptor file, *RSTS/E UG I* 3-1, *RSX UG I* 3-1
FILE-CONTROL paragraph, *LRM* 3-10 to 3-12
File-handling
  specifying input-output status, *LRM* 3-17
File-names
  assigning file specifications to, *LRM* 3-16
  defined, *LRM* 1-5
Fill bytes, defined, *LRM* 4-7
FILL-SIZE phrase of the APPLY clause, *LRM* 3-23
FILLER data items, *LRM* 4-29
/FIPS:74 compiler switch, *RSTS/E UG I* D-3t, D-6, *RSX UG I* D-3t, D-6
Fixed insertion editing, *LRM* 4-48
Fixed-length records, *LRM* 4-53 to 4-55
Floating insertion editing, *LRM* 4-49
/FMS:NORESIDENT qualifier, *RSTS/E UG I* 4-2
/FMS:RESIDENT qualifier, *RSTS/E UG I* 4-2
Footing area, *UG IV* 8-16
Form control bytes, *UG IV* 1-8

Format
  of print files, *LRM* 4-34 to 4-37
  record (RECORD clause), *LRM* 4-53 to 4-55
Format conversion
  ANSI to terminal, *UG II* 1-1
  terminal to ANSI, *UG II* 1-3
Format, source program
  See Source program reference formats
Format, syntax
  See General format
FROM option of statements, *LRM* 5-28 to 5-29

G

General format
  defined, *LRM* 1-12
  function of, *LRM* 1-26
  notation used in, *LRM* 1-12
General rules, defined, *LRM* 1-26
Generic term, defined, *LRM* 1-26
GIVING phrase
  in SORT statement, *UG IV* 10-2
Global entry point, *UG II* 6-13
GO TO DEPENDING phrase
  advantages of using, *UG III* 4-4
GO TO statement, *LRM* 5-71 to 5-72
Group data item, *LRM* 4-2
Group indicating, *UG IV* 8-29
Group items
  nonnumeric, *UG III* 2-2
Group moves, *LRM* 4-8, 5-89, *UG III* 1-11
  description, *UG III* 1-11
  nonnumeric data, *UG III* 2-7

H

/HELP switch
  with C81 command, *RSTS/E UG I* D-2, *RSX UG I* D-2
  with BLDODL utility, *RSTS/E UG I* D-7, *RSX UG I* D-7
HELP, DCL command, *RSTS/E UG I* 1-2, 1-6, *RSX UG I* 1-6
  example, *RSTS/E UG I* 1-2, *RSX UG I* 1-2
HELP, Debugger command, *UG II* 3-2t, 3-5
HIGH-VALUE figurative constant, *LRM* 1-8, 3-7
Horizontal tab, *LRM* 1-12

Hyphen indicator character (-)
  See also Continuation character (-)
  in ANSI format, *LRM* 1-19
  in terminal format, *LRM* 1-16

I

I-O status
  See Input-output status
I-O-CONTROL paragraph, *LRM* 3-21 to 3-25
Identification area
  in ANSI format, *LRM* 1-20
  in terminal format, *LRM* 1-16
Identification Division
  syntax and general rules for, *LRM* 2-1 to 2-3
Identifiers
  defined, *LRM* 5-11
  subscripted data-name, *LRM* 5-9
Identifying a subprogram, *UG II* 6-2
  with /SUBPROGRAM compiler qualifier, *UG II* 6-2
  with USING phrase, *UG II* 6-2
Identifying table elements, *UG III* 3-10 to 3-20
IF statement, *LRM* 5-73 to 5-75
Illegal values for numeric data items, *UG III* 1-9
Image size and performance trade offs, *UG II* 5-1
Imperative sentence, *LRM* 5-4
Imperative statement, *LRM* 5-3
Improving I/O performance, *UG II* 5-1, 5-3, *UG IV* 1-2
Improving program performance
  /CHECK compiler qualifier, *UG II* 5-2
  /NOCHECK compiler qualifier, *UG II* 5-2
  /TEMPORARY compiler qualifier, *UG II* 5-2
  using BLDODL switches, *UG II* 5-2
  using compiler qualifiers, *UG II* 5-1, 5-2
  using data handling techniques, *UG II* 5-3
  using terminal format, *UG II* 5-3
Indentation, relation to level-numbers, *LRM* 4-3

Index data items, *LRM* 5-10, *UG III* 3-12, 3-13, 3-14
  as VAX-11 COBOL incompatibility, *LRM* D-1
  comparing, *LRM* 5-16
  declaration, *UG III* 3-13
  defining in program, *LRM* 4-67
  modifying with SET, *UG III* 3-13
  where defined, *UG III* 3-3
Index-names
  comparing, *LRM* 5-16
  defined, *LRM* 1-5
  rules associated with, *LRM* 4-39
  storing value of in a data item, *LRM* 5-10
Indexed file
  access modes, *UG IV* 4-3
  alternate key, *UG IV* 4-1
  at end condition, handling, *UG IV* 5-2
  backing up an, *UG IV* 4-15
  bucket, *UG IV* 7-8
  bucket size, *UG IV* 7-8
  bucket size calculation, *UG IV* 7-24
  buffer size calculation, *UG IV* 7-23
  CONTIGUOUS PREALLOCATION, *UG IV* 7-3
  corrupt, fixing a, *UG IV* 4-15
  creating an, *UG IV* 4-4
  default number of I-O buffers for, *LRM* 3-20
  DEFERRED-WRITE, *UG IV* 7-1
  defining an, *UG IV* 4-4
  design considerations, *UG IV* 4-2
  EXTENSION, *UG IV* 7-2
  file status values, using, *UG 'V* 5-3
  FILL-SIZE, *UG IV* 7-2
  I/O statements, *UG IV* 4-3
  index, *UG IV* 4-2
  invalid key condition, handling the, *UG IV* 5-2
  key, *UG IV* 4-1
  key length, *UG IV* 4-1
  key location, *UG IV* 4-1
  MASS-INSERT, *UG IV* 7-2
  open modes, *UG IV* 4-3
  optimization techniques, *UG IV* 7-1
  optional key, *UG IV* 4-1
  organization, *UG IV* 4-1
    advantages, *UG IV* 1-2
    disadvantages, *UG IV* 1-2
  population, initial, *UG IV* 4-5
  PREALLOCATION, *UG IV* 7-2
  primary key, *UG IV* 4-1
  reading an, *UG IV* 4-6
Indexed file (Cont.)
  recreating an, *UG IV* 4-15
  reorganization of, *UG IV* 4-5
  reserving buffer areas, *UG IV* 7-6
  starting position in, *LRM* 5-131 to 5-133
  updating an, *UG IV* 4-10
  WINDOW, *UG IV* 7-3
Indexes, *UG III* 3-2
  initializing, *UG III* 3-13
    with SET statement, *UG III* 3-13
  setting values for, *LRM* 5-117, 5-124 to 5-125
Indexing, *LRM* 5-10 to 5-11
  advantages, *UG III* 4-3
  basis for, *LRM* 4-38
  efficiency order, *UG III* 4-3
  in an identifier, *LRM* 5-11
  versus subscripting, *UG III* 4-3
Indicator character
  in ANSI format, *LRM* 1-19
  in terminal format, *LRM* 1-16
/-INF compiler switch, *RSTS/E UG I* D-3t, D-6, *RSX UG I* D-3t, D-6
Informational diagnostics, *RSTS/E UG I* 3-7, *RSX UG I* 3-7
Initializing
  alphanumeric items, *UG III* 3-9f
  data item values
    in Linkage Section, *LRM* 4-17
    in Working-Storage Section, *LRM* 4-16
  mixed usage items, *UG III* 3-9f
  tables, *UG III* 3-8 to 3-10
INPUT PROCEDURE phrase, usage, *UG IV* 10-2
Input-output
  of low-volume data
    using ACCEPT statement, *LRM* 5-34 to 5-45
    using DISPLAY statement, *LRM* 5-59 to 5-65
  specifying buffers for, *LRM* 3-20
  status, *LRM* 5-24 to 5-27
    specifying in a program, *LRM* 3-17
    values for, *LRM* 5-24
Input-Output Section, *LRM* 3-10
INSPECT statement, *LRM* 5-76 to 5-81
  using, *UG III* 2-31
Inspecting data
  active/inactive arguments, *UG III* 2-35
  BEFORE/AFTER phrase, *UG III* 2-32
  common errors when, *UG III* 2-47
  example, *UG III* 2-35f

Inspecting data (Cont.)
- finding a match, *UG III* 2-36
- implicit redefinition, *UG III* 2-33
- INSPECT operation, *UG III* 2-34
- interference in tally argument list, *UG III* 2-40
- matching delimiter characters, *UG III* 2-33f
- replacing phrase, *UG III* 2-43
- results of implicit redefinition, *UG III* 2-34t
- results of separate scan tallies, *UG III* 2-40f
- setting the scanner, *UG III* 2-35
- subscripted items, *UG III* 2-37
- tally argument, *UG III* 2-38
- tally counter, *UG III* 2-38
- TALLYING phrase, *UG III* 2-37

Interference
- in replacement argument list, *UG III* 2-46
- in tally argument list, *UG III* 2-40

Intermediate data item, *LRM* 5-22
- size of, *LRM* 5-22

Intermediate results
- for arithmetic statements, *UG III* 1-15

Interrupting DCL commands, *RSTS/E UG I* 1-2

INTO phrase, *LRM* 5-29

Invalid decimal data, detecting, *LRM* D-7

Invalid key condition, *LRM* 5-27
- planning for, *UG IV* 5-2

/IO: BLDODL switch, *RSTS/E UG I* D-9, *RSX UG I* D-9

/IO:DECOV BLDODL switch, *RSTS/E UG I* D-9, *RSX UG I* D-9

/IO:MEMRES BLDODL switch, *RSTS/E UG I* D-9, *RSX UG I* D-9

/IO:NONOV BLDODL switch, *RSTS/E UG I* D-9, *RSX UG I* D-9
- for nonoverlayable RMS-11, *UG II* 5-2

/IO:USEROV BLDODL switch, *RSTS/E UG I* D-9, *RSX UG I* D-9

J

JUSTIFIED clause, *LRM* 4-31
- related to Standard Alignment Rules, *LRM* 4-6

Justified moves, *UG III* 2-9

K

/KER compiler switch, *RSTS/E UG I* D-3t, D-6, *RSX UG I* D-3t, D-6

Key word, *LRM* 1-6

Keys
- ascending, *UG III* 3-2, 3-5
- descending, *UG III* 3-2, 3-5

L

LABEL RECORDS clause, *LRM* 4-32

Left-to-right storage allocation
- compared to major-minor storage allocation, *LRM* 4-8
- defined, *LRM* 4-7

Level indicators, *LRM* 1-25

Level-numbers, *LRM* 4-2 to 4-4, 4-33
- 66, *LRM* 4-3, 4-33
- 77, *LRM* 4-4, 4-33
- 88, *LRM* 4-4, 4-33
- defined, *LRM* 1-5
- for records, *LRM* 4-2
- 01 through 49, *LRM* 4-2

Libraries, *UG II* 4-2 to 4-6
- advantages of
  - clustering option, *UG II* 4-3
  - disk, *UG II* 4-2
  - resident, *UG II* 4-3
- defined, *UG II* 4-2
- using
  - disk, *UG II* 4-5
  - resident, *UG II* 4-6
  - to reduce task size, *UG II* 4-2 to 4-6

Library file, *RSTS/E UG I* D-11, *RSX UG I* D-11
- C81CIS, *RSTS/E UG I* 4-2, *RSX UG I* 4-2
- C81LIB, *RSTS/E UG I* 4-2, *RSX UG I* 4-2

Library text, copying into source program, *LRM* 6-1 to 6-5

LINAGE clause, *LRM* 4-34 to 4-37
- usage, *UG IV* 8-16

LINAGE-COUNTER, *LRM* 1-7, 4-35
- special register, *UG IV* 8-17
- usage, *UG IV* 8-21

Linage-file report
- bottom margin, *UG IV* 8-16
- footing area, *UG IV* 8-16
- logical page, *UG IV* 8-15
- makeup, *UG IV* 8-15
- page advancing, *UG IV* 8-17
- page body, *UG IV* 8-16

Linage-file report (Cont.)
page-overflow condition, *UG IV* 8-17
printing the, *UG IV* 8-24, 8-25
top margin, *UG IV* 8-16
Line continuation
in ANSI format, *LRM* 1-20
in terminal format, *LRM* 1-17
Line length
in terminal format, *LRM* 1-18
Linear search
See Sequential search
LINK/C81 qualifiers
/DEBUG, *RSTS/E UG I* 4-3, *RSX UG I* 4-3
/FMS:NORESIDENT, *RSTS/E UG I* 4-2
/FMS:RESIDENT, *RSTS/E UG I* 4-2
/MAP, *RSTS/E UG I* 4-3, *RSX UG I* 4-3
/NODEBUG, *RSTS/E UG I* 4-3, *RSX UG I* 4-3
/NOMAP, *RSTS/E UG I* 4-3, *RSX UG I* 4-3
/OTS, *UG II* 4-6
/OTS:NORESIDENT, *RSTS/E UG I* 4-2, *RSX UG I* 4-2
/OTS:RESIDENT, *RSTS/E UG I* 4-2, *RSX UG I* 4-2
/RMS, *UG II* 4-6
/RMS:NORESIDENT, *RSTS/E UG I* 4-2, *RSX UG I* 4-2
/RMS:RESIDENT, *RSTS/E UG I* 4-2, *RSX UG I* 4-2
LINK/C81, DCL command, *RSTS/E UG I* 1-1, 4-1, *RSX UG I* 1-1, 4-1
functions, *RSTS/E UG I* 4-1, *RSX UG I* 4-1
Linkage Section, *LRM* 4-17, *UG II* 6-8
contents, *UG II* 6-8
function, *UG II* 6-8
/LIST compiler qualifier, *RSTS/E UG I* 3-2t, 3-5, 3-5t, *RSX UG I* 3-2t, 3-5, 3-5t
Literal subscripts
accessing tables, *UG III* 3-10
defined, *UG III* 3-10
Literals, *LRM* 1-9 to 1-10
in arithmetic expressions, *LRM* 5-12
nonnumeric, *LRM* 1-10
numeric, *LRM* 1-9
Location equivalence, *LRM* 4-8 to 4-14
Locking operations on files, *LRM* 5-51 to 5-54
Logical block, *UG IV* 7-8
Logical data characteristics, *LRM* 4-1
Logical names, *RSTS/E UG I* 1-5, *RSX UG I* 1-5, *UG IV* 1-11
Logical page
defined, *UG IV* 8-6
horizontal spacing on the, *UG IV* 8-6
structure, *UG IV* 8-6
vertical spacing on the, *UG IV* 8-6
Logical records, mapping to physical records, *LRM* 4-26 to 4-27
Logical Unit Number (LUN), *UG IV* B-1
LOW-VALUE figurative constant, *LRM* 1-8, 3-7
Lowercase letters, compiler treatment of, *LRM* 1-3
Lowercase words, use in general formats, *LRM* 1-13
/LRG BLDODL switch, *RSTS/E UG I* D-9, *RSX UG I* D-9
for overlayable RMS-11, *UG II* 5-2
LUN
See Logical Unit Number (LUN)

M

MACRO programs
calling, *UG II* 6-13
calling command format, *UG II* 6-14
global entry point, *UG II* 6-13
in COBOL-81 programs, *UG II* 6-13
Magnetic tape
block size limit, *UG IV* 7-10
Main program
defined, *UG II* 6-1
Major-minor storage allocation, *LRM* 4-8 to 4-14
compared to left-to-right storage allocation, *LRM* 4-8
/MAP BLDODL switch, *RSTS/E UG I* D-8, *RSX UG I* D-8
obtaining memory allocation map, *UG II* 4-11
/MAP compiler switch, *RSTS/E UG I* D-3t, D-5, *RSX UG I* D-3t, D-5
/MAP qualifier, *RSTS/E UG I* 4-3, *RSX UG I* 4-3
Mapping a simple table into memory, *UG III* 3-5f
Margin A, *LRM* 1-19
Margin B, *LRM* 1-19
Margin C, *LRM* 1-19
Margin L, *LRM* 1-19
Margin R, *LRM* 1-19
MASS-INSERT phrase of the APPLY clause, *LRM* 3-23
Matching arguments
inspecting data, *UG III* 2-36

Matching delimiter characters, *UG III* 2-33f
MCR commands
  C81, *RSX UG I* D-2
  commas, use in compiler command line, *RSX UG I* D-2
  compiler command line format, *RSX UG I* D-2
  default file types, *RSX UG I* D-2
  examples, *RSX UG I* D-2
Memory allocation
  segmented program, *UG II* 4-9f
Memory allocation map
  example, *UG II* 4-13f
  obtaining, using /MAP switch, *UG II* 4-11
  reading, *UG II* 4-11
MEMORY SIZE clause, *LRM* 3-3
/MER BLDODL switch, *RSTS/E UG I* D-9, *RSX UG I* D-9
MERGE statement, *LRM* 5-82 to 5-86
  sample program, *UG IV* 10-7, 10-8
  using, *UG IV* 10-7
Merging files, *LRM* 4-21, 5-82 to 5-86
  using the RETURN statement, *LRM* 5-112 to 5-113
Meta-language, *LRM* 1-12
Mnemonic-names, *LRM* 3-5
  defined, *LRM* 1-5
Monitor Console Routine (MCR)
  See MCR commands
MOVE statement, *LRM* 5-87 to 5-90, *UG III* 1-11, 2-6
  common errors, *UG III* 1-14
MOVE, Debugger command, *UG II* 3-2t, 3-6
Multiple delimiters
  for unstringing data, *UG III* 2-23, 2-24t
Multiple operands
  in ADD and SUBTRACT statements, *UG III* 1-18
Multiple program task
  defined, *UG II* 6-1
Multiple receiving items
  for arithmetic operations, *LRM* 5-22
  for unstringing data, *UG III* 2-18
Multiple record definitions, *LRM* 4-33
Multiple results
  See Multiple receiving items
Multiple sending items
  for stringing data, *UG III* 2-11
Multiple-key binary search, *UG III* 3-16
MULTIPLY statement, *LRM* 5-91 to 5-92

N

/NAMES compiler qualifier, *RSTS/E UG I* 3-2t, 3-5, *RSX UG I* 3-2t, 3-5
  using to compile subprograms, *UG II* 6-2
Naming a COBOL program, *LRM* 2-1 to 2-2
Naming files in a COBOL program, *LRM* 3-11
Nesting CALL statements, *UG II* 6-4
NO ECHO clause, *UG IV* 9-13
/NOANSI_FORMAT compiler qualifier, *RSTS/E UG I* 3-2t, 3-3t, *RSX UG I* 3-2t, 3-3t
/NOCHECK compiler qualifier, *RSTS/E UG I* 3-2t, *RSX UG I* 3-2t
  for improving program performance, *UG II* 5-2
/NOCROSS_REFERENCE compiler qualifier, *RSTS/E UG I* 3-2t, 3-4t, *RSX UG I* 3-2t, 3-4t
/NODEBUG compiler qualifier, *RSTS/E UG I* 3-2t, 3-4t, *RSX UG I* 3-2t, 3-4t
  with LINK/C81 command, *RSTS/E UG I* 4-3, *RSX UG I* 4-3
/NODIAGNOSTICS compiler qualifier, *RSTS/E UG I* 3-2t, 3-5t, *RSX UG I* 3-2t, 3-5t
/NOLIST compiler qualifier, *RSTS/E UG I* 3-2t, *RSX UG I* 3-2t
/NOMAP qualifier, *RSTS/E UG I* 4-3, *RSX UG I* 4-3
Non-COBOL-81 programs
  including in a task, *UG II* 6-12
Non-overlayable RMS-11 routines
  using /IO:NONOV compiler switch, *UG II* 5-2
Nonnumeric data
  classes of, *UG III* 2-4
  concatenating items, *UG III* 2-11
  edited moves, *UG III* 2-8
  elementary moves, *UG III* 2-7, 2-7t
  group moves, *UG III* 2-7
  justified moves, *UG III* 2-9
  organization of, *UG III* 2-2
  receiving items, *UG III* 2-9
  special characters, *UG III* 2-2
  STRING statement, *UG III* 2-11
  subscripted moves, *UG III* 2-10
  transferring
    with MOVE CORRESPONDING statement, *UG III* 2-10
    with MOVE statement, *LRM* 5-87 to 5-90, *UG III* 2-6

Nonnumeric data
transferring (Cont.)
with the ACCEPT statement, *LRM* 5-40
with the DISPLAY statement, *LRM* 5-63
with the STRING statement, *LRM* 5-135 to 5-139
Nonnumeric data items
elementary, *UG III* 2-2
testing, *UG III* 2-3
Nonnumeric literals, *LRM* 1-10
/NOOBJECT compiler qualifier, *RSTS/E UG I* 3-3t, 3-5, *RSX UG I* 3-3t, 3-5
/NOSHOW compiler qualifier, *RSTS/E UG I* 3-3t, 3-5t, *RSX UG I* 3-3t, 3-5t
/NOSUBPROGRAM compiler qualifier, *RSTS/E UG I* 3-3t, 3-5t, *RSX UG I* 3-3t, 3-5t
/NOTRUNCATE compiler qualifier, *RSTS/E UG I* 3-3t, 3-6t, *RSX UG I* 3-3t, 3-6t
/NOWARNINGS compiler qualifier, *RSTS/E UG I* 3-3t, 3-6, *RSX UG I* 3-3t, 3-6
Numeric class tests, *UG III* 1-10
Numeric data
class test, *UG III* 1-9
compared, *UG III* 1-6
illegal values in, *UG III* 1-9
optimizing, *UG III* 4-1
relation test, *UG III* 1-9
representation of, *UG III* 4-1
sign test, *UG III* 1-9
storage of, *UG III* 1-1
testing, *UG III* 1-9
transferring
with the ACCEPT statement, *LRM* 5-40
with the DISPLAY statement, *LRM* 5-63
with the MOVE statement, *LRM* 5-87 to 5-90
Numeric data items
maximum number of digit positions, *LRM* 4-45
Numeric data types
comparing efficiency, *UG III* 4-1t
Numeric edited data items
contents, *UG III* 1-13
description, *UG III* 1-13
example of, *UG III* 1-14f
maximum number of digit positions, *LRM* 4-45
rules for, *UG III* 1-13
Numeric edited moves
elementary, *UG III* 1-13
Numeric editing
symbols, *UG III* 1-13
Numeric literals, *LRM* 1-9
Numeric moves
elementary, *UG III* 1-11
NUMERIC test, *LRM* 5-16

O

OBJ file type, *RSTS/E UG I* 3-1, *RSX UG I* 3-1
/OBJ BLDODL switch, *RSTS/E UG I* D-9, *RSX UG I* D-9
/OBJECT compiler qualifier, *RSTS/E UG I* 3-3t, 3-5, *RSX UG I* 3-3t, 3-5
Object Time System
See OTS (Object Time System)
OBJECT-COMPUTER paragraph, *LRM* 3-3 to 3-4
OCCURS clause, *LRM* 4-38 to 4-42
options
indexes, *UG III* 3-2
keys, *UG III* 3-2
related to subscripting, *LRM* 5-8
ON EXCEPTION clause, *UG IV* 9-9
Open mode
EXTEND, *UG IV* 1-13
I-O, *UG IV* 1-13
INPUT, *UG IV* 1-13
OUTPUT, *UG IV* 1-13
OPEN statement, *LRM* 5-93 to 5-97
effect on LINAGE values, *LRM* 4-35
Optional words, *LRM* 1-6
ORGANIZATION clause, *LRM* 3-18
OTS (Object Time System)
diagnostics, *RSTS/E UG I* 5-2, *RSX UG I* 5-2
error checking, *UG II* 6-12
functions, *RSTS/E UG I* 5-1, *RSX UG I* 5-1
/OTS:NORESIDENT qualifier, *RSTS/E UG I* 4-2, *RSX UG I* 4-2
/OTS:RESIDENT qualifier, *RSTS/E UG I* 4-2, *RSX UG I* 4-2
OUTPUT PROCEDURE phrase, usage, *UG IV* 10-2
Overflow statements
sample, *UG III* 2-15t
Overlapping operands, *LRM* 5-24
Overlayable RMS-11 routines
using /LRG switch, *UG II* 5-2

P

Packed-decimal data format, *LRM* 4-68
Page
  logical, *UG IV* 8-6
  physical, *UG IV* 8-6
  size definition, *UG IV* 8-24
Page body, *UG IV* 8-16
Page footing, *UG IV* 8-4
Page heading, *UG IV* 8-4
Paragraph
  defined, *LRM* 1-24
  header, *LRM* 1-24
  in Procedure Division, *LRM* 5-32
Paragraph-names
  defined, *LRM* 1-5
  rules for, *LRM* 1-24
Parentheses, *LRM* 1-11
  in arithmetic expressions, *LRM* 5-12
/-PER compiler switch, *RSTS/E UG I* D-3t, D-6, *RSX UG I* D-3t, D-6
PERFORM statement, *LRM* 5-98 to 5-106
Performance, improving, *UG II* 5-1, *UG IV* 1-2
Period
  as a separator, *LRM* 1-11
  in general formats, *LRM* 1-15
Physical block, *UG IV* 7-8
Physical data characteristics, *LRM* 4-1
Physical page
  defined, *UG IV* 8-6
Physical records, mapping logical records to, *LRM* 4-26 to 4-27
PICTURE character-strings, *LRM* 1-11
PICTURE clause, *LRM* 1-11, 4-43 to 4-52
  editing methods for, *LRM* 4-47 to 4-51
  specifying the currency symbol, *LRM* 3-7
  symbol precedence rules for, *LRM* 4-51
Preallocation of disk blocks, *LRM* 3-23
PREALLOCATION phrase, *LRM* 3-23
PRINT command, for LINAGE files, *LRM* 4-36
Print file, *UG IV* 2-4
  format for sequential files, *LRM* 3-23, 4-34 to 4-37
PRINT-CONTROL phrase, *LRM* 3-23
Print-controlled file, *UG IV* 1-4, 1-8
Procedure Division
  header, *LRM* 5-32
Procedure-names
  defined, *LRM* 5-32
PROCEED, Debugger command, *UG II* 3-2t, 3-10
PROCEED, with SET BREAKPOINT command, *UG II* 3-7
Program execution
  terminating with STOP statement, *LRM* 5-134
Program function keys, *UG IV* 9-17
Program listing
  example, *UG II* 2-3 to 2-5
  explanation of, *UG II* 2-1 to 2-2
PROGRAM-ID paragraph, *LRM* 2-2
Program-name
  as incompatibility with VAX-11 COBOL, *LRM* D-9
  defined, *LRM* 1-5
Project-Programmer Number (PPN), *RSTS/E UG I* 1-3
PROTECTED clause, *UG IV* 9-11
PSECT names
  assigned by default, *UG II* 4-10
  uniqueness in subprograms, *UG II* 6-2
  using /NAMES:XX switch, *UG II* 4-9

Q

Qualification, *LRM* 5-6 to 5-8
  in an identifier, *LRM* 5-11
Qualifiers, compiler
  See Compiler qualifiers
Quotation marks, *LRM* 1-12
QUOTE figurative constant, *LRM* 1-8

R

RAB
  See Record Access Block
READ statement, *LRM* 5-107 to 5-110
Receiving items
  nonnumeric data, *UG III* 2-9
Record
  areas, sharing, *UG IV* 7-4
  as a logical concept, *LRM* 4-1
  as a physical concept, *LRM* 4-2
  attributes, *UG IV* 1-3
  blocking, specifying, *UG IV* 1-3
  cells, *UG IV* 3-1
  defining length of, *LRM* 4-53 to 4-55
  deleting from files, *LRM* 5-57 to 5-58
  fixed-length, *UG IV* 1-4
  format, *UG IV* 1-3
  locking, *UG IV* 6-1, 6-9
  maximum size, *UG IV* 1-4
  record-length field, *UG IV* 1-4
  size, *UG IV* 7-8

- Record
  - size (Cont.)
    - related to storage medium, *LRM* 4-26 to 4-27
  - space needs on a physical device, *UG IV* 1-3
  - specifying size, *UG IV* 1-3
  - unit of transfer for, *LRM* 4-5
  - unit size, *UG IV* 7-8
  - variable-length, *UG IV* 1-4
- Record access
  - by alternate key, *LRM* 3-15
  - by primary key, *LRM* 3-19
  - order of, *LRM* 3-13 to 3-14
  - using the READ statement, *LRM* 5-107 to 5-110
  - using the RELEASE statement, *LRM* 5-111
  - using the RETURN statement, *LRM* 5-112 to 5-113
  - using the START statement, *LRM* 5-131 to 5-133
- Record Access Block (RAB), *LRM* 3-24 to 3-25
- Record alignment boundaries, *LRM* 4-7
- Record allocation, *LRM* 4-7 to 4-14
- RECORD clause, *LRM* 4-53 to 4-55
- Record description
  - hierarchical structure of, *LRM* 4-2
  - purpose of, *LRM* 4-1
- RECORD KEY clause, *LRM* 3-19
- Record transfer
  - using the WRITE statement, *LRM* 5-151 to 5-156
- Record-length descriptions, multiple, *UG IV* 1-8
- Record-name
  - defined, *LRM* 1-5
- Records
  - logical characteristics of, *LRM* 4-2
  - physical characteristics of, *LRM* 4-4
- REDEFINES clause, *LRM* 4-56 to 4-59
- Redefinition
  - implied when inspecting data, *UG III* 2-33
- Reducing compile time
  - using terminal format, *UG II* 5-3
- REFORMAT utility, *UG II* 5-3
  - error messages, *UG II* 1-4 to 1-5
  - executing, *UG II* 1-2, 1-3
- Relation condition, *LRM* 5-14 to 5-16
- Relation tests
  - description, *UG III* 1-9
  - equivalent sign tests, *UG III* 1-10t
- Relation tests (Cont.)
  - nonnumeric data, *UG III* 2-3
- Relational operators
  - defined, *LRM* 5-14
  - description of, *UG III* 2-3f
- Relative file
  - access modes, *UG IV* 3-3
  - at end condition, handling, *UG IV* 5-2
  - bucket, *UG IV* 7-8
  - bucket size calculation, *UG IV* 7-16
  - buffer size calculation, *UG IV* 7-16
  - capabilities, *UG IV* 3-2
  - CONTIGUOUS PREALLOCATION, *UG IV* 7-3
  - creating a, *UG IV* 3-4
  - default number of I-O buffers for, *LRM* 3-20
  - DEFERRED-WRITE, *UG IV* 7-1
  - defining a, *UG IV* 3-4
  - deleting records in a, *UG IV* 3-12
  - design considerations, *UG IV* 3-2
  - EXTENSION, *UG IV* 7-2
  - file status values, using, *UG IV* 5-3
  - I/O statements, *UG IV* 3-3
  - invalid key condition, handling the, *UG IV* 5-2
  - open modes, *UG IV* 3-3
  - optimization techniques, *UG IV* 7-1
  - organization, *UG IV* 3-1
    - advantages, *UG IV* 1-2
    - disadvantages, *UG IV* 1-2
  - PREALLOCATION, *UG IV* 7-2
  - reading a, *UG IV* 3-6
  - reserving buffer areas, *UG IV* 7-6
  - rewriting records in a, *UG IV* 3-9
  - RMS-11 allocation for a cell, *UG IV* 3-2
  - starting position in, *LRM* 5-131 to 5-133
  - tables, similarity to, *UG IV* 3-2
  - updating a, *UG IV* 3-9
  - using, *UG IV* 3-2
  - WINDOW, *UG IV* 7-3
- Relative indexing, *UG III* 3-13
  - system overhead, *UG III* 3-13
- Relative record number, *UG IV* 3-1
- RELEASE statement, *LRM* 5-111, *UG IV* 10-2
- Removal operations for file media, *LRM* 5-51 to 5-54
- RENAME, DCL command, *RSTS/E UG I* 1-6t, *RSX UG I* 1-6t
- RENAMES clause, *LRM* 4-60 to 4-61
- Replacement argument, *UG III* 2-45, 2-45f

Replacement argument list
  interference in, *UG III* 2-46
  to inspect data, *UG III* 2-45
Replacement value, *UG III* 2-45
Replacing characters in a data item, *LRM* 5-76 to 5-81
Replacing phrase
  to inspect data, *UG III* 2-43
Replacing records (with REWRITE statement), *LRM* 5-114 to 5-116
Report
  bolding items in a, *UG IV* 8-31
  bottom margin, *UG IV* 8-16
  components of a, *UG IV* 8-4
  control footing, *UG IV* 8-4
  control heading, *UG IV* 8-4
  conventional, *UG IV* 8-10
  crossfoot totals, *UG IV* 8-8
  design, *UG IV* 8-1
  detail lines, *UG IV* 8-4
  footing area, *UG IV* 8-16
  layout worksheet, *UG IV* 8-2
  line counter usage, *UG IV* 8-12
  logical page, *UG IV* 8-10, 8-15
  modes of printing, *UG IV* 8-6
  online printing, *UG IV* 8-7
  page advancing, *UG IV* 8-10
  page body, *UG IV* 8-16
  page footing, *UG IV* 8-4
  page heading, *UG IV* 8-4
  page-overflow condition, *UG IV* 8-11
  printing the, *UG IV* 8-24
  printing totals before detail lines, *UG IV* 8-30
  problem solving, *UG IV* 8-25
  report footing, *UG IV* 8-4
  report heading, *UG IV* 8-4
  rolled forward totals, *UG IV* 8-8
  spooling, *UG IV* 8-7
  streamlining your, *UG IV* 8-30
  subtotals, *UG IV* 8-8
  top margin, *UG IV* 8-16
  total accumulating, *UG IV* 8-8
  underlining in a, *UG IV* 8-31
Representation of numeric data, *UG III* 4-1
Required words, *LRM* 1-6
Requirements for binary search, *UG III* 3-15
Requirements for sequential search, *UG III* 3-14
RERUN clause
  general rules for, *LRM* 3-24
/RES BLDODL switch, *RSTS/E UG I* D-9, *RSX UG I* D-9
RESERVE clause, *LRM* 3-20
Reserved words, *LRM* 1-4 to 1-9
  list of, *LRM* A-1
Resident libraries
  See Libraries
Resultant identifiers
  purpose of, *LRM* 5-22
RETURN statement, *LRM* 5-112 to 5-113
  using, *UG IV* 10-2
Rewind operations for file media, *LRM* 5-51 to 5-54
REWRITE statement, *LRM* 5-114 to 5-116
RMS-11, *UG IV* 1-1
  bucket filling for indexed files, *LRM* 3-23
  completion codes, *UG IV* 5-5
  file extension, *LRM* 3-23
  file mapping using the WINDOW phrase, *LRM* 3-23
  preallocation of disk blocks, *LRM* 3-23
RMS-11 libraries, *RSTS/E UG I* 4-2, *RSX UG I* 4-2
RMS-STS, *LRM* 1-7, *UG IV* 5-5
  as VAX-11 COBOL incompatibility, *LRM* D-7
RMS-STV, *LRM* 1-7, *UG IV* 5-5
  as VAX-11 COBOL incompatibility, *LRM* D-7
/RMS:NORESIDENT qualifier, *RSTS/E UG I* 4-2, *RSX UG I* 4-2
/RMS:RESIDENT qualifier, *RSTS/E UG I* 4-2, *RSX UG I* 4-2
Rounding off arithmetic results, *LRM* 5-22, *UG III* 1-16
Rules for numeric editing, *UG III* 1-13
Run time, see performance
RUN, DCL command, *RSTS/E UG I* 1-1, *RSX UG I* 1-1
Run-time environment, documenting, *LRM* 3-3 to 3-4
Run-time error messages
    list of, *RSTS/E UG I* C-1, *RSX UG I* C-1

S

SAME AREA clause, *LRM* 3-22 to 3-23
SAME RECORD AREA clause, *LRM* 3-23
SAME SORT AREA clause, *LRM* 3-24
Sample overflow statements, *UG III* 2-15t
Scaling
  defined, *UG III* 4-2
Scaling and Mixing data types, *UG III* 4-2

Scaling position, decimal, *UG III* 1-6
Scope of statements, *LRM* 5-4
Screen positioning
  absolute, *UG IV* 9-5
  relative, *UG IV* 9-5
SD
  See Sort-merge file description
SEARCH ALL statement
  advantages of using, *UG III* 4-5
  requirements for using, *UG III* 4-5
Search argument
  use in REPLACING phrase, *UG III* 2-44
SEARCH statement, *LRM* 5-117 to 5-123
Searching tables, *UG III* 3-14
Section headers
  elements in, *LRM* 1-23
Section, in Procedure Division, *LRM* 5-32
Section-name
  defined, *LRM* 1-5
  using in segmentation, *UG II* 4-7
Segment numbers
  using in segmentation, *UG II* 4-7
SEGMENT-LIMIT clause
  rules for, *LRM* 3-4
  using in segmentation, *UG II* 4-7
Segment-number, defined, *LRM* 1-5
Segmentation, *LRM* 5-29 to 5-30
  in a multiple program task, *UG II* 4-9
  in a single program task, *UG II* 4-8
  nonoverlayable, *UG II* 4-7
  overlayable, *UG II* 4-7
  programming considerations, *UG II* 4-7
  using PSECT names, *UG II* 4-9
  with SEGMENT-LIMIT clause, *LRM* 3-4
Segmented program example, *UG II* 4-12f
Segmented task image
  creating, *UG II* 4-8, 4-11
Semicolon, as a separator, *LRM* 1-11
Sentences, COBOL, *LRM* 5-1, 5-32
  compiler-directing, *LRM* 5-3
  conditional, *LRM* 5-4
  imperative, *LRM* 5-4
Separators, *LRM* 1-11 to 1-12
  defined, *LRM* 1-1
Sequence numbers
  in ANSI format, *LRM* 1-19
  in terminal format, *LRM* 1-16
Sequential file
  access modes, *UG IV* 2-3
  at end condition, handling, *UG IV* 5-2
  buffer size, *UG IV* 7-8
  buffer size calculation, *UG IV* 7-9
  CONTIGUOUS PREALLOCATION, *UG IV* 7-3
Sequential file (Cont.)
  creating a, *UG IV* 2-4, 2-5
  default number of I-O buffers for, *LRM* 3-20
  defining a, *UG IV* 2-3
  design, *UG IV* 2-2
  end-of-file mark, *UG IV* 2-1
  end-of-volume label, *UG IV* 2-2
  extending a, *UG IV* 2-4, 2-8
  EXTENSION, *UG IV* 7-2
  file status values, using, *UG IV* 5-3
  I/O statements, *UG IV* 2-3
  multiple volumes, *UG IV* 2-2
  open modes, *UG IV* 2-3
  optimization techniques, *UG IV* 7-1
  organization, *UG IV* 2-1
  organization of
    advantages, *UG IV* 1-2
    disadvantages, *UG IV* 1-2
  PREALLOCATION, *UG IV* 7-2
  print file, as a, *UG IV* 2-4
  reading a, *UG IV* 2-6
  reserving buffer areas, *UG IV* 7-6
  rewriting records in a, *UG IV* 2-7
  storage file, as a, *UG IV* 2-4
  unit of transfer, *UG IV* 7-8
  WINDOW, *UG IV* 7-3
Sequential search, *LRM* 5-118
  requirements for, *UG III* 3-14
  results of using, *UG III* 3-14
  with AT END statement, *UG III* 3-14
Serial search
  See Sequential search
SET BREAKPOINT, Debugger command, *UG II* 3-2t, 3-7
SET statement, *LRM* 5-124 to 5-125
  indexing function, *UG III* 3-13
Setting program switches, *LRM* 5-18
Setting the scanner
  inspecting data, *UG III* 2-35
Sharing execution control
  in multiple subprograms, *UG II* 6-5
Short lines
  in ANSI format, *LRM* 1-21
  in terminal format, *LRM* 1-18
SHOW BREAKPOINTS, Debugger command, *UG II* 3-2t, 3-8
/SHOW compiler qualifier, *RSTS/E UG I* 3-3t, 3-5, *RSX UG I* 3-3t, 3-5
SHOW SYNONYMS, Debugger command, *UG II* 3-2t, 3-10
/SHOW:MAP compiler qualifier, *RSTS/E UG I* 3-3t, 3-5, *RSX UG I* 3-3t, 3-5

/SHOW:NOMAP compiler qualifier, *RSTS/E UG I* 3-5t, *RSX UG I* 3-5t
Sign
  conventions, *UG III* 1-6
  default for unsigned operands, *LRM* 5-15
  in arithmetic expressions, *LRM* 5-12
  sharing same byte with digit, *UG III* 1-8t
  specifying position of, *LRM* 4-62 to 4-63
  specifying representation of, *LRM* 4-62 to 4-63
  storage
    COMP-3 data items, *UG III* 1-7
SIGN clause, *LRM* 4-62 to 4-63
Sign condition, *LRM* 5-18
Sign control symbols, *LRM* 4-48
  in fixed insertion editing, *LRM* 4-48
  in floating insertion editing, *LRM* 4-49
Sign tests
  description of, *UG III* 1-10
  equivalent relation tests, *UG III* 1-10t
Significant digits, *UG III* 4-2
Signs valid for COMP-3, *UG III* 1-7
Simple insertion editing, *LRM* 4-48
Size
  fixed-length tables, *UG III* 3-2
  variable-length tables, *UG III* 3-5
SIZE clause, *UG IV* 9-12
Size error condition
  and evaluation of exponentiation, *LRM* 5-13
  description of, *LRM* 5-23
/-SKL compiler switch, *RSTS/E UG I* D-3t, D-6, *RSX UG I* D-3t, D-6
SKL, skeleton descriptor file, *RSTS/E UG I* 3-1, *RSX UG I* 3-1
Slash indicator character (/), *RSTS/E UG I* 2-2, *RSX UG I* 2-2
  in ANSI format, *LRM* 1-19
  in terminal format, *LRM* 1-16
Sort
  declarative procedure, *UG IV* 10-6
  features
    file organization, *UG IV* 10-5
    multiple sorts, *UG IV* 10-5
  hierarchy, *UG IV* 10-1
  intermediate key, *UG IV* 10-1
  major key, *UG IV* 10-1
  minor key, *UG IV* 10-1
  programming considerations
    preventing I/O aborts, *UG IV* 10-6
    USE statement, *UG IV* 10-7
Sort
  programming considerations (Cont.)
    variable-length records, *UG IV* 10-6
  sample program, *UG IV* 10-8
SORT statement, *LRM* 5-126 to 5-130
Sort-merge file description, *LRM* 4-21
  clauses of, *LRM* 4-16
  structure of, *LRM* 4-16
Sorting records, *LRM* 4-21, 5-126 to 5-130
  using the RELEASE statement, *LRM* 5-111
  using the RETURN statement, *LRM* 5-112 to 5-113
Source program reference formats, *LRM* 1-15 to 1-22
SOURCE-COMPUTER paragraph, *LRM* 3-2
Space characters, *LRM* 1-11
  as delimiters of
    arithmetic operators, *LRM* 5-12
    relational operators, *LRM* 5-15
SPACE figurative constant, *LRM* 1-8
Space indicator character
  in ANSI format, *LRM* 1-19
Spaces, as zero replacements, *LRM* 4-25
Special characters
  nonnumeric data, *UG III* 2-2
Special insertion editing, *LRM* 4-48
Special registers, *LRM* 1-7
  as VAX-11 COBOL incompatibility, *LRM* D-7
  LINAGE-COUNTER, *LRM* 1-7
  RMS-STS, *LRM* 1-7
  RMS-STV, *LRM* 1-7
Special-character words
  defined, *LRM* 1-6
  use in general formats, *LRM* 1-12
SPECIAL-NAMES paragraph, *LRM* 3-5 to 3-9
Special-purpose words, *LRM* 1-7
Spooler, system, *UG IV* 8-24
/STA:VAX compiler switch, *RSTS/E UG I* D-3t, D-5, *RSX UG I* D-3t, D-5
Standard Alignment Rules, *LRM* 4-6
START statement, *LRM* 5-131 to 5-133
Statements, COBOL, *LRM* 5-1, 5-32
  compiler-directing, *LRM* 5-3
  conditional, *LRM* 5-4
  delimiting, *LRM* 5-4
  imperative, *LRM* 5-3
  options of, *LRM* 5-22 to 5-24
Status Key 1, *LRM* 5-25
Status Key 2, *LRM* 5-26

STB file type, *UG II* 3-2
STOP statement, *LRM* 5-134
STOP, DCL command, *RSTS/E UG I* 1-2
STOP, Debugger command, *UG II* 3-2t, 3-11
Storage allocation, *LRM* 4-7 to 4-14
  differences for COMP and COMP SYNC data items, *UG III* 1-3f
  effect of fill bytes on, *UG III* 1-3, 3-6
  for COMP and COMP SYNC items, *LRM* 4-7, 4-10 to 4-14, *UG III* 1-3
  for COMP-3 data items, *UG III* 1-5f
  for elementary items, *LRM* 4-8, 4-9
  for group items, *LRM* 4-7, 4-8, 4-9, 4-10 to 4-14
  for INDEX data items, *LRM* 4-7
  for records, *LRM* 4-7
  for redefined items, *LRM* 4-7
  left-to-right technique, *LRM* 4-7
  major-minor technique, *LRM* 4-8 to 4-14
  of table data, *UG III* 3-5
  of tables containing COMP or COMP SYNC items, *UG III* 3-6
  of tables not containing COMP, COMP SYNC, or USAGE INDEX items, *UG III* 3-5
  when multiple entries describe the same area, *LRM* 4-56 to 4-59
  word boundaries, *UG III* 3-6
Storage file, *UG IV* 2-4
Storage format of a data item, *LRM* 4-66 to 4-70
Storing numeric data, *UG III* 1-1
STRING statement, *LRM* 5-135 to 5-139
Stringing data
  with DELIMITED BY phrase, *UG III* 2-12
  with multiple sending items, *UG III* 2-11
  with OVERFLOW statement, *UG III* 2-14
  with POINTER phrase, *UG III* 2-12
  with subscripted items, *UG III* 2-15
/SUB compiler switch, *RSTS/E UG I* D-3t, D-5, *RSX UG I* D-3t, D-5
/SUBPROGRAM compiler qualifier, *RSTS/E UG I* 3-3t, 3-5, *RSX UG I* 3-3t, 3-5
  using to identify a subprogram, *UG II* 6-2
Subprograms
  defined, *UG II* 6-1
  identifying, *UG II* 6-2
  unique PSECT names, *UG II* 6-2
  using to reduce task size, *UG II* 4-6
Subscript sequence evaluation, *UG III* 2-30
Subscripted items
  inspecting data, *UG III* 2-37
  to string data, *UG III* 2-15
  to unstring data, *UG III* 2-29
Subscripted moves
  nonnumeric data, *UG III* 2-10
Subscripting, *LRM* 5-8 to 5-9
  basis for, *LRM* 4-38
  in an identifier, *LRM* 5-11
  with index-name items, *UG III* 3-12f
Subscripts
  defined, *UG III* 3-10
SUBTRACT statement, *LRM* 5-140 to 5-142
SWITCH clause, *LRM* 3-6
Switch-status condition, *LRM* 5-18
Switches
  See also BLDODL utility switches
  See also Compiler switches
  setting values for, *LRM* 5-18
  specifying in SPECIAL-NAMES paragraph, *LRM* 3-6
Symbolic Debugger
  See Debugger
Symbols
  numeric editing, *UG III* 1-13
SYNCHRONIZED clause, *LRM* 4-64 to 4-65
Syntax rules, defined, *LRM* 1-26
System spooler, *UG IV* 8-24
System-names, *LRM* 1-4

T

Tab characters
  in ANSI format, *LRM* 1-21
  in terminal format, *LRM* 1-18
  purpose of, *LRM* 1-12
Tab stops
  in ANSI format, *LRM* 1-21
  in terminal format, *LRM* 1-18
Table access
  with SEARCH statement, *UG III* 3-14
Table elements
  initializing, *UG III* 3-8
Table handling
  binary search for a table element, *LRM* 5-119
  searching for a table element, *LRM* 5-117 to 5-123
  sequential search for a table element, *LRM* 5-118

Tables
  accessing
    with indexes, *UG III* 3-10, 3-12
    with subscripts, *UG III* 3-10, 3-11, 3-11f
  defining with OCCURS clause, *LRM* 4-38 to 4-42, *UG III* 3-1
  fixed-length
    multidimensional, *UG III* 3-3
    one-dimensional, *UG III* 3-2
    specifying size of, *UG III* 3-2
  indexing
    rules for, *LRM* 5-10 to 5-11
  initializing, *UG III* 3-8f
    effect of fill bytes on, *UG III* 3-10
    redefining group level, *UG III* 3-8
    with VALUE clause, *UG III* 3-8
  multidimensional, *UG III* 3-1
  storage allocation for, *UG III* 3-5
  subscripting
    rules for, *LRM* 5-8 to 5-9 *UG III* 3-11f
    with data-names, *UG III* 3-12, 3-12f
  variable-length, *UG III* 3-5
Tally argument
  to inspect data, *UG III* 2-38
Tally counter
  to inspect data, *UG III* 2-38
Task Builder
  diagnostics, *RSTS/E UG I* 4-3, *RSX UG I* 4-3
Task image
  defined, *UG II* 4-1
  size of, *UG II* 4-1
Task reduction techniques, *UG II* 4-1
Task-building
  command line format, *RSTS/E UG I* D-11, *RSX UG I* D-11
  with CMD files, *RSTS/E UG I* D-10, *RSX UG I* D-10
/TEMPORARY compiler qualifier, *RSTS/E UG I* 3-3t, 3-6, *RSX UG I* 3-3t, 3-6
  for improving program performance, *UG II* 5-2
Terminal format, *LRM* 1-16 to 1-19, *RSTS/E UG I* 2-2, *RSX UG I* 2-2
  for improving program performance, *UG II* 5-3
  for reducing compile time, *UG II* 5-3
  limitations, *RSTS/E UG I* 2-2, *RSX UG I* 2-2
  versus ANSI format, *UG II* 5-3
Testing
  for the sign of a value, *LRM* 5-18

Testing (Cont.)
  nonnumeric data items, *UG III* 2-3
  numeric items, *UG III* 1-9
Text-name, defined, *LRM* 1-5
/TMP compiler switch, *RSTS/E UG I* D-3t, D-7, *RSX UG I* D-3t, D-7
Top margin, *UG IV* 8-16
Top-of-page character (/), *RSTS/E UG I* 2-2, *RSX UG I* 2-2
Transferring execution control
    with CALL statement, *LRM* 5-48 to 5-50, *UG II* 6-3
    with EXIT PROGRAM statement, *LRM* 5-70
    with GO TO statement, *LRM* 5-71 to 5-72
    with IF statement, *LRM* 5-73 to 5-75
    with MERGE statement, *LRM* 5-82 to 5-86
    with PERFORM statement, *LRM* 5-98 to 5-106
    with READ statement, *LRM* 5-107 to 5-110
/TRU compiler switch, *RSTS/E UG I* D-3t, D-5, *RSX UG I* D-3t, D-5
/TRUNCATE compiler qualifier, *RSTS/E UG I* 3-3t, 3-6, *RSX UG I* 3-3t, 3-6
Truth value
  defined, *LRM* 5-1
  of conditional expressions, *LRM* 5-14
TSK file
  compared to task image, *UG II* 4-1
TYPE, DCL command, *RSTS/E UG I* 1-6t, *RSX UG I* 1-6t

U

/ULIB BLDODL switch, *RSTS/E UG I* D-8, *RSX UG I* D-8
UNDEFINE, Debugger command, *UG II* 3-2t, 3-10
Undefined results in a data-handling operation, *LRM* 5-24
Uniqueness of Reference, *LRM* 1-4
UNSTRING statement, *LRM* 5-143 to 5-148
  using, *UG III* 2-18
Unstringing data, *UG III* 2-18
  common errors, *UG III* 2-31
  COUNT phrase, *UG III* 2-24
  delimiting with all asterisks, *UG III* 2-22t

Unstringing data (Cont.)
delimiting with all double asterisks, *UG III* 2-23t
delimiting with asterisk, *UG III* 2-20t
delimiting with two asterisks, *UG III* 2-22t
multiple delimiters, *UG III* 2-23, 2-24t
multiple receiving items, *UG III* 2-18
OVERFLOW statement, *UG III* 2-28
POINTER phrase, *UG III* 2-26
receiving items based on sending item, *UG III* 2-19t
sending item too short, *UG III* 2-19t
TALLYING phrase, *UG III* 2-27
using subscripted items, *UG III* 2-29
with DELIMITED BY phrase, *UG III* 2-20
with DELIMITER phrase, *UG III* 2-25
Uppercase words, as used in general formats, *LRM* 1-12
USAGE clause, *LRM* 4-66 to 4-70
USE statement, *LRM* 5-149 to 5-150
and invalid key condition, *LRM* 5-28
User-defined words, *LRM* 1-3 to 1-4
uniqueness of, *LRM* 5-6
USING phrase
in SORT statement, *UG IV* 10-2
of CALL statement, *LRM* 5-32, 5-48 to 5-49
of Procedure Division header, *LRM* 4-17, 5-32, 5-48 to 5-49

V

VALUE IS clause, *LRM* 4-71 to 4-73
use in Linkage Section, *LRM* 4-17
use in Working-Storage Section, *LRM* 4-16
VALUE OF ID clause, *LRM* 4-74
using the, *UG IV* 1-10
Variable-length records, *LRM* 4-53 to 4-55
creation of, *UG IV* 1-4 to 1-8
record-length field, *UG IV* 1-4
VAX-11 COBOL
ensuring COBOL-81 compatibility with, *LRM* D-1 to D-9
Verbs, COBOL, *LRM* 5-1

W

Warning diagnostics, *RSTS/E UG I* 3-7, *RSX UG I* 3-7
/WARNINGS compiler qualifier, *RSTS/E UG I* 3-6, *RSX UG I* 3-6
/WARNINGS:INFORMATIONAL compiler qualifier, *RSTS/E UG I* 3-3t, 3-6, *RSX UG I* 3-3t, 3-6
/WARNINGS:NOINFORMATIONAL compiler qualifier, *RSTS/E UG I* 3-3t, 3-6, *RSX UG I* 3-3t, 3-6
WINDOW phrase of the APPLY clause, *LRM* 3-23
WITH DUPLICATES IN ORDER phrase, *UG IV* 10-4
Word boundaries
effects on storage allocation, *UG III* 3-6
with COMP and COMP SYNC, *UG III* 1-2
Working-Storage Section of Data Division, *LRM* 4-16
WRITE statement, *LRM* 5-151 to 5-156
effect on LINAGE values, *LRM* 4-35

Z

ZERO figurative constant, *LRM* 1-8
Zero suppression editing, *LRM* 4-50

HOW TO ORDER ADDITIONAL DOCUMENTATION

DIRECT TELEPHONE ORDERS

In Continental USA
and Puerto Rico
call **800–258–1710**

In Canada
call **800–267–6146**

In New Hampshire,
Alaska or Hawaii
call **603–884–6660**

DIRECT MAIL ORDERS (U.S. and Puerto Rico*)

DIGITAL EQUIPMENT CORPORATION
P.O. Box CS2008
Nashua, New Hampshire 03061

DIRECT MAIL ORDERS (Canada)

DIGITAL EQUIPMENT OF CANADA LTD.
940 Belfast Road
Ottawa, Ontario, Canada K1G 4C2
Attn: A&SG Business Manager

INTERNATIONAL

DIGITAL EQUIPMENT CORPORATION
A&SG Business Manager
c/o Digital's local subsidiary
or approved distributor

Internal orders should be placed through the Software Distribution Center (SDC), Digital Equipment Corporation, Northboro, Massachusetts 01532

*Any prepaid order from Puerto Rico must be placed
with the Local Digital Subsidiary:
809–754–7575

COBOL-81
RSX-11M/M-PLUS
User's Guide
AA-M179B-TC

Reader's Comments

Note: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. If you require a written reply and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Did you find this manual understandable, usable, and well-organized? Please make suggestions for improvement.____

Did you find errors in this manual? If so, specify the error and the page number.________________

Please indicate the type of user/reader that you most nearly represent.

- □ Assembly language programmer
- □ Higher-level language programmer
- □ Occasional programmer (experienced)
- □ User with little programming experience
- □ Student programmer
- □ Other (please specify)____________________________

Name____________________________ Date__________________________

Organization____

Street____

City____________________________ State___________ Zip Code or Country __________

------Do Not Tear - Fold Here and Tape --

digital

No Postage
Necessary
if Mailed in the
United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

ATTN: BSSG Publications ZKO1–3/J35
DIGITAL EQUIPMENT CORPORATION
110 SPITBROOK ROAD
NASHUA, N.H. 03062

------Do Not Tear - Fold Here and Tape --

Cut Along Dotted Line

digital

DIGITAL EQUIPMENT CORPORATION

Printed in U.S.A.