pdp 11

PDP-11

COBOL User's Guide

Order No. AA-1757C-TC

digital

April 1977

This document describes how to use Version 3 of the
PDP-11 COBOL compiler. It is a companion guide to
the *PDP-11 COBOL Language Reference Manual.*

PDP-11

COBOL User's Guide

Order No. AA-1757C-TC

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.

Digital Equipment Corporation assumes no responsibility for the use
or reliability of its software on equipment that is not supplied by
DIGITAL.

The postage prepaid READER'S COMMENTS form on the last page of this
document requests the user's critical evaluation to assist us in pre-
paring future documentation.

The following are trademarks of Digital Equipment Corporation:

DIGITAL	DECsystem-10	MASSBUS
DEC	DECtape	OMNIBUS
PDP	DIBOL	OS/8
DECUS	EDUSYSTEM	PHA
UNIBUS	FLIP CHIP	RSTS
COMPUTER LABS	FOCAL	RSX
COMTEX	INDAC	TYPESET-8
DDT	LAB-8	TYPESET-10
DECCOMM	DECsystem-20	TYPESET-11

CONTENTS

CONTENTS (Cont.)

iv

CONTENTS (Cont.)

CONTENTS (Cont.)

CONTENTS (Cont.)

CONTENTS (Cont.)

CONTENTS (Cont.)

ix

CONTENTS (Cont.)

FIGURES

FIGURES (Cont.)

TABLES

FOREWORD

The PDP-11 COBOL User's Guide is intended primarily for reference use. It is a companion guide to the PDP-11 COBOL Language Reference Manual. Because it is not a tutorial guide for beginning programmers, you should have a working knowledge of the COBOL language.

This guide describes the COBOL file structures, data formats, some of the features of the PDP-11 COBOL Version 3 compiler, error messages generated by the compiler and run-time systems, I/O devices available with the system, some hints on good programming practices, some techniques for debugging source programs, and a description of the PDP-11 COBOL utility programs COBRG and REFORMAT.

Those wishing to learn the COBOL language are referred to the following tutorial manuals:

Farina, Mario V., COBOL Simplified,
New Jersey, Prentice Hall, Inc., 1968.

McCameron, Fritz A., COBOL Logic and Programming,
Homewood, Illinois, Richard D. Irwin, Inc., 1970

McCracken, Daniel D. and Garbassi, Umberto,
A Guide to COBOL Programming, Second Edition,
New York, John Wiley and Sons, Inc., 1970

NOTE

These publication dates are the latest available. They will all probably be revised to reflect ANS-74 standards.

The PDP-11 COBOL compiler accepts COBOL language elements that are a true subset of ANS-74 COBOL. The PDP-11 COBOL Reference Manual and the PDP-11 COBOL Reference Card both use shading to indicate extensions to the standard. This guide gives all extensions with explanations of the extension without shading. Appendix A of this guide (COBOL Formats), however, does use shading to indicate extensions.

ACKNOWLEDGMENT

CHAPTER 1

INTRODUCTION

Before a program can be run on a computer, it must be translated from
the form that a person can understand (near-English) into a form that
a computer can understand (machine language). This translation is
performed by COBOL compiler systems. Because each manufacturer's
system has its own unique machine language, each system has its own
COBOL compiler. The COBOL compiler for the PDP-11 system is the
PDP-11 COBOL compiler.

The PDP-11 COBOL compiler translates ANS-74 COBOL source programs into
relocatable object modules. The compiler runs under the supervision
of the following PDP-11 operating systems and conforms to all the
conventions and restrictions of the system in control.

- RSTS/E

- RSX-11M

- IAS

The compiler itself requires disk storage space for its work file
system and temporary files. (The work file system requires a minimum
of 128 blocks to a maximum of 512 blocks. The temporary file and
object program file requirements grow in proportion to the size of the
source program.)

To run a COBOL program, you follow a five-step process:

- Prepare a source program

- Compile a source program

- Merge or prepare an overlay description file (optional)

- Task-build object modules into an executable task

- Execute the task

The PDP-11 COBOL compiler accepts COBOL source statements from source
input files. This means that you must manually enter your source
statements onto an acceptable medium prior to the compilation process
(Section 2.1, Choosing a Reference Format; and Section 2.2 Choosing
an Input Medium).

Once you have decided upon an input medium and format for your source
input files and have created them, you compile the source program.

The PDP-11 COBOL compiler reads source statements from the source
input file, translates them into object code modules consisting of
program sections (PSECTs), and produces the following files:

1-1

- Listing (LST)

- Object (OBJ)

- Overlay Description Language (ODL)

The listing file (LST) contains a listing of the source statements in the order in which they were compiled, any diagnostic error messages, and any optional special format listings, e.g., cross-reference listings and data and procedure maps. You obtain special format listings by appending an appropriate switch to the COBOL command line at compile-time, (see Section 2.5.3, Compiler Switches).

The object file (OBJ) contains a collection of program sections called PSECTs which are not executable. They must be linked into an executable task image by an operating system task called Task Builder. The ability to compile COBOL subprograms to produce linkable object files independently, enables you to create modular programs.

The Overlay Description Language (ODL) file contains directives that describe the overlay structure of the object module generated from the COBOL source program. ODL directives are generated into the ODL file for each overlayable object module program section.

The ODL file generated by the compiler is not a complete ODL file. You must either hand-tailor command information into it, build your own specialized version, or specify the compiler-generated ODL file as input to the Merge Utility to complete the file (see Section 2.6, Using the Merge Utility; Chapter 11, Hand-Tailoring ODL Files).

The compiler can compile only one source program or subprogram per command string execution. Therefore, a program which consists of a main program plus one or more subprograms requires multiple executions of the compiler. Each compilation generates a separate listing, ODL, and object file. The ODL files, in this instance, must be merged together into a single ODL file to be submitted as input to the Task Builder.

To accomplish this, you use the Merge Utility, which performs the following functions:

1. merges the ODL statements from one or more ODL files into a single ODL file

2. analyzes the I/O requirements for the entire task and provides directives to include only those I/O routines required

3. inserts the missing but required ODL directives not provided by the compiler

Whether you have a large segmented program, a main program plus subroutines, or a small stand-alone program, the ODL files generated by the compiler require merging or modification. You are advised to use the Merge Utility to perform this merging/modification, although you can hand-tailor your own ODL file.

NOTE

Do not attempt to hand-tailor your own
ODL file unless you have in-depth
knowledge of your operating system and
PDP-11 COBOL. The following references
will provide you with the information
required to hand-tailor ODL files:

● Your system Task Builder Manual

● Standard ODL file Format
    Chapter 12 of this manual

● COBOL Segmentation
    Chapter 10 of this manual

● Code PSECT Naming Conventions
    Appendix D of this manual

● Interprogram Communications
    Chapter 11 of this manual

When you have a single ODL file that contains all of the required
overlay descriptor language, you can execute the Task Builder.

Task Builder provides you with a facility for linking separately
compiled object modules into an executable task image. You can,
depending on your knowledge of your operating system and PDP-11 COBOL,
link object modules created by another programming language into your
COBOL task image. Task Builder also allows you to take full advantage
of the COBOL system library to selectively link into your COBOL task
only those runtime support routines actually needed to run your task.

The Task Builder, using the ODL file as a guide, provides the facility
to build large amounts of code into a task by careful use of code
segments that overlay each other. Careful use of segmentation or
calls to subprograms within your COBOL source program will allow you
to compile and execute large and complex COBOL programs. If you add
some functionality to an existing COBOL program and find that, after
task-building, the resulting task will not fit in memory, you have an
alternative other than reprogramming: you can segment the program or
make subprograms out of some of the existing procedures, replacing
these procedures with CALL statements to the newly created
subprograms. When the source program has been compiled, the ODL file
merged, and task-building accomplished, the task is ready to be
executed (Section 2.8, Executing a COBOL Task).

Figure 1-1 shows the process of preparing a COBOL program for
execution:
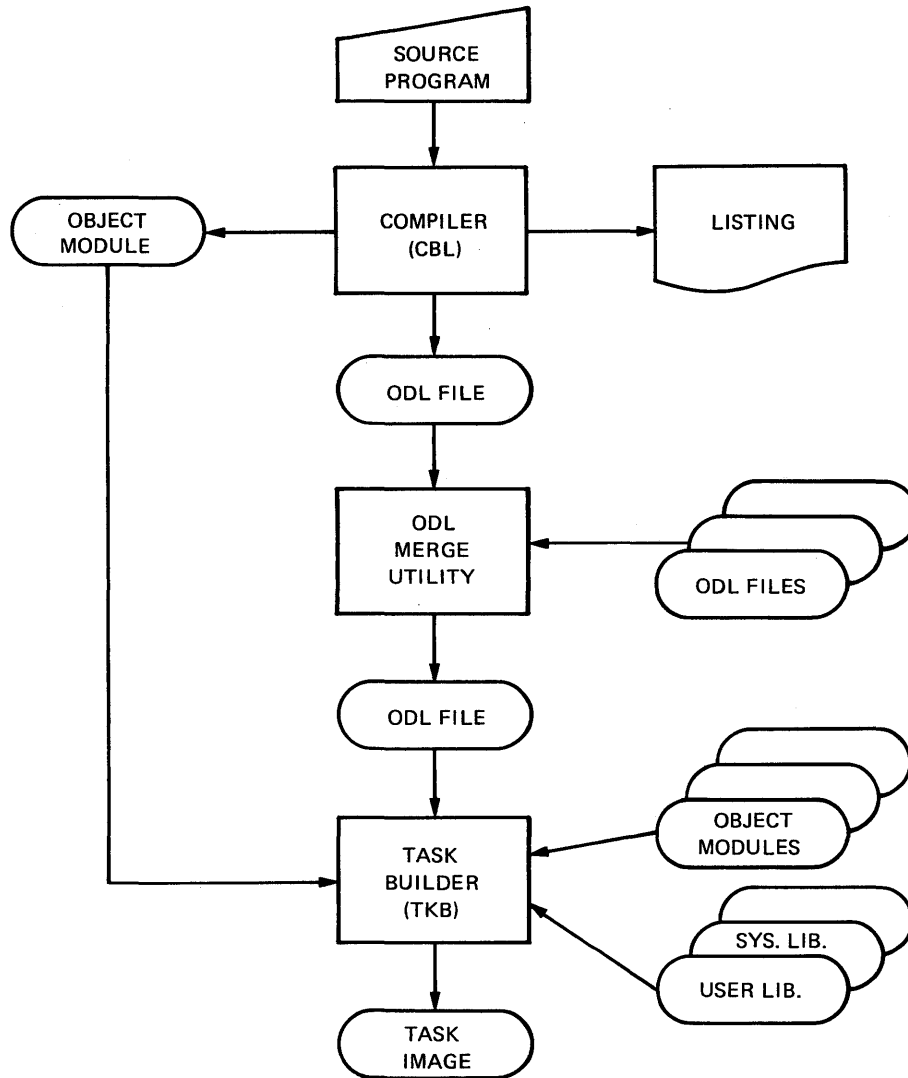
Figure 1-1   Building A COBOL Task Image

1.1 THE COBOL SOURCE PROGRAM

A COBOL source program consists of four major divisions, which must appear in the following order:

1. IDENTIFICATION DIVISION.

2. ENVIRONMENT DIVISION.

3. DATA DIVISION.

4. PROCEDURE DIVISION.

Each of the COBOL divisions is further divided into sections that may be divided into paragraphs. Paragraphs contain COBOL sentences. Sentences contain COBOL statements. The following subsections (1.1.1 through 1.1.4) individually discuss each division and its major sections.

1.1.1 The Identification Division

The Identification Division identifies the COBOL program and contains such optional documentary information as the name of the programmer, the name of the installation, and the date the program was written and last compiled.

1.1.2 The Environment Division

The Environment Division identifies the hardware configuration of the system that is compiling the program (SOURCE-COMPUTER) and the hardware configuration of the system that is running the program (OBJECT-COMPUTER). The division is divided into the following two major sections:

- Configuration Section -- This section is required and contains the names of the source and object computers and any mnemonic names that are to be assigned to devices.

- Input-Output Section -- This section is optional and contains descriptions of all the external files being manipulated by the program. The section is required if there are external files.

1.1.3 The Data Division

The Data Division contains complete descriptions of all data to be processed by the program. In this division, the programmer must assign a data-name to and describe every data item referred to in the Procedure Division.

The Data Division is composed of three optional major sections:

- File Section -- Contains the descriptions of all input and output files and their records.

- Working-Storage Section -- Contains the descriptions of temporary records and data items.

- Linkage Section -- Describes the data that is available to a called program and is referenced in both the calling and called program.

1.1.4 The Procedure Division

The Procedure Division contains the program's procedural statements. Within this division, the program specifies manipulation of the data items described in the Data Division.

The Procedure Division may begin with the DECLARATIVES, which contain USE procedure declarative sections for processing I/O errors (Section 12.3).

The programmer may divide the Procedure Division into sections and paragraphs that each perform a function.

The Procedure Division makes COBOL's advantages (excellent documentation and programming simplicity) most apparent. Figure 1-2 (sample Procedure Division coding) illustrates the documentation capabilities of COBOL programs:

```
    PROCEDURE DIVISION.

    BEGIN.

        OPEN INPUT TRANSACTION-FILE.

        OPEN OUTPUT MASTER-FILE.

    READ-A-RECORD.

        READ TRANSACTION-FILE NEXT RECORD

            AT END GO TO CLOSE-ROUTINE.

        READ MASTER-FILE NEXT RECORD

            AT END GO TO CLOSE-ROUTINE.

    PROCESS-A-RECORD.

        IF TRANS-ACCT-NUMBER IS GREATER THAN MAST-ACCT-NUMBER

            GO TO READ-MAS-RECORD.

        IF TRANS-ACCT-NUMBER IS LESS THAN MAST-ACCT-NUMBER

            GO TO ERROR-ROUTINE.

        .
        .
        .
```

Figure 1-2 Sample COBOL Procedural Coding

1.2 THE COBOL UTILITY PROGRAMS

PDP-11 COBOL offers three utility programs:

- COBRG -- Produces report-generating COBOL programs.

- REFORMAT -- Converts PDP-11 terminal format COBOL programs into conventional format ANSI COBOL programs.

- MERGE -- Merges ODL file(s) created by one or more COBOL compilations into a single ODL file to be submitted as input to the Task Builder.

The following subsections (1.2.1, 1.2.2, and 1.2.3) describe these utility programs. Chapter 8 in this manual discusses the use of COBRG and REFORMAT in detail. The SORT utility is discussed in the PDP-11 SORT Reference Manual.

1.2.1 COBRG

COBRG generates a report-writing source program ready for compilation. The user specifies certain parameters, and COBRG builds a COBOL program that produces a tailored report when it is compiled, task-built, and run.

1.2.2 REFORMAT

PDP-11 COBOL accepts source programs that were coded using either the ANSI 80-column card reference format or the shorter, terminal-oriented PDP-11 reference format. The REFORMAT utility program reads source programs that were coded in the PDP-11 terminal format and converts them to 80-column ANSI-compatible source programs (Section 2.1, Choosing A Reference Format, discusses the two reference formats).

1.2.3 MERGE

The Merge Utility program merges ODL files generated by COBOL compilations into a single ODL file. This file is submitted as input to the Task Builder. It describes the overlay structure of the resulting task image.

CHAPTER 2

USING THE PDP-11 COBOL SYSTEM

This chapter discusses the following topics:

- Choosing a reference format

- Choosing an input medium

- Creating a source input file

- Using the library facility (COPY statement)

- Using the COBOL compiler

- Using the MERGE Utility

- Using the Task Builder

- Executing a COBOL task

The order of topic presentation is designed to give you a step-by-step approach to preparing a COBOL program for execution. The chapter provides an in-depth description of each step in the process and lists, where applicable, alternate methods for achieving the same goal.

2.1 CHOOSING A REFERENCE FORMAT

PDP-11 COBOL provides two formats for coding your source programs, conventional and terminal. Both formats are described in terms of character positions in a line on an input medium.

NOTE

The rules for spacing given in this
discussion of reference formats take
precedence over all other rules for
spacing.

The conventional format is based on the traditional COBOL format as applied to an 80-column punched card. The terminal format is a DEC specified format and allows a source line to be shortened through the use of horizontal tabs and carriage returns. The terminal format is very convenient for use with a text editor and an on-line computer terminal.

NOTE

The PDP-11 COBOL compiler assumes the
terminal format as a default when
compiling, but is capable of compiling
either format (Section 2.5.3, the /CVF
Switch).

A reformatting program (REFORMAT) reformats terminal format programs
into conventional format for ease in transporting source programs to
other COBOL compilers. The REFORMAT program is described in Chapter
8.

2.1.1 Conventional Reference Format

The conventional reference format is based on the traditional COBOL
format as applied to an 80-column punched card. If you choose this
format, you will probably code your source program on a standard COBOL
coding form (Figure 2-1 COBOL Programming Form).

COBOL PROGRAMMING FORM

PROGRAMMER _____

PROGRAM NAME _____

PAGE _____ OF _____

IDENT

PAGE	SERIAL	CONT.	START DIVISION, SECTION, PARAGRAPH OR FILE DESCRIPTION HERE / START OPERAND OR CONTINUATION LINE HERE

Figure 2-1   COBOL Programming Form

2.1.1.1 Sequence Numbers - Character positions 1 through 6 of the standard format are reserved for source line sequence numbers. This sequence number field serves only to assist you in locating and editing source lines within a source file or listing. Sequence numbers are ignored by the compiler.

2.1.1.2 Continuation/Comment Indicator Area - Character position 7 can be used to direct the compiler to process the source line as specified by the character in this column. Your options are:

Option	Results
blank ( )	Default - The source line is treated normally.
hyphen (-)	Continuation line - The compiler processes this line as a continuation of the previous source line (see Section 2.1.1.6, Continuation of Lines).
asterisk (*)	Comment line - The compiler transfers the contents of this line, as is, to the source listing. No syntax checking is performed on this line (see Section 2.1.1.8, Comment Lines).
slash (/)	Comment line - The compiler treats the line as though it were a comment line except it advances the source listing to the top of the next page before printing the contents of the line.

2.1.1.3 Area A - Character positions 8 through 11 constitute Area A of the conventional format. This area is reserved for the beginning of   division   headings,   section-names,   paragraph-names, level-indicators, and certain level-numbers.

2.1.1.4 Area B - Character positions 12 through 72 constitute Area B of the conventional format. This area is reserved for all other COBOL text.

2.1.1.5 Identification Field - Character position 73 through 80 constitute the identification field. This field is for documentation purposes only and has no effect on compilation.

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

When you break a word or numeric literal from one line to the next, you   must   place   a   hyphen   (-)   in character position 7 of the continuation line. The first nonblank character that you enter in Area B will become the next character of the word or numeric literal being continued.

When you break an alphanumeric literal from one line to the next, you must place a hyphen in character position 7 of the continuation line, and you must also precede the first character of the continuation literal with a quotation mark. The literal may begin anywhere within area B of the continuation line. Consider the following example:

```
000008 WORKING-STORAGE SECTION.
001010 01  CONTINUATION-NUMERIC.
001020     02  NUMERIC-LITERAL          PIC 9(18) VALUE IS 12345678912345
001030-    6789.
001040 01  CONTINUATION-ALPHANUMERIC.
001050     02  ALPHANUMERIC-LITERAL    PIC X(26) VALUE IS "ABCDEFGHIJKLM
001060-    "NOPGRSTUVWXYZ".
001070 PROCEDURE DIVISION.
001080 CONTINUATION-SENTENCE.
001090     IF NUMERIC-LITERAL NOT EQUAL TO ALPHANUMERIC-LITERAL
001100              GO TO END-PROGRAM
001110          ELSE GO TO CONTINUATION-SENTENCE.
001120 END-PROGRAM.
001130          STOP RUN.
```

Source lines 001010 through 001030 show how a numeric literal can be continued onto another line. Source lines 001040 through 001060 show how an alphanumeric literal can be continued onto another line. Source lines 001090 through 001110 show how a sentence can be continued onto successive lines.

2.1.1.7 **Blank Lines** - A blank line is one that is blank from character position 7 through 72. A blank line can not immediately precede a continuation line. Otherwise, a blank line can appear anywhere in the source program.

2.1.1.8 **Comment Lines** - A comment line is any line with an asterisk (*) in character position 7. A comment line can not precede the Identification Division header. Otherwise, a comment line may appear anywhere in a source program.

A comment line may be composed of any of the characters from the full COBOL character set. Comments can begin in Area A or B of the source line. Each comment line is reproduced on the source listing, but serves as documentation only. Successive comment lines are allowed, but each must contain an asterisk in character position 7.

NOTE

If a slash character (/) is used instead of an asterisk (*), the results are the same except the source listing is advanced to the top of the next page before the comment entry is printed.

2.1.1.9 **Short Lines and Tab Characters** - Conventional format source lines may be shortened if a medium other than punched cards is used. This is accomplished by terminating the line by a carriage return or by inserting tab characters within the line to replace space characters, or a combination of both.

The compiler treats a carriage return character as a redefinition of character position 72. When a tab character is encountered, the compiler generates the required number of space characters consistent with the tab character's position on the line. Tab stops are set, within the compiler, at character positions 7, 8, 12, 20, 28, 36, 44, 52, 60, 68, and 73. Consider the following example:

NOTE

(RET) = carriage return character

(TAB) = tab character

Shortened conventional source line:

    000130 01 (TAB) FILE-A. (RET)

    000140 (TAB) 02 DATA-FIELD-A. (RET)

    000150 (TAB) (TAB) 03 DESCRIPTION-A (TAB) PIC X(20). (RET)

    000160 (TAB) (TAB) 03 DESCRIPTION-B (TAB) PIC X(20). (RET)

    000170 (TAB) (TAB) 03 DESCRIPTION-C (TAB) PIC X(20). (RET)

Source line as interpreted by the compiler:

```
00015 000130 01   FILE-A.
00016 000140      02  DATA-FIELD-A.
00017 000150            03 DESCRIPTION-A     PIC X(20).
00018 000160            03 DESCRIPTION-B     PIC X(20).
00019 000170            03 DESCRIPTION-C     PIC X(20).
```

2.1.2 Terminal Reference Format

Terminal reference format is the PDP-11 COBOL default format. It makes your life easier by providing you with a format which is easy to use with a computer terminal. Terminal format is shorter and less space consuming than its conventional counterpart. The sequence number and identification fields are eliminated and the indicator field is the first character position within Area A.

Tab characters can be used to position source entries within a line, and a line ends at the first occurrence of a carriage return character.

The terminal reference format for a source line is represented as follows:

Character Position	Contents
1 through 4	Area A
5 through 65	Area B

NOTE

Continuation line (-), comment line (*), and skip to top of page (/) indicator characters, when used, must be placed in character position 1.

Area A and Area B, for the terminal format, contain the same kinds of source entries as their conventional format counterparts (see Sections 2.1.1.3, Area A and 2.1.1.4, Area B).

As with the conventional format, tab characters, when encountered by the compiler, generate a number of spaces commensurate with the tab character's position on the line. Tab stops are set to character positions 5, 13, 21, 29, 37, 45, 53, 61, and 66.

2.2 CHOOSING AN INPUT MEDIUM

The input medium you select is dependent on the devices you have at your disposal. The PDP-11 COBOL compiler accepts source input from any of the following devices:

- Card Reader

- Magnetic Tape

- DECtape

- Disk

- Terminal

2.3 CREATING A SOURCE FILE

Using text editors under PDP-11 operating systems generally makes program preparation, modification, and storage from a console terminal simpler than using punched cards.

NOTE

RSTS/E users:

See the following manuals,

- RSTS-11 System User's Guide

- PDP-11 EDIT Text Editor

RSX-11M users:

See the RSX-11D Utility Program Procedures Manual.

IAS users:

See the IAS Editing Utilities Reference Manual.

When creating source programs at the terminal, use the appropriate text editor to create source files on disk, DECtape, or magtape. When entering them under other PDP-11 operating systems that do not support COBOL, write them on DECtape or magtape first and use the Filex (FLX) utility program to read them into the operating system (DOS DECtape and ANSI magtape are common media among PDP-11 operating systems).

The following suggestion makes editing a source program from a
terminal even more convenient:

> Use the terminal reference format (Section 2.1.2). The
> terminal format is designed for use at a terminal and
> eliminates the conventional COBOL fields in columns 1-6 and
> 73-80. It allows the line to be further shortened by use of
> the horizontal tab character to vertically align similar
> statements or phrases, and by the use of the carriage return
> to terminate the line. A typical conventional format line
> can often be reduced to less than 40 characters in terminal
> format with no loss in source text readability.

2.4 USING THE LIBRARY FACILITY (COPY)

The PDP-11 COBOL library facility provides you with a means of copying
COBOL source language text from a library of source material into a
COBOL program during compilation. One COPY statement can place large
amounts of library source text into a source program, thereby saving
repetitious coding. The compiler treats the copied material as though
it were part of the program being compiled. The copied material,
however, does not physically alter the source program file in any way.

The COBOL library facility provides two important benefits:

1. Standardization of File and Coding Conventions

    A typical data file is processed by more than one program,
    and each processing program must describe the characteristics
    of that file (file-name, blocking factor, field names, etc.).
    Often the programs are written by one programmer, then
    maintained and updated by another programmer, who has to try
    to understand a program that was written by someone else.
    Since this situation arises at most sites, it is good
    practice to design a standardized file description (keeping
    in mind the programs that process that file) and place it in
    the library. Then a COPY statement in each processing
    program can merge it into the program at compile-time.

    This technique applies as well to any procedure that is used
    in many different applications. For example, the library
    could contain a standardized routine that converts calendar
    dates to Julian dates to be merged into each source program
    that uses this function.

2. Time Savings for Initial Coding and Updating

    Defining and coding file and record descriptions is a
    time-consuming chore. If the descriptions exist in the
    library, a single COPY statement will save the time required
    to code those entries into programs using them.

    Changing a file format is another time-consuming chore.
    Typically, when a file format changes, you must change and
    recompile all the programs that use that file. If the file
    description is in a library file, the programmer has to
    change only the library file. The source programs, of
    course, still have to be recompiled but require no individual
    coding changes.

    Putting commonly used Procedure Division procedures in a
    library file yields the same time-saving benefits.

2.4.1 Creating a COBOL Library File

Each line of a COBOL library file must be in a form such that, when it is merged into the source program, it forms a syntactically correct COBOL clause, phrase, or sentence. It can meet this condition either by being syntactically correct itself, or by becoming so when it is merged with the source program.

The library text must conform to the rules for the COBOL reference format (Library Module in the PDP-11 COBOL Reference Manual). You may write it in either the conventional (with page-line numbers) or the terminal (without page-line numbers) format. However, the library file and the source programs it is merged into must be in the same format. (A conventional format library file cannot be merged properly into a terminal format source program and vice-versa.)

When writing text for library files in the terminal format, place at least four space characters or a tab character before any entry that normally begins in Area B of the COBOL source program. Left justify, without space characters, entries that normally begin in Area A or at character position 0.

Since each library file contains all the source language text to be merged into a source program by one COPY statement, the COPY statement text-name must refer to the library file-name.

To create the library file, either punch it into cards and use PIP to put it on DECtape or disk or enter it directly onto one of these media via a terminal. The operating system provides a text editor for accomplishing this function. There is no method for updating COBOL library files on magtape. The available media for these files are DECtape and disk storage.

2.4.2 The COPY Statement

The COPY statement is a compile-time function that merges a COBOL library file into a COBOL source program.

The statement must begin with the word COPY and end with a period. The compiler logically replaces the COPY statement (including the period) with the library file named by the statement. However, both the COPY statement and the library text material appear in the source listing (Section 2.4.4, The Source Listing). The statement may appear anywhere in a source program that a COBOL word is allowed. The simplest form of the statement is:

        COPY text-name.

Text-name must specify either a file-name or a alphanumeric literal. If a file-name is specified, the compiler assumes standard file specification defaults and copies the text from the latest version of that file into the source program.

The format for the full file specification is:

        device:[UIC]file-name.file-type;version-number

The specification defaults are:

    device: -- the standard system device or the device specified in the batch JCL

    [user-identification]   -- the UIC that you are logged in under

    file-name   -- (no default)

    .file-type   -- .LIB

    ;version-number   -- the latest version. Note: RSTS/E does not support the version number feature.

For example, the following text-name entry copies a library file named BILBOS.LIB from the system disk to the source program:

        COPY BILBOS.

This text-name entry causes the compiler to search the system disk for a file named BILBOS.LIB. This search takes place in the user's directory only.

If a alphanumeric literal is entered, it may specify the full file specification for that file. For example, the following entry copies the library file BILBOS.LIB from DK1 into the source program:

        COPY "DK1:BILBOS.LIB".

This text-name entry causes the software to search DK1 for the file named BILBOS.LIB.

Only four situations require the use of the alphanumeric literal option to indicate the full file specification for the COPY statement:

    1.   when the library has a file type other than .LIB

    2.   when there is more than one device containing a library file

    3.   when the RSX-11M or IAS library (not RSTS/E) contains more than one version of the file and you want to copy a version other than the latest

    4.   when the library file is in another directory

The following examples use only the file-name option:

```
┌─────────────────────────────────┬─────────────────────────────────┐
│ COBOL SOURCE PROGRAM            │ RESULTING SOURCE PROGRAM        │
│                                 │                                 │
│      IDENTIFICATION DIVISION.   │     IDENTIFICATION DIVISION.    │
│                                 │                                 │
│      PROGRAM-ID. SAMPLE.        │     PROGRAM-ID SAMPLE.          │
│                                 │                                 │
│      COPY BILBOS.               │    ⎛AUTHOR.  BILBO BAGINS.      │
│                                 │    ⎜                            │
│      ENVIRONMENT DIVISION.      │    ⎨DATE-COMPILED.  TODAY.      │
│                                 │    ⎜                            │
│          ...                    │    ⎝SECURITY.  NONE.            │
│                                 │                                 │
│                                 │     ENVIRONMENT DIVISION.       │
│ ┌─────────────────────────────┐ │                                 │
│ │ LIBRARY FILE (BILBOS.LIB)   │ │                                 │
│ │                             │ │                                 │
│      AUTHOR. BILBO BAGINS.     │                                 │
│                                 │                                 │
│      DATE-COMPILED. TODAY.      │                                 │
│                                 │                                 │
│      SECURITY. NONE.            │                                 │
└─────────────────────────────────┴─────────────────────────────────┘
```
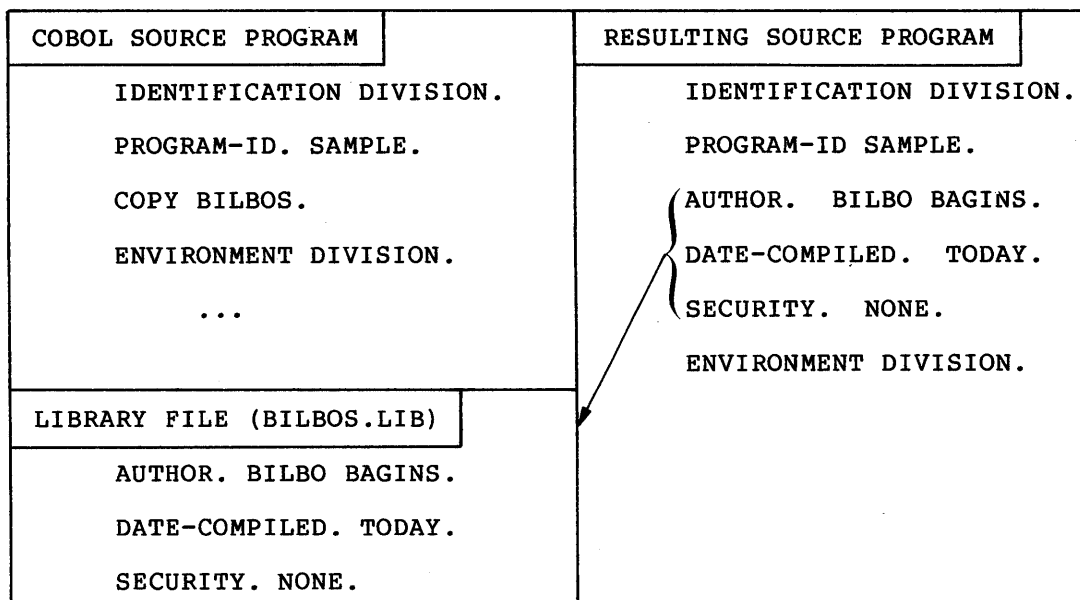
Figure 2-2 Merging a Library File

The library file in Figure 2-3 is formatted (four spaces before each entry) so that when it is merged into the source program, each entry begins in Area B. If the four spaces are not there, the text is moved into Area A and syntax errors result (Area A is reserved for division headers, paragraph headers, etc.).

```
┌─────────────────────────────────┬─────────────────────────────────┐
│ COBOL SOURCE PROGRAM            │ RESULTING SOURCE PROGRAM        │
│                                 │                                 │
│      ...                        │      ...                        │
│                                 │                                 │
│      PROCEDURE DIVISION.        │      PROCEDURE DIVISION.        │
│                                 │                                 │
│      BEGIN. COPY STARTUP.       │      BEGIN.                     │
│                                 │                                 │
│      READ-PROCEDURE.            │    ⎛OPEN INPUT FILE-A.          │
│                                 │    ⎜                            │
│          READ FILE-A. ...       │    ⎜OPEN OUTPUT FILE-B.         │
│                                 │    ⎨MOVE ZERO TO                │
│ ┌─────────────────────────────┐ │    ⎜ACCUMULATORS.              │
│ │ LIBRARY FILE (STARTUP.LIB)  │ │    ⎜                            │
│ │                             │ │    ⎝SET INDEX-1 TO 1.           │
│                                 │      READ-PROCEDURE.            │
│      OPEN INPUT FILE-A.         │                                 │
│                                 │          READ FILE-A. ...       │
│      OPEN OUTPUT FILE-B.        │                                 │
│                                 │                                 │
│      MOVE ZERO TO               │                                 │
│                                 │                                 │
│      ACCUMULATORS.              │                                 │
│                                 │                                 │
│      SET INDEX-1 TO 1.          │                                 │
└─────────────────────────────────┴─────────────────────────────────┘
```
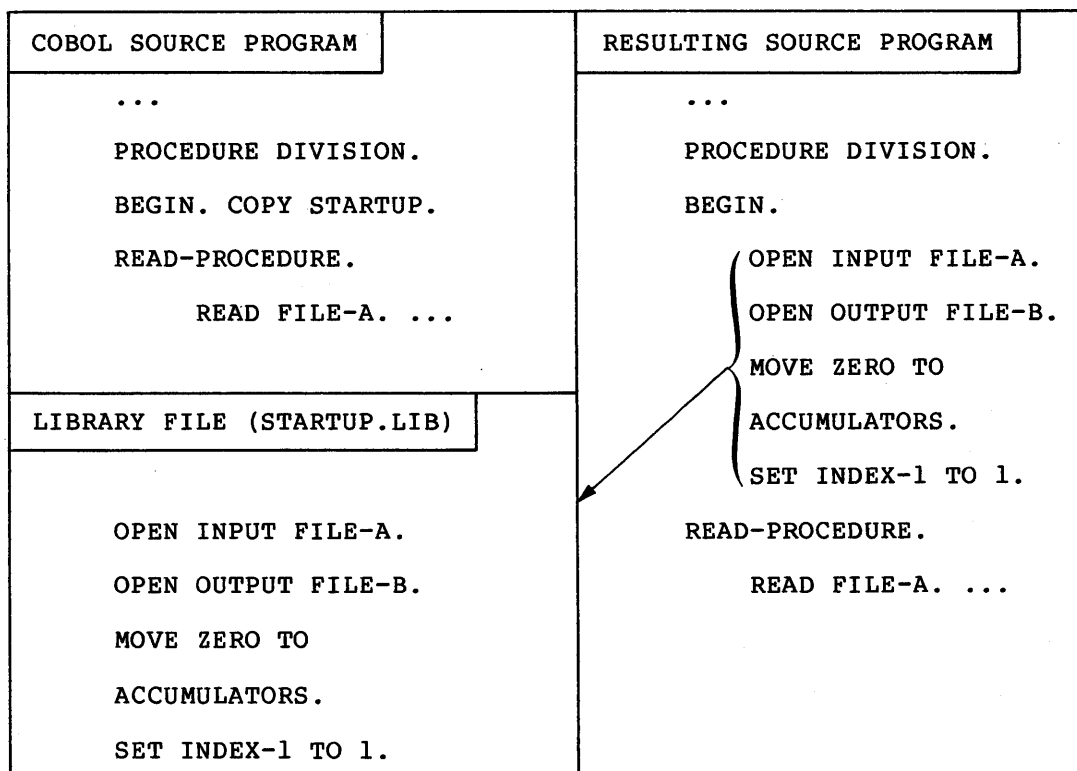
Figure 2-3 Merging a Library File Area B

Since the COPY statement may appear anywhere in a source program that a COBOL word is allowed, it can be used in various ways to solve a particular programming problem. For example, if a library file named DRACULA contains the single entry BLOOD-RATE, the entry could be used in the Data Division as follows:

SOURCE COPY STATEMENT		RESULTING SOURCE STATEMENT	
02 COPY DRACULA. PIC 99V99.		02 BLOOD-RATE PIC 99V99.	

Figure 2-4   Using the COPY Statement in a Data Description

The entry could be used in the Procedure Division as follows:

SOURCE COPY STATEMENT		RESULTING SOURCE STATEMENT	
MULTIPLY 40 BY COPY DRACULA.		MULTIPLY 40 BY BLOOD-RATE	
GIVING PLASMA.		GIVING PLASMA.	

Figure 2-5   Using the COPY Statement in a Procedural Statement

The periods terminating the COPY statements in both these examples are replaced by the text file. No periods appear in the resulting source program unless they are in the text file (if a source statement needs a period, the text file must have a period at the required place).

NOTE

The preceding two examples are not recommended uses of the COPY statement. This chapter includes them only to illustrate the mechanics of the PDP-11 COBOL library facility.

2.4.3   The COPY REPLACING Statement

Sometimes it may be necessary to tailor library text material for use in a particular program, for example, if a data description in the library text has level numbers incremented by 1 -- 01, 02, etc.   and you want them incremented by four -- 01, 05, etc.   The COPY statement can replace, during the copying process, all occurrences of a given literal or word with an alternate literal or word.   A sample COPY REPLACING statement is:

    COPY WALDEN REPLACING 02 BY 05.

This sample statement causes the compiler to scan the library file WALDEN searching for 02.   Wherever it finds a 02, it replaces it with a 05.   A match occurs only if the compiler finds a 02 (not just a 0 or just a 2).   The REPLACING character string, which may be a literal or a word, must compare equally, character for character, with the entire character string in the library text.   The following table shows some successful (YES) and unsuccessful (NO) matches:

Table 2-1
Successful and Unsuccessful Replacing Matches

GIVEN LITERAL OR WORD	LIBRARY TEXT	MATCH?
"ABC"	"ABCD"	NO
HRLY-RATE	HRLY-RATE	YES
1	1	YES
"2"	2	NO
" 15"	"15"	NO
"012"	"12"	NO
012	12	NO
SUBTRACT	SUBTRACT	YES
"012"	"012"	YES

2.4.3.1 **Examples, COPY REPLACING** - The following examples of the COPY REPLACING statement all refer to the library file named NEWSBOY:

NEWSBOY (library filename)

        01 A.

            02 B PIC 99.

            02 C PIC 99 VALUE 2.

            02 D PIC X(5) VALUE "ABCDE".

            02 E PIC 99V99 VALUE 3.75.

            02 F PIC 99 VALUE 02.

Example 1

        COPY NEWSBOY REPLACING B BY X.

This statement merges the entire file named NEWSBOY into the source program and changes data-name B to X. It does not change the character B in the character string of data-name D because it is part of a nonmatching character string. This statement causes the compiler to merge the following text into the source program:

        01 A.

            02 X PIC 99.

            02 C PIC 99 VALUE 2.

            02 D PIC X(5) VALUE "ABCDE".

            . . .

Example 2

```
COPY NEWSBOY REPLACING 2 BY 6.
```

This statement merges the entire file named NEWSBOY into the source program and changes the 2 in the entry for data-name C to a 6. It does not change the 02 in the literal entry for data-name F nor any of the 02 level numbers because they contain a nonmatching character 0. This statement causes the compiler to merge the following text into the source program:

```
01 A.

    02 B PIC 99.

    02 C PIC 99 VALUE 6.

    02 D PIC X(5) VALUE "ABCDE".

    02 E PIC 99V99 VALUE 3.75.

    02 F PIC 99 VALUE 02.
```

Example 3

```
COPY NEWSBOY REPLACING 02 BY 63.
```

This statement merges the entire file named NEWSBOY into the source program and changes not only the 02 literal entry in data-name F, but also all of the 02 level numbers to 63. Since level number 63 is illegal, this causes the compiler to produce syntax errors. The replacing process is not sensitive to the syntax of the text. The string of characters in the library may be literals, level-numbers, data-names, etc.; if they match the REPLACING string, character for character, the compiler replaces them. Consider the results of this statement:

```
01 A.

    63 B PIC 99.

    63 C PIC 99 VALUE 2.

    ...

    63 F PIC 99 VALUE 63.
```

Example 4

```
COPY NEWSBOY REPLACING B BY X,
                "ABCDE" BY "HIJKL",
                3.75 BY 4.23.
```

This statement shows how a single COPY statement can replace more than one literal or word. It causes the compiler to merge the following text into the source program:

```
01 A.

    02 X PIC 99.

    02 C PIC 99 VALUE 2.

    02 D PIC X(5) VALUE "HIJKL".

    02 E PIC 99V99 VALUE 4.23.

    02 F PIC 99 VALUE 02.
```

2.4.4 The Source Listing

Depending on how the COPY statement is written, the PDP-11 COBOL compiler lists library text material either before or after the COPY statement (Figure 2-6).

2.4.4.1 Before the COPY Statement - If other source material (including spaces) follows the COPY statement on the same source line, the compiler lists the library text before the COPY statement (see Figure 2-6).

SOURCE LINE	SOURCE LISTING
...	...
COPY CHANGES. ADD A TO B.	text-line-1
	text-line-2
	text-line-3
	COPY CHANGES. ADD A TO B.

Figure 2-6 Placing the Library Text Before the COPY Statement

The compiler does not print a source line until it has scanned the entire line. Therefore, in Figure 2-6 (CHANGES), the compiler takes the following steps, in order:

1. scans the COPY statement

2. recognizes that the COPY statement is followed by more information on the same line

3. prints the library text

4. scans the rest of the line

5. prints the entire source line

This results in a somewhat confusing listing and should be avoided. When the library text follows the COPY statement, a much more readable listing is produced.

2.4.4.2 After the COPY Statement - If the COPY statement terminates the source line, the compiler merges the library text after the COPY statement. Consider Figure 2-7:

SOURCE LINE		SOURCE LISTING	
...		...	
COPY CHANGES.		COPY CHANGES.	
ADD A TO B.		text-line-1	
		text-line-2	
		text-line-3	
		ADD A TO B.	

Figure 2-7   Placing the Library Text After the COPY Statement

In this case, the compiler takes the following steps, in order:

1. scans the COPY statement

2. prints the COPY statement

3. prints the library text

4. prints the next sequential statement

2.4.5 Common Errors in Using the Library Facility

Common errors to avoid when using the library facility are:

● Failing to follow the rules for COBOL reference format when creating the library file

● Writing the library file in one format (conventional or terminal) and the source program in the other

● Forgetting to terminate the COPY statement with a period

● Using data-names in the library file that also appear in the source program, thus causing duplicate names

● Writing the library text so that when it is merged into the source program, it becomes syntactically incorrect

● Merging the wrong library file, either because multiple versions exist or because of misspelling

● Writing other source material on the same line following the COPY statement, thus causing confusion in the source program listing

● Forgetting that numeric literals (such as 02, 77, etc.) used in the REPLACING option replace level-numbers, picture descriptions, and paragraph or section names, when they find matches.

● Forgetting that a period must appear in the text file if it
  is to appear in the source program (the period that
  terminates the COPY statement is replaced by the text).

2.5 USING THE COBOL COMPILER

The compilation process begins by invoking the PDP-11 COBOL compiler
and giving it a command line to process.

NOTE

The compiler is invoked in response to
an operating system prompting message or
code (Table 2-2).

2.5.1 Invoking the PDP-11 COBOL Compiler

The compiler can be invoked in either of two ways:

1. Enter the compiler name followed by a carriage return to
   allow multiple executions of the compiler

   or

2. Enter the compiler name followed by a COBOL command line and
   a carriage return, to execute the compiler only once

Table 2-2 depicts the system prompting message and compiler name for
each of the supported operating systems:

Table 2-2
Operating System Prompt/Compiler Name

Operating System	Prompt	Compiler Name
RSX-11M	>	CBL
IAS	PDS>	COBOL
RSTS/E	READY	COBOL

For example:

If you are running on a RSTS/E operating system, and you want to
execute the compiler for a number of consecutive compilations, enter
the following command in response to the system READY prompt:

COBOL (RET)

When the compiler is ready to process command lines, it issues the
following prompt:

CBL>

Enter the COBOL command line for the first compilation to be performed as follows:

    CBL> command-line (RET)

When the compilation is completed, the compiler reissues the CBL> prompt; you can now enter another command line.

To terminate the compiler, enter a CNTRL Z as follows:

    CBL>^Z

To execute the compiler only once, enter the following command in response to the system READY prompt:

    COBOL command-line (RET)

When the compilation is completed, the operating system issues the READY prompt.

2.5.2 COBOL Command Line

The COBOL command line has the following format:

    Object-file, Listing-file = Source-file

                    or

    @Command-file

where:

Object-file	is the file specification for the file that is to contain the compiler-generated object file.
Listing-file	is the file specification for the file that is to contain the compiler-generated listing file.
Source-file	is the file specification for the file that contains the COBOL source program to be compiled.
Command-file	is the file specification for the file that contains the COBOL command line(s) to be processed. Up to two levels of indirect command files are permitted.

Each file specification identifies a file to be used as an output or input file by the compiler.

                    NOTE

            File specifications for IAS differ in
            syntax from RSX-11M and RSTS/E. Refer
            to the User's Guide manuals for your
            particular operating system for specific
            syntax rules. For the purposes of this
            discussion, RSTS/E syntax is used.

Each file specification has the following syntactical format:

        dev:[account]filename.typ/sw.../sw

where:

dev:	= The unit on which the volume containing the desired file is mounted, e.g., DK0:. The name consists of two ASCII characters followed by an optional 1- or 2-digit octal (decimal for RSTS/E) unit number and a colon. The default value is SY:, the system disk.
[account]	= The account number under which the file is stored. In RSTS/E, this number is referred to as the project-programmer number (PPN). In RSX-11M and IAS, it is referred to as the user identification code (UIC). The default value is the account under which the system program is running.
file-name	= The name of the file. A file-name can contain up to six alphanumeric characters.
.typ	= A file type (or extension) can contain up to three alphanumeric characters. System programs default the file type to an appropriate standard type, (e.g., CBL).
/sw	= One or more ASCII characters, preceded by a slash, specifying a switch option (see Section 2.5.3 for a complete description of the PDP-11 COBOL Compiler Switches).

A command line in the syntax appropriate for your operating system is used to specify output and input files to the compiler. A maximum of two output files (object and listing) and one input (source) can be specified. The generation of either of the output files can be inhibited by omitting its file specification from the command line. Default file types are assigned to the compiler-generated output files. Table 2-3 shows the default file types expected or generated by the compiler.

<div align="center">

Table 2-3
PDP-11 COBOL Compiler Default File Types

</div>

File	Default Type	Compiler Usage
Object	OBJ	Output
Listing	LST	Output
Source	CBL	Input
Command	CMD	Input
Overlay Description Language	ODL	Output

Figure 2-8 contains samples of how the PDP-11 COBOL compiler is invoked and executed on each of the supported operating systems.

RSX-11M	<u>></u> CBL (RET)
	<u>CBL></u> OBJECT,LIST=SOURCE (RET)
	<u>CBL></u>^Z
RSTS/E	<u>READY</u>
	COBOL (RET)
	<u>CBL></u> OBJECT,LIST=SOURCE (RET)
	<u>CBL></u>^Z
IAS	<u>PDS></u> COBOL (RET)
	<u>CBL></u> OBJECT:OBJECT/LIST:LIST SOURCE (RET)
	<u>CBL></u>^Z

Figure 2-8   Sample COBOL Command Sequence

Each command line in Figure 2-8 directs the compiler to perform the following:

1. Compile the source program contained on source file   SOURCE.CBL

2. Produce a compilation listing on file   LIST.LST

3. Store the relocatable object modules on file   OBJECT.OBJ

4. Generate an overlay description language file on file   OBJECT.ODL

NOTE

An ODL file is generated whenever an object file is generated except when it is explicitly suppressed via the /-ODL switch. The file specification of the ODL file is the same as the file specification of the object file except that the extension for the ODL file is .ODL.

Either of the output files can be omitted by omitting its file specification from the command line. For example:

CBL> OBJECT=SOURCE (RET)

produces OBJECT.OBJ and OBJECT.ODL on the system device but no listing file is produced;

CBL> ,LIST=SOURCE (RET)

produces a listing file (LIST.LST) but no object module or overlay description language output files.

NOTE

In IAS, if an object file-name is not
given, the default is to use the input
file-name with the extension .OBJ. If a
listing file-name is not given, the
listing file is automatically printed on
the system line printer. The generated
list file is automatically deleted after
printing.

2.5.3 Compiler Switches

The PDP-11 COBOL compiler provides a series of compile-time switches.
You may tailor your compilation listing and assign particular
characteristics to the generated object modules via these switches.
Table 2-4 provides a list of the compiler switches and their meanings:

Table 2-4
COBOL Compiler Switches

Switch	Meaning
/ACC:n	Produce an object program only if the source program contains diagnostics with severities equal to or less than n. The range of n must be $0<n<2$, Where: 0 = Informational diagnostics 1 = Warning diagnostics 2 = Fatal diagnostics The default is /ACC:1.
/CREF	Include a cross-reference listing as part of the listing file output. When /CREF is specified, data-names, procedure-names, and source line numbers are sorted into ascending order and appended to the end of the compilation listing. The symbol # is used to indicate the line in which the referenced name is defined. NOTE The use of /CREF significantly slows down the compilation of large programs.
/CSEG:nnnn	Allows you to specify the maximum size procedural code PSECT to be produced by the compiler where nnnn is the maximum size procedural code PSECT, in decimal bytes. The minimum value of nnnn is 100.

Table 2-4 (Cont.)
COBOL Compiler Switches

Switch	Meaning
/CVF	Indicates to the compiler that the source program is in conventional format (i.e., 80-character images with Area A beginning in character position 8).
/ERR:n	Suppress the printing of diagnostics with a severity number less than n. The range of n must be 0<n<2. Where: 0 = Informational diagnostics 1 = Warning diagnostics 2 = Fatal diagnostics The switch cannot suppress severity 2 (fatal) diagnostics. (An entry of 2 suppresses the printing of all severity numbers that are less than 2.) The default is /ERR:0.
/HELP	Display on the user terminal information about how to use the compiler switches.
/KER:kk	Instruct the compiler to generate PSECT names using the two-character kernel specified by kk to make them unique to this compilation, where kk is a two character string that may contain the numbers 0 through 9 and the letters A through Z. NOTE The sample program listed in Figure 2-9 was compiled using the /KER:kk switch. See Figure 2-12 which contains the Procedure Map generated for this program. Notice that the PSECTs generated all contain the two character kernel XX.
/MAP	Produce the following special map listings: ● Data Division (Figure 2-11) ● Procedure Map (Figure 2-12) ● External Subprograms Referenced (Figure 2-17) ● Data and Control PSECTs (Figures 2-14 and 2-16) ● OTS Routines Referenced (Figure 2-15) ● Segmentation Map (Figure 2-13)
/NL	Instruct the compiler not to list the source statements copied from a library file. The resultant source listing contains only the COPY statement.

Table 2-4 (Cont.)
COBOL Compiler Switches

Switch	Meaning
/OBJ	Print the object location in which the code for each verb of the program is located. The information is listed on the line preceeding the source statement it describes (Figure 2-13).
/ODL	Instruct the compiler to generate an ODL file (default condition). To override the default condition, enter /-ODL.
/OV	Instruct the compiler to make all procedural PSECTs (segments) overlayable. Therefore, the root or main program will contain no procedural statements.
/PFM:nn	Allows you to define the maximum number of nested PERFORM statements in the program being compiled. If specified, the compiler generates a nested PERFORM stack equal in depth to the decimal number specified by nn. The default nested perform size is 10. It is to your advantage to use this switch to adjust the nested PERFORM stack size to the exact number required. This assures maximum utilization of memory in that only the exact amount of PERFORM stack space is generated.
/PLT	Directs the COBOL compiler to automatically pool literals to minimize the memory required to store them (default condition). Pooling of literals, however, slows down compiler execution speed. To bypass literal pooling for increased compiler speed, enter /-PLT.
/RO	Directs the compiler to generate read-only PSECTs for the Procedure Division object modules. The default status is read/write.
/SYM:n	Allows you to obtain more symbol table space for the compilation, where "n" (an integer in the range of 1 through 4) specifies the space required for the maximum number of data-names and procedure-names allowed in the compilation. See Table 2-5 for the correspondence between the integer specified by n, and the number of data-names and procedure-names assigned.

Table 2-5
/SYM:n Switch Values

n	Maximum Data-Names	Maximum Procedure-Names
1	761	761 (default)
2	1021	1021
3	1531	1531
4	2039	2039

The following Figures 2-9 through 2-20 show and describe sample output produced by the COBOL compiler:

```
COBOL  3.00 SRC:MAP.CBL;0                05-JAN-77  16:05:40 PAGE 001


CMD:MAP,MAP/MAP=MAP/KER:XX
IDENT: 005160


  00001           IDENTIFICATION DIVISION.
  00002           PROGRAM-ID. MAP.
  00003         *
  00004         *     EXERCISE COMPILER MAP PROCESSORS.
  00005         *
  00006          ENVIRONMENT DIVISION.
  00007          CONFIGURATION SECTION.
  00008          SOURCE-COMPUTER. PDP-11.
  00009          OBJECT-COMPUTER. PDP-11.
  00010
  00011          DATA DIVISION.
  00012          WORKING-STORAGE SECTION.
  00013          77 DEC PIC 9(4).
  00014          77 BIN PIC 9(4) USAGE COMP.
  00015          77 CHR PIC X(4).
  00016          LINKAGE SECTION.
  00017          77 L1 PIC X.
  00018          77 L2 PIC X.
  00019          77 L3 PIC X.
  00020          77 L4 PIC X.
  00021          PROCEDURE DIVISION USING L1 L3 .
  00022          S0 SECTION.
  00023          P0.
  00024          STOP RUN.
  00025          P1.
  00026          DISPLAY L3.
  00027          DISPLAY "ABC".
  00028          S1 SECTION.
  00029          P2.
  00030          MOVE DEC TO DEC.
  00031          MOVE DEC TO BIN.
  00032          MOVE BIN TO BIN.
  00033          MOVE BIN TO DEC.
  00034          MOVE CHR TO CHR.
  00035          MOVE ALL SPACES TO CHR.
  00036          MOVE DEC TO CHR.
```

Figure 2-9 Sample Source Program Listing

```
    00037          ADD DEC TO DEC.
    00038          ADD DEC TO DEC ROUNDED.
    00039          SUBTRACT DEC FROM DEC.
    00040          SUBTRACT DEC FROM DEC ROUNDED.
    00041          SUBTRACT BIN FROM BIN.
    00042          SUBTRACT BIN FROM BIN ROUNDED.
    00043          MULTIPLY BIN BY BIN GIVING BIN.

I 00043 0371  POSSIBLE HIGH ORDER RECEIVING FIELD TRUNCATION.

    00044              MULTIPLY DEC BY DEC GIVING DEC.

I 00044 0371  POSSIBLE HIGH ORDER RECEIVING FIELD TRUNCATION.

    00045              DIVIDE BIN BY BIN GIVING BIN.
    00046              DIVIDE DEC BY DEC GIVING DEC.
    00047              DIVIDE DEC BY DEC GIVING DEC ROUNDED.
    00048              DIVIDE BIN BY BIN GIVING BIN ROUNDED.


COBOL  3.00 SRC:MAP.CBL;0              05-JAN-77  16:05:40 PAGE 002


    00049          P4.
    00050              CALL "A"
    00051              CALL "AB".
    00052              CALL "ABC".
    00053              CALL "ABC000".
    00054              CALL "ABC001".
    00055              CALL "ABC002".
    00056              CALL "ABC003".
    00057              CALL "ABC004".
    00058              CALL "ABC005".
    00059              CALL "ABC006".
    00060              CALL "ABC007".
    00061              CALL "ABC008".
    00062              CALL "ABC009".
    00063              CALL "ABC010".
    00064              CALL "ABC011".
    00065              CALL "ABC012".
```

Figure 2-9 (Cont.) Sample Source Program Listing

```
COBOL    3.00  SRC:ALLORG.CBL;0              01-MAR-77  16:00:33 PAGE 003



      FILE-TO-LUN ASSIGNMENT TABLE


            NAME                   SOURCE    RELATIVE

                                   LINE       LUN


SQ-FS1                             00019     00001.
IX-FS1                             00025     00002.
RL-FS1                             00033     00003.

where:

      NAME                is a file-name that appears in the SELECT clause
                          in  the  Input-Output  Section.   The file names
                          appear in the order in which they occur  in  the
                          Input-Output Section.

      SOURCE LINE         is  the  line  on  which  the  File   Definition
                          appears.

      RELATIVE LUN        is  the  relative  logical  unit  number   (LUN)
                          assigned, beginning with 1.
```

Figure 2-10 File-to-Relative-LUN Assignment Table

```
COBOL  3.00 SRC:MAP.CBL;0               05-JAN-77  16:05:40 PAGE 003


                                 DATA MAP

   LEVEL                NAME           SOURCE  DDIV   DIR    USAGE CLASS OCC LEN
                                       LINE    LOCN   LOC

     01    DEC                         00013  000224 000000   DSP  NUM   00 0004
     01    BIN                         00014  000230 000006   CMP  NUM   00 0002
     01    CHR                         00015  000232 000014   DSP  AN    00 0004
 L   01    LS-ALPHA-DATA               00036  000000 ******   DSP  AN    00 0066
 L      05 LA1                         00037  000000 000000   DSP  AN    00 0006
 L      05 LA2                         00038  000006 000006   DSP  AN    00 0008
 L      05 LA3                         00039  000016 000014   DSP  AN    00 0010
 L      05 LA4                         00040  000030 000022   DSP  AN    00 0012
 Ł      05 LA5                         00041  000044 000030   DSP  AN    00 0014
 L      05 LA6                         00042  000062 000036   DSP  AN    00 0016
 L   01    LS-NUM-DISP-DATA            00043  000000 ******   DSP  AN    00 0021
 L      05 LN1                         00044  000000 000044   DSP  NUM   00 0004
 L      05 LN2                         00045  000004 000052   DSP  NUM   00 0005
 L      05 LN3                         00046  000011 000060   DSP  NUM   00 0005
 L      05 LN4                         00047  000016 000066   DSP  NUM   00 0002
 L      05 LN5                         00048  000020 000074   DSP  NUM   00 0002
 L      05 LN6                         00049  000022 000102   DSP  NUM   00 0003
 L   01    LS-COMP-DATA                00050  000000 ******   DSP  AN    00 0020
 L      05 LC1                         00051  000000 000110   CMP  NUM   00 0002
 L      02 LC2                         00052  000002 000116   CMP  NUM   00 0002
 L      05 LC3                         00053  000004 000124   CMP  NUM   00 0004
 L      05 LC4                         00054  000010 000132   CMP  NUM   00 0004
 L      05 LC5                         00055  000014 000140   CMP  NUM   00 0004
 L      05 LC6                         00056  000020 000146   CMP  NUM   00 0004
```

COLUMN	CONTENTS
L	Data-item is defined in linkage section
LEVEL	Level number of data-item
NAME	Data-item name
SOURCE LINE	Source line number where data-item is defined
DDIV LOCN	Octal byte offset of data-item in data storage PSECT (program section).

> NOTE: 1. For Linkage Section items, this offset is always relative to the 01 entry.
>
> 2. For non-Linkage Section items, this offset is relative to data PSECT $KKDAT (KK=kernel).

Figure 2-11 Sample Data Map

COLUMN	CONTENTS
DIR LOC	Octal byte offset of data-item's directory in a directory PSECT. NOTE: 1. For Linkage Section items, this offset is relative to data PSECT $KKARG (KK=kernel). 2. For non-Linkage Section items, this offset is relative to data PSECT $KKDDD (KK=kernel). 3. If data-item is not referenced, no directory is allocated and ****** appears.
USAGE	One of the following abbreviations: DSP - Display CMP - Computational NDX - Index
CLASS	One of the following abbreviations: ALPHA - Alphabetic NUM - Numeric AN - Alphanumeric ANEDIT - Alphanumeric edited NMEDIT - Numeric edited
OCC	The number of subscripts associated with this data-item.
LEN	The length in bytes of data-item.

Figure 2-11 (Cont.) Sample Data Map

```
COBOL  3.00 SRC:MAP.CBL;0                05-JAN-77  16:05:40 PAGE 004


                     PROCEDURE NAME MAP
     NAME                        SOURCE  PSECT   OFFSET SEG  SECT  PARA
                                 LINE

     S0                          00022   $XX001  000024  00    S
     P0                          00023   $XX001  000024  00           P
     P1                          00025   $XX001  000036  00           P
     S1                          00028   $XX002  000024  00    S
     P2                          00029   $XX002  000024  00           P
     P4                          00049   $XX002  000312  00           P
```

Column	Contents
NAME	Procedure-name
SOURCE LINE	Source line number where procedure appears
PSECT	Executable code PSECT name (program section) which contains the procedure
OFFSET	Octal byte offset of procedure in its executable code PSECT
SEG	The segment-number of the section containing the procedure
SECT	If the procedure is a section, an S will appear in this column
PARA	If the procedure is a paragraph, a P will appear in this column

Figure 2-12   Sample Procedure-Name Map

```
COBOL   3.00 SRC:MAP.CBL;0                  05-JAN-77  16:05:40 PAGE 005


                           SEGMENTATION MAP

             SECTION NAME            SEGMENT NO.      NAME             SIZE

    S0                                  00          $XX001        000116     00039
    S1                                  00          $XX002        000532     00173
```

Column	Contents
SECTION NAME	The section-name.
SEGMENT NO.	The segment-number assigned to the section.
NAME	The name of the executable procedural PSECT (program section) generated for this section. If the executable code generated for a section exceeds the code segment limit (see /CSEG switch), more than one procedural PSECT is generated for the section. If this happens, the multiple PSECT names will appear in a vertical column.
SIZE	The size of the procedural psect, octal bytes followed by decimal words.

Figure 2-13   Sample Segmentation Map

```
COBOL   3.00 SRC:MAP.CBL;0                  05-JAN-77  16:05:40 PAGE 006


COMPILER GENERATED PSECTS

NAME            SIZE

$XXENT     000036      00015
$XX003     000046      00019
```

Column	Contents
NAME	The name of the compiler-generated psect.
SIZE	The size of the PSECT: octal bytes followed by decimal words.

NOTE: This map lists those executable PSECTs (program sections) that are generated by the compiler. These PSECTs are not the result of anything in the Procedure Division, but are generated to provide for runtime execution initialization.

Figure 2-14   Sample of Compiler-Generated PSECT Map

```
COBOL   3.00  SRC:MAP.CBL;0                 05-JAN-77  16:05:40 PAGE 007



REFERENCED OTS ROUTINES

$XMBB      $XMDD      $XMDB      $XMBD      $XMCC      $XMAL      $XADDD      $XADDR
$XSUBD     $XSUBR     $XMULB     $XSUBB     $XSBBR     $XMULB     $XDIVB      $XDIVR
$XEDIS     $XGO       $XENDP     $XSTPR     $XABRT     $XCALL     $XEXIT      $XSUBK
$XINIT
```

Contents

This map contains the names of all COBOL OTS (Object Time System)
routines that are referenced by the code generated by the compiler.

Figure 2-15   Sample Map of Referenced OTS Routines

```
COBOL   3.00   SRC:MAP.CBL;0                05-JAN-77  16:05:40 PAGE 008



DATA PSECT MAP

NAME            SIZE

$XXDAT    000236     00079
$XXDDD    000022     00009
$XXPDT    000014     00006
$XXARG    000006     00003
$XXWRK    000102     00033
$XXLIT    000006     00003
$XXLTD    000014     00006
$XXLST    000002     00001
$XXPFM    000214     00070
$XXSDT    000040     00016
$XXADT    000000     00000
$XXUSE    000030     00012
$CBIOT    000126     00043
$CBFA1    000000     00000
$CBXA1    000000     00000
$XXIOB    000000     00000
$CBIF1    000000     00000
$CBIR1    000000     00000
$CBKD1    000000     00000
$CBBD1    000000     00000
$CBKB1    000000     00000
$CBFD1    000000     00000
$CBSWT    000002     00001
```

Column	Contents
NAME	The name of the data PSECT generated by the compiler.
SIZE	The size of the data PSECT, in octal bytes, followed by decimal words.

NOTE: This map lists the data (nonexecutable) PSECTs generated by the
compiler. See Appendix D for a description of the Data PSECTs
generated for each compilation.

Figure 2-16   Sample Data PSECT MAP

```
COBOL   3.00   SRC:MAP.CBL;0              05-JAN-77   16:05:40 PAGE 009



EXTERNAL SUBPROGRAM REFERENCES

A          AB         ABC        ABC000    ABC001    ABC002    ABC003    ABC004
ABC009     ABC010     ABC011     ABC012
```
```
Contents

This map contains the names of all external subprograms referenced  by
CALL statements in the COBOL source program.
```

Figure 2-17   Sample Map of External Subprogram References MAP

```
COBOL   3.00   SRC:MAP.CBL;0              05-JAN-77   16:05:40 PAGE 010



SEVERITY      ERROR COUNT

    I            9
    W            4
    F            2
```
```
  Column                                  Contents

SEVERITY              Contains the error severity code.   The  following
                      list contains the possible severity codes:

                            I = Information

                            W = Warning

                            F = Fatal

ERROR COUNT           Contains the number of errors encountered.

NOTE:   This listing is generated for every COBOL compilation.
```

Figure 2-18   Sample Compilation Error Count Listing

```
COBOL  3.00  SRC:MAP.CBL;0              05-JAN-77  16:05:40 PAGE 011


   COMPILER GENERATED ODL FILE

;COBOL STANDARD ODL FILE GENERATED ON: 05-JAN-77  16:05:40
;COBOBJ=MAP.OBJ
;COBKER=XX
        .NAME XX$050,GBL
        .PSECT $XX003,GBL,I,RW,CON
XX050$: .FCTR *XX$050-$XX003
XXOVR$: .FCTR          XX050$
```

NOTE

This listing is generated whenever an
object file is generated.

Figure 2-19 Sample Compiler-Generated ODL File Listing

```
   00110            MOVE REPETITIONS(POWER) TO REPS.
PERFORM   : 001 000104
   00111            PERFORM TEST1 REPS TIMES.
MOVE      : 001 000122
   00112            MOVE TESTDELTA(TESTNUMBER) TO BASETIME(POWER) PEOPLE
DISPLAY   : 001 000160
   00113            DISPLAY "10**" POWER " REPETITIONS TOOK "
   00114             PEOPLETIME " HUNDREDTHS OF SECONDS.".
ADD       : 001 000206
   00115            ADD 1 TO POWER.
IF        : 001 000220
GO        : 001 000254
   00116            IF POWER NOT > POWERLIMIT GO TO I5.
GO        : 001 000264
   00117            GO TO I10.
```

For each COBOL verb, a line of the following format appears in the
listing, preceding the source line that contains the verb:

    VERB: PPP AAAAAA

where:

    1. VERB is the verb name

    2. PPP is the decimal number of the code PSECT containing the
        object code generated for the verb.

    3. AAAAAA is the octal byte offset within the PSECT at which the
        object code is generated.

Figure 2-20 Sample Output Using OBJ Switch

2.5.4 Examples of the COBOL Command Line

Example 1

CBL> BILBO,BILBO=BILBO (RET)

This command line instructs the compiler to compile source program
BILBO.CBL and write the compilation listing to a file named BILBO.LST.
It produces an object file named BILBO.OBJ, and an overlay description
language file named BILBO.ODL. The source program BILBO.CBL exists on
the system device and is expected to be in terminal format. All
output files exist or will be created on the system device.

Example 2

CBL> DK1:BILBO.OBJ,LP:=DT0:BILBO/CVF/MAP (RET)

This command line instructs the compiler to compile source program
BILBO.CBL. The compiler expects the source program to reside on
DECtape 0 and be in conventional format. This command line also
instructs the compiler to generate a compiler listing and the
following map listings on the system line printer:

- Data

- Procedure

- External Subprogram Reference

- Segmentation

- Data and Code PSECT

- OTS Routines

An overlay description language (ODL) file named BILBO.ODL and an
object module (OBJ) file named BILBO.OBJ are to be created or replaced
on disk device DK1:.

Example 3

CBL> BILBO,BILBO=BILBO/KE:BB (RET)

This command line instructs the compiler to compile source program
BILBO.CBL which resides in terminal format on the system device. The
output listing is to be spooled to a file named BILBO.LST. An object
file (BILBO.OBJ) and an overlay description language file (BILBO.ODL)
are to be generated and stored on the system device. The PSECT names
generated by the compiler for the object modules all begin with the
characters $BB.

Example 4

CBL> BILBO.OBJ/KE:BB,BILBO.LST/MAP=BILBO.CBL/CVF (RET)

This command line instructs the compiler to compile program BILBO.CBL,
which, is coded in conventional format and resides on the system
device. A listing file (BILBO.LST), created on the system device,
contains the following maps:

- Data

- Procedure

- External Subprogram Reference

- Segmentation

- Data and Code PSECT

- OTS Routines

An object file (BILBO.OBJ) and an overlay description language file (BILBO.ODL) are generated and stored on the system device. The PSECT names generated during this execution of the compiler begin with the characters $BB.

2.5.5 Error Message Summary

If any errors were detected during compilation, the compiler generates an error message summary and prints it on the user terminal. This summary contains the number of errors encountered. The error message summary has the following format:

CBL -- NNNNN ERROR(S), NNNNN FATAL

Where:

NNNNN is the number of errors encountered.

NOTE

If fatal errors are encountered, no object file is generated, unless specifically requested via the /ACC switch. (See Section 2.5.3, Compiler Switches).

2.5.6 Common Entry Errors, COBOL Command String

Common errors to avoid when entering the COBOL command string are:

- omitting the /CVF switch for programs that are in the conventional format, and causing a system error

- leaving the colon off the LP: specification, causing the listing to be spooled to a file named LP.LST

- turning ON switches that contradict the file specification parameters, e.g., entering the command string BILBO=BILBO/MAP, which does not request a listing file, yet does request a Data Division map

- omitting version numbers from one or more of the file specifications, causing the system to create a new version or to compile the wrong version (IAS or RSX-11M).

2.6 USING THE MERGE UTILITY

To convert the ODL file generated by the compiler into a complete ODL file, or to merge ODL files from more than one compilation into a single ODL file, you use the Merge Utility.

2.6.1 Invoking the Merge Utility

To invoke the Merge Utility, type MRG in response to a system prompt (Section 2.5, Using the COBOL Compiler). When the Merge Utility is loaded and ready to accept input specifications, it issues the following message:

    PLEASE ENTER FILE SPECIFICATION FOR OUTPUT FILE

Enter the file specification for the file that is to receive the merged ODL file. For example:

    <u>PLEASE ENTER THE FILE SPECIFICATION FOR OUTPUT FILE</u>

    BILBO.ODL (RET)

When the output file specification is received, Merge issues another prompt:

    DO YOU WANT AN ABBREVIATED OR MERGED ODL FILE?

    PLEASE ANSWER A (ABBREVIATED) OR M (MERGED)

If you enter M, the Merge Utility generates a file that is a concatenation of all the input ODL files. If you enter an A, an ODL file containing indirect command file specifications for each input ODL file is generated.

Input ODL Files	Merged ODL		Abbreviated ODL	
BILBO1.ODL : BILBO2.ODL : BILBO3.ODL :	BILBO1.ODL : BILBO2.ODL : BILBO3.ODL : Merge supplied ODL statements		@BILBO1.ODL @BILBO2.ODL @BILBO3.ODL Merge supplied ODL statements	

Figure 2-21   Merged vs. Abbreviated ODL File

For example, the following results in the generation of an abbreviated file:

    <u>DO YOU WANT AN ABBREVIATED OR MERGED ODL FILE?</u>

    <u>PLEASE ANSWER A (ABBREVIATED) OR M (MERGED)</u> A (RET)

Following the A or M specification, Merge make the following request:

    DO YOU WANT TO OVERLAY I/O SUPPORT ROUTINES?

    PLEASE ANSWER Y(ES) OR N(O)

Simply enter Y for yes and N for no, followed by a carriage return.

Merge now requests that you enter the file specification for the first or only ODL file to be merged. The following message is issued:

PLEASE ENTER FILE SPECIFICATION FOR INPUT ODL FILE

Enter the input ODL file specification in response to this message as follows:

file-specification ⏺RET

For example:

PLEASE ENTER FILE SPECIFICATION FOR INPUT ODL FILE

BILBO1.ODL ⏺RET

When Merge has completed processing this input file, it requests you to enter the device and directory under which the associated object file is stored. The following message is issued:

OBJECT PROGRAM REFERENCED IN ODL FILE IS:

object-filename.ext

PLEASE ENTER OBJECT FILE DEVICE AND PPN IN

THE FORMAT: DEV:[PROJECT,PROGRAMMER]

Enter the device code and PPN if different from the system default assignment. Otherwise, enter a carriage return only, e.g.:

OBJECT PROGRAM REFERENCED IN ODL FILE IS:

BILBO1.OBJ

PLEASE ENTER OBJECT FILE DEVICE AND PPN IN

THE FORMAT: DEV:[PROJECT,PROGRAMMER]

⏺RET

The processing of the input ODL file is complete. Merge now issues the following message:

ANY MORE INPUT ODL FILES?
PLEASE ANSWER Y(ES) OR N(O)

Enter Y for yes and N for no followed by a carriage return. If Y is entered, Merge reissues the PLEASE ENTER FILE SPECIFICATION FOR INPUT ODL FILE message, and the merging procedure is repeated. If N is entered, the output file is completed, and the following message is issued:

ODL MERGE IS COMPLETE
MERGED ODL FILE IS file-specification

Figure 2-22 shows a sample ODL file merge dialogue.

```
RUN MRG  (RET)

PLEASE ENTER FILE SPECIFICATION FOR OUTPUT FILE

DIVA.ODL  (RET)

DO YOU WANT AN ABBREVIATED OR MERGED ODL FILE?

PLEASE ANSWER A(BBREVIATED) OR M(ERGED)  M  (RET)

DO YOU WANT TO OVERLAY I/O SUPPORT ROUTINES?

PLEASE ANSWER Y(ES) OR N(O)  N  (RET)

PLEASE ENTER FILE SPECIFICATION FOR INPUT ODL FILE

DIVBUG.ODL  (RET)

OBJECT PROGRAM REFERENCED IN ODL FILE IS:

     DIVBUG.OBJ

PLEASE ENTER OBJECT FILE DEVICE AND UIC IN THE FORMAT:

DEV:[GROUP,MEMBER]

(RET)

ANY MORE INPUT ODL FILES?

PLEASE ANSWER Y(ES) OR N(O)  N  (RET)

ODL FILE MERGE COMPLETE

MERGED ODL FILE IS: DIVA.ODL

CBL -- 15: STOP RUN
```

Figure 2-22 Sample ODL File Merge Dialogue

2.6.2 Merge Utility Error Messages

Whenever the Merge Utility encounters an error in processing, it issues an error message to the user terminal. These error messages are listed in Table 2-6.

Table 2-6
Merge Error Messages

BAD FORMAT PPN:[p,p]

    Description   The PPN you specified does not conform to system standard syntax.

    User Action   Respecify the PPN using the correct syntax.

THIS ODL FILE CONTAINS A ;COBMAIN LINE

A ;COBMAIN LINE HAS ALREADY OCCURRED

THIS ODL FILE IS IGNORED

    Description   A ;COBMAIN line in an ODL file identifies the object program as a main program. This message is telling you that Merge has already processed an ODL file that contained a ;COBMAIN line. Since a task image can only contain one main program, this ODL file is ignored.

MULTIPLE ;COBKER HEADER LINE DETECTED

THIS ODL FILE IS IGNORED

    Description   A ;COBKER line is an ODL file specifies the 2 character kernel used to identify PSECTs for the object file corresponding to this ODL file. Only one ;COBKER line per ODL file is allowed. This ODL file is ignored.

MULTIPLE ;COBOBJ HEADER LINE DETECTED

THIS ODL FILE IS IGNORED

    Description   A ;COBOBJ line in an ODL file identifies the object file for which the ODL file was generated. Only one such object file specification is allowed in a compiler-generated ODL file. This ODL file is ignored.

NOT STANDARD COBOL ODL FILE

FILE IS IGNORED

    Description   The ODL file contains nonstandard ODL lines. See Chapter 11 for ODL file format.

OPEN UNSUCCESSFUL

    Description   One of the following conditions exists:

                1.   The device is not on line
                2.   The device is not mounted
                3.   The hardware has failed
                4.   The file does not exist
                5.   The user is not allowed access to the file

    User Action   1.   Determine which condition exists
                2.   Rectify the condition
                3.   Reenter the command

Table 2-6 (Cont.)
Merge Error Messages

```
READ ERROR -- MUST ABORT

    Description     An unrecoverable read error  occurred  when  the
                    Merge  Utility  attempted  to read the input ODL
                    file.  The  input  and  output  ODL  files  are
                    closed, and the Merge Utility terminates.

                    One of the following conditions exists:

                    1.  The device is not on line
                    2.  The device is not mounted
                    3.  The hardware has failed
                    4.  The volume is full

    User Action     1.  Determine which condition exists
                    2.  Rectify the condition
                    3.  Reenter the command
```

2.7 USING THE TASK BUILDER

The Task Builder (TKB) is a system program that links relocatable
object modules to create a task image. Invoke the Task Builder by
specifying TKB in response to an RSX-11M or RSTS/E operating system
prompt, or by specifying LINK in response to an IAS operating system
prompt. For example, you invoke TKB on a RSTS/E system as follows:

    READY

    TKB (RET)

When invoked, the Task Builder issues the following prompt:

    TKB>

You can now enter the TKB command line using the following format.

    task file, map file = ODL file/MP

where:

    task file is the file specification for the task image file to be
               generated.

    map file   is optional; if specified, it is the specification for
               the file that is to receive the memory map listing.

    ODL file   is the file specification for an Overlay Description
               Language file.

    /MP   is a switch which identifies the preceding   file
               specification as an ODL file.

When TKB receives the command string, it issues the following message:

    ENTER OPTIONS:

The RSTS/E user enters HISEG=RMS11 followed by a carriage return in response to this prompt. The IAS or RSX-11M user enters / followed by a carriage return unless other options are to be entered first. When appropriate, the UNITS and ASG options should be entered here. (See Section 6.5.3, Files and Logical Units.)

TKB then issues the following prompt:

> TKB>

Enter // in response to this prompt.

Consider the following example of running TKB on a RSTS/E operating system:

> TKB   (RET)
>
> TKB>DIVBUR,DIVBUG=DIVA/MP   (RET)
>
> ENTER OPTIONS:
>
> TKB>HISEG=RMS11   (RET)
>
> TKB>//   (RET)

2.7.1 Task Building COBOL Programs Using Direct Input

If your task does not contain calls to subprograms or segmentation and is composed of only a few object modules, you may want to bypass the Merge Utility process and enter a dialogue with TKB. The format of the command line is:

> task,map=object1,object2,...,[1,1]COBLIB/LB,[1,1]RMSLIB/LB

Where:

task	is the file specification for the task image file to be generated.
map	(optional) is the file specification for the file to receive the memory map listing.
object	is one or more object file specifications for the file(s) containing the object modules to be task-built.
[1,1]COBLIB	is the exact file specification for the COBOL library file. This file contains the COBOL object time system.
[1,1]RMSLIB	is the exact file specification for the RMS-11 library file. This file contains the RMS I/O routines.
/LB	is the Task Builder switch that describes the file as being a library file.

TKB is invoked and the options following the ENTER OPTIONS prompt are specified in the same manner as specified in Section 2.7.

The following dialogue shows TKB commands to task-build using object module(s) and libraries.

TKB (RET)

TKB>DIVBUG,DIVBUG=DIVBUG,SUBA,[1,1]COBLIB/LB,[1,1]RMSLIB/LB (RET)

TKB>/ (RET)

ENTER OPTIONS:

TKB>HISEG=RMS11 (RET)

TKB>// (RET)

2.7.2 Task Building and COBOL Program Size

The size of a COBOL object module created form a COBOL source file depends on the amount of memory required for the following elements:

1. data-items in the Working-Storage Section

2. number of files in the File Section

3. amount of code generated for the Procedure Division

4. number and length of all unique numeric and non-numeric literals used in the Procedure Division

5. total size of all runtime modules needed to support the executing program (includes such code as arithmetic support, I/O support, segmentation support, etc.)

6. push-down stack space required to support the executable code

7. directories for all referenced data-items and literals

The maximum size of a program task on the PDP-11 is 64K bytes. On systems that support PDP-11 COBOL, the maximum task is 56K bytes, where the remaining 8K bytes are reserved for sharable file system code to support I/O. A COBOL program, therefore, cannot occupy more than 56K bytes of memory.

Some programs may exceed the allowable storage byte limit. In this case, TKB issues a diagnostic and does not build your task. You may take either of two corrective measures:

1. create program overlays by using the COBOL segmentation facility (Chapter 9, Segmentation)

2. break up your program into a main program and one or more subprograms (Chapter 10, Interprogram Communication).

2.8 EXECUTING A COBOL TASK

Once a task image has been built, you execute it by entering the following command in response to a system prompt:

RUN filespec (RET)

where: filespec is the task image file specification.

2.8.1 COBOL Switch Setting

If you specified SWITCH ON or OFF in the SPECIAL-NAMES paragraph of your source program, at execution time the OTS will issue the following message to your terminal:

SPECIFY "ON" SWITCHES...

Enter a response in the following format:

N1,N2,N3,...,NK

where

N1,N2,N3,...,NK are decimal numbers in the range 1 thru 16 that specify the switch you want turned on.

For example:

SPECIFY "ON" SWITCHES ...

1,9,16 (RET)

This example causes switches 1, 9, and 16 to be set on.

If you enter an asterisk (*) in response to the switch message, all switches are set on.

CHAPTER 3

NON-NUMERIC CHARACTER HANDLING

3.1 INTRODUCTION

COBOL programs hold their data in fields whose sizes are described in their source programs. These fields are thus "fixed" during compilation to remain the same size throughout the lifespan of the resulting object program.

The data descriptions of the fields in a COBOL program describe them as belonging to any of three data classes -- alphanumeric, alphabetic, or numeric class. Numeric class data items contain only numeric values, alphabetic class only A-Z and space, but alphanumeric class data items may contain values that are all alphabetic, all numeric, or a mixture of alphabetic bytes, numeric bytes, or, in fact, any character from the ASCII character set.

Further, these three classes are subdivided into five categories: alphabetic, numeric, numeric edited, alphanumeric edited, and alphanumeric. Every elementary item except for an index data item belongs to one of the classes and further to one of the categories. The class of a group item is treated at object time as alphanumeric regardless of the classes of subordinate elementary items.

For alphabetic and numeric (data items) class and category are synonymous.

An alphabetic field is a field declared to contain only alphabetic (A-Z and space) characters.

An alphanumeric class field that is declared to contain any ASCII character is called an alphanumeric category field.

If the data description of an alphanumeric class field specifies that certain editing operations will be performed on any value that is moved into it, that field is called an alphanumeric or numeric edited category field.

When reading the following sections of this chapter, this distinction between the class or category of a data item and the actual value that the item contains should always be kept in mind.

Sometimes the text refers to alphabetic, alphanumeric, and alphanumeric edited data items as non-numeric data items. This is 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 object time, that is "illegal". Thus, non-numeric ASCII characters can be placed into a field described as numeric class, and an alphabetic class field may be loaded with non-alphabetic characters.

3-1

To increase readability, the following sections occasionally omit the word "class" when describing an item; however, the reader should regard the descriptive word, numeric, alphabetic, or alphanumeric, as referring to the class of an item unless it applies specifically to the value in the item.

This chapter discusses non-numeric class data and the non-arithmetic, non-input-output operations that manipulate this type of data.

3.2 DATA ORGANIZATION

Usually, the data areas in a COBOL program are organized into group items with subordinate elementary items. A group item is a data item that is followed by one or more data items (elementary items) with higher valued level numbers. An elementary item has no higher valued subordinate level number.

All of the data areas used by COBOL 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 fields and fixes them in size at compilation time.

The following sub-sections (3.2.1 and 3.2.2) discuss, on a general level, how the compiler handles group and elementary data items.

3.2.1 Group Items

The size of a group item is the total size of the data area occupied by its subordinate elementary items. The compiler considers group items to be alphanumeric DISPLAY items. Thus, the software manipulates group items as if they had been described as PIC X() items, and ignores the structure of the data contained within them.

3.2.2 Elementary Items

The size of an elementary item is determined by the number of allowable symbols it contains that represent character positions. For example, consider the following fields:

```
01 TRANREC.
    03 FIELD-1 PIC X(7).
    03 FIELD-2 PIC S9(5)V99.
```

Figure 3-1
Field Sizes

Both fields consume seven bytes of memory; however, FIELD-1 contains seven alphanumeric bytes while FIELD-2 contains decimal digits and an operational sign. Although certain verbs handle these two classes of data differently, the data, in either case, occupies seven bytes of PDP-11 memory. COBOL operations on such fields are independent of the mapping of the field into PDP-11 memory words (16-bit words that hold two 8-bit bytes). Thus, a field may begin in the left or right-hand byte of a word with no effect on the function of any operations that may refer to that field.

In effect, the compiler sees memory as a continuous array of bytes, not words. This becomes particularly important when declaring a table with the OCCURS clause (see Chapter 5, Table Handling).

Records (a 01 level entry and all of its subordinate entries) and data items that have a level number of 77 and all literal values given in the Procedure Division automatically begin on even byte addresses.

I/O verbs require that records be aligned on word boundaries because the PDP-11 COBOL file system reads and writes integral numbers of words.

Non-input-output verbs do not require alignment of the data. However, when two fields are aligned identically, the processing verb can sometimes increase its efficiency by processing them a word at a time rather that a byte at a time.

In all cases, automatic word alignment of literals, records, and/or 77 items increase the opportunity for more efficient processing.

3.3 SPECIAL CHARACTERS

COBOL allows the user 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 are performed in a manner that attaches no "meaning" to an 8-bit byte. Thus, the user 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 non-numeric 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 non-numeric literals.

Special characters may be placed into data fields of the object program by placing the binary value of the special character into a numeric COMP field and redefining that field 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.
```

Figure 3-2
Redefining Special Characters

The sample coding in Figure 3-2 introduces each character as a 1-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 fields.)

The following ASCII code chart may be used to determine the decimal value for any ASCII character. To use the chart, find the desired character; then add its row and column values together to determine the decimal integer to be supplied as a VALUE for the computational item.

Column Value / Row Value	000	008	016	024	032	040	048	056	064	072	080	088	096	104	112	120
0	NUL	BS	DLE	CAN	space	(	0	8	@	H	P	X	grave	h	p	x
1	SOH	HT	DC1	EM	!	)	1	9	A	I	Q	Y	a	i	q	y
2	STX	LF	DC2	SUB	"	*	2	:	B	J	R	Z	b	j	r	z
3	ETX	VT	DC3	ESC	#	+	3	;	C	K	S	[	c	k	s	|
4	EOT	FF	DC4	FS	$	,	4	<	D	L	T	\	d	l	t	|
5	ENQ	CR	NAK	GS	%	-	5	=	E	M	U	]	e	m	u	|
6	ACK	SO	SYN	RS	&	.	6	>	F	N	V	(↑)	f	n	v	(ESC)
7	BEL	SI	ETB	US	apos	/	7	?	G	O	W	(±)	g	o	w	DEL

Figure 3-3
ASCII Code Chart

3.4 TESTING NON-NUMERIC FIELDS

3.4.1 Relation Tests

An IF statement that contains a relation condition (greater-than, less-than, equal-to, etc.) can compare the value in a non-numeric 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 may be an identifier or a literal, except that both cannot be literals. If the relation exists between the two operands, the relation condition has a truth value of true.

Figure 3-4 illustrates the general format of a relation condition. (The relational characters_">," "<," and "=," although required, are not underlined to avoid confusion with other symbols such as greater-than-or-equal-to.)

```
                               ⎛IS [NOT] GREATER THAN⎞
                               ⎜IS [NOT] LESS  THAN  ⎟
⎛identifier-1            ⎞     ⎜IS [NOT] EQUAL  TO   ⎟     ⎛identifier-2            ⎞
⎨literal-1              ⎬     ⎨IS [NOT] >           ⎬     ⎨literal-2              ⎬
⎝arithmetic-expression-1⎠     ⎜IS [NOT] <           ⎟     ⎝arithmetic-expression-2⎠
                               ⎝IS [NOT] =           ⎠
```
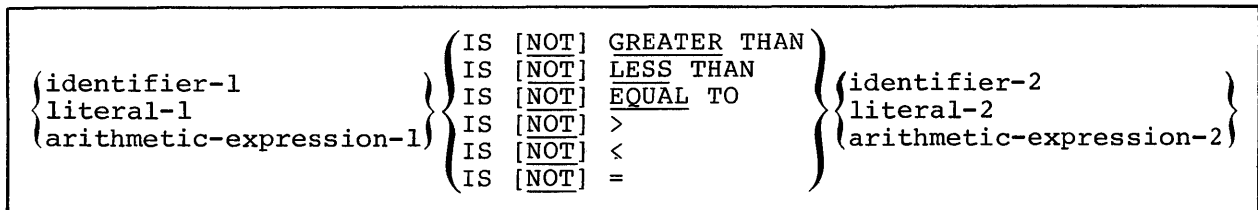
Figure 3-4
Relation Condition

When coding a relational operator, leave a space before and after each reserved word. When the reserved word NOT is present, the software considers it and the next key word or relational character to be one relational operator that defines the comparison. Figure 3-5 shows the meanings of the relational operators.

OPERATOR	MEANING
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.

Figure 3-5
The Meanings of the Relational Operators

3.4.1.1 **Classes of Data** - COBOL allows comparison of both numeric class operands and non-numeric class operands; however, it handles each class of data slightly differently. For example, it allows a comparison of two numeric operands regardless of the formats specified in their respective USAGE clauses, but 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 non-numeric (or a numeric and a non-numeric) 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; further, the manner in which the software handles numeric operands depends on the non-numeric operand. Consider the following two types of non-numeric operands:

1. If the non-numeric operand is an <u>elementary item</u> or a <u>literal</u>, the software treats the numeric operand as if it had been moved into an alphanumeric data item (which is the same size as the numeric operand) and then compared. This causes any operational sign, whether carried as a separate character or as an overpunch, to be stripped from the numeric item; thus, it appears to be an unsigned quantity. In addition, if the picture-string of the numeric item contains trailing P characters indicating that there are assumed integer positions that are not actually present, these are filled with zero digits during the operation of stripping any sign that is present. Thus, an item with a picture-string of S9999PPP is moved to a temporary location where it is described with a picture-string of 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 non-numeric operand is a <u>group item</u>, the software treats the numeric operand as if it had been moved into a group item (which is the same size as the numeric operand) and then compared. This is equivalent to a "group move". The software ignores the description of the numeric field (except for length) and, therefore, includes any operational sign, whether carried as a separate character or as an

overpunch, in its length. (Overpunched characters are never ASCII numeric digits, but characters in the range of from A through R, {, or }.) Thus, the sign and the digits, stored as ASCII bytes, take part in the comparison, and zeroes are not supplied for P characters in the picture-string.

The compiler will not accept a comparison between a non-integer numeric operand and a non-numeric operand, and any attempt to compare these two items will cause a diagnostic message at compile time.

3.4.1.2 **The Comparison Operation** - If the two operands are acceptable, the software compares them byte for byte starting at their left-hand end. It proceeds from left to right, comparing the characters in corresponding character positions until it either encounters a pair of unequal characters or reaches the right-hand end of the longer operand.

If the software 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 (040) to make them both the same length.

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

3.4.2 **Class Tests**

An IF statement that contains a class condition (NUMERIC or ALPHABETIC) can test the value in a non-numeric data item (USAGE DISPLAY only) to determine if it contains numeric or alphabetic data and use the result to alter the flow of control in the program.

Figure 3-6 illustrates the general format of a class condition. If the data item consists entirely of the ASCII characters 0123456789 with or without the operational sign, the class condition would determine that it is NUMERIC. If the item consists entirely of the ASCII characters A through Z and space, the class condition would determine that it is ALPHABETIC.
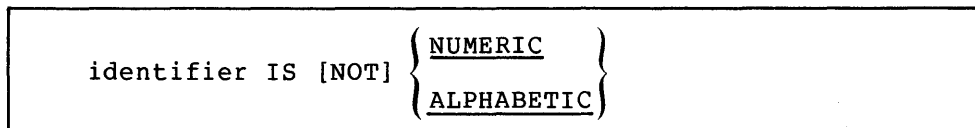
```
identifier IS [NOT]  ( NUMERIC    )
                     { ---------- }
                     ( ALPHABETIC )
```

Figure 3-6
Class Condition, General Format

When the reserved word, NOT, is present, the software considers it and the next key word as one class condition that defines the class test to be executed; for example, NOT NUMERIC is a truth test for determining if an operand contains at least one non-numeric byte.

If the item being tested was described as a numeric data item, it may only be tested as NUMERIC or NOT NUMERIC. (For further information on using class conditions with numeric items, see Chapter 4.) The NUMERIC test cannot examine an item that was described either as alphabetic or as a group item containing elementary items whose data descriptions indicate the presence of operational signs.

3.5 DATA MOVEMENT

COBOL provides three statements (MOVE, STRING, and UNSTRING) that perform most of the data movement operations required by business-oriented programs. The MOVE statement simply moves data from one field to another. The STRING statement concatenates a series of sending fields into a single receiving field. The UNSTRING statement disperses a single sending field into multiple receiving fields. Each has its uses and its limitations. This section discusses data movement situations which take advantage of the versatility of these statements.

The MOVE statement handles the majority of data movement operations on character strings. However, the MOVE statement has limitations in its ability to handle multiple fields; for example, it cannot, by itself, concatenate a series of sending fields into a single receiving field or disperse a single sending field into several receiving fields.

Two MOVE statements will, however, bring the contents of two fields together into a third (receiving) field if the receiving field has been "subdivided" with subordinate elementary items that match the two sending fields in size. If other fields are to be concatenated into the third field and they differ in size from the first two fields, then the receiving field will require additional subdivisions (through redefinition).

Another method of concatenation with the MOVE statement is to subdivide the receiving field into single character fields, creating a "table" of a single character field that occurs as many times as there are characters in the receiving field, and execute a data movement loop which moves each sending field, a character at a time, using a subscript that moves continuously across the receiving field.

Two MOVE statements can also be used to disperse the contents of one sending field to several receiving fields. The first MOVE statement can move the left-most end of the sending field to a receiving field; then the second MOVE statement can move the right-most end of the sending field to another receiving field. (The second receiving field must first be described with the JUSTIFIED clause.) Characters from the middle of the sending field cannot easily be moved to any receiving field without extensive redefinitions of the sending field or a character-by-character movement loop (as with concatenation).

The concatenation and dispersion limitations of the MOVE statement are handled quite easily by the STRING and UNSTRING statements. The following sections (3.6, 3.7, and 3.8) discuss these three statements in detail.

3.6 THE MOVE STATEMENT

The MOVE statement moves the contents of one field into another. The following illustration shows the two formats of the MOVE statement.

```
Format 1

        MOVE FIELD1 TO FIELD2

Format 2

        MOVE CORRESPONDING FIELD1 TO FIELD2


                    NOTE

Format 2 is discussed in Section 3.6.6.
```

FIELD1 is the name of the sending field and FIELD2 is the name of the receiving field. The statement causes the software to move the contents of FIELD1 into FIELD2. The two fields need not be the same size, class, or usage; and they may be either group or elementary items. If the two fields are not the same length, the software will align them on one end or the other -- and will truncate or pad (with spaces) the other end. The movement of group items and non-numeric elementary items is discussed below.

A point to remember when using the MOVE statement is that it will alter the contents of every character position in the receiving field.

3.6.1 Group Moves

If either the sending or receiving field is a group item, the software considers the move to be a group move. It treats both the sending and receiving fields in a group move as if they were alphanumeric class fields. If the sending field is a group item and the receiving field is an elementary item, the software ignores the receiving field description (except for the size description, in bytes, and any JUSTIFIED clause); therefore, the software conducts no conversion or editing on the receiving field.

3.6.2 Elementary Moves

If both fields of a MOVE statement are elementary items, their data description clauses control their data movement. (If the receiving field was described as numeric or numeric edited, the rules for numeric moves -- see Chapter 4, Numeric Character Handling -- control the data movement.)

The following table shows the legal (and illegal) non-numeric elementary moves.

NON-NUMERIC CHARACTER HANDLING

Table 3-1
Legal Non-Numeric Elementary Moves

SENDING FIELD CATEGORY	RECEIVING FIELD 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 of the legal moves shown above, the software treats the sending field as though it had been described as PIC X(). If the sending field description contains a JUSTIFIED clause, the clause will have no effect on the move. If the sending field picture-string contains editing characters, the software uses them only to determine the field's size.

Numeric class data must be in DISPLAY (byte) format and must be an integer.

If the description of the numeric data item indicates the presence of an operational sign (either as a character or an overpunch) or if there are P characters in the picture-string of the numeric data item, the software first moves the item to a temporary location. During this move, it removes the sign and fills out any P character positions with zero digits. It then uses the temporary value (which may be shorter than the original if a separate sign were removed, or longer if P character positions were filled in with zeroes) as the sending field as if it had been described as PIC X(), that is, as if its category were alphanumeric.

If the sending item is an unsigned numeric class field with no P characters in its picture-string, the software does not move the item to a temporary location.

A numeric integer data item sending field has no effect on the justification of the receiving field. If the numeric sending field is shorter than the receiving field, the software fills the receiving field with spaces.

In legal, non-numeric elementary moves, the receiving field actually controls the movement of data. All of the following items, in the receiving field, affect the move: (1) the size, (2) the presence of editing characters in its description, and (3) the presence of the JUSTIFIED RIGHT clause in its description. The JUSTIFIED clause and editing characters are mutually exclusive; therefore, the two classes are discussed separately below.

When a field that contains no editing characters or JUSTIFIED clause in its description is used as the receiving field of a non-numeric elementary MOVE statement, the statement moves the characters by starting at the left-hand end of the fields and scanning across, character-by-character to the right. If the sending item is shorter

than the receiving item, the software fills the remaining character positions with spaces.

3.6.2.1 **Edited Moves** - Alphabetic or alphanumeric fields may contain editing characters. Consider the following insertion editing characters. Alphabetic fields will accept only the B character; however, alphanumeric fields will accept all three characters.

    B -- blank insertion position

    0 -- zero insertion position

    / -- slash insertion position.

When a field that contains an insertion editing character in its picture-string is used as the receiving field of a non-numeric elementary MOVE statement, each receiving character position that corresponds to an editing character receives the insertion byte value. Figure 3-7 illustrates the use of such symbols with the statement, MOVE FIELD1 TO FIELD2. (Assume that FIELD1 was described as PIC X(7).)

	FIELD2	
FIELD1	PICTURE-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

Figure 3-7
Data Movement with Editing Symbols

Data movement always begins at the left end of the sending field, and moves only to the byte positions described as A, 9, or X in the receiving field picture-string. When the sending field is exhausted, the software supplies space characters to fill any remaining character positions (not insertion positions) in the receiving field. If the receiving field becomes exhausted before the last character is moved from the sending field, the software ignores the remaining sending field characters.

3.6.2.2 **Justified Moves** - A JUSTIFIED RIGHT clause in the data description of the receiving field causes the software to reverse its usual data movement conventions. (It starts with the right-hand characters of both fields and proceeds from right to left.) If the sending field is shorter than the receiving field, the software fills the remaining left-hand character positions with spaces. Figure 3-8 illustrates various data description situations for the statement, MOVE FIELD1 TO FIELD2, with no editing.

FIELD1		FIELD2	
PICTURE-STRING	CONTENTS	PICTURE-STRING (AND JUST CLAUSE)	CONTENTS AFTER MOVE
XXX	ABC	XX	AB
		XXXXX	ABC
		XX JUST	BC
		XXXXX JUST	ABC

Figure 3-8
Data Movement with No Editing

3.6.3 Multiple Receiving Fields

If a MOVE statement is written with more than one receiving field, it moves the same sending field value to each of the receiving fields. It has essentially the same effect as a series of separate MOVE statements that all have the same sending field. (For information on subscripted fields, see section 3.6.4.)

The receiving fields need have no relationship to each other. The software checks the legality of each one independently, and performs an independent move operation on each one.

Multiple receiving fields on MOVE statements provide a convenient way to set many fields 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 ZEROES TO EOL-FLAG, EXCEPT-FLAG, NAME-FLAG.

        MOVE 1 TO COUNT-1, CHAR-PTR, CURSOR.

3.6.4 Subscripted Moves

Any field of a MOVE statement may be subscripted and the referenced field may also be used to subscript another name in the same statement.

When more than one receiving field is named in the same MOVE statement, the order in which the software evaluates the subscripts affects the results of the move. Consider the following two situations:

```
Situation 1   MOVE FIELD1(FIELD2) TO FIELD2 FIELD3.

Situation 2   MOVE FIELD1 TO FIELD2 FIELD3(FIELD2).
```

Figure 3-9
Subscripted MOVE Statements

In situation 1, the software evaluates FIELD1(FIELD2) only once, before it moves any data to the receiving fields. In effect 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 situation 2, the software 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. In effect, it is as if the statement were replaced with the following statements:

        MOVE FIELD1 TO FIELD2.

        MOVE FIELD1 TO FIELD3(FIELD2).

3.6.5 Common Errors, MOVE Statement

A most important thing to remember when writing MOVE statements is that the compiler considers any MOVE statement that contains a group item to be a group move. It is easy to forget this fact when moving a group item to an elementary item, and the elementary item contains editing characters, or a numeric integer. These attributes of the receiving field (which would determine the action of an elementary move) have no effect on the action of a group move.

3.6.6 Format 2 - MOVE CORRESPONDING

Format 2 of the MOVE statement allows the programmer to move multiple elementary items from one group item to another, by using a single MOVE statement. When the corresponding phrase is used, selected elementary items in the sending field are moved to those elementary items in the receiving field whose data-names are identical. For example:

        01 A-GROUP   01 B-GROUP

            02 FIELD1   02 FIELD2

                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

                    OR

        MOVE CORRESPONDING FIELD1 TO FIELD2

The above examples are equivalent to the following series of MOVE statements:

        MOVE A OF FIELD1 TO A OF FIELD2

        MOVE C OF FIELD1 TO C OF FIELD 2

        MOVE E OF FIELD1 TO E OF FIELD2

3.7 THE STRING STATEMENT

The STRING statement concatenates the contents of two or more sending fields into a single field.

The statement has many forms; the simplest is equivalent, in function, to a non-numeric MOVE statement. Consider the following illustration; if the two fields are the same size, or if the sending field (FIELD1) is larger, the statement is equivalent to the statement, MOVE FIELD1 TO FIELD2.

```
STRING1 FIELD1 DELIMITED BY SIZE INTO FIELD2.
```

Figure 3-10
Sample STRING Statement

If the sending field is shorter than the receiving field, an important difference between the STRING and MOVE statements emerges: the software does not fill the receiving field with spaces. Thus, the STRING statement may leave some portion of the receiving field unchanged.

Additionally, the receiving field must be an elementary alphanumeric field with no JUSTIFIED clause or editing characters in its description. Thus, the data movement of the STRING statement always fills the receiving field from left-to-right with no editing insertions.

3.7.1 Multiple Sending Fields

An important characteristic of the STRING statement is its ability to concatenate a series of sending fields into one receiving field. Consider the following example of the STRING statement:

```
STRING FIELD1A FIELD1B FIELD1C DELIMITED BY SIZE
                INTO FIELD2.
```

Figure 3-11
Concatenation with the STRING Statement

In this sample STRING statement, FIELD1A, FIELD1B, and FIELD1C are all sending fields. The software moves them to the receiving field (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
fields, the software ignores the remaining characters of that field
and any other sending fields not yet processed. For example, if
FIELD2 became full while receiving FIELD1B, the software would ignore
the rest of FIELD1B and all of FIELD1C.

If the sending fields do not fill the receiving field, the operation
stops with the movement of the last character of the last sending item
(FIELD1C in Figure 3-11). The software does not alter the contents
nor space-fill the remaining character positions of the receiving
field.

The sending fields may be non-numeric 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 fields:

```
STRING CITY SPACE STATE ".  " ZIP

     DELIMITED BY SIZE INTO ADDRESS-LINE.
```

Figure 3-12
Literals as Sending Fields

Sending fields may also be subscripted. For example, the following
statement uses subscripts to concatenate the elements of a table
(A-TABLE) into a single field (A-FOUR). (I, of course, must be a
subscript or an index-name.)

```
STRING A-TABLE(I) A-TABLE(I+1) A-TABLE(I+2) A-TABLE(I+3)
       DELIMITED BY SIZE INTO A-FOUR.
```

Figure 3-13
Indexed Sending Fields

3.7.2 The POINTER Phrase

Although the STRING statement normally starts at the left-hand end of
the receiving field, with the POINTER phrase it is possible to start
it scanning at another point within the field. (The scanning,
however, remains left-to-right.)

```
MOVE 5 TO P.
STRING FIELD1A FIELD1B DELIMITED BY SIZE
       INTO FIELD2 WITH POINTER P.
```

Figure 3-14
Sample POINTER Phrase

When the POINTER phrase is used, the value of P determines the
starting character position in the receiving field. In Figure 3-14,
the 5 in P causes the software to move the first character of FIELD1A
into character position 5 of FIELD2 (the left-most character position
of the receiving field is character position 1) and leave positions 1
through 4 unchanged.

3-14

When the STRING operation is complete, the software leaves P pointing to one character position beyond the last character replaced in the receiving field. If FIELD1A and FIELD1B in Figure 3-14 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 12 characters long).

3.7.3 The DELIMITED BY Phrase

Although the sending fields of the STRING statement are fixed in size at compile time, they frequently contain variable-length items that are padded with spaces. For example, a 20-character city field may contain only the word MAYNARD and 13 spaces. A valuable feature of the STRING statement is that it may be used to move only the useful data from the left-hand end of the sending field. The DELIMITED BY phrase, written with a data-name or literal, instead of the word SIZE, performs this operation. (The delimiter may be a literal, a data item, a figurative constant, or the word SIZE. It may not be ALL literal since ALL literal has an indefinite length. When the phrase contains the word SIZE, the software moves each sending field, in total, until it either exhausts the sending field, or fills the receiving field.)

Consider the following example:

```
STRING CITY SPACE STATE ".  " ZIP
       DELIMITED BY SIZE INTO ADDRESS-LINE.
```

Figure 3-15
Delimiting with the Word SIZE

If CITY is a 20-character field, the result of the STRING operation shown in Figure 3-15 might look like the following:

```
AYER_____MA.   01432
                  _____16 spaces
```

A far more attractive printout can be produced by having the STRING operation produce the following:

```
AYER, MA.   01432
```

To accomplish this, use the figurative constant SPACE as a delimiter on the sending field; thus,

```
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.
```

Figure 3-16
SPACE as a Delimiter

This sample coding uses the pointer's characteristic of pointing to one character position beyond the last character replaced in the receiving field 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 field, another literal, and the 5-character ZIP field.)

The delimiter can be varied for each field within a single STRING statement by repeating the DELIMITED BY phrase after the sending field 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 this example, has no effect on the operation of the statement, but improves program readability and simplifies debugging.)

```
STRING CITY DELIMITED BY SPACE
            ", " STATE ". "
            ZIP DELIMITED BY SIZE
                INTO ADDRESS-LINE.
```

Figure 3-17
Repeating the DELIMITED BY Phrase

The sample STRING statement in Figure 3-17 cannot handle 2-word city names, such as New York, since the software would consider the space between the two words as a match for the delimiter SPACE. A longer delimiter, such as two or three spaces (non-numeric literal), can solve this problem. Only when a sequence of characters matches the delimiter will the movement stop for that data item. With a 2-byte delimiter, the same statement can be rewritten in a simpler form:

```
STRING CITY ", " STATE ". " ZIP
            DELIMITED BY "  " INTO ADDRESS-LINE.
```

Figure 3-18
Delimiting with More Than One Space Character

Since only the CITY field may contain two consecutive spaces (the entire STATE field is only two bytes long), the delimiter's search of the other fields will always be unsuccessful and the effect is the same as moving the full field (delimiting by SIZE).

Data movement under control of a data-name or literal is generally slower in execution speed than movement delimited by SIZE.

The example in Figure 3-18 illustrates a frequent source of error in the use of STRING statements to concatenate fields. The remainder of the receiving field 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 field concatenated by the STRING statement can be a field previously set to SPACES. (This sending field must be moved under control of a delimiter other than SPACE, of course.)

3.7.4 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, the programmer can tell, by simple addition, if the
receiving field is large enough to hold the sending fields. However,
if the DELIMITED BY phrase contains a literal or an identifier, or if
the pointer value is not predictable, it may be difficult to tell
whether the size of the receiving field is adequate, and an overflow
may occur.

Overflow occurs when the receiving field is full and the software is
either about to move a character from a sending field or is
considering a new sending field. Overflow may 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
field.   In this case, the software moves no data to the receiving
field 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.
```

Figure 3-19
The ON OVERFLOW Phrase

The ON OVERFLOW phrase cannot distinguish the overflow caused by a bad
initial value in pointer PNTR from the overflow caused by a receiving
field that is too short. Only a separate test, preceding the STRING
statement, can distinguish between the two.

The following examples illustrate the 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.
```

Figure 3-20
Various STRING Statements
Illustrating the Overflow Condition

The results of executing the numbered statements follow:

Table 3-2
Results of the
Preceding Sample 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

3.7.5 Subscripted Fields in STRING Statements

All data-names used in the STRING statement may be subscripted, and the pointer value may be used as a subscript.

Since the pointer value might be used as a subscript on one or more of the fields in the statement, it is important to understand the order in which the software evaluates the subscripts and exactly when it updates the pointer. (The use of the pointer as a subscript is not specified by ANS-74 COBOL. Before using it, read the note at the end of this subsection.)

The software updates the pointer after it moves the last character out of each sending field. Consider the following sample coding:

```
MOVE 1 TO P.
STRING "ABC"
       SPACE
       "DEF" DELIMITED BY SIZE
       INTO R WITH POINTER P.
```

Figure 3-21
STRING Statement with Pointer

During the movement of "ABC" into the receiving field (R), the pointer value remains at 1. After the move, the software increases the pointer value by 3 (the size of the sending field literal "ABC") and it takes on the value 4. The software then moves the figurative constant SPACE and increases the pointer value by 1 and it takes on the value 5. "DEF" is then moved and, on completion of the move, the software increases the pointer to its final value for this operation, 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.
```

Figure 3-22
Subscripting with the Pointer

If CHAR is a 1-character field in a table, the pointer increases by
one after each field has been moved and the software 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 2-character field, the pointer increases by
two after each field has been moved and the fields will move into R as
if they had been subscripted as CHAR(1), CHAR(3), CHAR(5), and
CHAR(7).

Thus, the software evaluates the subscript of a sending item once,
immediately before it considers the item as a sending item.

The software evaluates the subscript of a receiving item only once, at
the start of the STRING operation. Therefore, if the pointer is used
as a subscript on the receiving field, changes occurring to the
pointer during the execution of the STRING statement will not alter
the choice of which receiving string is altered.

Even the delimiter field can be subscripted, and it too can be
subscripted with the pointer. The software re-evaluates the delimiter
subscript once for each sending field, immediately before it compares
the delimiter to the field. Thus, by subscripting it with the pointer
value, the delimiter can be changed for each sending field. This has
the peculiar effect of choosing the next sending field's delimiter
based on the position, in the receiving field, into which its first
character will fall.   For example, consider the following sample
coding:

```
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.
```

Figure 3-23
Subscripting the Delimiter

The following table shows the value that will arrive in the receiving field (R) from the three "ABC" literals if DTABLE contains the values shown in the left-hand column:

Table 3-3
Results of the
Preceding Sample Statements

DTABLE Value	R Value
ABCDEFG	(Unchanged)
BCDEFGH	AABABC
CDEFGHI	ABABCABC
CCCCCCCC	ABABAB

NOTE

The rules in this section, concerning subscripts in the STRING statement, are rules that are not specified by 1974 American National Standard COBOL. Dependence on these rules, particularly those involving the use of the pointer field as a subscript, may produce programs that will not perform the same way on other COBOL compilers.

If the pointer field is not used as a subscript on any of the fields in the statement, the point at which the software evaluates the subscripts is immaterial to the execution of the statement. Thus, by avoiding the use of the pointer as a subscript, uniform results can be expected from all COBOL compilers that adhere to 1974 ANS COBOL.

3.7.6 Common Errors, STRING Statement

The most common errors made 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 fill of the receiving field when it is desirable.

3.8 THE UNSTRING STATEMENT

The UNSTRING statement disperses the contents of a single sending
field into multiple receiving fields.

The statement has many forms; the simplest is equivalent in function
to a non-numeric MOVE statement. Consider the following illustration;
the sample statement is equivalent to MOVE FIELD1 TO FIELD2,
regardless of the relative sizes of the two fields.

```
UNSTRING FIELD1 INTO FIELD2.
```

Figure 3-24
Sample UNSTRING Statement

The sending field (FIELD1) may be either a group item or an
alphanumeric, or alphanumeric edited elementary item. The receiving
field (FIELD2) may be alphabetic, alphanumeric, or numeric, but it
cannot specify any type of editing.

If the receiving field is numeric, it must be DISPLAY usage.   The
picture-string of a numeric receiving field may contain any of the
legal numeric description characters except for P and, of course, the
editing characters. The UNSTRING statement moves the sending field to
numeric receiving fields as if the sending field had been described as
an unsigned integer;   further, it automatically truncates or zero
fills as required.

If the receiving field is not numeric, the software follows the rules
for elementary non-numeric MOVE statements. It left-justifies the
data in the receiving field, truncating or space-filling as required.
(If the data-description of the receiving field contains a JUSTIFIED
clause, the software right-justifies the data, truncating   or
space-filling to the left as required.)

3.8.1 Multiple Receiving Fields

An important characteristic of the UNSTRING statement is its ability
to disperse one sending field into several receiving fields. Consider
the following example of the UNSTRING statement written with multiple
receiving fields:

```
UNSTRING FIELD1 INTO
        FIELD2A FIELD2B FIELD2C.
```

Figure 3-25
Multiple Receiving Fields

In this sample statement, FIELD1 is the sending field.   The software
performs the UNSTRING operation by scanning across FIELD1 from left to
right. When the number of characters scanned is equal to the number
of characters in the receiving field, the software moves the scanned
characters into the receiving field and begins scanning the next group
of characters for the next receiving field.

Assume that each of the receiving fields in the preceding illustration (FIELD2A, FIELD2B, and FIELD2C) is five characters long, and that FIELD1 is 15 characters long. The size of FIELD2A determines the number of characters for the first move. The software scans across FIELD1 until the number of characters scanned equals the size of FIELD2A (5). It then moves those first five characters to FIELD2A, and sets the scanner to the next (sixth) character position in FIELD1. The size of FIELD2B determines the size of the next move. The software begins this move by scanning across FIELD1 from character position six, until the number of scanned characters equals the size of FIELD2B (5). It then moves the sixth through the tenth characters to FIELD2B, and sets the scanner to the next (eleventh) character position in FIELD1. FIELD2C determines the size of the last move (for this example) and causes characters 11 through 15 of FIELD1 to be moved into FIELD2C, thus terminating this UNSTRING operation.

Each data movement acts as an individual MOVE statement, the sending field of which is an alphanumeric field equal in size to the receiving field. If the receiving field is numeric, the move operation will convert the data to the numeric form. For example, consider what would happen if the fields under discussion had the data descriptions and were manipulating the values shown in the following table:

Table 3-4
Values Moved Into the Receiving Fields
Based on the Value in the Sending Field

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 field and, therefore, the software simply conducts an elementary non-numeric move with the first five characters.

FIELD2B, however, has a leading separate sign that is not included in its size. Thus, the software 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 field should supply five numeric digits; but, since the sending field is alphanumeric, the software 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 (or the ASCII character, {). (There is no simple way to have UNSTRING recognize a sign character or a decimal point in the sending field.)

If the sending field is shorter than the sum of the sizes of the receiving fields, the software ignores the remaining receiving fields. If it reaches the end of the sending field before it reaches the end of one of the receiving fields, the software moves the scanned characters into that receiving field. It left-justifies and fills the remaining character positions with spaces for alphanumeric data, or decimal point aligns and zero fills the remaining character positions

for numeric data. Consider the following examples of a sending field
that is too short. (The statement is UNSTRING FIELD1 INTO FIELD2A
FIELD2B. FIELD2A is a 3-character alphanumeric field, and receives
the first three characters of FIELD1 (ABC) in every operation.
FIELD2B, however, runs out of characters every time before filling.
Since FIELD2A always contains the characters ABC, it is not shown.)

Table 3-5
Handling a Sending Field that is Too Short

FIELD1 PIC X(6) VALUE IS:	FIELD2B PICTURE IS:	FIELD2B Value after UNSTRING Operation
ABCDEF	XXXXX	DEF
ABC246	S99999	0024F
	S9V999	600
	S9999	+0246
	LEADING SEPARATE	

3.8.2 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 field. The DELIMITED BY
phrase supplies the delimiter characters.

UNSTRING delimiters are quite flexible; they can be literals,
figurative constants (including ALL literal), or identifiers
(identifiers may even be subscripted data-names). This sub-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 under Subscripted Fields in UNSTRING Statements.

Consider the following sample UNSTRING statement; it uses the
figurative constant, SPACE, as a delimiter:

```
UNSTRING FIELD1 DELIMITED BY SPACE INTO FIELD2.
```

Figure 3-26
Delimiting with a Space Character

In this example, the software scans the sending field (FIELD1),
searching for a space character. If it encounters a space, it moves
all of the scanned (non-space) characters that precede that space to
the receiving field (FIELD2). If it finds no space character, it
moves the entire sending field. When it has determined the size of
the sending field, the software moves the contents of that field
following the rules for the MOVE Statement, truncating or zero filling
as required.

The following table shows the results of an UNSTRING operation that
delimits with a literal asterisk (UNSTRING FIELD1 DELIMITED BY "*"
INTO FIELD2).

Table 3-6
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	ΔΔΔ
*ABCDE	XXX	ΔΔΔ
A*****	XXX JUSTIFIED	ΔΔA
246***	S9999	024F
12345*	S9999 SEPARATE TRAILING	2345+
2468**	S999V9 SEPARATE LEADING	+4680
*246**	9999	0000

If the delimiter matches the first character in the sending field, the software considers the size of the sending field to be zero. The movement operation still takes place, however, and fills the receiving field with spaces or zeroes depending on its class.

A delimiter may also be applied to an UNSTRING statement that has multiple receiving fields:

```
UNSTRING FIELD1 DELIMITED BY SPACE
      INTO FIELD2A FIELD2B.
```

Figure 3-27
Delimiting with Multiple Receiving Fields

The sample instruction in Figure 3-27 causes the software to scan 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 it finds a match, it moves all of the characters that lie between the character that first matched the delimiter and the character that matched on the second scan, and sets the scanner to the next character position to the right of the character that matched. (The DELIMITED BY phrase could handle additional receiving fields in the same manner as it handled FIELD2B.)

The following table shows the results of an UNSTRING operation that applies a delimiter to multiple receiving fields (UNSTRING FIELD1 DELIMITED BY "*" INTO FIELD2A FIELD2B).

Table 3-7
Results of Delimiting
Multiple Receiving Fields

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

The last two examples illustrate the limitations of a single character delimiter. Accordingly, the delimiter may be longer than one character and it may be preceded by the word ALL.

The following table shows the results of an UNSTRING operation that uses a 2-character delimiter (UNSTRING FIELD1 DELIMITED BY "**" INTO FIELD2A FIELD2B):

Table 3-8
Results of Delimiting
with Two Asterisks

FIELD1 PIC X(8) VALUE IS:	VALUES AFTER UNSTRING OPERATION	
	FIELD2A PIC XXX	FIELD2B PIC XXX JUSTIFIED
ABC**DEF	ABC	DEF
A*B*C*D*	A*B	ΔΔΔ
AB***C*D	ABΔ	C*D
AB**C*D*	ABΔ	*D*
AB**CD**	ABΔ	ΔCD
AB***CD*	ABΔ	CD*
AB*****CD	ABΔ	ΔΔΔ

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 software 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.

The following table illustrates the results of an UNSTRING operation that uses an ALL delimiter (UNSTRING FIELD1 DELIMITED BY ALL "*" INTO FIELD2A FIELD2B).

Table 3-9
Results of Delimiting
with ALL Asterisks

| FIELD1 PIC X(8) VALUE IS: | VALUES AFTER UNSTRING OPERATION | |
	FIELD2A PIC XXX	FIELD2B PIC XXX JUSTIFIED
ABC*DEF*	ABC	DEF
ABC**DEF	ABC	DEF
A******F	AΔΔ	ΔΔF
A*F*****	AΔΔ	ΔΔF
A*CDEFG	AΔΔ	EFG

The next table illustrates the results of an UNSTRING operation that combines ALL with a 2-character delimiter (UNSTRING FIELD1 DELIMITED BY ALL "**" INTO FIELD2A FIELD2B).

Table 3-10
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*

In addition to unchangeable delimiters, such as literals and figurative constants, delimiters may be designated by identifiers. Identifiers (which may even be subscripted data-names) permit variable delimiting. Consider the following sample statement:

```
UNSTRING FIELD1 DELIMITED BY DEL1
         INTO FIELD2A FIELD2B.
```

Figure 3-28
Delimiting with an Identifier

The data-name, DEL1, must be alphanumeric. It may be a group or elementary item, and it may be edited. (Since the delimiter is not a receiving field, any editing characters will not effect its use, other than contributing to the size of the item.)

If the delimiter contains a subscript, the subscript may be varied as a side effect of the UNSTRING operation. The evaluation of subscripts is discussed later in this section.

3.8.2.1 **Multiple Delimiters** - The UNSTRING statement has the ability to scan a sending field, searching for a match from a list of delimiters. This list may 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 separates a sending field into three receiving fields. The sending field consists of three strings separated by the following: (1) any number of spaces, or (2) a comma followed by a single space, or (3) a single comma, or (4) a tab character, or (5) a carriage return character. (The ", " must precede the "," in the list if it is ever to be recognized.)

```
UNSTRING FIELD1 DELIMITED BY
         ALL SPACE OR
         ", " OR
         "," OR
         TAB OR
         CR
         INTO FIELD2A FIELD2B FIELD2C.
```

Figure 3-29
Multiple Delimiters

The following table illustrates the potential of this statement. The tab (represented by the letter t) and carriage return (represented by the letter r) characters represent single character fields containing the ASCII horizontal tab and carriage return characters.

Table 3-11
Results of the Multiple Delimiters
Shown in Figure 3-29

FIELD1 PIC X(12)	FIELD2A PIC XXX	FIELD2B PIC 9999	FIELD2C PIC XXX
A,0,Cr	AΔΔ	0000	CΔΔ
At456,ΔE	AΔΔ	0456	EΔΔ
AΔΔΔ 3ΔΔΔ9	AΔΔ	0003	9ΔΔ
AttBr	AΔΔ	0000	BΔΔ
A,,C	AΔΔ	0000	CΔΔ
ABCD,Δ4321,Z	ABC	4321	ZΔΔ

t--tab character, r--carriage return character

3.8.3 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 field may vary widely (from zero to the full length of the field) and some programs may require knowledge of this length. For example, if it exceeds the size of the receiving field (which is fixed in size) some data may be truncated and the program's logic may require this information.

To use the phrase, simply follow the receiving field 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.
```

Figure 3-30
The COUNT Phrase

In this sample statement, the software will count the number of characters between the left-hand end of FIELD1 and the first asterisk in FIELD1 and place that value into COUNT2A; thus, COUNT2A contains the size of the first sending string. The software does not include the delimiter in the count (as it is not a part of the string).

The software then counts the number of characters in the second sending field and places that value into COUNT2B.

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 field is shorter than the value placed in the count field, the software truncates the sending string. (If the number of integer positions in a numeric field is smaller than the value placed into the count field, high-order numeric digits have been lost.)

If the software finds a delimiter match on the first character it examines, it places a zero in the count field.

The count field must be described as a numeric integer, either COMP or DISPLAY usage, with no editing symbols nor the character P in its picture-string. The software moves the count value into the count field according to the rules for an elementary numeric MOVE statement

The COUNT phrase may be used only in conjunction with the DELIMITED BY phrase.

3.8.4 The DELIMITER Phrase

The DELIMITER phrase causes the actual character or characters that delimited the sending field to be stored in a user-supplied data area. This phrase is most useful when: (1) the statement contains a delimiter list, (2) any one of the items in the list might have delimited the field, and (3) program logic flow depends on which one found a match. In fact, the DELIMITER and COUNT phrases could be used together and program logic flow could depend on both the size of the sending string and the delimiter character that terminated it.

To use the DELIMITER phrase, simply follow the receiving field name with the words DELIMITER IN and an identifier. (The software 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.
```

Figure 3-31
The DELIMITER Phrase

After moving the first sending string to FIELD2A, the software takes the character (or characters) that delimited that string and places it in DELIMA. DELIMA, then, contains a comma, or a tab, or a carriage return, or any number of spaces. Since the delimiter string is moved under the rules of the elementary non-numeric MOVE statement, the software truncates or space fills with left or right justification (depending on its data description).

The software 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 field (rather than a match on a delimiter) the delimiter string is of zero length. This causes the DELIMITER item to be 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 may contain editing characters and it may even be a group item.

When the DELIMITER and COUNT phrases are used together, they must appear in the correct order (DELIMITER phrase preceding the COUNT phrase). Both of the data items named in these phrases may be subscripted or indexed. If they are subscripted, the subscript may be varied as a side effect of the UNSTRING operation. (The evaluation of subscripts is discussed in section 3.8.8.)

3.8.5 The POINTER Phrase

Although the UNSTRING statement normally starts at the left-hand end of the sending field, the POINTER phrase permits the user to select a character position in the sending field for the software to begin scanning. (The scanning, however, remains left-to-right.)

When a sending field is to be dispersed into multiple receiving fields, it often happens that the choice of delimiters, the size of subsequent receiving fields, etc. depend on the value in the first sending string or the character that delimited that string. Thus, the program may need to move the first field, hold its place in the sending field, 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 field. When the first string has been moved to a receiving item, the software updates the pointer data item with a new position (one character beyond the delimiter that caused the interruption) to begin the next scanning operation. The program may then examine the new position, the receiving field, the delimiter value, the sending string size, and resume the scanning operation by executing another UNSTRING statement with the same sending field and pointer data item. Thus, the UNSTRING statement can move one sending string at a time, with the form of each move being dependent on the context of the preceding string of data.

The POINTER phrase must follow the last receiving item in the statement. Consider the following two UNSTRING statements with their accompanying POINTER phrases and tests:

```
MOVE 1 TO P.
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.
```

Figure 3-32
The POINTER Phrase

PNTR contains the current position of the scanner in the sending field. The second UNSTRING statement uses PNTR to begin scanning the additional sending strings in FIELD1.

Since the software considers the left-most character to be character position one, the value returned by PNTR may be used to examine the next character. To do this, simply use PNTR as a subscript on the sending field (providing that the sending field is also described as a table of characters). For example, consider the following sample coding:

```
01 FIELD1.
   02 FIELD1-CHAR OCCURS 40 TIMES.

   ...

   UNSTRING FIELD1
       ...
       WITH POINTER PNTR.
   IF FIELD1-CHAR(PNTR) = "X" ...
```

Figure 3-33
Examining the Next Character
By Using the Pointer Data
Item as a Subscript

Another way to examine the next character of the sending field is to use the UNSTRING statement to move it to a 1-character receiving field. Consider the following sample coding:

```
   UNSTRING FIELD1
       ...
       WITH POINTER PNTR.
   UNSTRING FIELD1 INTO CHAR1 WITH POINTER PNTR.
   SUBTRACT 1 FROM PNTR.
   IF CHAR1 = "X" ...
```

Figure 3-34
Examining the Next Character
By Placing It Into a 1-Character Field

The program must decrement PNTR in order for this case to work like the one illustrated in Figure 3-33, since the second UNSTRING statement will increment the pointer value by 1.

The program must initialize the POINTER phrase data item before the UNSTRING statement uses it. The software will terminate the UNSTRING operation if the initial value of the pointer is less than one or greater than the length of the sending field. (A pointer value that is less than one or greater than the length of the sending field causes an overflow condition. Overflow conditions are discussed in section 3.8.7.)

The POINTER and TALLYING phrases may be used together in the same UNSTRING statement; but, when both are used, the POINTER phrase must precede the TALLYING phrase.

3.8.6 The TALLYING Phrase

The TALLYING phrase counts the number of receiving fields that received data from the sending field.

When an UNSTRING statement contains several receiving fields, the possibility exists that there may not always be as many sending strings as there are receiving fields. The TALLYING phrase provides a convenient method for keeping a count of how many fields were acted upon.

```
MOVE 0 TO RCOUNT.
UNSTRING FIELD1 DELIMITED BY "," OR ALL SPACE
        INTO FIELD2A
             FIELD2B
             FIELD2C
             FIELD2D
             FIELD2E
             TALLYING IN RCOUNT.
```

Figure 3-35
The TALLYING Phrase

If the software has moved only three sending strings when it reaches the end of FIELD1, it adds 3 to RCOUNT. The first three fields (FIELD2A, FIELD2B, and FIELD2C) contain data from the operation, and the last two (FIELD2D and FIELD2E) do not.

The TALLYING data item always contains the sum of its initial contents plus the number of sending strings acted upon by the UNSTRING command just executed. Thus, the programmer may want to initialize the tally count before each use.

When used in the same statement with a POINTER phrase, the TALLYING phrase must follow the POINTER phrase and both phrases must follow all of the field names, the DELIMITER and COUNT phrases. The data items for both phrases must contain numeric integers, that is, be without editing characters or the letter P in their picture-strings; both data items may be either COMP or DISPLAY usage. They may be signed or unsigned and, if they are DISPLAY usage, they may contain any desired sign option.

The data items for both phrases may be subscripted or indexed, or they may be used as subscripts on other fields in the statement. (The evaluation of subscripts is discussed in section 3.8.8.) A convenient use of the TALLYING phrase data item is as a subscript of a receiving field. Consider the following sample coding, which causes program control to execute the UNSTRING statement repeatedly until it exhausts the sending field.

```
        MOVE 1 TO PNTR, TLY.
PAR1.   UNSTRING FIELD1 DELIMITED BY "," OR CR
             INTO FIELD2(TLY)
             DELIMITER IN DEL2
             WITH POINTER PNTR
             TALLYING IN TLY.
        IF DEL2 = "," GO TO PAR1.
```

Figure 3-36
The POINTER and TALLYING Phrases
Used Together

This sample coding causes program control to loop through the UNSTRING statement, using the pointer, PNTR, to scan across FIELD1 with successive executions. Each comma isolates a sending string until control reaches either a carriage return character or the end of FIELD1. If it reaches the end of the field without encountering a carriage return character, the software places a space into the delimiter field, DEL2, and control falls through the IF statement and out of the loop.

Since the TALLYING data item, TLY, is increased by 1 after each data movement, it serves as a subscript on the receiving field. In effect this causes the software to unpack the value in FIELD1 into an array of fixed-size fields. Further, an array of COUNT data items can be supplied and loaded by the UNSTRING/TALLYING statement by adding the following phrase to the coding in Figure 3-36:

```
COUNT IN C(TLY)
```

Figure 3-37
Subscripting the COUNT Phrase
With the TALLYING Data Item

The TALLYING data item, in the above example, is one greater than the number of receiving fields acted upon by the UNSTRING operation. This is 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.

3.8.7   The OVERFLOW Phrase

The OVERFLOW phrase detects the overflow condition and provides an imperative statement to be executed when it detects the condition. An overflow condition exists when either of the following two situations occurs:

1. The UNSTRING statement is about to be executed and its pointer data item contains a value of less than one or greater than the size of the sending field. When it detects this situation, the software executes the OVERFLOW phrase before it moves any data. Thus, the values of all of the receiving fields remain unchanged.

2. The UNSTRING statement has filled all of the receiving fields and data still remains in the sending field that has not been matched as a delimiter or included in a sending string. When it detects this situation, the software executes the OVERFLOW phrase after it has executed the UNSTRING statement. Thus, the values of all of the receiving fields are updated, but some data has not been moved.

If the UNSTRING operation causes the scanner to move off the end of the sending field (thus exhausting it), the software will not execute the OVERFLOW phrase.

Consider the following set of instructions, which cause program control to execute the UNSTRING statement repeatedly until it exhausts the sending field. The TALLYING data item is a subscript indexing the receiving field. (Compare this loop with the one in Figure 3-36, 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.
```

Figure 3-38
Using the OVERFLOW Phrase

NOTE

The overflow condition also occurs if
the value of a pointer data item lies
outside the sending field at the start
of execution of the UNSTRING statement.
(The pointer value must not be less than
1, nor greater than the length of the
sending field.) 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 fields remain unchanged.

3.8.8 Subscripted Fields in UNSTRING Statements

Since the flexibility of the UNSTRING statement is enhanced by
subscripting and indexing and particularly by subscripting with other
fields within the statement (such as subscripting the receiving field
with the TALLYING data item as discussed above), it is important to
understand how often and exactly when the software evaluates these
subscripts and indexes. This sub-section discusses the frequency and
times of subscript evaluation.

The software evaluates subscripts and indexes on the following items
only once, at the initiation of the UNSTRING statement; thus, any
change in subscript values during the execution of the statement has
no effect on these fields:

1. Sending field,

2. POINTER data item,

3. TALLYING data item.

The software evaluates subscripts and indexes on the following items
immediately before it moves data into the item. It moves the data to
these items in the order in which they are listed in the statement
(which is the same order as below):

1. Receiving field,

2. DELIMITER data item,

3. COUNT data item.

The software evaluates any subscripts and indexes on the data-names in the DELIMITED BY phrase (delimiters) immediately before it scans each sending string looking for a delimiter match. Thus, it re-evaluates these data-names once for each receiving field in the statement.

If any of the following items are used as subscripts on any receiving fields, the programmer must be aware of the point at which these items are updated:

- POINTER data-item,

- TALLYING data-item,

- COUNT data-item,

- Another receiving field.

Figure 3-39 illustrates, with a flow chart, the sequence of evaluation operations:
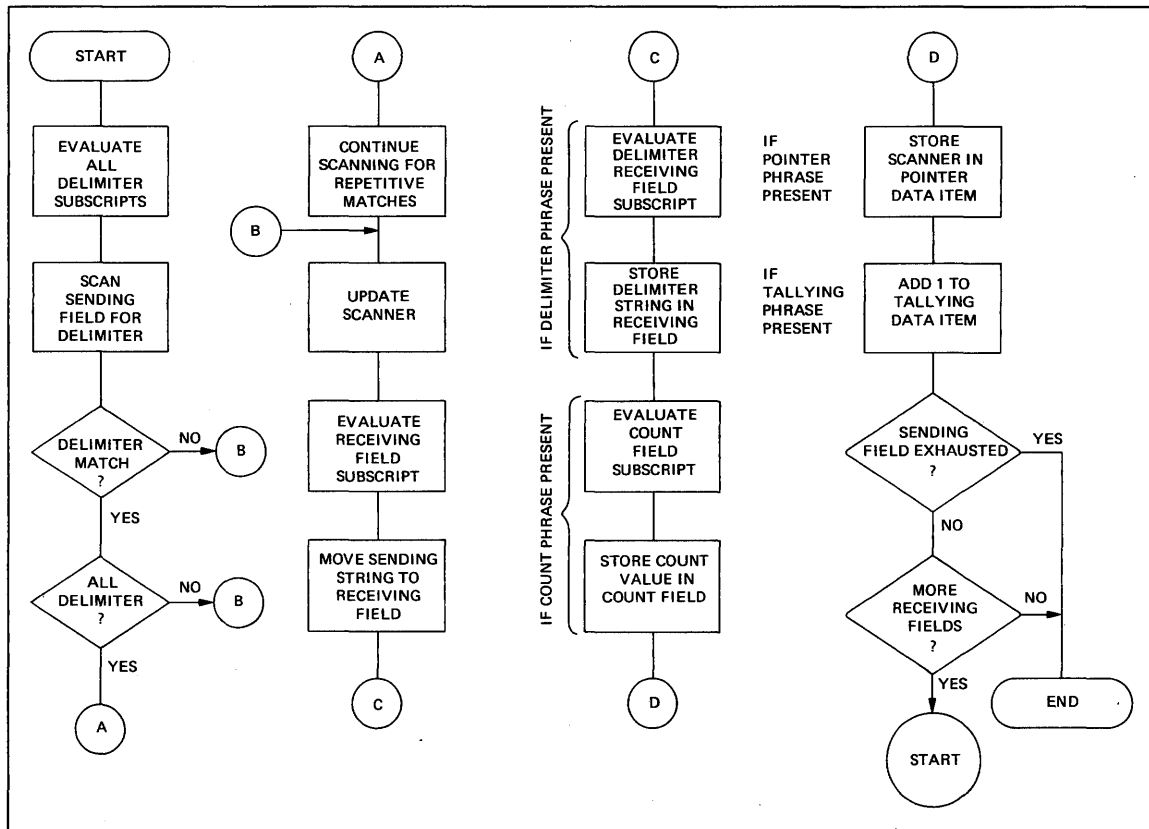
Figure 3-39
Sequence of Subscript Evaluation

NOTE

The rules in this section concerning the
exact point at which the software
evaluates the identifiers in the
DELIMITED BY phrase and the point at
which it updates the POINTER and
TALLYING data items, are rules that are
specified by 1974 American National
Standard COBOL, as opposed to the STRING
statement where these are not so
specified.

3.8.9 Common Errors, UNSTRING Statement

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);

- Leaving out the word INTO or writing it as TO;

- Repeating the word INTO where it is not needed; thus:

```
UNSTRING FIELD1 DELIMITED BY SPACE OR TAB
         INTO FIELD2A DELIMITER IN DELIMA
         INTO FIELD2B DELIMITER IN DELIMB
         INTO FIELD2C DELIMITER IN DELIMC.
```

Figure 3-40
Erroneously Repeating the Word INTO

- Writing the POINTER and TALLYING phrases in the wrong order (POINTER must precede TALLYING).

3.9 THE INSPECT STATEMENT

The INSPECT statement examines the character positions in a field and counts or replaces certain characters (or groups of characters) in that field.

Like the STRING and UNSTRING operations, INSPECT operations scan across the field from left to right; further, like those two statements, the INSPECT statement features a phrase which allows it to begin or terminate the scanning operation with a delimiter match. (Thus, the operation can begin within the field instead of at the left-hand end, or it may begin at the left-hand end and terminate within the field.)

The TALLYING operation (which counts certain characters in the field) and the REPLACING operation (which replaces certain characters in the field) are quite versatile and may be applied to all of the characters in the delimited area of the field being inspected, or they may be applied only to those characters that match a given character string

under stated conditions.   Consider the following sample statements, which both cause a scan of the complete field:

```
INSPECT FIELD1 TALLYING TLY FOR ALL "B".
```

Figure 3-41
Sample INSPECT...TALLYING Statement

This statement scans FIELD1 looking for the character B.   Each time it finds a B, it increments TLY by 1.

```
INSPECT FIELD1 REPLACING ALL SPACE BY ZERO.
```

Figure 3-42
Sample INSPECT...REPLACING Statement

This statement scans FIELD1 looking for space characters.   Wherever it finds a space character, it replaces it with zero.

One INSPECT statement can contain both a TALLYING phrase and a REPLACING phrase.   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 and, in fact, the software compiles such a statement into two distinct INSPECT statements.   (To simplify debugging, therefore, it is best to initially write the two phrases in separate INSPECT statements.)

3.9.1 The BEFORE/AFTER Phrase

The BEFORE/AFTER phrase acts as a delimiter and (possibly) restricts the area of the field being inspected.

The following sample statement would count only the zeroes that precede the percent sign (%) in FIELD1.

```
INSPECT FIELD1 TALLYING TLY
        FOR ALL ZEROES BEFORE "%".
```

Figure 3-43
Sample INSPECT...BEFORE Statement

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 may be alphabetic, alphanumeric, or numeric, and, it may 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 3.9.2.)

- If the delimiter is a literal, it must be non-numeric.

The software repeatedly compares the delimiter characters against an equal number of characters in the field being inspected. If none of the characters matches the delimiter, or if insufficient characters remain in the field for a full comparison (at the right-hand end), the software considers the comparison to be unequal.

The examples of the INSPECT statement in Figure 3-44, illustrate the way the delimiter character finds a match in the field being inspected. (The portion of the field 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.)

INSTRUCTION	FIELD1 VALUE
INSPECT FIELD1...BEFORE "E". INSPECT FIELD1...AFTER "E".	<u>ABCD</u>~~EFGHI~~ ~~ABCDE~~<u>FGHI</u>
INSPECT FIELD1...BEFORE "K". INSPECT FIELD1...AFTER "K".	<u>ABCDEFGHI</u> ~~ABCDEFGHI~~
INSPECT FIELD1...BEFORE "AB". INSPECT FIELD1...AFTER "AB".	~~ABCDEFGHI~~ ~~AB~~<u>CDEFGHI</u>
INSPECT FIELD1...BEFORE "HI". INSPECT FIELD1...AFTER "HI".	<u>ABCDEFG</u>~~HI~~ ~~ABCDEFGHI~~
INSPECT FIELD1...BEFORE "IΔ". INSPECT FIELD1...AFTER "IΔ".	<u>ABCDEFGHI</u> ~~ABCDEFGHI~~
The ellipsis represents the position of the TALLYING or REPLACING phrase.	

Figure 3-44
Matching the Delimiter Characters
to the Characters in a Field

The software scans the field 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. The importance of the separate scan is discussed further in Section 3.9.3.

3.9.2 Implicit Redefinition

The software requires that certain fields referred to by the INSPECT statement be alphanumeric fields. If one of these fields was described as another data class, the compiler redefines that field so the INSPECT statement can handle it as a simple alphanumeric string. This implicit redefinition is conducted as follows:

- If the field was described as alphabetic, alphanumeric edited, or unsigned numeric, the compiler simply redefines it as alphanumeric. This is a compile-time operation; no data movement occurs at object-time.

- If the field was described as signed numeric, the compiler first removes the sign and then redefines the field as alphanumeric. If the sign is a separate character, the compiler ignores that character, essentially shortening the

field, 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 3-12 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 3-12
Original, Altered, and Restored 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)

3.9.3 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 field. This section describes this method.

However, before discussing how the inspection operation is conducted, let's analyze the INSPECT statement itself:
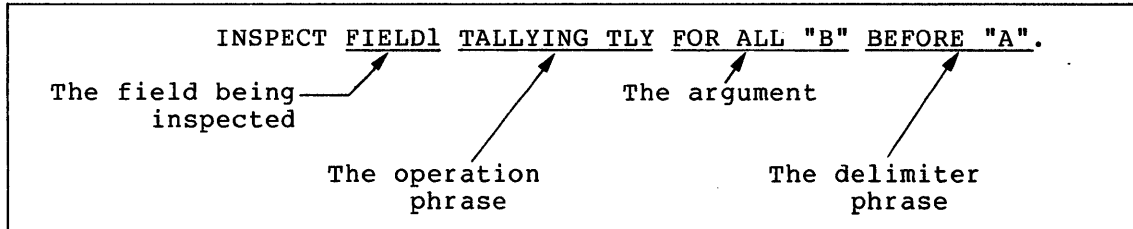
```
            INSPECT FIELD1 TALLYING TLY FOR ALL "B" BEFORE "A".
  The field being                             The argument
      inspected

            The operation                    The delimiter
              phrase                            phrase
```

Figure 3-45
Sample INSPECT Statement

The format of the INSPECT statement requires that a field be named which is to be inspected (FIELD1 above); the field name must be followed by an operation phrase (TALLYING TLY above); and, that phrase must be followed by one or more identifiers or literals ("B" above). These identifiers or literals comprise the "arguments" (items to be compared to the field being inspected). More than one argument makes up the "argument list".

- **TALLYING Arguments**

  Each argument in an argument list can have other fields associated with it. Thus, each argument that is used in a TALLYING operation must have a tally counter (TLY above) associated with it. The software increments the tally counter each time it matches the argument with a character or group of characters in the field being inspected.

- **REPLACING Arguments**

```
        INSPECT FIELD1 REPLACING ALL "0" BY "$".

                        replacing argument
```
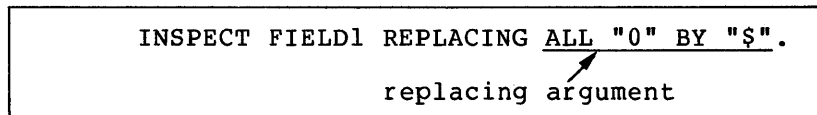
Figure 3-46
Sample REPLACING Argument

  Each argument in an argument list that is used in a REPLACING operation must have a replacement item ($ above) associated with it. The software uses the replacement item to replace each string of characters in the field that matches the argument.

Each argument in an argument list (that is used with either a TALLYING or REPLACING operation) may have a delimiter field (BEFORE/AFTER phrase) associated with it. If the delimiter field is not present, the software applies the argument to the entire field. If the delimiter field is present, the software applies the argument only to that portion of the field specified by the BEFORE/AFTER phrase.

3.9.3.1 **Setting the Scanner** - The INSPECT operation begins by setting the scanner to the leftmost character position of the field 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.

3.9.3.2 **Active/Inactive Arguments** - When an argument has a BEFORE/AFTER phrase associated with it, that argument has a delimiter and may 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 software retains the character position beyond the matched character string; then, when the scanner reaches or passes this position, the argument becomes active.

```
INSPECT FIELD1 TALLYING TLY
        FOR ALL "B" AFTER "X".
```

Figure 3-47
Sample AFTER Delimiter Phrase

If FIELD1 in Figure 3-47 has a value of "ABABXZBA", the argument B remains inactive until the scanner finds a match for the delimiter X. Thus, argument B remains inactive while the software scans character positions 1 through 5. At character position 5, the 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 and this causes TLY to be incremented by 1.

The examples in Figure 3-48 illustrate other situations where the arguments and/or the delimiters are longer than one character. (Consider the sample statement to be an INSPECT...TALLYING statement that is scanning FIELD1, tallying in TLY, and looking for the arguments and delimiters in the left-hand column. Assume that TLY is initialized to 0.)

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

Figure 3-48
Where Arguments Become Active in a Field

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. Additionally, regardless of the presence of the BEFORE delimiter, an argument becomes inactive when the scanner approaches the right-hand end of the field 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 field so it becomes inactive.)

Since the BEFORE/AFTER delimiters are found on a separate scan of the field, the software 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.

3.9.3.3 **Finding an Argument Match** - The software 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 software compares it to the character at the position of the scanner. If the active argument does not find a match, the software 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 right-hand end of the field.

When an active argument does find a match, the software 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 may contain additional conditions, which are described later in this section; this discussion, however, assumes that the argument match is allowed to take place and that inspection is allowed to continue following the match.)

The software 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.

3.9.4 Subscripted Fields in INSPECT Statements

Any identifier named in an INSPECT statement may be subscripted or indexed.

The software 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 illustration:

```
MOVE 1 TO TLY.
INSPECT FIELD1 TALLYING TLY
        FOR ALL X(TLY).
```

Figure 3-49
Sample Subscripted Argument

In this sample statement, the software evaluates the address of X(TLY) only once, before it begins inspecting the field; hence, it will evaluate X(TLY) as X(1). The alteration of TLY by the action of inspecting and tallying has no effect on the choice of the X operand. (X(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 field 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.

3.9.5 The TALLYING Phrase

An INSPECT statement that contains a TALLYING phrase counts the occurrence of various character strings under certain stated conditions. It keeps the count in a user-designated field called, here, a tally counter.

3.9.5.1 The Tally Counter - The identifier that follows the word TALLYING designates the tally counter. The identifier may be subscripted or indexed. The data item must be a numeric integer with no editing or P characters; it may be COMP or DISPLAY usage, and it may be signed (separate or overpunched).

Each time the tally argument matches the delimited string being inspected, the software adds 1 to the tally counter.

The programmer can initialize the tally counter to any numeric value. (The INSPECT statement does not initialize it.)

3.9.5.2 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 following figure shows the format of the tally argument:

```
( (ALL      )      (identifier))
{ {LEADING}        {literal    }}
( (CHARACTERS                   )
```
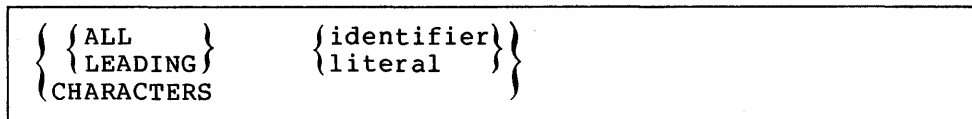
Figure 3-50
Format of the Tally Argument

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 ",".
```

Figure 3-51
CHARACTERS Form of the Tally Argument

The ALL and LEADING forms of the tally argument specify a particular character string, which may be represented by either a literal or an identifier. The tally argument character string may be any length; however, each character of the argument must match a character in the delimited string before the software considers the argument matched.

- A literal character string must be either non-numeric or a figurative constant (other than ALL literal). A figurative constant, such as SPACE, ZERO, etc., represents a single character and can be written as " ", or "0", etc., with the same effect.

- An identifier must be an elementary item of DISPLAY usage. It may be any data class. However, if it is other than alphanumeric, the software performs an implicit redefinition of the item. (This redefinition is identical to the BEFORE/AFTER delimiter redefinition discussed earlier in Section 3.9.1.)

NON-NUMERIC CHARACTER HANDLING

The words ALL and LEADING supply conditions that further delimit the inspection operation.

- The word 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."

- The word LEADING specifies that only adjacent matches of the TALLY argument at the left-hand end of the delimited character string be counted. At the first failure to match the tally argument, the software terminates counting and causes the argument to become inactive. Consider the examples in Figure 3-52. (The sample statement is an INSPECT...TALLYING statement, scanning FIELD1, tallying in TLY, and looking for the arguments and delimiters in the left-hand column. Assume that the program initializes TLY to 0.)

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

Figure 3-52
Results of Counting with the
LEADING Condition

3.9.5.3 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 sample statements show INSPECT statements with argument lists. The text following each one tells how that list will be processed.

```
INSPECT FIELD1 TALLYING T FOR
        ALL ","
        ALL "."
        ALL ";".
```

Figure 3-53
Argument List Adding Into
One Tally Counter

3-45

These three tally arguments have the same tally counter, T, and are
active over the entire field 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 ";".
```

Figure 3-54
Argument List Adding Into
Separate Tally Counters

Each tally argument in this statement has its own tally counter, and
is active over the entire field being inspected. Thus, the action of
this statement is to add 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 ";".
```

Figure 3-55
Argument List (with Delimiters) Adding
into Separate Tally Counters

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 field 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.

Figure 3-56 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.

FIELD1 VALUE	CONTENTS OF TALLY COUNTERS AFTER SCAN		
	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

Figure 3-56
Results of the Scan in Figure 3-55

The BEFORE/AFTER phrase applies only to the argument that precedes it,
and delimits the field for that argument only. Each BEFORE/AFTER
phrase causes a separate scan of the field to determine the limits of
the field for its corresponding argument.

3-46

3.9.5.4 Interference in Tally Argument Lists - When several tally arguments contain one or more identical characters that are active at the same time, they may interfere with each other (i.e., when one of the arguments finds a match, the scanner is stepped past the matching character(s) which prevents those character(s) from being considered for any other match).

The example in Figure 3-57 illustrates two identical tally arguments that 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".
```

Figure 3-57
Two Tallying Arguments that
Do Not Interfere with Each Other

The two identical tally arguments in Figure 3-58 will interfere with each other since both are active at the same time. (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 will always contain an accurate count of all of the commas in FIELD1, and T2 will always be unchanged.

```
INSPECT FIELD1 TALLYING
        T1 FOR ALL ","
        T2 FOR ALL "," AFTER "A".
```

Figure 3-58
Two Tallying Arguments that
Do Interfere with Each Other

The following statement achieves the same results as the statement in Figure 3-57. 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 could be more confusing to debug than the one in Figure 3-57.

```
INSPECT FIELD1 TALLYING
        T2 FOR ALL "," AFTER "A"
        T1 FOR ALL ",".
```

Figure 3-59
Two Tallying Arguments that,
Because of their Positioning,
Only Partially Interfere with
Each Other

The preceding three examples show that one INSPECT statement cannot count any character more than once. Thus, when 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 to the problem may require two or more INSPECT statements. Consider the following problem:

```
INSPECT FIELD1 TALLYING
        T1 FOR ALL "AB"
        T2 FOR ALL "BC".
```

Figure 3-60
An Attempt to Tally the Character B
with Two Arguments

If FIELD1 contains "ABCABC", after the scan T1 will be incremented by a 2 and T2 will be unaltered. The successful matching of the argument includes each B in the field. Each match resets the scanner to the character position to the right of the B, and causes the second argument to never be successfully matched. Reversing the order of the arguments has no effect, the results remain the same. Only separate INSPECT statements can develop the desired counts.

Sometimes the programmer 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 "*".
```

Figure 3-61
Tallying Asterisk Groupings

The argument list in Figure 3-61 counts all of the asterisks in FIELD1 but 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 field 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 "*".
```

Figure 3-62
Placing the LEADING Condition
in the Argument List

The placement of the LEADING condition in this sample statement causes
T1 to count only leading asterisks in FIELD1; the occurrence of any
other character stops this counting and 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 this statement results in an
argument list that can never increment T1.


```
INSPECT FIELD1 TALLYING
        T2 FOR ALL "*"
        T1 FOR LEADING "*".
```

Figure 3-63
Reversing the Argument
List in Figure 3-62

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 non-asterisk
character in FIELD1 will fail to match the first argument and the
second argument will become inactive. (The second argument becomes
inactive because it has not found a match in all of the preceding
characters.)

An argument with both a LEADING condition and a BEFORE phrase can
sometimes sucessfully "delimit" the field being inspected:


```
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.
```

Figure 3-64
An Argument List that Counts
Words in a Statement

The statements in Figure 3-64 count the number of "words" in the
English statement in FIELD1. (This assumes that no more than three
spaces separate the words in the sentence and that the sentence ends
with a period.) When FIELD1 has been scanned, T2 contains the number
of gaps between the words. Since a count of the gaps 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 non-space 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 right-hand end of FIELD1. If the sentence in FIELD1 is indented with tab characters instead of spaces, a second LEADING argument can count the tab characters. The following sample statement illustrates this technique:

```
INSPECT FIELD1 TALLYING
        T1 FOR LEADING SPACES
        T1 FOR LEADING TAB
        T2 FOR ALL "    " etc.
```

Figure 3-65
Counting Leading Tab or Space Characters

When an argument list contains a CHARACTERS argument, it should be the last argument in the list. Since the CHARACTERS argument always matches the field, it prevents the application of any of the following arguments in the list. However, as the last argument in an argument list, it can count the remaining characters in the field being inspected. Consider the following illustration.

```
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 ",".
```

Figure 3-66
Counting the Remaining Characters
With the CHARACTERS Argument

If FIELD1 is known to contain a number in the form frequently used to input data, it may contain a plus or minus sign, and a decimal point; further, the number may possibly be preceded by spaces and terminated by a comma. If this statement were compiled and executed, it would deliver the following results:

T1   would contain the number of leading spaces,

T2   would contain the number of periods,

T3   would contain the number of plus signs,

T4   would contain the number of minus signs,

T5   would contain the number of remaining characters (assumed to be numeric), and

the sum of T1 through T5 (plus 1) gives the character position occupied by the terminating comma.

3.9.6 The REPLACING Phrase

When an INSPECT statement contains a REPLACING phrase, that statement selectively replaces characters or groups of characters in the designated field.

The REPLACING phrase names a search argument consisting of a character string of one or more characters and a condition under which the string may be applied to the field 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 field being inspected, under the condition stated, the replacement value replaces the matched characters.

A BEFORE/AFTER phrase may be used to delimit the area of the field being inspected. A search argument applies only to the delimited area of the field.

3.9.6.1 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. Figure 3-67 shows the format of the search argument:

```
 ⎧ ⎛ ALL      ⎞                    ⎫
 ⎪ ⎜          ⎟                    ⎪
 ⎨ ⎨ LEADING  ⎬   ⎛ identifier ⎞  ⎬
 ⎪ ⎜          ⎟   ⎨            ⎬  ⎪
 ⎪ ⎝ FIRST    ⎠   ⎝ literal    ⎠  ⎪
 ⎪                                ⎪
 ⎩ CHARACTERS                     ⎭
```
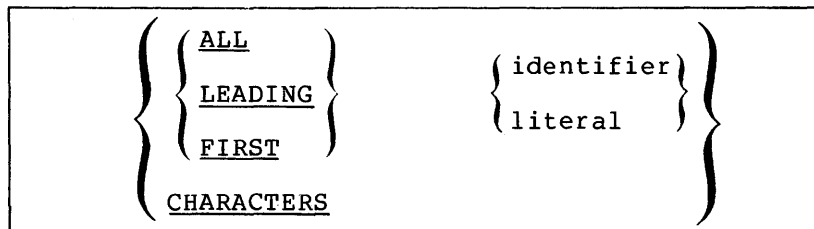
Figure 3-67
Format of the Search Argument

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. (The replacement value, in this case, must be one character long.)

The ALL, LEADING, and FIRST forms of the search argument specify a particular character string, which may be represented by a literal or an identifier. The search argument character string may be any length. However, each character of the argument must match a character in the delimited string before the software considers the argument matched.

- A literal character string must be either non-numeric or a figurative constant (other than ALL literal). A figurative constant, such as SPACE, ZERO, etc., represents a single character and can be written as " ", "0", etc. with the same effect. Since 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 may be any class. However, if it is other than alphabetic, the software performs an implicit redefinition of the item. (This redefinition is identical to the BEFORE/AFTER delimiter redefinition discussed in Section 3.9.1.)

The words ALL, LEADING, and FIRST supply conditions which further delimit the inspection operation:

- The word ALL specifies that each match that 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."

- The word LEADING specifies that only adjacent matches of the search argument at the left-hand end of the delimited character string be replaced. At the first failure to match the search argument, the software terminates the replacement operation and causes the argument to become inactive.

- The word 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.

3.9.6.2 The Replacement Value - Whenever the search argument finds a match in the field being inspected, the matched characters are replaced by the replacement value. The word BY followed by an identifier or literal specifies the replacement value.

$$\underline{BY} \quad \left\{ \begin{array}{l} \text{identifier} \\ \text{literal} \end{array} \right\}$$

Figure 3-68
Format of the Replacement Value

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 non-numeric literal or a figurative constant (other than ALL literal). A figurative constant represents as many characters as the length that the search argument requires.

If the replacement value is an identifier, it must be an elementary item of DISPLAY usage. It may be any class. However, if it is other than alphanumeric, the software conducts an implicit redefinition of the item. (This redefinition is the same as the BEFORE/AFTER redefinition discussed in Section 3.9.1.)

3.9.6.3 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.

```
         ALL ";" BY SPACE BEFORE "."

   search                              BEFORE/AFTER
   argument                            phrase (optional)
              replacement
                 value
```

Figure 3-69
The Replacement Argument

3.9.6.4 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.
```

Figure 3-70
Replacement Argument List that is
Active Over the Entire Field

These three replacement arguments all have the same replacement value, SPACE, and are active over the entire field being inspected.

Thus, this 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".
```

Figure 3-71
Replacement Argument List that
"Swaps" Ones for Zeroes and Zeroes for Ones

Each of these two replacement arguments has its own replacement value, and is active over the entire field being inspected. This statement exchanges zeros for ones and ones for zeroes, and leaves all other characters unchanged.

NOTE

When a search argument finds a match in
the field being inspected, the software
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

3-53

the new position. Thus, it never
inspects the new value that was supplied
by the replacement operation. Because
of this, the search arguments may 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.
```

Figure 3-72
Replacement Argument List that
Becomes Inactive with the
Occurrence of a Space Character

This sample statement is identical to the statement in Figure 3-71,
except that, here, the first occurrence of a space character 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.
```

Figure 3-73
Argument List with Three Arguments
That Become Inactive with the
Occurrence of a Space

Just as in the argument list in Figure 3-72, the first space character
causes all of these replacement arguments to become inactive. This
argument list exchanges zeroes for ones, ones for zeroes, 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 zeroes with asterisks. Beyond this area, it replaces
all characters (including the space that delimited FIELD1 for the
first two arguments and any zeroes and ones) with asterisks.

3.9.6.5 **Interference in Replacement Argument Lists** - When several
search arguments that are active at the same time contain one or more
identical characters, they may interfere with each other, and
consequently have an effect on the replacement operation. This
interference of one search argument with the matching of other search
arguments 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 software 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.

3-54

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 3.9.5.4 Interference in Tally Argument Lists, applies, generally, to replacement arguments as well.)

The following rules will help minimize interference in replacement argument lists:

1. Place search arguments with LEADING or FIRST conditions at the start of the list;

2. Place several arguments with the CHARACTERS condition at the end of the list;

3. Consider, very carefully, the order of appearance of any search arguments that contain one or more identical characters.

3.9.7 Common Errors, INSPECT Statement

The most common errors made 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" e.g.:

        INSPECT FIELD1 TALLYING TLY FOR SPACES.

CHAPTER 4

NUMERIC CHARACTER HANDLING

4.1 INTRODUCTION

This chapter discusses numeric class data and the COBOL operations
that may be performed on numeric class data. It is assumed that the
reader has read Chapter 3, and understands the concept of COBOL data
classes.

4.2 USAGES, DISPLAY/COMP

The USAGE of a numeric class item specifies the form in which that
item's data is held in memory. COBOL has two basic formats for data
storage, DISPLAY and COMPUTATIONAL (DISPLAY, DISPLAY-6, and DISPLAY-7
are all equivalent):

   ● ANS-74 COBOL standards prescribe DISPLAY usage to be a string
     of characters or bytes in decimal radix, with an assumed
     decimal point location and various sign conventions.

   ● The same standards prescribe COMPUTATIONAL (COMP) usage to be
     a real number with the same range of values as a DISPLAY usage
     number. However, with COMP usage, the compiler implementor
     has the liberty of specifying the form in which that number is
     held in memory.

PDP-11 COBOL stores COMP usage fields as binary numbers in one, two,
three or four words with an assumed decimal scaling position.
Consider the following field description:

        01 GEMINI PIC 99V99 COMP.

If this field contains the value 12.34, PDP-11 COBOL stores it as a
1-word binary number. The word contains the integer value 1234 and
another location contains the scaling factor. In this example, the
scaling factor records the fact that this integer has two decimal
fractional positions associated with it. Thus, the COBOL OTS knows
that the stored binary integer is 100 times larger than the programmer
intends it to be.

If the compiler encounters the following statement:

        ADD 1 TO GEMINI.

it generates instructions to add a 1 to the 1234 in GEMINI. The OTS,
however, scales the literal 1 up by two decimal places and adds the
resultant literal, 100, to the number in GEMINI. Thus, after the ADD
operation, GEMINI contains the new value 1334 (which is actually 13.34
with the stored decimal scaling position).

Thus, the PDP-11 COBOL compiler and OTS manipulate the data in both DISPLAY and COMP usage items in much the same way. Both usages have exactly the same accuracy and precision, and can be freely mixed in a program. If a DISPLAY usage number and a COMP usage number are both involved in the same arithmetic statement, the OTS converts them to a common radix, with no loss of information. It also converts the result (if necessary), before storing it, with no loss of significance.

The only effect of specifying the COMP usage is that it reduces the space required for most numbers and speeds up the execution of arithmetic statements.

4.2.1 Sign Conventions

DISPLAY or COMP usage numeric items may be signed or unsigned. Unsigned numbers may contain values that range from zero to the largest positive value allowed by their declared precision. Negative values are not allowed. All PDP-11 COBOL arithmetic operations yield signed results. When the OTS must store such a result, whether positive or negative, in an unsigned data item, it stores only the absolute value of the result. Thus, unsigned items always contain zero or positive values.

This guide does not recommend unsigned numbers for general use. They are usually a source of programming errors, and are handled less efficiently than signed quantities by the OTS.

Signed quantities always contain a numeric value and an operational sign. The OTS stores the sign with the numeric value in a variety of ways depending on the usage of the item and the presence of the SIGN clause.

NOTE

If numeric data is read into a field
described using the picture character S,
then that data must include an
operational sign of the appropriate
format to pass the NUMERIC test.

PDP-11 COBOL always stores signed COMP items in two's complement binary form. Thus, the high-order bit indicates the sign of the item.

PDP-11 COBOL always stores signed DISPLAY items as a sequence of byte positions containing numeric ASCII characters. It may include the sign in the high-order byte, the low-order byte, or as a separate, extra, byte on either the high-order or low-order end of the item.

When the OTS stores the sign as part of a byte that also contains a numeric digit, the sign causes a value change in that byte and, hence, changes the value of the numeric digit. Table 4-1 shows the actual ASCII character that results when a numeric value and a sign share the same byte.

Table 4-1
The Resulting ASCII Character From a
Sign and Digit Sharing the Same Byte

| | | DIGIT VALUE | | | | | | | | | |
		0	1	2	3	4	5	6	7	8	9
SIGN	+	{	A	B	C	D	E	F	G	H	I
	−	}	J	K	L	M	N	O	P	Q	R

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.

When the OTS stores the sign as a separate distinct character, the actual ASCII character that it stores is the graphic plus sign (octal 053) or the graphic minus sign (octal 055).

4.2.2 Illegal Values in Numeric Fields

All PDP-11 COBOL arithmetic operations store legal values in their result fields. However, it is possible, by reading invalid data or through redefinition and group moves, to store data in numeric fields that do not obey the descriptions of those fields. (For example, it is possible to place signed values into unsigned fields, and to place non-numeric or improperly signed data into signed numeric DISPLAY fields.) PDP-11 COBOL handles this data in the following manner.

NOTE

The following four compiler techniques are not specified by ANS-74 COBOL standards. Dependence on them may yield programs that are not compiler independent.

1. When a quantity described as unsigned enters into an arithmetic operation, the OTS treats it as a signed quantity. If it contains no sign, the OTS either considers the sign to be positive, or ignores the sign if the value of the field is zero. If the field does contain a legally constructed sign, the OTS interprets the sign as if the item had been described as a signed quantity.

   Thus the OTS treats a negative value in an unsigned COMP item as a negative value. Likewise a negative sign, stored as J through R, in the rightmost digit of an unsigned DISPLAY item, causes the OTS to treat that value as a negative value.

   When an arithmetic operation or legal elementary MOVE statement places its result in an unsigned item, that item receives the absolute value only (no sign information is encoded in the result).

For example the following coding results in an unsigned value of 15 in field B:

```
        .
        .
        .
    02 A PIC S99 VALUE IS 5
    02 C PIC XX VALUE IS "2 ".
    02 B REDEFINES C PIC 99.
        .
        .
        .
    ADD A TO B.
```

However, given the same original values in A and B, the following statement would result in a value of -15 in field A. (Field B remains at -20.)

```
    ADD B TO A.
```

2. When a signed quantity enters into an arithmetic operation, the OTS interprets the sign as follows:

    ●   If the item is COMP usage, the OTS takes the value as a two's complement number;

    ●   If the item is DISPLAY usage and the sign is encoded within the leading or trailing digit position, the OTS takes the sign as positive if that position contains either a { (octal 173) or any ASCII byte that collates less than J. Thus, the OTS considers a space, 0 through 9, and A through I, to be positive. Further, it considers any ASCII character that collates equal-to or higher-than J (except { --octal 173) to be negative. (The OTS conducts this sign determination separately from the numeric value determination for the same byte.)

VALUE	SIGN DETERMINATION
000A	+0001
000J	-0001

    ●   If the item is DISPLAY usage and the sign is encoded as a separate leading or trailing byte, the OTS takes the sign as negative only if that byte contains the ASCII character - (octal 055). The OTS considers that all other ASCII characters in the separate sign position indicate a positive field.

3. A COMP usage item may receive a value that is larger than the specified range. For example, the OTS stores a field described as PIC S9999 COMP as a 16-bit binary number. The declared range is four decimal digits, but the field has the capability of storing any value from -32,768 to +32,767 (decimal). The OTS stores the results of an arithmetic operation on such a field as a value modulo the declared decimal range. Thus, any value that exceeds 9,999 stores a modulo 10,000 value. (The binary value of 10,000 stores as a zero.)

When a COMP usage field enters an arithmetic operation, however, the OTS uses the full binary number as the binary value of the field. Thus, a value stored in a COMP field by a group move may cause the field to contain a value that exceeds the declared range. Arithmetic operations will use that value as found.

4. A DISPLAY usage item may contain a value in which some or all of the numeric digit positions contain illegal values.

When a DISPLAY usage numeric item enters an arithmetic operation, the OTS converts each character to a binary value either before or during the operation. This conversion maps certain ASCII characters into the numeric values 0 through 9, and all other ASCII characters into the numeric value 0. Table 4-2 shows these conversion values:

<div align="center">

Table 4-2
Conversion Values

ASCII CHARACTERS			BINARY VALUES
A	J	1	0001 (1)
B	K	2	0010 (2)
C	L	3	0011 (3)
D	M	4	0100 (4)
E	N	5	0101 (5)
F	O	6	0110 (6)
G	P	7	0111 (7)
H	Q	8	1000 (8)
I	R	9	1001 (9)
ALL OTHERS			0000 (0)

</div>

All arithmetic operations (including the numeric elementary MOVE) deliver the ASCII characters 1 through 9 and 0 into all digit positions of a numeric DISPLAY field. A digit position that also contains a sign value receives a correctly coded sign value as shown in Table 4-1.

4.3 TESTING NUMERIC FIELDS

COBOL provides the following three kinds of tests for evaluating
numeric fields:

1. Relation tests, that compare the field's contents to another
    numeric value;

2. Sign tests, that examine the field's sign to see if it is
    positive or negative; and,

3. Class tests, that inspect the field's digit positions for
    legal numeric values.

The following sub-sections explain these tests in detail.

4.3.1 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 FIELD1 to FIELD2 and determines if the numeric
value of FIELD1 is greater than the numeric value of FIELD2. If so,
the relation condition is true and program control takes the True path
of the statement.

        IF FIELD1 > FIELD2 ...

Either field in a relation test may 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 sizes of the fields in a numeric relation test do not have to be
the same (this includes the sizes of numeric literals). The
comparison operation aligns both fields on their assumed decimal
positions (through actual scaling operations in temporary locations or
by accessing the individual digits) and supplies leading or trailing
(as required) zeroes to either or both fields.

The comparison operation always compares the signs of non-zero fields
and considers positive fields to be greater than negative fields.
However, since it does not compare them, positive zeroes and negative
zeroes are equal. (A negative zero could arrive in a field through
redefinition of the field or a MOVE to a group item.) Further, the
operation considers unsigned numeric fields to be positive.

The form of representation of the number (COMP or DISPLAY usage) and
the various methods of storing DISPLAY usage signs have no effect on
numeric relation tests.

For comparison purposes, the operation converts any illegal characters
stored in DISPLAY usage fields to zeroes. It does not, however, alter
the actual values in those fields.

4.3.2 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 would 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 4-3 shows the sign tests and their equivalent relation tests as
applied to FIELD1.

Table 4-3
The Sign Tests

SIGN TEST	EQUIVALENT RELATION TEST
IF FIELD1 POSITIVE ... IF FIELD1 NOT POSITIVE ... IF FIELD1 NEGATIVE ... IF FIELD1 NOT NEGATIVE ... IF FIELD1 ZERO ... IF FIELD1 NOT ZERO ...	IF FIELD1 > 0 ... IF FIELD1 NOT > 0 ... IF FIELD1 < 0 ... IF FIELD1 NOT < 0 ... IF FIELD1 = 0 ... IF FIELD1 NOT = 0 ...

Sign tests have no execution speed advantage over relation tests. The
compiler actually substitutes the equivalent relation test for every
correctly written sign test. (Sections 4.2.1 and 4.2.2 discuss the
acceptable sign values and the treatment of illegal sign values.)

4.3.3 Class Tests

The class test interrogates a numeric field to determine if it
contains numeric or alphabetic data, and uses the result to alter the
flow of control in a program. For example, the following statement
determines if FIELD1 contains numeric data. If so, the test condition
is true and program control takes the true path of the statement.

        IF FIELD1 IS NUMERIC ...

When reading in newly prepared data, it is often desirable to check
certain fields for valid values. Relation tests and sign tests can
only determine if the field's contents are within a certain range, and
these tests both treat illegal characters in DISPLAY usage items as
zeroes. Thus, some data preparation errors could pass both of these
tests.

The NUMERIC class test checks numeric (or alphanumeric) DISPLAY usage
fields for valid numeric digits.

If the field being tested contains a sign (whether carried as an
overpunch 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. If the
character position contains a valid sign and all digit positions in
the field contain valid numeric digits, the NUMERIC class test passes
the item and program control takes the true path of the IF statement.

The ALPHABETIC class test checks alphabetic (or alphanumeric) fields for valid alphabetic characters and the space character. If all of the character positions of the field contain ASCII characters (A-Z or space), 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 Chapter 3, Section 3.3.2.)

4.4 THE MOVE STATEMENT

The MOVE statement moves the contents of one field into another. The following sample MOVE statement moves the contents of FIELD1 into FIELD2.

        MOVE FIELD1 TO FIELD2.

Section 3.5 discusses the basic MOVE statement. This section considers MOVE statements as applied to numeric fields. These MOVE statements can be grouped into the following three categories:

1. Group moves,

2. Elementary moves with numeric receiving fields, and

3. Elementary moves with numeric edited receiving fields.

The following three sub-sections (4.4.1, 4.4.2, and 4.4.3) discuss each of these categories separately.

4.4.1 Group Moves

The software considers a move to be a group move if either the sending field or the receiving field is a group item. It treats both fields in a group move as alphanumeric class fields and performs the move as an alphanumeric to alphanumeric elementary move.

If either field in a group move is a numeric elementary item, the OTS treats the storage area occupied by that item as a field of alphanumeric bytes; thus, it ignores the USAGE, sign, and decimal point location characteristics 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 with the numeric digit positions.

4.4.2 Elementary Numeric Moves

If both fields of a MOVE statement are elementary items and the receiving field is numeric, the OTS considers the move to be an elementary numeric move. (The sending field may be either numeric or alphanumeric.) The numeric receiving field may be either DISPLAY or COMP usage. The elementary numeric move converts the data format of the sending field to the data format of the receiving field.

An alphanumeric sending field may be either an elementary data item or 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 fields as unsigned integers of DISPLAY usage.

If necessary, the numeric move operation converts the sending field to the data format of the receiving field and aligns the sending field's decimal point on that of the receiving field. It then moves the sending field digits to their corresponding receiving field digits.

If the sending field has more digit positions than the receiving field, the decimal point alignment operation truncates the sending field, with the resultant loss of digits. The end truncated (high-order or low-order) depends upon the number of sending field digit positions that find matches on each side of the receiving field's decimal point. If the receiving field has fewer digit positions on both sides of the decimal point, the operation truncates both ends of the sending field. Thus, if a field described as PIC 999V999 is moved to a field described as PIC 99V99, it loses one digit from the left end and one from the right end. Figure 4-1 illustrates this alignment operation (the carat (∧) indicates the stored decimal scaling position):

```
    01 GANDALF PIC 99V99.

    ...

    MOVE 123.321 TO GANDALF.
```

Before execution   00∧00

After execution   23∧32

Figure 4-1
Truncation Caused By Decimal Point Alignment

If the sending field has fewer digit positions than the receiving field, the move operation supplies zeroes for all unfilled digit positions. Figure 4-2 illustrates this alignment (the carat (∧) indicates the stored decimal scaling position):

```
    01 RIVENDELL PIC 999V99.

    ...

    MOVE 1 TO RIVENDELL.
```

Before execution   000∧00

After execution   001∧00

Figure 4-2
Zero Filling Caused By Decimal Point Alignment

The following statement produces the same results:

```
    MOVE 001.00 TO RIVENDELL.
```

4-9

Consider the following two MOVE statements and their resultant truncating and zero-filling effects:

STATEMENT	RIVENDELL AFTER EXECUTION
MOVE 00100 TO RIVENDELL	100 00
MOVE "00100" TO RIVENDELL	100 00

Literals with leading or trailing zeroes have no significant advantage in space or execution speed with PDP-11 COBOL, and the zeroes are often lost by decimal point alignment.

The MOVE statement's receiving field dictates how the sign will be moved. A signed DISPLAY usage receiving field causes the sign to be moved as a separate quantity. An unsigned DISPLAY usage receiving field causes no sign movement. A COMP usage receiving field, whether signed or unsigned, causes the sign to be moved; however, if the receiving field is unsigned, the OTS sets its value to absolute.

4.4.3 Elementary Numeric Edited Moves

The PDP-11 COBOL object time system considers an elementary numeric move to a receiving field of the numeric edited category to be an elementary numeric edited move. The sending field of an elementary numeric edited move may be either numeric or alphanumeric and, if numeric, it can be either DISPLAY usage or COMP usage. The OTS treats alphanumeric sending fields in numeric edited moves as unsigned DISPLAY usage integers.

The OTS considers the receiving field to be numeric edited category if it is described with a BLANK WHEN ZERO clause, or a combination of the following symbols:

B   Space insertion position;

P   Decimal scaling position;

V   Location of assumed decimal point;

Z   Leading numeric character position to be replaced by a space if the position contains a zero;

0   Zero insertion position;

9   Position contains a numeric character;

/   Slash insertion position;

,   Comma insertion position;

.   Decimal point insertion position;

*   Leading numeric character position to be replaced by an asterisk if the position contains a zero;

+   Positive editing sign control symbol;

-    Negative editing sign control symbol;

CR   Credit editing sign control symbol;

DB   Debit editing sign control symbol;

cs   Currency symbol ($) insertion position.

A numeric edited field may contain 9, V, and P, but combinations of those symbols without an editing character do not make the field numeric edited.

The numeric edited move operation first converts the sending field to DISPLAY usage and aligns both fields on their decimal point locations, truncating or padding (with zeroes) the sending field until it contains the same number of digit positions on both sides of the decimal point as the receiving field. It then moves the resulting digit values to the receiving field digit positions following the COBOL editing rules.

The COBOL editing rules allow the numeric edited move operation to perform any of the following editing functions:

● Suppress leading zeroes with either spaces or asterisks;

● Float a currency sign and a plus or minus sign through suppressed zeroes, inserting the sign at either end of the field;

● Insert zeroes and spaces;

● Insert commas and a decimal point.

Figure 4-3 illustrates several of these functions with the statement, MOVE FRODO TO RIVENDELL. (Assume that FRODO is described as S9999V99.)

FRODO	RIVENDELL	
	PICTURE STRING	CONTENTS AFTER MOVE
0023ˆ00	ZZZZ.99	Δ Δ23.00
0085ˆ9O	++++.99	Δ -85.96
1234ˆ00	Z,ZZZ.99	1,234.00
0012ˆ34	$,$$$.99	Δ $12.34
0000ˆ34	$,$$9.99	Δ Δ$0.34
1234ˆ00	$$,$$$.99	$1,234.00
0012ˆ34	$$9,999.99	$0,012.34
0012ˆ34	$$$$,$$$.99	ΔΔΔΔ $12.34
0000ˆ00	$$$,$$$.$$	ΔΔΔΔΔΔΔΔΔ
0012ˆ3M	++++.99	-12.34
0012ˆ34	$***,***.99	$**1,234.00
0012ˆ34	$***,***.99	$*****12.34

Figure 4-3
Numeric Editing

The currency symbol ($) 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.

4.4.4 Common Errors, Numeric MOVE Statements

The most common errors made when writing numeric MOVE statements are:

- Placing an incorrect number of replacement characters in a numeric edited item.
- Moving non-numeric data into numeric fields 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 which can be completely replaced).

4.5 THE ARITHMETIC STATEMENTS

The COBOL arithmetic statements, ADD, SUBTRACT, MULTIPLY, DIVIDE, and COMPUTE allow COBOL programs to perform simple arithmetic operations on numeric data.

This section covers the use of the COBOL arithmetic statements. The first five sub-sections (4.5.1 through 4.5.5) discuss the common features of the statements and the last five (4.5.6 through 4.5.10) discuss the individual arithmetic statements themselves.

4.5.1 Intermediate Results

Most forms of the arithmetic statements perform their operations in temporary work locations, then move the results to the receiving fields, aligning the decimal points and truncating or zero filling the resultant values.

This temporary work field, called the intermediate result field, has a maximum size of 18 numeric digits. The actual size of the intermediate result field 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 field exceeds 18 digits, it truncates the excess high-order digits. Thus, a program that requests a multiplication operation between the following two fields,

> PIC 9(18) and PIC V99.

(which would otherwise cause the compiler to set up a 20-digit intermediate result field -- 9(18)V99) actually causes the following intermediate result field

> PIC 9(16)V99.

PDP-11 COBOL truncates high-order digits or low-order digits to the right of the decimal point, based on the assumption that most large data declarations are larger than ever need be, so zeroes occupy most of their high-order digit positions. Numeric data may be declared as PIC 9(12) or PIC 9(15) but the values that are placed in these fields will probably not exceed nine digits of range (1 billion) in most applications.

When 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.

If truncation is a possibility, reduce the size of the number by dividing it by a power of 10 prior to the arithmetic operation. (This scaling down operation causes the low-order end to lose digits, but these are probably less critical.) Then, after the arithmetic operation, multiply the result by the same power of 10.

To save the low-order digits in such an operation, move the field to a temporary location before the scaling DIVIDE, perform separate, identical arithmetic operations on both the original and the temporary fields, then, after the scaling MULTIPLY, combine their results.

4.5.2 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 arithmetic operations.

The phrase may be used on any COBOL 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 field.

PDP-11 COBOL rounds-off by adding a 5 to the leftmost truncated digit of the absolute value of the intermediate result before it stores that result.

Consider the following illustration and assume an intermediate result of 54321.2468:

```
┌─────────────────────────────┬──────────────────────────────────┐
│ Coding:                     │                                  │
├─────────────────────────────┴──────────────────────────────────┤
│                                                                 │
│     01 BILBO PIC S9(5)V9999.                                    │
│     01 FRODO PIC S9(5)V99.                                      │
│     ...                                                         │
│         ADD BILBO TO FRODO ROUNDED.                             │
│     ...                                                         │
├─────────────────────────────┬──────────────────────────────────┤
│ Intermediate result field:  │                                  │
├─────────────────────────────┴──────────────────────────────────┤
│     PIC S9(6)V9999.                                             │
├─────────────────────────────┬──────────────────────────────────┤
│ The ROUNDED operation:      │                                  │
├─────────────────────────────┴──────────────────────────────────┤
│                                                  Truncated      │
│                                                 ↙   digits      │
│   Intermediate result field: 054321.24  │68                     │
│                                              ↘ LEFT-MOST        │
│   ROUNDED:           (ADD)          .00  │50   truncated        │
│   FRODO's ROUNDED result:     054321.25  │18       digit        │
└─────────────────────────────────────────────────────────────────┘
```

Figure 4-4
Rounding Truncated Decimal Point Positions

The following ROUNDING example rounds-off to the decimal scaling position (P). Assume an intermediate result of 24680. (Section 4.5.4 discusses the GIVING phrase in numeric operations.)

```
Coding:

        01  GANDALF PIC 9999.
        01  SARUMAN PIC 9999PP.


            ...
            MULTIPLY GANDALF BY 10

                GIVING SARUMAN ROUNDED.
            ...


Intermediate result field:

        PIC 999999.

The ROUNDED operation:

                                                Truncated

        Intermediate result  field:  0246  80.        digits

        ROUNDED                (ADD):      50.

        SARUMAN's    ROUNDED result:  0247  30.
```

Figure 4-5
Rounding Truncated Decimal Scaling Positions

4.5.3 The SIZE ERROR Phrase

The SIZE ERROR phrase detects the loss of high-order non-zero digits
in the results of COBOL arithmetic operations.

The phrase may be used on any COBOL 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 execution of a statement
with a SIZE ERROR phrase results in a size error, the OTS discards the
entire result (it does not alter the receiving fields 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 phrase cannot be used on numeric MOVE statements.   Thus, if a
program moves a numeric quantity to a smaller numeric field, it may
inadvertently lose high-order digits.   For example, consider the
following MOVE of a field to a smaller field:

        01 BIGFOOT PIC 9(8)V99.

        01 LITTLEFOOT PIC 9(4)V99.

            ...

        MOVE BIGFOOT TO LITTLEFOOT.

This MOVE operation always loses four of BIGFOOT's high-order digits.
Either of the following two statements could determine whether these
digits are zero or non-zero, and could be tailored to any size field:

1. IF BIGFOOT NOT > 9999.99
   MOVE BIGFOOT TO LITTLEFOOT
   ELSE ...

2. ADD ZERO TO BIGFOOT GIVING LITTLEFOOT
   ON SIZE ERROR ...

Both of these alternatives allow the MOVE operation to occur only if
BIGFOOT loses no significant digits. If the value in BIGFOOT is too
large, both alternatives avoid altering LITTLEFOOT and take the
alternative execution path.

4.5.4 The GIVING Phrase

The GIVING phrase moves the intermediate result field of an arithmetic
operation to a receiving field. (The phrase acts exactly like a MOVE
statement with the intermediate result serving as a sending field and
the data item following the word GIVING (in the statement) serving as
a receiving field.)

The phrase may be used on the ADD, SUBTRACT, MULTIPLY, and DIVIDE
statements.

If the data item following the word GIVING is a numeric edited field,
the OTS performs the editing the same way it does for MOVE statements.

4.5.5 Multiple Operands in ADD and SUBTRACT Statements

Both the ADD and SUBTRACT statements may contain a string of more than
one operand 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 field.

The following three equivalent coding groups illustrate how the
software executes the multiple operand statements:

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.
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.

(Just as with all COBOL statements, any commas in these statements are optional.)

Only statement 3 may have a numeric edited receiving field, since it is the only statement containing a GIVING phrase.

4.5.6 The ADD Statement

The ADD statement adds two or more operands together and stores the result.

The statement may contain multiple operands (with the exception of Format 3) and the ROUNDED and SIZE ERROR phrases. It may be written in one of the following formats:

    <u>Format 1.</u>   ADD FIELD1 ...TO FIELD2 FIELD3 ... .

    <u>Format 2.</u>   ADD FIELD1 FIELD2 ...GIVING FIELD3 FIELD4 ... .

    <u>Format 3.</u>   ADD CORRESPONDING FIELD1 TO FIELD2.

In Format 1, the receiving fields (FIELD2, FIELD3) are one of the addends. These must not be in the numeric edited category.

In Format 2, the receiving fields (FIELD3, FIELD4) are not one of the addends. They may either be numeric or numeric edited. When using this format, omit the word TO.

In Format 3, the receiving field (FIELD2) is one of the addends. Both FIELD1 and FIELD2 must be group items. The corresponding elements of FIELD1 are added to the corresponding elements of FIELD2.

4.5.7 The SUBTRACT Statement

The SUBTRACT statement subtracts one, or the sum of two or more, operands from another operand and stores the result.

The statement may contain multiple operands (with the exception of Format 3) and the ROUNDED and SIZE ERROR phrases. It may be written in one of the following formats:

    <u>Format 1.</u>   SUBTRACT FIELD1 ... FROM FIELD2 FIELD3 ... .

    <u>Format 2.</u>   SUBTRACT FIELD1 ... FROM FIELD2
                      GIVING FIELD3 FIELD4 ... .

    <u>Format 3.</u>   SUBTRACT CORRESPONDING FIELD1 FROM FIELD2.

In Format 1, the receiving fields (FIELD2, FIELD3) are both the subtrahend and the difference (the result). These must not be in the numeric edited category.

In Format 2, the receiving fields (FIELD3, FIELD4) are used only to store the result. They may be either numeric or numeric edited.

In Format 3, the receiving field (FIELD2) is both the subtrahend and the difference (results). Both FIELD1 and FIELD2 must be group items. The corresponding elements of FIELD2.

4.5.8 The MULTIPLY Statement

The MULTIPLY statement multiplies one operand by another and stores
the result.

The statement may contain the ROUNDED and SIZE ERROR phrases. It may
contain multiple receiving operands. It may be written in either of
the following formats:

>   Format 1.   MULTIPLY FIELD1 BY FIELD2, FIELD3 ... .

>   Format 2.   MULTIPLY FIELD1 BY FIELD2 GIVING FIELD3, FIELD4 ... .

In Format 1, the receiving fields (FIELD2, FIELD3) are also the
multipliers. These must not be in the numeric edited category.

In Format 2, the receiving fields (FIELD3, FIELD4) are neither
multiplier nor multiplicand. These may be either numeric or numeric
edited.

COBOL's "near English" format could cause a problem with the MULTIPLY
statement, since it is common to speak of multiplying a number
(multiplicand) by another number (multiplier) and to think of the
result as a new value for the multiplicand; thus:

>   MULTIPLY EARNINGS BY 0.24.
>   Multiplier
>   Multiplicand

This statement is incorrect since the OTS stores the result in the
multiplier field, and this multiplier is a literal. The compiler
could diagnose this error, but would not diagnose it if the multiplier
were a data item.   Consider this multiplier written as a data item:

>   MULTIPLY EARNINGS BY TAX-RATE.

The compiler would not diagnose this statement's error, and would
store the result of the operation in TAX-RATE. A good practice when
using MULTIPLY statements is to always write them in Format 2.   This
ensures that the result is properly stored.   The following two
statements safely capture their results:

>   MULTIPLY EARNINGS BY 0.24 GIVING EARNINGS.

>   or

>   MULTIPLY EARNINGS BY TAX-RATE GIVING EARNINGS.

4.5.9 The DIVIDE Statement

The DIVIDE statement divides one operand into another and stores the
result.

The statement may contain the ROUNDED and SIZE ERROR phrases.   With
the exception of Formats 4 and 5, it may not contain multiple
receiving operands. It may be written in any of the following
formats:

Format 1.   DIVIDE FIELD1 INTO FIELD2 FIELD3 ... .

Format 2.   DIVIDE FIELD1 INTO FIELD2 GIVING FIELD3 FIELD4 ... .

Format 3.   DIVIDE FIELD2 BY FIELD1 GIVING FIELD3 FIELD4 ... .

Format 4.   DIVIDE FIELD1 INTO FIELD2 GIVING FIELD3 REMAINDER
                    FIELD4.

Format 5.   DIVIDE FIELD1 BY FIELD2 GIVING FIELD3 REMAINDER
                    FIELD4.

In Format 1, the receiving fields (FIELD2, FIELD3) are also the dividends. These must not be in the numeric edited category.

In Formats 2 and 3, the receiving fields (FIELD3, FIELD4 ...) are neither dividends nor divisor. These may be either numeric or numeric edited.

In Formats 4 and 5, the receiving field (FIELD3) is neither a dividend nor a divisor. FIELD4 is the remainder. The receiving field and the remainder may be either numeric or numeric edited.

4.5.10 The COMPUTE Statement

The COMPUTE statement computes the value of an arithmetic expression and stores the value in the result.

The statement may contain the ROUNDED and SIZE ERROR phrases. It may contain multiple receiving operands. The COMPUTE statement has the following format:

        COMPUTE FIELD1 FIELD2 ... = arithmetic-expression.

The receiving fields (FIELD1, FIELD2) may be either numeric or numeric edited.

4.5.11 Common Errors, Arithmetic Statements

The most common errors made when using arithmetic statements are:

- Using an alphanumeric class field in an arithmetic statement. The MOVE statement allows data movement between alphanumeric class fields and certain numeric class fields, but arithmetic statements require that all fields be numeric.

- Writing the ADD or SUBTRACT statements without the GIVING phrase, but attempting to put the result into a numeric edited field.

- Writing a Format 2 ADD statement with the word TO; For example:

        ADD A TO B GIVING C.

- Subtracting a 1 from a numeric counter that was described as an unsigned quantity, and 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 in such a way as to generate an intermediate result that is larger than 18 digits when the final result will be fewer digits. (The programmer should be careful to intersperse divisions with multiplications or to drop non-significant digits that result from multiplying large numbers (or numbers with many decimal places).

● Performing an operation on a field that contains a value greater than the precision of its data description. This can happen only if the field was disarranged by a group move or redefinition.

● Forgetting that, in an arithmetic statment containing multiple receiving fields, the ROUNDED phrase must be specified for each receiving field that is to be rounded.

● Forgetting that, in an arithmetic statement containing multiple receiving fields, the ON SIZE ERROR phrase, if specified, applies to all receiving fields. Only those receiving operands for which a size error condition is raised are left unaltered. The ON SIZE ERROR imperative statement is executed after all the receiving fields are processed by the OTS.

4.6 ARITHMETIC EXPRESSION PROCESSING

4.6.1 Motivation for Intermediate Results

COBOL provides language facilities for manipulating user-defined data arithmetically. In particular, the language provides the arithmetic statements ADD, SUBTRACT, MULTIPLY, and DIVIDE and the facilities of arithmetic expressions using the +, -, *, /, and ** operators. In simple terms, a given arithmetic functionality may be expressed in one of several ways. For example, consider a COBOL application in which the total yearly sales of a salesman are to be computed as the sum of the four individual sales quarters. Figure 4-6 illustrates one method of expressing a solution to this problem in COBOL:

```
.

.

.

MOVE 0 TO TEMP.

ADD 1ST-SALES TO TEMP.

ADD 2ND-SALES TO TEMP.

ADD 3RD-SALES TO TEMP.

ADD 4TH-SALES TO TEMP GIVING TOTAL-SALES.

.

.

.
```

Figure 4-6 Explicit Programmer-Defined Temporary Work Area

In figure 4-6, the COBOL programmer chooses to use a series of single ADD statements to develop the final value for TOTAL-SALES. In the process of computing TOTAL-SALES, a COBOL data-name, called TEMP, is used to develop the partial sums (i.e., intermediate results). The important point here is that the programmer explicitly defines and declares the temporary work area TEMP in the data division of the COBOL program. That is, the attributes (i.e., class, USAGE, number of integer and decimal places to be maintained) are specified explicitly by the COBOL programmer.

Figure 4-7 below illustrates another way of expressing a solution to the problem:

```
.

.

.

ADD 1ST-SALES, 2ND-SALES, 3RD-SALES, 4TH-SALES
    GIVING TOTAL-SALES.

.

.

.
```

Figure 4-7
Arithmetic Statement Intermediate Result Field Attributes
Determined from Composite of Operands

In this example, the programmer chooses to compute TOTAL-SALES with a single ADD statement. Analogous to the previous example, an intermediate result field is required to develop the partial sums of the four quarterly sales quantities. In Figures 4-6, the programmer is cognizant of this requirement, but chose to define the intermediate result area TEMP explicitly in the data division of his COBOL program. However, for the example in Figure 4-7, the software defines the intermediate result field in a manner transparent to the COBOL source program. That is, the software allocates storage for and assigns various attributes to this "transparent" intermediate result field according to a well-defined set of rules defined by the COBOL language specification. In particular, the attributes of number-of-integer-places, number-of-decimal-places, and USAGE assigned by the software to the intermediate result field are a function of the composite of source operands in the ADD statement. (The reader should read the PDP-11 COBOL Reference Manual for details concerning the composite of operands for the arithmetic statements.) The important point here is that the ANS-74 COBOL language standard prescribes rules for determining the attributes of intermediate result fields for the arithmetic statements and the associated language processor (e.g., PDP-11 COBOL compiler) must implement those rules.

As a final example, consider the following solution to our problem:

```
.

.

.

COMPUTE TOTAL-SALES = 1ST-SALES + 2ND-SALES + 3RD-SALES
        + 4TH-SALES.

.

.

.
```

Figure 4-8
Arithmetic Expression Intermediate Result Field
Attributes Determined by Implementor-Defined Rules

In Figure 4-8, the programmer solves the problem by using a single COMPUTE statement with an embedded arithmetic expression. Again, an intermediate result field is required and, as in Figure 4-7, is defined by the software. However, in defining the attributes of intermediate result fields for COBOL arithmetic expressions, the ANS-74 COBOL language standard is not as helpful to the user as it could be. In fact, the COBOL language standard gives almost complete freedom to the implementor in defining the attributes of the arithmetic expression intermediate result fields. The only rules imposed by the ANS-74 COBOL language specifications are:

1. Arithmetic operations are to be combined without restrictions on the composite of operands and/or receiving fields.

2. Each implementor will indicate techniques used in handling arithmetic expressions.

Thus, the user can and should expect differences between various implementations of ANS-74 COBOL. The purpose of the remainder of this section is to specify the conceptual algorithms used by the software to compute the attributes for the arithmetic expression intermediate result field; i.e., the number-of-integer-places and number-of-decimal-places to be maintained (for a given intermediate result field) as a function of a particular arithmetic operator and its associated operands.

4.6.2 Intermediate Results for Arithmetic Expressions

In the compile-time and object-time processing of arithmetic expressions, the software maintains a maximum precision of 18 decimal digits for intermediate result fields. One of the major compile-time functions in processing an arithmetic expression operator x is to determine the following attributes for the associated object-time intermediate result field IR(x):

1. USAGE type,

2. number of integer places, I(IR(x)), to be maintained, and

3. number of decimal places, D(IR(x)), to be maintained.

All arithmetic expression intermediate result fields are treated as signed data. The software must determine the USAGE type for an intermediate result in order to determine the form of its internal representation; the number of integer places and decimal places are determined to know how much object-time storage is required for the intermediate result.

With the exception of the exponentiation operator (**), all infix arithmetic operators yield an intermediate result field whose USAGE type is determined as a function of the USAGE types of its two source operands. That is, if both source operands are DISPLAY, the USAGE type for the intermediate result field is also DISPLAY. If one of the operands associated with an infix operator has a USAGE IS COMPUTATIONAL declaration, the USAGE type of the intermediate result field is also COMPUTATIONAL (i.e., the intermediate result is represented as COMPUTATIONAL data at object-time). The USAGE type of an intermediate result for the exponentiation operator is the same USAGE type as its first operand. Moreover, the only unary operator which requires an intermediate result field is the unary negate operator. In this case, the USAGE type of its intermediate result is the same as its singular operand. The unary plus operator is essentially a "no-op" operator and, thus, is ignored at compile-time and has no impact at object-time.

The process of determining the final number of integer places I(IR(x)) and the final number of decimal places D(IR(x)) to be maintained for an intermediate result field IR(x) resulting from an arithmetic expression operator x is conceptually the five step procedure outlined in Figure 4-9 below. In computing the final values for I(IR(x)) and D(IR(x)), remember that these values are subject to the following criterion:

$$1 <= I(IR(x)) + D(IR(x)) <= 18.$$

That is, a maximum precision of 18 decimal digits is maintained for an intermediate result field.

```
┌─────────────────────────────────────────────────────────────────┐
│  1.  Record the largest number of integer places, IMAX, declared │
│      for any source operand in an entire COBOL expression.        │
│                                                                   │
│  2.  Record the largest number of decimal places, DMAX, declared │
│      for any source operand in an entire COBOL expression.        │
│                                                                   │
│  3.  Compute the number of integer places, I(x), and number of   │
│      decimal places, D(x), for an arithmetic expression operator x│
│      as a function of the operands associated with operator x.    │
│                                                                   │
│  4.  Using IMAX, DMAX, I(x), and D(x) computed above, apply the   │
│      following criterion and redefinition:                        │
│                                                                   │
│      a.  If IMAX > I(x) then redefine I(x) = IMAX.                │
│                                                                   │
│      b.  If DMAX > D(x) then redefine D(x) = DMAX.                │
│                                                                   │
│  5.  Using the (possible redefined) values of I(x) and D(x) from  │
│      step 4, apply truncation criterion to determine the final    │
│      values I(IR(x)) and D(IR(x)).                                │
└─────────────────────────────────────────────────────────────────┘
```

Figure 4-9 Procedure to Determine I(IR(x)) and D(IR(x))
for an Arithmetic Expression Result Field IR

Before pursuing the procedure outlined in Figure 4-9 in more depth, the following notational conventions are adopted to facilitate the subsequent discussion:

x	–	a COBOL expression operator from the set +,-,*,/,**, and unary -.
IR(x)	–	Intermediate result field obtained from the execution of an arithmetic operation x.
OP1(x)	–	First operand in arithmetic operation x.
OP2(x)	–	Second operand in arithmetic operation x.
I(OP1(x)	–	Number of integer places declared for OP1 in arithmetic operation x.
D(OP1(x))	–	Number of decimal places declared for OP1 in arithmetic operation x.
I(OP2(x))	–	Number of integer places declared for OP2 in arithmetic operation x.
D(OP2(x))	–	Number of decimal places declared for OP2 in arithmetic operation x.
IMAX	–	Maximum of number of integer places for any source operand (except for exponents) in a COBOL expression.
DMAX	–	Maximum number of decimal places for any source operand (except for exponents) in a COBOL expression.
I(x)	–	Number of integer places for an arithmetic expression operator x computed as a function of the operator's source operands.

D(x) — Number of decimal places for an arithmetic expression operator x computed as a function of the operator's source operand(s).

I(IR(x)) — Final number of integer places to be maintained for an intermediate result IR obtained by the execution of operator x.

D(IR(x)) — Final number of decimal places to be maintained for an intermediate result IR obtained by the execution of operator x.

To determine the values for IMAX and DMAX (i.e., steps 1 and 2, Figure 4-9), the compile-time software inspects the operands of arithmetic operators and all data-name references which are compared to an arithmetic expression in relation conditions. In the process of inspecting the operands of arithmetic expressions, the software ignores OP2 of the exponentiate (**) operator. Moreover, since the software essentially "forgets" the presence of an unary plus operator, its singular operand has no role in the determination of IMAX and DMAX. Having determined the values of IMAX and DMAX, the software then iteratively applies steps 3 through 5 of the procedure to each arithmetic operator in an arithmetic expression. The algorithms for computing I(x) and D(x) are summarized below:

Unary Negate (x = "-")

    I(unary -) = I(OP1(unary -))

    D(unary -) = D(OP1(unary -))

Addition (x = "+")

    I(+) = MAX(I(OP1(+)), I(OP2(+))) + 1

    D(+) = MAX(D(OP1(+)), D(OP2(+)))

Subtraction (x = "-")

    I(-) = MAX(I(OP1(+)), I(OP1(-))) + 1

    D(-) = MAX(D(OP1(-)), D(OP1(-)))

Multiplication (x = "*")

    I(*) = I(OP1(*)) + I(OP2(*))

    D(*) = D(OP1(*)) + D(OP2(*))

Division (x = "/")

    I(/) = I(OP1(/)) + D(OP2(/))

    D(/) = MAX(D(OP1(/)), D(OP2(/))) + 1

Exponentiation (x = "**")

Case 1: OP2(**) is a data-name exponent:

I(**) = I(OP1(**)) * F(I(OP2(**)))

where:

I(OP2(**))	F(I(OP2(**)))
0 OR 1	9
> 1	18

D(**) = DMAX

Case 2: OP2(**) is a numeric integer literal exponent

I(**) = I(OP1(**)) * OP2(**)

D(**) = D(OP1(**)) * OP2(**)

With the values of I(x), D(x) IMAX, and DMAX known, the software applies step 4 of Figure 4-9 to possibly redefine the values of I(x) and D(x) in light of the known values of IMAX and DMAX. The purpose of step 4 is to ensure that the object-time software will maintain sufficient precision for intermediate field IR resulting from an arithmetic operation in the context in which that arithmetic operation occurs. Finally, step 5 applies the truncation criterion specified in Figure 4-9 to determine the final values of I(IR(x)) and D(IR(x)).

I(x) + D(x)	D(x)	I(x) + DMAX	Compiler Action
<18 =18	Any Value	Any Value	I(IR(x)) = I(x) D(IR(x)) = D(x)
>18	< DMAX	Any	I(IR(x) = 18 - D(x) [High order trunc]
	= DMAX	Value	D(IR(x)) = D(x)
	>DMAX	<18	I(IR(x) = I(x)
		=18	D(IR(x)) = 18 - I(x) [low order trunc]
		>18	I(IR(x)) = 18 - DMAX [high order trunc] D(IR(x)) = DMAX [Low order trunc]

Figure 4-10
Truncation Criterion and I(IR(x)) and D(IR(x)) Computation

4.6.3 Example of Intermediate Result Fields

Figure 4-11 illustrates the application of the conceptual algorithms for computing the attributes of intermediate result fields.

```
    •

    •

    •

    01    A    PIC    999V99        USAGE  IS  DISPLAY.

    01    B    PIC     99V9         USAGE  IS  COMPUTATIONAL.

    01    C    PIC    99V999        USAGE  IS  DISPLAY.

    •

    •

    •

          IF  A + 2 * B = C GO TO TAG-1.

    •

    •

    •
```

Figure 4-11
Example of Intermediate Results

The IF statement given in Figure 4-11 contains a relation condition which specifies the comparison of an arithmetic expression to a data-name. The arithmetic expression "A + 2 * B" gives rise to the creation of the following intermediate result fields:

```
    IR'(*) = 2 * B
    IR"(+) = A + IR'(+)
```

where the "primes" indicate the order in which the intermediate result fields are created. The major problem here is to determine the USAGE types of IR' and IR", respectively, and to determine the values of I(IR'(*)), D(IR'(*)), I(IR''(+)), and D(IR''(+)).

To begin the development of solution, we observe that the USAGE of IR'(*) is COMPUTATIONAL since OP2(*) = B has a COMPUTATIONAL USAGE. Then, it follows that the USAGE of IR"(+) is COMPUTATIONAL since OP2(+) = IR'(*) has a COMPUTATIONAL USAGE type. Now, before applying the five step procedure in Figure 4-9, the following conditions are obtained from the declarations of A,B,C, and the specification of the IF statement in Figure 4-11:

    I(OP1(*)) = 1 and D(OP1(*)) = 0 for OP1(*) =2,

    I(OP2(*)) = 2 and D(OP2(*)) = 1 for OP2(*) = B,

    I(OP1(+)) = 3 and D(OP1(+) = 2 for OP1(+) = A,

    I(C) = 2 and C(C) = 3 for the C dataname reference.

It should be noted, that since OP2(+) = "2 * B" is not a data name or literal reference, OP2(+) does not enter into the specification of the initial conditions, and therefore into the computation of the values for IMAX and DMAX. Further, we note the values of I(C) and D(C) for the dataname C reference since the relation condition involves the comparison of an arithmetic expression to a data name reference. The values of I(C) and D(C) are needed for the subsequent calculation of IMAX and DMAX for the COBOL expression "A + 2 * B = C".

Given these initial conditions, now apply step 1 and 2 of Figure 4-3 to calculate the values of IMAX and DMAX:

    IMAX = MAX(I(OP1(+)), I(OP1(*)), I(OP2(*)), I(C))

    IMAX = MAX(3,1,2,2)

    IMAX = 3

    DMAX = MAX(D(OP1(+)), D(OP1(*)), D(OP2(*)), D(C))

    DMAX = MAX(2,0,1,3)

    DMAX = 3

Therefore, from the application of steps 1 and 2, IMAX = 3 and DMAX = 3 for the COBOL expression "A + 2 * B = C". Next, iteratively apply steps 3-5 to each arithmetic expression operator (in the order in which the expression is evaluated at run time) to determine the values of I(IR'(*)), D(IR'(*)), I(IR''(+)), and D(IR''(+)). Thus, applying step 3 to OP1(*) and OP2(*), we determine the following values for I(*) and D(*):

    I(*) = I(OP1(*)) + I(OP2(*))

    I(*) = 1 + 2

    I(*) = 3

    D(*) = D(OP1(*)) + D(OP2(*))

    D(*) = 0 + 1

    D(*) = 1

With I(*) = 3 and D(*) = 1 determined from step 3, now apply step 4 to discover that I(*) = 3 and D(*) is redefined to 3 since DMAX > D(*) = 1. Thus, step 4 yields the following values:

    I(*) =3

    D(*) = 3

Finally, apply step 5 to the values of I(*) and D(*) to yield

    I(IR'(*)) = 3

    D(IR'(*)) = 3

since

    I(*) + D(*) < 18.

Thus, the object-time software will maintain three integer places and three decimal places for IR'(*) resulting from the evaluation of "2 * B".

We wish to apply steps 3-5 to OP1(+) and OP2(+) to determine the values for I(IR''(+)) and D(IR''(+)).   Note that, since OP2(+) = IR'(*), we have the following values for I(OP2(+)) and D(OP2(+)):

$$I(OP2(+)) = I(IR'(*))$$

$$I(OP2(+)) = 3$$

$$D(OP2(+)) = D(IR'(*))$$

$$D(OP2(+)) = 3.$$

With I(+) = 4 and D(+) = 3 determined from step 3 of Figure 4-9, now apply step 4 to find that I(+) = 4 and D(+) = 3 remain unchanged since IMAX < I(+) and DMAX = D(+). Finally, apply step 5 to the values of I(+) and D(+) to yield:

$$I(IR''(+)) = 4$$

$$D(IR''(+)) = 3$$

since

$$I(+) + D(+) < 18.$$

Hence, the object-time software will maintain four integer places and three decimal places for IR''(+) resulting from the evaluation of "A + 2 * B = C".

CHAPTER 5

TABLE HANDLING

5.1 INTRODUCTION

With COBOL, as with any other language, any data item to which the
program refers must be uniquely identified. This unique
identification of data items is usually accomplished by assigning a
unique name to each item. However, in many applications this is
tedious and inconvenient; often programs require too many names for
items that have different names but contain the same type of
information. Tables provide a simple solution to this problem.
PDP-11 COBOL includes full table handling capabilities as outlined for
standard COBOL in the 1974 ANSI Standards.

A table is a repetition of one item (element) in memory. This
repetition is accomplished by the use of the OCCURS clause in the data
description entry. The literal value in the OCCURS clause causes the
software to duplicate the data description entry as many times as
indicated by that value, thus creating a matrix or table.

The elements may be initialized with the VALUE clause or with a
procedural instruction. They may contain synchronized or
unsynchronized data. They may be accessed only with subscripted
procedural instructions. A subscript is a parenthesized integer or
data name (with an integer value). The integer value represents the
desired occurrence of the element.

This chapter discusses how to set up tables and access them accurately
and efficiently. It attempts to cover any problems that may be
encountered while handling tables. Read it through carefully before
setting up tables with PDP-11 COBOL. Section 6 of the PDP-11 COBOL
Reference Manual for the PDP-11 contains reference information on the
individual table handling instructions (OCCURS, USAGE IS INDEX, SET,
and SEARCH).

5.2 DEFINING TABLES

To define a table with PDP-11 COBOL, simply complete a standard data
description for one element of the table and follow it with an OCCURS
clause. The OCCURS clause contains an integer which dictates the
number of times that element will be repeated in memory, thus creating
a table.

The OCCURS phrase has two formats:

Format 1

OCCURS integer-2 TIMES

```
┌──────────────────────────────────────────────────────────────────────────┐
│ ⎡ ⎧ASCENDING ⎫                                             ⎤              │
│ ⎢ ⎨          ⎬  KEY IS data-name-2  [, data-name-3]  ...  ⎥   ...        │
│ ⎣ ⎩DESCENDING⎭                                             ⎦              │
│                                                                          │
│        ⎡ INDEXED BY index-name-1  [, index-name-2] ... ⎤                │
└──────────────────────────────────────────────────────────────────────────┘
```

```
┌──────────────────────────────────────────────────────────────────────────┐
│ Format 2                                                                 │
│                                                                          │
│ OCCURS integer-1 TO integer-2 TIMES DEPENDING ON data-name-1            │
│                                                                          │
│ ⎡ ⎧ASCENDING ⎫                                             ⎤             │
│ ⎢ ⎨          ⎬  KEY IS data-name-2  [, data-name-3] ...   ⎥   ...       │
│ ⎣ ⎩DESCENDING⎭                                             ⎦             │
│                                                                          │
│       ⎡ INDEXED BY index-name-1  [, index-name-2] ... ⎤                 │
└──────────────────────────────────────────────────────────────────────────┘
```

In either format, the system generates a buffer large enough to accommodate integer-2 occurrences of the data description. Therefore, the amount of storage allocated in either case is equal to the amount of storage required to repeat the data entry integer-2 times.

The software will automatically map the elements into memory. When mapping a table into memory, the software follows the rules for mapping which depend on whether the element contains synchronized items or not. If they do not contain synchronized items, the software maps them into adjacent memory locations and the size of the table can be easily calculated by multiplying the size of the element times the number of occurrences (5X10 for the table illustrated in Figure 5-1, or 50 bytes of memory).

```
┌──────────────────────────────────────────────────────────┐
│   01 A-TABLE                                             │
│      03 A-GROUP PIC X(5) OCCURS 10 TIMES.                │
└──────────────────────────────────────────────────────────┘
```

Figure 5-1
Defining a Table

5.2.1 The OCCURS Phrase - Format 1

When Format 1 is used, a fixed length table is generated, whose length (number of occurrences) is equal to the value specified by integer-2. This format is useful for storing large amounts of frequently used reference data whose size never changes. Tax tables, used in payroll deduction programs, are an excellent example of where a Format 1 (fixed length) table might be used.

5.2.2 The OCCURS Phrase - Format 2

Format 2 is used to generate variable length tables. When used, a table whose length (number of occurrences) is equal to the value specified by data-name-1 is generated.

NOTE

Data-name-1 must always be a positive
integer whose value is equal to or
greater than integer-1 but not greater
than integer-2.

Unlike format 1 tables, the number of occurrences of data items in
format 2 tables can be dynamically expanded or reduced to satisfy user
needs.

By generating a variable length table, the user is, in effect, saying;
"build me a table that can contain at least integer-1 occurrences, but
no more than integer-2 occurrences, and set its number of occurrences
equal to the value specified by data-name-1".

Data-name-1 always reflects the number of occurrences available for
user access. To expand the size (number of occurrences available for
use) of a table, the user need only increase the value of data-name-1
accordingly.

Likewise, reducing the value in data-name-1 will reduce the number of
occurrences available for user access.

5.3 MAPPING TABLE ELEMENTS

As mentioned in Section 5.2, when the software detects an OCCURS
clause in an unsynchronized item, it maps the table elements into
adjacent locations in memory. Consider the following data description
of a simple table and the way it is mapped into memory:

Table Description: 01 A-TABLE.
                      03 A-GROUP PIC X(5) OCCURS 10 TIMES.

Memory Map:

| words | I | II | III | IV | V | VI | VII | VIII | IX | X | ... |

bytes

A-GROUP   A-GROUP   A-GROUP   A-GROUP   ...
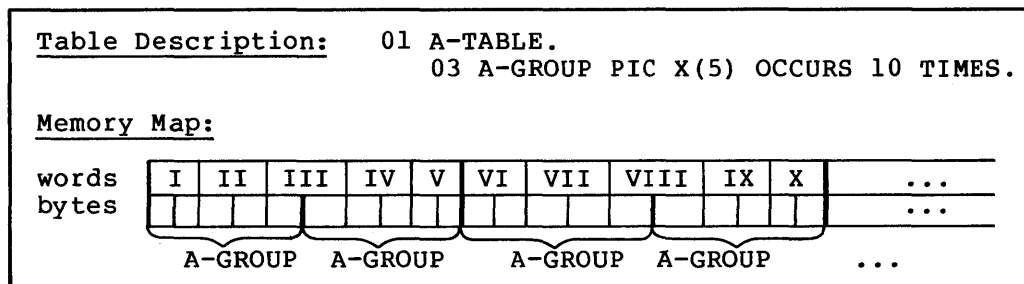
Figure 5-2
Mapping a Table into Memory

The data description in Figure 5-2 causes the software to set up ten
items of five bytes each (elements) and place them in adjacent
ascending memory locations for a total of 50 character positions, thus
creating a table. Since the length of each A-GROUP element is odd
(5), the memory addresses of each subsequent element will alternate
between odd and even locations.

The SYNCHRONIZED clause causes the software to add a fill byte to
items that contain an odd number of bytes, thereby making the number
of bytes in that item even. This ensures that each subsequent
occurrence of the element will not alternate between odd and even
addresses, but will map the same (odd or even) as the first repetition
of that element.

If the data description of A-GROUP contained a SYNCHRONIZED clause, the software would map it quite differently. If A-GROUP were synchronized, it would expand its length to three words. The item will, by default, be synchronized to the left occupying the first five characters of the three words. The software supplies a padding character to fill out the third word. This padding character is not a part of the A-GROUP element and table instructions referring to A-TABLE will not detect the presence or absence of the character.

The padding character does, however, affect the overall length of the group item and, hence, the table. Without the SYNCHRONIZED clause, A-TABLE required only 50 character positions; now, with the clause, it requires 60 character positions. (This length includes the last padding character -- following the tenth element in the table.)

Although the SYNCHRONIZED clause has little value when used with alphanumeric fields, an understanding of the concept is essential before attempting to use COMP and INDEX data items in tables. The software automatically synchronizes all COMP and INDEX usage data items, and will most probably alter the size of any table (often drastically) that contains these data types. Consider the following illustration of a synchronized data item being mapped by the software:
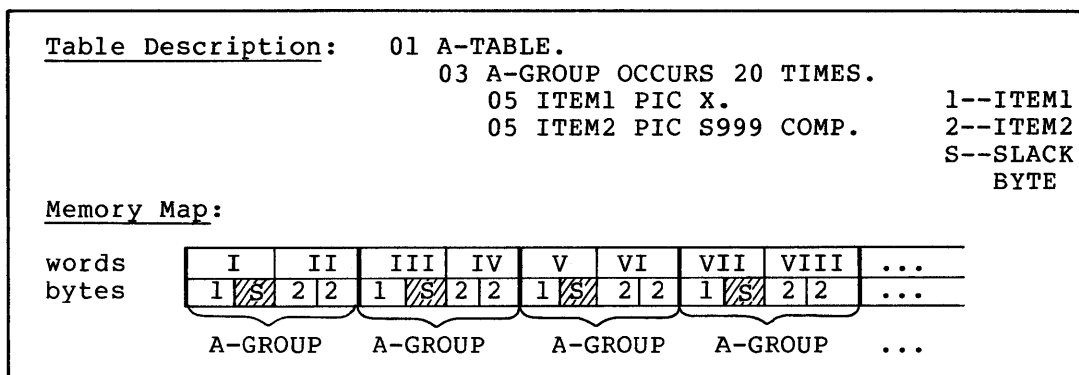
Figure 5-3
Synchronized COMP Item in a Table

Since the software synchronizes the ITEM2 fields (COMP), these fields each occupy a single word in memory; thus, a slack byte follows each occurrence of ITEM1. Each repetition of A-GROUP consumes four bytes of memory -- one byte for ITEM1, one byte for the slack byte, and two bytes for ITEM2. A-TABLE, then, requires 80 bytes of memory (20 elements of four bytes each).

Now, consider the effect of adding a 1-byte field to A-TABLE. If we place the field between ITEM1 and ITEM2, it will take the space formerly occupied by the slack byte. This has the effect of adding a data byte but leaving the size of the table unchanged. Consider the following illustration:

```
Table Description:        01 A-TABLE.
                            03  A-GROUP OCCURS 20 TIMES.
                              05 ITEM1 PIC X.
                              05 ITEM3 PIC X.
                              05 ITEM2 PIC S999 COMP.


                                                          1--ITEM1
Memory Map:                                               2--ITEM2
                                                          3--ITEM3
words     | I   | II  | III | IV  | V   | VI  |...
bytes     |1|3  |2|2  |1|3  |2|2  |1|3  |2|2  |...
            \___/   \___/   \___/
          A-GROUP A-GROUP A-GROUP
```
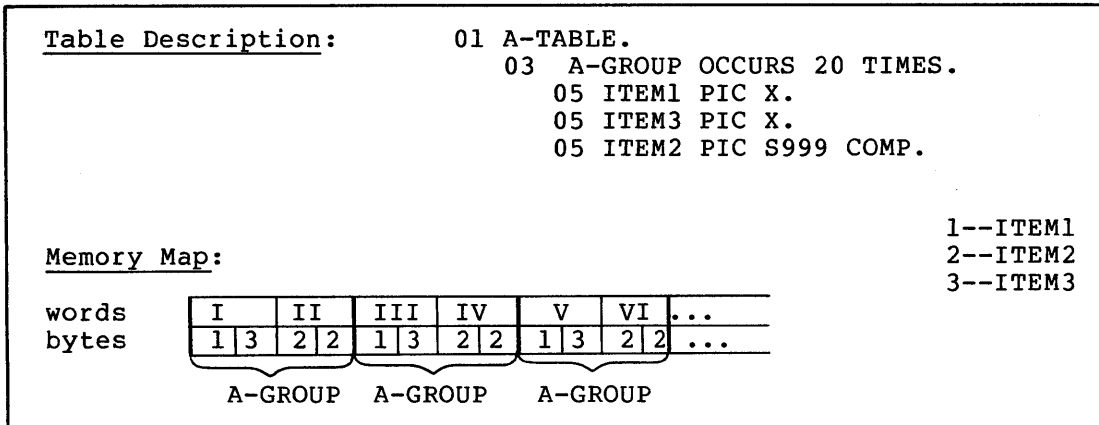
Figure 5-4
Adding a Field without Altering the Table Size

If, however, we place the 1-byte field after ITEM2, it has the effect
of adding its own length plus another slack byte. Now, each element
requires six full bytes and the complete table consumes 120 bytes of
memory (6X20)! This is due to the fact that the first repetition of
ITEM1 falls on an even byte and, in order to keep the mapping of each
A-GROUP element the same, the software allocates each successive
repetition of ITEM1 to an even byte address. Thus, it assigns ITEM3
to the even byte of the third word and adds a slack byte to guarantee
that the next element begins on an even byte. Consider the following
illustration:

```
Table Description:        01 A-TABLE.
                            03 A-GROUP OCCURS 20 TIMES.
                              05 ITEM1 PIC X.
                              05 ITEM2 PIC S999 COMP.
                              05 ITEM3 PIC X.

Memory Map:

Odd or Even  E O E O E O E O E O E O E O E O E O      ...
words        | I  | II |III |IV  | V  | VI |VII |VIII| IX |  ...
bytes        |1|S/|2|2 |3|S/|1|S/|2|2 |3|S/|1|S/|2|2 |3|S/|  ...
               _____/      _____/      _____/
             A-GROUP       A-GROUP       A-GROUP
```
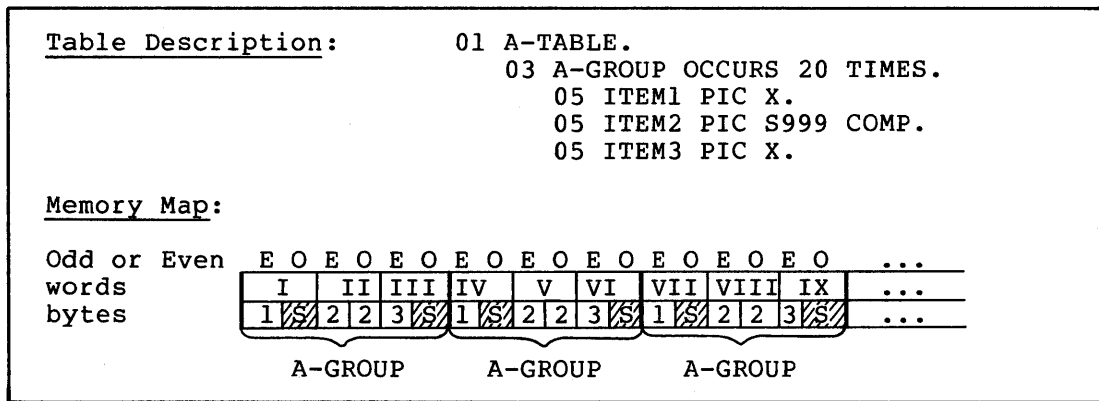
Figure 5-5
Adding One Byte which Adds Two Bytes
to the Element Length

NOTE

The illustrations in this section show
each byte with an even address (E) as
the leftmost byte, and each byte with an
odd address (O) as the rightmost byte.
(The two bytes, odd and even, are
reversed in actual memory.)

If, however, we use a FILLER byte to force the first allocation of ITEM1 to occur on an odd byte, A-GROUP again requires only four bytes and no slack bytes. Figure 5-6 illustrates this. Since the FILLER item occupies the even byte of the first word, ITEM1 falls on an odd byte. The software requires that each repetition of ITEM1 must be an even number of bytes in length in order to guarantee that the synchronized item(s) will map onto word boundaries. No slack bytes are needed and A-GROUP elements are again only four bytes long, and A-TABLE requires only 81 bytes.

```
Table Description:        01 A-TABLE.
                             03 FILLER PIC X.
                             03 A-GROUP OCCURS 20 TIMES.
                                05 ITEM1 PIC X.
                                05 ITEM2 PIC S999 COMP.
                                05 ITEM3 PIC X.

Memory Map:

odd or even   E O   E O   E O   E O   E O   E O   E O   ...
words       |  I  |  II |  III|  IV |  V  |  VI | VII |  ...
bytes       |F|1| 2 |2| 3|1| 2 |2| 3|1| 2 |2| 3|  ...

        FILLER     A-GROUP    A-GROUP    A-GROUP
```
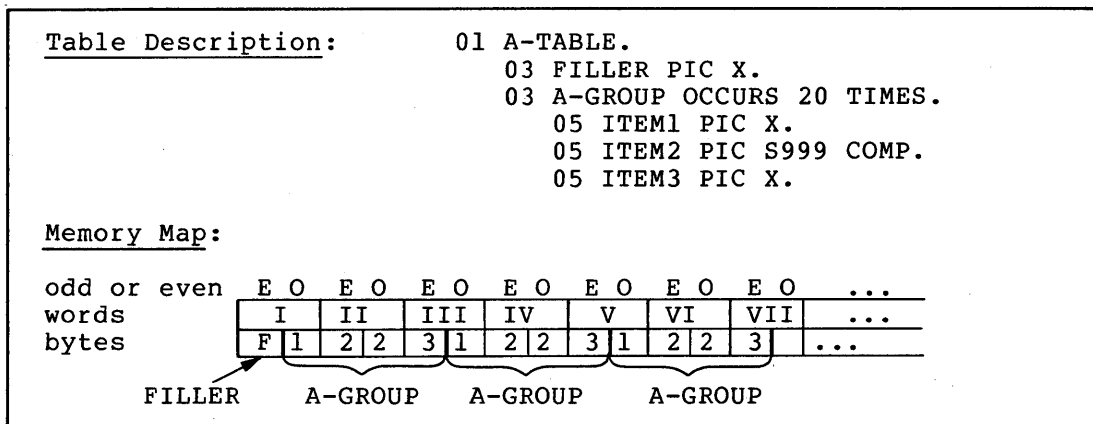
Figure 5-6
Forcing an Odd Address By Adding a 1-Byte FILLER
Item to the Head of the Table

If we try to force ITEM1 onto an odd byte with a SYNCHRONIZED RIGHT clause, the software maps ITEM1 into the odd byte, but prohibits all repetitions of the element from using the even byte. Thus, the first repetition of A-GROUP has a slack byte at its beginning and, so that the next element can begin (with a slack byte) at an even address, another slack byte (odd) following ITEM3. This expands the element length to six bytes and the table length to 120 bytes.

```
Table Description:        01 A-TABLE.
                             03 A-GROUP OCCURS 20 TIMES.
                                05 ITEM1 PIC X SYNCHRONIZED RIGHT.
                                05 ITEM2 PIC S999 COMP.
                                05 ITEM3 PIC X.

Memory Map:

Odd or Even   E O   E O   E O   E O   E O   E O   E O   E O   E O   ...
words       |  I  |  II | III |  IV |  V  |  VI | VII |VIII | IX  |  ...
bytes       |/S/|1| 2 |2| 3|S/|/S/|1| 2 |2| 3|S/|/S/|1| 2 |2| 3|S/|  ...

              A-GROUP       A-GROUP           A-GROUP    ...
```
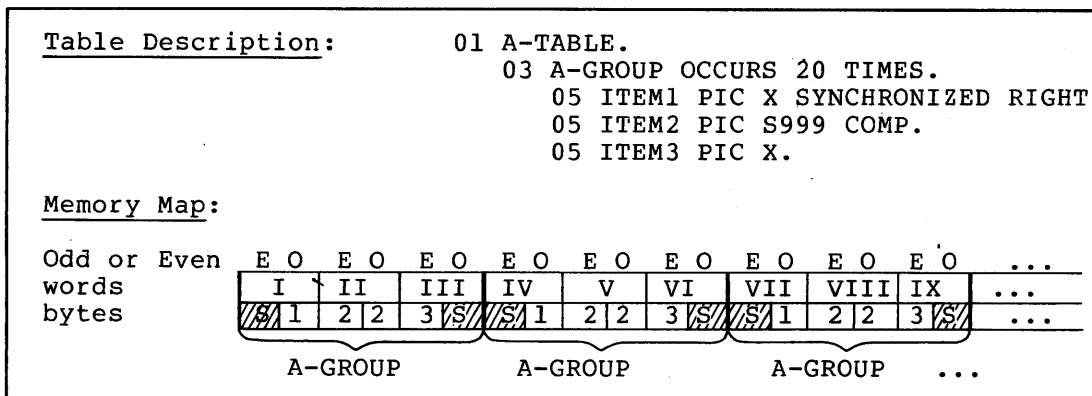
Figure 5-7
The Effect of a SYNCHRONIZED RIGHT Clause Instead
of a FILLER Item as shown in Figure 5-6

To determine how the software will map a given table, apply the following two rules:

1. The software maps all items in the first repetition of a table element into memory words as with any item properly defined with a data description, obeying any implicit or explicit synchronization requirements.

2. If the first repetition contains any elementary items with implicit or explicit synchronization, the software maps each successive repetition of the element into memory words in the same way as the first repetition. It does this by adding one slack byte, if necessary, to make the size of the element even.

5.3.1 Initializing Tables

If a table contains only DISPLAY items, it can be set to any desired initial value (initialized). To initialize a table, simply specify a VALUE phrase on the record level preceding the item containing the OCCURS clause. The sample data definitions, below, will set up initialized tables:
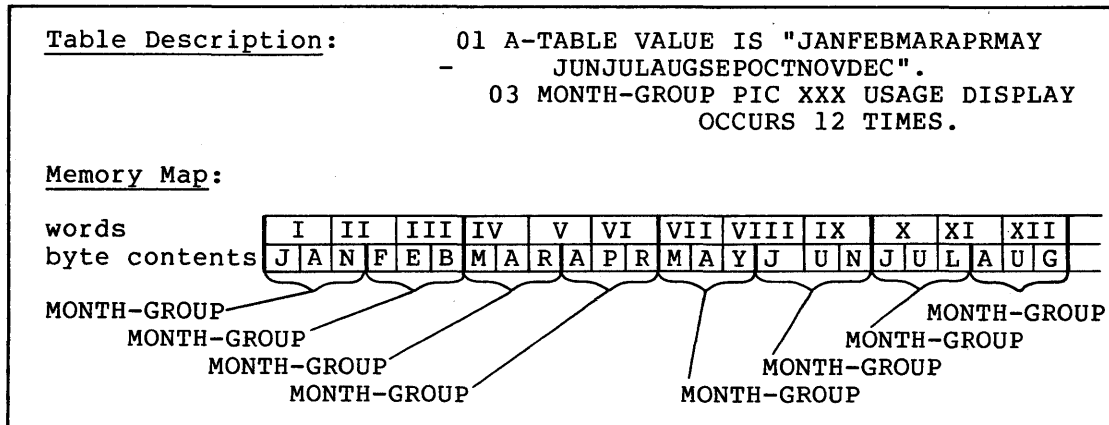
Figure 5-8
Initializing Tables

Often a table is too long to initialize with a single literal, or it contains items that cannot be initialized (numeric, alphanumeric, or COMP). These items can be individually initialized by redefining the group level preceding the level that contains the OCCURS clause. Consider the following sample table descriptions:

```
Table Description:          01 A-RECORD-ALT.
                               05 FILLER PIC XX VALUE "AX".
                               05 FILLER PIC 99 COMP VALUE 1.
                               05 FILLER PIC XX VALUE "BX".
                               05 FILLER PIC 99 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.

Memory Map:                        Binary 1  Binary 2

words                         |  I | II | III|IV |  ...
byte contents at              | A| X|  |  |  |  |    B  X  ...
  initialization time           A-GROUP   A-GROUP
```
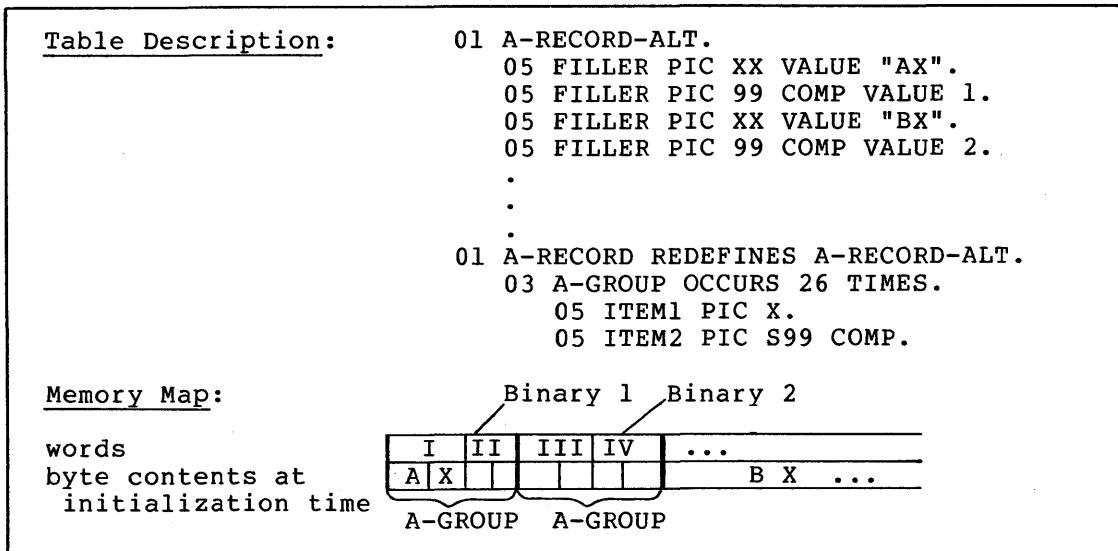
Figure 5-9
Initializing Mixed Usage Fields

In the preceding example, the slack bytes in the alphanumeric fields
(ITEM1) are being initialized to X.

```
Table Description:          01 A-RECORD-ALT.
                               03 FILLER PIC X(30) VALUE IS
                                  "AAAAAAAAAABBBBBBBBBBCCCCCCCCCC".
                               03 FILLER PIC X(30) VALUE IS
                                  "DDDDDDDDDDEEEEEEEEEEFFFFFFFFFF".
                            (etc.)

                            01 A-RECORD REDEFINES A-RECORD-ALT.
                               03 ITEM1 PIX X(10) OCCURS 26 TIMES.

Memory Map:

word           | I | II | III|IV |  V |VI |VII |VIII| IX |  X |XI |  ...
byte           |A|A|A|A|A|A|A|A|A|A|B|B|B|B|B|B |B|B|B|B|C|C|  ...
contents at
initialization      ITEM1                 ITEM1               ...
time
```
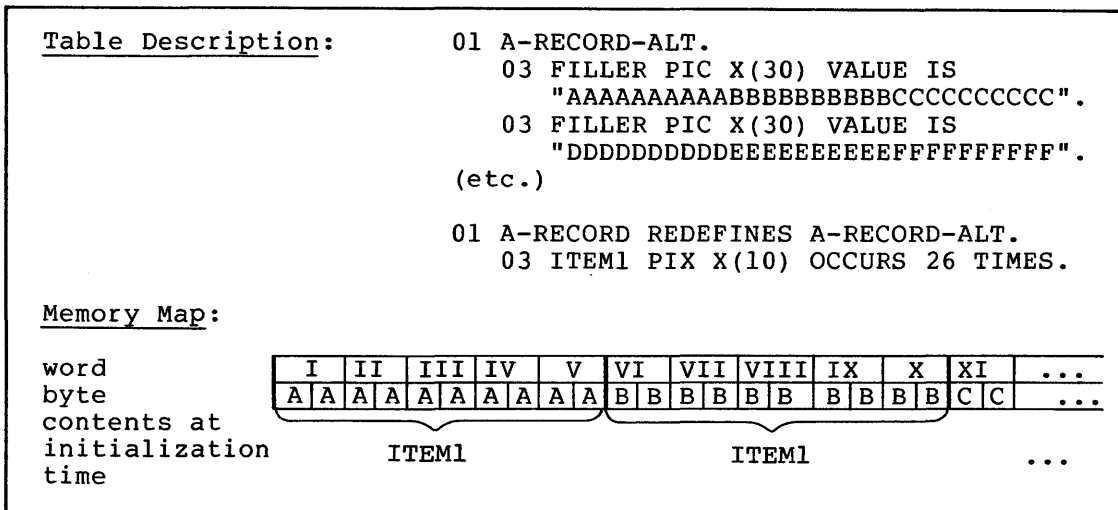
Figure 5-10
Initializing Alphanumeric Fields

In the preceding example, each FILLER item initializes three 10-byte
table elements.

When redefining or initializing table elements, allow space for any
slack bytes that may be added due to synchronization (implicit or
explicit). The slack bytes do not have to be initialized; however,
they may be and, if initialized to an uncommon value, they may even
serve as a debugging aid for situations such as a statement referring
to the record level above the OCCURS clause or another record
redefining that level.

Sometimes the length and format of table items are such that they would best be initialized by statements in the Procedure Division. This initialization coding could be executed once and then overlaid (due to the automatic segmentation feature) if the entire Procedure Division is too large to be held in memory at one time.

Once the OCCURS clauses have established the necessary tables, the program must be able to access the elements of those tables individually. Subscripting and indexing are the two methods provided by COBOL for accessing individual elements.

5.4 SUBSCRIPTING AND INDEXING

To refer to a particular element within a table, simply follow the name of the desired element with a parenthesized subscript or index. A subscript is an integer or a data-name that has an integer value; the integer value represents the desired occurrence of the element -- an integer value of 3, for example, refers to the third occurrence of the element. An index is a data-name that has been named in an INDEXED BY phrase in the OCCURS clause.

5.4.1 Subscripting with Literals

A literal subscript is simply a parenthesized integer whose value represents the occurrence number of the desired element. In figure 5-11, the literal subscript in the MOVE instruction (2) causes the software to move the contents of the second element of the table, A-TABLE, to I-RECORD.

Table Description	01 A-TABLE.
	   03 A-GROUP PIC X(5)
	                  OCCURS 10 TIMES.

Procedural Instruction	MOVE A-GROUP(2) TO I-RECORD.

Figure 5-11
Literal Subscripting

If the table has more than one level (or dimension), follow the name of the desired 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 item is subordinate. (This is the most encompassing level -- A-GROUP in the following example.) The second subscript in the list applies to the next most encompassing level, and the last subscript applies to the lowest level OCCURS clause being accessed (or the desired occurrence number of the item named in the procedural instruction -- ITEM5 in the following example).

Consider Figure 5-12; the subscripts (2,11,3) in the MOVE instruction cause the software to 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) would refer to the first occurrence of ITEM5 in the table and ITEM5(5,20,4) would refer to the last occurrence of ITEM5.)

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.

Figure 5-12
Subscripting a Multi-Dimensional Table

NOTE

Since ITEM5 is not subordinate to ITEM2,
an occurrence number for ITEM2 is not
permitted in the subscript list.

Figure 5-13 summarizes the subscripting rules for each of the above
items and shows the size of each field in bytes.

NAME OF FIELD	NUMBER OF SUBSCRIPTS REQUIRED TO REFER TO THE NAMED FIELD	SIZE OF FIELD
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 slack byte		

Figure 5-13
Subscripting Rules for a
Multi-Dimensional Table

5.4.2 Operations Performed by the Software

When a literal subscript is used to refer to an item in a table, the
software performs the following steps to determine the exact address
of the item:

1. The compiler converts the literal to a 1-word binary value.

2. The compiler range checks the subscript value (the value must
    not be less than 1 nor greater than the number of repetitions
    specified by the OCCURS clause) and prints a diagnostic
    message if the value is out of range.

3. The compiler decrements the value of the subscript by 1 and
    multiplies it by the size of the item that contains the
    OCCURS clause corresponding to this subscript, thus forming
    an index value; it then stores this value, plus the literal
    subscript, in the object program.

4. At object execution time for a fixed length table, the run time system adds the index value (from 3 above) to a base address, thus determining the address of the desired item. For a variable length table reference, the procedure for fixed length tables is preceded by the procedure described in Section 5.4.6.

5.4.3 Subscripting with Data-Names

As discussed earlier in this section, subscripts may also be specified using data-names instead of literals. To use a data-name as a subscript, simply define it as a numeric integer (COMP or DISPLAY). It may be signed, but the sign must be positive at the time it is used as a subscript.

The sample subscripts in figure 5-14 refer to the same element accessed in Figure 5-12, (2, 11, 3).

Data Descriptions of Subscript data-names	01 KEY1 PIC 99 USAGE DISPLAY. 01 KEY2 PIC 99 USAGE COMP. 01 KEY3 PIC S99.

Procedural Instructions	MOVE 2 TO KEY1.
	MOVE 11 TO KEY2.
	MOVE 3 TO KEY3.
	    GO TO TABLERTN.
	TABLERTN.
	    MOVE ITEM5(KEY1 KEY2 KEY3) TO
	                I-FIELD5.

Figure 5-14
Subscripting with Data-Names

5.4.4 Operations Performed by the OTS

When a data-name subscript is used to refer to an item in a table, the OTS performs the following steps at object execution time:

1. If the data-name's data type is DISPLAY, the software converts it to a 1-word binary value.

2. For fixed length tables, the software range checks the subscript value (the value must not be less than 1 nor greater than the number of repetitions specified by the OCCURS clause) and terminates the object program (with a diagnostic message) if it is out of range. For variable length tables, the procedure described in Section 5.4.6 is followed.

3. The software decrements the value of the subscript by 1 and multiplies it by the size of the item that contains the OCCURS clause corresponding to this subscript, thus forming an index value.

4. The software adds the index value (from 3 above) to a base address, thus determining the address of the desired item.

5.4.5 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.

An index-name item (an item named in the INDEXED BY phrase of the OCCURS clause) has the ability to hold an index value. (The index value is the product formed in step 3 of the operations performed by the software for literal or data-name subscripts -- the relative location, within the table, of the desired item.)

The compiler allocates a 2-part data item for each name that follows an INDEXED BY phrase. These index-name items cannot be accessed as normal data items; they cannot be moved about, redefined, written to a file, etc. However, the SET verb can change their values and relation tests can examine their values. One part of the 2-part index-name item contains a subscript value and the other part contains an index value. Consider the following illustration:
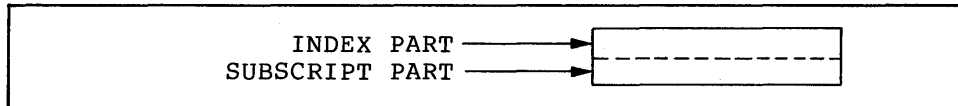
Figure 5-15
Index-Name Item

Whenever a SET statement places a new value in the subscript part, the software performs an index value computation and stores the result in the index part. Only the subscript part of the item acts as a sending or receiving field. The index part is never altered by any other operation and is never moved to another item. It is used only when the index-name is used as an index referring to a table item. The sample MOVE statement in Figure 5-16 would move the contents of the third repetition of A-GROUP to I-FIELD. (For illustration simplicity, once again, I-FIELD is not defined.)

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.

Figure 5-16
Subscripting With Index-name Items

5.4.6 Operations Performed by the OTS

The OTS performs the following steps when it executes the SET statement:

1. The OTS converts the contents of the sending field of the SET statement to a 1-word binary value.

2. The OTS range checks the value (the value must not be less than 1 nor greater than the number of repetitions specified in the OCCURS clause) and terminates the object program with a diagnostic message if it is out of range.

3. The OTS decrements the value by 1 and multiplies it by the size of the item that contains the OCCURS clause, thus forming an index value.

For fixed length tables, once the SET statement has been executed and the software has encountered the index-name item as an index, it only has to add the index value (from 3 above) to a base address to determine the address of the desired item. Since this is the only action performed, the execution speed of a procedural statement with an indexed data-name is equivalent to a reference with a literal subscript.

For a variable length table, when the index-name is encountered as an index, the procedure described in Section 5.4.6 is invoked before following the fixed length table logic. However, the SET statement itself is not impacted by the fixed/variable characteristic of the associated table.

PDP-11 COBOL initializes the value of all index-name items to a subscript value of 1 (index value of 0), hence an attempt to use an index-name item as an index before it has been the receiving field of a SET verb will not result in an out-of-range termination.

NOTE

Initialization of index-name items is an extension to the ANSI COBOL standards. Users concerned with writing COBOL programs that adhere to standard COBOL should not rely on this feature.

5.4.7 Relative Indexing

To perform relative indexing, when referring to a table item, simply follow the index-name with a plus or minus sign and an integer literal. Relative indexing, albeit easy to use, causes additional overhead to be generated each time a table item is referenced in this fashion. At compile time, the compiler has to compute the index value corresponding to the specified literal; and transfer this index value to the object file. At object run time, the index value for the literal is added to (+) or subtracted from (-) the index value of the index-name. The resulting index value is stored in a temporary location. The OTS adds this temporary index value to the base address of the table to determine the address of the desired table item. At this point, a range check is performed on the temporary index value to insure that the resulting index is within the permissible range for the table.

For fixed length tables, this index manipulation is relatively straightforward. The size of the table is known at compilation time, and this size is passed along to the OTS in the object file. A simple compare against this fixed value is all that is required to determine if a given index value is within the permissable range for the table.

For a variable length table, however, the process is more involved. The current number of occurrences (data-name-1) for the table must be determined and range checked; the index value corresponding to the current number of occurrences must be calculated; then the temporary index value must be range checked using the current number of occurrence's index value.

The object time overhead required for the relative indexing of variable length tables is significantly greater than that required for fixed length tables. In either case, the index portion of the index-name is not altered. If any of the range checks reveals an illegal (out of range) value, execution is terminated with an apropriate error message.

The sample MOVE instruction in Figure 5-17 moves the fourth repetition of A-GROUP to I-FIELD if IND-NAME has not been altered with a SET verb.

```
MOVE A-GROUP(IND-NAME + 3) TO I-FIELD.
```

Figure 5-17
Relative Indexing

The actual operation of accessing a table element is shorter at run time since the compiler has calculated the index value of the literal at compile time and has stored it in the object program ready for use. Relative indexing, therefore, involves two additions and a range check during object execution. It leaves the index-name item unaltered.

5.4.8 Index Data Items

Often a program will require that the value of an index-name item be stored outside of that item. It is for this purpose that PDP-11 COBOL provides the index data item.

Index data items are 1-word binary integers with implicit synchronization. (The 1-word size corresponds to the subscript part of the index-name item.) They must be declared with a USAGE IS INDEX phrase and they may be modified (explicitly) only by the SET statement.

```
Subscript Part ———————▶ [          ]
```
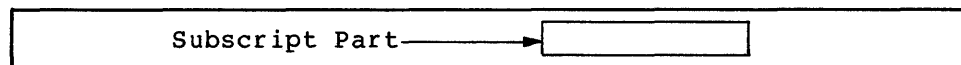
Figure 5-18
Index Data Item

Since index data items are considered to contain only the subscript part of an index-name item, when a SET statement "moves" an index-name item to an index data item, only the subscript part is moved. Likewise, when a SET statement "moves" an index data item to an index-name item, a new index value is computed by the software. This is done to guarantee that an index-name item will always contain a good index value.

The only advantage gained by using index data items over numeric, COMP items is that the data description is shorter, easier to write, and more self-documenting. Further, the restrictions placed on access to index items may be useful in debugging the program.

5.4.9 The SET Statement

The SET statement alters the value of index-name items and copies their value into other items. When used without the UP BY/DOWN BY clause, it functions like a MOVE statement. Figure 5-19 illustrates the legal data movements that the SET statement can perform.

```
                            INDEX-NAME ITEM
   NUMERIC LITERAL          (INDEX PART)          INDEX DATA ITEM
   ┌──────────────┐  ──────►(SUBSCRIPT PART)◄──── ┌──────────────┐
   └──────────────┘         └──────────────┘  ──► └──────────────┘

   NUMERIC DATA NAME                           INDEX-NAME ITEM
   (COMP OR DISPLAY)                           (INDEX PART)
   ┌──────────────┐                            (SUBSCRIPT PART)
   └──────────────┘
```
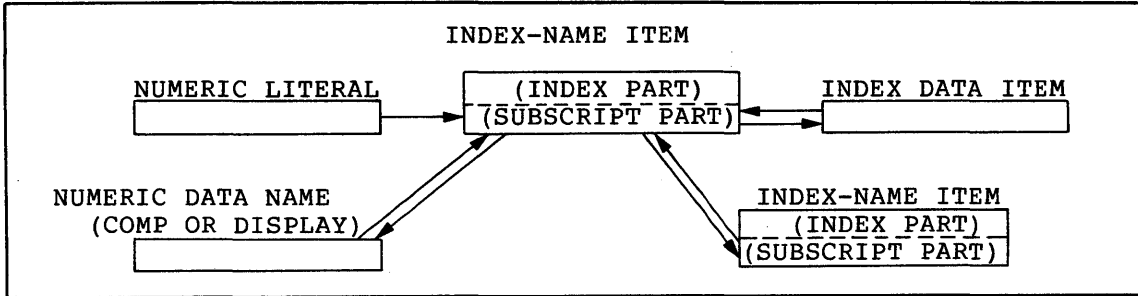
Figure 5-19
Legal Data Movement with the SET Statement

The SET statement may be used with the UP BY/DOWN BY clause to alter the value of an index-name item arithmetically. The numeric literal is added to (UP BY) or subtracted from (DOWN BY) the subscript part, and the index part is recalculated by the software after the appropriate range check against the number of repetitions for the table. The SET statement is not affected by whether the table is fixed or variable length.

5.4.10 Referencing a Variable Length Table Element at OTS Time

At OTS time, when a procedural reference involves an element in a variable length table, the following procedure is used:

1. Determine the number of occurrences in the table (the value contained in data-name-1), and verify its legality.

    (integer-1 <= data-name-1 <= integer-2)

2. Verify that the subscript is within the legal range.

    (subscript <= data-name-1)

If any of the above checks fails, execution is terminated with an appropriate error message.

5.4.11 Referencing a Dynamic Group at OTS Time

A dynamic group is defined as a group item that contains a subordinate item that is a variable length table. At OTS time, when a dynamic group is referenced, the following procedures are followed:

1. The number of occurrences of the subordinate variable length table is determined, and checked for legality; i.e., integer-1<=data-name-1<=integer-2. If this check fails, execution terminates and the appropriate error message is issued.

2. The size of the dynamic group is calculated. The number of occurrences of the variable length table (data-name-1) is multiplied by the size of one table entry. The resulting number is then added to the fixed size of the dynamic group.

TABLE HANDLING

NOTE

The fixed size of a dynamic group is the
size of the group up to but not
including the variable length table.

5.4.12 The SEARCH Verb

The SEARCH verb has two formats: Format 1, which performs a
sequential search of the specified table beginning with the current
index setting; and Format 2 which performs a selective (binary)
search of the specified table, beginning with the middle of the table.

Both formats allow the programmer to specify imperative statements
within the SEARCH verb. At OTS time, an imperative statement
contained within a search verb is executed only when one of the exit
paths (success or failure) is taken.

The failure path is defined either explicitly by the AT END statement,
in which case the imperative statement which follows it is executed;
or by default, in which case control is passed to the next procedural
sentence. In either case (success or failure), after an imperative
statement is executed, control is passed to the next procedural
sentence.

5.4.13 The SEARCH Verb - Format 1

Format 1 directs the OTS to search the indicated table sequentially.
The OCCURS clause for the table being searched must contain the
INDEXED by phrase. Unless otherwise specified in the SEARCH
statement, the first index is the controlling index for the table
search. The search begins with the current index setting, and
progresses through the table, augmenting the index by one as each
occurrence is interrogated. If any of the specified conditions is
true (success), the associated imperative statement is executed; the
search exits; and the index remains at the current setting.

If the possible number of occurrences for the table is exhausted
before any of the specified conditions are met, the specified failure
exit path is taken. That is, either the AT END exit path (if
specified) is taken, or control is passed to the next procedural
sentence.

Figure 5-20 contains an example of using the SEARCH verb to search a
table in a serially.

Associated with Format 1 is the optional VARYING phrase. This phrase
can be specified by using any of the following methods:

1. default - phrase omitted

2. VARYING index-name-n

3. VARYING identifier-2

4. VARYING index-name-2

5-16

NOTE

> The following is true regardless of which of the above methods is used.

a. An index name associated with the table is methodically augmented by one, by the OTS, for each cycle of the serial search. This controlling index, when compared to the allowable number of occurrences for the table, dictates the permissible range of search cycles at OTS time. When an exit occurs (success or failure), this index remains at the current setting.

b. The OTS will not initialize the index when the search begins. It is the programmers responsibility to insure that the initial index setting is the appropriate one. The OTS will begin processing the table with the setting it finds when the search is initiated.

When method 1 is used, the first index name (index-name-1) associated with the table is used as the controlling index. Only this index is set to consecutive values by the OTS serial search processor. See Figure 5-20, Example 2, for an example of using method 1.

When method 2 is used, index-name-n is any index that is associated with the table being searched. It becomes the controlling index for the table. It alone is set to consecutive values by the OTS search processor. See Figure 5-20, Example 3, for an example of using method 2.

When method 3 is used, identifier-2 is augmented by one each time the first index (controlling index) for the table is augmented by one. Identifier-2 is not a substitute index. It merely allows the programmer to maintain an additional pointer to elements within a table. See Figure 5-20, Example 4, for an example of method 3.

When method 4 is used, index-name-2 is an index that is associated with a table other than the one being searched. Each time the controlling index (1st index for the table) of the searched table is augmented, index-name-2 is also augmented. See Figure 5-20, Example 5.

5.4.14 The SEARCH Verb - Format 2

Format 2 is used to direct the OTS to search the indicated table selectively. The selective (binary) search is predicated upon the ASCENDING/DESCENDING KEY attributes of the table being searched. Therefore, an ASCENDING and/or DESCENDING KEY(s) must be specified in the OCCURS clause that defines the table, to inform the OTS that the keys are stored within the table in ascending or descending order.

The INDEXED BY phrase must also be specified. When the binary search is executed, the OTS uses the first or only index associated with the table as the controlling index for the search.

The selective (binary) search is implemented in the OTS as follows:

1. The OTS examines the range of permissible values for the index of the table being searched; selects the median value; and assigns this median value to the index.

2. The OTS then proceeds to process the sequence of simple tests for equality, beginning with the first, with the index set to the median value.

3. If all of the tests for equality are true (success), the search is terminated; the associated imperative statement is executed; the search exits; and the index retains its current value.

4. If any of the tests for equality is false, the following results occur.

   a. The OTS determines if all of the possible occurrences for the table have been tested. If the table has been exhausted, the imperative statement which accompanies the AT END statement (if specified) is executed. In either case, control is passed to the next procedural statement.

   b. The OTS will now determine which half of the table is to be eliminated from further consideration. This determination is predicated on whether the key being tested is in ascending or descending order, and whether the test failed because of a greater than or less than comparison. For example, if the key values being tested are stored in ascending order, and the median table element being tested is greater than the value being tested for equality, the OTS will assume that all key elements following the one tested are also greater than the value being tested for equality. Therefore, the lower half of the table, those items which follow the current index setting, are no longer in contention.

   c. Once the direction of search is determined, half of the table is eliminated from further consideration. A new range of permissible index values is computed from the remaining half of the table.

   d. Processing begins all over again from step 1.

See Figure 5-20, Example 6, for an example of searching a table using Format 2 of the SEARCH verb.

```
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  SINGLES-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
        "048000960000000017
        "0960017300008I620
        "1730026400023561 7
        "2640034600039032 5
        "3460043300059532 8
        "4330050000083893 2
        "500009999910533361".
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-TAX               PIC 999V99,
        04  M-PERCENT           PIC V99,
TEMP-INDEX                          USAGE INDEX,
```

Figure 5-20
Example of Using SEARCH
To Search a Table

```
Example 1

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 2

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

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.
```

Figure 5-20 (Cont.)
Example of Using SEARCH
To Search a Table

```
Example 4

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 5

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 6

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 MARRITAL-STATUS OF OUTPUT-MASTER = "M"
                GO TO MARRIED.
```

Figure 5-20 (Cont.)
Example of Using SEARCH
To Search a Table

CHAPTER 6

FILE HANDLING

PDP-11 COBOL provides three ways to arrange the records in its files
(file organization): sequential, relative, and indexed. The
ORGANIZATION in the Environment Division clause specifies the file
organization for COBOL files.

NOTE

Indexed file organization is available
only to users with RMS-11K software.

PDP-11 COBOL provides three ways to process the records in its files
(file access): sequential, random, and dynamic. The ACCESS MODE
clause specifies the file access mode for each file used by COBOL
programs. The following chart shows the three file organizations and
the file access methods that apply to each of them:

FILE ORGANIZATION	FILE ACCESS
SEQUENTIAL	SEQUENTIAL
RELATIVE	SEQUENTIAL RANDOM DYNAMIC
INDEXED	SEQUENTIAL RANDOM DYNAMIC

Once a program creates a file, all other programs that access it must
describe it with the same file organization. For example, it is not
possible to create a sequential file in one program and read it as a
relative file with another program. However, programs can use
different access methods to process records in the same file as long
as the organization of the file supports the access method.

The following table compares the different file organizations with their file manipulation capabilities.

Table 6-1
COBOL File Types

Access	File Type (Organization)		
Capabilities	Sequential	Relative	Indexed
Sequential	Yes	Yes	Yes
Random	No	Yes	Yes
Record Replacement	Limited	Yes	Yes
Record Addition (at end of file)	Yes	Yes	Yes
Record Insertion	No	Limited	Yes
Record Deletion	No	Yes	Yes

COBOL I/O statements allow COBOL programs to communicate with the system devices. These statements differ for sequential, relative, and indexed file organizations. Therefore, the COBOL I/O statements are discussed separately by file organization. Section 6.1 discusses sequential organization, Section 6.2 discusses relative organization, and Section 6.3 discusses indexed organization. All file processing is performed by the COBOL object time system (OTS), regardless of organization.

Table 6-2 shows which statements apply to each file organization methods:

Table 6-2
I/O Statements

Sequential I/O Statements	Relative and Indexed I/O Statements
CLOSE	CLOSE
OPEN	DELETE
READ	OPEN
REWRITE	READ
WRITE	REWRITE
	START
	WRITE

6.1 SEQUENTIAL FILE ORGANIZATION

Sequential file organization arranges the records in a file serially; each record (except the first) has another record preceding it and each record (except the last) has another record following it. The records remain in the order in which they were written. Thus, COBOL statements cannot delete records from the file, insert new records between existing records, or alter the order of the existing records in any way. However, they can replace existing records (providing the length of the replacement record is identical to the original) and add new records onto the end of the file.

The opening operation for reading, writing, or updating sequential files must begin with the first record in the file and proceed by the prescribed order through the file. For example, to read a particular record in the file, say the 15th record, the program must open the file and successfully execute 14 READ statements before the 15th execution can read the desired record. The program can read all of the remaining records (from record 16 on), but it cannot read any record prior to record 16 without opening the file again and beginning with record 1.

Sequential files always contain an end-of-file mark that designates the end of the file. COBOL statements can write over the end-of-file mark and, thus, extend the length of a file. (The software inserts another end-of-file mark after the last record written.) Since the end-of-file mark indicates the end of useful data, PDP-11 COBOL 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. See Figure 6-1.

End-of-File Mark————————/   Unused Portion of Medium
                                          Reserved for File
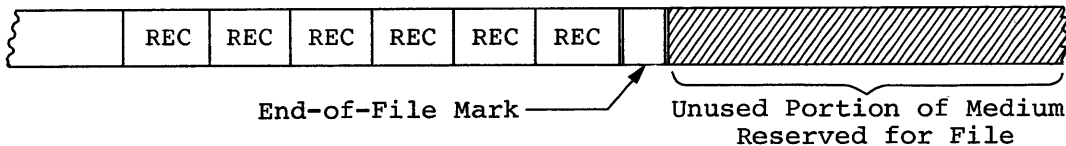
Figure 6-1 Placement of End-of-File Mark

Occasionally a file with sequential organization is so large that it requires more than one volume (such as a multi-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 multi-volume files, the end-of-file mark appears once, at the end of the last record on the last volume. See Figure 6-2.

NOTE

RSTS/E does not support multi-volume files.
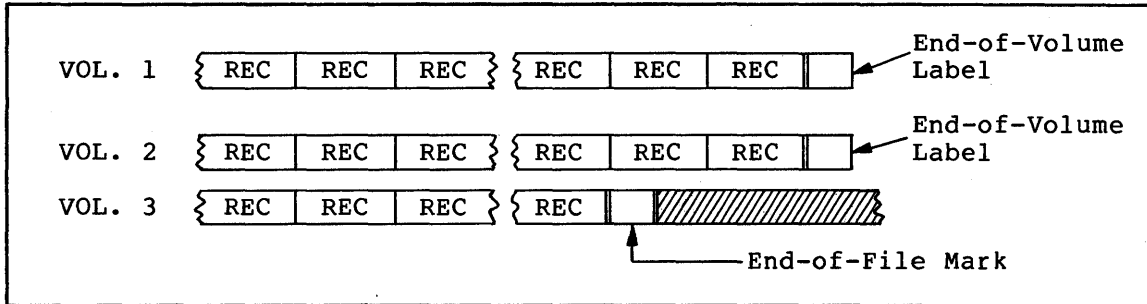
Figure 6-2   Placement of the End-of-Volume Label and
End-of-File Mark in a Multi-Volume File

6.1.1 Record Size

If there is only one record description for a file or if there are
more than one that describe the same length record, that file contains
fixed-length records. If the data descriptions for a sequential file
consist of more than one record description, which describe several
different-sized records, that file contains variable-length records.

When a program creates a sequential file with variable-length records,
the software places a count field in front of each record it writes
into the file. This count field contains the number of character
positions in the record. When a COBOL statement requests the record,
the software releases a record whose length is that specified by the
count field. The OTS creates and uses the count field automatically.
COBOL statements cannot access it during input operations, and the 01
level record description entries must not describe it.

6.1.2 RECORD CONTAINS Clause

The RECORD CONTAINS clause, when specified without the "integer-1 TO"
option, is for documentation purposes only. The compiler determines
record size from the data descriptions. When the "integer-1 TO"
option is specified, it forces the compiler to generate a variable
length record file, even if the data descriptions describe fixed
length records.

Conversely, if the data descriptions for a sequential file describe
variable-length records, the software sets up variable sized records
automatically and ignores this clause.

Even though the software ignores the values in the "integer-1 TO..."
phrase, the clause may be used in any program to document record
sizes.

6.1.3 SAME RECORD AREA Clause

The file system reserves a record processing area in memory for each
file. This area is the current record area. The system fixes the
location of the current record area when it opens the file. It also
reserves a byte preceding and following each current record area for
possible print-control characters. The current record area always
begins on an even byte boundary. Two or more files may share a
current record area if a SAME RECORD AREA clause contains their
file-names. This clause causes the system to begin the current record
area of each file listed at a common location. (Thus, current record
areas that share space are aligned on their leftmost bytes.) The
records do not have to be the same size and the current record areas
need not have the same maximum size. The following sample statement
would cause FILEA and FILEB to share the same current record area:

        I-O-CONTROL.

        SAME RECORD AREA FOR FILEA FILEB

        . . .

Since the system places a file's current record area in a separate
location from its buffers, each READ, WRITE, and REWRITE operation
causes a record to move between the buffers and the current record
area. When a program reads a record from a file, modifies it, and
writes it into another file, a SAME RECORD AREA clause, containing
both file-names, can save an entire move of the record. The following
illustration shows these record movements:
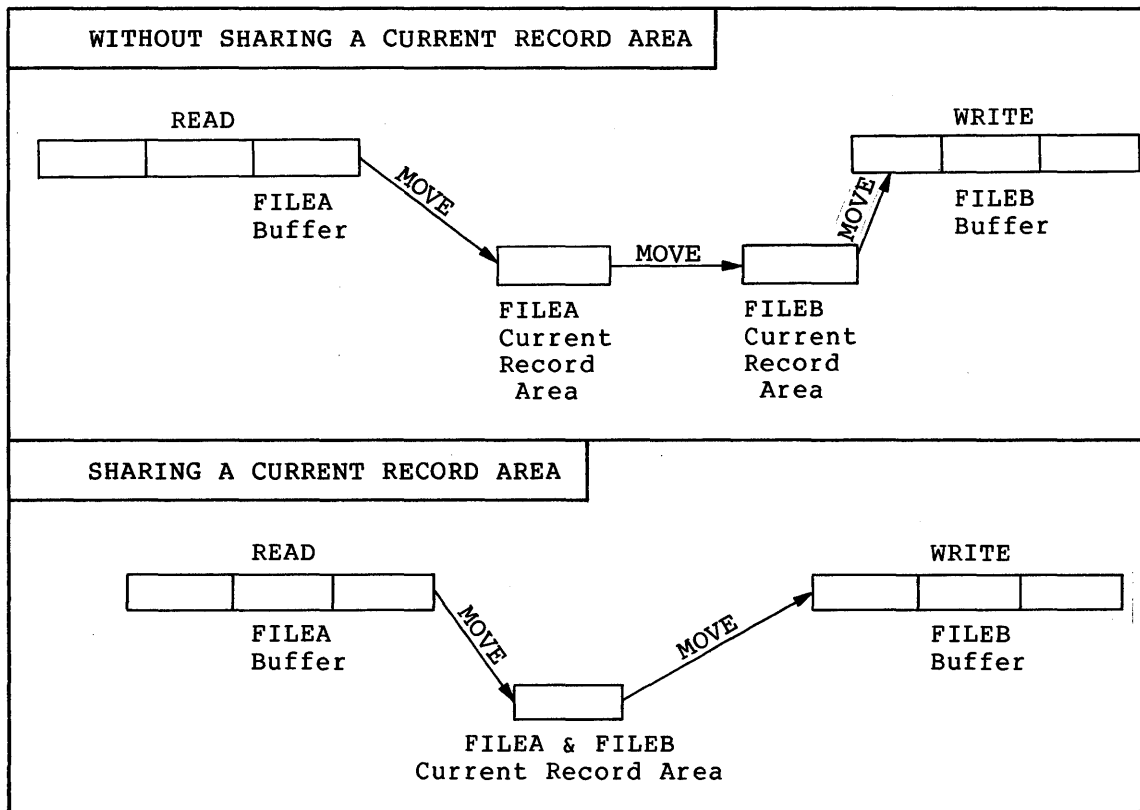
Record Movement Caused by
Reading, Processing, and Writing
Records in Two Files

6.1.4 Print-Controlled Records

If a sequential file is described in a LINAGE IS clause, an APPLY PRINT-CONTROL clause, or is referenced in a WRITE statement with the ADVANCING clause specified, and the file is not going directly to a printing device (is going to be spooled), the software designates the file as a print-controlled file. Print-controlled files contain form advancing information with each record. Explicit forms control bytes are placed directly into the file. Therefore, any COBOL program trying to process a print-controlled file may have unpredictable results.

6.1.5 Record Blocking

The manner in which the file system blocks the records of sequential files depends on the device to which the file is assigned and the presence and format of the BLOCK CONTAINS clause.

COBOL programs can assign sequential files to disk which requires fixed-length virtual blocks, and to magnetic tape, which allows variable-length blocks.

The BLOCK CONTAINS clause of a COBOL program refers to a logical block size. For magnetic tape, the logical block size and virtual block size are the same. For disk, however, the logical block size is equal to one or more virtual blocks. (A virtual block on disk is 512 bytes).

For files assigned to disk, the OTS packs records together (end-to-end) until a logical block is filled. The logical block is written to disk, and any portion of the previously processed record that did not fit into the logical block is put into the next logical block. This process is called record spanning in that it allows records to span virtual block boundaries.

Record spanning is prohibited for files assigned to magnetic tape. For these files, only complete records (fixed or variable length) are placed end-to-end in a logical block. The OTS writes the logical block out to the file when it determines that the block is full to the extent that the next record will not fit into it.

There are three ways to specify block size in a COBOL program; by default; by using the BLOCK CONTAINS integer RECORDS clause; or by using the BLOCK CONTAINS integer CHARACTERS clause. The default philosophy is to make the logical block size as small as possible; thus minimizing the memory buffer space required. By using the BLOCK CONTAINS (integer RECORDS or integer CHARACTERS) clause, you can increase the memory buffer space required. Increasing the buffer space, allows for faster I/O by decreasing the number of I/O operations required to process a file. Use the BLOCK CONTAINS clause only if you can afford the price of additional memory buffer space for the ability to process your files faster. The following paragraphs further define the three blocking methods:

Default

By default, the logical block size is made equal to the record size (add four bytes for variable length records on magnetic tape or two bytes for variable length records on disk). For disk files, the logical block size is rounded up to the next even multiple of 512 bytes to make the logical block size an integral number of virtual blocks. For example:

If the maximum record size for a disk file is 510 bytes, and the file contains variable length records, then the logical block size is 1024 bytes. (510 plus 4 for variable length records is 514, and 514 rounded up to the next even multiple of 512 is 1024.)

BLOCK CONTAINS integer RECORDS

If this clause is used, the logical block size is equal to the record size (plus four bytes for variable length records on magnetic tape or two bytes for variable length records on disk) times the number of records per block. For disk files, the logical block size is rounded up to the next even multiple of 512 bytes to make the block size an integral number of virtual blocks. For example:

If the record size for a fixed-length disk file is 100 bytes and the clause BLOCK CONTAINS 10 RECORDS is specified, the logical block size is 1024 bytes. (100 times 10 is 1000, and 1000 rounded up to the next even multiple of 512 is 1024).

BLOCK CONTAINS integer CHARACTERS

If this clause is used, the logical block size is equal to the number of characters given in the clause. If the specified number of characters is less than the actual record size (plus four bytes for variable-length records on magnetic tape or two bytes for variable length records on disk) the compiler generates a block size that is equal to the actual record size. For disk files, the specified number of characters must equal an even multiple of 512. If the number you specify is not correct, the OTS will round the logical block size it finds to the next even multiple of 512 bytes.

When a program assigns a file to magnetic tape, all programs that access the file must describe it the same way that the creating program described it in order to guarantee an accurate allocation of buffers.

Note: The previous discussion has used the following format:

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

If the following format is used:

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

the compiler ignores integer-1, and integer-2 is used as the integer.

6.1.6 Buffering

When the system performs an input operation, it reads a block from the medium into the buffer, and moves a record from the buffer to the current record area. Each subsequent read operation moves a record from the buffer to the current record area. When it has exhausted the buffer (has read an entire block), the system reads another block into the buffer.

When performing an output operation, each write operation moves a record from the file's current record area into the file's buffer. Each subsequent write operation moves a record from the current record area into the buffer. The system writes the block to the medium when it has filled the buffer.

The following subsections discuss the size of the buffers, the number of buffers, and the sharing of buffers.

6.1.6.1 **Buffer Size** - Buffer size depends on the size of the largest record in the file and on the blocking factor. For files with sequential organization, the buffer size will be at least 512 bytes.

6.1.6.2 **I-O Buffer Areas** - The RESERVE clause in the Environment Division specifies the number of I-O buffer areas to be allocated for each file. Each I-O area represents the space for one logical block. A minimum of one and a maximum of two are permitted for sequential files. One is the default. Since two I-O areas do not increase the speed of access and take additional memory space, it is recommended that this clause not be used.

6.1.6.3 **Buffer Space** - To calculate the total amount of buffer space required for each sequential file, the following algorithm may be used:

    Buffer space = record size + (logical blocksize * no. of areas)
        + 234

In addition there are 76 bytes of buffer space that are shared among all files.

6.1.6.4 **Sharing Buffer Space Among Files** - The SAME AREA clause provides a simple method of sharing buffer space among several files. Two or more files may share the same buffers if the SAME AREA clause contains their file-names and only one of them is open at any time during program execution. Further, since only one file is open at a time, the files will also share the same current record area. The size of the current record area is set to the size of the largest record description specified in the group.

If only one of these files is open at a time, the following sample statement causes them all to share the same buffer and current record area.

        I/O-CONTROL.

            SAME AREA FOR FILEA FILEB FILEC.

            . . .

6.1.7 Sequential I/O Statements

PDP-11 COBOL provides the following I/O statements for sequential files:

- CLOSE
- OPEN
- READ
- REWRITE
- WRITE

Before a COBOL program can access a file, it must open the file; then, when the program is finished with the file, it must close the file.

A COBOL program may open a sequential file in one of four modes, INPUT, OUTPUT, I-O (input/output), or EXTEND. In INPUT mode, records may be read from the file; in OUTPUT mode the file is created and records can only be written to it; in I-O mode, records can be read from the file and updated; in EXTEND mode, records may be added onto the end of the file. Table 6-3 shows which statements apply to the four different OPEN modes of sequential files. (The table does not include the OPEN and CLOSE statements since they apply to all modes.)

Table 6-3
Sequential OPEN Modes

Statement	Open Mode			
	Input	Output	Input-Output	Extend
READ	X		X	
REWRITE			X	
WRITE		X		X

6.1.7.1 Opening Sequential Files – The OPEN statement makes a file available for processing by a COBOL program. A program must execute an OPEN statement for a file before it executes any other I/O statement for that file. Consider the following sample OPEN statement. It opens the file named THOREAU for input/output. The program containing this statement could, after executing it, READ, REWRITE, and CLOSE THOREAU.

```
        ENVIRONMENT DIVISION.
        INPUT-OUTPUT SECTION.
        FILE-CONTROL.
            SELECT THOREAU
            ASSIGN TO "DK1:".
        .
        .
        .

        DATA DIVISION.
        FILE SECTION.
        FD  THOREAU
        .
        .
        .

        PROCEDURE DIVISION.
        .
        .
        .

            OPEN I-O THOREAU.
        .
        .
        .
```

The OPEN statement must refer to the file by the file-name appearing
in both the SELECT clause in the Environment Division and the FD
paragraph in the Data Division.

When the OTS executes an OPEN statement, it performs the following
actions for the file named in the statement:

- If the file is already open, the OTS generates an error
  message and performs the USE procedure section (if
  specified). (Section 6.9 discusses USE procedures and
  Section 12.3 discusses error messages.)

- When opening an existing file, the attributes (i.e., record
  length, block size, etc.) of the file are used for accessing
  the file. Those specified in the program are ignored. Be
  sure that the attributes specified in the COBOL program agree
  with the actual attributes of the file.

- If the SELECT clause of the File-Control paragraph declares
  the file OPTIONAL, the OTS displays the following message:

      "FILE nnn ... OPTIONAL FILE MOUNTED? Y OR N?"

  (nnn represents the file-name.)

  If the file is available for processing, type a Y. If not,
  type an N. If the file is not available (N), the OTS
  disables all I/O processing on the file except READ and
  CLOSE; a later READ statement causes program control to take
  the AT END imperative path.

- If a SAME AREA clause contains the name of the file and none
  of the other files named in the clause is open, the OTS
  allocates buffer space for the file.

- When the file has passed all of the preceding checks and is
  ready for opening, the OTS instructs the Record Management
  Services to open the file. If the Record Management Services
  fails to open the file, the OTS reports an error condition
  and performs any applicable USE procedure (if present).

- If the program is creating the file (OPEN OUTPUT) and the file description specifies LINAGE or APPLY PRINT-CONTROL, the OTS initializes the LINAGE counters.

- Finally, depending on which statements apply to the open mode, the OTS enables or disables all of the program's I/O statements that refer to the file (see Table 6-3). For example, if the OPEN mode is INPUT, it enables all READ statements for that file and disables all REWRITE and WRITE statements for that file.

Since the EXTEND mode simply allows the WRITE statement to add records onto the end of the file, files opened in this mode must already exist on disk or tape (only the last file on magnetic tape can be extended). If the file does not exist, the OPEN statement fails and the OTS issues an error message.

6.1.7.2 **Reading Sequential Files** - The READ statement makes the next logical record of an open sequential file available to the program. If the preceding I/O operation was an OPEN, it makes the first record of the file available to the program.

Consider the following example. If the last I/O operation on the file named THOREAU was an OPEN, this statement would provide the program with the first record in the file THOREAU. Every time the statement is executed, it provides the program with the next sequential record in THOREAU. Program control transfers to the paragraph named LIBRARY when an end-of-file mark is encountered during the READ.

```
BEGIN.  OPEN THOREAU.
LOOP.   READ THOREAU AT END GO TO LIBRARY.
        .
        .
        .
        GO TO LOOP.
```

If the file contains variable-length records, the program must determine the length of the record just read. No such information is supplied to the user program.

NOTE

RSTS/E processes records from unit record devices as variable length records. This means that records read in from a card reader or paper tape reader will have trailing blanks deleted. For example, reading blank records will not change the user record area. To avoid problems, move blanks into the record area before each read.

If the file is open in the I-O mode, the successful execution of a READ statement enables any following REWRITE of the record just read. (For further information on the REWRITE statement, see the next subsection -- 6.1.7.3.)

If the file has more than one record description, the records automatically share the same current record area. The OTS does not clear this area before it executes the READ statement (no blank filling, etc.). Therefore, if the record read by the latest READ statement does not fill the entire current record area, the area not overlaid by the incoming record remains unchanged. For example, if the file's record area contains ten 3's, and a READ operation moves in a 6-character record containing all 1's, the current record area then contains six 1's followed by four 3's. Consider the following example:

Current Record Area with all 3's $\boxed{3333333333}$

Next Record in the File $\boxed{111111}$

Current Record Area after READ $\boxed{1111113333}$

6.1.7.3 **Rewriting Records into Sequential Files** – The REWRITE statement places the record just read from an input-output file back into its file on disk or magnetic tape. (The WRITE statement cannot access I-O files.) The following sample statement writes the record, REC1, back into its file. (REC1, of course, must be a record in the file read by the preceding READ for that file.)

        REWRITE REC1

Before the REWRITE statement can refer to a record, the program containing the statement must meet the following conditions:

   • The file containing the record must be open in the I-O mode;

   • The last I/O operation on the file containing the record must have been a successful READ;

   • The record length of the record to be rewritten must be the same as the record last read from the file.

6.1.7.4 **Writing Sequential Files** – The WRITE statement releases a logical record to an output file, thereby creating an entirely new record in the file.

The following sample WRITE statement releases the record PRINT-LINE to the device assigned to that record's file, then skips three lines. When it reaches the end of a page (as specified by the LINAGE clause), it causes program control to transfer to the subroutine, HEADER-RTN.

        WRITE PRINT-LINE BEFORE ADVANCING 3 LINES

        AT EOP GO TO HEADER-RTN.

Note that this produces two blank lines following every line printed.

The WRITE statement releases records to files that are open in either the output or extend mode. The following text discusses the two modes separately.

   • OUTPUT Mode – The WRITE statement can create the following two kinds of files in the OUTPUT mode:

1. <u>Print-files</u> - A print-file produces a listing on a printing device. The LINAGE clause, the APPLY PRINT-CONTROL clause, or a WRITE statement with the ADVANCING option included, 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 printing device, but may also release them to a storage medium such as disk for printing at a later time.

2. <u>Storage files</u> - A storage file remains on disk or tape for future reference. All files that are not print-files are storage files. A sample storage file WRITE statement follows; this statement writes a record named WALDEN into a file:

    WRITE WALDEN

● <u>EXTEND Mode</u> - A WRITE to a storage file opened in the EXTEND mode simply adds new records logically in sequence after the last record in the file. As the statement extends the file, the Record Management Services automatically handles requests for additional storage space. (Print-files on disk should only be opened for EXTEND if they are being opened as a print-file.)

6.1.7.5 **Closing Sequential Files** - The CLOSE statement terminates processing on the file referred to in the statement. The following sample CLOSE statement terminates processing on the file named THOREAU:

    CLOSE THOREAU

When the CLOSE statement closes a file, no other I/O operation can access that file until another OPEN statement opens the file.

If the statement specifies the LOCK option, the program cannot open the file again in this run. The CLOSE statement with the LOCK clause is shown below:

    CLOSE THOREAU WITH LOCK

The lock option has no effect on the physical device containing the file.

If a SAME AREA clause contains the name of the file just closed, the program may open one of the other files named in the clause.

6.2 **RELATIVE FILE ORGANIZATION**

Relative file organization arranges the records of the file into numbered record positions. It assigns each record position a number that identifies that position relative to the beginning of the file (the first record position in the file has record number 1, the second has record number 2, etc.).

1	2	3	4	5	6	7	8	9	10	...

Record Positions in a Relative File

When a program executes a random DELETE, REWRITE, READ or WRITE operation on a relative file, the value in the relative key is used to select records from these numbered record positions in the same way that a subscript selects an item in a table.

Thus, while sequential and relative files both arrange their record positions in a serial order, COBOL statements can address the record positions of a relative file by their position numbers, and successive accesses do not have to proceed through the file in a prescribed, serial, order.

Another significant feature of relative file organization is that each record position does not have to contain a valid record. Although each position actually occupies one record space, a byte preceding the record on the storage medium indicates whether or not that space contains a valid record. Thus, a file may have fewer records than it has record positions, and the indicated empty record positions may be anywhere in the file.

The numerical order of the record positions remains the same during all operations on a relative file; however, the accessing statements can move a record from one position to another, delete a record from a position, or insert new records into empty positions.

6.2.1 Record Size

A relative file may contain either fixed-length or variable-length records. (Fixed-length records have one or more record descriptions that describe the same size record. Variable-length records have more than one record description that describe several different sized records.) However, the COBOL compiler allocates a record area on the I/O device, equal to the largest record described plus one. This extra byte is an existence byte. It indicates whether the record area contains a valid record. For variable length records in a relative file, the software adds a two byte count field. On a write operation the actual record is written out to the I/O device not the maximum length record. The length of this record is placed in the two byte count field. On a read operation this two byte count field is used to determine the length of the record to be read in.

6.2.2 RECORD CONTAINS Clause

The RECORD CONTAINS clause, when specified without the "integer-1 TO" option, is for documentation purposes only. The compiler determines record size from the data descriptions. When the "integer-1 TO" option is specified, it forces the compiler to generate a variable length record file, even if the data descriptions describe fixed length records.

Conversely, if the data descriptions for a sequential file describe variable-length records, the software sets up variable sized records automatically and ignores this clause.

Even though the software ignores the values in the "integer-1 TO..." phrase, the clause may be used in any program to document record sizes.

6.2.3 SAME RECORD AREA Clause

The SAME RECORD AREA clause is identical for all file organizations. See Section 6.1.3.

6.2.4 Record Blocking

The size of a file is expressed as an integral number of virtual blocks. Virtual blocks are physical storage structures. That is, each virtual block within a file is a unit of data whose size depends on the physical medium on which the file resides.

Relative files may reside only on disk. The size of virtual blocks within files on disk devices is always 512 bytes.

Relative files use a logical storage structure known as a logical block or bucket. A bucket consists of from 1 to 32 virtual blocks.

This distinction should be made clear. A virtual block is a physical entity which is fixed in size and cannot be changed. A bucket, however, is a logical entity. Its size is directly under your control. Records may span virtual block boundaries. They may never span bucket boundaries.

Increasing the bucket size increases the speed of sequential processing of a file because fewer I/O operations are needed to access the smaller number of buckets in the file. On the other hand, a larger bucket size means that more memory space is taken up by the I/O buffers. Increasing the bucket size may not increase the speed of random processing of a relative file.

There are three ways that the bucket size may be specified in a COBOL program; by default, by using the construct BLOCK CONTAINS integer RECORDS, or by using the construct BLOCK CONTAINS integer CHARACTERS.

The default is to make the bucket size as small as possible, to minimize the memory buffer space required. By using the BLOCK CONTAINS integer (RECORDS of integer CHARACTERS) clause, you can increase the memory buffer space required. Increasing the buffer space allows for faster I/O by decreasing the number of operations required to access a file. The following paragraphs further define the three blocking methods:

Default

The default philosophy is to make the bucket size as small as possible to minimize the memory buffer space required. The algorithms for calculating the bucket size follow:

$$Bnum = ((1+Rlen)/512)+1 \quad \text{Fixed length record}$$

$$Bnum = ((3+Rmax)/512)+1 \quad \text{Variable length record}$$

Where:

Bnum	is the number of virtual blocks per bucket, ranging from 1 to 9.
Rlen	is the fixed record length (in bytes).
Rmax	is the maximum record length (in bytes) if the record length is variable.

The number 1 is for the existence byte. The number 3 is for the existence byte plus 2 bytes for the record length.

Table 6-4 gives the bucket size for possible record lengths.

Table 6-4
Bucket Sizes for Possible Record Lengths

Bnum	Rlen	Rmax
1	1-511	1-509
2	512-1023	510-1021
3	1024-1535	1022-1533
4	1536-2047	1534-2045
5	2048-2559	2046-2557
6	2560-3071	2558-3069
7	3072-3583	3070-3581
8	3584-4095	3582-4093
9		4094-4095

BLOCK CONTAINS Rnum RECORDS

If the BLOCK CONTAINS Rnum RECORDS clause is used, where Rnum is an integer, then the following algorithms are used to calculate the bucket size.

$$Bnum = (((Rlen+1)*Rnum)/512)+1 \quad \text{Fixed length record}$$

or

$$Bnum = (((Rmax+3)*Rnum)/512)+1 \quad \text{Variable length record}$$

Where:

Bnum   is the number of virtual blocks per bucket, ranging from 1 to 32.

Rlen   is the fixed record length (in bytes).

Rmax   is the maximum record length (in bytes) if the record length is variable.

Rnum   is the number of records per bucket as given in the BLOCK CONTAINS clause.

BLOCK CONTAINS Cnum CHARACTERS

If the BLOCK CONTAINS Cnum CHARACTERS clause is used, where Cnum is an integer, then Cnum is subject to the following constraints.

(1)   Cnum   Rlen+1   for fixed length records
                  or
      Cnum   Rmax+3   for variable length records

(2)   Cnum mod 512 = 0

Based on CNUM, the bucket size is calculated as follows:

Bnum=Cnum/512

where:

Cnum   is the number of characters per bucket as given in the BLOCK CONTAINS clause.

Rlen   is the fixed record length (in bytes).

Rmax   is the maximum record length (in bytes) if the record length is variable.

Bnum   is the number of virtual blocks per bucket, ranging from 1 to 32.

Violation of constraint (1) causes a warning error and the default method is used to calculate the bucket size. Constraint (2) means that Cnum should be a multiple of 512. If not, a warning error is given and Cnum is increased to the next even multiple of 512.

The bucket size must be the same when the file is created and each time the file is accessed. Therefore, the BLOCK CONTAINS clause must never change for a particular file.

Note: The previous discussion has used the following format:

$$\left[\underline{\text{BLOCK}} \text{ CONTAINS integer} \begin{Bmatrix} \underline{\text{RECORDS}} \\ \text{CHARACTERS} \end{Bmatrix} \right]$$

If the following format is used:

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

the compiler ignores integer-1, and integer-2 is used as the integer.

6.2.5 Buffering

When the system performs a sequential or random input operation, it reads a bucket from the medium into the buffer, and moves a record from the buffer to the current record area. Any subsequent sequential read operations move a record from the buffer to the current record area. When it has exhausted the buffer (has read an entire bucket), the system reads another bucket into the buffer.

When performing a random read operation, the appropriate bucket is read into a file's buffer. The record is then moved from the buffer to the current record area.

When performing a sequential output operation, each write operation moves a record from the file's current record area into the file's buffer. Each subsequent sequential write operation moves a record from the current record area into the buffer. The system writes the bucket to the medium when it has filled the buffer.

When performing a random output operation, the appropriate bucket is read and the record is moved from the file's current record area into the appropriate position in the file's buffer. The system writes the bucket back out to the medium before reading any additional blocks.

The following subsections discuss the size of the buffers, the number of buffers, and the sharing of buffers.

6.2.5.1 Buffer Size - Buffer size depends on the size of the largest record in the file and on the blocking factor. For relative files, buffer size must be some multiple of 256 words (512 bytes).

6.2.5.2 I/O Buffer Areas - The RESERVE clause in the Environment Division specifies the number of I/O buffer areas to be allocated for each file where an area represents the space for one bucket. A minimum of one and a maximum of two I/O areas are permitted for relative files. One is the default. It is recommended that this clause not be used, because two I/O areas do not increase the speed of access and take up additional space.

6.2.5.3 Buffer Space - To calculate the total amount of buffer space in bytes required for each Relative file, the following algorithm may be used:

Buffer space = record size + bucket size + 266

In addition, there are 76 bytes of buffer space that are shared among all files.

6.2.5.4 Sharing Buffer Space Among Files - The SAME AREA clause provides a simple method of sharing buffer space among several files. This clause is identical for all file organizations. See Section 6.1.6.4.

6.2.6 Relative I/O Statements

The COBOL I/O statements, CLOSE, DELETE, OPEN, READ, WRITE, REWRITE, and START can refer to relative files.

A COBOL program may open a relative file in one of three modes, INPUT, OUTPUT, or I-O, and access an open relative file in one of three ways, sequentially, randomly, or dynamically. In INPUT mode, records may be read from the file; in OUTPUT mode, the file is created and records may be written to the file; in I-O mode, records may be read from the file, updated on the file, deleted from the file, or written to the file. The following table shows which statements and access methods apply to the three different OPEN modes of relative files.

Table 6-5
Relative OPEN Modes

FILE ACCESS MODE	STATEMENT	OPEN MODE		
		INPUT	OUTPUT	I-O
Sequential	DELETE READ REWRITE START WRITE	X X 	 X	X X X X
Random	DELETE READ REWRITE START WRITE	 X 	 X	X X X X
Dynamic	DELETE READ READ NEXT REWRITE START WRITE	 X X X 	 X	X X X X X X

NOTE

The term, current record pointer, used
in the following sections, refers to a
location in the operating system used to
determine the record number of the next
available record in a file.

6.2.6.1 **Access Modes** - The ACCESS MODE clause in the File-Control
paragraph dictates which of the three access modes may be used on that
file.

When the ACCESS MODE clause specifies SEQUENTIAL, the I/O statements
must refer to the records in the file sequentially, starting (after
opening) with the first record and stepping through with each
reference to the end of the file. The I/O statements ignore record
positions that do not contain valid records.

When the ACCESS MODE clause specifies RANDOM, the I/O statements refer
to the records in the file by record position. Thus, the statements
may refer to record positions that do not contain valid records.   The
program must specify the desired record position number by placing a
value in the file's relative key. If an I/O statement refers to a
record position with the relative key, and that record position does
not exist (either because the position does not contain a record or
because it is beyond the end of the file), the INVALID KEY imperative
statement may be executed depending on the particular I/O statement
used. (The INVALID KEY imperative statement is explained with each of
the relative I/O statements in this section.)

When the ACCESS MODE clause specifies DYNAMIC, the I/O statements may refer to the records in the file either sequentially or randomly. The OTS determines which access method to use from the OPEN mode (INPUT, OUTPUT, or I-O) and the form of the I/O statement. For example, if the statement READ ...INVALID KEY is to access an open input file dynamically, the OTS uses the relative key for random access; if the statement READ NEXT ... is to access an open input file dynamically, the OTS sequentially accesses the next existing record.

The following sections (6.2.5.2 through 6.2.5.8) on using the I/O statements themselves contain additional information about access modes.

6.2.6.2 Opening Relative Files - The OPEN statement for a relative file makes an INPUT, OUTPUT, or I-O mode file available so the COBOL program can access the records in the file sequentially, randomly, or dynamically.

The OPEN statement sets the current record pointer for the file to zero.

For example, the following sample OPEN statement opens the file named ARTICHOKE for input, and sets ARTICHOKE's current record pointer to zero. The program containing this statement could, after executing the statement, access ARTICHOKE with READ and START statements in the sequential access mode, READ statements in the random access mode, or READ, READ NEXT, and START statements in the dynamic access mode.

OPEN INPUT ARTICHOKE.

When the OTS executes an OPEN statement, it performs the following actions for the file named in the statement:

- If the file is already open, the OTS generates an error message and performs the USE procedure section (if specified). (Section 6.9 discusses USE procedures and Section 12.3 discusses error messages.)

- When opening an existing file, the attributes (i.e., record length, block size, etc.) of the file are used for accessing the file. Those specified in the program are ignored. Be sure that the attributes specified in the COBOL program agree with the actual attributes of the file.

- If a SAME AREA clause contains the name of the file and none of the other files named in the clause is open, the OTS allocates buffers space for the file.

- When the file has passed all of the preceding checks and is ready for opening, the OTS instructs the Record Management Services to open the file. If the Record Management Services fails to open the file, the OTS reports an error condition and performs any applicable USE procedure (if present).

- Finally, depending on which statements apply to the open mode, the OTS enables or disables all of the program's I/O statements that refer to the file (see Table 6-5). For example, if the OPEN mode is INPUT, it enables all READ and START statements for that file and disables all REWRITE, DELETE and WRITE statements for that file.

If the file is being accessed randomly or dynamically, the program must maintain a correct value in the relative key. If the file is being accessed sequentially, the OTS ignores the value of the relative key, but updates it to contain the position number of the record being accessed.

6.2.6.3 Reading Relative Files – When applied to a file being accessed sequentially, the READ statement makes the next logical record of an open file available to the program.

When applied to a file being accessed randomly, the READ statement selects a specified record from an open file and makes it available to the program. The value of the relative key for the file identifies the specific record.

When applied to a file being accessed dynamically, the READ statement has two formats so that it can either select the next logical record (sequentially) or select a specified record (randomly) and make it available to the program. The READ NEXT statement takes the number in the current record pointer and finds the next present record. The following sample READ statement reads the file named ARTICHOKE sequentially and, when it exhausts the file, causes program control to transfer to the subroutine named FILEOUT:

        READ ARTICHOKE NEXT RECORD
            AT END GO TO FILEOUT.

For further information concerning the mechanics of the READ NEXT statement, see Section 6.2.5.7, Specifying the Next Record to be Read.

The READ with key takes the value in the relative key, moves it to the current record pointer, and reads the record being pointed to. The following READ (with key) statement reads the file named ARTICHOKE randomly, selecting records through the value in the file's relative key. If the relative key supplies a value that does not contain a valid record, the statement causes program control to transfer to the subroutine named NO-REC.

        READ ARTICHOKE RECORD
            INVALID KEY GO TO NO-REC.

If the file has more than one record description, the records automatically share the same current record area. The OTS does not clear this area before it executes the READ statement (no blank filling, etc.). Therefore, if the record read by the latest READ statement does not fill the entire current record area, the area not overlaid by the incoming record remains unchanged. For example, if the file's record area contains ten 3's, and a READ operation moves in a 6-character record containing all 1's, the current record area then contains six 1's followed by four 3's. Consider the following example:

Current Record Area with all 3's | 3333333333 |

Next Record in the File | 111111 |

Current record Area after READ | 1111113333 |

6.2.6.4 Rewriting Records into a Relative File - The REWRITE statement places a record back into its file on disk or magnetic tape. The following sample statement writes the record, BREAKERS, back into its file.

        REWRITE BREAKERS.

If the file is open in the sequential access mode, the statement rewrites the record just successfully read. If the file is open in either the random or dynamic access mode, the statement rewrites the record to the record position specified by the relative key.

6.2.6.5 Writing Records in a Relative File - The WRITE statement releases a logical record to a file. The following sample WRITE statement releases the record BREAKERS to the device assigned to that record's file. If the record already exists, program control transfers to the subroutine, WRITE-ERR.

        WRITE BREAKERS
            INVALID KEY GO TO WRITE-ERR.

The WRITE statement releases records to files that are open in either the OUTPUT or I/O mode. The following text discusses the two modes separately:

- OUTPUT Mode - The WRITE statement's only function with output files is to place entirely new records into the file. If more space is required for new record positions, the Record Management Services automatically extends the file size, regardless of the access mode being employed.

- I/O Mode - The statement's function with input-output files is to place records in record positions that already exist and are empty. The length of the records must not exceed the maximum length record specified for the file when it was created.

The relative WRITE statement creates only storage files since print-files are sequential files. The following SAMPLE statement writes a record named BREAKERS into its file:

        WRITE BREAKERS.

6.2.6.6 Deleting Records from a Relative File - The DELETE statement logically removes an existing record from a relative file. After a DELETE statement has successfully removed a record from a file, that record can no longer be accessed.

If the file is open in the sequential access mode, the statement removes the record just successfully read. For example, the following sample statement removes the record just read from the file named ARTICHOKE:

        DELETE ARTICHOKE RECORD.

If the file is open in either the random or dynamic access mode, the statement removes the record from the record position specified by the relative key. For example, the following sample statement deletes the record specified by the relative key from the file named ARTICHOKE; if the relative key supplies a value that does not contain a valid record, the statement transfers control to the subroutine named NO-REC.

```
        DELETE ARTICHOKE RECORD
            INVALID KEY GO TO NO-REC.
```

6.2.6.7 Specifying the Next Record to be Read - The START statement specifies which record in a file will be the next one to be referenced sequentially.

A READ NEXT statement should follow the START statement since the READ NEXT statement reads the next record from the one being pointed to by the current record pointer.

If the data area, SOMETHING, in the following example contains a 30 and position 33 in the file contains the next present record, the START statement sets the current record pointer to one less than 33 (32). The READ NEXT statement would then find the next present record, which we know is 33.

```
        WORKING-STORAGE SECTION.
        77 SOMETHING PIC S99 VALUE 30.
        77 ARTKEY PIC 99.
        .
        .
        .
        RD-SET.    MOVE SOMETHING TO ARTKEY.
                   START ARTICHOKE
                       KEY IS GREATER THAN ARTKEY
                       INVALID KEY GO TO NEWKEY.

        IN1.       READ ARTICHOKE NEXT RECORD
                       AT END GO TO FILEOUT.
```

The value of the RELATIVE KEY data item specified in the statement (ARTKEY in the preceding example) together with the conditional phrase specified in the statement (IS GREATER THAN in the preceding example) determines which record in the file will be accessed by the READ NEXT statement.

The START statement uses the value in the RELATIVE KEY data item to set the current record pointer. If record positions 30 and 33 contain valid records and ARTKEY contains 30, the START statement would set the current record pointer and RELATIVE KEY data item as follows:

1. If the conditional phrase specifies KEY IS GREATER THAN ARTKEY, the statement sets the current record pointer to 32.

2. If the conditional phrase specifies KEY IS EQUAL TO ARTKEY or NOT LESS THAN ARTKEY, the statement sets the current record pointer to 29.

The READ NEXT statement takes the number in the current record pointer and finds the next present record from that number. (If the pointer contains a 30 and the next present record is in position 33, it finds record number 33). The READ NEXT statement gets that record and places its record position number (33) into the current record pointer and the relative key.

A subsequent READ NEXT takes the number in the current record pointer, which is now 33 in our example, and finds the next present record. It fetches that record and places its record position number in the current record pointer and relative key.

6.2.6.8 Closing Relative Files - The CLOSE statement terminates processing on the file referred to in the statement. The following sample CLOSE statement terminates processing on the file named ARTICHOKE:

        CLOSE ARTICHOKE.

When the statement closes a file, no other I/O operation can access that file until another OPEN statement opens the file.

If the statement specifies the LOCK option, the program cannot open the file again in that run.

If a SAME AREA clause contains the name of the file just closed, the program may open one of the other files named in the clause.

6.3 INDEXED FILE ORGANIZATION

                        WARNING

        Indexed file organization is available
        only to users having RMS-11K software.

Unlike the physical ordering of records in a sequential file or the relative positioning of records in a relative file, the location of records in the indexed file organization is transparent to your program. The presence of keys in the records of the file governs the placement of records in an indexed file.

A key is a character string present in every record of an indexed file. The location and length of this character string is identical in all records. When creating an indexed file, you decide which character string in the file's records is to be a key. By selecting such a character string, the contents (i.e., key value) of that string in any particular record written to the file can be used by a program to identify that record for subsequent retrieval.

You must define at least one key for an indexed file. This mandatory key is the primary key of the file. Optionally, you can define up to 255 additional keys (i.e., alternate keys). Each alternate key represents an additional character string in records of the file. The key value in any one of these additional strings can also be used as a means of identifying the record for retrieval.

As programs write records into an indexed file, the values contained in the primary and alternate keys are used to locate the record in the file. From the values in keys within records a tree-structured table known as an index is built. An index consists of a series of entries. Each entry contains a key value copied from a record that a program wrote into the file. With each key value is a pointer to the location in the file of the record from which the value was copied. A separate index is built and maintained for each key you define for the file. Each index is stored in the file. Thus, every indexed file contains

at least one index, the primary key index. When you define alternate keys, an additional index is built and maintained for each alternate key. Figure 6-3 shows the general structure of an indexed file that has been defined with only a single key. Figure 6-4 depicts an indexed file defined with two keys, a primary key and one alternate key.
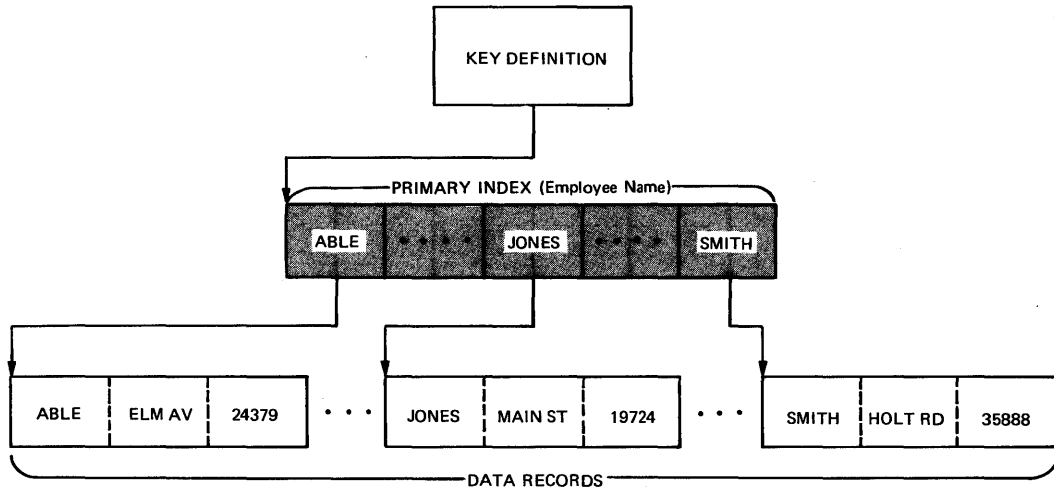
Figure 6-3 Single Key Indexed File Organization
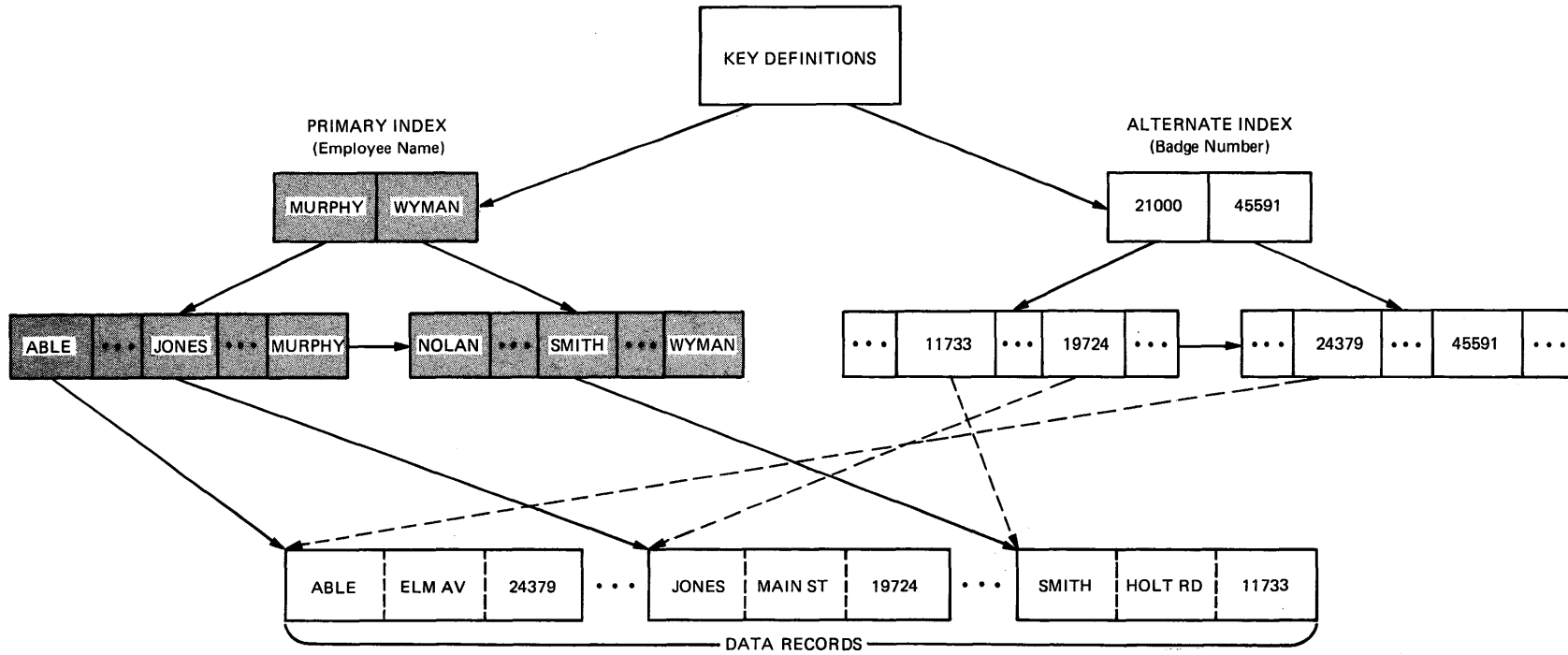
Figure 6-4   Multi-key Indexed File Organization

6.3.1 Record Size

A relative file may contain either fixed-length or variable-length records. (Fixed-length records have one or more record descriptions that describe the same size record. Variable-length records have more than one record description that describe several different sized records.) For variable length records in an indexed file, the software adds a two byte count field. On a write operation the actual record is written out to the I/O device not the maximum length record. The length of this record is placed in the two byte count field. On a read operation this two byte count field is used to determine the length of the record to be read in.

6.3.2 RECORD CONTAINS Clause

The RECORD CONTAINS clause, when specified without the "integer-1 TO" option, is for documentation purposes only. The compiler determines record size from the data descriptions. When the "integer-1 TO" option is specified, it forces the compiler to generate a variable length record file, even if the data descriptions describe fixed length records.

Conversely, if the data descriptions for a sequential file describe variable-length records, the software sets up variable sized records automatically and ignores this clause.

Even though the software ignores the values in the "integer-1 TO..." phrase, the clause may be used in any program to document record sizes.

6.3.3 SAME RECORD AREA Clause

The SAME RECORD AREA clause is identical for all file organizations. See Section 6.1.3.

6.3.4 Record Blocking

The size of a file is expressed as an integral number of virtual blocks. Virtual blocks are physical storage structures. That is, each virtual block within a file is a unit of data whose size depends on the physical medium on which the file resides.

Indexed files may reside only on disk. The size of virtual blocks within files on disk devices is always 512 bytes.

Indexed files, like relative files, use a logical storage structure known as a logical block or bucket. A bucket consists of 1 to 32 virtual blocks. The user may specify the number of virtual blocks contained within each bucket by using the BLOCK CONTAINS clause. This distinction should be made clear. A virtual block is a physical entity which is fixed in size and cannot be changed by the user. A bucket is a logical entity and its size is directly under user control. Records may span virtual block boundaries. They may never span bucket boundaries.

Increasing the bucket size increases the speed of processing of a file because fewer I/O operations are needed to access the smaller number of buckets in the file. On the other hand, a larger bucket size means that more memory space is taken up by the I/O buffers.

There are three ways that the bucket size may be specified by the user in a COBOL program: by default, by using the construct BLOCK CONTAINS integer RECORDS, or by using the construct BLOCK CONTAINS integer CHARACTERS.

The default is to make the bucket size as small as possible to minimize the memory buffer space required. By using the BLOCK CONTAINS (integer RECORD or integer CHARACTERS) clause, you can increase the memory buffer space required. Increasing the buffer space allows faster I/O by decreasing the number of operations to process a file. The following paragraphs further define the three blocking methods:

Default

The default philosophy is to make the bucket size as small as possible to minimize the memory buffer space required. The algorithms for calculating the bucket size follow:

Bnum= ((22+Rlen)/512)+1 Fixed length record

or

Bnum= ((24+Rmax)/512)+1 Variable length record

where:

Bnum   is the number of virtual blocks per bucket, ranging from 1 to 9.

Rlen   is the fixed record length (in bytes).

Rmax   is the maximum record length (in bytes) if the record length is variable.

The number 22 comes from a bucket overhead of 15 bytes and a fixed length record header of 7 bytes; 24 comes from a bucket overhead of 15 bytes and a variable length record header of 9 bytes.

Table 6-6 gives the bucket size for possible record lengths.

Table 6-6
Bucket Size for Possible Record Lengths

Bnum	Rlen	Rmax
1	1-490	1-488
2	490-1002	489-1000
3	1003-1514	1001-1512
4	1515-2026	1513-2024
5	2027-2538	2025-2536
6	2539-3050	2537-3048
7	3051-3562	3049-3560
8	3563-4074	3561-4072
9	4075-4095	4073-4095

BLOCK CONTAINS Rnum RECORDS

If the BLOCK CONTAINS num RECORDS clause is used, where num is an integer, then the following algorithms are used to calculate the bucket size.

$$Bnum = ((15+(Rlen+7)*Rnum)/512)+1 \quad \text{Fixed length record}$$

or

$$Bnum = ((15+(Rlen+9)*Rnum)/512)+1 \quad \text{Variable length record}$$

where:

Bnum	is the number of virtual blocks per bucket, ranging from 1 to 32.
Rlen	is the fixed record length (in bytes).
Rmax	is the maximum record length (in bytes) if the record length is variable.
Rnum	is the number of records per bucket as given in the BLOCK CONTAINS clause.

The number 15 is bucket overhead, 7 is the fixed length record header and 9 is the variable length record header.

BLOCK CONTAINS Cnum CHARACTERS

If the BLOCK CONTAINS Cnum CHARACTERS clause is used, where Cnum is an integer, then Cnum is subject to the following constraints.

(1) $Cnum \quad Rlen+1 \qquad$ for fixed length records
$\qquad\qquad\qquad$ or
$\qquad Cnum \quad Rmax+3 \qquad$ for variable length records

(2) $Cnum \bmod 512 = 0$

Based on Cnum, the bucket size is calculated as follows:

$$Bnum = Cnum/512$$

where:

Cnum	is the number of characters per bucket as given in the BLOCK CONTAINS clause.
Rlen	is the fixed record length (in bytes).
Rmax	is the maximum record length (in bytes) if the record length is variable.
Bnum	is the number of virtual blocks per bucket, ranging from 1 to 32.

Violation of constraint (1) causes a fatal error and the default method is used to calculate the bucket size. Constraint (2) means that Cnum should be a multiple of 512. If not, a warning error is given and Cnum is increased to the next even multiple of 512.

The bucket size must be the same when the file is created and each time the file is accessed. Therefore, the BLOCK CONTAINS clause must never change for a particular file.

Note: The previous discussion has used the following format:

$$\left[\underline{\text{BLOCK}} \text{ CONTAINS integer} \quad \left\{ \begin{matrix} \underline{\text{RECORDS}} \\ \text{CHARACTERS} \end{matrix} \right\} \right]$$

If the following format is used:

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

the compiler ignores integer-1, and integer-2 is used as the integer.

6.3.5 Buffering

When the system performs a sequential or random input operation, one or more index buckets are read into the buffer area until the bucket containing the specified record is located. The bucket containing the record is then read into the buffer area. Any subsequent sequential read operations will use the current index buffer to locate and read subsequent records in the current or other record buckets. When it has exhausted the current index buffer (has read all the records identified in the bucket), the system reads the next index bucket into the buffer area.

When performing a sequential or random output operation, the system moves a record from the files current record area into the files buffer. Each subsequent write operation moves a record from the current record area into the buffer. The system writes the bucket to the medium when it has filled the buffer. Every output operation also causes the appropriate index bucket to be read into the buffer area, the indexes for each of the keys to be added to the appropriate buckets, and the buckets to be rewritten to the storage medium.

6.3.5.1 Buffer Size - Buffer size depends on the size of the largest record in the file and on the blocking factor. For indexed files, buffer size must be some multiple of 256 words (512 bytes).

6.3.5.2 I/O Buffer Areas - The RESERVE clause in the Environment Division specifies the number of I/O buffer areas to be allocated for each file. Each I/O area represents the space for one bucket. A minimum of two is required for an Indexed file (this is the default). Three areas will increase the speed of random access. Four areas will increase the speed of random access if the file is being accessed on two different keys. For each additional key, an additional area will increase the speed of access. Therefore, to speed up random access

time, the optimum number of buffer areas is equal to the number of
keys by which the file is being accessed plus two. Of course, each
area means that more memory space is being taken up.

6.3.5.3 Buffer Space - To calculate the total amount of buffer space
required for each Indexed file, the following algorithm may be used:

    Buffer Space = record size+((bucket size+20)*no. of areas)

    +(48*no. of keys in file)+((MAXKSIZ*2+MAXNKEY+3)/4*4)

    +272

where:

    MAXKSIZ   is the maximum key size in the program.

    MAXNKEY   is the maximum number of record keys for any file in
               the program.

Note that in the division, the result is truncated to the next lowest
integer.

In addition to the above, there are 76 bytes of buffer space that are
shared among all files and 44 times MAXNKEY bytes of buffer space that
are shared among all indexed files.

6.3.5.4 , Sharing Buffer Space Among Files - The SAME AREA clause
provides a simple method of sharing buffer space among several files.
This clause is identical for all file organizations and is described
in Section 6.1.6.4.

6.3.6 Indexed I/O Statements

The COBOL I/O statements, CLOSE, DELETE, OPEN, READ, WRITE, REWRITE,
and START can refer to indexed files.

A COBOL program may open an indexed file in one of three modes, INPUT,
OUTPUT, or I-O, and access an open indexed file in one of three ways,
sequentially, randomly, or dynamically. In INPUT mode, records may be
read from the file; in OUTPUT mode, the file is created and records
may be written to the file; in I-O mode, records may be read from the
file, updated on the file, deleted from the file, or written to the
file. The following table shows which statements and access methods
apply to the three different OPEN modes of indexed files.

Table 6-7
Indexed OPEN Modes

File Access Mode	Statement	Open Mode		
		Input	Output	I-O
Sequential	DELETE READ REWRITE START WRITE	X X	 X	X X X X
Random	DELETE READ REWRITE START WRITE	X	 X	X X X X
Dynamic	DELETE READ READ NEXT REWRITE START WRITE	 X X X	 X	X X X X X X

NOTE

The term, current record pointer, used in the following sections, refers to a location in the operating system used to store the record number of available record in a file.

6.3.6.1 Access Mode - The ACCESS MODE clause in the File-Control paragraph indicates which of the three access modes may be used on that file. When the ACCESS MODE clause specifies SEQUENTIAL, the I/O statements must refer to the records in the file sequentially, starting (after opening) with the first record and stepping through with each reference to the end of the file.

When the ACCESS MODE clause specifies RANDOM, the I/O statements refer to the records in the file by the value of the key or keys. Usually the prime key is used unless a specific alternate key is designated. If an I/O statement refers to a record with a key, and that record does not exist, the INVALID KEY imparative statement may be executed depending on the particular I/O statement used. (The INVALID KEY imperative statement is explained with each of the indexed I/O statements in this section.)

When the ACCESS MODE clause specifies DYNAMIC, the I/O statements may refer to the records in the file either sequentially or randomly. The OTS determines which access method to use from the OPEN mode (INPUT, OUTPUT, or I-O) and the form of the I/O statement. For example, if the statement READ ...INVALID KEY is to access an open input file dynamically, the OTS uses the designated key for random access. If the statement READ NEXT ... is to access an open input file dynamically, the OTS sequentially accesses the next existing record.

The following sections (6.3.6.2 through 6.3.6.8) on using the I/O statements themselves contain additional information about access modes.

6.3.6.2 **Opening Indexed Files** - The OPEN statement for an indexed file makes an INPUT, OUTPUT, or I-O mode file available so the COBOL program can access the records in the file sequentially, randomly, or dynamically. Consider the following example:

Example

The following sample OPEN statement opens the file named ARTICHOKE for input, and sets ARTICHOKE's current record pointer to the first record in the file. The program containing this statement could, after executing the statement, access ARTICHOKE with READ and START statements in the sequential access mode, READ statements in the random access mode, or READ, READ NEXT, and START statements in the dynamic access mode.

> OPEN INPUT ARTICHOKE.

When the OTS executes an OPEN statement, it performs the following actions for the file named in the statement:

- If the file is already open, the OTS generates an error message and performs the USE procedure section (if specified). (Section 6.9 discusses USE procedures and Section 12.3 discusses error messages.)

- When opening an existing file, the attributes (i.e., record length, block size, etc.) of the file are used for accessing the file. Those specified in the program are ignored.   Be sure that the attributes specified in the COBOL program agree with the actual attributes of the file.

- If a SAME AREA clause contains the name of the file and none of the other files named in the clause is open, the OTS allocates buffer space for the file.

- When the file has passed all of the preceding checks and is ready for opening, the OTS instructs the Record Management Services to open the file. If the Record Management Services fails to open the file, the OTS reports an error condition and performs any applicable USE procedure (if present).

- Finally, depending on which statements apply to the open mode, the OTS enables or disables all of the program's I/O statements that refer to the file (see Table 6-7).   For example, if the OPEN mode is INPUT, it enables all READ and START statements for that file and disables all REWRITE, DELETE and WRITE statements for that file.

The OPEN statement sets the current record pointer for the file to the first existing record in the file as established by the prime record key. If the file is being accessed randomly or dynamically, the program should maintain correct values in the prime and aternate key fields.

6.3.6.3 Reading Indexed Files - When applied to a file being accessed sequentially, the READ statement makes the next logical record·of an open file available to the program. The information made available is based on positioning by the OPEN, START, or last READ operation.

When applied to a file being accessed randomly, the READ statement selects a specified record from an open file and makes it available to the program. The value of the specified key (prime key, if the no key is specified) identifies the record.

When applied to a file being accessed dynamically, the READ statement has two formats so that it can either select the next logical record (sequentially) or select a specified record (randomly) and make it available to the program. The READ NEXT statement takes the number in the current pointer and finds the next present record. The following sample READ statement reads the file named ARTICHOKE sequentially and, when it exhausts the file, causes program control to transfer to the subroutine named FILEOUT:

        READ ARTICHOKE NEXT RECORD
            AT END GO TO FILEOUT.

For more information concerning the mechanics of the READ NEXT statement see Section 6.3.6.6, Specifying the Next Record To Be Read.

The READ with key takes the value in the specified key, moves it to the current record pointer, and reads the record being pointed to. The following READ (with key) statement reads the file named ARTICHOKE randomly, selecting records through the value in the file's primary key. If the designated key supplies a value that is not identified with a valid record, the statement causes program control to transfer to the subroutine named NO-REC.

        READ ARTICHOKE RECORD

        INVALID KEY GO TO NO-REC.

Note: a random read repositions the current record pointer and thus effects further sequential reads.

If the file has more than one record description, the records automatically share the same current record area. The OTS does not clear this area before it executes the READ statement (no blank filling, etc.). Therefore, if the record read by the latest READ statement does not fill the entire current record area, the area not overlaid by the incoming record remains unchanged. For example, if the file's record area contains ten 3's, and a READ operation moves in a 6-character record containing all 1's, the current record area then contains six 1's followed by four 3's. Consider the following example:

Current Record Area with all 3's | 3333333333 |

Next Record in the File | 111111 |

Current Record Area after READ | 1111113333 |

6.3.6.4 Rewriting Records into an Indexed File – The REWRITE statement releases a logical record to an output or input-output file. In all of the access modes, the record is positioned based on the prime key, any alternate keys are also processed properly, including

duplicate keys. If more space is required for new record positions, the Record Management Services automatically extends the file size, regardless of the access mode being employed.

If the file is open in sequential access mode and the records are not written in ascending order of the prime key values, an INVALID KEY condition exists. In any access mode an attempt to write an existing record having the same prime key value or an alternate key value where duplicates are not allowed, results in an INVALID KEY condition.

The following sample WRITE statement releases the record BREAKERS to the indexed file. If the record already exists, program control transfers to WRITE-ERR.

        WRITE BREAKERS
            INVALID KEY GO TO WRITE-ERR.

The indexed WRITE statement creates only storage files because print-files are sequential files.

6.3.6.5 **Deleting Records from an Indexed File** - The DELETE statment logically removes an existing record from a file. After a DELETE statement has successfully removed a record from a file, that record can no longer be accessed.

If the file is open in the sequential access mode, the statement removes the record just successfully read. For example, the following sample statement removes the record just read from the file named ARTICHOKE:

        DELETE ARTICHOKE RECORD.

If the file is open in either the random or dynamic access mode, the statement removes the record from the record specified by the prime key. For example, the following sample statment deletes the record specified by the prime key from the file named ARTICHOKE. If the prime key supplies a value that does not contain a valid record, the statement transfers control to NO-REC.

        DELETE ARTICHOKE RECORD
            INVALID KEY GO TO NO-REC.

6.3.6.6 **Specifying the Next Record to be READ** - The START statement specifies which record will be the next record to be referenced sequentially in a file opened for INPUT or I-O processing. The START statement updates the current record pointer for future sequential READs.

Suppose we have the following START statement:

        START FILE-A KEY IS EQUAL TO SUB-KEY-A.

SUB-KEY-A must be alphanumeric. In addition, SUB-KEY-A must be a record key or alternate record key or subordinate to a record key or alternate record key whose leftmost character position corresponded to its own leftmost character position. For example, if the following fields were defined in the record:

```
02 KEY-A.
      03 SUB-KEY-A.
            04 SUB-KEY-A1 PIC XXX.
            04 SUB-KEY-A2 PIC XX.
      03 SUB-KEY-B PIC XXX.
```

and if KEY-A was a record key or alternate record key, then the following would be legal START statements:

    START FILE-A KEY IS EQUAL TO KEY-A.

    START FILE-A KEY IS EQUAL TO SUB-KEY-A.

    START FILE-A KEY IS EQUAL TO SUB-KEY-A1.

The following START statements are illegal.

    START FILE-A KEY IS EQUAL TO SUB-KEY-A2.

    START FILE-A KEY IS EQUAL TO SUB-KEY-B.

The leftmost character positions of SUB-KEY-A2 and SUB-KEY-B do not correspond to the leftmost character position of KEY-A.

The relational operator IS EQUAL TO (or IS =) means that the current record pointer is set to point to the record associated with the first key equal to SUB-KEY-A. If SUB-KEY-A is shorter than the record key or alternate record key, then the record keys or alternate record keys in the file are truncated on the right to the same length as SUB-KEY-A for the purposes of the comparison.

If the following START statement is used:

    START FILE-A KEY IS GREATER THAN SUB-KEY-A.

            or

    START FILE-A KEY IS > SUB-KEY-A.

then the current record pointer is set to point to the record associated with the first key that is greater than SUB-KEY-A. Thus, if the file had records with the following keys:

Record #	743	629	015	891	233	371
KEY-A	ABCDDZZX	ABCDEABC	ABCDEXYZ	ABCDEZZZ	ABCDGAAA	ABCDGZZX

and SUB-KEY-A contained ABCDE, then the current record pointer would be set to point to record number 233.

If the following START statement is used:

    START FILE-A KEY IS NOT LESS THAN SUB-KEY-A.

            or

    START FILE-A KEY IS NOT < SUB-KEY-A.

then the current record pointer is set to point to the record associated with the first key that is greater than or equal to SUB-KEY-A. In the previous example that would be record number 629.

If there is no record that satisfies the comparison, the invalid key exit is taken. In our example the following statement:

    START FILE-A KEY IS EQUAL TO SUB-KEY-A.

would take the invalid key exit if SUB-KEY-A contained ABCDF.

If the comparison is satisfied and the current record pointer is set, then subsequent READs would update the current record pointer using KEY-A as the key of reference.

If the key phrase is not specified, then the default key is the prime record key and the default comparison is IS EQUAL TO.

6.3.6.7 **Closing Indexed Files** - The CLOSE statement terminates processing on the file referred to in the statement. The following sample CLOSE statement terminates processing on the file named ARTICHOKE:

    CLOSE ARTICHOKE.

When the statement closes a file, no other I/O operation can access that file until another OPEN statement opens the file. If the statement specifies the LOCK option, the program cannot open the file again in that run. If a SAME AREA clause contains the name of the file just closed, the program may open one of the other files named in the clause.

6.4 **DEVICES**

The PDP-11 COBOL object time system supports any devices supported by the Record Management Services. Table 6-8 contains a partial list of these devices:

Table 6-8
Device Codes

Device	Device Code
Card Reader	CR
Disk (RK03/RK05)	DK
Disk (RF11/RS11)	DF
Disk (RS03/RS04)	DS*
Disk (RP11/RP02/RP03)	DP
Disk (RJP04)	DB
Line Printer	LP
Magnetic Tape (TU10/TS03)	MT
Magnetic Tape (TU16/TU45)	MM

---

*The DS device code applies to non-RSTS/E only.

Some devices are better suited to certain uses than others. For example, since PDP-11 COBOL is a disk-oriented system, the disk provides COBOL files with the best performance and reliability. On the other hand, COBOL files on magnetic tape are limited to sequential organization.

The following subsections discuss the devices that are available and how to use them to best advantage.

6.4.1 Disk

The primary means for storage and processing of PDP-11 COBOL files is a disk. Several disk units are supported, including RK05, RF11, RP11/RP03, RP04 and RS04. Each device has its own file handling characteristics, and differs with respect to capacity, speed, and portability. The following table compares these characteristics. The values, low, mod (moderate), and high, describe the relative characteristics of the devices in the table. The efficiency characteristic commbines the values of capacity, speed, and portability for that unit into a single value.

Table 6-9
Comparison of PDP-11 Disk Devices

Device	RK05	RF11	RP11/RP03/RP04	RS04*
CAPACITY	LOW/MOD (4800 BLOCKS)	LOW (1024 BLOCKS)	VERY HIGH (80,000 BLOCKS) (RP04 160,000 BLOCKS)	LOW (2048 BLOCKS)
SPEED	MOD	HIGH	HIGH	HIGH
PORTABILITY	HIGH (EASY)	NONE	HIGH (BULKY)	NONE
EFFICIENCY	HIGH	MOD	HIGH	MOD

The characteristics of the devices in the table suggest suitable applications. Consider the following examples:

● The portable, moderate capacity RK05 is ideal for storage of COBOL source and object files as well as small data files;

● The fast, low capacity RF11 is ideal for scratch files requiring either random or sequential access;

● The high-capacity RP11/RP03/RP04 is excellent for large files requiring high volume access in either the random or sequential modes. Further, its portability makes it ideal for master file storage on multiple systems.

---

*RSTS/E supports the RS04 only as a swapping device. It cannot be used for user files.

6.4.2 Magnetic Tape

All PDP-11 operating systems support magnetic tape files; all COBOL operations concerned with magnetic tape are fully supported by the compiler, including the MULTIPLE-FILE TAPE clause and the CLOSE REEL [WITH NO REWIND] clause. (RSTS/E does not support multi-reel files.)

6.4.3 Card Reader and Line Printer

COBOL programs can use both the card reader and the line printer as I/O devices.

If these devices have been assigned logical names in the Special-Names paragraph of the Environment Division, the ACCEPT and DISPLAY statements can access them. For example, consider the following coding:

```
ENVIRONMENT DIVISION.

    ...

SPECIAL-NAMES.
    CARD-READER IS CARDS
    LINE-PRINTER IS LOG.

    ...

PROCEDURE DIVISION.

    ...

    ACCEPT INREC FROM CARDS.

    ...

    DISPLAY PRINTREC UPON LOG.

    ...
```

Figure 6-5 Assigning Logical Names to the
Card Reader and Line Printer

If filenames have been assigned to these devices in the SELECT clause of the File-Control paragraph, the READ and WRITE statements can access them for I/O files. For example, consider the following coding:

```
      ...

FILE-CONTROL.
    SELECT INFILE ASSIGN TO "CR:".
    SELECT OUTFILE ASSIGN TO "LP:".


    ...

FD OUTFILE
    DATA RECORD IS OUTREC.


    ...

PROCEDURE DIVISION.

    ...

    READ INFILE.

    ...

    WRITE OUTREC.
```

Figure 6-6 Assigning the Card Reader and
Line Printer to Files

6.5 FILES AND FILENAMES

The OTS and the operating system use the device codes described in
Section 6.4 to communicate with the devices. Further, the COBOL OTS
uses the operating system's file specification and interfaces for all
file manipulation with file storage devices (disk and magtape). The
VALUE OF ID clause (discussed in the following subsection) in the FD
entry describes the file specification to the OTS. The format for the
full file specification follows:

        dev:[uic] filename.typ;version/switches

where:

        dev: - device code

        [uic]   - user's identification code or the code of the user
                     for whom the file was created - the user directory
                     ID. (The brackets [ ] are required.)

        filename   - an alphanumeric field containing up to nine
                     characters that identifies the file (RSTS/E allows a
                     length of from one to six characters).

        typ   - an alphanumeric field containing up to three
                     characters that qualify the filename.

        version   - a numeric field containing up to five octal digits
                     that give the version number of the file. By
                     specifying version numbers, the user can maintain
                     several versions of the same file on a directory
                     device. (Not available with RSTS/E.)

        switches   - identifies certain actions for the operating system
                     to perform for the file. (Subsection 6.5.1.1
                     discusses these switches.)

These entries default as follows:

    dev: - the device code of the disk containing the operating
                   system.

    [uic]   - the user identification code of the user currently
                   using the system.

    filename   - null

    typ   - null

    version   - the version number defaults differently for IAS and
                   RSX-11M input and output files.

                   ● input files - the highest numbered version of the
                     file (thus selecting the latest version);

                   ● output files - one greater than that of the
                     highest numbered version of the file (thus
                     creating the latest version).

    switches   - null

For example, the following sample file specification causes the file
system to process version 3 of a file on disk named ARIES. The user
has an identification code of 140,222.

    DK:[140,222]ARIES;3

The following sample RSTS/E file specification causes the file system
to process a file on disk named ARIES. The user has an identification
code of 140,222. Note that RSTS/E does not support the version number
feature.

    DK:[140,222]ARIES.

6.5.1 Using Explicit Filenames (VALUE OF ID Clause)

The VALUE OF ID clause, in the FD entry, describes the file
specification to the COBOL OTS. The VALUE OF ID clause is optional;
however, the system requires it whenever the program refers to an
explicit file unless a sufficient file description is provided in the
ASSIGN clause. The clause accepts either a literal entry or an
identifier entry.   Consider the following sample literal form of the
clause:

    VALUE OF ID IS "DK:[140,222]ARIES;3"

Elements of the file specification appearing in the VALUE OF ID clause
supersede their counterparts specified in the ASSIGN clause for the
file.   (Subsection 6.5.2 discusses the ASSIGN clause.)

When written in the literal form, the literal may be a complete file
specification or a part of a file specification.

When written in the identifier form, the value of the identifier may
be a complete or partial file specification.

The identifier form of this clause is especially useful when different
runs of a program process different files. If a program must process
different files in the same way on different runs, an ACCEPT statement

in the Procedure Division can request a file specification from the user at the user's console or from a batch input stream.

The following example illustrates how a COBOL program could request a file specification from an interactive terminal:

```
    DATA DIVISION.

        .

        .

        .
    FD FILEIN
        VALUE OF ID IS INFILE.

        .

        .

        .
    WORKING-STORAGE SECTION.
        77 INFILE PIC X(20).
    PROCEDURE DIVISION.

        .

        .

        .
        DISPLAY "TYPE IN INPUT FILE SPEC".
        ACCEPT INFILE.

        .

        .

        .
        OPEN INPUT FILEIN.
```

This sample coding causes the following interaction between the program and the user (the message printed by the program is underlined):

    TYPE IN INPUT FILE SPEC
    DK1:THOREAU   (RET).

Following this interaction, the sample OPEN statement will open (for input) the file, THOREAU on DK1.

6.5.1.1 **Switches** – There are four optional switches which may qualify the file specification. These switches modify the processing performed by the COBOL OTS when it opens the file.

Table 6-10 contains a list of the file switches and their meanings to the OTS.

Table 6-10
File Specifier Switches

SWITCH	MEANING
/CL:n	Allocate disk space in clusters of n virtual blocks whenever the file needs additional storage space during output operations. (n may be any number from 1 to 256 in powers of 2.) If the number is followed by a decimal point, the software considers the number to be decimal; if it does not have a decimal point, it is considered to be octal). This switch would be used only when very large files are to be created and the output device can hold the entire file (i.e., an RP03 disk). The effect of this switch is to make file accessing faster when the file is being processed sequentially.
/CO:n	(Not supported by RSTS/E) allocate a contiguous file of n disk blocks to the file when it is opened. This switch ensures that n blocks are available for the file prior to actual processing. (When many users are sharing the same disk, you can use this switch to ensure that your entire file will fit on the disk.) It applies only to output files being created. (If n applies to an RK type disk, the maximum value of n is 4000 (decimal). If a decimal point follows the number, the system considers it to be decimal; otherwise, octal.)
/SH	This file is shared for output or I/O mode, available for writing or altering by other tasks (or jobs) running concurrently with the COBOL program. The /SH switch must not be specified for sequential files. For all other files, the following rules apply. ● The /SH switch should be used consistently among concurrently executing tasks. That is, if the /SH is specified for one task sharing the file, all tasks sharing the file should have it specified, and vice-versa. ● If a file is being opened for OUTPUT or I/O with the /SH switch specified, all other tasks currently using the file must also have the /SH switch specified. ● If a file is being opened for input without the /SH switch set, no other task can be using the file for output or I/O. ● If a file is being opened for INPUT and no /SH switch is specified, all other tasks currently using the file should not have the /SH switch specified. If access is denied when the file is opened because of one of the above reasons, a file status code of 91 is stored in the FILE-STATUS data-item associated with the file if one is specified.
/AL:n	Same as the /CO:n switch with the following exception. The /CO:n specifies that all blocks be contiguous, and the /AL:n switch specifies that all blocks need not be contiguous.

6.5.2 Device Assignment by ASSIGN Clause

If the VALUE OF ID clause does not specify a complete file
specification, the ASSIGN clause in the File-Control paragraph can
assign a defalt to those components not specified. The ASSIGN clause
must be written as part of the SELECT statement as shown below:

    SELECT THOREAU ASSIGN TO "DK1:"

This example assigns a default device code "DK1:" for the location of
the file THOREAU. Another device code specification in the VALUE OF
ID clause could override it later in the source program.

6.5.3 Files and Logical Units

Each file in an executable task must have a unique Logical Unit Number
(LUN) assigned to it. The COBOL compiler can only generate a relative
LUN assignment for each file in a COBOL program, because there may be
multiple COBOL programs in a task. (See Figure 2-10 which contains a
sample file-to-relative-LUN assignment table.) Actual LUN assignments
are made by the COBOL Object Time System (OTS) at task execution time.
The number of LUNs needed by a task is equal to 1+n, where n is the
total number of individual files included in each program comprising
the task. For example, if a task consists of three programs, each
program requiring three files, then the number of LUNs required is 10.
(The first LUN is reserved for ACCEPT/DISPLAY and message
processing.) If more than six LUNs are required for an executable
task, the UNITS option must be specified at task-build time, because
the Task Builder default is 6.

Each LUN must have a physical device associated with it before the
associated file can be opened. You can assign a physical device to
the file by specifying the VALUE OF ID or ASSIGN clause in your COBOL
program, or you can specify the ASG option at task-build time.

                              NOTE

            The default LUN assignments generated by
            the Task Builder do not always equate to
            the system device.

(Refer to the Task Builder Manual for your particular operating system
for more information concerning task builder options.)

As previously stated, each COBOL program receives relative LUN
assignments for its files by the compiler. At task-execution time,
the OTS converts these relative LUN assignments to actual assignments
according to the following rules:

1. If the task consists of only one COBOL program, the OTS adds
    1 to each of the relative LUN assignments yielding the actual
    assignments. Therefore, a file receiving a relative LUN
    assignment of 2 by the compiler would receive an actual LUN
    assignment of 3 at execution time.

2. If the task consists of more than one COBOL program having
    files assigned to it, simply adding 1 to the relative LUN
    assignments would obviously yield duplicate actual LUN
    assignments. The OTS, in the case of multiple program tasks,
    utilizes the relative assignment +1 formula for the first

program in the task. For each subsequent program, it takes the highest actual LUN assignment for the previous program and adds 1 to it to arrive at the first LUN assignment. It then applies the +1 formula to this first LUN assignment to arrive at each subsequent assignment for the program. Consider the following example:

Example

A task consists of three programs (PROGA, PROGB, and PROGC). Each program has three files with relative LUN assignments of 1, 2, and 3. At execution time, assuming that the programs were presented to the Task Builder or Merge Utility in the order PROGA, PROGB, and PROGC, the OTS would assign actual LUNs as follows:

Program	LUN assignment
	1 (reserved for ACCEPT/DISPLAY and message processing)
PROGA	2 1st. File 3 2nd. File 4 3rd. File
PROGB	5 1st. File 6 2nd. File 7 3rd. File
PROGC	8 1st. File 9 2nd. File 10 3rd. File

6.6 OPTIMIZATION

At times a user may wish to optimize his program with regards to space or time. Often, there is a trade-off between the two. The default philosophy of the COBOL compiler has been to optimize space allocation. A discussion of the two types of optimization follows.

6.6.1 Speed Optimization

The following COBOL clauses and phrases may be used to increase execution speed.

SAME RECORD AREA Phrase

Default	- No Same Record Area
Optimal	- Use SAME RECORD AREA phrase
Effect	- May save compute time. If records are being copied from one file to another and if both files share the same Record Area, then no move statement is needed to move the records from one record area to the other.
Use more space?	- No, uses less space
Other potential drawbacks	- Records from both files will not be available simultaneously (unless one is moved so it can be saved), because one record will wipe out the other.

BLOCK CONTAINS Clause

Default	- Bucket or logical block consists of smallest number of virtual blocks.
Optimal	- Make bucket or logical block as large as possible.
Effect	- Speeds sequential access by reducing amount of I/O to disk.
Use more space?	- Yes
Other potential drawbacks	- None

For indexed files, the following clauses and phrases may be used to further increase execution speed.

RESERVE Clause -

Default	- 2 AREAS
Optimal	- Make the number of areas equal to the number of keys of access + 2.
Effect	- Speeds up random access.
Use more space?	- Yes
Other potential drawbacks	- None

WITH DUPLICATES Phrase -

Default	- Duplicates not allowed
Optimal	- Allow duplicates by using this phrase

Effect   - Speeds up WRITE and REWRITE time. If
                                      this phrase is not used, then the
                                      program must check each alternate
                                      record key on a write or rewrite, to
                                      check that it doesn't already exist
                                      on the file.

Use more space?   - No

Other potential drawbacks - Duplicate keys are allowed and this
                                      may not be wanted. For example, if
                                      the social security number is a key,
                                      duplicate social security numbers may
                                      be illegal.

6.6.2 Space Optimization

The default philosophy is to optimize space usage.   However, the
following COBOL clauses and phrases may be used to further reduce
space requirements.

SAME RECORD AREA

Default   - No Same Record Area

Optimal   - Use SAME RECORD AREA phrase for as
                                      many files as possible.

Effect   - Instead of having a record area for
                                      each file, all the files use the same
                                      record area.

Slows execution time   - No

Other potential drawbacks - Records from both files will not be
                                      available simultaneously (unless one
                                      is moved in order to save   it),
                                      because one record will wipe out the
                                      other.

SAME AREA

Default   - No Same Area

Optimal   - Use SAME AREA phrase for as many
                                      files as possible.

Effect   - The buffer areas for all the files in
                                      the SAME AREA phrase are shared.
                                      Saves much more space than SAME
                                      RECORD AREA.

Slows execution time   - No

Other potential drawbacks - Not more than one of the files listed
                                      in the SAME AREA phrase may be open
                                      at any time.

6.7 COMMUNICATING WITH THE PROGRAM

The ACCEPT and DISPLAY statements allow low-volume, terminal-oriented interaction between a COBOL program and the user of the program.

While these statements are primarily intended for use with keyboard devices, PDP-11 COBOL allows the ACCEPT statement to accept cards from a card reader, and the DISPLAY statement to display data on a line printer. The following two Sections (6.7.1 and 6.7.2) discuss these capabilities in greater detail.

6.7.1 Using the ACCEPT Statement

The ACCEPT statement makes small amounts of data available to the specified data item.

Consider the following example; it causes data to be transferred from the device identified by the mnemonic-name, OPERATOR, to the area represented by the identifier, COMM-AREA.

ACCEPT COMM-AREA FROM OPERATOR.

OPERATOR must be a mnemonic-name specified for a device in the Special-Names paragraph (in this example, possibly the console). The area represented by the identifier COMM-AREA receives the data requested without any editing.

The ACCEPT statement causes the transfer of a stream of bytes from the device specified in the FROM phrase, if it is present (OPERATOR in the previous example). If the FROM phrase is not present, the data is transferred from the user's console.

Enough bytes are transferred to fill the identifier's area. The size of COMM-AREA in this example dictates the number of bytes being transferred.

If the device contains more data than there are bytes in COMM-AREA, the data is truncated.

- If the length of the identifier exceeds 80 bytes, the OTS performs one or more additional transfers of data until it either fills the identifier or transfers less than 80 bytes.

- If the length of the identifier is less than or equal to 80 bytes and the length of the data is less than the identifier on a teletype or cards, the OTS pads the identifier with blank characters.

The ACCEPT statement has a second format that allows it to retrieve the current DAY, DATE, or TIME from the system and store it in the specified identifier. (DAY, DATE, and TIME are reserved words that the user does not define. The user must define identifiers into which to accept the values in DAY, DATE, or TIME.) The following sample statement places the current date in the identifier, GREENWICH:

ACCEPT GREENWICH FROM DATE.

DAY and DATE are treated as equivalent. A facility for specifying the day of the year is currently not provided. The date, however, is provided as follows (YY is the year; MM is the month; DD is the day):

DATE -- YYMMDD

If the date were July 4, 1976, the systems would provide GREENWICH with the number 760704.

The systems provide the time as follows (HH is the hour; MM is the minutes; SS is the seconds; CC is the hundredths of a second):

TIME -- HHMMSSCC

If the time were 20 seconds after 5:15 PM, the systems (which have a 24-hour clock) would provide the numbers 17152000. (Since the PDP-11 clock has no hundredths of a second capability, the systems place zeroes in the last two positions.)

The identifier receives the data according to the rules for the MOVE statement. Chapters 3 and 4 discuss the MOVE statement as applied to non-numeric fields (Chapter 3) and numeric fields (Chapter 4).

6.7.2 Using the DISPLAY Statement

The DISPLAY statement transfers small amounts of data from he specified data item or literal to the specified device.

Consider the following example; it causes the transfer of data from the area represented by the identifier, COMM-AREA, to the device with the mnemonic-name, OPERATOR:

DISPLAY COMM-AREA UPON OPERATOR.

OPERATOR must be a mnemonic-name specified for a device in the Special-Names paragraph (possibly the console in this example). The area represented by the identifier, COMM-AREA, contains the data being transferred.

The DISPLAY statement causes the transfer of a stream of bytes to the device specified in the UPON phrase if it is present (OPERATOR in the preceding example). If the UPON phrase is not present, the OTS transfers the data to the user's console.

All of the bytes in all of the identifiers or literals in the DISPLAY statement are transferred first. The size of COMM-AREA, in this example, dictates the number of bytes being transferred.

The system does not convert COMP items from binary to ASCII; it simply transfers them as they exist in storage.

If a single DISPLAY statement must transfer large amounts of data, that data must contain appropriate vertical and horizontal form control characters. If the data being transferred does not contain form control characters and the length of the data stream exceeds the device's single line capacity, the excess characters will all print in the last position (overprinting each other).

Table 6-11 contains several of the terminal form control characters:

Table 6-11
Form Control Characters

Octal Code	Control Character	Function
007	BEL (CTRL G)	Bell ringer
011	HT (CTRL I)	Horizontal tab
012	LF (CTRL J)	Line feed or line space (new line)
013	VT (CTRL K)	Vertical tab
014	FF (CTRL L)	Form feed to head of form
015	CR (CTRL M)	Carriage return

When it has transferred all of the data from all of the items listed
in the statement, a carriage return and linefeed character are
automatically appended onto the data. The WITH NO ADVANCING phrase
suppresses this appending operation.

NOTE

The WITH NO ADVANCING phrase is an
extension to the ANS-74 standard.

6.8   FILE COMPATIBILITY WITH OTHER PROGRAMMING LANGUAGES

All files generated by other programming languages are compatible with
COBOL provided that they were generated using Record Management
Services. Files generated by other file systems must conform to
Record Management Services formats.

6.8.1   Writing Files For Other Programming Languages

PDP-11 COBOL writes files that can be read only by languages using the
Record Management Services system interface for user program I/O.
When creating a file that is to be read by a language, that does not
use the Record Management Services interface (i.e., BASIC-PLUS),
adhere to the following restrictions:

- Ensure that the file has sequential file organization.

- Ensure that the file is not a COBOL print-file (no LINAGE or
  APPLY PRINT-CONTROL clauses are applicable to the file).
  Printer control is handled differently by each PDP-11
  programming language.

- Do not use the ADVANCING option in WRITE statements when
  creating the file.

The file may contain fixed-length or variable-length records, and the
records should only contain only printable ASCII character data.

6.8.2 Reading Files Written in Other Programming Languages

PDP-11 COBOL reads files that were written only by languages using the Record Management Services system interface for user program I/O. Before reading a file that was written by another language that does not use the Record Management Services interface, be certain that the file meets the following restrictions:

- Ensure that the file is an ASCII file.

- Ensure that the file does not have a carriage control attribute (the FORTRAN carriage control file attribute must not be set).

FORTRAN meets these restrictions when it writes ASCII (not binary) data with formatted WRITE statements. However, the user must disable the carriage control attributes in the OPEN statement for the file.

```
CALL OPEN (UNIT=n, CARRIAGE CONTROL="NONE"
           .
           .
           .
WRITE (n,100) list

FORMAT (.....)
```

BASIC+2 is capable of reading and writing all Record Management Services files. Therefore, files written by BASIC+2 programs are compatible with COBOL.

BASIC-PLUS meets all of these restrictions when it writes a formatted ASCII (sometimes called sequential) file as described in the BASIC-PLUS Language Manual. PDP-11 COBOL cannot read BASIC-PLUS Virtual Array files.

6.8.3 Data File Transportability

The user who wishes to transport data files from one language processor to another or from one system to another (RSX-11M to RSTS/E or vice versa) should be careful to write such files using the Record Management Services. Record Management Services is the only file interface used by PDP-11 COBOL.

Non-printable ASCII characters are subject to misinterpretation by the different language processors and operating system utilities. If, for example, COBOL were to write records which contained COMPUTATIONAL (binary) data items, the values these items could contain would be written in the file in the same binary format as represented in the computer. Such binary values may look like non-printable ASCII characters such as CR, LF, CTRL/Z, escape, which could cause system utilities to perform in an unpredictable manner while processing the records.

Other ways that non-printable ASCII characters can get into a file are:

1. having data definitions that contain the USAGE IS INDEX clause;

2. moving HIGH-VALUES or LOW-VALUES;

3. moving any redefinition of a COMP or USAGE IS INDEX field;

4. reading a data file that contains non-printable ASCII characters;

5. having multiple record definitions of varying sizes and filling a shorter record area then writing a longer one. (The excess characters, not filled, may be non-printing.)

This list is not complete. There are many other ways for non-printing ASCII characters to find their ways into printable ASCII files.

6.9 PROCESSING I/O ERRORS - USE STATEMENT

The USE statement provides COBOL programs with a way to process I/O errors. It allows the program to specify possible recovery steps following the I/O handling procedures performed by the software.

When a COBOL program contains a USE procedure and an I/O error occurs, the OTS and Record Management Services execute their standard I/O error handling procedures and then transfer control to the procedure following the USE statement. (For further information concerning run-time I/O errors, see section 12.3.)

Consider the following sample coding. When either THOREAU or ARTICHOKE causes an I/O error, the OTS executes its standard I/O error procedures and then transfers control to the paragraph (or paragraphs) that follow the USE statement.

```
PROCEDURE DIVISION.
DECLARATIVES.
REPAIR SECTION.
    USE AFTER STANDARD ERROR PROCEDURE
              ON THOREAU ARTICHOKE.
DISPLAY-ERROR.
    IF ...
```

The paragraphs following the USE statement may contain any valid COBOL statement, except for the following:

1. Those statements that refer to a procedure outside of the DECLARATIVES. (Any attempt to transfer control out of the DECLARATIVES causes the OTS to abort the program.)

2. Those statements that would cause the USE procedure being executed to be invoked again. (Recursive USE procedures cause the OTS to abort the program.)

   USE procedures are executed in the same manner PERFORM ranges in Procedure Division coding. Therefore, paragraphs with the USE procedure section should follow all rules specified for paragraphs within PERFORM ranges. For further information on PERFORM ranges, see Use of the PERFORM Statement in Chapter 7 of this guide.)

   If a status key is declared for the file in error, all status information is made available for processing in the USE procedure.

CHAPTER 7

GOOD PROGRAMMING PRACTICES

7.1 FORMATTING THE SOURCE PROGRAM

Since most COBOL programs are usually long, the programmer needs techniques that will help him to simplify and improve the readability of his COBOL programs. The guidelines in this chapter, if followed, will help produce source programs that are easy to read and maintain.

Before considering these guidelines, consider the reference formats that are available with PDP-11 COBOL:

1. the Conventional (ANS) format, and

2. the Terminal format.

Although the Conventional format produces ANS compatible programs, it also produces source printouts that are somewhat more cluttered than those produced by the Terminal format. These guidelines, therefore, recommend the use of Terminal format and all of the following suggestions and examples assume the use of that format. Besides the obvious advantage of an uncluttered printout, the Terminal format has other programming advantages:

1. it requires less storage area;

2. it requires no line numbers;

3. its statements may be aligned with tab characters.

Further, whenever required, the REFORMAT utility program will convert Terminal format programs to the Conventional format. (The REFORMAT utility program is discussed in Chapter 11).

The following suggestions should help to further simplify even the most complicated source programs.

1. Begin division, section, and paragraph names in column 1. Although these names may start anywhere in Area A, aligning them in column 1 produces a much more readable listing. When required, place the * and - in column 1. (Column 1 then becomes column 0.)

2. Insert a blank line, or one or more comment lines (describing the purpose of the file) before each SELECT statement in the FILE-CONTROL paragraph. Place the phrases of the SELECT statement on separate lines and begin each of them in column 5 (use the tab character to skip over Area A). Consider the following illustration of a typical SELECT statement:

```
        AREA A          |  AREA B

          1 . . .       |  5 . . . . . . .
                        |  SELECT MASTER-FILE
                        |  ASSIGN TO "DK1:"
                        |  ORGANIZATION IS RELATIVE
                        |  ACCESS IS SEQUENTIAL.
```

3. Place the phrases of the file description statement on separate lines and begin each of them in column 5. (Use the tab to skip over Area A.) Consider the following illustration of a typical file description entry:

```
        AREA A          |  AREA B

          1 . . .       |  5 . . . . . . .
          FD            |  MASTER-FILE
                        |  LABEL RECORDS ARE STANDARD
                        |  VALUE OF ID IS MASTER-FILE-NAME
                        |  DATA RECORD IS MASTER-RECORD.
```

4. In both the File and Working-Storage sections, begin all 01 level items in column 1.

   Indent, by four columns, all subordinate items with higher-valued level numbers. (For example, if the item that is subordinate to a 01-level record description is 05, begin the record description level number in column 1 and the 05 level number in column 5.) Use the tab character for the first indentation, a tab and four spaces for the second, two tabs for the third, etc. When indented in this manner, the listing will show, clearly and neatly, the hierarchical relationships of all of the data names in the program as well as their level number values.

   Increment level numbers by 5; then later, if it becomes necessary to insert additional group items, they may be inserted without having to change the level numbers of all items that are subordinate to that group.

   If desired, write the level numbers as single digits (such as 1 instead of 01).

   Use level number 01 instead of 77 in the Working-Storage Section. (77, as a level number has the same meaning as 01, and 77 may eventually be omitted from the COBOL standard.)

   Since all elementary items, except for index data items, require PICTURE clauses, these clauses fill a good part of the source program listing. However, the PICTURE clause itself may be simplified to enhance the listing's readability as follows:

   a. use PIC as an abbreviation for PICTURE,

   b. omit the noiseword IS, and

   c. align the PIC clauses on successive lines. (Use the tab character to align the clauses.)

5. Put all paragraph name declarations in the Procedure Division on lines separate from the statements in the paragraph. This not only makes the program more readable, it also makes modification of the first statement in the paragraph easier.

6. Follow all imperative statements with a period, making them
    1-statement sentences. Place only one statement on a line.
    In addition to making the lines shorter and more readable,
    this will prove quite helpful when debugging the program.
    For example, if the program contains a coding error, it will
    be on one line and therefore easier to modify without
    affecting the other portions of the sentence;   further, the
    diagnostic messages will refer to the correct line and their
    meanings will be clearer.

    Since left-aligned statements in any program enhance the
    readability of that program, develop the habit of starting
    all COBOL sentences in column 5. (Use the tab character to
    skip over Area A.) Some statements, however, should be
    further indented, as explained in the following paragraphs.

7. If the true path of a conditional statement contains another
    conditional statement or more than one imperative statement,
    place all statements in the true path on lines immediately
    following the conditional statement and indent them to show
    their dependence upon that statement. Consider the following
    illustration of an IF statement and its true path:

        IF COMPUTED-TAX > TAX-LIMIT
            SUBTRACT TAX-LIMIT FROM COMPUTED-TAX GIVING EXCESS-TAX
            MOVE TAX-LIMIT TO COMPUTED-TAX
            ADD EXCESS-TAX TO TOTAL-EXCESS-TAX.

    If the statement has an ELSE (or false) path, align the word
    ELSE under the preceding IF and indent all statements that
    are dependent on the ELSE statement. Thus:

        IF condition
            true path statement
            true path statement
            . . .
        ELSE
            false path statement
            false path statement.

    Be sure to place the period after the last statement only!

    Another good method for simplifying conditionals is to write
    only a single imperative statement in the true or false path.
    If the path requires more statements, place them in a
    separate paragraph and either PERFORM the paragraph from the
    path or GO to it. This technique avoids the possibility of
    inadvertently placing a period at the end of a statement
    within the path, thereby terminating it prematurely.

    When writing a GO TO ... DEPENDING statement, place each
    procedure name on a separate line and indent them all.
    Consider the readability of the following sample statement:

            GO TO P35
                  P40
                  P45
                  P60
                  P65
                  DEPENDING ON P-SWITCH.

8. When grouping statements into paragraphs and sections, use the following organizational ideas:

Group together logical units of processing into a section. Select a section name that reflects the type of processing being conducted within that section (such as TAX-COMPUTATION SECTION, PRINT-LINE-FORMATTER SECTION, etc.). Follow the section name with sufficient comment lines to explain the processing that is carried out by the statements within that section.

Make paragraph names as short and simple as possible. A numbered abbreviation of the section name often suffices. Thus the paragraph names in the TAX-COMPUTATION section might be TC10, TC20, TC30, etc. Use paragraph names sparingly, placing them only where the true and false paths of conditional statements require branch points for GO TO statements. If the temptation arises to give a paragraph a longer name in an attempt to reflect the type of processing in that paragraph, use comment lines instead. (Comment lines usually convey more information, more clearly.)

When using simple numbered paragraph names, assign increasing numeric characters to sequential paragraphs. If the numeric portion of the names increases by 5 or 10, new ones may be inserted later without disturbing the sequence of the names.

Do not use the PERFORM verb in the form, PERFORM a THRU b. If the paragraphs a thru b must be performed, place them in a section by themselves and PERFORM the section, thus avoiding the use of the THRU option.

Place single paragraphs that are to be performed into sections and use the section name as the object of PERFORM verbs. Then, if future design changes introduce complicated conditional logic into the paragraph, requiring additional paragraph names, the PERFORM statements need not be altered.

The preceding guidelines divide the Procedure Division into modular blocks of coding. If these guidelines are used, the following additional techniques may be applied.

a. Restrict entry to all sections through the first statement of the section by use of a GO TO, a PERFORM, or a "fall through" from the preceding section;

b. Ensure that all GO TO statements refer to only section names or paragraph names that are internal to the section containing the GO TO statement.

7.2 USE OF PUNCTUATION

Avoid using the COBOL punctuation characters, comma and semicolon. They lend little to the readability of programs that have their statements neatly aligned, as discussed earlier in this chapter. Further, it is quite easy to misuse these characters, which can cause serious errors for many compilers. (Other compilers either ignore incorrect punctuation characters or flag them with warning messages.) At best, even when used correctly and in the proper places, they have no effect on the meaning of the program.

7.3 USE OF THE ALTER STATEMENT

Avoid using the ALTER statement to change the flow of control in a program. It is impossible to test the setting of an alterable GO statement except by executing it. Also, unless explicit comments accompany an alterable GO statement, it is difficult to tell whether or not it is referenced by ALTER statements or what the possible destinations might be. All of this makes debugging programs that contain these statements quite difficult. There are two other techniques that may be used in their place:

1. If control branches one of two ways (i.e., a binary switch), write the switch as a conditional variable. Consider the following sample coding:

```
01  P-SWITCH PIC S9 COMP VALUE 0.
88 NO-PRINT VALUE 1.

MOVE 1 TO P-SWITCH
        .
        .
        .
IF NO-PRINT GO TO P40.

P40.
     MOVE 0  TO P-SWITCH.
```

2. If control branches more than two ways, use MOVE statements to place integers into a data item, and a GO TO ... DEPENDING ... statement to test the data item and branch accordingly. Consider the following sample coding:

```
01  P-SWITCH PIC S9999 COMP VALUE 0.
        .
        .
        .
MOVE 1 TO P-SWITCH
        .
        .
        .
MOVE 3 TO P-SWITCH.
        .
        .
        .
GO TO
        PART-TIME
        PIECE-WORK
        HOURLY
        SALARIED-WEEKLY
        SALARIED-OTHER
        DEPENDING ON P-SWITCH.
*    FALL THROUGH IS A BUG
     DISPLAY "?17".
     STOP RUN.
```

7.4 USE OF THE PERFORM STATEMENT

The general rules for the PERFORM statement are augmented with the following rules:

GOOD PROGRAMMING PRACTICES

1. The endpoint of a section and the endpoint of the last paragraph in the same section are two distinct points. This means that it is possible to execute a PERFORM of the section, then while that PERFORM is still active, to execute a PERFORM of the last paragraph.

2. On the start of a PERFORM, if the end point of the new PERFORM is the end point of an already active PERFORM, the OTS aborts the task and issues an error message.

3. At the end of any procedure, a check is made to see if the procedure being ended is the end of the most recent PERFORM range. If so, the most recent PERFORM range is exited. If not, the end point of the most recent procedure is checked against the end point of all currently active PERFORMs. If the end point of the procedure is the end point of any currently active PERFORM range, the OTS issues an error message and aborts the task because the perform ranges are not being exited in the reverse of the order in which they were entered.

NOTE

The OTS error messages are discussed in
Section 12.4, Run-time Error Messages.

7.5 USE OF LEVEL 88 CONDITION-NAMES

Condition-names provide a convenient method for testing a value or range of values in a field. The use of condition-names makes programs easier to maintain, because it ensures a uniform method of testing fields and helps to reduce recoding when the specifications of the program change.

The following example illustrates the use of condition-names and shows the advantages inherent in their use.

Suppose the records of a file each describe a student in an educational institution (or an employee in a corporation). Some of the records contain categories of information which are not present in other records. A "code" field, which contains a digit or letter, indicates the presence (or type) of some categories; while a special value in the information itself (such as a numeric value being zero, negative, or maximum) indicates the presence of other categories. The processing of such a record may vary considerably depending on these indicator fields. The fields may require interrogation at various points in the program, and the interrogation may require more than a simple relation test.

Consider a "code" field that holds one of seven values, coded as a mnemonic character. For example, S,1,2,3,4,G,P might be seven values that indicate student categories of Special, 1st year, 2nd year, 3rd year, 4th year, Graduate, and Postgraduate. The field is described as follows:

        05 STUDENT-CATEGORY PIC X.

GOOD PROGRAMMING PRACTICES

Program logic requires certain processing for enrolled undergraduates, different processing for special students, and still different processing for all students except enrolled undergraduates. Without the aid of condition-names, statements might be written as follows to resolve this problem:

```
IF STUDENT-CATEGORY = "S" ...

IF STUDENT-CATEGORY NOT LESS THAN "1"
        IF STUDENT-CATEGORY NOT GREATER THAN "4" ...

IF STUDENT-CATEGORY EQUAL TO "G" NEXT SENTENCE
        ELSE IF STUDENT-CATEGORY EQUAL TO "P"
                NEXT SENTENCE ELSE GO TO ...
```

However, if various level 88 entries follow the STUDENT-CATEGORY description, as shown below, condition-names can simplify this coding.

```
05 STUDENT-CATEGORY PIC X.
    88 UNDERGRADUATE VALUE "1" THRU "4".
    88 SPECIAL-STUDENT VALUE "S".
    88 GRAD-STUDENT VALUE "G" "P".
    88 SENIOR VALUE "4".
    88 NON-DEGREE-STUDENT VALUE "S" "P".
```

Now, the following procedural statements can solve the problem:

```
IF SPECIAL-STUDENT ...
IF UNDERGRADUATE ...
IF GRAD-STUDENT ...
```

Procedural statements with condition-names are much easier to read and debug than those containing the complete test. For example, the procedural statements, IF UNDERGRADUATE ..., and IF STUDENT-CATEGORY NOT LESS THAN "1" IF STUDENT-CATEGORY NOT GREATER THAN "4" both accomplish the same thing, but the first statement is simpler and less confusing.

In addition, the statement, IF NOT UNDERGRADUATE ... can test the category of not being an undergraduate, which is equivalent to any one of the following statements:

```
IF NOT (STUDENT-CATEGORY NOT < "1" AND
        STUDENT-CATEGORY NOT > "4") ...
```

                                or

```
IF STUDENT-CATEGORY < "1" OR
    STUDENT-CATEGORY > "4" ...
```

                or

```
IF STUDENT-CATEGORY < "1" NEXT SENTENCE
    ELSE IF STUDENT-CATEGORY > "4" NEXT SENTENCE
        ELSE GO TO ...
```

Statements such as these are tedious to write and a frequent source of coding errors. Further, if a change creates a new student category, the recoding takes more time and is even more error prone. A careful and controlled use of condition-names forces a higher degree of programming control and checkout. If the program logic does require the modification of the STUDENT-CATEGORY field, it can even be named FILLER thus removing the opportunity to shortcut the use of condition-names.

To apply condition-names, follow the description of the item to be tested with a level 88 entry. The item being tested, known as the conditional variable (STUDENT-CATEGORY in the preceding illustrations), may be either DISPLAY or COMPUTATIONAL usage, but not INDEX usage; it may also be a group item.

The compiler stores all of the values supplied by the level 88 entries in the object program exactly as written. (They are pooled with all of the literals from the Procedure Division.) A value supplied by a level 88 entry for a conditional variable of COMPUTATIONAL usage is stored in binary format to save conversion at object time. The compiler stores all other values as byte strings with the proper attributes. It does not make the level 88 entries equal to their conditional-variables in size. This means that it neither truncates nor pads (with spaces) non-numeric literals. Further, it neither truncates nor pads (with zeros) numeric literals, but stores them as written or, if converted to binary, in the minimum size COMP item that will hold the converted value. It stores signs as trailing overpunches on numeric DISPLAY literals, and removes and remembers decimal points.

Do not enter level 88 items under group items that have subordinate entries containing any of the following clauses: SYNCHRONIZED, JUSTIFIED, COMPUTATIONAL, INDEX.

7.6 USE OF QUALIFIED REFERENCES

7.6.1 Qualified Data References

The COBOL language provides facilities to define and reference user-defined data items. Data items are programmer-defined variables declared in the Data Division of a COBOL program. Such variables include, among others, file record descriptions and internal working areas. These data items are processed by procedural statements such as the WRITE, MOVE, and ADD statements. Procedural operations on these data are facilitated through references to the data items by name. For example, to update a variable, YTD-GROSS-PAY, by a weekly gross pay amount WEEKLY-GROSS, write the program fragment shown in Figure 7-1.

```
        .
        .
        .
    WORKING-STORAGE SECTION.
    01 YTD-GROSS-PAY  PIC 9(5)V99.
    01 WEEKLY-GROSS  PIC 999V99.
        .
        .
        .
        ADD WEEKLY-GROSS TO YTD-GROSS-PAY.
```

Figure 7-1
Unqualified Data Item Reference

In this example, YTD-GROSS-PAY and WEEKLY-GROSS are defined in the Working Storage Section of the Data Division as COBOL variables with a level number of 01. The variable representing the "year-to-date gross

pay (YTD-GROSS-PAY)" is computed by incrementing its present value by
the "weekly gross pay (WEEKLY-GROSS)" amount through reference to the
appropriate data items in the ADD statement. References are made to
the data items by the singular, unqualified names of YTD-GROSS-PAY and
WEEKLY-GROSS. Since YTD-GROSS-PAY and WEEKLY-GROSS are defined with
level numbers of 01 in the Working Storage Section, these variables
must be unique in their spelling and, hence, can only be referenced by
the spelling of each data item's name without any COBOL qualification.

The example in Figure 7-1 is artificial because the data item
representing the "year-to-date gross pay" is defined as a level 1
variable in the Working Storage Section. More realistically,
YTD-GROSS-PAY is defined as a field within an employee payroll record
residing on an external master payroll file. The process of updating
the "year-to-date gross pay" by a "weekly gross pay" amount is shown
more appropriately in Figure 7-2.

```
        .
        .
        .
   FILE SECTION.
   FD MASTER-IN
       LABEL RECORD IS STANDARD
       VALUE OF ID IS "MASTER.PAY".
   01 PAY-RECORD.
       03 NAME            PIC X(30).
       03 EMPLOYEE-NO     PIC 9(9).
       03 YTD-GROSS-PAY   PIC 9(5)V99.
        .
        .
        .
   FD MASTER-OUT
       LABEL RECORD IS STANDARD
       VALUE OF ID IS "MASTER.PAY".
   01 PAY-RECORD.
       03 NAME            PIC X(30).
       03 EMPLOYEE-NO     PIC 9(9).
       03 YTD-GROSS-PAY   PIC 9(5)V99.
        .
        .
        .
   WORKING-STORAGE SECTION.
   01 WEEKLY-GROSS        PIC 999V99.
        .
        .
        .
   PROCEDURE DIVISION.
   INIT.
       OPEN INPUT MASTER-IN.
       OPEN OUTPUT MASTER-OUT.
        .
        .
        .

       ADD WEEKLY-GROSS, YTD-GROSS-PAY OF MASTER-IN
               GIVING YTD-GROSS-PAY OF MASTER-OUT.
        .
        .
        .
```

Figure 7-2
Qualified Data Item Reference

In this example, YTD-GROSS-PAY is defined as a field in both the input and output record descriptions. There are two separate data items whose spellings are identical.

To reference each data item, it is necessary to qualify the name of each data item with sufficient information to constitute a unique reference. Thus, to reference the "year-to-date gross pay" amount in the output record, we write "YTD-GROSS-PAY OF MASTER-OUT" where such a reference is called a qualified reference. The filename MASTER-OUT is functioning as a qualifier in the reference. The reserved word "OF" is the qualification connector and may be used interchangeably with the reserved word "IN" in this context. Another way of referencing the same data item is to write "YTD-GROSS-PAY OF PAY-RECORD IN MASTER-OUT". This reference is called a completely qualified reference because all possible qualifiers are specified in the reference. A reference of the form "YTD-GROSS-PAY" or "YTD-GROSS-PAY OF PAY-RECORD" is illegal since it does not uniquely identify which of the two data items is desired. Such a reference is termed an ambiguous reference.

In the area of data item definition and referencing, COBOL is unlike other languages such as FORTRAN and ALGOL 60. While FORTRAN requires each data item to have a unique name (i.e., no two data items may have a name of identical spelling), COBOL relaxes this requirement to the extent that each data item must be uniquely referable. That is, two or more data items may have their names spelled identically, but there must exist a way to reference each distinct data item. Thus, there is a distinction between a data item and its name. Central to understanding this distinction is understanding the concept of unique referability.

The functionalities of data item definition and referencing may be understood by stating three guidelines which relate the concepts of data item definition, reference format, and unique referability.

7.6.2 Guideline 1 (Data Item Definition)

Each data item has a name. Each name is immediately preceded by an associated positive integer called its level number. A name either refers to an elementary item or else it is the name of a group of one or more items whose names follow. In the latter case, each item in the group must have the same level number, which must be greater than the level number of the group item.

7.6.3 Guideline 2 (Reference Format)

Data-name qualification is performed by following a data-name or condition-name by one or more phrases of a qualifier preceded by IN or OF. IN and OF are logically equivalent. The general format of a qualified reference to an elementary item or group of items named "name-0" is given in Figure 7-3.

```
name-0 OF name-1...OF name-m
```

Figure 7-3
General Format of a Qualified Data Reference

where m >= 0 and where, for 0 <= j < m, name-j is the name of some item contained directly or indirectly within a group item named "name-j+1". A reference of the form given in Figure 7-3 is called a (partially) qualified reference with name-1,name-2,...,name-m being called qualifiers. Such a reference is termed a completely qualified reference if "name-j+1" is the father of name-j for 0 <= j <= m-1.

In the hierarchy of qualification, names associated with an FD indicator are the most significant, then the names associated with level-number 01, then names associated with level-number 02,...,49. The most significant name in the hierarchy must be unique and cannot be qualified. Subscripted or indexed data-names, unsubscripted data-names, and condition variables may be made unique by qualification. The name of a condition variable can be used as a qualifier for any of its condition-names. Enough qualification must be mentioned to make the reference unique; however, it may not be necessary to mention all levels of the hierarchy as the example in Figure 7-2 demonstrates.

7.6.4 Guideline 3 (Unique Referability)

If more than one data item is defined with the same name "name-0", there must be a way to refer to each use of the name by using qualification. That is, each definition of "name-0" must be uniquely referable. A data item is uniquely referable if the complete set of qualifiers for the data item are not identical to any partial (including complete) set of qualifiers for another data item.

7.6.5 Qualified Procedure References

The facility of qualification may be applied to procedure references. A procedure name is either a paragraph or section name. By definition, a paragraph name is unique only within a section containing the paragraph while, on the other hand, section names must be unique within a COBOL program. The general format of a qualified procedure reference is shown in Figure 7-4.

```
paragraph-name OF section-name
```

Figure 7-4
General Format of a Qualified Procedure Reference

A paragraph name may be qualified by its containing section name; a section name may never be qualified in a procedure reference. When a paragraph name is referenced without an explicit section name qualifier, the paragraph name is implicitly qualified by the appropriate section name.

If a paragraph name is unique within a COBOL program it is not necessary to qualify the paragraph name in the procedure reference. Finally, if a paragraph name is not unique within a COBOL program, the paragraph name must be qualified in a procedure reference when the reference is made outside of the section which contains the paragraph.

7.6.6 Qualification and Compiler Performance

Qualification is a powerful language facility for the development of COBOL programs. Used wisely, it increases the readability of COBOL programs. However, the user pays a price for utilization of this facility in terms of a slower compilation rate (i.e., COBOL source lines per unit of time).

Qualification requires a tree-structured symbol table at compile-time. The time required for building and looking up on a tree-structured symbol table is considerably longer than for a non-tree-structured symbol table. This translates into a general degradation of compiler performance. If qualification is not employed in a program compiled by the PDP-11 COBOL compiler, compilation speed is not affected. However, when qualification is used, the compilation rate slows down due to the additional system overhead.

In general, if there are deeper levels of qualification, there will be a slower compilation. This is especially so at the end of the Data Division text where duplicate data-name declarations are detected by the compiler. Object-time performance is not affected by usage of the qualification facility.

CHAPTER 8

PDP-11 COBOL UTILITY PROGRAMS

8.1 COBRG (COBOL REPORT GENERATOR)

8.1.1 Introduction

The COBRG program provides a simple mechanism for producing printed
reports from data files. And, although it may be tempting to design
and write personally tailored report generating algorithms, it is
usually easier and faster to use this utility program.

Most personally tailored programs that produce printed reports adhere
to the following pattern:

1. The program reads an input file that has been sorted on any
   number of keys;

2. It prints the contents of certain fields from each input
   record on a printed report (usually the fields are in
   designated columns with a heading on each page to label the
   columns);

3. It keeps running totals in accumulators to be printed on the
   report (and zeroed for new summations) whenever designated
   fields change values, thus developing and printing subtotals
   at the end of each section or subsection of the report;

4. Finally, at the end of the input file, it prints cumulative
   totals for the entire report.

An analysis of typical report generating programs would show that they
contain several of the following elements:

● Description of the page headings;

● Description of the input and output files;

● Set of rules for moving information from the input records to
  the detail print lines;

● Set of rules for adding values into accumulators;

● Set of rules for monitoring the sort keys for value changes;

● Set of rules for constructing and printing the accumulator
  values.

The COBRG utility program features input specification lines which
provide all of the preceding elements. COBRG uses these specification

lines to produce a tailored COBOL source program. This program, when compiled and executed, generates the actual report. The following list contains a brief description of the input specification lines along with the information each provides:

- NAME Specification -- provides the program name and the description of the input and output files;

- INPUT Specification -- provides a description of the records in the input file;

- OUTPUT Specification -- provides a description of the records in the output file (detail lines and total messages);

- HEADER Specification -- provides a description of the page headers;

- BREAK Specification -- provides monitoring information for all control break fields;

- ACCUMULATOR Specification -- provides information on all fields to be added and sets up accumulators to contain the sums;

- TOTAL Specification -- provides information on accumulators that are to be moved directly to control footing print lines;

- EMIT Specification -- provides textual information for explaining the totals on control footing lines;

- LIST Specification -- provides detail line listing (output) information.

Subsequent sections of this chapter describe, in detail, these specification lines. For illustration purposes, sample entries for preparing a sample report accompany the rules for forming the lines. A description of the sample problem follows.

8.1.2 COBRG Sample Problem

The input file contains sales information. Each record within this file contains the customer's name and location, along with the sales to that customer for a given period. A previous sort run has sorted the file on three location keys; a CITY key, a STATE key, and a REGIONAL key.

Management requires a report which lists all of their customers, shows the sales to those customers, and provides subtotals for each city, for each state, and for each region.

The input file contains 80-character records organized as follows:

```
01  INPUT-RECORD.
    05 CUSTOMER-NAME       PIC X(20).
    05 CUSTOMER-ADDRESS    PIC X(20).
    05 CITY                PIC X (10).
    05 STATE               PIC XX.
    05 ZIP                 PIC X(5).
    05 REGION              PIC X.
    05 SALES               PIC S9(10)V99.
    05 OTHER-DATA          PIC X(10).
```

The preceding sort run sorted the file on REGION as the major sort key, STATE as the secondary sort key, and CITY as the minor sort key. Thus, for example, if Region 1 is New England, Connecticut will probably be first (CT), and the cities of Bridgeport, Hartford, and Waterbury, if present, will appear in that order. (Section 9.1.6.4 shows the first 50 records of the file.)

8.1.3 COBRG Specification Lines

Before COBRG can produce a report-generating COBOL source program, it requires a file containing one or more sets of the specification lines discussed earlier. These specification lines describe the report or reports to be produced by the compiled source program. The user must ensure that they are filled out completely and accurately, as indicated in this and following sections.

Use a comma to separate each parameter in a specification line from the next and a carriage return to terminate each complete specification line.

Although the COBRG program requires most of the parameters in a specification line, some of them may be omitted. (This chapter denotes those that may be omitted and explains the circumstances for omission.) To omit a parameter from the middle of the line, enter a comma in its place. This comma, following the separator comma of the preceding parameter, notifies the software that a parameter is missing. If the omission occurs at the end of the line, simply enter the usual carriage return (the software requires no additional missing-parameter indicator at the end of a line).

The first parameter of every line identifies the type of specification line and should be placed in column 1. The comma following this parameter is optional.

Since COBRG generates COBOL source language programs containing COBRG-generated data and procedure names, avoid using names that conflict with the generated names. Consider them as an extension of the COBOL reserved word list.

The following generated names, in addition to the standard PDP-11 COBOL reserved words, cannot be used as data-names in the COBRG specification of the input and output files.

ACCUMULATOR-x-n	LEVEL-x-BREAK	PRINT-LINE
BEGIN	LEVEL-n-SW	PRINT-CH-n
FINISH	L-PRINT	PRINT-DETAIL
HEADER	OUTPUT-LINE	PRINT-n
HEADER-PAGE	PAGE-COUNT	READ-IN
IN-FILE	PL-EXIT	RESET-LEVEL-x
LEVEL-x-ACS	PL-HDR	SAVE-nn

NOTE

n may be any number from 0-9; x may be any number from 0-9 or F.

8.1.3.1 **NAME Specification** - This specification line must be the first specification in the input sequence. It provides the name of the COBOL source program and describes the input file and the output file (the report). In fact, the NAME specification supplies

parameters for setting up the IDENTIFICATION, ENVIRONMENT, and DATA DIVISIONS of the COBOL source program. It must contain the following entries:

Parameter 1 -- <u>Specification identification.</u> Enter the letter N to identify this line as the NAME specification.

Parameter 2 -- <u>Program name.</u> Enter a unique name, up to nine characters long, to identify the COBOL source program.

Parameter 3 -- <u>Input blocking factor (optional).</u> Enter the blocking factor for the input data file. If this parameter is omitted or zero, the software assumes that the file is not blocked.

Parameter 4 -- <u>Input file specification.</u> Enter the complete file specification for the input file, with delimiting quotes.

Parameter 5 -- <u>Output file specification.</u> Enter the complete file specification for the output file, with delimiting quotes.

Parameter 6 -- <u>Author (optional).</u> Enter your name (up to 20 characters long). This parameter may be omitted.

The following entries would satisfactorily complete the NAME specification line for the sample program:

```
         N,COBRGTEST,  ,"SALES","LP:",F.X.DOE
Parameter)   /   /    /    /     /    /
         )   1   2    3    4     5    6
Numbers  )
```

8.1.3.2 **INPUT Specifications** - The NAME specification line has already described the input file; now all of the records in the file must be described. The INPUT specification lines provide the software with the names and descriptions of the records and fields of the input file.

These specifications are simply individual lines of a complete COBOL record description; however, each line begins with an I in character position one.

Parameter 1 -- <u>Specification identification.</u> Enter the letter I to identify this line as an INPUT specification.

Parameter 2 -- <u>Data description.</u> Enter one line of a COBOL data description, including the level-number, data-name, and PICTURE clause. If the record description already exists in a library file, enter the word COPY, followed by the name of the library file.

Since the software requires several INPUT specification lines to complete one record description, the record description (01-level) must be presented to the software first, with the field descriptions following in the same order as they appear in the record; however, the software requires only the names of those fields that are to be used in the report. All others may be described as FILLER items.

The following INPUT specifications would satisfactorily complete the record description for the input file of the sample program:

```
I01     INPUT-RECORD.
I       05   COMPANY     PIC  X(20).
I       05   FILLER      PIC  X(20).
I       05   CITY        PIC  X(10).
I       05   STATE       PIC  XX.
I       05   FILLER      PIC  X(5).
I       05   REGION      PIC  X.
I       05   SALES       PIC  S9(10)V99.
I       05   FILLER      PIC  X(10).
```
```
        ↗                      ↗
        1                      2
```

If the input file for the sample program already existed in a library file (named SALESR.LIB) which contained the actual complete record description, the following INPUT specification would satisfactorily complete the record description:

```
ICOPY SALESR.
  ↗     ↗
  1     2
```

COBRG removes the I and places each line in the File Section of the COBOL source program under the FD entry for the input file. Hence all rules for the formation of data descriptions in the File Section apply. (In particular, VALUE clauses are not allowed.)

COBRG does not check the contents of the INPUT specifications for correct COBOL syntax, therefore, any coding errors will not be discovered until the COBOL compiler compiles the source program.

If horizontal tab characters are included in I specifications, a warning diagnostic (probably harmless) will be issued upon subsequent compilation of the generated program.

8.1.3.3 **OUTPUT Specification** - The NAME specification line has already described the output file; now the detail lines of the report must be described. The OUTPUT specification lines provide the software with descriptions of the records and fields of the output file (the detail lines), and the control footing messages.

Like the INPUT specifications, the OUTPUT specifications are simply individual lines of a complete COBOL record description, describing a detail line of the output file; except that each line begins with an O in character position one and the record description appears in the WORKING-STORAGE Section instead of in the FILE Section. Further, the COBRG program will not accept a 01-level record description entry. (COBRG supplies a standard 01-level and name -- 01 OUTPUT-LINE.)

Parameter 1 -- Specification identification. Enter the letter O to identify this line as an OUTPUT specification.

Parameter 2 -- Data description. Enter one line of a COBOL data description, including the level-number, data-name, and PICTURE clause. Do not enter a 01-level record-name for the first OUTPUT specification (COBRG provides a standard one, as discussed above).

If the record area is to be redefined, enter any level-number higher than 01 (ensuring, of course, that

it is lower than the level-number of the fields in the record). Then, use the same level-number for the redefined record. Consider the entries below for the sample program: Level 03 defines the records and 05 defines the fields. COBRG will subordinate both record areas to 01 OUTPUT-LINE.

If the record description already exists in a library file, enter the word COPY, followed by the name of the library file.

Since the software requires several OUTPUT specification lines to complete one detail line, any record-name entry must be presented to COBRG first, with the field descriptions following in the same order as they are to appear on the report. Use FILLER items to separate the various fields. (All FILLER items will be space filled on the report.) The following OUTPUT specification entries would satisfactorily complete the description of one detail line of the report for the sample program, as well as a message line for the control footing totals. The message line, which must be described on an EMIT specification, redefines the detail record area. (EMIT specifications are discussed later in this chapter.)

```
OO3   DETAIL-LINE.
O     05   PRINT-COMPANY         PIC X(20).
O     05   FILLER                PIC XX.
O     05   PRINT-CITY            PIC X(10).
O     05   FILLER                PIC XX.
O     05   PRINT-STATE           PIC XX.
O     05   FILLER                PIC XX.
O     05   PRINT-SALES           PIC Z,ZZZ,ZZZ,ZZZ.ZZ-.
OO3   CFLINE REDEFINES DETAIL-LINE.
O     05   MESS1                 PIC X(20).
O     05   ACCOUNT               PIC ZZZZZ.
O     05   MESS2                 PIC X(13).
O     05   FILLER                PIC X(22).
```
    1   2

If a library file (named, for example, PRTLIN.LIB) already existed which contained these entries, the following OUTPUT specification would satisfactorily complete the record description:

```
OCOPY PRTLIN.
```
  1   2

COBRG strips off the O and places each line in the WORKING-STORAGE Section of the COBOL source program under the 01-level entry for the output record. Hence, all rules for the formation of data descriptions in the WORKING-STORAGE Section apply.

COBRG does not check the contents of the OUTPUT specifications for correct COBOL syntax, therefore, any coding errors will not be discovered until the COBOL compiler compiles the source program.

If OUTPUT Specifications contain horizontal tab characters, a warning diagnostic (probably harmless) will be issued upon subsequent compilation of the generated program.

8.1.3.4 **HEADER Specifications** – The HEADER specification lines provide COBRG with descriptions of the page headings.

COBRG assumes that the report is to be printed on standard printer forms, and allows 66 lines per page. It prints on only 60 of the lines and leaves three blank lines at the top and three blank lines at the bottom of each page. The first printing line (line 4) may be used as a header line and a number of blank lines may be designated to follow the header line. The remainder of the printing area is called the body of the page and is devoted to printing detail lines and lines that contain totals.

The first HEADER specification supplies header data for columns 1 through 60, and the second HEADER specification (if required) supplies header data for columns 61 through 132.

Parameter 1 -- Specification identification. Enter the letter H to identify this line as the HEADER specification.

Parameter 2 -- Page number field size. Enter the size of the field that will contain the page number in the heading line.

Parameter 3 -- Page number location. Enter the number of the starting column for the page number in the heading line.

Parameter 4 -- Blank lines. Enter the number of lines to be left blank after printing the heading line; an entry of 0 indicates no blank lines.

Parameter 5 -- Skip channel (not yet implemented). Enter the number (0-8) of the printer channel to which a skip is to be made after printing the heading line; an entry of 0 indicates no skipping. At present, COBRG treats a non-zero entry as a 1 (top-of-page skip).

Parameter 6 -- Header data. Enter the data to be placed in columns 1 through 60 at the top of each page of the report. Do not include commas within the header. COBRG will not accept imbedded commas since it considers them to be parameter separators.

Parameter 7 -- Page number reset (optional). Enter the break priority number at which the page number is to be reset to 1 (break levels are discussed under BREAK Specification in the next section). The page number is always 1 on the first page. If the page number is not to be reset, this parameter can be omitted.

If the information to be printed on the header line exceeds 60 columns, a second HEADER specification may be completed to supply up to 72 columns of additional header information. Since the information supplied by parameters 2, 3, 4, 5, and 7 has already been presented by the first HEADER specification, the second one requires only the identification and header data parameters:

Parameter 1 -- Specification identification. Enter the letter H to identify this line as a HEADER specification.

Parameter 2 -- Header data. Enter the data to be placed in columns 61 through 132 of the header line. Do not include commas within the header.

Consider the report to be produced by the sample program. Each detail line requires 60 columns and contains the customer-name, a line

number, and the sales to that customer. The first detail line after
any subtotal line also contains the city and state for the detail
lines that follow.

Section 8.1.6.5 provides an example of the desired output format. The
HEADER specification, which is the second line of input to COBRG in
the sample program, contains a description of the header line and some
of the spacing information.

The following HEADER specification entries would satisfactorily
complete the description of the report header for the sample program.

```
       H,3,58,3,0,COMPANY>——<CITY>——<STATE>———<SALES>—<PG.
        / / / / /        /                                              ↘1 space
       / / / / /        /        15 spaces   ↘
      / / / / /        /                       6 spaces  ↘
      1 2 3  4 5      6                                10 spaces
```
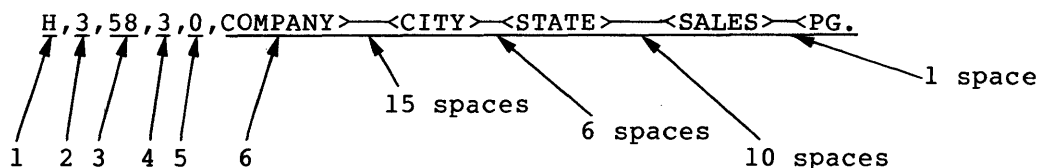
                                NOTE

        The spacing between the words, in
        parameter 6, aligns the header with the
        detail lines, thus forming columns.

8.1.3.5 BREAK Specification - This specification line provides the
information that COBRG requires to monitor the control break fields.
Its parameters identify the fields and the levels of those fields
within the control break hierarchy; further, its parameters provide
the information for any required printer manipulation for control
footing.

COBRG considers the entire report as a hierarchy of levels. The
lowest level is the control field that is expected to change most
often. (The lowest level control field usually corresponds to the
minor sort key.) In the sample program, the lowest level control field
is the CITY field.

As input records are read and detail lines are printed, the COBOL
program supplied by COBRG monitors the control field associated with
each level in the report's hierarchy of control fields. When it
detects a change in value in a control field (a control break has
occurred), it prints a control footing. Control footings contain
totals and literal values with spacing requirements (to make them
stand out from the detail lines) that may differ from those of the
detail lines. Control footings, therefore, usually require separate
descriptions which are supplied by Parameters 3 through 6 of this
specification.

COBRG accepts up to 25 BREAK specifications completed as discussed
below:

Parameter 1 -- Specification identification. Enter the letter B to
                identify this line as the BREAK specification.

Parameter 2 -- Break level. Enter a digit from 0 to 9 or F (for
                final). The number 0 has the lowest priority and the
                number 9 has the highest priority. More than one field
                can have the same priority number, in which case a
                break of the same level occurs whenever any of those
                fields change value.

Parameter 3 -- <u>Blank lines, before printing</u>. Enter a number from 0 to 9 that represents blank lines to space before printing the control footing; an entry of 0 indicates no spacing.

Parameter 4 -- <u>Skip channel, before printing (not yet implemented)</u>. Enter a number from 0 to 8 that represents the printer control channel to which a skip is to be made before printing the control footing; an entry of 0 indicates no skipping. At present, COBRG treats a non-zero entry as a 1 (top-of-page skip).

Parameter 5 -- <u>Blank lines, after printing</u>. Enter a number from 0 to 8 that represents the number of blank lines to space after printing the control footing; an entry of 0 indicates no blank lines.

Parameter 6 -- <u>Skip channel, after printing (not yet implemented)</u>. Enter a number from 0 to 8 that represents the printer control channel to which a skip is to be made after printing the control footing; an entry of 0 indicates no skipping. At present, COBRG treats a non-zero entry as a 1 (top-of-page skip).

Parameter 7 -- <u>Field size</u>. Enter the number of characters or digits in the break field.

Parameter 8 -- <u>Field name</u>. Enter the name of the break field. The field must be named in (and spelled the same as) the input record description. The following example contains the BREAK specification for the sample program:

```
1 2 3 4 5 6 7   8
↘ ↘ ↘ ↘ ↘ ↘ ↘   ↘
B,0,1,0,0,0,10,CITY
B,1,1,,,,,2,STATE
B,2,1,,,,,1,REGION
B,F,1
```

The COBOL program supplied by COBRG looks for a break on every level from high to low in every input record and when it detects a break, it "forces" breaks to occur at all lower levels, regardless of whether or not it detected breaks at those levels. Therefore, in the sample program (which has the data file arranged so that the last input record for FALMOUTH, MA is followed immediately by the record for FALMOUTH, ME), it would detect a break at the state level (level 1) between the last FALMOUTH, MA record and the first FALMOUTH, ME record and force a break to occur at the city level (level 0).

8.1.3.6 ACCUMULATOR Specification - While the object program produced by COBRG is reading input records and printing detail lines, it can also add values from the input record into accumulators and print the totals on control footing lines when control breaks occur. This specification line provides the information that COBRG requires to set up these accumulators. In addition to describing the accumulator (name and size), its parameters also specify the name of the field to be added into the accumulator and, optionally, the name of the field in the detail line print record into which the accumulator is to be moved for detail listing.

COBRG accepts up to 10 ACCUMULATOR specification lines completed as discussed below:

Parameter 1 -- Specification identification. Enter the letter A to identify this line as the ACCUMULATOR specification.

Parameter 2 -- Accumulator number. Enter a unique digit that specifies the accumulator and that can be used to refer to it. Any number from 0-9 may be entered.

Parameter 3 -- Accumulator size. Enter a number that specifies the size of the accumulator in digits.

Parameter 4 -- Decimal positions. Enter a number that specifies the number of decimal places in the accumulator.

Parameter 5 -- Input field. Enter the name of the field in the input record that is to be accumulated; the name may be up to 24 characters long.

Parameter 6 -- Output field (optional). Enter the name of the field in the printer record into which the accumulator is to be moved for listing on detail lines. If the accumulator is not to be listed on detail lines, omit this parameter.

When determining the accumulator size, ensure that enough integer and decimal positions are reserved. COBRG always provides a signed field for accumulators.

The input field that is to be added to the accumulator (Parameter 5), although usually a field in the input record, does not have to be in the input record at all, but may, in fact, be a literal (without commas) or even a field in the output record. If this field is to appear on a detail line, it must be named in a LIST specification.

The following ACCUMULATOR specifications will set up the accumulators required by the sample program (two are required):

        A,0,14,2,SALES

        A,1,5,0,1,LINENO
        ↗ ↗ ↗ ↗ ↗
        1 2 3 4 5 6

In the sample problem entries above, accumulator 0 contains a summation of the SALES field from the input records and accumulator 1 contains a count of the detail lines within the lowest level (the program uses this accumulator to number the detail lines); the literal 1, in Parameter 5, is the quantity added to this accumulator for each detail line in the field named LINENO. (Accumulator 0 is not printed on the detail lines.)

Accumulators are numeric fields in the COBOL source program produced by COBRG. By moving them to fields described on the OUTPUT specifications, they can be printed on the report. Further, by using the OUTPUT specification to describe a numeric edit picture, they may be edited with any legal COBOL numeric editing mask.

Each accumulator number requested on this specification line actually causes the software to set up a family of accumulators, one for every defined break level.

At the beginning of execution of the object program, the software initializes all accumulators to zero values and begins adding the requested fields or literals into the lowest level. Whenever a break occurs on any level, the software adds the contents of the accumulator at the lowest level in the family to the next level accumulator and zeros the lower; this accumulator is then added to the next higher level accumulator and zeroed, and so on up to the level of the break. At the level of each break, the software moves the contents of the accumulator at that level to its associated control footing line, prints the control footing line, adds the contents of that accumulator to the next higher level (or F-level if the break occurred on the highest requested level), and zeros the accumulator. This summing and zeroing of levels upward through the family is called "rolling forward" and is the technique employed by COBRG to acquire cumulative totals.

The TOTAL specification line (discussed in the next sub-section) coordinates the accumulators and break levels, and specifies which accumulators to roll forward and print on which break levels. Of course, all break levels and accumulators coordinated by the TOTAL specification line must have been defined on the BREAK and ACCUMULATOR specification lines respectively.

8.1.3.7 **TOTAL Specification** - This specification line provides the information that COBRG requires to move the accumulated totals to the control footing lines and print them. It supplies COBRG with the number of the accumulator to be printed, the name of the output field into which that accumulator is to be moved, and a list of the control break levels which will cause that accumulator to be rolled forward, moved, and printed.

COBRG accepts up to 50 TOTAL specification lines completed as discussed below:

Parameter 1 -- <u>Specification identification.</u> Enter the letter T to identify this line as the TOTAL specification.

Parameter 2 -- <u>Accumulator number.</u> Enter the number of the accumulator that is to be moved into an output field and printed.

Parameter 3 -- <u>Output field.</u> Enter the name of the output field into which the accumulator is to be moved. The name may contain up to 24 characters.

Parameter 4 -- <u>Break numbers.</u> Enter the break numbers (0-9,F) which will cause the accumulator (identified in parameter 2 above) to be printed; do not separate these numbers with commas. An entry of F signifies the final break at the end of input data.

The sample program requires control footings at the CITY, STATE, and REGION levels; as well as a final (national) level at the end of the report. In the control footing caused by the CITY (lowest) level, the program must print the total sales for each city. In the control footing caused by the STATE (second) level, the program must print the total sales for each state, which is actually the sum of that state's CITY totals. Likewise, it must print REGIONAL (the sum of the states within the region), and final (the sum of all the regions) totals.

The following TOTAL specification entries will complete the description of the totals required by the sample program.

     T,0,PRINT-SALES,012F

     <u>T</u>,<u>1</u>,<u>ACCOUNT</u>,<u>2F</u>
      1 2 3   4

(Accumulator 0, in the sample problem, prints in all control footings and will appear beneath the SALES column.)

Consider what happens when the first break occurs at the STATE level in the sample program. Accumulator 0 contains a summation of the SALES field from all of the detail records that have been accessed up to the record that caused the break. The break at the STATE level forces a break to occur on the CITY level (all lower levels). Thus the program produces a control footing for the CITY level followed by a control footing for the STATE level. Step-by-step, the software takes the following actions when it detects the STATE level break.

1.   It prepares and prints the control footing for control level 0 (CITY).

2.   It rolls forward the level 0 accumulator by adding the contents of that accumulator to the level 1 accumulator and zeroing the level 0 accumulator.

3.   It prepares and prints the control footing for control level 1 (STATE) which is the level of the break.

4.   It rolls forward the level 1 accumulator by adding the contents of that accumulator to the level 2 (REGION) accumulator and zeroing the level 1 accumulator.

5.   It then returns to printing detail lines.

Thus, the program has printed the total of SALES (accumulator 0) for the CITY level, added them to the STATE level, printed them for the STATE level, and added them to the REGION level. The accumulators for the CITY and STATE levels contain zeroes and the REGION level contains the total sales thus far. When the REGION level breaks, the software will add the sum for the last state in the region to the accumulator for the region level, print that sum in a control footing for the region, add the sales for that region to the final level accumulator (F), and zero the region level accumulator.

The control footing lines for all of these totals will probably require additional textual information to explain the totals being printed. The following specification line provides this information.

8.1.3.8 **EMIT Specification** - This specification line provides any textual information required by the control footing totals. It contains the break level and the message to be displayed, as well as the name of the field into which the message is to be moved.

COBRG accepts up to 50 EMIT specification lines completed as discussed below:

Parameter 1 -- <u>Specification identification.</u> Enter the letter E to identify this line as the EMIT specification.

Parameter 2 -- <u>Break number.</u> Enter the break priority number which will cause this message to print with its total line.

Parameter 3 -- <u>Message.</u> Enter the message to be placed on the total line. If the information is an alphanumeric literal, enclose it within quotation marks. If the information is either a numeric literal or a data-name, do not enclose it within quotation marks. COBRG will accept a message length of up to 24 characters including the quotation marks; it will not, however, accept imbedded commas since it considers commas to be parameter separators.

Parameter 4 -- <u>Output field.</u> Enter the name of the output field into which this message is to be moved. (This field must have been defined on an OUTPUT specification.)

The following EMIT specification lines will satisfactorily meet the requirements of the total lines in the sample problem:

```
        E,0,"CITY SALES",MESS2
        E,1,"STATE SALES",MESS2
        E,2,"TOTAL SALES =",MESS2
        E,F,"TOTAL SALES =",MESS2
         ↗ ↗  ↗             ↗
         1 2 3             4
```

8.1.3.9 LIST Specification - This specification determines what will be listed on each detail line. It may be thought of as a MOVE statement, and is, in fact, turned into a MOVE statement by the COBRG program. The LIST specification refers to a sending field and a receiving field, which both appear in the source program produced by COBRG as operands in the MOVE statement.

The sending field may be a literal (numeric or alphanumeric) or a field in the input record, and the receiving field must be a field that has been described in an OUTPUT specification (it may be a group item).

The LIST specification contains a control break level field (parameter 2), and if this field contains a break level number from 0 to 9, the field described in parameter 3 will be moved to the field described in parameter 4 and printed only on the first detail line printed after a break on that level. This feature suppresses the printing of columns that would normally contain the same, repetitive information.

COBRG accepts up to 50 LIST specification lines completed as discussed below:

Parameter 1 -- <u>Specification identification.</u> Enter the letter L to identify this line as the LIST specification.

Parameter 2 -- <u>Control break level.</u> Enter a control break level number or the letter A. If this parameter contains a break level number (0-9), it will cause the item named in parameter 3 (below) to print only after the total lines caused by a break at this level. If this parameter contains an A, the program will print this item on every detail line.

Parameter 3 -- <u>Sending field.</u> Enter the name of the input item to be listed. This item may be a literal.

Parameter 4 -- <u>Receiving field</u>. Enter the name of the output field
<u>into which the</u> item in parameter 3 (above) is to be
moved for listing.

In the sample program, the software is to print the CITY and STATE
fields only after either of them breaks, but it is to detail list the
COMPANY and SALES fields. The following LIST specifications will
accomplish the requirements of the sample problem:

```
        L,A,COMPANY,PRINT-COMPANY
        L,A,SALES,PRINT-SALES
        L,0,CITY,PRINT-CITY
        L,0,STATE, PRINT-STATE
        ↗ ↗ ↗        ↗
        1 2 3        4
```

8.1.4 Output from COBRG

The output from COBRG consists of one or more COBOL source programs
and a listing file. The name of each COBOL source program is the
program name entered in parameter 2 of the NAME specification line;
however, COBRG adds an extension of .CBL to each of these names. The
listing file contains a list of input specifications for all of the
programs that were generated, and any errors that were detected.
(COBRG's error messages are discussed later in this chapter.)

8.1.5 COBRG Command String

To operate COBRG, simply run the program and, in response to the
prompting message, enter a command string of the following form
(listfile.ext is the name and extension of the listing file, and
infile.ext is the name and extension of the input file containing the
specifications for one or more programs):

        <u>CRG></u>listfil.ext=infile.ext

8.1.5.1 Default Assumptions - If the name of the listing file is
omitted, COBRG assigns the name of the input file to the listing file.
If the extension of the listing file is omitted, COBRG assigns .LST as
the extension.

The name of the input file must be entered; COBRG has no default
assumptions for the input file name. If the extension of the input
file is omitted, COBRG assigns .CRG as the extension.

All version numbers are default. Version numbers may not appear in
the command string. Device specifications may not appear in the
command string. To assign devices, assign the appropriate LUN at
task-build time by using the ASG option. See Section 6.5.3, Files and
Logical Units.

For example, to produce a COBOL source program and a listing file from
an input file named COBTST.CRG, enter either of the following two
command strings (they are semantically equivalent).

<u>CRG</u> (RET)	<u>CRG</u> (RET)
<u>CRG></u>COBTST.LST=COBTST.CRG (RET)	<u>CRG></u>=COBTST (RET)

Do not enter the names of the COBOL source programs being produced by
COBRG.   They are not a part of the command string;   COBRG places them
on the device associated with the appropriate LUN with user-assigned
names and extensions of .CBL.

8.1.6 COBRG Sample Program

The following sample program produces a COBOL source program that
lists all of the customers for a marine hardware supply house. It
shows the sales to those customers and provides subtotals for each
city, for each state, and for each region. (The problem is discussed
in detail earlier in this chapter.)

8.1.6.1 Specification Lines - The following input specification lines
will produce the desired source program:

```
N,COBRGTEST,,"SALES.DAT","SALES.LST",F.X. DOE
H,3,58,3,0,COMPANY                    CITY          STATE       SALES PG.
I01   INPUT-RECORD.
I     02   COMPANY          PIC X(20).
I     02   FILLER           PIC X(20).
I     02   CITY             PIC X(10).
I     02   STATE            PIC XX.
I     02   FILLER           PIC X(5).
I     02   REGION           PIC X.
I     02   SALES            PIC S9(10)V99.
I     02   FILLER           PIC X(10).
O     03   DETAIL-LINE.
O          05   PRINT-COMPANY        PIC X(20).
O          05   FILLER               PIC XX.
O          05   PRINT-CITY           PIC X(10).
O          05   FILLER               PIC XX.
O          05   PRINT-STATE          PIC XX.
O          05   FILLER               PIC XX.
O          05   PRINT-SALES          PIC Z,ZZZ,ZZZ,ZZZ.ZZ-.
O          05   LINENO               PIC Z(5).
O     03   CFLINE REDEFINES DETAIL-LINE.
O          05   MESS1                PIC X(20).
O          05   ACCOUNT              PIC Z(5).
O          05   MESS2                PIC X(13).
O          05   FILLER               PIC X(22).
B,0,1,0,0,0,10,CITY
B,1,1,0,0,0,2,STATE
B,2,1,0,0,0,1,REGION
B,F,1,0,0,0
A,0,14,2,SALES
A,1,5,0,1,LINENO
T,0,PRINT-SALES,012F
T,1,ACCOUNT,2F
E,0," CITY SALES =",MESS2
E,1,"STATE SALES =",MESS2
E,2,"REGIONAL ACCOUNTS =",MESS1
E,2,";        SALES =",MESS2
E,F,"NATIONAL ACCOUNTS =",MESS1
E,F,";        SALES =",MESS2
L,A,COMPANY,PRINT-COMPANY
L,A,SALES,PRINT-SALES
L,0,CITY,PRINT-CITY
L,0,STATE,PRINT-STATE
```

8.1.6.2 **Listing File** - When the preceding specification lines have been entered, COBRG produces a listing file containing the following and names it COBTST.LST.

	NAME	BLOCKING	INPUT FILE SPEC	OUTPUT FILE SPEC	DATE	AUTHOR
N	COBRGTEST	O	"SALES.DAT"	"SALES.LST"	08/15/74	F. X. DOE

	PAGE SIZE	HEADER LOC	CONTROL	RESET PAGE	HEADING			
H	3	058	30		COMPANY	CITY	STATE	SALES PG.

	LEVEL	PRINTER-CONTROL BEFORE	AFTER	BREAK SIZE	NAME
B	0	10	00	010	CITY
B	1	10	00	002	STATE
B	2	10	00	001	REGION
B	F	10	00		

	ACCUMULATOR #	SIZE	POINT	INPUT NAME	LISTING NAME
A	0	14	02	SALES	
A	1	05	00	1	LINENO

	AC NUMBER	WHEN	OUTPUT NAME
T	0	012F	PRINT-SALES
T	1	2F	ACCOUNT

	BREAK	INPUT NAME	OUTPUT NAME
E	0	" CITY SALES ="	MESS2
E	1	"STATE SALES ="	MESS2
E	2	"REGIONAL ACCOUNTS ="	MESS1
E	2	"; SALES ="	MESS2
E	F	"NATIONAL ACCOUNTS ="	MESS1
E	F	"; SALES ="	MESS2

WHEN		INPUT NAME	OUTPUT NAME
L	A	COMPANY	PRINT-COMPANY
L	A	SALES	PRINT-SALES
L	0	CITY	PRINT-CITY
L	0	STATE	PRINT-STATE

8.1.6.3 **Source Program** - In addition to the listing file, COBRG also produces the following source program and names it COBRGTEST.CBL.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. COBRGTEST.
AUTHOR. F.X.DOE
DATE-WRITTEN. 05/01/74.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT IN-FILE          ASSIGN TO "SALES.DAT".
    SELECT L-PRINT          ASSIGN TO "SALES.LST".

DATA DIVISION.
FILE SECTION.

FD   IN-FILE
    LABEL RECORDS ARE STANDARD
    VALUE OF ID IS "SALES.DAT".

01   INPUT-RECORD.
    02   COMPANY        PIC X(20).
    02   FILLER         PIC X(20).
    02   CITY           PIC X(10).
    02   STATE          PIC XX.
    02   FILLER         PIC X(5).
    02   REGION         PIC X.
    02   SALES          PIC S9(10)V99.
    02   FILLER         PIC X(10).

FD   L-PRINT
    LINAGE IS 60 LINES
    LINES AT TOP 3
    LINES AT BOTTOM 3
    LABEL RECORDS ARE STANDARD
    VALUE OF ID IS "SALES.LST".

01   PRINT-LINE DISPLAY-7.
    02   FILLER          PIC X(57).
    02   HEADER-PAGE     PIC ZZ9.
    02 FILLER            PIC X(72).

/
WORKING-STORAGE SECTION.

77   PAGE-COUNT         PIC S9(3)       COMP.

77   SAVE-01            PIC X(10).
77   SAVE-02            PIC XX.
77   SAVE-03            PIC X.

77   LEVEL-0-SW         PIC 9.
77   LEVEL-1-SW         PIC 9.
77   LEVEL-2-SW         PIC 9.
```

```
01   OUTPUT-LINE DISPLAY-7.
     03   DETAIL-LINE.
          05   PRINT-COMPANY      PIC X(20).
          05   FILLER            PIC XX.
          05   PRINT-CITY        PIC X(10).
          05   FILLER            PIC XX.
          05   PRINT-STATE       PIC XX.
          05   FILLER            PIC XX.
          05   PRINT-SALES       PIC Z,ZZZ,ZZZ,ZZZ.ZZ-.
          05   LINENO            PIC Z(5).
     03   CFLINE REDEFINES DETAIL-LINE.
          05   MESS1             PIC X(20).
          05   ACCOUNT           PIC Z(5).
          05   MESS2             PIC X(13).
          05   FILLER            PIC X(22).


01   LEVEL-0-ACS COMP.
     02   ACCUMULATOR-0-0    PIC S9(12)V9(2).
     02   ACCUMULATOR-0-1    PIC S9(5).

01   LEVEL-1-ACS COMP.
     02   ACCUMULATOR-1-0    PIC S9(12)V9(2).
     02   ACCUMULATOR-1-1    PIC S9(5).

01   LEVEL-2-ACS COMP.
     02   ACCUMULATOR-2-0    PIC S9(12)V9(2).
     02   ACCUMULATOR-2-1    PIC S9(5).

01   LEVEL-F-ACS COMP.
     02   ACCUMULATOR-F-0    PIC S9(12)V9(2).
     02   ACCUMULATOR-F-1    PIC S9(5).

01   HEADER.
     02   FILLER               PIC X(30)
            VALUE "COMPANY               CITY     ".
     02   FILLER               PIC X(30)
            VALUE " STATE          SALES PG.      ".
     02 FILLER                 PIC X(36)
            VALUE "                                   ".
     02 FILLER                 PIC X(36)
            VALUE "                                  ".
        VALUE "                                      .

/
PROCEDURE DIVISION.

BEGIN.
     OPEN INPUT IN-FILE.
     OPEN OUTPUT L-PRINT.
     MOVE SPACES TO OUTPUT-LINE.

     MOVE ZERO TO PAGE-COUNT.
     PERFORM PL-HDR THRU PL-EXIT.

     READ IN-FILE AT END
          DISPLAY "? EMPTY INPUT FILE" STOP RUN.
     GO TO RESET-LEVEL-F.

READ-IN.
     READ IN-FILE AT END GO TO LEVEL-0-BREAK.
```

```
    IF SAVE-03 NOT = REGION
        MOVE 1 TO LEVEL-2-SW
        GO TO LEVEL-0-BREAK

        GO TO LEVEL-0-BREAK.

    IF SAVE-02 NOT = STATE
        MOVE 1 TO LEVEL-1-SW
        GO TO LEVEL-0-BREAK.

    IF SAVE-01 NOT = CITY
        MOVE 1 TO LEVEL-0-SW
        GO TO LEVEL-0-BREAK.

PRINT-DETAIL.
    ADD SALES                       TO ACCUMULATOR-0-0.
    ADD 1                           TO ACCUMULATOR-0-1.

    MOVE COMPANY                    TO PRINT-COMPANY.
    MOVE SALES                      TO PRINT-SALES.

    MOVE ACCUMULATOR-0-1 TO LINENO.

    MOVE OUTPUT-LINE TO PRINT-LINE.
    PERFORM PRINT-1 THRU PL-EXIT.
    MOVE SPACES TO OUTPUT-LINE.

    GO TO READ-IN.

/
LEVEL-0-BREAK.
    MOVE SPACES TO PRINT-LINE.
    PERFORM PRINT-1 THRU PL-EXIT.

    MOVE ACCUMULATOR-0-0 TO PRINT-SALES.
    MOVE " CITY SALES ="            TO MESS2.
    MOVE OUTPUT-LINE TO PRINT-LINE.
    PERFORM PRINT-1 THRU PL-EXIT.
    MOVE SPACES TO OUTPUT-LINE.

    ADD ACCUMULATOR-0-0 TO ACCUMULATOR-1-0.
    ADD ACCUMULATOR-0-1 TO ACCUMULATOR-1-1.

    IF LEVEL-0-SW = 1 GO TO RESET-LEVEL-0.

LEVEL-1-BREAK.
    MOVE SPACES TO PRINT-LINE.
    PERFORM PRINT-1 THRU PL-EXIT.

    MOVE ACCUMULATOR-1-0 TO PRINT-SALES.
    MOVE "STATE SALES ="            TO MESS2.
    MOVE OUTPUT-LINE TO PRINT-LINE.
    PERFORM PRINT-1 THUR PL-EXIT.
    MOVE SPACES TO OUTPUT-LINE.

    ADD ACCUMULATOR-1-0 TO ACCUMULATOR-2-0.
    ADD ACCUMULATOR-1-1 TO ACCUMULATOR-2-1.

    IF LEVEL-1-SW = 1 GO TO RESET-LEVEL-1.

LEVEL-2-BREAK.
    MOVE SPACES TO PRINT-LINE.
    PERFORM PRINT-1 THRU PL-EXIT.
```

```
        MOVE ACCUMULATOR-2-0 TO PRINT-SALES.
        MOVE ACCUMULATOR-2-1 TO ACCOUNT.
        MOVE "REGIONAL ACCOUNTS ="    TO MESS1.
        MOVE ";      SALES ="         TO MESS2.
        MOVE OUTPUT-LINE TO PRINT-LINE.
        PERFORM PRINT-1 THRU PL-EXIT.
        MOVE SPACES TO OUTPUT-LINE.

        ADD ACCUMULATOR-2-0 TO ACCUMULATOR-F-0.
        ADD ACCUMULATOR-2-1 TO ACCUMULATOR-F-1.

        IF LEVEL-2-SW = 1 GO TO RESET-LEVEL-2.

    LEVEL-F-BREAK.
        MOVE SPACES TO PRINT-LINE.
        PERFORM PRINT-1 THUR PL-EXIT.

        MOVE ACCUMULATOR-F-0 TO PRINT-SALES.
        MOVE ACCUMULATOR-F-1 TO ACCOUNT.
        MOVE "NATIONAL ACCOUNTS ="    TO MESS1.
        MOVE ";      SALES ="         TO MESS2.
        MOVE OUTPUT-LINE TO PRINT-LINE.
        PERFORM PRINT-1 THRU PL-EXIT.
        MOVE SPACES TO OUTPUT-LINE.

        GO TO FINISH.

    /
    RESET-LEVEL-F.
        MOVE LOW-VALUES TO LEVEL-F-ACS.

    RESET-LEVEL-2.
        MOVE LOW-VALUES TO LEVEL-2-ACS.
        MOVE REGION                  TO SAVE-03.
        MOVE ZERO TO LEVEL-2-SW.

    RESET-LEVEL-1.
        MOVE LOW-VALUES TO LEVEL-1-ACS.
        MOVE STATE                   TO SAVE-02.
        MOVE ZERO TO LEVEL-1-SW.

    RESET-LEVEL-0.
        MOVE LOW-VALUES TO LEVEL-0-ACS.
        MOVE CITY                    TO SAVE-01.
        MOVE CITY                    TO PRINT-CITY.
        MOVE STATE                   TO PRINT-STATE.
        MOVE ZERO TO LEVEL-0-SW.
        GO TO PRINT-DETAIL.

    /
    PRINT-1.
        WRITE PRINT-LINE BEFORE 1 LINES
             AT EOP GO TO PL-HDR.
        GO TO PL-EXIT.

    PRINT-3.
        WRITE PRINT-LINE BEFORE 3 LINES
             AT EOP GO TO PL-HDR.
        GO TO PL-EXIT.

    PL-HDR.
        MOVE SPACES TO PRINT-LINE.

    PRINT-CH-1.
        WRITE PRINT-LINE BEFORE PAGE.
```

```
    MOVE HEADER TO PRINT-LINE.
    ADD 1 TO PAGE-COUNT.
    MOVE PAGE-COUNT TO HEADER-PAGE.
    WRITE PRINT-LINE BEFORE 3 LINES.

PL-EXIT. EXIT.

FINISH.
    CLOSE IN-FILE.
    CLOSE L-PRINT.

    STOP RUN.
```

8.1.6.4 **Input File** - COBRG's job is finished when it has produced the listing file and source program. The source program must now be compiled, task-built, and executed with the following input file:

```
WIDGET BOATS          13217 ATLANTIC AVE  BRIDGEPORTCT066041000002201942
TURNBUCKLE MARINE     477 MAIN ST         BRIDGEPORTCT066041000001196700
ANCHOR BOATS          991 SEAWARD RD      BRIDGEPORTCT066041000000812559
CHAIN REPAIR          94 WATER ST         BRIDGEPORTCT066041000000487300
BLOCK MARINA          9473 ATLANTIC AVE   BRIDGEPORTCT066041000000023173
WINDY BOAT SALES      741 FRONT ST        NEW HAVEN CT065131000001278811
WINDWARD SAILS        21 OCEANSIDE        NEW LONDONCT063201000000734612
HARBOR MARINA         137 FRONT ST        NEW LONDONCT063201000004669400
SNUG HARBOR MARINE    14 HIDDEN COVE RD   NEW LONDONCT063201000000445227
BLUE WATER SAILS      BOX 973             NEW LONDONCT063201000001350000
CABIN OUTFITTERS      4973 ATLANTIC AV    BOSTON    MA021011000003151991
SAFE HARBOR MARINA    788 ATLANTIC AV     BOSTON    MA021031000001246774
HIDDEN COVE MARINA    HIDDEN COVE RD      BOSTON    MA021031000000962588
SPINNAKER SPECIALTY   24 SEAWARD LANE     BOSTON    MA021021000001837300
CUDDY BOAT SALES      244 HARBOR RD       BOSTON    MA021041000002173100
HARBOR MARINE         T WHARF             BOSTON    MA021011000001228800
WIDGET SALES & SERV.  17MAIN ST           FALMOUTH  MA025401000004184651
HARBOR RIGGING        49 FRONT ST         FALMOUTH  MA025401000000929451
HARBOR BOAT SALES     49 FRONT ST         FALMOUTH  ME041051000000895200
OCEAN SAILS           51 FRONT ST         FALMOUTH  ME041051000000300054
OCEAN HARDWARE        LAND'S END          FALMOUTH  ME041051000001981800
TURNBUCKLE HARDWARE   HARBOR LANE         PORTLAND  ME041011000002726629
ANCHOR SALES          95 ATLANTIC ST      PORTLAND  ME041011000002726629
SALT WATER HARDWARE   14 BAY RD           PORTLAND  ME041011000004617257
SAFE ANCHORAGE        LEEWARD LANE        PORTLAND  ME041011000000173000
HARBOR RIGGING        73 HARBOR RD        PORTLAND  ME041011000000138917
OUT-BOUND MARINE      721 ATLANTIC AVE    PORTLAND  ME041011000001464773
HARBOR TACKLE         24 EAST MAIN ST     PORTLAND  ME041011000002529524
OCEAN BOATS           196 BAY RD          PORTLAND  ME041011000004255300
DOCKSIDE REPAIR       MAIN WHARF          PORTLAND  ME041011000001310171
MARINE HARDWARE       94 FRONT ST         PORTLAND  ME041011000000981922
HARBOR ROPE WORKS     BACKWATER LANE      PORTLAND  ME041011000000076700
WIDGET HARDWARE       14 MAIN ST          PORTSMOUTHNH038011000000822599
SNUG COVE TACKLE      SNUG COVE RD        PORTSMOUTHNH038011000002137313
OCEAN SAILMAKERS      NEWELL LANE         PORTSMOUTHNH038011000004713129
WIDGET BOAT HAUL      317 RIVER ST        PORTSMOUTHNH038011000001248837
HARBOR BOAT SALES     245 RIVER ST        PORTSMOUTHNH038011000007654644
GALLEY HARDWARE       724 RIVER ST EXTEN. PORTSMOUTHNH038011000000369475
RACING RIGGING        BOX 427             PORTSMOUTHNH038011000003595200
BOAT BUILDERS         425 TIDE FLAT RD    PORTSMOUTHNH038011000000400022
INLAND BOAT SALES     137 LAKE ST         MONTPELIERVT056021000000101999
SNUG COVE BOATS       21 BAY ST           MONTPELIERVT056021000000106748
EASTERN LAKE MARINA   91 HARBOR RD        BUFFALO   NY142222000000402575
GREAT LAKE BOATS      1321 HARBOR RD      BUFFALO   NY142252000000347300
HARBOR SLS & SERV     FRONT ST            BUFFALO   NY142252000000233194
```

8.1.6.5 Printed Report - As it reads the preceding input file, the
task image containing COBRGTEST.OBJ produces the following report:

```
COMPANY                CITY        STATE       SALES PG.      1


WIDGET BOATS           BRIDGEPORT  CT          22,019.42      1
TURNBUCKLE MARINE                              11,867.00      2
ANCHOR BOATS                                    8,125.54      3
CHAIN REPAIR                                    4,873.00      4
BLOCK MARINE                                      231.73      5

                       CITY SALES =            47,216.74
WINDY BOAT SALES       NEW HAVEN   CT          12,788.11      1

                       CITY SALES =            12,788.11
WINDWARD SAILS         NEW LONDON  CT           7,346.12      1
HARBOR MARINA                                  86,694.00      2
SNUG HARBOR MARINE                              4,452.27      3
BLUE WATER SAILS                              13,500.00      4

                       CITY SALES =            71,992.39

                       STATE SALES =          131,987.24
CABIN OUTFITTERS       BOSTON      MA          31,519.91      1
SAFE HARBOR MARINA                            12,467.74      2
HIDDEN COVE MARINA                             9,625.88      3
SPINNAKER SPECIALTY                           18,373.00      4
CUDDY BOAT SALES                              21,731.00      5
HARBOR MARINE                                 12,288.00      6

                       CITY SALES =          106,005.53
WIDGET SALES & SERV.   FALMOUTH    MA          41,846.51      1
HARBOR RIGGING                                 9,294.51      2

                       CITY SALES =           51,141.02

                       STATE SALES =          157,146.55
HARBOR BOAT SALES      FALMOUTH    ME           8,952.00      1
OCEAN SAILS                                     3,000.54      2
OCEAN HARDWARE                                19,818.00      3

                       CITY SALES =           31,770.54
TURNBUCKLE HARDWARE    PORTLAND    ME          27,266.29      1
ANCHOR SALES                                  27,266.29      2
SALT WATER HARDWARE                           46,172.57      3
SAFE ANCHORAGE                                 1,730.00      4
HARBOR RIGGING                                 1,389.17      5
OUT-BOUND MARINE                              14,647.73      6
HARBOR TACKLE                                 25,295.24      7
OCEAN BOATS                                   42,553.00      8
DOCKSIDE REPAIR                               13,101.71      9
MARINE HARDWARE                                9,819.22     10
HARBOR ROPE WORKS                                767.00     11

                       CITY SALES =          210,008.22

                       STATE SALES =          241,778.76
WIDGET HARDWARE        PORTSMOUTH  NH           8,225.99      1
SNUG COVE TACKLE                              21,373.13      2
OCEAN SAILMAKERS                              47,131.29      3
WIDGET BOAT HAUL                              12,488.37      4
```

COMPANY	CITY	STATE	SALES PG.	2

HARBOR BOAT SALES	76,546.44		5	
GALLEY HARDWARE	3,694.75		6	
RACING RIGGING	35,952.00		7	
BOAT BUILDERS	4,000.22		8	
	CITY SALES =		209,412.19	
	STATE SALES =		209,412.19	
INLAND BOAT SALES	MONTPELIER VT		1,019.99	1
SNUG COVE BOATS	1,067.48		2	
	CITY SALES =		2,087.47	
	STATE SALES =		2,087.47	
REGIONAL ACCOUNTS =	42; SALES =		742,422.21	
EASTERN LAKE MARINA	BUFFALO NY		4,025.75	1
GREAT LAKE BOATS	3,473.00		2	
HARBOR SLS & SERV	2,331.94		3	
	CITY SALES =		9,830.69	
	STATE SALES =		9,830.69	
REGIONAL ACCOUNTS =	3; SALES =		9,830.69	
NATIONAL ACCOUNTS =	45; SALES =		752,252.90	

8.1.7 COBRG Error Messages

If any of the specification lines contain errors, or if the number of one type of specification lines exceeds COBRG's limit for that type, the listing file will contain the following error messages and COBRG will not produce the COBOL program specified.

Before issuing an error message, COBRG assumes that any non-numeric character appearing in a numeric parameter is a zero. It ignores imbedded blanks; however, if a numeric parameter contains all blanks, COBRG assumes the blanks to be zeroes.

SPECIFICATION LINE	MESSAGE	EXPLANATION
(ALL)	IMPROPER INPUT CARD	Parameter 1 (specification identification) of a specification line does not contain one of the following characters: N, I, O, H, B, A, T, E, L. (COBRG accepts 1 in place of I and 0 in place of O.)
ACCUMULATOR	DUPLICATE AC	Parameter 2 (accumulator number) contains a number that has already been specified.
	IMPROPER AC SIZE	Parameter 3 (accumulator size) does not contain a number from 1 to 18.

SPECIFICATION LINE	MESSAGE	EXPLANATION
	IMPROPER POINT LOCATION	Parameter 4 (decimal positions) contains a number that places the decimal point outside of the accumulator.
	NO INPUT NAME	Parameter 5 (input field) does not contain an entry, leaving the accumulator with no input.
BREAK	BOTH SPACE AND SKIP	Both parameter 3 (blank lines, before printing) and parameter 4 (skip channel, before printing) contain entries, or both parameter 5 (blank lines after printing) and parameter 6 (skip channel, after printing) contain entries. (COBRG will not accept both a space and a skip before printing, or both a skip and a space after printing.)
	IMPROPER FIELD SIZE	Parameter 7 (field size) contains a zero.
	IMPROPER LEVEL	Parameter 2 (break level) contains an entry other than 0-9, or F.
	IMPROPER SPACE OR SKIP	Either parameter 3 (blank lines, before printing) does not contain a number from 0-9; or parameter 4 (skip channel, before printing), parameter 5 (blank lines, after printing), or parameter 6 (skip channel, after printing) does not contain a number from 0-8. COBRG looks for the first non-blank character in the parameter and assumes non-numeric characters to be zeroes.)
	TOO MANY BREAKS	COBRG found more than 25 BREAK specification cards.
EMIT	IMPROPER BREAK NUMBER	Parameter 2 (break number) contains a number other than 0-9 or F.
	NO INPUT NAME	Parameter 3 (message) does not contain an entry.
	NO OUTPUT NAME	Parameter 4 (output field) does not contain an entry.
	TOO MANY EMIT CARDS	COBRG found more than 50 EMIT specification lines.
	UNDEFINED BREAK NUMBER	Parameter 2 (break number) contains a break level number that has not been defined on a BREAK specification line.

SPECIFICATION LINE	MESSAGE	EXPLANATION
HEADER	IMPROPER BREAK FOR RESET	Parameter 7 (page number reset) contains an F. The page counter cannot be reset on an F (final) break level (the report is completed with that break).
	IMPROPER HEADER SKIPPING	Parameter 5 (skip channel) contains a 9.
	NO PAGE SIZE	Parameter 2 (page number field size) contains no entry; however, parameter 3 (page number location) does contain an entry (indicating that a page number field is required by the program).
	NON-EXISTENT BREAK FOR RESET	Parameter 7 (page number reset) contains a break level number that has not been defined on a BREAK specification line.
	PAGE LOC TOO FAR RIGHT	Parameter 3 (page number location) contains a column number that places it so far to the right of the page that it runs off the right side (past column 132).
	SAME BREAK NOTED TWICE	Parameter 7 (page number reset) contains more than one entry of the same break level.
	TOO MANY HEADER CARDS	COBRG found more than two HEADER cards.
INPUT	MORE THAN 1 BATCH OF I CARDS	COBRG found some other specification line separating the INPUT specification lines. (All INPUT specification lines must be consecutive, as must all OUTPUT specification lines.)
	TOO MANY RECORD CARDS	The sum of all of the INPUT and OUTPUT specification lines exceeds 997.
LIST	TOO MANY LIST CARDS	COBRG found more than 50 LIST specification lines.
	IMPROPER BREAK NUMBER	Parameter 2 (control break level) contains an entry other than 0-9 or A.
	NO INPUT NAME	Parameter 3 (sending field) does not contain an entry.
	NO OUTPUT NAME	Parameter 4 (receiving field) does not contain an entry.

SPECIFICATION LINE	MESSAGE	EXPLANATION
	UNDEFINED BREAK LEVEL	<u>Parameter 2</u> (control break level) contains a break level number that has not been defined on a BREAK specification line.
NAME	NO PROG-ID	<u>Parameter 2</u> (program name) is missing.
	NO INPUT FILE SPEC	<u>Parameter 4</u> (input file specification) is missing.
	NO OUTPUT FILE SPEC	<u>Parameter 5</u> (output file specification) is missing.
OUTPUT	MORE THAN 1 BATCH OF O CARDS	COBRG found some other specification line separating the OUTPUT specification lines. (All OUTPUT specification lines must be consecutive, as must all INPUT specification lines.)
	TOO MANY RECORD CARDS	The sum of all of the OUTPUT and INPUT specification lines exceeds 997.
TOTAL	DUPLICATE BREAKS	<u>Parameter 4</u> (break numbers) contains the same break level number twice.
	IMPROPER BREAK	<u>Parameter 4</u> (break numbers) contains an entry other than 0-9 or F
	NON-EXISTENT AC	<u>Parameter 2</u> (accumulator number) contains an accumulator number that has not been set up with an ACCUMULATOR specification line.
	NO BREAKS SPECIFIED	<u>Parameter 4</u> (break numbers) does not contain an entry.
	NO OUTPUT NAME	<u>Parameter 3</u> (output field) does not contain an entry.

8.2 REFORMAT

8.2.1 Introduction

COBOL, as implemented on the PDP-11, accepts source programs that were coded using either the conventional 80-column card reference format or the shorter, terminal-oriented PDP-11 terminal format. The REFORMAT utility program reads source programs that were coded in the terminal format and converts them to 80-column conventional format source programs.

Consider the two formats (Section 2.1, Choosing a Reference Format, discusses both formats in greater detail):

- The terminal format is designed for ease of use with context editors controlled from an on-line console keyboard and is compatible for use with the PDP-11 system. It eliminates the line-number and identification fields. It allows horizontal tab characters and short lines.

- The conventional format produces source programs that are compatible with the reference format of other COBOL compilers throughout the industry.

REFORMAT lets you write source programs in the Terminal format; then, if compatibility is ever required for any of those programs, it provides a simple method for conversion to the Conventional format.

REFORMAT uses the following steps to expand each line of Terminal format coding to the 80-character Conventional format coding:

- It generates a 6-character line number of 000010, places that number in the first six character positions of the line, and increases it by 000010 for each subsequent line;

- It places any continuation or comment symbols (-,*, or /) into character position 7;

- It places the coding from the Terminal format line into character positions 8-72, thereby creating a line of Conventional format coding;

- It replaces any horizontal tabs with the appropriate number of space characters to simulate tab stops at character positions 5,13,21,29,37,45,53,61, and 66 of the Terminal format line:

- It moves spaces into any character positions left between the last character of coding and character position 73;

- It places either identification characters (if they were supplied at program initialization) or spaces into character positions 73-80;

- It right justifies (against margin R) the first line of a continued non-numeric literal, thus guaranteeing that the literal will remain the same length as it was in the default format;

- It right justifies (against margin R) the first part of any COBOL word that is split over two lines;

- It creates a line containing a slash (/) in position 7 and space characters in positions 8 through 72 for every form-feed character that it encounters.

8.2.2 REFORMAT Command String

Since REFORMAT is written in COBOL, it runs as a COBOL object program. It has no logical switches. To run it, simply type in the following sequence of responses (prompting messages typed by REFORMAT are underlined):

RFM (RET)

This causes REFORMAT to begin execution. REFORMAT immediately requests the file specifications for the two files (input and output) to be processed. In response to its prompting messages, type in the file specifications for your two files.

<u>RFM-INPUT FILE SPEC:</u>
<u>RFM-OUTPUT FILE SPEC:</u>

When the system has successfully opened both files, REFORMAT types the following request for an identification entry in columns 73 through 80. If you desire an identification entry, type in from one to eight characters. REFORMAT places these characters, left justified, in columns 73 through 80 of each output line. If no entry is required, type a carriage return.

<u>RFM-COLS 73 TO 80:</u>

Following this response, REFORMAT reads the input file and writes it as 80-character records, in Conventional reference format.

When it has processed the last record in the file, REFORMAT displays the following messages; the first indicating the number (nnnnn) of output records produced and the second requesting another input file.

<u>RFM-nnnnn LINES PROCESSED.</u>
<u>RFM-INPUT FILE SPEC:</u>

If there is another file to be reformatted, follow the same sequence with the specifications for the next file. If not, type Control Z to terminate execution.

8.2.3 REFORMAT Error Messages

If any of the responses to the prompting messages contain detectable errors, REFORMAT displays the following messages indicating the problem.

<u>RFM-ERROR IN OPENING INPUT FILE</u>
<u>RFM-TRY AGAIN</u>
<u>RFM-INPUT FILE SPEC:</u>

The system could not open the input file. Either the file is not present on the device specified (the default device is SY:) or the file name is typed incorrectly. The usual I/O error messages precede this message.

To continue processing that file, examine the input file spec and type in a corrected version. To process another file, type in a new input file specification. To terminate execution, type Control Z.

<u>RFM-ERROR IN OPENING OUTPUT FILE</u>
<u>RFM-TRY AGAIN</u>
<u>RFM-OUTPUT FILE SPEC:</u>

The system could not open the output file. An incorrectly typed file specification usually causes this error. (The default device is SY:.) The usual I/O error messages precede this message.

To continue, examine the output file specification and type in a
corrected version. To terminate execution, type Control Z.

> RFM-INPUT FILE IS EMPTY
> RFM-INPUT FILE SPEC:

The system successfully opened the input file, but the first READ
statement encountered the AT END condition.

To continue, type in a new input file specification for another file.
To terminate execution, type Control Z.

> RFM-ERROR IN READING INPUT FILE
> RFM-INPUT FILE SPEC:

The first attempt to read the input file was unsuccessful. This error
is usually caused by an input record length exceeding 86 characters.
(Although terminal format records should not exceed 66 characters in
length, REFORMAT provides a record area of 86 characters and ignores
the right-most 20 characters.)

To continue, type in a new input file specification for another file.
To terminate execution, type Control Z.

> RFM-ERROR IN READING INPUT FILE
> RFM-REFORMATTING ABORTED
> RFM-nnnnn LINES PROCESSED
> RFM-INPUT FILE SPEC:

While reading input records (other than the first record), REFORMAT
was unsuccessful in an attempt to read a record. It terminates
execution and closes both files.

To process another file, type in a new input file specification and
continue with the prompting message sequence. To terminate execution,
type Control Z.

> RFM-ERROR IN WRITING OUTPUT FILE
> RFM-REFORMATTING ABORTED
> RFM-nnnnn LINES PROCESSED
> RFM-INPUT FILE SPEC:

REFORMAT was unsuccessful in an attempt to write an output record. It
terminates execution and closes both files.

To process another file, type in a new input file specification and
continue with the prompting message sequence. To terminate execution,
type Control Z.

CHAPTER 9

SEGMENTATION

PDP-11 COBOL allows you to break the Procedure Division up into
overlayable and non-overlayable program segments to optimize memory
utilization. An overlayable program segment can be overlayed by any
other overlayable segment. A non-overlayable program segment,
however, can never be overlayed.

NOTE

The object code generated for the
Identification Division through the Data
Division is non-overlayable.

9.1 USING THE PDP-11 COBOL SEGMENTATION FACILITY

The PDP-11 COBOL Segmentation Facility allows you to specify your own
segmentation requirements. To effect segmentation, you must define a
segment limit by specifying the SEGMENT-LIMIT IS clause in the
Environment Division of your source program. The value you specify in
this clause is used by the compiler as a basis for determining whether
a program segment is overlayable or non-overlayable. A segment
consists of one or more COBOL sections. Each COBOL section should be
composed of a series of closely related operations designed to
collectively perform a particular function. To designate a section as
belonging to an overlayable or non-overlayable segment, assign a
segment number to it using the following format:

    Section-name SECTION segment-number.

Where:

    Section-name   Is a user-defined COBOL word that names the
                    section

    Segment-number Is an integer ranging from 0 to 49.

If you specify a segment-number whose value is less than the value
specified in the SEGMENT-LIMIT IS clause, you have defined the section
as being non-overlayable. A segment-number whose value is greater
than or equal to the value specified in the SEGMENT-LIMIT IS clause
defines the segment as being overlayable.

9.1.1 Programming Considerations

The most frequently used sections of your program should be made non-overlayable. Assign segment-numbers that are less than the value specified in the SEGMENT-LIMIT IS clause to these 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 IS clause to these sections. Sections that communicate with each other should be assigned to the same segment. Assign the same segment-number to these sections. Sections having identical segment-numbers are assigned to the same segment.

9.2 SEGMENTATION AND THE PDP-11 COBOL COMPILER

The previous sections told you how to effect segmentation. This section tells you what segmentation means in terms of code generation. The PDP-11 COBOL compiler breaks up the object code it generates into program sections called PSECTs. One or more PSECTs are generated for each program SECTION. The maximum size PSECT generated is 2000 decimal words. However, this maximum size can be altered by specifying the /CSEG:nnnn switch in the compiler command line. (PSECTs are described in Appendix D: The /CSEG:nnnn Switch is described in Section 9.4, and Section 2.5.3).

Also generated, are Overlay Description Language (ODL) directives that group together all PSECTs that belong in the same overlay. These ODL directives are placed in an ODL file to be used as input to the Task Builder. The Task Builder uses the ODL file to generate a task image containing the correct combination of overlayable/non-overlayable PSECTs.

If the source program is written without explicit segmentation, all of the generated PSECTs are concatenated into one non-overlayable program. If the source program does contain explicit segmentation, ODL directives are created to group PSECTs together into the correct combination of overlayable and non-overlayable program segments.

9.3 SEGMENTATION USING THE /OV SWITCH

The /OV switch, when appended to the compiler command line, directs the compiler to produce ODL directives that make all of the procedural PSECTs overlayable. Therefore, the amount of memory required to store the program is equal to that required to contain the root (non-overlayable portion) and the largest PSECT. (See Figure 9-1, Segmentation Using The /OV Switch; and Section 2.5.3, Compiler Switches).

The /OV switch is particularly useful for quickly segmenting programs that were written without explicit segmentation or for overriding explicit segmentation.
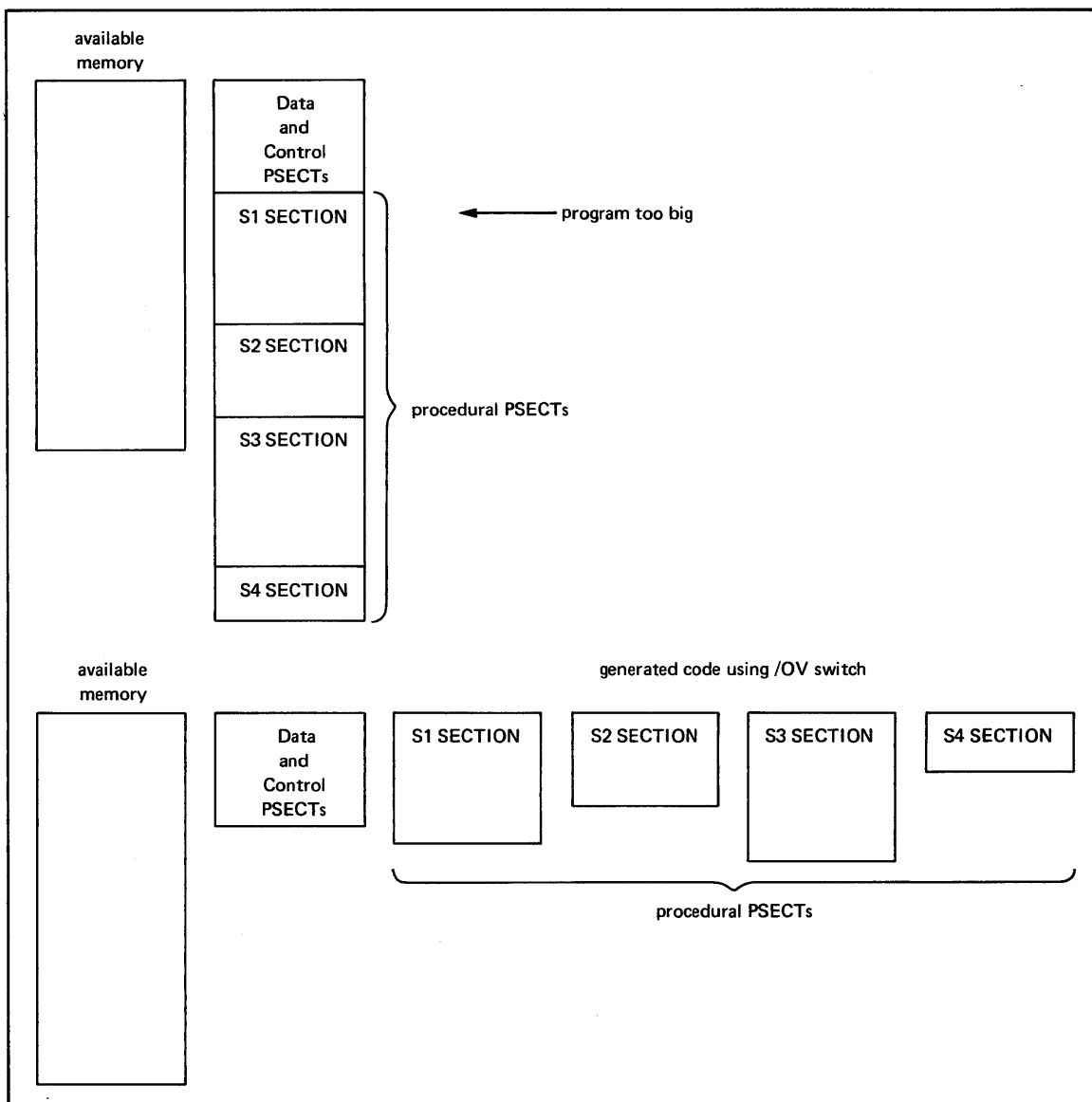
Figure 9-1   Segmentation Using the /OV Switch

9.4   USING THE /CSEG:nnnn SWITCH

The PDP-11 COBOL compiler generates a PSECT for each COBOL section.
If the code generated for a particular section exceeds the default
maximum size for a PSECT (4000 decimal bytes), more than one PSECT is
generated. PDP-11 COBOL provides a switch (/CSEG:nnn) that allows you
to control the size of PSECTs generated by the compiler. (See Section
5.2.3 Compiler Switches).

If, for example, you compile a program that produces a PSECT that is
too large to be task built, you can recompile the program using the
/CSEG:nnn switch to reduce the size of the PSECTs generated. See
Figure 9-2 for an example of using the /CSEG:nnnn switch.

Command Line (Without /CSEG:nnnn switch specified)

　　CBL> CBLMRG,CBLMRG/MAP=CBLMRG (RET)

SEGMENTATION MAP

SECTION NAME	SEGMENT NO.	NAME		SIZE
OUTPUT-ODL-USE	00	C001	000172	00061
INPUT-ODL-USE	00	C002	000172	00061
* MAIN-CONTROL	00	C003	003336	00879
PROCESS-INPUT-ODL	00	C004	001130	00300
HDR-CHECK	00	C005	000322	00105

* One large PSECT is generated

Command Line (With the /CSEG:nnnn switch specified)

　　CBL> CBLMRG,CBLMRG/MAP/CSEG:1000=CBLMRG (RET)

SEGMENTATION MAP

SECTION NAME	SEGMENT NO.	NAME		SIZE
OUTPUT-ODL-USE	00	C001	000172	00061
INPUT-ODL-USE	00	C002	000172	00061
* MAIN-CONTROL	00	C003	001744	00498
	00	C004	001426	00395
PROCESS-INPUT-ODL	00	C005	001130	00300
HDR-CHECK	00	C006	000322	00105

* Two PSECTs are generated

Figure 9-2　Using the /CSEG:nnnn Switch

CHAPTER 10

INTER-PROGRAM COMMUNICATIONS

Inter-program communications is the passing of control and optional
data from one program within a task to another. PDP-11 COBOL provides
you with the ability to Task-build separately compiled COBOL programs
into a single task image. During task execution, these separately
compiled programs can communicate with each other via the COBOL CALL
statement.

A task can consist of a stand-alone program or a main program and one
or more subprograms. A stand-alone program is one that does not call
subprograms and cannot itself be called. A COBOL main program is one
that calls subprograms but can never be called in return. A COBOL
subprogram, however, is always called by another program, either the
main program or a subprogram. Inter-program communications deals only
with main programs and subprograms.

Developing a program as a main program and a set of subprograms offers
a number of advantages:

1. Large monolithic programs are no longer required. These
    large programs can be replaced by a controlling main program
    and a set of subprograms, where each subprogram is designed
    to perform a well-defined function.

2. Small subprograms can be developed independently by several
    members of a programming staff.

3. Small subprograms can be tested more easily than large
    programs.

4. Small subprograms can be modified and recompiled faster than
    large programs.

5. General purpose subprograms can be developed and used in more
    than one programming application.

10.1 COBOL MAIN PROGRAMS VS SUBPROGRAMS

A COBOL main program is one that calls other programs (subprograms)
but cannot be called in return. A COBOL main program contains at
least one CALL statement. A COBOL subprogram is one that is called by
another program, either the main program or another subprogram. The
main program is automatically activated at task execution time. A
subprogram, however, is activated only when called by another program.

The COBOL compiler differentiates between a main program and a
subprogram by the presence or absence of a USING phrase in the
Procedure Division header of the program being compiled. The USING

phrase is used only in COBOL subprograms. It defines the program as being a subprogram and optionally identifies the data expected from the calling program. The Procedure Division header has the following format:

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

A subprogram, that does not process data (arguments) passed to it by a calling program, has only the word USING appended to the Procedure Division header. For example:

    PROCEDURE DIVISION USING.

A subprogram that processes passed data, has a USING phrase with one or more data-names specified. If a data-name(s) is specified, the program must also contain a Linkage Section in the Data Division. The Linkage Section describes the size and type of data being passed. (See Figure 10-1, Sample LINKAGE SECTION and USING phrase).

```
LINKAGE SECTION.

*      SUBPROGRAM-DATA.

01      1ST-PARAMETER  PIC X(5).

01      2ND-PARAMETER  PIC X(5).

01      3RD-PARAMETER  PIC X(5).

PROCEDURE DIVISION USING 2ND-PARAMETER, 3RD-PARAMETER.
```

                                 NOTES
    1. All of the data-names appearing in the using phrase must also
        appear in the LINKAGE SECTION.

    2. Not all of the data-names in the LINKAGE SECTION need appear
        in the USING phrase.

    3. A LINKAGE SECTION can appear in a subprogram even if the
        USING phrase does not contain a data-name. However, if any
        of the data-items contained in the LINKAGE SECTION are
        referenced in procedures, the compiler will issue a fatal
        diagnostic.

          Figure 10-1 Sample LINKAGE SECTION and USING Phrase

10.1.1 Calling a COBOL Subprogram from a COBOL Program

To call a subprogram from a COBOL program, a CALL statement having the following format must be used:

    CALL literal [USING data-name-1 [, data-name-2]...].

Where:

      literal   is the name that appears in the PROGRAM-ID entry of the called program.

      data-name-1   identify those data-items in the calling program
        through   that can be referred to by the called program.
      data-name-n

10.1.2 Returning from a COBOL Subprogram

In addition to the required USING phrase and optional LINKAGE SECTION, a subprogram should contain at least one EXIT PROGRAM statement. The EXIT PROGRAM statement identifies the subprogram return point.   That is, the point in the subprogram at which control is returned to the calling program. If the EXIT PROGRAM statement is missing, the COBOL compiler will generate one after the last statement in the program.

NOTE

      More than one EXIT PROGRAM statement is
      allowed in a subprogram.

10.2 UNIQUENESS OF PSECT NAMES

The names of all PSECTs within a task must be unique. When a task is composed of more than one COBOL program, you must insure that the PSECTs generated by the COBOL compiler for each program are unique. (See Section D.1, PSECT Naming Conventions).

10.3 COBOL OTS - ERROR CHECKING

At task execution, the COBOL OTS performs a check to insure that the number of arguments passed to a called COBOL subprogram is the same as the number expected. That is, the subprogram Procedure Division USING phrase must contain the same number of data-names as the USING phrase in the calling programs CALL statement. If the number of data-names in each USING phrase are not equal, the OTS issues a diagnostic error message and aborts the task. No checks are made to insure that the passed arguments are the same size as the expected arguments. It is your responsibility to insure that these sizes are compatible.

Recursive calls to COBOL subprograms are not allowed.   If a COBOL subprogram contains a CALL statement that directly or indirectly causes a subprogram to be re-entered before it has exited from its original entry, the OTS will issue a diagnostic error message and abort the task.

10.4 INCLUDING A NON-COBOL OBJECT MODULE IN A TASK

Non-COBOL object modules can be combined with COBOL object modules at task-build time to produce a single task image. However, you are advised to use the same language to write programs that perform I/O operations. This note of caution is very important, because, the PDP-11 programming languages do not share a common OTS. The following is a list of the PDP-11 programming languages that can be used in conjunction with COBOL:

            IAS and RXS-11M   RSTS/E

            FORTRAN IV   MACRO
            FORTRAN IV+   BASIC+2
            BASIC+2
            MACRO

To activate a COBOL subprogram, a non-COBOL calling program must contain the equivalent of a COBOL CALL statement. If data is being passed to the COBOL 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 10-2, Argument List Format).

A non-COBOL subprogram, to be activated by a COBOL program, must contain the equivalent of the COBOL PROGRAM-ID statement and the COBOL EXIT PROGRAM statement (See Example 1 below). If data is being passed, the non-COBOL subprogram can access that data via program register R5.

The following sections provide an example of how non-COBOL programs can be written for inclusion in a COBOL task image. The MACRO programming language is used for the purposes of this example.

Example 1 - (Calling MACRO Programs from COBOL)

The format for calling any program from COBOL is:

        CALL literal [USING data-name-1[, data-name-2]...]

when a MACRO program is being called, literal contains the global entry point specified in the MACRO program. If the COBOL program contains:

        CALL "BILBO" USING BOFFIN, BOMBUR, BOFUR.

The MACRO program must contain:

        .GLOBL BILBO ⎞
                      ⎬ ;entry point - equivalent to PROGRAM-ID
        BILBO: ⎠
            .
            .
            .
            .
            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 via program register R5.

Example 2 - (Calling COBOL Programs from MACRO)

When the calling program is a MACRO program, control is passed to the called program with the following instruction:

        JSR PC,subprogram-name

Where: Subprogram-name is the first six characters of the COBOL PROGRAM-ID.

If the MACRO program contains:

        .GLOBL FRODO
            .
            .
            .
            .
            MOV #ARGLST,R5   ;point R5 to argument list

            JSR PC,FRODO   ;subprogram call statement

The COBOL subprogram will contain:

        PROGRAM-ID. FRODO
            .
            .
            .
            .
        LINKAGE SECTION.

        *   FRODO-ARGUMENTS.

        01   BOFFIN   PIC X(5).

        01   BOMBUR   PIC X(5).

        01   BOFUR   PIC X(5).
            .
            .
            .
            .
        PROCEDURE DIVISION USING BOFFIN,BOMBUR.
            .
            .
            .
            .
            EXIT PROGRAM.

The MACRO program, in this example, has set R5 to point to the argument list expected by the COBOL program. The COBOL OTS will use R5 to access the passed arguments.
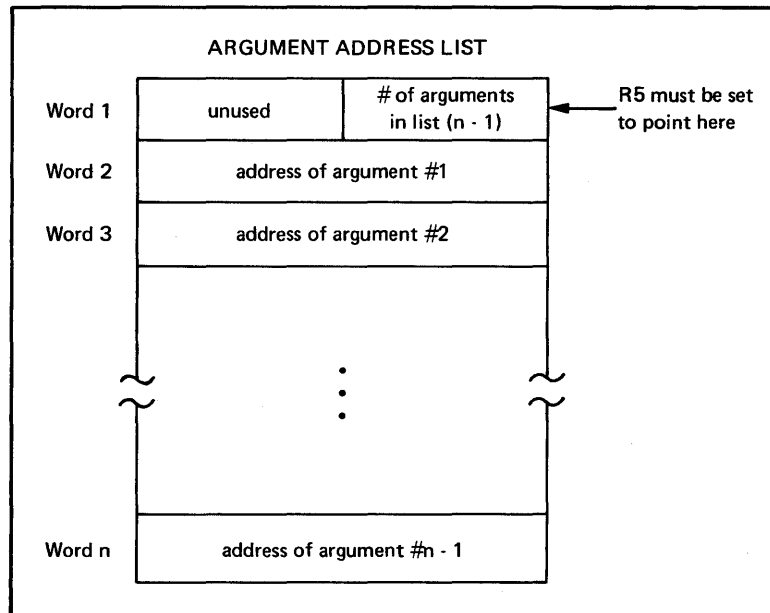
Figure 10-2   Argument Address List

CHAPTER 11

HAND-TAILORING ODL FILES

This chapter is provided as a guide to those of you who are faced with the problem of having to generate ODL files that are compatible with either the Merge Utility or the Task Builder. The most common reason for having to hand tailor an ODL file occurs when non-COBOL programs are being merged into a COBOL task image. The information presented here is predicated on the assumption that you have read and are familiar with the Task Builder Reference Manual that pertains to your operating system. The following sections describe the standard ODL file as it pertains to PDP-11 COBOL.

11.1 STANDARD ODL FILE

The standard ODL file generated by the PDP-11 COBOL compiler consists of a header and a body. The header contains information that is required to merge one or more ODL files. The body contains ODL directives that describe the object program.

11.2 ODL FILE HEADER

The ODL file header consists of a sequence of comment lines. Two are required in every ODL file, others are supplied as needed. The required comment lines are:

```
;COBOBJ=XXXXXX.OBJ
;COBKER=KK
```

Where:

      XXXXXX.OBJ   is the name of the object module being described

      KK   is the kernel that was used to generate the PSECT names for the COBOL program.

The following comment lines are supplied as needed:

      ;COBMAIN   This comment line is supplied if the program being described is a main program. The absence of this line means that the ODL file was generated for a COBOL subprogram.

      ;RMSSEQ=CIOOXY This comment line is specified if the program requires RMS-11 I/O support. One or more lines may be supplied. X and Y represent integer codes that respectively specify the file organization and operational support required for that

organization.   File   organization is specified by
the following codes:

CODE   ORGANIZATION

1   sequential

2   relative

3   indexed

The values allowed for   the   operational support
code are meaningful only to future versions of
PDP-11 COBOL and the Merge Utility.   Therefore,
they are not defined here.

11.3 ODL FILE BODY

The ODL file body describes the overlay structure of the COBOL
program.   The body contains the following ODL directive types:

1. .PSECT   defines the name of the code PSECT and makes it
              known to the Task Builder.

2. .NAME   defines the name to be assigned to the overlay
              segment by the Task Builder.

3. .FCTR   describes the contents of the segments.

4. .ROOT   defines the root.

5. .END   informs the Task Builder that the end of the ODL
              file has been reached.

6. ;comments contains comment entries.

The .ROOT and .END directives are not supplied by the COBOL compiler.
They are inserted into the ODL file generated by the Merge Utility.
If you are generating a stand alone ODL file, these directives must be
supplied by you. If the ODL file you are generating is to be used as
input to the Merge Utility, leave these directives out.

Within a compiler-generated ODL file, the directives .PSECT, .NAME,
and .FCTR are generated around the PSECT kernel. If the PSECT name
kernel for a given program is KK, the format of the names generated in
the ODL file is:

Entity   Format of Name

.PSECT $KKMMM

.NAME   KK$MMM

.FCTR   KKMMM$

Each .PSECT defined in the ODL file begins with a $ followed by the
two character kernel ($KK). Each .NAME directive begins with the two
character kernel followed by $ (KK$). Finally, each .FCTR directive
begins with the two-character kernel and ends with a $ (KKMMM$).

11.4 COMPILER-GENERATED ODL FOR COBOL PSECTS

The following sections discuss the ODL directives generated for different types of overlay requirements. The characters NNN when used in examples refer to the three character suffix generated by the compiler for each PSECT. The characters KK refer to the kernel characters that make the PSECT unique to a particular compilation.

11.4.1 ODL Generated for Overlays Containing Only One PSECT

For overlays containing only one PSECT, the following lines are generated:

```
        .PSECT    $KKNNN,GBL,RW,CON,I

        .NAME     KK$NNN,GBL

KKNNN$  .FCTR     *KK$NNN-$KKNNN
```

11.4.2 ODL Generated for Overlays Containing More Than One PSECT

For each overlay that contains more than one PSECT, a .PSECT directive is generated for each PSECT in the overlay. These .PSECT directives are followed by a .NAME and .FCTR directive. Consider the following example.

Example

Two PSECTs, $AA001 and $AA002, are to be placed in the same overlay. The segment-number assigned to the PSECTs is 20. The following ODL directives are generated:

```
        ;DEFINE    PSECT $AA001

        .PSECT     $AA001,GBL,RW,CON,I

        ;DEFINE    PSECT $AA002

        .PSECT     $AA002,GBL,RW,CON,I

        ;DEFINE THE OVERLAY NAME

        .NAME      AA$020,GBL

        ;DEFINE OVERLAY CONTENTS

AA020$:  .FCTR     *AA$020-$AA001-$AA002
```

11.4.3 ODL Generated for All Overlayable PSECTS

All .FCTR directives that describe the overlayable PSECTs must be collapsed into one final .FCTR directive. This directive describes the entire overlayable portion of the object code. The name associated with this .FCTR directive is derived from the two-character kernel assigned to the PSECTs. If the kernel is KK, then the name of the .FCTR directive that describes the entire overlayable part of the object code is KKOVR$.

The following example shows how the KKOVR$ factor is developed for various overlay configurations:

Example 1: All Code Psects Overlay One Another

```
         .PSECT   $AA001,GBL,RW,CON,I
         .NAME    AA$001,GBL
AA001:   .FCTR    *AA$001-$AA001
         ;
         .PSECT   $AA002,GBL,RW,CON,I
         .NAME    AA$002,GBL
AA002$:  .FCTR    *AA$002-$AA002
         ;
         .PSECT   $AA003,GBL,RW,CON,I
         .NAME    AA$003,GBL
AA003$:  .FCTR    *AA$003-$AA003
         ;
         .PSECT   $AA004,GBL,RW,CON,I
         .NAME    AA$004,GBL
AA004$:  .FCTR    *AA$004-$AA004
         ;
         .PSECT   $AA005,GBL,RW,CON,I
         .NAME    AA$005,GBL
AA005$:  .FCTR    *AA$005-$AA005
         ;IN THIS EXAMPLE, ALL PSECTS OVERLAY
         :ONE ANOTHER.
AAOVR$:  .FCTR    (AA001$,AA002$,AA003$,AA004$,AA004$,AA005$)
```

Example 2: Two Code Psects Are in the Same Overlay

```
         .PSECT   $AA001,GBL,RW,CON,I
         ;
         .PSECT   $AA002,GBL,RW,CON,I
         ;
         .NAME    AA$001,GBL
AA001$:  .FCTR    *AA$001-$AA001-$AA002
         ;
         .PSECT   $AA003,GBL,RW,CON,I
         .NAME    AA$003,GBL
AA003$:  .FCTR    *AA$003-$AA003
         ;
         .PSECT   $AA004,GBL,RW,CON,I
         .NAME    AA$004,GBL
AA004$:  .FCTR    *AA$004-$AA004
         ;
         .PSECT   $AA005,GBL,RW,CON,I
         .NAME    AA$005,GBL
AA005$:  .FCTR    *AA$005-$AA005
         ;
AAOVR$:  .FCTR    AA001$,AA003$,AA004$,AA005$
```

Example 3: Two Occurrences of Two Psects in the Same Overlay

```
         ;IN THIS EXAMPLE, PSECTS $AA001 AND $AA002
         ;ARE IN THE SAME OVERLAY. PSECTS $AA003
         ;AND $AA004 ARE IN THE SAME OVERLAY.
         ;PSECT $AA005 IS IN AN OVERLAY ALL BY ITSELF
         ;
         ;PSECT   $AA001,GBL,RW,CON,I
         ;
         ;PSECT   $AA002,GBL,RW,CON,I
         ;
         .NAME    AA$001,GBL
```

```
AA001$:    .FCTR     *AA$001-$AA001-$AA002
           ;
           ;PSECT    $AA003,GBL,RW,CON,I
           ;
           .PSECT    $AA004,GBL,RW,CON,I
           ;
           .NAME     AA$003,GBL
AA003$:    .FCTR     *AA$003-$AA003-$AA004
           ;
           .PSECT    $AA005,GBL,RW,CON,I
           .NAME     AA$005,GBL
AA005$:    .FCTR     *AA$005-$AA005
           ;
AAOVR$:    .FCTR     AA001$,AA003$,AA005$
```

11.5 MERGING STANDARD ODL FILES

To develop an ODL file for a task composed of more than one COBOL
object program, it is necessary to merge the ODL files for each
individual object program into a single ODL file that describes the
overlay requirements for the task.

All of the ODL files to be merged are partial ODL files.   That is,
none of these ODL files can be submitted directly to the Task Builder
to build a task; because, none of the compiler generated ODL files
contain a .ROOT directive.   The .ROOT directive that describes the
task is supplied by the Merge Utility.

Merging COBOL compiler generated ODL files is accomplished by
executing the ODL merge utility. (See Section 2.6, Using The Merge
Utility).

11.6 INCLUDING NON-COBOL PROGRAMS IN A TASK

To use the Merge Utility to include a non-COBOL object module in a
task image, you must:

1. Create a standard COBOL ODL file (use any text editor)

2. Specify this ODL file as input to the Merge Utility.

11.6.1 Creating a Standard COBOL ODL File

A standard COBOL ODL file for a non-COBOL object module contains one
or two directive lines:

1. 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.

2. Main Program ID Line - This line is present only for
    non-COBOL object modules that are main programs as opposed to
    being subprograms. The format of the line is:

        ;COBMAIN

For each invocation of the COBOL ODL Merge Utility, one and only one
main program ODL file can be specified. If no main program ODL file
is specified, the Merge Utility continues to request more input until
a main program ODL file is specified. If more than one main program
ODL file is specified, all but the first is rejected, and appropriate
diagnostic error messages are issued. 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 ODL file to be
hand-generated is:

    ;COBOBJ=START.OBJ
    ;COBMAIN

Example 2

Macro subprogram SUBX.OBJ is to be part of a task image that consists
of several COBOL subprograms and a COBOL main program. The ODL file
to be hand-generated is:

    ;COBOBJ=SUBX.OBJ

11.7 REARRANGING A COMPILER-GENERATED ODL FILE

The ODL file generated by the compiler can be rearranged to modify the
overlay structure of a task. If the ODL file describes a task that
has overlayable segments, one or more of these segments can be
converted into non-overlayable segments by:

1. Modifying the compiler-generated ODL file.

2. Specifying a one-line Task Builder option at task-build time
    for each segment made non-overlayable.

11.7.1 Modifying the Compiler-Generated ODL File

Modifying the compiler generated ODL file requires the following
steps:

1. Each overlayable segment is named in the ODL file by an ODL
    .NAME directive. This .NAME directive must be removed.

2. Each name appearing in a .NAME directive is marked with an *
    and placed as the first element of a .FCTR directive. For
    each .NAME directive removed by step 1, this .FCTR directive
    must be removed.

3. All references to the name of the .FCTR directive removed in step 2 must be removed from the ODL file.

4. All PSECTs referenced in the .FCTR directive that was removed in step 3, must be removed from the ODL file.

Example

The task image contains three overlayable segments, C$$010, C$$015, and C$$020. Segment C$$020 is to be forced into the root. Figure 11-1 contains a listing of the merged ODL file.

```
;MERGED ODL FILE CREATED ON 26-JAN-77 AT 10:50:00
;COBOL STANDARD ODL FILE GENERATED ON: 26-JAN-77    10:48:37
;COBOBJ=TEST1.OBJ
;COBKER=C$
;COBMAIN
           .NAME C$$010,GBL
           .PSECT $C$003,GBL,I,RW,CON
C$010$:    .FCTR *C$$010-$C$003
           .NAME C$$015,GBL
           .PSECT $C$004,GBL,I,RW,CON
C$015$:    .FCTR *C$$015-$C$004
           .NAME C$$020,GBL
           .PSECT $C$005,GBL,I,RW,CON
C$020$:    .FCTR *C$$020-$C$005
C$OVR$:    .FCTR         C$010$,C$015$,C$020$
CBOBJ$:    .FCTR TEST1.OBJ
CBOVR$:    .FCTR C$OVR$
CBOTS$:    .FCTR [320,13]COBLIB/LB
RMS$:      .FCTR   [1,1]RMSLIB/LB
OBJRT$:    .FCTR CBOBJ$-CBOTS$-RMS$
           .ROOT OBJRT$-(CBOVR$)
           .END
```

Figure 11-1 Merged ODL File Listing

To force segment C$$020 into the root, the merged ODL file must be modified as follows:

1. The .NAME directive referencing C$$020 must be removed.

2. The .FCTR directive containing *C$$020 must be removed.

3. All references to the PSECTs in the removed .FCTR directive must be removed.

Figure 11-2 contains the ODL listing after the modifications have been made.

```
;MERGED ODL FILE CREATED ON 26-JAN-77 AT 10:55:22
;COBOL STANDARD ODL FILE GENERATED ON: 26-JAN-77    10:48:37
;COBOBJ=TEST1.OBJ
;COBKER=C$
;COBMAIN
            .NAME C$$010,GBL
            .PSECT $C$003,GBL,I,RW,CON
C$010$:     .FCTR *C$$010-$C$003
            .NAME C$$015,GBL
            .PSECT $C$004,GBL,I,RW,CON
C$015$:     .FCTR *C$$015-$C$004
C$OVR$:     .FCTR            C$010$,C$015$
CBOBJ$:     .FCTR TEST1.OBJ
CBOVR$:     .FCTR C$OVR$
CBOTS$:     .FCTR [1,1]COBLIB/LB
RMS$:       .FCTR   [1,1]RMSLIB/LB
OBJRT$:     .FCTR CBOBJ$-CBOTS$-RMS$
            .ROOT OBJRT$-(CBOVR$)
            .END
```

Figure 11-2 Modified ODL File

11.7.2 Specifying Task Builder Options

For each overlayable segment made non-overlayable, a GBLDEF Task Builder option must be specified at task-build time. The format of the option is:

    GBLDEF=KK$MMM:O

Where:

> KK$MMM is the name of the segment that is being made non-overlayable. (This is the name in the .NAME ODL directive that was deleted when the ODL file was modified).

Consider the following example.

Example

To make the overlayable segment (C$$020) described in the example in Section 11.7 non-overlayable, enter the following in response to the Task Builder ENTER OPTIONS prompt:

    GBLDEF=C$$020:0 (RET)

Figure 11-3 shows the overlay description of the task image before and after segment C$$020 was made non-overlayable.

```
BEFORE

TEST1.TSK;1     MEMORY ALLOCATION MAP   TKB M27
                          26-JAN-77    10:51



PARTITION NAME  : GEN
IDENTIFICATION  : 026108
TASK   UIC      : [320,4]
STACK     LIMITS: 000176 001175 001000 00512.
PRG XFR ADDRESS: 022514
TOTAL ADDRESS WINDOWS: 1.
TASK   IMAGE   SIZE  : 6880. WORDS
TASK ADDRESS LIMITS: 000000 032657



TEST1.TSK;1 OVERLAY DESCRIPTION:


BASE       TOP          LENGTH


000000     032507     032510  13640.        TEST1
032510     032607     000100  00064.             C$$010    3 overlayable
032510     032657     000150  00104.             C$$015
032510     032617     000110  00072.             C$$020    segments
```

```
AFTER


TEST2.TSK;2     MEMORY ALLOCATION MAP   TKB M27
                          26-JAN-77    10:57



PARTITION NAME  : GEN
IDENTIFICATION  : 026108
TASK   UIC      : [320,4]
STACK     LIMITS: 000176 001175 001000 00512.
PRG XFR ADDRESS: 022514
TOTAL ADDRESS WINDOWS: 1.
TASK   IMAGE   SIZE  : 6912. WORDS
TASK ADDRESS LIMITS: 000000 032743



TEST2.TSK;2 OVERLAY DESCRIPTION:


BASE       TOP          LENGTH


000000     032573     032574  13692.        TEST1
032574     032673     000100  00064.             C$$010    2 overlayable
032574     032743     000150  00104.             C$$015    segments
```

Figure 11-3 Overlay Description Map Before and After Modification

CHAPTER 12

ERROR MESSAGES

12.1 COMPILER SYSTEM ERRORS

The PDP-11 COBOL compiler is a complex system program consisting of many program overlays that manipulate numerous data structures. Throughout the compiler, consistency checks are performed on program flow and the contents of data fields. If the compiler detects an inconsistency, it types a message on the console and terminates the compilation. A system error message has the following format:

    SYSTEM ERROR NNNNN

where NNNNN is a number used by the DEC COBOL developers to determine the probable cause of the error. When a system error occurs, the compiler's input file is closed and all output files (object, list, and ODL) are closed and deleted.

In the event of a PDP-11 COBOL compiler system error, contact your DEC Software Support Specialist immediately. See Appendix G, Compiler System errors.

12.2 DIAGNOSTIC ERROR MESSAGES

This chapter contains a numerical listing of the diagnostic messages generated by the PDP-11 COBOL compiler. The compiler generates these messages whenever it detects an error in the source program. In general, a source error detected by the compiler results in the associated diagnostic message being embedded within the source program listing. That is, when an error is detected in the source program, the compiler prints the diagnostic message either before or after the erroneous source program line. There are two exceptions to the general concept of "embedded diagnostics":

1. There may be diagnostic messages listed after the last entry in the Data Division and before the Procedure Division header. These are diagnostics which logically can not be issued until the entire Data Division text is processed.

2. There may be diagnostic messages listed after the last line of the Procedure Division. These are diagnostics which logically can not be issued until the entire Procedure Division text is processed.

In addition to the error message number and message text, the display contains a source line number, which identifies the error line, and an alphabetic code (discussed below) which informs the user of the seriousness of the error. The information within a diagnostic message line is displayed (from left to right) in the following order:

1. Alphabetic code,

2. Source line number,

3. Numerical error number,

4. Text of the diagnostic message.

For convenience, the alphabetic code is left-justified in the listing so the user merely scans the listing to identify any diagnostic message issued during compilation. Again, for the user's convenience, a summary of the number of errors detected during the compilation is given at the end of the source listings. If no errors are detected during the compilation, the compiler prints "NO ERRORS" at the end of the source listing.

The following illustration shows a typical diagnostic message and the manner in which it appears on the source listing:

COBOL 3.00 SRC:MAP.CBL;0   05-JAN-77   18:49:10   PAGE 003

```
00096          MOVE 72.5 TO N2
00097          IF N2 NOT = T2 DISPLAY "? #10".
00098     *
00099          MOVE 3250 TO N3.
```

I 00099 372 POSSIBLE LOW ORDER RECEIVING FIELD TRUNCATION.

```
00100          IF N3 NOT = T3 DISPLAY "? #11".
00101     *
00102          MOVE -432 TO N4.
00103          IF N4 NOT = T4 DISPLAY "? #12".
00104     *
```

In the example, the diagnostic message is immediately identified by the appearance of the left-justified alphabetic code I. The alphabetic code indicates that the message is an I-type (informational) diagnostic; the diagnostic is issued for source line number 99; the error number is 372; and the text of the message is POSSIBLE LOW ORDER RECEIVING FIELD TRUNCATION. Note that the diagnostic message line, in this example, appears after the source line for which it was issued.

The error messages, used in conjunction with this chapter, provide the user with an important debugging tool. This chapter contains information necessary for interpreting the messages. It explains what caused the error and how the compiler handled the error.

Since different errors cause varying degrees of problems for the compiler (some do not affect the compilation at all, while others may be so critical that they cause an abort of the compilation), the PDP-11 COBOL compiler provides four general types (or severity levels) of diagnostic messages. Alphabetic codes (I, W, F, and A) identify these error levels. When it detects an error in the source program, the compiler attempts to recover from the error and continue to compile the program. This recovery action may force the compiler to make an assumption about the source program. The four levels of diagnostic messages are categorized according to the likelihood that the result of the compiler's assumption will be an object program that runs as originally intended by the programmer.

The following list explains the purpose of and the compiler's action for each of the four message levels:

I   (Informational) Informative diagnostic. The purpose of such a diagnostic is to convey information to the user in an observational or advisory capacity. The compiler's error recovery (if any is required) is almost certain to be that desired by the user.

W   (Warning) Warning diagnostic. The purpose of this type of message is to warn the user that something is wrong with the associated source statement, but that the compiler can take corrective action on the source element in error. The compiler's recovery action may not be that desired by the user, but the statement, as corrected by the compiler, will be executable.

F   (Fatal) Fatal diagnostic. The purpose of such a diagnostic is to indicate to the user that something is fatally wrong with the indicated source statement. By fatal, the compiler means it cannot generate the object code required for the functionality the programmer coded in the erroneous source statement. The compiler's error recovery action will probably leave out a portion of the source program. In general, the compiler will not produce an object program for COBOL source programs which have F-type errors in them. However, the user can force the compiler to generate an object program by specifying the /ACC:2 switch in the command string input to the compiler prior to compilation (See Section 5.2.3, Compiler Switches for a detailed explanation of the /ACC:n switch.) The /ACC:2 switch instructs the compiler to generate an object program, even if the source program contains F-type errors. In this case, when an F-type error is detected in the Procedure Division, the compiler generates special error trap object code in place of the incorrect source statement. When the object program is executed and the special error trap code is encountered, the software displays the following message on the console and aborts the program execution:

     FATAL ERROR ON SOURCE LINE XXXXX

     where XXXXX is the source line number for which an F-type diagnostic was issued during compilation. For F-type diagnostics issued in the Identification, Environment, and Data Divisions, no special error trap coding is generated since, in general, executable code is not generated for these divisions. However, the fact that F-type diagnostics are issued for these divisions can have a definite effect on the behavior of the execution of the object program.

                              WARNING

          When the user specifies the /ACC:2 switch, the user
          is formally acknowledging to the software a
          willingness to let the program go into execution
          even though it may have fatal errors in it. Because
          the source program has very severe errors in it, the
          behavior of the associated object program is, in
          general, unpredictable. In certain cases, such as a
          COBOL program with files opened in I-O mode, letting
          the program with F-type errors go into execution
          could be disastrous. Thus, the /ACC:2 switch should
          be used with caution. The facility is provided as
          an extra debugging option. It can be useful in
          shortening the compile-debug cycle, particularly if

applied to large COBOL programs which take
considerable compilation time. The point is that
the user should use the /ACC:2 facility wisely and
discretely.

A   (Abortive) Abortive diagnostic. The purpose of this type of
     diagnostic is to inform the user that the compiler must
     abort compilation. The compiler's error recovery is not
     possible: it can make no valid assumptions and has no
     choice but to abort the compilation.

     Appendix H contains the PDP-11 COBOL compiler diagnostic
     error messages arranged in numerical order for easy
     reference.

12.3 RUNTIME FILE I/O ERROR PROCEDURES

When an error condition occurs during I/O operations, the following
procedure is used:

1.   If the file status key for the file is present, it is set to
     the appropriate code for the error condition. (See Table
     12-1 for sequential file status keys, or Table 12-2 for
     relative and indexed file status keys.)

2.   If an AT END or INVALID KEY imperative condition is specified
     for the I/O operation, the path indicated by the imperative
     statement is taken. The file system performs no other
     processing in the file for the current statement. The USE
     procedure, if one is declared for the file, is not performed.

3.   If no AT END or INVALID KEY imperative condition is specified
     for the I/O operation and a USE procedure is declared for the
     file, the USE procedure section is performed, and then
     control is returned to the program. The file system performs
     no further processing for this file.

     If no USE procedure is declared for the file, a fatal error
     condition exists; the OTS aborts the program and displays
     the following I/O error message:

               "CBL -- 37: FILE: NN... NO USE PROCEDURE FOR
                         I/O ERROR"
               "CBL -- RECORD MANAGEMENT SERVICES - XX"

     NN   represents the name of the file:

     XX   represents the Record Management Services error code.
          (See Appendix I for these error codes.)

ERROR MESSAGES

The following tables show various error numbers and error codes that identify error conditions and messages. The error codes in Tables 12-1 and 12-2 are accessible to the user's program through the declaration and use of the FILE STATUS key in the program. The error codes in Appendix I are not returned to the user's program but represent error conditions detected by Record Management Services.

The error message numbers in Appendix I are merely identifying numbers for the messages and appear at the user terminal in the following form:

"CBL --nn: message ... "

nn is the message number.

Table 12-1 and 12-2 contain status key codes. The left-hand digit of the status key code is status key 1, and the right-hand digit is status key 2.

Table 12-1
Sequential I/O File Status Key Values (ASCII)

Status Key Code	Meaning
00	No further information (successful)
10	End-Of-File indicator detected
30	Permanent error
34	Permanent error (boundary error on WRITE statement)
91	File locked by another task
93	REWRITE attempted without prior READ
94	Improper operation attempted
95	Allocation failure on OPEN (no file space on device)
96	No buffer space (program tried to open a file that is sharing buffer space (SAME AREA) with another file)
97	No such file (the file named in an OPEN statement was not found)
98	CLOSE error (error discovered while in the process of closing the file)

ERROR MESSAGES

Table 12-2
Relative And Indexed I/O File Status Key Values (ASCII)

Status Key Code	Meaning
00	No further information (successful)
02	Duplicate alternate record key values were successfully created during the execution of a WRITE or REWRITE statement
10	End-Of-File indicator detected
21	Sequence error on primary key during the execution of a WRITE or REWRITE statement
22	Duplicate key error
23	No such record error
24	Boundary error on WRITE statement
30	Permanent error
91	File locked by another task
92	Record locked by another task
93	REWRITE or DELETE attempted without prior READ
94	Improper operation attempted
95	Allocation failure (no file space on device)
96	No buffer space (program tried to open a file that is sharing buffer space (SAME AREA) with another file)
97	No such file (the file named in an OPEN statement was not found)
98	CLOSE error (error discovered while in the process of closing the file)

12.4 RUN-TIME ERROR MESSAGES

Appendix J contains a list of the COBOL Object Time System (OTS) error messages. Wherever it can, the COBOL OTS will list auxiliary information along with the error message. This auxiliary information is defined in Section 12.4.1.

12.4.1 OTS Auxiliary Error Message Information

Following each OTS error message, the OTS will attempt to display additional clarifying information. This information is intended to direct you to the exact source line statement causing the error. The auxiliary information has the following format:

PROGRAM-ID AAAAAA , IDENT: BBBBBB

IN PSECT: CCCCCC

AT OFFSET: DDDDDD

Where:

AAAAAA   is the first six characters of the PROGRAM-ID specified
           in the source program.

BBBBBB   is the value appearing in the IDENT field of compiler
           listing.

CCCCCC   is the name of the procedural code PSECT containing the
           error.

DDDDDD   is the octal byte offset (within PSECT CCCCCC) at which
           the error occurred.

If the statement in error is a PERFORM, the nested PERFORM stack, containing the source line location of every PERFORM statement encountered thus far, is displayed (see Example 1). If an error is detected while a chain of nested CALL statements is being processed, auxiliary information is displayed for each element in the chain (see Example 2).

To take full advantage of this auxiliary information, you must have compiled the source program with the /MAP and /OBJ switches specified. Using the PSECT name (CCCCCC) and octal byte offset (DDDDDD), in conjunction with the Procedure Name Map, you can identify the two source procedure names that bracket the location of the error. Also, using the octal byte offset (DDDDDD) you can (via the /OBJ output listing) identify the specific verb causing the error. Consider the following examples:

Example 1

Figure 12-1 contains the listings generated for the COBOL program used in this example. Execution of the program depicted in Figure 12-1 will yield the following results:

RUN DIAG3

```
CBL -- 25:  ILLEGAL NESTED PERFORM AT SOURCE LINE 16
       PROGRAM ID:  DIAG3 , IDENT:  038105
       IN PSECT:  $ZZ001
       AT OFFSET:  000074
       NESTED PERFORM SOURCE LINE NUMBERS:
            00014
            00015

CBL -- 15:  STOP RUN

Ready
```

The PROGRAM-ID and IDENT line refer to the corresponding lines is the compiler listing (DIAG3 and 038105 in this example). The IN PSECT line identifies the exact PSECT containing the error ($ZZ001 in this example). See the Procedure Name Map in Figure 12-1. The AT OFFSET line identifies the octal byte offset within the PSECT at which the statement in error exists (000074 in this example). See the /OBJ compiler listing in Figure 12-1. Finally, the last three lines identify the source line location of every PERFORM statement encountered thus far in the program.

ERROR MESSAGES

SAMPLE /OBJ LISTING

```
COBOL  3.00 SRC:DIAG3.CBL;0              07-FEB-77  10:34:03 PAGE 001

CMD:DIAG3,DIAG3/MAP/OBJ/KE:ZZ=DIAG3
IDENT:  038105
```

	00001	IDENTIFICATION DIVISION.
	00002	PROGRAM-ID. DIAG3.
	00003	*
	00004	* INVOKE "?ILLEGAL NESTED PERFORM AT LINE XXXXX"
	00005	*
	00006	ENVIRONMENT DIVISION.
	00007	CONFIGURATION SECTION.
	00008	SOURCE-COMPUTER. PDP-11.
	00009	OBJECT-COMPUTER. PDP-11.
	00010	DATA DIVISION.
	00011	WORKING-STORAGE SECTION.
	00012	PROCEDURE DIVISION.
	00013	S0 SECTION.
PERFORM : 001 000024		
	00014	P0. PERFORM P1 THRU P5.
PERFORM : 001 000050		
	00015	P1. PERFORM P2 THRU P4.
PERFORM : 001 000074		
	00016	P2. PERFORM P3 THRU P4.
	00017	P3.
	00018	P4.
	00019	P5.

SAMPLE PROCEDURE NAME MAP
```
COBOL  3.00 SRC:DIAG3.CBL;0              07-FEB-77  10:34:03 PAGE 002
```

PROCEDURE NAME MAP

NAME	SOURCE LINE	PSECT	OFFSET	SEG	SECT	PARA
S0	00013	$ZZ001	000024	00	S	
P0	00014	$ZZ001	000024	00		P
P1	00015	$ZZ001	000050	00		P
P2	00016	$ZZ001	000074	00		P
P3	00017	$ZZ001	000120	00		P
P4	00018	$ZZ001	000126	00		P
P5	00019	$ZZ001	000134	00		P

Figure 12-1
Sample Listing of Program Used in Example-1

12-8

Example 2

Figure 12-2 contains the listings generated for the COBOL programs used in this example.   Execution of the programs depicted in Figure 12-2 will yield the following results:

       RUN TEST
       BEGIN MAIN PROGRAM
       BEGIN SUB1 SUBPROGRAM
       BEGIN SUB2 SUBPROGRAM

       CBL -- 13: NULL ALTERABLE GO TO
               PROGRAM ID: SUB2 , IDENT: 080130
               IN PSECT: $CC001
               AT OFFSET: 000062
               LISTING OF NESTED ENVIRONMENTS:
               PROGRAM ID: SUB1 , IDENT: 080130
               AT OFFSET: 000054
               PROGRAM ID: MAIN , IDENT: 080129
               AT OFFSET: 000042

       CBL -- 15: STOP RUN

       Ready

As in the previous example, the PROGRAM-ID and IDENT lines refer to the corresponding lines in the compiler listing; the IN PSECT line identifies the exact PSECT containing the error; and the AT OFFSET line identifies the octal byte offset. Note also, that this information is repeated for every program within the chain of subprograms comprising the task.

```
SAMPLE /OBJ LISTING (Main Program)


     COBOL  3.00 SRC:MAIN.CBL;0            21-MAR-77  12:59:08 PAGE 001

CMD:MAIN,MAIN/KE:AA/OBJ/MAP=MAIN
IDENT: 080129



                                 00001          IDENTIFICATION DIVISION.
                                 00002          PROGRAM-ID. MAIN.
                                 00003          ENVIRONMENT DIVISION.
                                 00004          CONFIGURATION SECTION.
                                 00005          SOURCE-COMPUTER. PDP-11.
                                 00006          OBJECT-COMPUTER. PDP-11.
                                 00007          DATA DIVISION.
                                 00008          WORKING-STORAGE SECTION.
                                 00009          77 A PIC 99.
                                 00010          77 B PIC 99.
                                 00011          77 C PIC 99.
                                 00012          PROCEDURE DIVISION.
                                 00013          S0 SECTION.
                                 00014          P0.
     DISPLAY  :  01  000024
                                 00015             DISPLAY "BEGIN MAIN PROGRAM".
     CALL     :  01  000042
                                 00016             CALL "SUB1" USING A B C .
     DISPLAY  :  01  000060
                                 00017             DISPLAY "ENDING MAIN PROGRAM".
     STOP     :  01  000076
                                 00018             STOP RUN.
```

```
SAMPLE PROCEDURE MAP (Main Program)


     COBOL  3.00 SRC:MAIN.CBL;0            21-MAR-77  12:59:08 PAGE 003


                         PROCEDURE NAME MAP

     NAME                        SOURCE  PSECT   OFFSET  SEG  SECT   PARA
                                 LINE

     S0                          00013   $AA001  000024  00   S
     P0                          00014   $AA001  000024  00          P
```

Figure 12-2 Sample Listing of Program Used in Example-2

```
SAMPLE /OBJ LISTING (Subprogram)


      COBOL  3.00 SRC:SUB1.CBL;0              21-MAR-77  13:00:22 PAGE 001

CMD:SUB1,SUB1/KE:BB/OBJ/MAP=SUB1
IDENT: 080130



                                00001          IDENTIFICATION DIVISION.
                                00002          PROGRAM-ID. SUB1.
                                00003          ENVIRONMENT DIVISION.
                                00004          CONFIGURATION SECTION.
                                00005          SOURCE-COMPUTER. PDP-11.
                                00006          OBJECT-COMPUTER. PDP-11.
                                00007          DATA DIVISION.
                                00008          WORKING-STORAGE SECTION.
                                00009          LINKAGE SECTION.
                                00010          77 D PIC 99.
                                00011          77 E PIC 99.
                                00012          77 F PIC 99.
                                00013          PROCEDURE DIVISION USING D E F .
                                00014          S1 SECTION.
                                00015          P1.
     DISPLAY  :  01  000024
                                00016              DISPLAY "BEGIN SUB1 SUBPROGRAM".
     SUBTRACT :  01  000042
                                00017              SUBTRACT D FROM E GIVING F .
     CALL     :  01  000054
                                00018              CALL "SUB2" USING D E F .
     DISPLAY  :  01  000072
                                00019              DISPLAY "EXITING SUB1 SUBPROGRAM".
     STOP     :  01  000110
                                00020              STOP RUN .
```

```
SAMPLE PROCEDURE MAP (Subprogram)


      COBOL  3.00 SRC:SUB1.CBL;0              21-MAR-77  13:00:22 PAGE 003


                        PROCEDURE NAME MAP

     NAME                        SOURCE  PSECT    OFFSET  SEG  SECT   PARA
                                 LINE

     S1                          00014   $BB001   000024  00   S
     P1                          00015   $BB001   000024  00          P
```

Figure 12-2(Cont.) Sample Listing of Program Used in Example-2

APPENDIX A

THE COBOL FORMATS

COBOL NOTATION USED IN FORMATS

- Underlined upper-case words (key words) - required words;

- Upper-case words (not underlined) - optional words;

- Lower-case words - generic terms, must be supplied by the user;

- Brackets [] - enclosed portion is optional; if several enclosed words are vertically stacked, only one of them may be used;

- Braces {} - a selection must be made from the vertical stack of enclosed words;

- Ellipsis ... - the position at which repetition may occur;

- Comma and semicolon - optional punctuation;

- Period - required where shown in the formats.

NOTE: Shaded items represent PDP-11 COBOL extensions to the ANS-74 list of COBOL formats.

IDENTIFICATION DIVISION.

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

ENVIRONMENT DIVISION.

CONFIGURATION SECTION.

SOURCE-COMPUTER.   PDP-11

OBJECT-COMPUTER.   PDP-11 $\begin{bmatrix} \text{MEMORY SIZE integer} & \begin{Bmatrix} \underline{\text{WORDS}} \\ \underline{\text{CHARACTERS}} \\ \underline{\text{MODULES}} \end{Bmatrix} \end{bmatrix}$

[PROGRAM COLLATING SEQUENCE IS alphabet-name]
[SEGMENT-LIMIT IS segment-number].

[SPECIAL-NAMES.

[CARD-READER IS mnemonic-name-1]
[CONSOLE IS mnemonic-name-2]
[LINE-PRINTER IS mnemonic-name-3]
[PAPER-TAPE-PUNCH IS mnemonic-name-4]
[PAPER-TAPE-READER IS mnemonic-name-5]

$$\left[\text{SWITCH integer-1} \quad \left\{ \begin{array}{l} \text{ON STATUS IS condition-name-1} \\ \text{OFF STATUS IS condition-name-2} \end{array} \right. \right.$$

$$\left. \left. \begin{array}{l} \text{[OFF STATUS IS condition-name-2]} \\ \text{[ON STATUS IS condition-name-1]} \end{array} \right\} \right] \quad \ldots$$

$$\left[\text{Alphabet-name IS} \quad \left\{ \begin{array}{l} \text{NATIVE} \\ \text{STANDARD-1} \end{array} \right\} \right]$$

[CURRENCY SIGN IS literal-1]
[DECIMAL-POINT IS COMMA.]]

[INPUT-OUTPUT SECTION.

FILE-CONTROL. {file-control-entry}... .,

Format 1:

SELECT [OPTIONAL] file-name

    ASSIGN TO literal-1

$$\left[; \text{RESERVE integer-1} \quad \left[\begin{array}{l} \text{AREA} \\ \text{AREAS} \end{array} \right] \right]$$

    [; ORGANIZATION IS SEQUENTIAL]
    [; ACCESS MODE IS SEQUENTIAL]
    [; FILE STATUS IS data-name-1] .

Format 2:

SELECT file-name

    ASSIGN TO literal-1

$$\left[; \text{RESERVE integer-1} \quad \left[\begin{array}{l} \text{AREA} \\ \text{AREAS} \end{array} \right] \right]$$

    ; ORGANIZATION IS RELATIVE

$$\left[; \text{ACCESS MODE IS} \left\{ \begin{array}{l} \text{SEQUENTIAL} \\ \left\{ \begin{array}{l} \text{RANDOM} \\ \text{DYNAMIC} \end{array} \right\} \end{array} \right. \left. \begin{array}{l} \text{[, RELATIVE KEY IS data-name-1]} \\ \text{RELATIVE KEY IS data-name-1} \end{array} \right\} \right]$$

    [; FILE STATUS IS data-name-2] .

Format 3:

<u>SELECT</u> file-name

    <u>ASSIGN</u> TO literal-1

    $\left[; \underline{\text{RESERVE}} \text{ integer-1} \quad \begin{bmatrix} \text{AREA} \\ \text{AREAS} \end{bmatrix} \right]$

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

    $\left[; \underline{\text{ACCESS}} \text{ MODE IS} \quad \begin{Bmatrix} \underline{\text{SEQUENTIAL}} \\ \underline{\text{RANDOM}} \\ \underline{\text{DYNAMIC}} \end{Bmatrix} \right]$

    ; <u>RECORD</u> KEY IS data-name-1
    [; <u>ALTERNATE</u> <u>RECORD</u> KEY IS data-name-2 [WITH <u>DUPLICATES</u>]]...
    [; FILE <u>STATUS</u> IS data-name-3] .

[I-O-CONTROL.
    [<u>SAME</u> [<u>RECORD</u>] AREA FOR file-name-1 {file-name-2}...]...
    [<u>MULTIPLE</u> <u>FILE</u> TAPE CONTAINS file-name-3 [<u>POSITION</u> integer-1]
      [file-name-4 [<u>POSITION</u> integer-2]...]...

    $\left[\underline{\text{APPLY}} \quad \underline{\text{PRINT-CONTROL}} \quad \underline{\text{ON}} \text{ file-name-5 [file-name-6]...} \right] ... \Big]$

<u>DATA</u> <u>DIVISION</u>.

[<u>FILE</u> <u>SECTION</u>.
[<u>FD</u> file-name

    $\left[\underline{\text{BLOCK}} \text{ CONTAINS [integer-1 } \underline{\text{TO}}\text{] integer-2} \quad \begin{Bmatrix} \underline{\text{RECORDS}} \\ \underline{\text{CHARACTERS}} \end{Bmatrix} \right]$
    [<u>RECORD</u> CONTAINS [integer-3 <u>TO</u>] integer-4 CHARACTERS]
    $\underline{\text{LABEL}} \begin{Bmatrix} \underline{\text{RECORD}} \text{ IS} \\ \underline{\text{RECORDS}} \text{ ARE} \end{Bmatrix} \begin{Bmatrix} \underline{\text{STANDARD}} \\ \underline{\text{OMITTED}} \end{Bmatrix}$
    $\left[\underline{\text{VALUE}} \underline{\text{ OF}} \underline{\text{ ID}} \text{ IS} \begin{Bmatrix} \text{data-name-1} \\ \text{literal-1} \end{Bmatrix} \right]$

    $\left[\underline{\text{DATA}} \begin{Bmatrix} \underline{\text{RECORD}} \text{ IS} \\ \underline{\text{RECORDS}} \text{ ARE} \end{Bmatrix} \text{ data-name-3} \quad \text{[data-name-4]} ... \right] ...$
    $\left[\underline{\text{LINAGE}} \text{ IS} \begin{Bmatrix} \text{data-name-5} \\ \text{integer-5} \end{Bmatrix} \text{LINES} \left[\text{WITH } \underline{\text{FOOTING}} \text{ AT} \begin{Bmatrix} \text{data-name-6} \\ \text{integer-6} \end{Bmatrix} \right] \right.$

    $\left. \left[\text{LINES AT } \underline{\text{TOP}} \begin{Bmatrix} \text{data-name-7} \\ \text{integer-7} \end{Bmatrix} \right] \left[\text{LINES AT } \underline{\text{BOTTOM}} \begin{Bmatrix} \text{data-name-8} \\ \text{integer-8} \end{Bmatrix} \right] \right]$
    [<u>CODE-SET</u> IS alphabet-name].
[record-description-entry]...]...]
[<u>WORKING-STORAGE</u> <u>SECTION</u>.
    $\begin{bmatrix} \text{77-level-description-entry} \\ \text{record-description-entry} \end{bmatrix} \quad ...]$
[<u>LINKAGE</u> <u>SECTION</u>.
    $\begin{bmatrix} \text{77-level-description-entry} \\ \text{record-description-entry} \end{bmatrix}...]$

Data description entry:

Format 1:

level-number $\left\{\begin{array}{l}\text{data-name-1}\\\underline{\text{FILLER}}\end{array}\right\}$

$\qquad$ [REDEFINES data-name-2]

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

$\left[[\underline{\text{USAGE}}\text{ IS}]\left\{\begin{array}{l}\underline{\text{COMPUTATIONAL}}\\\underline{\text{COMP}}\\\underline{\text{DISPLAY}}\\\underline{\text{DISPLAY-6}}\\\underline{\text{DISPLAY-7}}\\\underline{\text{INDEX}}\end{array}\right\}\right]$

$\left[\quad[\underline{\text{SIGN}}\text{ IS}]\quad\left\{\begin{array}{l}\underline{\text{LEADING}}\\\underline{\text{TRAILING}}\end{array}\right\}\qquad[\underline{\text{SEPARATE}}\text{ CHARACTER}]\right]$

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

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

$\qquad$ [BLANK WHEN ZERO]

$\qquad$ [VALUE IS literal]

$\left[\underline{\text{OCCURS}}\quad\left\{\begin{array}{l}\text{integer-1}\ \underline{\text{TO}}\ \text{integer-2 TIMES}\ \underline{\text{DEPENDING}}\ \text{ON data-name-3}\\\text{integer-2 TIMES}\end{array}\right\}\right.$

$\qquad\left[\left\{\begin{array}{l}\underline{\text{ASCENDING}}\\\underline{\text{DESCENDING}}\end{array}\right\}\text{ KEY IS data-name-4}\qquad[\text{data-name-5}]...\right]...$

$\qquad\left.[\underline{\text{INDEXED}}\text{ BY index-name-1}\qquad[\text{index-name-2}]...]\right]$.

Format 2:

66 data-name-1 $\underline{\text{RENAMES}}$ data-name-2

$\left[\left\{\begin{array}{l}\underline{\text{THROUGH}}\\\underline{\text{THRU}}\end{array}\right\}\qquad\text{data-name-3}\right]$

Format 3:

88 condition-name $\left\{\begin{array}{l}\underline{\text{VALUE}}\text{ IS}\\\underline{\text{VALUES}}\text{ ARE}\end{array}\right\}$ literal-1 $\left[\left\{\begin{array}{l}\underline{\text{THROUGH}}\\\underline{\text{THRU}}\end{array}\right\}\text{ literal-2}\right]$

$\left[\text{literal-3}\quad\left[\left\{\begin{array}{l}\underline{\text{THROUGH}}\\\underline{\text{THRU}}\end{array}\right\}\quad\text{literal-4}\right]\right]$ ... .

$\underline{\text{PROCEDURE}}$ $\underline{\text{DIVISION}}$ [$\underline{\text{USING}}$ [data-name-1][,data-name-2] ...].

Format 1:

[DECLARATIVES.
{section-name $\underline{\text{SECTION}}$ [segment-number] . declarative-sentence
[paragraph-name.[sentence]...]...}...
$\underline{\text{END}}$ $\underline{\text{DECLARATIVES}}$.]
{section-name $\underline{\text{SECTION}}$ [segment-number].
[paragraph-name.[sentence]...]...}...

Format 2:

{paragraph-name.[sentence]...}...

THE COBOL FORMATS

STATEMENTS

```
ACCEPT identifier [FROM mnemonic-name]
                           ( DATE )
ACCEPT identifier FROM     { DAY  }
                           ( TIME )


ADD    {identifier-1}   [identifier-2]   ...  TO identifier-m [ROUNDED]
       {literal-1   }   [literal-2   ]

       [identifier-n[ROUNDED]]...[ON SIZE ERROR imperative-statement]

ADD    {identifier-1}   {identifier-2}   [identifier-3]  ...
       {literal-1   }   {literal-2   }   [literal-3   ]
     GIVING identifier-m [ROUNDED]   [identifier-n [ROUNDED]]   ...
               [ON SIZE ERROR imperative-statement]


ADD    { CORRESPONDING }                    identifier-1 TO identifier-2 [ROUNDED]
       { CORR          }
          [ON SIZE ERROR imperative-statement]


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


CALL literal-1
     [USING data-name-1 [,data-name-2]...]

                          [ ( REEL )  [WITH NO REWIND] ]
                          [ { UNIT }  [FOR REMOVAL   ] ]
CLOSE file-name-1         [                            ]
                          [      WITH { NO REWIND }    ]
                          [           { LOCK      }    ]


     [                 [ ( REEL )  [WITH NO REWIND] ] ]
     [                 [ { UNIT }  [FOR REMOVAL   ] ] ]
     [ file-name-2     [                            ] ]
     [                 [      WITH { NO REWIND }    ] ]
     [                 [           { LOCK      }    ] ]


COMPUTE identifier-1 [ROUNDED] [identifier-2 [ROUNDED]] ...
     = arithmetic-expression [ON SIZE ERROR imperative-statement]


DELETE file-name RECORD      [INVALID KEY imperative-statement]
DISPLAY  {identifier-1}   [identifier-2]   ...
         {literal-1   }   [literal-2   ]
     [UPON mnemonic-name] [WITH NO ADVANCING]


DIVIDE   {identifier-1}      INTO identifier-2 [ROUNDED]
         {literal-1   }
             [identifier-3[ROUNDED]] ... [ON SIZE ERROR imperative-statement]


DIVIDE   {identifier-1}   INTO   {identifier-2}   GIVING identifier-3[ROUNDED]
         {literal-1   }          {literal-2   }
             [identifier-4[ROUNDED]]...[ON SIZE ERROR imperative-statement]


DIVIDE   {identifier-1}   BY   {identifier-2}   GIVING identifier-3[ROUNDED]
         {literal-1   }        {literal-2   }
             [identifier-4[ROUNDED]]...[ON SIZE ERROR imperative-statement]


DIVIDE   {identifier-1}   INTO   {identifier-2}   GIVING identifier-3[ROUNDED]
         {literal-1   }          {literal-2   }
         REMAINDER identifier-4[ON SIZE ERROR imperative-statement]


DIVIDE   {identifier-1}   BY   {identifier-2}   GIVING identifier-3[ROUNDED]
         {literal-1   }        {literal-2   }
         REMAINDER identifier-4[ON SIZE ERROR imperative-statement]
```

<u>EXIT</u> [PROGRAM]

<u>GO</u> TO [procedure-name-1]
<u>GO</u> TO procedure-name-1 [procedure-name-2]...procedure-name-n <u>DEPENDING</u> ON identifier

<u>IF</u> condition $\left\{\begin{array}{l}\text{statement-1}\\\underline{\text{NEXT SENTENCE}}\end{array}\right\}$ $\left\{\begin{array}{l}\underline{\text{ELSE}}\text{ statement-2}\\\underline{\text{ELSE NEXT SENTENCE}}\end{array}\right\}$

<u>INSPECT</u> identifier-1 <u>TALLYING</u>

$\left\{\text{identifier-2 } \underline{\text{FOR}} \left\{\left\{\begin{array}{l}\underline{\text{ALL}}\\\underline{\text{LEADING}}\\\underline{\text{CHARACTERS}}\end{array}\right\} \begin{array}{l}\text{identifier-3}\\\text{literal-1}\end{array}\right\} \left[\begin{array}{l}\underline{\text{BEFORE}}\\\underline{\text{AFTER}}\end{array}\right] \text{INITIAL}\right.$

$\left.\begin{array}{l}\text{identifier-4}\\\text{literal-2}\end{array}\right]\right\} ... \right\} ...$

<u>INSPECT</u> identifier-1 <u>REPLACING</u>

$\left(\begin{array}{l}\underline{\text{CHARACTERS}}\ \underline{\text{BY}}\ \left\{\begin{array}{l}\text{identifier-6}\\\text{literal-4}\end{array}\right\}\ \left[\begin{array}{l}\underline{\text{BEFORE}}\\\underline{\text{AFTER}}\end{array}\right]\ \text{INITIAL}\ \left[\begin{array}{l}\text{identifier-7}\\\text{literal-5}\end{array}\right]\end{array}\right.$

$\left\{\left\{\begin{array}{l}\underline{\text{ALL}}\\\underline{\text{LEADING}}\\\underline{\text{FIRST}}\end{array}\right\}\ \left\{\begin{array}{l}\text{identifier-5}\\\text{literal-3}\end{array}\right\}\ \underline{\text{BY}}\ \left\{\begin{array}{l}\text{identifier-6}\\\text{literal-4}\end{array}\right\}\ \left[\begin{array}{l}\underline{\text{BEFORE}}\\\underline{\text{AFTER}}\end{array}\right]\ \text{INITIAL}\right.$

$\left.\left.\begin{array}{l}\text{identifier-7}\\\text{literal-5}\end{array}\right]\right\} ...\right\} ...\right)$

<u>INSPECT</u> identifier-1 <u>TALLYING</u>

$\left\{\text{identifier-2 } \underline{\text{FOR}} \left\{\left\{\begin{array}{l}\underline{\text{ALL}}\\\underline{\text{LEADING}}\\\underline{\text{CHARACTERS}}\end{array}\right\} \begin{array}{l}\text{identifier-3}\\\text{literal-1}\end{array}\right\} \left[\begin{array}{l}\underline{\text{BEFORE}}\\\underline{\text{AFTER}}\end{array}\right] \text{INITIAL}\right.$

$\left.\begin{array}{l}\text{identifier-4}\\\text{literal-2}\end{array}\right]\right\} ... \right\} ...$

<u>REPLACING</u>

$\left(\begin{array}{l}\underline{\text{CHARACTERS}}\ \underline{\text{BY}}\ \left\{\begin{array}{l}\text{identifier-6}\\\text{literal-4}\end{array}\right\}\ \left[\begin{array}{l}\underline{\text{BEFORE}}\\\underline{\text{AFTER}}\end{array}\right]\ \text{INITIAL}\ \left[\begin{array}{l}\text{identifier-7}\\\text{literal-5}\end{array}\right]\end{array}\right.$

$\left\{\left\{\begin{array}{l}\underline{\text{ALL}}\\\underline{\text{LEADING}}\\\underline{\text{FIRST}}\end{array}\right\}\ \left\{\begin{array}{l}\text{identifier-5}\\\text{literal-3}\end{array}\right\}\ \underline{\text{BY}}\ \left\{\begin{array}{l}\text{identifier-6}\\\text{literal-4}\end{array}\right\}\ \left[\begin{array}{l}\underline{\text{BEFORE}}\\\underline{\text{AFTER}}\end{array}\right]\ \text{INITIAL}\right.$

$\left.\left.\begin{array}{l}\text{identifier-7}\\\text{literal-5}\end{array}\right]\right\} ...\right\} ...\right)$

<u>MOVE</u> $\left\{\begin{array}{l}\text{identifier-1}\\\text{literal}\end{array}\right\}$ <u>TO</u> identifier-2 [identifier-3] ...

<u>MOVE</u> $\left\{\begin{array}{l}\underline{\text{CORRESPONDING}}\\\underline{\text{CORR}}\end{array}\right\}$ identifier-1 <u>TO</u> identifier-2

<u>MULTIPLY</u> $\left\{\begin{array}{l}\text{identifier-1}\\\text{literal-1}\end{array}\right\}$ <u>BY</u> identifier-2 [<u>ROUNDED</u>]
 [identifier-3 [<u>ROUNDED</u>]] ... [ON <u>SIZE</u> <u>ERROR</u> imperative-statement]

<u>MULTIPLY</u> $\left\{\begin{array}{l}\text{identifier-1}\\\text{literal-1}\end{array}\right\}$ <u>BY</u> $\left\{\begin{array}{l}\text{identifier-2}\\\text{literal-2}\end{array}\right\}$ <u>GIVING</u> identifier-3 [<u>ROUNDED</u>]
 [identifier-4 [<u>ROUNDED</u>]] ... [ON <u>SIZE</u> <u>ERROR</u> imperative-statement]

OPEN $\left\{\begin{array}{l}\underline{\text{INPUT}}\text{ file-name-1[WITH }\underline{\text{NO}}\text{ }\underline{\text{REWIND}}]\text{ [file-name-2 [WITH }\underline{\text{NO}}\text{ }\underline{\text{REWIND}}]]...\\ \underline{\text{OUTPUT}}\text{ file-name-3[WITH }\underline{\text{NO}}\text{ }\underline{\text{REWIND}}]\text{ [file-name-4 [WITH }\underline{\text{NO}}\text{ }\underline{\text{REWIND}}]]...\\ \underline{\text{I-O}}\text{ file-name-5 [file-name-6]...}\\ \underline{\text{EXTEND}}\text{ file-name-7 [file-name-8]...}\end{array}\right\}$ ...

PERFORM procedure-name-1 $\left[\left\{\begin{array}{l}\underline{\text{THROUGH}}\\ \underline{\text{THRU}}\end{array}\right\}\text{procedure-name-2}\right]$

PERFORM procedure-name-1 $\left[\left\{\begin{array}{l}\underline{\text{THROUGH}}\\ \underline{\text{THRU}}\end{array}\right\}\text{procedure-name-2}\right]$ $\left\{\begin{array}{l}\text{identifier-1}\\ \text{integer-1}\end{array}\right\}$ $\underline{\text{TIMES}}$

PERFORM procedure-name-1 $\left[\left\{\begin{array}{l}\underline{\text{THROUGH}}\\ \underline{\text{THRU}}\end{array}\right\}\text{procedure-name-2}\right]$ $\underline{\text{UNTIL}}$ condition-1

PERFORM procedure-name-1 $\left[\left\{\begin{array}{l}\underline{\text{THROUGH}}\\ \underline{\text{THRU}}\end{array}\right\}\text{procedure-name-2}\right]$

$\underline{\text{VARYING}}$ $\left\{\begin{array}{l}\text{identifier-2}\\ \text{index-name-1}\end{array}\right\}$ $\underline{\text{FROM}}$ $\left\{\begin{array}{l}\text{identifier-3}\\ \text{index-name-2}\\ \text{literal-1}\end{array}\right\}$

$\underline{\text{BY}}$ $\left\{\begin{array}{l}\text{identifier-4}\\ \text{literal-2}\end{array}\right\}$ $\underline{\text{UNTIL}}$ condition-1

$\left[\underline{\text{AFTER}}\right.$ $\left\{\begin{array}{l}\text{identifier-5}\\ \text{index-name-3}\end{array}\right\}$ $\underline{\text{FROM}}$ $\left\{\begin{array}{l}\text{identifier-6}\\ \text{index-name-4}\\ \text{literal-3}\end{array}\right\}$

$\underline{\text{BY}}$ $\left\{\begin{array}{l}\text{identifier-7}\\ \text{literal-4}\end{array}\right\}$ $\underline{\text{UNTIL}}$ condition-2

$\left[\underline{\text{AFTER}}\right.$ $\left\{\begin{array}{l}\text{identifier-8}\\ \text{index-name-5}\end{array}\right\}$ $\underline{\text{FROM}}$ $\left\{\begin{array}{l}\text{identifier-9}\\ \text{index-name-6}\\ \text{literal-5}\end{array}\right\}$

$\underline{\text{BY}}$ $\left\{\begin{array}{l}\text{identifier-10}\\ \text{literal-6}\end{array}\right\}$ $\left.\left.\underline{\text{UNTIL}}\text{ condition-3}\right]\right]$

READ file-name [NEXT]RECORD[INTO identifier][AT END imperative-statement]
READ file-name RECORD[INTO identifier] [INVALID KEY imperative-statement]
READ file-name RECORD[INTO identifier] [;KEY IS data-name]
     [;INVALID KEY imperative-statement]

REWRITE record-name[FROM identifier][INVALID KEY imperative-statement]

SEARCH identifier-1 $\left[\underline{\text{VARYING}}\left\{\begin{array}{l}\text{identifier-2}\\ \text{index-name-1}\end{array}\right\}\right]$ [AT END imperative-statement-1]

$\underline{\text{WHEN}}$ condition-1 $\left\{\begin{array}{l}\text{imperative-statement-2}\\ \underline{\text{NEXT}}\text{ }\underline{\text{SENTENCE}}\end{array}\right\}$

$\left[\underline{\text{WHEN}}\text{ condition-2}\left\{\begin{array}{l}\text{imperative-statement-3}\\ \underline{\text{NEXT}}\text{ }\underline{\text{SENTENCE}}\end{array}\right\}\right]$ ...

SEARCH ALL identifier-1[AT END imperative-statement-1]

$\underline{\text{WHEN}}$ $\left\{\begin{array}{l}\text{data-name-1}\\ \\ \text{condition-name-1}\end{array}\right.$ $\left\{\begin{array}{l}\text{IS }\underline{\text{EQUAL}}\text{ TO}\\ \text{IS }=\end{array}\right\}$ $\left.\left\{\begin{array}{l}\text{identifier-3}\\ \text{literal-1}\\ \text{arithmetic-expression-1}\end{array}\right\}\right\}$

$\left[\underline{\text{AND}}\left\{\begin{array}{l}\text{data-name-2}\\ \\ \text{condition-name-2}\end{array}\right.\left\{\begin{array}{l}\text{IS }\underline{\text{EQUAL}}\text{ TO}\\ \text{IS }=\end{array}\right\}\left.\left\{\begin{array}{l}\text{identifier-4}\\ \text{literal-2}\\ \text{arithmetic-expression-2}\end{array}\right\}\right\}\right]$ ...

$\left\{\begin{array}{l}\text{imperative-statement-2}\\ \underline{\text{NEXT}}\text{ }\underline{\text{SENTENCE}}\end{array}\right\}$

SET $\left\{ \begin{array}{ll} \text{identifier-1} & \text{[identifier-2]...} \\ \text{index-name-1} & \text{[index-name-2]...} \end{array} \right\}$ TO $\left\{ \begin{array}{l} \text{identifier-3} \\ \text{index-name-3} \\ \text{integer-1} \end{array} \right\}$

SET   index-name-4   [index-name-5]... $\left\{ \begin{array}{l} \underline{\text{UP BY}} \\ \underline{\text{DOWN BY}} \end{array} \right\}$ $\left\{ \begin{array}{l} \text{identifier-4} \\ \text{integer-2} \end{array} \right\}$

START file-name $\left[\text{KEY} \left\{ \begin{array}{l} \text{IS } \underline{\text{EQUAL}} \text{ TO} \\ \text{IS } = \\ \text{IS } \underline{\text{GREATER}} \text{ THAN} \\ \text{IS } > \\ \text{IS } \underline{\text{NOT}} \underline{\text{LESS}} \text{ THAN} \\ \text{IS } \underline{\text{NOT}} < \end{array} \right\} \text{data-name} \right]$

[INVALID KEY imperative-statement]

STOP $\left\{ \begin{array}{l} \underline{\text{RUN}} \\ \text{literal} \end{array} \right\}$

STRING $\left\{ \begin{array}{l} \text{identifier-1} \\ \text{literal-1} \end{array} \right\}$ $\left[\begin{array}{l} \text{identifier-2} \\ \text{literal-2} \end{array} \right]$ ... DELIMITED BY $\left\{ \begin{array}{l} \text{identifier-3} \\ \text{literal-3} \\ \underline{\text{SIZE}} \end{array} \right\}$

$\left[\left\{ \begin{array}{l} \text{identifier-4} \\ \text{literal-4} \end{array} \right\} \left[\begin{array}{l} \text{identifier-5} \\ \text{literal-5} \end{array} \right] ... \underline{\text{DELIMITED}} \text{ BY} \left\{ \begin{array}{l} \text{identifier-6} \\ \text{literal-6} \\ \underline{\text{SIZE}} \end{array} \right\} \right]$ ...

INTO identifier-7 [WITH POINTER identifier-8]
     [ON OVERFLOW imperative-statement]

SUBTRACT $\left\{ \begin{array}{l} \text{identifier-1} \\ \text{literal-1} \end{array} \right\}$ $\left[\begin{array}{l} \text{identifier-2} \\ \text{literal-2} \end{array} \right]$ ... FROM identifier-m[ROUNDED]
     [identifier-n[ROUNDED]]...[ON SIZE ERROR imperative-statement]

SUBTRACT $\left\{ \begin{array}{l} \text{identifier-1} \\ \text{literal-1} \end{array} \right\}$ $\left[\begin{array}{l} \text{identifier-2} \\ \text{literal-2} \end{array} \right]$ ... FROM $\left\{ \begin{array}{l} \text{identifier-m} \\ \text{literal-m} \end{array} \right\}$

     GIVING identifier-n[ROUNDED] [identifier-o[ROUNDED]]...
     [ON SIZE ERROR imperative-statement]

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

UNSTRING identifier-1

$\left[\underline{\text{DELIMITED}} \text{ BY [}\underline{\text{ALL}}\text{] } \left\{ \begin{array}{l} \text{identifier-2} \\ \text{literal-1} \end{array} \right\} \left[\underline{\text{OR}} \text{ [}\underline{\text{ALL}}\text{] } \left\{ \begin{array}{l} \text{identifier-3} \\ \text{literal-2} \end{array} \right\} \right] ... \right]$
     INTO identifier-4[DELIMITER IN identifier-5] [COUNT IN identifier-6]
     [identifier-7 [DELIMITER IN identifier-8][COUNT IN identifier-9]]...
     [WITH POINTER identifier-10][TALLYING IN identifier-11]
     [ON OVERFLOW imperative-statement]

USE AFTER STANDARD $\left\{ \begin{array}{l} \underline{\text{EXCEPTION}} \\ \underline{\text{ERROR}} \end{array} \right\}$ PROCEDURE ON $\left\{ \begin{array}{l} \text{file-name-1[file-name-2]...} \\ \underline{\text{INPUT}} \\ \underline{\text{OUTPUT}} \\ \underline{\text{I-O}} \\ \underline{\text{EXTEND}} \end{array} \right\}$

WRITE record-name [FROM identifier-1]

$$\left[\left\{ \begin{array}{l} \underline{BEFORE} \\ \underline{AFTER} \end{array} \right\} \quad ADVANCING \quad \left\{ \left\{ \begin{array}{l} identifier-2 \\ integer \end{array} \right\} \left[\begin{array}{l} LINE \\ LINES \end{array} \right] \right\} \right]$$
$$[PAGE]$$

$$[AT \quad \left\{ \begin{array}{l} \underline{END-OF-PAGE} \\ \underline{EOP} \end{array} \right\} \quad imperative-statement]$$

WRITE record-name [FROM identifier] [INVALID KEY imperative-statement]

COPY $\left\{ \begin{array}{l} text-name \\ literal-3 \end{array} \right\}$

$$\left[\underline{REPLACING} \quad \left\{ \left\{ \begin{array}{l} literal-1 \\ word-1 \end{array} \right\} \quad \underline{BY} \quad \left\{ \begin{array}{l} literal-2 \\ word-2 \end{array} \right\} \right\} \quad \ldots \right]$$

NOTE: A COPY statement may appear anywhere that a word
appears in the COBOL source program.

APPENDIX B

LOGICAL UNIT NUMBER (LUN) ASSIGNMENTS

LUN	ASSIGNMENT
1	Console, input
2	Console, output
3	Source input file
4	Source listing output file
5	Object output file
6	ODL output file
7	CREF scratch file
8	COPY library input file
9	Work file
10	Work file
11	Intermediate file
12	Sort work file
13	Sort work file
14	Sort work file

APPENDIX C

PDP-11 COBOL COMPILER IMPLEMENTATION LIMITATIONS

This appendix documents the implementation limitations for the PDP-11 COBOL compiler system (compiler and OTS). The reader should not confuse the term "limitations" with "restrictions". Restrictions delimit those language facilities which are not implemented or should not be used due to known bugs in their existent implementation. Implementation limitations quantify the limits of a particular language facility supported by the system. Practical implementation limitations exist in every compiler.

Such limitations are due to the finite size of various compiler tables, compiler data structure representations, etc. Since the PDP-11 COBOL compiler employs a Virtual Memory System to support many compiler data structures, the quantities specified for various implementation limitations are approximations. However, as a general rule, the following guidelines should not be exceeded in the development of a COBOL program.

IMPLEMENTATION LIMITATIONS

1. The maximum length of any COBOL data item (group item, elementary item, table) is 4095 characters.

2. The default depth of dynamic PERFORM statement nesting is 10. The default depth can be modified by using the /PFM switch at compile time.

3. The maximum number of sending operands in a DISPLAY statement is 16.

4. The maximum number of data-name definitions in a COBOL program is approximately 2000.

5. The maximum number of procedure name definitions in a COBOL program is approximately 2000.

6. The maximum nesting depth of matching parentheses in a COBOL expression is 10.

7. The maximum number of qualifiers in a qualified data-name reference is 48.

8. The maximum number of procedure names in a GO TO DEPENDING statement is 16.

APPENDIX D

COMPILER GENERATED PSECTS

An object program generated by the PDP-11 COBOL compiler is composed
of program sections called PSECTs. Three types of PSECTs are
generated:

- Data Psects   Contain the memory for the Data Division
  of a COBOL program.

- Control PSECTS   Contain the data that is required by the
  OTS during program execution.

- Procedural PSECTS   Contain the object code generated for
  the Procedure Division.

Data and Control PSECTs are always non-overlayable. Procedural
PSECTs, however, can be optionally overlayable or non-overlayable.

D.1 PSECT NAMING CONVENTIONS

The PSECTs generated by the PDP-11 COBOL compiler are named entities.
Each PSECT name is composed of a three character prefix followed by a
three character suffix. There are two different forms of the prefix:

- $KK

  Where: $ Is a sentinel character and is always present.

  KK   Is a two character kernel that identifies the
  PSECT. It is this kernel character that is
  specified by the /KER:kk switch. The /KER:kk
  switch is appended to the compiler command line to
  assign a unique kernel value to the PSECTs
  generated during the compilation. The default
  kernel assignment is C$.

- $CB

  Where: $ Is a sentinel character, and is always present.

  CB   Is a two character code that identifies the PSECT
  as a COBOL compiler generated PSECT.

COMPILER GENERATED PSECTS

PSECTs with the prefix $CB are generated to provide the control and work space required for I/O operations.

PSECTs with these same names are generated for each COBOL compilation. They are either overlayed or concatenated at task-build time. Those that are overlayed, have a known fixed length at task-build time. Those that are concatenated, have a known length at compile-time and contribute their size to the total size of the PSECT that is built by the Task Builder.

The three character suffix identifies the type of code or data the PSECT contains. Table D-1 describes the suffixes assigned to $KK type PSECTs, and Table D-2 describes the suffixes assigned to $CB type PSECTs.

Table D-1
$KK PSECT Name Suffixes

Type	Suffix	Content
Data	DAT	Data Division data storage areas.
	DDD	Data Division directories – contains descriptions of referenced Data Division items.
	ARG	Directories of referenced Linkage Section items.
	LIT	Literal Pool – contains all of the literals referenced in the program.
	LTD	Literal Directory.
	IOB	Input/Output buffers.
Control	WRK	COBOL compile unit work space – contains a description of the compile unit environment.
	PDT	PSECT dispatch table – used for intra-program control of segmented COBOL programs.
	SDT	Subprogram dispatch table – used for inter-program control (i.e., calling subprograms).
	LST	Argument list work space – used to contain the argument list passed to the called subprogram.
	PFM	Perform work space – used to provide control and checking of nested PERFORM statements.
	ADT	ALTER Dispatch Table – used to contain the destination of alterable GO TO statements.

COMPILER GENERATED PSECTS

Table D-1 (Cont.)
$KK PSECT Name Suffixes

Type	Suffix	Content
	USE	Default USE procedure table – used to access the default OPEN mode (INPUT, OUTPUT, I-O, or EXTEND) USE procedures, if present.
Procedural	ENT	Code generated by the compiler for the program entry point.
	nnn	Numbered suffixes beginning with 001. These numbered PSECTs contain the object code generated for the Procedure Division of a COBOL program.

Table D-2
PSECT Name Suffixes

Allocation	Suffix	Content
OVR	IOT	Input/Output Table – contains a reference to each COBOL Input/Output OTS routine required by the COBOL compilation.
OVR	FAl	File Access Block (FAB) – used to transmit information to RMS at open and close time.
OVR	XAl	Auxiliary Access Blocks (XABs) – used to transmit information on the keys for indexed files to RMS at open time.
OVR	SWT	COBOL switches flag PSECT. Indicates whether COBOL switches are referenced in the COBOL program.
CON	IF1	Internal File Access Blocks (IFABs) – used internally by RMS to store information.
CON	IRI	Internal Record Access Blocks (IRABs) – used internally by RMS to store information.
CON	KDl	Internal Key Descriptors – used internally by RMS to store information on the keys for indexed files.
CON	BDl	Buffer Descriptor Blocks (BDBs) – used internally by RMS to store information on the buffers.

Table D-2 (Cont.)
PSECT Names Suffixes

Allocation	Suffix	Content
CON	KB1	Key Buffers - used internally by RMS to store keys for indexed files.
CON	FD1	FDA Index Vector - contains address of first FDA in program.
		Note:
		OVR indicates overlayable PSECT.
		CON indicates concatenatable PSECT.

APPENDIX E

SORTING FILES IN A COBOL PROGRAM

Files prepared for or by COBOL programs may be sorted using the SORT utility, which is discussed in the PDP-11 SORT Reference Manual. A major portion of that facility is available to the COBOL programmer through usage of a set of subroutine linkages, described in detail in this chapter. All such linkages involve use of a CALL statement with an appropriate parameter list.

E.1 CALL STATEMENTS REQUIRED

A set of five CALL statements, each calling a particular SORT subroutine, is required within a COBOL program in order to produce a sorted output file. Each of these subroutines (RSORT, RELES, MERGE, RETRN, ENDS) performs a specialized function in the SORT procedural sequence and lets the COBOL programmer both specify sorting parameters and perform special operations on individual records as they pass through the initial and final phases.

E.1.1 Initializing the SORT - CALL RSORT

The following statement is needed to initialize the sorting operation:

    CALL "RSORT" USING IERROR, KEYSIZ, MAXREC, KEYLOC, SRTBUF,
    BUFSIZ, SCRNUM.

Parameter usage is as follows:

IERROR -	location in which a SORT subroutine may place a non-zero error code, if necessary, in COMP form, value less than 100.
KEYSIZ -	location containing byte count of total key size in COMP form, a positive even integer.
MAXREC -	location containing byte count of maximum data record size in COMP form, a positive even integer. The sum of KEYSIZ and MAXREC cannot exceed 16,383 (decimal).
KEYLOC -	address of most major word in key. See Section E.2 for details on setting up sort key.
SRTBUF -	address of first word in sort work area.
BUFSIZ -	location containing byte count of sort work area size in COMP form.

SCRNUM –   location containing number of scratch files available to the SORT (not less than 3, not more than 8), in COMP form.

E.1.2 Passing a Record to the Sort – CALL RELES

The following statement is needed to pass a record to the sort:

CALL "RELES" USING IERROR, RECSIZ, INREC.

Parameter usage is as follows:

IERROR –   usage is as described above.

RECSIZ –   location containing byte count of data record size in COMP form, a positive even integer not greater than value in MAXREC.

INREC –   address of record to be passed to the sort.

E.1.3 Merging the Scratch Files – CALL MERGE

The following statement is needed to merge the scratch files in the sort after all input records have been passed to the sort:

CALL "MERGE" USING IERROR.

IERROR usage is as described above.

E.1.4 Requesting an OUTPUT Record – CALL RETRN

The following statement is needed to request the output records, one at a time, produced in sorted order by the sort:

CALL "RETRN" USING IERROR, RECSIZ, OUTREC.

Parameter usage is as follows:

IERROR –   usage is as described above.

RECSIZ –   location to receive byte count of returned data record size in COMP form, a positive even integer not greater than value in MAXREC.

OUTREC –   address of area to receive returned data record.

NOTE

RETRN indicates "no more records" by placing a negative value in IERROR.

E.1.5 Terminating the Sort - CALL ENDS

The following statement is needed to terminate the sort after all
sorted output records have been returned:

    CALL "ENDS" USING IERROR.

IERROR usage is as described above.

E.2 SETTING UP THE KEY

Before CALL RELES is executed, the COBOL programmer must first set up
the key in an area outside the record itself. Since the key area must
begin and end on a word boundary, usage of an 01 level description in
the Working - Storage Section is recommended. The most major byte for
the key, that byte "on the left", must be stored in the highest memory
location of the key area, and the most minor byte, that byte "on the
right", must be stored in the lowest memory location.

Thus the data must be moved byte by byte, NOT word by word, to the key
area, resulting in the key being stored "backwards" by bytes. If the
actual key contains an odd number of bytes, the last unused position
must be zeroed out, to insure proper results from word compares. Thus
for a key of 7 bytes, KEYSIZ - 8; the contents of the lowest byte
address should always be zero.

The form of the comparison is logical, i.e., all eight bits of a byte
are significant;   there is no implied sign.   The programmer is
responsible for organizing the key data passed to the sort in a form
which ensures the correct sequence.

E.3 WORK AREA SIZE

The size of the sort work area, BUFSIZ, must be at least as large as
the result of the following calculation:

    Minimum BUFSIZE = SCRNUM * (1110 + MAXREC + KEYSIZE)

If less space is provided, the sort will keep decreasing the number of
work files until either the above equation is satisfied or the number
of files drop below three; the latter is an error condition (error
code 17).

Any extra memory will be used to expand the in-core sort area.   Thus,
in general, the more space supplied, the faster the sort.

E.4 TYPICAL USAGE SEQUENCE

Sort the file SORT-IN to produce the file SORT-OUT.

    1. Open SORT-IN.

    2. Call RSORT to initialize the sort.

    3. Read the next logical record from SORT-IN. If no more data,
        go to step 7.

4. Perform any desired operations upon the input record. If it is not to be submitted to the sort, go to step 3.

5. Set up the keys from the new record.

6. Call RELES to give the record to the sort, then loop back to step 3.

7. Close SORT-IN.

8. Call MERGE to collate the records submitted to the sort.

9. Open SORT-OUT.

10. Call RETRN to get the next sorted output record. If no more records, go to step 13.

11. Perform any desired operations upon the sorted output record. If it is not to be included in the SORT-OUT file, go to step 10.

12. Write the record onto SORT-OUT, then loop back to step 10.

13. Close SORT-OUT.

14. Call ENDS to clean up the sort scratch files.

E.5 LINKING SORT ROUTINES WITH A COBOL PROGRAM

The actual sorting subroutines are contained in SORTS.OBJ and SIORMS.OBJ which are included in the COBOL object library (COBLIB). The programmer can link these to his own calling program, by following the usual procedure for using the Task Builder to task-build any COBOL program.

Note that the sort subroutines use LUNs 5, 6, ... 12 for the scratch files. Use the task builder device assignment (ASG) command appropriately. The LUN can be overridden by globally patching location $RFIRL. Insure that the LUNs used by the sort subroutines do not conflict with the LUNs assigned to files in the COBOL program that might be open when the sort subroutines are called.

E.6 COMPARISON WITH ANS COBOL SORT VERB

Readers familiar with the ANS COBOL SORT verb will recognize that a substantial portion of that capability has been described in this chapter. The following points of comparison will be helpful in converting from such usage to the described facility:

1. INPUT PROCEDURES are available thru the CALL RELES usage.

2. OUTPUT PROCEDURES are available thru the CALL RETRN usage.

3. Only ASCENDING keys are supported. The programmer can get the effect of DESCENDING key fields by simply complementing them when he stores them in KEYLOC. Note that the data record itself is unaffected by this procedure, so restoration of such fields after the sort is unnecessary.

4. The COLLATING-SEQUENCE option is not directly available.
    Again, however, the programmer could transform key fields
    when storing them in KEYLOC to achieve the desired effect.

5. There is no MERGE feature.

6. Multiple usages of the sort may occur within a given COBOL
    program provided that "RSORT" and "END" bracket each usage.

7. There is no restriction on the presence of COBOL code in
    addition to INPUT and OUTPUT PROCEDURES.

E.7 ERROR CODES

Whenever the sort detects an error, it returns a non-zero code to the
location specified by the programmer (IERROR in discussion above).
The error codes (octal representation) and their meanings are:

DEC	OCTAL	
	00	No errors
	01	Device input error
	02	Device output error
	03	OPEN INPUT failure
	04	OPEN OUTPUT failure
	05	Size of current record is greater than maximum size
	06	Not enough work area
	07	"RETRN" was called after it had exited with a negative error code (end of sort).
8	10	SORT routine called out of order. The order of the calls must be RSORT, RELES, MERGE, RETRN, ENDS.
9	11	Sort already in progress. To do a second sort, ENDS must be called to clean up the first sort.
10	12	Key size is not positive, SORTS detected a zero or negative key size in its calling parameter.
11	13	Record size not positive.
12	14	Key address not even. The keys must start at an even address (SORT uses word moves).
13	15	Record address not even.

SORTING FILES IN A COBOL PROGRAM

DEC	OCTAL	
14	16	Scratch records will be too large. The size of the keys plus the size of the largest record must be less than 377776 (octal).
15	17	Too few scratch files. A minimum of 3 scratch files must be specified.
16	20	Too many scratch files. A maximum of 10 scratch files may be specified.
17	21	End-of-string record was detected where none was expected.
18	22	Like 21, but for End-of-File.
19	23	SORT found a record larger than it expected.
20	24	Record length is non-standard for SORTT, SORTA, SORTI.

[COMP items are displayed in DECIMAL!]

APPENDIX F

COBOL TERMINAL HANDLING SERVICES ON RSTS/E

The COBOL-11 runtime library contains a set of callable subroutines that support multi-terminal access from a single COBOL program. These subroutines run only on RSTS/E.

The purpose of this subroutine package is to provide asynchronous terminal I/O support for COBOL programs running on RSTS/E.

F.1 GENERAL SERVICES

The subroutines provide the following services:

1. The ability to assign and deassign available terminal (keyboards) to the running COBOL program.

2. The ability to OPEN or CLOSE a specific I/O channel and logical unit pair for terminal INPUT/OUTPUT.

3. The ability to WRITE a message to any single terminal assigned to the program.

4. The ability to READ a message from a specific terminal.

5. The ability to READ a message from any terminal in the group assigned to the program and have the subroutine identify from which terminal the message came. This technique is known as polling.

6. In conjunction with the READ capabilities, the user has the ability to specify how long to wait for a message from any terminal before returning to the user program.

F.1.1 Open a Logical Unit for Terminal I/O

This function must be called to initialize the multi-terminal subroutines to expect terminal I/O on a specified logical unit (LUN). The LUN is required by the subsequent terminal I/O subroutines to function properly.

The form of the call is

    CALL "KBOPEN" USING ERR, LUN

Where:

> ERR - is a comp data-item [PIC 9(4) COMP] that contains the returned error status code. (See Section F.4.)
>
> LUN - a comp data-item [PIC 9(4) COMP] that contains the decimal logical unit number to use.

Example

MOVE 14 TO LUN.
CALL "KBOPEN" USING ERR, LUN.

An error code of zero indicates a successful call.

The choice of LUN number is very important and must comply with the following rules:

1. It must be in the range of 1 to 15.

2. It must not conflict with a LUN number assigned by the COBOL compiler to a file in the COBOL program.

F.1.2 Close a Terminal Logical Unit

This function disassociates a LUN and all keyboards assigned to the LUN from the running COBOL program. The form of the call is

> CALL "KBCLOS" USING ERR, LUN

Where:

> ERR and LUN are as specified in
>
> > KBOPEN

F.1.3 Assigning a Terminal

In order to use the RSTS/E COBOL multi-terminal functions, each terminal must be assigned to the COBOL program. A CALL is made to the subroutine KBASGN to assign a specific terminal or keyboard (KB) to a logical unit (LUN). This LUN must have been the subject of a previously executed CALL to KBOPEN in the COBOL program.

The form of the CALL is

> CALL "KBASGN" USING ERR, KB-UNIT

Where:

> ERR - A 2-byte COMP data-item [PIC 9(4) COMP] that contains an error code returned by the subroutine (see Section F.4).
>
> KB-UNIT - A 2-byte comp data-item [PIC 9(4) COMP] that contains the keyboard number in decimal.

F.1.4 Deassigning a Terminal

This function removes the specified terminal unit from the list assigned to the specified LUN. The form of the CALL is

        CALL "KBDEAS" USING ERR, KB-UNIT

Where:

        ERR AND KB-UNIT are as specified for the call to KBASGN.

F.1.5 Write to a Specific Terminal

Assuming that the specified terminal has been assigned, this function delivers a message to the indicated terminal. The form of the call is

        CALL "KBWRIT" USING ERR, COUNT, MESSAGE, LUN, KB-UNIT

Where:

        ERR -   the 2-byte comp data-item as specified earlier.

        COUNT -   a 2-byte comp data-item [PIC 9(4) COMP] that contains
                    the length the message in bytes.

        MESSAGE -   the data-item that contains the message to be written.
                    This message must contain all vertical and horizontal
                    formatting characters such as carriage returns, line
                    feeds and tabs.

        LUN -   a 2-byte comp item [PIC 9(4) COMP] as previously
                    specified.

        KN-UNIT -   a 2-byte comp data-item [PIC 9(4) COMP] identifying the
                    specific terminal to which the message is written..

F.1.6 Read from a Specific Terminal

This function allows a COBOL program to read a message from a specific terminal and optionally waits for as much as 255 seconds for input from the terminal. The form of the call is

        CALL "KBREAD" USING ERR, COUNT, MESSAGE, LUN, KB-UNIT, TIME

Where:

        ERR -   is the 2-byte comp data-item where a success/error code
                    will be returned.

        COUNT -   is the 2-byte comp data-item which contains the length
                    of the message just read.

        MESSAGE -   defines the data-item into which the message is read.
                    This data item should be long enough to contain the
                    longest anticipated message to be read.

COBOL TERMINAL HANDLING SERVICES ON RSTS/E

When a message is read from a terminal, the message is
prefixed by a 1-byte field which contains the terminal
unit number in binary. Therefore space should be
reserved in the message input data item for this byte.

i.e.,

        01 MESSAGE
        02 KB-NUM PIC X.
        02 REAL-MESSAGE PIC X(80)

All messages are returned as ASCII strings with no
conversions taking place.

LUN -   a 2-byte comp data-item containing the LUN number.

KN-UNIT - a 2-byte comp data-item containing the terminal number
         to be used in the READ.

TIME -   a 2-byte comp data-item containing a value from 1 to
         255 which is the amount of time in seconds to wait for
         input from the terminal(s). If no message is available
         in this time, then an error 13 (user data error on
         device) is returned.

         If TIME is given a zero value, then the system will
         wait indefinitely for input from the terminal(s).

         If a terminal operator types CONTROL-Z at a terminal,
         error code 11 (end-of-file on device) is returned.

F.1.7   READ Unsolicited from Any Terminal Assigned

This function allows a COBOL program to read a message from any
terminal assigned to the program. The read is called unsolicited
because no specific terminal is identified and what is expected is a
message from the first terminal found to have typed in a message.
This function can also wait for input from a terminal for a specified
length of time - up to 255 seconds. If no message is available from
any assigned terminal within this time, then an error condition is
returned. The error code is 13 - a user data error on device
condition that is generated by the RSTS/E monitor.

Using this function, the COBOL program can effectively poll group of
terminals requesting input as any is available.

The form of this call is

    CALL "KBREAU" USING ERR, COUNT, MESSAGE, LUN, KB-UNIT TIME

Where:

    ERR, COUNT, MESSAGE, LUN and TIME are as specified in the
    description of KBREAD

    and

    KB-UNIT is a 2-byte comp data-item that contains (upon return
    from the call to KBREAD) the unit number (in binary) of the
    terminal from which the current message was read.

F.2 ERROR CODES DURING MULTI-TERMINAL HANDLING

These values are returned in binary in the 2-byte comp data-item used in every call to the multi-terminal subroutines.

CODE	MEANING

0   Successful.

6   NOT A VALID DEVICE. An attempt was made to WRIT to an unassigned device with DBWRIT.

7   I/O CHANNEL ALREADY OPEN. A KBOPEN call attempted to use a LUN that was already in use by the program for a file or for other terminal operations because of a previous KBOPEN call.

8   DEVICE NOT AVAILABLE. A KBASGN call was made to assign a terminal that is unavailable to the program and reserved by another user.

9   I/O CHANNEL NOT OPEN. A call to KBASGN, KBREAD, KBREAU or KBWRIT was made using a LUN that was not OPENed by a KBOPEN call.

11   END OF FILE ON DEVICE. A user at an assigned terminal typed CONTROL/Z during a KBREAD call.

12   FATAL SYSTEM I/O FAILURE. A system level I/O error occurred - the user has no guarantee that the last operation was performed.

13   USER DATA ERROR ON DEVICE. BAD DATA may have been transmitted during the previous I/O call or a call to KBREAD or KBREAU did not get any data in the requested wait time.

15   Keyboard wait exhausted. The WAIT time requested for input during a KBREAD or KBREAU call.

31   ILLEGAL BYTE COUNT FOR I/O. A bad message length value was used as a parameter during a DBWRIT call.

APPENDIX G
COMPILER SYSTEM ERRORS

(To be supplied in subsequent update.)

APPENDIX H

DIAGNOSTIC ERROR MESSAGES

This Appendix contains a numerical listing of the diagnostic messages generated by the PDP-11 COBOL compiler. The general format of presentation is to give the error message number and the text of the diagnostic message to the left. On the right, a detailed explanation of the diagnostic is given indicating the reason(s) for which the diagnostic message is issued and the recovery action taken by the compiler.

NOTE

In many explanations, the word "Fatal." appears as the very last sentence of the explanation. This means that this is a fatal diagnostic issued in the Procedure Division. If the /ACC:2 switch is specified in the command string input to the compiler, the associated diagnostic message will cause the generation of the special error trap coding discussed previously.

001 CONTINUE PUNCH WITH BLANK STATEMENT. IGNORED.

A blank line has a continue punch. The continue punch is ignored.

002 QUOTE OR CONTINUE PUNCH MISSING. QUOTE ASSUMED.

A non-numeric literal has no quote and the following line has no continue punch. A terminal quote is assumed at the end of the line.

003 VIOLATION OF AREA A. ASSUMED CORRECT.

The first non-blank character on a continued line occurs in Area A. The error is ignored.

004 LINE LENGTH EXCEEDS INPUT BUFFER. TRUNCATED.

Continuation lines cause a COBOL word to exceed the capacity of the input buffer. The word is truncated on the right; the number of characters retained depends on the type of word being processed.

005 .IO CONTROL. WITHOUT .FILE CONTROL. IGNORED.

An I-O-CONTROL paragraph appears when no FILE-CONTROL paragraph was present. The I-O-CONTROL paragraph is ignored.

006 .STRING. DATA ITEM MUST HAVE DISPLAY USAGE.

A data item in a STRING statement has been given a COMP or INDEX usage. Fatal.

007 NAME EXCEEDS 30 CHARACTERS. TRUNCATED TO 30.

A character string which appears to be a name exceeds 30 characters in length. The string is truncated on the right to 30 characters.

010 NUMERIC LITERAL OVER 18 DIGITS. TRUNCATED TO 18.

A numeric literal exceeds 18 digits in length. The literal is truncated on the right, with any necessary adjustment to scaling. The sign is retained.

011 NUMERIC LITERAL HAS MULTIPLE DECIMAL POINTS.

A numeric literal has more than one decimal point.

012 PICTURE CLAUSE ILLEGAL ON GROUP LEVEL. IGNORED.

A group level item has a PICTURE clause. The clause is ignored.

013 .SELECT. NOT FOUND. SENTENCE IGNORED.

A FILE-CONTROL statement should begin with the word SELECT, but does not. All words up to the next period are ignored.

014 JUST.SYNC.BLANK CLAUSES WRONG AT GROUP. IGNORED.

A group level item may not contain JUSTIFIED, SYNCHRONIZED, or BLANK WHEN ZERO clauses. The clause is ignored.

015 FILENAME MISSING OR INVALID. SELECT IGNORED.

A SELECT statement either contains no user name or the the user name is invalid. The SELECT statement is ignored.

016 USAGE CONFLICTS WITH GROUP USAGE. USES GROUP.

The usage specified for this item differs from the usage stated at a higher group level. The group level usage is used.

017 ILLEGAL NUMERIC DATANAME IN .STRING.

A numeric data item in a STRING statement has an illegal description. Fatal.

020 .ALL. ILLEGAL IN CONTEXT OF .STRING. STATEMENT.

An ALL literal has been used in a STRING statement. Fatal.

021 SYNTAX ERROR OR NO
     TERMINATOR. CLAUSES SKIPPED.

A SELECT statement is missing
its terminating period or an
error causes the statement to
be processed before all
clauses were found. The
SELECT statement is ignored.

022 NUMERIC LITERAL ILLEGAL
     IN THIS STATEMENT.

A STRING, UNSTRING, or
INSPECT statement contains a
numeric literal. Fatal.

023 SENDING LIST OMITTED IN
     .STRING. STATEMENT.

A STRING statement contains
no sending fields before a
DELIMITED BY phrase. Fatal.

024 MORE THAN ONE FILENAME
     IN .ASSIGN.

The non-numeric literal of an
ASSIGN clause contains more
than one file specification.
Only the first specification
is used.

025 ILLEGAL DATANAME FOLLOWS
     .INTO. IN .STRING.

The receiving field of a
STRING statement is invalid.
Fatal.

026 SUBSCRIPTING DEPTH EXCEEDS
     3. OVER 3 IGNORED.

This OCCURS clause is nested
more than three deep. The
OCCURS clause is ignored.

027 VALUE   ILLEGAL IN OCCURS
     ITEM.   IGNORED.

A VALUE clause appears in an
item with an OCCURS clause or
in an item subordinate to an
OCCURS clause. The VALUE
clause is ignored.

030 VALUE ILLEGAL IN
     REDEFINES ITEM. IGNORED.

A VALUE clause appears in an
item which either contains a
REDEFINES clause, or is
subordinate to an item with a
REDEFINES clause.

031 NO TERMINATOR FOR .IO
     CONTROL. PARAGRAPH.

The I-O-CONTROL paragraph is
not terminated by a period.
The terminator is assumed
present.

032 .MAP. NO LONGER APPLICABLE.
     IGNORED.

An APPLY clause with the MAP
option is not applicable for
this version and future
versions of PDP-11 COBOL. The
APPLY clause is ignored.

033 AN IO CONTROL CLAUSE
     WITHOUT FILES.

A file-name is missing in a
clause of the I-O-CONTROL
paragraph. The clause is
ignored.

034 SYNTAX ERROR IN .APPLY..

An APPLY clause has illegal
syntax. The clause is
ignored.

035 INVALID ACCESS MODE.
TREAT AS SEQUENTIAL.

The SELECT statement contains an invalid ACCESS mode. SEQUENTIAL ACCESS mode is assumed.

036 INVALID FILE ORGANIZATION.
TREAT AS SEQUENTIAL.

THE SELECT statement contains an invalid ORGANIZATION specification. SEQUENTIAL organization is assumed.

037 NO SELECT STATEMENTS.

A FILE-CONTROL paragraph either contains no SELECT statements or none of those present are valid. The FILE-CONTROL paragraph is ignored.

040 .ASSIGN. OMITTED FROM
SELECT. SELECT IGNORED.

A SELECT statement contains no ASSIGN clause. The SELECT statement is ignored.

041 DECIMAL PLACES TRUNCATED.

Decimal places have been truncated from a numeric literal during conversion for use as an integer. The integer positions are used.

042 INTEGER EXPECTED, ZERO
ASSUMED.

An integer literal was expected but fractional positions were found. The literal is ignored and a value of zero is assumed.

043 INTEGER VALUE TOO BIG.
LARGEST VALUE USED.

A numeric literal is too big for conversion as an integer in the given context. A value of 32,767 is used.

044 ERROR IN DATA RECORDS
CLAUSE. CLAUSE SKIPPED.

The word DATA is not followed by RECORD or RECORDS in the DATA RECORDS clause. The DATA RECORDS clause is ignored.

045 ERROR IN LABEL RECORDS
CLAUSE. CLAUSE SKIPPED.

The word LABEL is not followed by RECORD or RECORDS in the LABEL RECORDS clause. The LABEL RECORDS clause is ignored.

046 NO INTEGER IN BLOCK
CLAUSE. CLAUSE SKIPPED.

The BLOCK clause does not contain a numeric literal. The BLOCK clause is ignored.

047 BAD VALUE IN BLOCK
CLAUSE. CLAUSE SKIPPED.

The numeric literal in the BLOCK clause causes an illegal block size. The block size in bytes must be greater than 0 and less than 32768. The BLOCK clause is ignored.

050 NO INTEGER IN RECORD
CLAUSE. CLAUSE SKIPPED.

The RECORD CONTAINS clause does not contain a numeric literal. The RECORD CONTAINS clause is ignored.

051 INVALID VALUE IN RECORD   The numeric literal in the
     CLAUSE.   CLAUSE SKIPPED.   RECORD CONTAINS clause is not
                                      greater than zero. The
                                      RECORD CONTAINS clause is
                                      ignored.

052 INVALID FILENAME.   The word following FD is not
     FD SKIPPED.   valid as a file-name. The
                                     FD entry is ignored.

053 FD TERMINATOR MISSING.   The file description entry
     ASSUMED PRESENT.   contains no period
                                     terminator. The error is
                                     ignored.

054 KEY WORD EXPECTED.   A keyword, which begins a
     REMAINING CLAUSES SKIPPED.   clause, such as BLOCK, LABEL,
                                     DATA, etc. is missing. The
                                     remainder of the FD entry is
                                     ignored.

055 NO LABEL CLAUSE IN FD.   The FD entry contains no
     .STANDARD. ASSUMED.   LABEL RECORD clause. LABEL
                                     RECORD IS STANDARD is assumed.

056 NO SELECT. FILE   The FD entry's file-name has
     DELETED.   no corresponding SELECT
                                     statement. The FD entry is
                                     ignored. All references to
                                     the filename will be
                                     diagnosed as undefined.

057 ALLOCATED SPACE EXCEEDS   The maximum record size
     LARGEST RECORD.   specified by the RECORD
                                     CONTAINS clause exceeds the
                                     space required for any 01
                                     entry under the same file.
                                     The value specified by the
                                     RECORD CONTAINS clause is
                                     used.

060 RECORD AREA EXTENDED TO   The space required by the
     CONTAIN LARGEST RECORD.   largest 01 record under a
                                     file description exceeds the
                                     space required by the RECORD
                                     CONTAINS clause in the FD
                                     entry. The value derived
                                     from the 01 record
                                     description is used.

061 NO RECORD AREA.   FILE   No record area is allocated
     DELETED.   for a file description. The
                                     file description is ignored.
                                     All references to the file
                                     will be diagnosed as
                                     undefined.

062 ILLEGAL DATANAME FOLLOWS   The data item used as a
     .WITH POINTER.   PHRASE.   pointer in a STRING or
                                     UNSTRING statement is
                                     illegal. Fatal.

063 ILLEGAL SYNTAX IN .STRING.   A STRING statement contains
     STATEMENT.   illegal syntax. Fatal.

064 77 ILLEGAL IN FILESECTION.
CHANGED TO 01.

A 77 level item description has been found in the FILE SECTION. The 77 level is treated as an 01 level.

065 ILLEGAL WORD FOLLOWS
.DELIMITED BY. PHRASE.

A data-name or literal is expected following a DELIMITED BY phrase in a STRING or UNSTRING statement. Fatal.

066 ILLEGAL USE OF .ALL..
IGNORED.

In the VALUE clause, an ALL numeric literal is detected. This is illegal. ALL is ignored by the compiler.

067 CONDITION NAME MISSING OR
INVALID. 88 IGNORED.

The condition-name in an 88 level entry is either missing or invalid. The entire entry is ignored.

070 TWO INDEXED KEYS START AT
SAME OFFSET IN RECORD

The leftmost character position of the RECORD KEY or ALTERNATE RECORD KEY dataname corresponds to the leftmost character position of some other RECORD KEY or ALTERNATE RECORD KEY data-name. The clause is ignored.

071 .REDEFINES. ON 01 LEVEL
IN FILE SECTION INVALID.

The REDEFINES clause is present on the 01 level in the FILE SECTION, where redefinition is implicit. REDEFINES clause is ignored.

072 PICTURE IGNORED
FOR INDEX ITEM.

An item defined as USAGE INDEX has a PICTURE clause. The PICTURE clause is is ignored.

073 NONNUMERIC PIC ON COMP
ITEM. TREATED AS DISPLAY.

An item defined as USAGE COMP has a picture-string with non-numeric characters. The stated usage is ignored. The item is treated as USAGE DISPLAY.

074 SUBSCRIPT OUT OF RANGE.
ASSUME 1.

A literal subscript is either less than 1 or greater than the maximum allowable value. A value of 1 is used.

075 .STATUS. OMITTED FROM
.FILE STATUS.. ASSUMED.

The FILE STATUS clause has incorrect syntax. The error is ignored.

076 SOME FILES WITHOUT POSIT.
NO. IN MUL. FILE TAPE.

A MULTIPLE FILE TAPE clause contains file-names with POSITION Clauses. Not all the file-names contain POSITION clauses. The error is ignored. File searching during OPEN will find the file.

077 .MULTIPLE FILE TAPE. SYNTAX
      ERROR.

A MULTIPLE FILE TAPE clause contains a syntax error. The clause is ignored.

100 OPERAND CLASSES IN CONFLICT.

One or more operands in a statement have invalid class. Fatal.

101 POSSIBLE RECEIVING
      FIELD TRUNCATION.

A MOVE statement results in right hand truncation of the receiving field value. This is not an error and is ignored.

102 TOO FEW SOURCE FIELDS
      FOR ADD .GIVING..

At least two valid source operands must appear in an ADD...GIVING statement. Fatal.

103 .EXIT. WAS NOT THE ONLY
      VERB IN PARAGRAPH.

An EXIT statement is not the only statement in a paragraph. The EXIT statement is ignored.

104 SENDING ITEM INVALID
      OR OMITTED.

A MOVE statement contains an invalid or missing sending operand. Fatal.

105 SENDING ITEM NOT FOLLOWED
      BY .TO..

A MOVE statement does not have a TO following the sending operand. Fatal.

106 RECEIVING ITEM INVALID OR
      OMITTED.

A MOVE statement has no valid receiving operand. Fatal.

107 INVALID CLASS FOR
      DESTINATION FIELD.

The receiving operand of an ADD or SUBTRACT statement is not numeric or numeric-edited. Fatal.

110 RELATIVE OR RECORD KEY OR STATUS
      NAME INVALID.

The name referenced in a RELATIVE KEY, RECORD KEY, ALTERNATE RECORD KEY or FILE STATUS clause is invalid. The clause is ignored.

111 .STOP. SYNTAX ERROR.

The STOP statement is not followed by a literal or the word RUN. Fatal.

112 .SIZE ERROR. STATEMENT
      INCORRECT.

The word ERROR is not found in ON SIZE clause. Fatal.

113 .PROCEDURE DIVISION. OMITTED.

The source program does not contain a PROCEDURE DIVISION. Fatal.

114 INTERMEDIATE RESULT TOO LARGE.
      HIGH ORDER TRUNC.

An arithmetic statement calls for an intermediate result in excess of 18 digits. The intermediate result is truncated on the left to 18 digits with a possible loss of high order non-zero digits at execution time.

115	INTERMEDIATE RESULT TOO LARGE. LOW ORDER TRUNC.	An arithmetic expression calls for an intermediate result in excess of 18 digits. The intermediate result is truncated on the right to 18 digits with a possible loss of low order non-zero digits at execution time.
116	.DIVISION. OMITTED AFTER .PROCEDURE..	The word DIVISION is missing in the PROCEDURE DIVISION header. The error is ignored.
117	TERMINATOR MISSING AFTER DIVISION HEADER.	The period terminator is missing from a division header. The error is ignored.
120	LITERAL INCOMPATIBLE WITH ATTEMPTED USAGE.	Conversion of a literal from one form to another has failed. Fatal.
121	DATANAME MUST FOLLOW .INTO. IN THIS STATEMENT.	A valid data-name is not present following INTO in a STRING or UNSTRING statement. Fatal.
122	NUMERIC SUBJECT OR OBJECT MUST BE INTEGER.	A numeric, non-integer subject or object is invalid in the context of this relation condition. Fatal.
123	OPERANDS CONFLICT IN .SET...	A SET...TO statement TO. STATEMENT. references invalid operands. Fatal.
124	OPERANDS CONFLICT IN .SET ... BY. STATEMENT.	A SET...BY statement references invalid operands. Fatal.
125	ILLEGAL FILENAME LITERAL OR FILENAME DATANAME.	An ASSIGN statement or a VALUE OF ID statement contains an invalid file specification or data-name. The statement is ignored.
126	INVALID SUBJECT OF SIGN CONDITION.	The subject of a sign condition is not a valid arithmetic expression. Fatal.
127	ITEM IN TABLE MAY NOT BE USED AS A SUBSCRIPT.	A data item used as a subscript is itself a table element. Fatal.
130	.POINTER. MUST FOLLOW .WITH. IN THIS STATEMENT.	A STRING or UNSTRING statement has an invalid WITH POINTER phrase. Fatal.
131	RELATIVE KEY INVALID FOR THIS FILE. IGNORED.	A RELATIVE KEY clause has been applied to a file which

does not have RELATIVE
organization.
The RELATIVE KEY clause is
ignored.

132 SUBJECT OR OBJECT OMITTED IN
RELATION CONDITION.

The subject or object is
omitted in a COBOL relation
condition. The condition
expression is considered
syntactically invalid. Fatal.

133 UNIDENTIFIABLE WORD FOUND IN
SUBSCRIPT.

A subscript list contains
a word which is neither a
data-name or numeric literal.
The remainder of the list or
sentence is ignored.
Fatal.

134 INVALID SUBJECT OR OBJECT IN
RELATION CONDITION.

The subject or object of a
relation condition is an
invalid operand. Fatal.

135 SUBSCRIPTS OMITTED. ASSUME
VALUE OF 1.

A reference to a table item
contains no subscript list.
Literal subscripts of 1 are
supplied as defaults.

136 RELATIVE INDEX LITERAL OUT
OF   RANGE.   INDEX USED.

The literal value of a
relative index causes an
out of range reference
to the table. The literal
value is ignored, and the
index-name only is used.

137 SUBSCRIPTS GIVEN WHERE NOT
REQUIRED. IGNORED.

A reference is made to
a non-table item, and a
subscript list follows the
reference. The subscript
list is ignored.

140 TOO FEW SUBSCRIPTS GIVEN.
ASSUME 1 FOR REST.

A reference to a table item
contains a subscript list
with too few subscripts.
Default literal subscripts of
1 are supplied for missing
subscripts.

141 TOO MANY SUBSCRIPTS
GIVEN. IGNORE EXCESS.

A reference to a table item
contains too many subscripts
in the subscript list. Extra
subscripts are ignored.

142 SUBJECT AND OBJECT USAGE
MUST MATCH.

A relation condition between
non-numeric operands requires
the same usage for both
operands. Fatal.

143 ARITHMETIC EXPRESSION
REQUIRED IN THIS CONTEXT.

An arithmetic expression is
required in the context of
the COBOL statement being
compiled. The compiler has
failed to recognize the
arithmetic expression in this
context. Fatal.

144 CONDITION EXPRESSION REQUIRED
IN THIS CONTEXT.

A condition expression is
required in the context of
the COBOL statement being
compiled. The compiler has
failed to recognize the
condition expression in this
context. Fatal.

145 ILLEGAL OPERAND FOUND IN
COBOL EXPRESSION.

An invalid data-name or
literal has been found in the
COBOL statement being
compiled. The class or USAGE
of the data item may be
invalid in the context as a
reference in an expression.
Fatal.

146 OPERATOR IS MISSING IN COBOL
EXPRESSION.

An operator is omitted in the
specification of this COBOL
expression. The compiler
cannot recognize this
expression as a syntactically
valid COBOL expression.
Fatal.

147 ABSOLUTE VALUE STORED.

A negative value has been
supplied for an unsigned
numeric item. The absolute
value of the numeric literal
is stored in the item.

150 ILLEGAL WORD FOUND AFTER
.NOT. IN EXPRESSION.

The compiler has detected an
illegal expression operator
following a NOT keyword in
the COBOL expression being
compiled. The COBOL
expression is considered
syntactically invalid. Fatal.

151 VERB FOUND IN AREA A.
ALLOWED.

A statement begins in
Area A. The error is ignored.

152 EXPECTED .RELATIVE KEY.
DATANAME NOT DEFINED.

The data-name given in a
RELATIVE KEY clause has not
been defined in the Data
Division.

153 .LINAGE. CLAUSE DATAITEM
IS TOO LONG.

A data item named in a LINAGE
clause is declared in the
Data Division with more than
four decimal integer
positions of precision.

154 PROCEDURE NAME DUPLICATES
DATA NAME. ALLOWED.

A procedure name is identical
to a data-name. The error is
ignored, since there can be
no ambiguity in legal
references.

155 STATEMENTS FOLLOWING .GO.
CAN NEVER BE EXECUTED.

A statement follows an
unconditional GO statement.
The statements following the
GO are compiled, but can
not be executed.

156 NONSEQUENTIAL FILE MAY
    NOT BE OPTIONAL.

The SELECT statement may
specify OPTIONAL only on
files with sequential
organization. The word
OPTIONAL is ignored.

157 FILE HAS IO CONTROL
    CLAUSE CONFLICTS.

A file is given conflicting
clause specifications in the
I-O-CONTROL paragraph of the
INPUT-OUTPUT SECTION.

160 FILE REQUIRES REL. KEY.
    TREATED AS SEQ. ACCESS.

A file with relative
organization and random
or dynamic access has no
RELATIVE KEY clause. The
access mode is changed to
SEQUENTIAL.

161 INVALID INDEX DATAITEM USE IN
    RELATIONAL.

The compiler detects the
invalid use of an index data
item reference as the subject
or object of a relation
condition. Fatal.

162 UNKNOWN WORD. SCAN TO
    NEXT CLAUSE.

An unknown word is
encountered when a clause
keyword is expected. All
words are ignored up to the
next valid clause.

163 CLAUSE DUPLICATED. SECOND
    OCCURRENCE USED.

A SELECT statement contains
two occurrences of the same
clause. The second
occurrence is used.

164 NO FD FOR THIS SELECT.

The file-name supplied in a
SELECT statement is not
further described in an FD
in the Data Division. The
SELECT statement is ignored,
causing the filename to
become undefined.

165 DIFFERENT SAME REC. AREAS
    FOR SAME AREA.

The compiler detects a
conflict between the SAME
RECORD AREA clause and the
SAME AREA clause.

166 .READ. WITHOUT .INVALID KEY.
    .AT END. OR .USE.

A READ statement contains
no conditional clauses and
the file being read has no
USE procedure applied to it.
Fatal.

167 IO CONTROL CLAUSE HAS FILE
    WITH NO .SELECT.

An I-O-CONTROL clause
references a file-name which
was not named in a SELECT
statement. The file-name is
ignored in the I-O-CONTROL
statement.

170 INTEGER OMITTED IN
    .RESERVE.. DEFAULT ASSUMED.

A RESERVE clause fails
to specify the number of
buffer areas to reserve. The

clause is ignored, and a
default of one area for
SEQUENTIAL and RELATIVE or two
areas for INDEXED is supplied.

171   INVALID SUBJECT OF CLASS
      CONDITION.

The subject of a class
condition is not a data
item with acceptable class.
Fatal.

172   VALUE EXCEEDS FIELD CAPACITY.
      TRUNCATED.

A numeric literal supplied
by a VALUE clause exceeds the
length of the field. The
value is right truncated and
stored in the field.

173   NO DATA DIVISION STATEMENTS
      PROCESSED.

The Data Division contains
no valid entries. This
is an observation only.

174   INVALID GRP LEV NUM.
      REST OF RECORD IGNORED.

A level-number is encountered
which terminates a previous
group item, but does not
match any previous group
item's level-number. All
data entries are skipped
until the next 01 level,
level indicator or header.

175   INVALID PROCEDURE NAME
      DEFINITION IN AREA A.

The compiler detects source
text in Area A of the
Procedure Division which does
not conform to the rules for
the definition of a
legitimate paragraph or
section name. Source text
found in Area A of the
Procedure Division is
interpreted by the compiler
as a user attempt to define a
new paragraph or section
name. The compiler supplies a
system-defined procedure name
and proceeds with the
processing of the source line
text containing the invalid
Area A text. The system-
defined procedure name is
transparent and, thus,
inaccessible to the user.

176   MISSING QUOTE ON CONTINUE
      LINE.   QUOTE ASSUMED.

A non-numeric literal is
continued, but the first
non-space character is not a
quote. The error is ignored
by assuming a quote in front
of the first non-space
character.

177   COMPARISON OF LITERALS IS
      NOT PERMITTED.

A relation condition has
a literal as both subject and
object. Fatal.

200	COPY IGNORED WITHIN LIBRARY TEXT.	A COPY statement is encountered within library text. The COPY statement is ignored.
201	INVALID FILENAME ON COPY. COPY IGNORED.	A COPY statement supplies a file specification which is invalid. The COPY statement is ignored.
202	COPY FILENAME NOT FOUND.	A COPY statement supplies a valid file specification, but the file cannot be found on the specified device. The COPY statement is ignored.
203	PERIOD OMITTED AFTER .DECLARATIVES..	The word DECLARATIVES is not followed by a period. The error is ignored.
204	.DECLARATIVES. OMITTED FROM .END. STATEMENT.	The word END is not followed by DECLARATIVES. END DECLARATIVES is assumed.
205	PERIOD OMITTED AFTER .END DECLARATIVES..	The words END DECLARATIVES are not followed by a period. The error is ignored.
206	SOURCE PROGRAM ENDS IN DECLARATIVES.	The end of the source program occurs in the Declaratives area. Fatal.
207	DATANAME MUST FOLLOW .WITH POINTER. PHRASE.	A STRING or UNSTRING statement contains an invalid WITH POINTER phrase. Fatal.
210	.OVERFLOW. MUST FOLLOW .ON. IN THIS STATEMENT.	A STRING or UNSTRING statement contains an invalid ON OVERFLOW phrase. Fatal.
211	ILLEGAL SENDING FIELD DATANAME IN .UNSTRING.	The sending field of an UNSTRING statement has invalid class. Fatal.
212	ILLEGAL SYNTAX IN .UNSTRING. STATEMENT.	An UNSTRING statement has invalid syntax. Fatal.
213	MULTIPLE SIGN CLAUSES ON THIS ITEM.	More than one SIGN clause appears in a data description. (SEPARATE must follow LEADING or TRAILING.) The second clause is used.
214	ILLEGAL SYNTAX IN COBOL EXPRESSION.	The compiler detects a syntax error of a general nature in the COBOL expression being compiled. Fatal.
215	SIGN CLAUSE ON NONNUMERIC ITEM.	A SIGN clause appears in a non-numeric data description. The SIGN clause is ignored.

216 SIGN CLAUSE APPLIED   A SIGN clause appears in
TO NONDISPLAY ITEM.   a numeric data description
with usage other than
DISPLAY. The SIGN clause is
ignored.

217 SIGN CLAUSE APPLIED   A SIGN clause appears in a
TO UNSIGNED DATAITEM.   numeric data description
which has no "S" in its
PICTURE string. The SIGN
clause is ignored.

220 ILLEGAL DELIMITING DATA   An UNSTRING statement
ITEM IN .UNSTRING.   references an invalid
delimiter. Fatal.

221 .ALL. FIGURATIVE CONSTANT An UNSTRING statement
ILLEGAL IN .UNSTRING.   contains an ALL literal
reference. Fatal.

222 ILLEGAL RECEIVING DATANAME An UNSTRING statement
IN .UNSTRING.   references a receiving data
item which is invalid.
Fatal.

223 .DELIMITED. CLAUSE REQUIRED An UNSTRING statement
IN THIS .UNSTRING.   contains no DELIMITED BY
clause. Fatal.

224 DATANAME MUST FOLLOW   An UNSTRING statement
.DELIMITER IN. PHRASE.   contains a DELIMITER
IN phrase with an illegal
reference. Fatal.

225 ILLEGAL DATANAME FOLLOWS An UNSTRING statement
.DELIMITER IN. PHRASE.   contains a DELIMITER IN
phrase referencing a data
item which is invalid.
Fatal.

226 DATANAME MUST FOLLOW   An UNSTRING statement
.COUNT IN. PHRASE.   contains a COUNT IN phrase
with an illegal reference.
Fatal.

227 ILLEGAL DATANAME FOLLOWS An UNSTRING statement
.COUNT IN. PHRASE.   contains a COUNT IN phrase
which references a data item
which is invalid. Fatal.

230 DATANAME MUST FOLLOW   An UNSTRING statement
.TALLYING IN. PHRASE.   contains a TALLYING phrase
with an illegal reference.
Fatal.

231 ILLEGAL DATANAME FOLLOWS An UNSTRING statement
.TALLYING IN. PHRASE.   contains a TALLYING
phrase referencing a data
item which is invalid.
Fatal.

232 DATANAME MUST FOLLOW   A valid data-name reference
.INSPECT. VERB.   does not follow the INSPECT
keyword. Fatal.

233 ILLEGAL DATANAME FOLLOWS .INSPECT. VERB.

An INSPECT statement references a data item which is invalid. Fatal.

234 ILLEGAL DATANAME PRECEDES .FOR. IN .INSPECT.

An INSPECT...TALLYING statement references a tally data item which is invalid. Fatal.

235 .FOR. OMITTED IN .INSPECT. STATEMENT

An INSPECT...TALLYING statement has invalid syntax. Fatal.

236 DATANAME MUST FOLLOW .TALLYING. PHRASE.

An INSPECT...TALLYING statement does not reference a tally data-name. Fatal.

237 ILLEGAL WORD FOLLOWS .FOR. IN .INSPECT.

An INSPECT...TALLYING statement does not state a valid search condition. Fatal.

240 DATAITEM OMITTED AFTER .ALL. .LEADING. OR .FIRST.

An INSPECT statement does not reference a valid search argument. Fatal.

241 .ALL. FIGURATIVE CONSTANT ILLEGAL IN .INSPECT.

An ALL literal appears in an INSPECT statement. Fatal.

242 ILLEGAL DATANAME FOLLOWS .ALL. OR .LEADING.

An INSPECT statement does not reference a valid search argument. Fatal.

243 ILLEGAL DATANAME FOLLOWS .BEFORE. OR .AFTER.

An INSPECT statement does not reference a valid delimiter in the BEFORE/ AFTER phrase. Fatal.

244 ILLEGAL DATANAME FOLLOWS .BY.

An INSPECT statement does not reference a valid replacement argument. Fatal.

245 ILLEGAL DATANAME PRECEDES .BY.

An INSPECT statement does not reference a legal data-name or literal preceding the BY phrase. Fatal.

246 DATAITEM OMITTED IN .BEFORE. OR .AFTER. PHRASE.

An INSPECT statement does not reference a legal data-name or literal after the BEFORE or AFTER phrase. Fatal.

247 ILLEGAL SYNTAX IN .INSPECT. STATEMENT.

Both the TALLYING and REPLACING keywords are missing in the INSPECT statement. Fatal.

250 .BY. MUST FOLLOW .CHARACTERS. IN REPLACING LIST.

The INSPECT...REPLACING statement must have CHARACTERS BY phrase completely specified. Fatal.

251 DATAITEM OMITTED AFTER
.BY. IN .INSPECT.

The INSPECT...REPLACING
statement does not reference
a legal data-name or literal
after BY. Fatal.

252 DATAITEM FOLLOWING .BY.
EXCEEDS 1 CHARACTER.

In an INSPECT...REPLACING
statement, either when the
CHARACTERS BY phrase is
specified or when a
figurative constant preceding
the BY keyword of the ALL,
LEADING, or FIRST phrase is
specified, the data-name or
literal after the BY keyword
must be defined as one
character in length. Fatal.

253 DATAITEMS BEFORE AND AFTER
.BY. UNEQUAL IN SIZE.

In an INSPECT...REPLACING
statement, the data items
before and after the BY
keyword of the ALL, LEADING,
or FIRST phrase must be equal
in length. Fatal.

254 .BEFORE. OR .AFTER. OPERAND
EXCEEDS 1 CHARACTER.

In an INSPECT...REPLACING
CHARACTERS BY statement, the
data-name or literal follow-
ing the BEFORE or AFTER
keyword must be one character
in length. Fatal.

255 ILLEGAL WORD FOLLOWS
.REPLACING. IN INSPECT..

A legal keyword was not
recognized following
REPLACING in the INSPECT
statement. Fatal.

256 .BY. OMITTED AFTER REPLACING
COMPARISON OPERAND.

The keyword BY is omitted in
the ALL, LEADING, or FIRST
phrase where it separates
operands to be compared.
Fatal.

257 TOO MANY RIGHT PARENTHESES IN
COBOL EXPRESSION.

The compiler detects an
excess of right parentheses
in the COBOL expression being
compiled. Parentheses must be
specified in balanced pairs
i.e., a left parenthesis for
each right parenthesis speci-
fied. This COBOL expression
is considered syntactically
incorrect. Fatal.

260 TOO MANY LEFT PARENTHESES IN
COBOL EXPRESSION.

The compiler detects an
excess of left parentheses in
the COBOL expression being
compiled. Parentheses must
be specified in balanced
pairs i.e., a right paren-
thesis for each left paren-
thesis specified. This COBOL
expression is considered syn-
tactically incorrect. Fatal.

261 MISSING OPERAND IN ARITHMETIC
    EXPRESSION.

An operand is omitted in a
COBOL arithmetic expression.
The COBOL expression is con-
sidered syntactically in-
valid. Fatal.

262 ILLEGAL OPERAND IN ARITHMETIC
    EXPRESSION.

The compiler detects an
illegal operand in a COBOL
arithmetic expression. The
class or usage of the operand
may be invalid in the context
as a reference in an arith-
metic expression. Fatal.

263 NONINTEGER EXPONENT FOUND IN
    COBOL EXPRESSION.

The compiler detects a
non-integer, numeric exponent
in a COBOL arithmetic
expression. The arithmetic
expression is considered
invalid. Fatal.

264 SUBJECT OMITTED IN
    CLASS CONDITION.

The compiler detects the
omission of the subject in a
NUMERIC or ALPHABETIC class
condition. The class
condition is considered
syntactically invalid. Fatal.

265 SUBJECT OMITTED IN SIGN
    CONDITION.

The compiler detects the
omission of the subject in a
sign condition. The sign
condition is considered
syntactically invalid.
Fatal.

266 OPERAND MISSING IN
    COMPLEX CONDITION.

The compiler detects the
omission of an operand in an
AND or OR complex condition.
The COBOL condition
expression.is considered
syntactically invalid. Fatal.

267 INVALID OPERAND IN
    COMPLEX EXPRESSION.

The compiler detects a
complex condition operand
which, in turn, is neither a
simple condition, combined
condition, nor a complex
condition. Fatal.

270 ILLEGAL SYNTAX IN NEGATED
    SIMPLE CONDITION.

The compiler detects illegal
syntax in a COBOL negated
simple condition. Fatal.

271 INVALID NEGATED SIMPLE
    CONDITION.

The compiler detects the
application of the NOT
keyword to an invalid simple
condition. Fatal.

272 ILLEGAL SYNTAX IN
    COMPUTE. STATEMENT.

The compiler detects illegal
syntax in a COMPUTE
statement. The left side of
the assignment symbol, or
the assignment symbol itself
may have been omitted. Fatal.

273   .AT END. ILLEGAL FOR   Either the file has ACCESS
       RANDOM .READ   RANDOM or DYNAMIC without the
                                         word NEXT being included in
                                         the READ statement. In either
                                         case, the AT END clause is
                                         illegal and is treated as an
                                         INVALID KEY clause.

274   INVALID KEY ILLEGAL FOR   Either the file has ACCESS
       SEQUENTIAL .READ.   SEQUENTIAL or the READ
                                           statement contains the word
                                         NEXT. In either case, the
                                         INVALID KEY clause is illegal.
                                         It is treated as an AT END
                                         clause.

275   INDEX DATA ITEM ILLEGAL   An index data item is used as
       AS INDEX ON TABLE.   an index on a table. The
                                           index data item reference is
                                         ignored. A literal subscript
                                         of 1 replaces the index
                                         data item reference.

276   INDEX NAME NOT DEFINED   An index-name used in a sub-
       FOR THIS TABLE.   script list either is not
                                           defined for this table or
                                         appears in the wrong logical
                                         position of the subscript
                                         list for this table. The
                                         index-name is ignored and a
                                         default value of 1 is assumed
                                         as the subscript.

277   RELATIVE INDEX IS INVALID.   The literal component of a
                                         relative index is zero or
                                         less in value or is an
                                         invalid word. Relative
                                         indexing is ignored and the
                                         index-name only is used.

300   PROGRAM NAME OMITTED   The program-name is omitted
       AFTER .CALL. VERB   after the key word CALL in
                                           a CALL statement. This is
                                         syntactically invalid. Fatal.

301   LINAGE 0 OR LESS   The LINAGE clause must
       THAN FOOTING.   specify a page body of at
                                           least one line and that page
                                         body size must be equal to or
                                         greater than the footing size
                                         specified in the FOOTING
                                         phrase.

302   FILE CLOSED BUT NOT   A CLOSE statement was seen
       OPENED.   for a file that was not
                                           OPENed in this program.
                                         Fatal.

303   PRINT CONTROL ON NON SEQUENTIAL   An APPLY PRINT-CONTROL clause
       FILE.   IGNORED.   references a file which does
                                           not have SEQUENTIAL
                                         organization. The file-name
                                         is ignored in the APPLY
                                         clause.

304 DATANAME OMITTED IN .KEY
IS. PHRASE.

The KEY IS phrase of the START
statement is not followed by a
data-name. The prime RECORD
KEY data-name is assumed
present.

305 SECTION OR PARAGRAPH
NAME MISSING.

The Procedure Division does
not start with a section or
paragraph name or a section
header is not followed by a
paragraph name. Fatal.

306 .PROCEDURE. MISSING IN .USE.
STATEMENT. ASSUMED.

The keyword PROCEDURE is
missing in the USE
statement. It is assumed and
processing is continued.

307 .START. WITHOUT .INVALID
KEY. OR .USE.

The INVALID KEY option is
missing from the START
statement or no USE proce-
dure is declared for the
referenced file. Fatal.

310 .WRITE. WITHOUT .INVALID
KEY. OR .USE.

THE INVALID KEY option
is missing from the WRITE
statement or no USE
procedure is declared for
the referenced file. Fatal.

311 DATA DIVISION MUCH TOO
LARGE.

Too much buffer space is
being used for the
files in this program. Too
many files are declared to be
OPEN simultaneously. Fatal.

312 .REDEFINES. SPECIFIES INVALID
REDEFINITION.

The compiler detects the
invalid application of
REDEFINES to a data
description entry which
contributes new character
positions between the data
description entry con-
taining the REDEFINES clause
and the item being redefined.
Also, the source of error may
be the definition of another
data description entry with a
lower level number appearing
between the data description
entry containing the
REDEFINES clause and the item
being redefined. The compiler
ignores the REDEFINES clause
and continues processing the
data description entry.

313 ILLEGAL TO REDEFINE ANOTHER
REDEFINITION.

The REDEFINES clause speci-
fies the redefinition of a
data item whose data
description entry contains a
REDEFINES clause itself. This
is syntactically invalid. The
compiler ignores the
REDEFINES clause and continues

processing of the data
description entry.

314 ILLEGAL TO REDEFINE A
COBOL TABLE.

The REDEFINES clause speci-
fies the redefinition of a
data item whose data descrip-
tion entry contains an OCCURS
clause. This is syntactically
invalid. The compiler ignores
the REDEFINES clause and
continues processing of the
data description entry.

315 .REDEFINES. APPLIED TO
VARIABLE LENGTH DATAITEM.

The compiler detects an
application of the REDEFINES
clause to a data item whose
length is variable at run-
time. This data item is
variable in length because it
has a subordinate data item
whose data description entry
contains an OCCURS DEPENDING
ON clause. The application of
the REDEFINES clause to such
a data item is syntactically
invalid. The compiler ignores
the REDEFINES clause and
continues processing of the
data description entry.

316 .OCCURS DEPENDING ON. ILLEGAL
IN REDEFINITION.

The compiler detects a
redefinition which contains a
data description entry
declared with an OCCURS
DEPENDING ON clause. The
OCCURS DEPENDING ON clause
causes the redefinition to
contain a data item whose
length is variable at run-
time. This is syntactically
invalid. The DEPENDING ON
phrase is ignored and pro-
cessing continues.

317 PICTURE EXCEEDS 30
CHARACTERS. PIC X ASSUMED.

The unexpanded PICTURE string
exceeds 30 characters in
length. This is syntactically
invalid. The compiler ignores
the user-supplied PICTURE
and declares the data-name
alphanumeric with a
"PICTURE X" declaration.

320 FILENAME MUST FOLLOW .CLOSE
VERB.

The data item following the
CLOSE verb was not a file-
name. Fatal.

321 .NO. MUST FOLLOW .WITH.
IT IS ASSUMED.

The keyword NO is missing in
the WITH NO REWIND phrase of
the CLOSE statement. NO is
assumed present.

322 .REWIND. MUST FOLLOW .NO.
IT IS ASSUMED.

The WITH NO REWIND phrase of
the CLOSE statement must be

completely specified. It is
assumed present.

323 .REMOVAL. MUST FOLLOW .FOR.
IT IS ASSUMED.

The FOR REMOVAL phrase of the
CLOSE statement must be
completely specified. It is
assumed present.

324 .LOCK. OMITTED AFTER .WITH.
IT IS ASSUMED.

The keyword WITH in a CLOSE
statement is recognized but
is not followed by one of the
keywords NO or LOCK. The
WITH LOCK phrase is assumed
present.

325 DATANAME SPECIFIED WHERE
FILENAME EXPECTED.

The name used in an I/O verb
to reference a file was not a
file name but was some other
data-name. Fatal.

326 FILENAME MUST FOLLOW
MODE SPEC. IN .OPEN..

The OPEN statement does not
reference a valid file name
where a file-name reference
is expected. Fatal.

327 ILLEGAL MODE SPECIFIED
AFTER .OPEN. VERB.

One of the OPEN mode keywords
INPUT, OUTPUT, I-O, or EXTEND
is required immediately after
the OPEN verb. None of these
four keywords was
recognized. Fatal.

330 .END. MUST FOLLOW .AT..
IT IS ASSUMED.

The keyword END was omitted
in the AT END phrase of the
READ statement. The AT END
phrase is assumed present.

331 FILENAME MUST FOLLOW
.READ. VERB.

Either the file-name was
omitted following the READ
verb or the data item
following the READ verb is
not a valid file-name
reference. Fatal.

332 DATANAME OMITTED AFTER .INTO.
IN .READ.

The data-name reference
following the INTO keyword
of the READ statement was
omitted. Fatal.

333 RECORDNAME MUST FOLLOW
.WRITE. OR .REWRITE.

The 01 record-name reference
immediately following the
WRITE or REWRITE verb was
omitted. Fatal.

334 STATEMENT IGNORED DUE
TO ILLEGAL RECORDNAME.

The data-name immediately
following the WRITE or
REWRITE verb is not a valid
01 record-name reference.
Fatal.

335 .ADVANCING. OPTION OMITTED
IN .WRITE. 1 ASSUMED.

A data-name reference, numer-
ic integer literal reference,
or the keyword PAGE was not
recognized in the BEFORE/

AFTER ADVANCING phrase of the
WRITE statement. A numeric
integer literal value of 1 is
assumed.

336 .EOP. MUST FOLLOW .AT..
    IT IS ASSUMED.

The keyword EOP was omitted
in the AT EOP phrase of the
WRITE statement. The AT EOP
phrase is assumed present.

337 DATANAME OMITTED AFTER
    .FROM.

The data-name reference
following the FROM keyword of
the WRITE or REWRITE
statement was omitted.
Fatal.

340 .ADVANCING. INTEGER TOO BIG.
    TRUNCATED TO 63.

The numeric integer in the
BEFORE/AFTER ADVANCING phrase
of the WRITE statement is
greater than 63. 63
is assumed.

341 .NO REWIND. ILLEGAL WITH
    .IO. OR .EXTEND. MODE.

An OPEN statement with the
I-O or EXTEND mode specified
cannot have the NO REWIND
phrase also specified.
Fatal.

342 ILLEGAL .ADVANCING. DATANAME.
    1 IS ASSUMED.

The data-name in the BEFORE/
AFTER ADVANCING phrase of the
WRITE statement is not an
elementary numeric integer
data-name reference. A
numeric integer literal value
of 1 is assumed.

343 FILENAME MUST FOLLOW
    .DELETE. VERB.

Either the file-name was
omitted following the DELETE
verb or the data item
following the DELETE verb is
not a valid file-name
reference. Fatal.

344 FILENAME MUST FOLLOW
    .START. VERB.

Either the file name was
omitted following the START
verb or the data item
following the START verb is
not a valid file name
reference. Fatal.

345 .LESS. OMITTED AFTER .NOT.
    IN .START. ASSUMED.

The keyword LESS is omitted
after NOT in the relational
condition of the START
statement. LESS is assumed
present.

346 DATANAME OMITTED IN .KEY
    IS. PHRASE. ASSUMED.

The RELATIVE KEY data-name
for the referenced file was
omitted in the KEY IS phrase
of the START statement. The
RELATIVE KEY data-name is
assumed present.

347   RELATIONAL WORD OMITTED   None of the relational
        AFER .KEY IS. PHRASE.   keywords EQUAL, GREATER, or
                                         NOT was recognized following
                                         the KEY IS phrase of the
                                         START statement. Fatal.

350   TERMINATOR IGNORED IN   A clause is terminated by a
        .IO CONTROL. PARAGRAPH.   period, but a header does not
                                           follow in Area A. The period
                                           is ignored. The compiler
                                           assumes it is still in the
                                           I-O-CONTROL paragraph.

351   TERMINATOR IGNORED IN   A clause is terminated by a
        .SPECIAL NAMES. PARAPGH.   period, but is not followed
                                           by a header in Area A. The
                                           period is ignored, and the
                                           compiler continues processing
                                           the SPECIAL-NAMES paragraph.

352   .NATIVE. MISSING IN   The alphabet-name clause does
        SPECIALNAMES CLAUSE.   not contain NATIVE or
                                           STANDARD-1. The alphabet-
                                           name clause is ignored.

353   SYNTAX ERROR IN .OBJECT   The OBJECT-COMPUTER paragraph
        COMPUTER. PARAGRAPH.   contains an unrecognizable
                                           word. Recovery is made by
                                           scanning over all words until
                                           a word is found in area A.

354   TERMINATOR OMITTED IN   The OBJECT-COMPUTER paragraph
        .OBJECT COMPUTER. PARA.   is not terminated by a
                                           period. Recovery is made by
                                           scanning over all words until
                                           a word is found in area A.

355   DATANAME FOLLOWING .KEY IS.   The data-name following the
        PHRASE IS ILLEGAL   KEY IS phrase of the START
                                           statement is not a RECORD KEY
                                           associated with the referenced
                                           indexed file, nor is it
                                           subordinate to a RECORD KEY
                                           whose left-most character
                                           position corresponds to its
                                           own left-most character
                                           position. FATAL.

356   INVALID USAGE ON   The level 88 condition
        CONDITIONAL VARIABLE.   variable does not have
                                           DISPLAY or COMPUTATIONAL
                                           usage.

357   ILLEGAL SEPARATOR IN   An illegal character was
        COBOL STATEMENT. IGNORED.   detected between two
                                           consecutive words of a COBOL
                                           statement. The illegal
                                           character is ignored.

360   ILLEGAL CHARACTER FOUND   Illegal characters were found
        WITHIN A COBOL WORD.   in an alphanumeric COBOL
                                           word, not within an
                                           alphanumeric literal. The
                                           illegal characters are

replaced by dollar signs in
the internal representation
of the COBOL word.

361 UNRECOGNIZABLE TEXT FOUND
IN COBOL STATEMENT.

In scanning the source text,
the compiler was unable to
recognize an alphanumeric
COBOL word (i.e., a keyword
or user-defined word), an
alphanumeric literal, or a
numeric literal. The error
is not internally corrected
and usually will propagate
further error messages.

362 COBOL WORD BEGINS WITH
OR ENDS IN HYPHEN.

In attempting to recognize a
keyword or user-defined word,
the compiler has detected
that the COBOL word begins
or ends with a hyphen
character.

363 NONNUMERIC LITERAL TOO LONG.
TRUNCATED TO MAX.

An alphanumeric literal
greater than 132 characters
in length is detected. The
literal is truncated on right,
retaining the first 132 char-
acters as the literal.

364 COBOL SOURCE LINE TOO LONG.
TRUNCATED TO MAX.

The indicated COBOL source
line contains more than 65
characters in terminal
format. The excess
characters are ignored and
only those characters in the
printed COBOL source line are
retained.

365 .BY. OMITTED IN REPLACING
OPTION. COPY IGNORED.

The keyword BY was not found
in this COPY...REPLACING
statement. The statement
will be ignored.

366 TERMINATOR OMITTED IN
.COPY. IT IS ASSUMED.

The required period
terminating the COPY
statement is omitted. It is
assumed present.

367 .LINAGE. CLAUSE DATANAME
MUST BE AN INTEGER.

A data-name referenced in the
LINAGE clause of the FILE
SECTION is defined in the
WORKING-STORAGE SECTION with
decimal places.

370 .LINAGE.CLAUSE DATANAME
MUST BE UNSIGNED.

A numeric data-name refer-
enced in the LINAGE clause of
the FILE SECTION is defined
in the WORKING-STORAGE SEC-
TION as a signed data item.

371 POSSIBLE HIGH ORDER
RECEIVING FIELD TRUNCATION.

Truncation of high order
information during a MOVE
or an arithmetic operation
upon a receiving field is

possible. This is an
observation only.

372 POSSIBLE LOW ORDER
RECEIVING FIELD TRUNCATION.

Truncation of low order
information during a MOVE
or an arithmetic operation
upon a receiving field is
possible. This is an
observation only.

373 PD HEADER NOT FOLLOWED
BY AN AREA A WORD.

The word following the
PROCEDURE DIVISION header
does not begin in Area A. A
scan is made over all words
until a word is found in Area
A.

374 OPEN OPTIONAL FILES ONLY
IN .INPUT. MODE.

An OPTIONAL file can be
OPENed in INPUT mode only.
The compiler assumes that the
OPTIONAL file is OPENed in
INPUT mode.

375 EXPECTED .FILE STATUS.
DATANAME NOT DEFINED.

A data-name referenced in a
FILE STATUS phrase of a
SELECT clause in the FILE-
CONTROL paragraph is not
defined in the WORKING-
STORAGE SECTION of the DATA
DIVISION.

376 EXPECTED .VALUE OF ID.
DATANAME NOT DEFINED.

The data-name referenced in a
VALUE OF ID clause of an FD
is not defined in the
WORKING-STORAGE SECTION of
the DATA DIVISION. Fatal.

377 EXPECTED .LINAGE. CLAUSE
DATANAME NOT DEFINED.

A data-name referenced in the
LINAGE clause of the FILE
SECTION is not defined in the
WORKING-STORAGE SECTION of
the DATA DIVISION.

400 .RELATIVE KEY. DATANAME
HAS INVALID CLASS.

A data-name referenced in a
RELATIVE KEY phrase of a
SELECT clause in the FILE-
CONTROL paragraph is defined
in the WORKING-STORAGE
SECTION with non-numeric
class.

401 .RELATIVE KEY. DATANAME
HAS INVALID USAGE.

A data-name referenced in a
RELATIVE KEY phrase of a
SELECT clause must be defined
with COMPUTATIONAL or
DISPLAY usage in the WORKING-
STORAGE SECTION.

402 .RELATIVE KEY. DATAITEM
IS TOO LONG.

A numeric integer data-name
referenced in a RELATIVE KEY
phrase is defined with more
than eight digits of
precision in the WORKING-
STORAGE SECTION.

403 .RELATIVE KEY. DATANAME
MUST BE AN INTEGER.

A numeric data-name refer-
enced in a RELATIVE KEY
phrase is defined in the
WORKING-STORAGE SECTION with
decimal places.

404 .FILE STATUS. DATANAME
HAS INVALID CLASS.

A data-name referenced in a
the FILE STATUS phrase of a
SELECT clause must be defined
in with DISPLAY usage in the
WORKING-STORAGE SECTION.

405 .FILE STATUS. DATA NAME
HAS INVALID USAGE.

A data-name referenced in a
FILE STATUS phrase of a
SELECT clause is defined with
DISPLAY USAGE in the WORKING-
STORAGE SECTION.

406 LENGTH OF .FILE STATUS.
DATAITEM IS ILLEGAL.

An alphanumeric data-name
referenced in a FILE STATUS
phrase of a SELECT clause
must be defined as an
alphanumeric variable
consisting of two characters
in the WORKING-STORAGE
SECTION.

407 .VALUE OF ID. DATANAME
HAS INVALID CLASS.

A data-name referenced in a
VALUE OF ID clause of an FD
is defined in the WORKING-
STORAGE SECTION with non-
alphanumeric class.

410 .VALUE OF ID. DATANAME
HAS INVALID USAGE.

A data-name referenced in a
VALUE OF ID clause of an FD
must be defined with DISPLAY
usage in the WORKING-STORAGE
SECTION.

411 LENGTH OF .VALUE OF ID.
DATAITEM IS ILLEGAL.

An alphanumeric data-name
referenced in a VALUE OF ID
clause of an FD must be
defined in the
WORKING-STORAGE section as
alphanumeric variable whose
length L falls in the range
$9 <= L <= 40$ characters.

412 .LINAGE. CLAUSE DATANAME
HAS INVALID CLASS.

A data-name referenced in
the LINAGE clause of the. FILE
SECTION is defined in the
WORKING-STORAGE SECTION with
non-numeric class.

413 .LINAGE. CLAUSE DATANAME
HAS INVALID USAGE.

A data-name referenced in the
LINAGE clause of the FILE
SECTION must be defined with
COMPUTATIONAL USAGE in the
WORKING-STORAGE SECTION.

414 INVALID RECEIVING OPERAND
IN .SET.. IGNORED.

A receiving operand of a
SET statement is invalid.
Fatal.

415 NO RECEIVING OPERAND
SPECIFIED IN .SET..

No receiving operands are
specified in a SET statement.
Fatal.

416 OMITTED OR ILLEGAL OPERAND
AFTER .TO. IN .SET..

A SET statement has no valid
sending operand. Fatal.

417 ILLEGAL SYNTAX IN
.SET. STATEMENT.

The words TO, UP or DOWN do
not follow the receiving
operands of a SET statement.
Fatal.

420 .BY. MUST FOLLOW .UP.
OR .DOWN.. ASSUMED.

The keyword BY does not
follow the word UP or DOWN in
a SET statement. BY is
assumed present.

421 OMITTED OR ILLEGAL OPERAND
AFTER .BY. IN .SET..

The operand following the UP
BY or DOWN BY phrase in a SET
statement is invalid or
omitted. Fatal.

422 NO OPERANDS SPECIFIED
IN .DISPLAY.

No operands to be displayed
were recognized by the
compiler in this DISPLAY
statement. Fatal.

423 SETTING INDEX NAME OUT
OF RANGE. .SET. IGNORED.

A SET statement is attempting
to set an index name using a
literal that is too large.
Fatal.

424 .IF. TRUE PATH OMITTED.
ASSUME .NEXT SENTENCE.

The true path code is omitted
from the IF statement. NEXT
SENTENCE is assumed as the
true path of the IF
statement.

425 CONFLICTING SIGN SYMBOLS
IN PICTURE STRING.

The compiler recognizes
both the + and - sign symbols
in this PICTURE string. The
compiler ignores the user-
supplied PICTURE and declares
the data-name alphanumeric
with a "PICTURE X"
declaration

426 ZERO SUPPRESSION CONFLICTS
IN PICTURE STRING.

The compiler recognizes
both the Z and * zero
suppression symbols in this
PICTURE string. The compiler
ignores the user-supplied
PICTURE and declares the
data-name alphanumeric with a
"PICTURE X" declaration.

427 ILLEGAL CHARACTER IN
THE PICTURE STRING.

A character which is not in
the PICTURE string character
set is recognized in this
PICTURE by the compiler. The
compiler ignores the user-
supplied PICTURE and declares
the data-name alphanumeric
with a "PICTURE X"
declaration.

430 .BLANK WHEN ZERO. CONFLICTS
WITH ZERO SUPPRESS.

A BLANK WHEN ZERO clause is
recognized with a zero
suppression field specified
in the PICTURE string. The
compiler ignores the BLANK
WHEN ZERO clause and
continues with its
processing.

431 PARENTHESIZED SPECIFIER
EXCEEDS 18 DIGITS

The specification contained
inside parentheses of a
PICTURE string exceeds 18
digits in length. The
compiler ignores the user-
supplied PICTURE and declares
the data-name alphanumeric
with a "PICTURE X"
declaration.

432 SPECIFIER MISSING INSIDE
PARENTHESES.

The specification contained
inside parentheses of a
PICTURE string is missing.
The compiler ignores the
user-supplied PICTURE and
declares the data-name
alphanumeric with a "PICTURE
X" declaration.

433 ILLEGAL SYMBOL PRECEDES
LEFT PAREN. IN PICTURE.

The compiler recognizes
an S, V, CR, DB, or "."
character preceding a left
parenthesis in a PICTURE
string. The error is ignored
and processing continues.

434 TERMINATOR OMITTED IN
.NOTE. PARAGRAPH

A NOTE paragraph that does
not end with a period
was detected.

435 INVALID OPERAND IN .VARYING.
OR .AFTER. PHRASE.

The expected operand is not a
valid name reference in the
VARYING or AFTER phrase of
this PERFORM VARYING
statement. Fatal.

436 INVALID OPERAND IN .FROM. OR
.BY. PHRASE.

The FROM or BY phrase of this
PERFORM VARYING statement
does not contain a valid
operand reference. Fatal.

437 TOO MANY .AFTER. PHRASES IN
.PERFORM. STATEMENT.

The compiler detects more
than two AFTER phrases in the
PERFORM VARYING statement
being compiled. This is
syntactically invalid.
Fatal.

440 .FROM. OR .BY. OR .UNTIL.
MISSING IN PERFORM.

The compiler detects the
omission of the keywords
FROM, BY, or UNTIL in the
PERFORM VARYING statement.
Fatal.

441 ILLEGAL CONDITION EXPRESSION
IN THE PERFORM.

The compiler detects an
invalid condition expression
in the PERFORM statement.
Fatal.

442 NONPOSITIVE LITERAL IN .FROM.
OR .BY. PHRASE.

The compiler detects a
non-positive, numeric integer
literal in this PERFORM
statement. This is
syntactically invalid.
Fatal.

443 INVALID RELATION CONDITION IN
.SEARCH ALL.

The compiler detects either a
syntax error or an
invalid operand in the
restricted form of a relation
condition in the SEARCH ALL
statement. Fatal.

444 NONINTEGER DATA CONFLICTS
WITH INDEXNAME USAGE.

The compiler detects a
non-integer data item
reference in a PERFORM
VARYING statement in which
the VARYING, AFTER, and/or
FROM phrase contains an
index-name reference. This is
syntactically invalid. Fatal.

445 IMPLICIT REFERENCE TO BAD
CONDITION VALUES.

Through a reference to a
condition-name, the compiler
detects a reference to an
associated condition-value
which is improperly declared
in the Data Division. The
compiler considers this to be
syntactically invalid.
Fatal.

446 IMPLICIT REFERENCE TO BAD
CONDITION VARIABLE.

Through a reference to a
condition-name, the compiler
detects that the associated
condition-variable is
improperly declared in the
Data Division. The compiler
considers this to be
syntactically invalid. Fatal.

447 TOO MANY NAMES IN COBOL
PROGRAM. RECOMPILE.

The COBOL program being
compiled has too many data-
names or procedure-names.
This condition has caused a
compiler table to overflow
with the resultant action of
aborting the compilation. The
user is advised to recompile
the program using the
"/SYM:N" switch to get more
space for the compiler
symbol tables.

450 REFERENCE TO UNDEFINED
DATANAME. IGNORED.

The COBOL statement being
compiled contains a reference
to an undefined data-name.
The compiler considers this

to be syntactically invalid and ignores the reference. This diagnostic may be issued in conjunction with other diagnostics for the erroneous statement.

451 QUALIFIED REFERENCE ILLEGAL IN THIS CONTEXT.

The compiler detects a qualified reference in a context in which an unqualified reference is required. The compiler permits the qualified reference in this context and continues with the compilation of the statement containing the reference.

452 QUALIFIER OMITTED IN QUALIFIED REFERENCE.

A data-name is omitted after the keyword OF or IN in a qualified reference in the COBOL statement being compiled. The reference is ignored. This diagnostic may be issued in conjunction with other diagnostics for the statement in error.

453 TOO MANY QUALIFIERS IN QUALIFIED REFERENCE.

The compiler detects more than 48 qualifiers in a qualified reference. The excess qualifiers are ignored in the reference.

454 UNDEFINED QUALIFIER IN QUALIFIED REFERENCE.

The compiler detects a qualified reference one of whose qualifiers is a reference to an undefined data-name. The compiler considers this to be syntactically invalid and ignores the entire qualified reference. This diagnostic may be issued in conjunction with other diagnostics for the erroneous statement containing the reference.

455 COBOL STATEMENT CONTAINS AMBIGUOUS REFERENCE.

The compiler detects a reference to COBOL data which is not uniquely referenceable through qualification. The compiler uses a reference to COBOL datum which satisfies the reference in the text of the COBOL program. This diagnostic may be issued in conjunction with other diagnostics for the statement in error.

456 DATANAME REFERENCE EXPECTED IN THIS CONTEXT.

The compiler detects a reference to a COBOL datum

which is not alphabetic, numeric, alphanumeric, alphanumeric-edited, or numeric-edited. The context of this reference requires that the reference be to one of these classes of data items.
The compiler considers the referenced item to be syntactically invalid. This diagnostic may be issued in conjunction with other diagnostics for the statement in error.

457 ILLEGAL REFERENCE DETECTED IN THIS CONTEXT.

The compiler detects a reference to a COBOL datum which is invalid in the context of its usage. The compiler considers the referenced item to be syntactically invalid. This diagnostic may be issued in conjunction with other diagnostics for the statement in error. Fatal.

460 PARENTHESIZED SPECIFIER LARGER THAN 4095.

The specification contained inside parentheses of a PICTURE string is larger than 4095 in value. The specifier exceeds an implementation limitation of 4095. The compiler assumes 4095 and continues with the processing of the PICTURE string.

461 EXTRA OPENING QUOTE ON LITERAL IS IGNORED.

The compiler detects a superfluous quote at the beginning of a non-numeric literal specification. The compiler ignores the extra quote and continues with the processing of the non-numeric literal.

462 PROGRAM NAME MUST BE A NONNUMERIC LITERAL

The program-name literal following the key word CALL is not a nonnumeric literal. This is syntactically invalid. Fatal.

464 LITERALS ARE ILLEGAL IN ARGUMENT LIST OF .CALL..

Literals are not allowed in the argument list of a CALL statement. Fatal.

465 ARGUMENT LIST OMITTED AFTER .USING. IN .CALL..

The required argument list is missing after the key word USING in the CALL statement. Fatal.

470 ILLEGAL SYNTAX IN .CODE SET. CLAUSE. IGNORED.

A valid alphabet-name reference is omitted in the

CODE-SET clause. The compiler ignores the CODE-SET clause and continues to process the remainder of the FD.

471   DATANAME IN .KEY IS. PHRASE
      NOT ALPHANUMERIC.

The data-name following the KEY IS phrase in a START statement referencing an indexed file must be alphanumeric. FATAL.

472   .RECORD KEY. DATAITEM
      LENGTH GREATER THAN 255.

A data-name referenced in a RECORD KEY or ALTERNATE RECORD KEY phrase of a SELECT clause in the FILE-CONTROL paragraph must be defined in the FILE SECTION as an item whose length is less than or equal to 255.

473   DATANAME IN .KEY IS PHRASE
      IS SUBSCRIPTED OR INDEX

The data-name following the KEY IS phrase in a READ or START statement referencing an indexed file must not be subscripted or index. Fatal.

474   .RECORD KEY. DATAITEM MUST
      NOT BE A COBOL TABLE

A data-name referenced in a RECORD KEY or ALTERNATE RECORD KEY phrase of a SELECT clause in the FILE-CONTROL paragraph must not be defined in the FILE SECTION with an OCCURS clause or subordinate to an item with an OCCURS clause.

475   .RECORD. OMITTED FROM
      .ALTERNATE RECORD. ASSUMED.

The reserved word RECORD is missing from the ALTERNATE RECORD KEY clause. The error is ignored.

476   UNDEFINED .ALTERNATE RECORD
      KEY. DATANAME.

The data-name given in an ALTERNATE RECORD KEY clause has not been defined in the Data Division.

477   .ALTERNATE RECORD KEY.
      CLAUSES ARE SEPARATED

In the SELECT statement the ALTERNATE RECORD KEY clauses are interleaved among the other clauses. The ALTERNATE RECORD KEY clauses should follow one another with no intervening clauses. This error is ignored.

500   LINKAGE SECTION ITEM
      APPEARS TWICE IN .USING.

A LINKAGE SECTION data item must not appear more than once in the USING phrase of a PROCEDURE DIVISION USING header. FATAL.

501   ILLEGAL .SEGMENT-LIMIT.
      VALUE. IGNORED

The segment-limit is either not a numeric literal or a numeric literal whose value is outside of allowed segment-limit range.

502 INTEGER 1 BEYOND AREA A
    TREATED AS LEVEL NUMBER.

An 01 level item was detected beyond Area A and accepted as if in Area A.

503 MULTIPLE PICTURES FOR
    SAME ITEM. LAST USED.

A data item has more than 1 PICTURE clause. The compiler used the last PICTURE clause specified.

504 CLOSING PARENTHESIS MISSING
    IN PICTURE.

The right parenthesis is missing in the PICTURE string. The compiler uses the last four digits of the PICTURE string.

505 OT A SUBPROGRAM .PROGRAM.
    IGNORED.

An EXIT PROGRAM has been detected, but the COBOL program being compiled is not a subprogram. Because EXIT PROGRAM is meaningful only in a subprogram, the word PROGRAM is ignored, and the statement is treated as if it were a simple EXIT statement.

506 EXPANDED PICTURE STRING TOO
    LONG. PIC X ASSUMED.

The process of expanding a PICTURE string specification produces a string which exceeds implementation limitation. The compiler ignores the user-supplied PICTURE and declares the data-name with a "PICTURE X" declaration.

507 SPECIFIER OMITTED BEFORE
    LEFT PAREN. IN PIC.

The first character of a PICTURE string is a left parenthesis. The compiler ignores the user-supplied PICTURE and declares the data-name alphanumeric with a "PICTURE X" declaration.

510 SECTION NO. GREATER THAN
    49 TREATED AS 49.

A segment number greater than 49 follows the word SECTION. The segment is treated as if it were 49.

511 INVALID ITEM LENGTH IN
    PARENTHESES OF PICTURE.

The parenthesized length specifier in a PICTURE contains non-numeric characters. The compiler ignores the user-supplied PICTURE and declares the data-name alphanumeric with a "PICTURE X" declaration.

512 VALUE CLAUSE NOT ALLOWED
    IN LINKAGE SECTION.

The VALUE clause cannot appear in data items in the LINKAGE SECTION. The only place the VALUE clause can appear in the LINKAGE SECTION is in a condition name definition.

513 OPERAND IN .USING. MUST BE
    LINKAGE SECTION ITEM.

Only level 01 or 77 LINKAGE
SECTION items may appear in
the USING phrase of a
PROCEDURE DIVISION header.
FATAL.

514 MULTIPLE FLOATING FIELDS
    IN NUMERIC EDIT ITEM.

The PICTURE string contains
multiple floating fields.
The compiler ignores the
user-supplied PICTURE and
declares the data-name
alphanumeric with a "PICTURE
X" declaraton.

515 MULTIPLE ZERO SUPPRESS
    FIELDS IN PICTURE STRING.

Multiple zero suppression
fields are detected in
PICTURE string. The compiler
ignores the user-supplied
PICTURE and declares the
data-name alphanumeric with a
"PICTURE X" declaration.

516 ZERO SUPPRESSION ILLEGAL
    WITH FLOATING FIELD.

The PICTURE string contains
both floating and zero
suppression fields. The
compiler ignores the user-
supplied PICTURE and declares
the data-name alphanumeric
with a "PICTURE X"
declaration.

517 ILLEGAL SYNTAX IN
    PICTURE STRING.

The PICTURE string is not
specified correctly according
to the rules of PICTURE
string syntax. The compiler
ignores the user-supplied
PICTURE and declares the
data-name alphanumeric with a
"PICTURE X" declaration.

520 MULTIPLE DECIMAL POINTS
    IN PICTURE.

The PICTURE string contains
multiple decimal point
specifications (V's, P's, or
periods). The compiler
ignores the user-supplied
PICTURE and declares the
data-name alphanumeric with a
"PICTURE X" declaration.

521 OPERAND IN USING MUST BE
    LEVEL 01 OR 77

Only level 01 or 77 LINKAGE
SECTION items may appear in
the USING phrase of a
PROCEDURE DIVISION header.
FATAL.

522 INVALID USAGE. IGNORED.

The USAGE clause contains an
invalid word. The compiler
ignores the entire USAGE
clause.

523 MULTIPLE USAGE CLAUSES.
    LAST USED.

The defined data-name has
multiple USAGE clauses
specified. The last USAGE
clause specified is used by
the compiler.

524 MULTIPLE OCCURS CLAUSES. LAST USED.

The defined data-name has multiple OCCURS clauses specified. The compiler uses the last OCCURS clause specified.

525 OCCURS SPECIFICATION ERROR. 1 ASSUMED.

The integer entry of the OCCURS clause is either non-numeric or non-integer or does not lie in the range 1 to 4095. The compiler assumes an integer value of 1.

526 DATANAME OMITTED IN DATA DESCRIPTION ENTRY.

The data-name declaration is omitted after a level-number in the data description entry. The compiler supplies a system-defined name and proceeds with the processing of the data description entry. The system-defined name is transparent and, thus, inaccessible to the user.

527 INVALID INDEX NAME. IGNORED.

The compiler did not recognize a valid index name in the INDEXED BY phrase. The compiler ignores the INDEXED BY phrase.

530 USAGE OPTION NOT YET IMPLEMENTED. IGNORED.

The compiler detected COMP-1 in the USAGE clause. This option is not implemented and is ignored. The default USAGE of DISPLAY is used by the compiler.

531 TERMINATOR OMITTED AFTER DATAITEM DESCRIPTION.

A data item description entry in the DATA DIVISION is not terminated by a period. The compiler assumes the period is present and continues processing.

532 INVALID SIGN IN NUMERIC PICTURE.

The sign character S is detected in a position other than the leading character position of a numeric PICTURE string. The compiler ignores the user-supplied PICTURE and declares the data-name alphanumeric with a "PICTURE X" declaration.

533 PICTURE CLAUSE OMITTED ON ELEMENTARY ITEM.

An elementary item is recognized with its PICTURE clause omitted in the description. The compiler declares the data-name alphanumeric with a PICTURE X declaration.

534 NUMERIC ITEM EXCEEDS 18
DIGIT MAX. TRUNCATED.

A numeric field is defined in
this PICTURE with more than
18 digits of precision. The
numeric field is truncated to
18 digits.

535 COMP ITEM EXCEEDS 18
DIGITS. ASSIGN 4 WORDS.

A COMPUTATIONAL data item
exceeds 18 digits in its
specification. The compiler
truncates it and allocates
four words for its run-time
storage.

536 INDEX ITEM HAS
ILLEGAL CLAUSE.

The compiler recognized a
JUSTIFIED, SYNCHRONIZED,
VALUE, PICTURE, or SIGN
clause on a data-item
description which has INDEX
USAGE. This is illegal. The
compiler ignores the
offensive clause.

537 NUMERIC VALUE FOR
DISPLAY ITEM. IGNORED.

The VALUE clause specifies
numeric value initialization
for a non-numeric data-item
which is defined with DISPLAY
USAGE. This is illegal. The
VALUE clause is ignored.

540 VALUE TOO LONG.
TRUNCATED.

The length of the non-numeric
literal in the VALUE clause
is longer than the associated
data-item. The literal is
truncated on the right to fit
in the storage allocated to
the data-item.

541 CLAUSE DUPLICATION. IGNORED.

This clause has been
previously recognized for
this item. The duplicate
clause is ignored.

542 INVALID WORD IN .BLANK
WHEN ZERO.. IGNORED.

The keyword ZERO was not
recognized in the BLANK WHEN
ZERO clause. The entire
clause is ignored.

543 LEVEL NUMS UNEQUAL IN
.REDEFINES. CLAUSE IGNORED.

A REDEFINES clause attempts
to redefine two items of
different level numbers. The
REDEFINES clause is ignored.

544 POSSIBLE OVERLAP OF DEPENDING
ON ITEM AND TABLE

The depending on item and
variable length table are
both defined in the LINKAGE
SECTION. Because LINKAGE
SECTION items are associated
with data items appearing in
a CALL statement, there is
no way at compile time to
insure that the depending on
items and table do not
overlap. The COBOL run-time
OTS does not check for overlap

of the depending on item and
the table during execution.
It is, therefore, your
responsibility to insure
that overlap does not occur.

545 LEVEL ILLEGAL AFTER 77.   An invalid level number
     TREATED AS 01.   (02-49) follows a 77 level
                                     item. The 77 level item is
                                     treated as an 01 level item.
                                     This action may propagate
                                     further diagnostics if it is
                                     not a valid group item.

546 PERIOD OMITTED AFTER .EXIT   The words EXIT PROGRAM are not
     PROGRAM.   followed by a period. The
                                     error is ignored.

547 .EXIT PROGRAM. NOT LAST   An EXIT PROGRAM statement
     STMT OF SENTENCE.   appears in a sequence of
                                     statements within a sentence.
                                     But, it is not the last
                                     statement. All of the
                                     statements following it are
                                     compiled, but can never be
                                     reached during execution.

550 REDEFINING LENGTH SHOULD   The length of a non-01 level
     MATCH ORIGINAL LENGTH.   redefines item is not the
                                     same as the length of the
                                     item it REDEFINES. The new
                                     length is used.

551 REDEFINITION OF .OCCURS.   Items with OCCURS cannot
     ITEM. IGNORED   be redefined. REDEFINES
                                     is ignored.

552 PROCESSING RESUMES AFTER   Prior to issuing this
     BAD FD.   message, the compiler had
                                     discovered bad syntax in the
                                     FD of the FILE SECTION. The
                                     compiler at that time issued
                                     an error message identifying
                                     the syntax error. Then the
                                     compiler went into recovery
                                     mode attempting to recognize
                                     another FD, the WORKING-
                                     STORAGE SECTION header or the
                                     PROCEDURE DIVISION. Upon
                                     recognizing one of these
                                     three language elements, the
                                     compiler issues this
                                     diagnostic indicating that
                                     normal processing resumes.

553 INVALID CLAUSE KEYWORD.   A reserved clause keyword was
     OTHER CLAUSES SKIPPED.   expected at this point in a
                                     data item description entry
                                     of the DATA DIVISION, but was
                                     not recognized by the
                                     compiler. The compiler skips
                                     to the next level number
                                     data item description.

554 INVALID WORD FOLLOWING
.VALUE.. IGNORED.

The VALUE clause contains an
invalid word for this data
description. The entire
VALUE clause is ignored.

555 VALUE CONFLICT.
GROUP VALUE USED.

This VALUE clause assigns
a value to an item
subordinate to a group item
that also has a VALUE clause.
The subordinate VALUE clause
is ignored.

556 LEVEL NUMBER OMITTED.
ITEM IGNORED.

The level number has been
omitted in a data-item
description. All the source
text is ignored up to and
including the next period.

557 NO VALUE AFTER CONDITION
NAME. 88 IGNORED.

An 88 level condition-name
has no VALUE clause
specified. The entire 88
level data-item is ignored.

560 SYNTAX ERROR IN SWITCH
CLAUSE. CLAUSE IGNORED.

The SWITCH clause has a
syntax error in its
specification. The compiler
ignores the entire clause.

561 .NO. MISSING IN
ADVANCING PHRASE. ASSUMED.

The keyword NO is missing in
the ADVANCING phrase of the
DISPLAY statement. NO is
assumed present.

562 .ADVANCING. MISSING AFTER
.NO.. ASSUMED.

The keyword ADVANCING is
missing in the ADVANCING
phrase of the DISPLAY
statement. ADVANCING is
assumed present.

563 DUPLICATE DATANAME
DECLARATION DETECTED.

In the ENVIRONMENT and/or
DATA DIVISION, a data-name is
defined which, if referenced,
is not uniquely referenceable
even with complete
qualification.

564 ILLEGAL PARAGRAPH HEADER
ID DIV. PAR IGNORED.

An illegal paragraph header
appears in the IDENTIFICATION
DIVISION. The paragraph is
ignored.

565 ILLEGAL PARAGRAPH HEADER
ENV DIV. PAR IGNORED.

An illegal paragraph header
appears in the ENVIRONMENT
DIVISION. The paragraph is
ignored.

566 NUMERIC LITERAL ILLEGAL
ON GROUP ITEM. IGNORED.

A numeric literal is illegal
in the VALUE clause of a
group item. The VALUE clause
is ignored.

567 .ENVIRONMENT. NOT FOLLOWED
    BY .DIVISION..

The word ENVIRONMENT is
not followed by the word
DIVISION. DIVISION is
assumed present.

570 TERMINATOR MISSING AFTER
    .DATA DIVISION. HEADER.

The DATA DIVISION header is
not followed by a period.
The period is assumed present
and processing continues.

571 TERMINATOR MISSING AFTER
    PARAGRAPH HEADER.

A paragraph header in the
IDENTIFICATION or ENVIRONMENT
DIVISION is not terminated by
a period. The period is
assumed present and
processing continues.

572 .RENAMES. SPECIFIES STORAGE
    OVERLAP ON RIGHT.

In processing the RENAMES
clause, the compiler detects
the condition in which the
end of the storage allocated
to the data-name after the
THRU keyword is not position-
ally to the right of the end
of the storage allocated to
the data-name after the
RENAMES keyword. This is
syntactically invalid. The
compiler ignores the entire
RENAMES data description
entry.

573 .SECTION. OMITTED FROM
    SECTION HEADER.

An ENVIRONMENT DIVISION
section name is not followed
by the word SECTION. The
error is ignored.

574 TERMINATOR MISSING AFTER
    SECTION HEADER.

An ENVIRONMENT DIVISION
section header is not
terminated by a period. The
error is ignored.

600 ILLEGAL LEVEL NUMBER.
    TREAT AS 01.

This level number is not an
01-49, 66, 77, or 88 level
number. The level number is
assumed to be 01.

601 TERMINATOR MISSING AFTER
    ENV DIV HEADER.

The ENVIRONMENT DIVISION
header is not terminated by a
period. The period is assumed
present and processing
continues.

602 .DATA. NOT FOLLOWED BY
    .DIVISION.

The word DATA is not
followed by the word
DIVISION. DIVISION is
assumed present.

603 ENVIRONMENT DIVISION HEADER
    OMITTED.

The program contains no
ENVIRONMENT DIVISION header.
The compiler resumes
processing at the next
paragraph header.

604 UNRECOGNIZABLE COBOL PROGRAM
     FORMAT. ABORT.

The compiler is unable to
recognize the reserved word
IDENTIFICATION as the first
word required in a COBOL
source program. Failure to
recognize this required
reserved word may be due to
one of the following reasons:
(1)IDENTIFICATION is, in
fact, omitted as the first
word of the source file, (2)
the user is attempting to
compile a COBOL source
program in conventional
format without specifying the
"/CVF" switch, or (3) the
user is attempting to compile
a file which is not a COBOL
source program. The compiler
issues a string of
diagnostics to the printer,
informs the user on the
system console, and then
aborts the compilation.

605 .IDENTIFICATION. NOT FOLLOWED
     BY .DIVISION..

The word IDENTIFICATION is
not followed by the word
DIVISION. DIVISION is
assumed present.

606 TERMINATOR OMITTED AFTER
     .ID DIVISION. HEADER.

The IDENTIFICATION DIVISION
header is not terminated by a
period. The period is
assumed present and
processing continues.

607 .PROGRAMID. EXPECTED AFTER
     DIVISION HEADER.

The IDENTIFICATION DIVISION
header is not followed by
the PROGRAM-ID paragraph.
The error is ignored and
processing continues.

610 TERMINATOR OMITTED AFTER
     .PROGID. PARA HEADER.

The PROGRAM-ID paragraph-name
is not terminated by a
period. The period is
assumed present and
processing continues.

611 INVALID PROGRAM NAME IN
     .PROGRAM ID. PARAGRAPH.

The program name of the
PROGRAM-ID paragraph contains
an invalid character or
exceeds nine characters in
length. The error is ignored
and processing continues.

612 TOO MANY FILES FOR LUNS OR TEMPORARY SPACE.

The compiler has discovered either that more than 30 files are declared in the program or that more than 30 SAME RECORD AREA clauses are specified in the program. The compiler imposes a limit of 30 in both cases, because the associated compiler and/ or object time table space is exhausted.

613 INVALID WORD SUSPENDS PROCESSING. SCAN FORWARD.

An unidentifiable word is found where a verb is expected. A scan is made to a verb, or period, or word in Area A.

614 PROCESSING RESTARTS ON VERB.

Due to a previous syntax error, the compiler went into recovery mode looking for the next verb, period, or Area A word upon which to resume compilation. The compiler has recognized a verb and resumes normal compilation at this point. This message is an observation only.

615 PROCESSING RESTARTS ON PROCEDURE NAME.

Due to a previous syntax error, the compiler went into recovery mode looking for the next verb, period, or Area A word upon which to resume compilation. The compiler has recognized an Area A word and resumes compilation at this point. This message is an observation only.

616 PROCESSING RESTARTS AFTER TERMINATOR.

Due to a previous syntax error, the compiler went into recovery mode looking for the next verb, period, or Area A word upon which to resume compilation. The compiler has recognized a period and resumes normal compilation on the word following the period. This is an observation only.

617 .IDENTIFICATION. KEYWORD NOT IN AREA A.

The compiler detects that the IDENTIFICATION keyword is not in Area A. The compiler ignores the error and continues processing.

620 PARAGRAPH TERMINATOR ASSUMED OMITTED.

A paragraph was terminated without a period. The period is assumed and processing continues.

621 .LINAGE. INVALID FOR THIS
FILE. CLAUSE IGNORED.

The LINAGE clause must not be
specified for a file which
has RELATIVE or INDEXED
organization. The LINAGE
clause is ignored.

622 TERMINATOR MISSING AFTER
PROCEDURE NAME.

A section or paragraph
name is not terminated by a
period. The period is
assumed present and
processing continues.

623 .ELSE DOES NOT HAVE
ASSOCIATED .IF.. IGNORED.

The word ELSE has no
associated IF statement. The
ELSE is ignored.

624 VERB EXPECTED TO FOLLOW
ELSE.. .ELSE. IGNORED.

A SENTENCE ENDS WITH THE
word ELSE. The ELSE is
ignored.

625 .JUSTIFY. WITH NUMERIC OR
EDITED ITEM. IGNORED.

The JUSTIFIED clause must not
be specified for a numeric or
numeric-edited dataitem. The
JUSTIFIED clause is ignored.

626 .BLANK WHEN ZERO. ILLEGALLY
SPECIFIED. IGNORED.

The BLANK WHEN ZERO clause
must be specified only for a
numeric or numeric-edited
data-item. The clause is
ignored.

627 INVALID OR MISSING DATANAME
AFTER .REDEFINES..

The compiler detects the
omission of a valid data-name
reference following the
keyword REDEFINES. The
compiler ignores the
REDEFINES clause and
continues with the processing
of the data description
entry.

630 .REDEFINES. MUST FOLLOW
DATA NAME. IGNORED.

The REDEFINES keyword appears
in the wrong position of a
data description entry. The
REDEFINES clause is ignored.

631 DEPTH OF NESTED .IF.
EXCEEDS LIMIT.

A nested IF statement has
exceeded the maximum depth of
30 levels. The compiler
ignores nesting beyond this
depth of nesting.

632 DUPLICATE PROCEDURE
NAME DETECTED.

In the Procedure Division, a
paragraph or section-name is
defined which, if referenced,
is not uniquely reference-
able, even with
qualification.

633 REFERENCE TO UNDEFINED
PARAGRAPH NAME.

In the Procedure Division, an
explicit qualified reference
is made to a paragraph-name
which is undefined in the
section specified by the
qualifier.

634 FILENAME LITERAL TOO LONG.   A file specification in the
     TRUNCATED.   ASSIGN clause exceeds 40
                                       characters in length. It is
                                       truncated to 40 characters.

635 ILLEGAL SYNTAX IN .GO   The compiler detects illegal
     TO. STATEMENT.   syntax in the GO TO
                                       statement. Fatal.

636 INVALID INTEGER OR   In the LINAGE clause, the
     DATANAME.   compiler failed to recognize
                                       a non-negative integer
                                       literal or a numeric integer
                                       data-name. This phrase of
                                       the LINAGE clause is ignored.

637 .GO TO. HAS MULTIPLE   A simple GO TO statement
     PROCEDURE NAMES.   (i.e., without the DEPENDING
                                       ON phrase) has more than one
                                       procedure-name. Fatal.

640 INVALID WORD FOLLOWS   The word following the DATA
     .DATA DIVISION.   DIVISION header either does
                                       not start in Area A or is not
                                       one of the reserved words
                                       FILE, WORKING-STORAGE,
                                       LINKAGE, or PROCEDURE. The
                                       compiler goes into recovery
                                       mode skipping all source
                                       text until one of the
                                       keywords FILE, WORKING-
                                       STORAGE, LINKAGE, or
                                       PROCEDURE is recognized.

641 INVALID WORD IN FILE   An invalid word was detected
     SECTION. SCAN FORWARD.   in the FILE SECTION where
                                       the keyword FD is expected.
                                       The compiler goes into
                                       recovery mode skipping all
                                       source text until one of the
                                       keywords FD, WORKING-STORAGE,
                                       LINKAGE, or PROCEDURE
                                       is recognized.

642 .OMITTED LABELS IGNORED   The LABEL RECORDS ARE OMITTED
     WITH .VALUE OF ID.   clause is ignored if VALUE OF
                                       ID is specified for a file.
                                       STANDARD labels are assumed.
                                       WARNING.

643 .SECTION. EXPECTED AFTER   The keyword SECTION is
     HEADER WORD.   omitted after the word FILE,
                                       WORKING-STORAGE, OR LINKAGE
                                       SECTION is assumed present
                                       and processing continues.

644 TERMINATOR EXPECTED AFTER   The FILE SECTION, WORKING-
     SECTION HEADER.   STORAGE SECTION, or LINKAGE
                                       header is not terminated
                                       by a period. The
                                       period is assumed and
                                       processing continues.

646 .OF. OR .ID. MISSING
IN .VALUE OF ID..

One or both of the keywords
OF or ID is omitted in the
VALUE OF ID clause. Their
presence is assumed and
processing continues.

647 ILLEGAL WORD IN AREA A.
SCAN FORWARD.

In the WORKING-STORAGE
SECTION, an 01 or 77
level number or the
PROCEDURE keyword was
expected in Area A, but was
not recognized. The compiler
goes into recovery mode
skipping source text until
one of the three language
elements aforementioned is
recognized in Area A.

650 GROUP LEVEL .VALUE.
DISALLOWED.

The VALUE clause on this
group item is not permitted
because a subordinate
elementary item has a non-
DISPLAY usage specified or
has a SYNCHRONIZED clause
specified. The group VALUE
clause is ignored.

651 REFERENCED LINKAGE SECTION
ITEM NOT IN .PD. USING..

This LINKAGE SECTION item has
been referenced in the
PROCEDURE DIVISION. However,
neither this item nor the
level 01 to which
it is subordinate, appeared in
the PROCEDURE DIVISION USING
phrase. Only those LINKAGE
SECTION items appearing in the
PROCEDURE DIVISION USING
phrase, or items subordinate
to them may be referenced in
the PROCEDURE DIVISION of a
COBOL program. FATAL.

652 NON-SEQ FILE IN .MULTIPLE.
FILE TAPE. CLAUSE.

In the I-O CONTROL paragraph,
the MULTIPLE FILE TAPE clause
is specified for a file whose
organization is not
SEQUENTIAL. This is illegal.
The MULTIPLE FILE TAPE clause
is ignored for this file.

653 .VALUE. CLAUSE ILLEGAL IN
FILE SECTION.

A VALUE clause is specified
for a data description entry
given in the FILE SECTION.
This is illegal. The VALUE
clause is ignored.

654 SYNTAX ERROR IN CURRENCY
CLAUSE.

The alphanumeric literal
expected in the CURRENCY SIGN
clause of the SPECIAL-NAMES
paragraph is omitted. The
clause is ignored and the
currency sign defaults to the
dollar sign.

655 ILLEGAL CURRENCY SIGN.

The alphanumeric literal in the CURRENCY SIGN clause is not allowed as the currency sign either because the literal is longer than one character or because it is an invalid COBOL currency sign. The CURRENCY SIGN clause is ignored and the currency sign defaults to the dollar sign.

656 SPECIALNAMES CLAUSE INVALID.

An unrecognizable word appears in a position where a SPECIAL-NAMES paragraph clause keyword is expected. All source text is skipped up to the next recognizable keyword.

657 SYNTAX ERROR IN DECIMALPOINT CLAUSE.

The keyword COMMA is omitted in the DECIMAL-POINT IS COMMA clause of the SPECIAL-NAMES paragraph. The clause is ignored.

660 .AFTER. MISSING IN .USE. STATEMENT. ASSUMED.

The keyword AFTER is omitted in the USE statement. AFTER is assumed present and processing continues.

661 NO .ERROR. OR .EXCEPTION. IN .USE. ASSUMED.

One of the keywords ERROR or EXCEPTION is omitted in the USE statement. The missing keyword is assumed present and processing continues.

662 NO KNOWN CLAUSES IN SPECIALNAMES.

The SPECIAL-NAMES paragraph contains no valid clauses. This is an observation only.

663 REDUNDANT .USE. COVERAGE. PREV. .USE. IGNORED.

Multiple USE statements have referenced the same file. The last USE statement specified is then applied to the referenced file. Fatal.

664 UNKNOWN OPEN MODE IN .USE. STATEMENT.

An unrecognizable OPEN mode option was specified in the USE statement. Fatal.

665 GROUP ITEM HAS BEEN CALLED FILLER.

A FILLER item cannot have any elementary items subordinate to it. The compiler replaces the FILLER declaration with a system-defined name and proceeds with the processing of the newly named group item. The system-defined name is transparent and inaccessible to the user.

666 MISSING ENVIRONMENT
DIVISION.

The program does not contain
an ENVIRONMENT DIVISION. The
compiler skips to the DATA
DIVISION and continues
processing.

667 DIVISION BY ZERO.

The divisor of a DIVIDE
statement is a literal of
zero value. The error is
ignored.

670 VALUE NOT PERMITTED WITH
THIS ITEM.

A VALUE clause is recognized
in a data description entry
which contains a REDEFINES or
an OCCURS clause. This is
illegal. The VALUE clause is
ignored.

671 INVALID CONSTANT OR
LITERAL FOLLOWING .ALL..

The reserved word ALL is not
followed by a non-numeric
literal or a figurative
constant. Thus, this is not
a valid ALL literal. ALL is
ignored and processing
continues.

672 BAD FILENAME IN .USE.
STATEMENT.

An unrecognizable word
appears where a filename is
expected in the USE
statement. Fatal.

673 FILE NOT CLOSED.

The referenced file was
OPENed but there was no CLOSE
statement detected for this
file in the program.

674 SUBJECT OF .ALTER. IS
SECTION NAME.

The ALTER statement references
a section name. Only paragraph
names may be altered. If this
statement is reached during
execution, the program will be
aborted.

675 FILE COVERED BY CONFLICTING
USE PROCEDURE.

There was more than one
conflicting USE procedure
specified for the referenced
file. Fatal.

676 DATAITEM LENGTH EXCEEDS 4095
CHARACTERS.

An elementary or group item
is longer than the
implementation limit of 4095
characters. The compiler
declares the data item with a
length of 4095 characters and
proceeds with the processing
of the data item.

677 SUPPLIED VALUE INVALID FOR
NUM ITEM. IGNORED.

The VALUE clause specifies
invalid value initialization
for a numeric data item. The
compiler ignores the VALUE
clause.

700 FILE ACCESSED BY VERB
REQUIRING REL. OR IDX ORG.

A file whose organization is SEQUENTIAL is referenced by the START or DELETE verbs or by an I/O verb which has the INVALID KEY clause specified. This is illegal. In all these cases, the referenced file must have RELATIVE or INDEXED organization. Fatal.

701 FILE ACCESSED BY VERB REQ.
SEQUENTIAL ORG.

A file whose organization is RELATIVE or INDEXED is referenced by an I/O verb which has the AT EOP or ADVANCING clauses specified. This is illegal. The referenced file must have SEQUENTIAL organization. Fatal.

702 VERB NOT IMPLEMENTED.

An ANS 1974 COBOL verb appears that is not implemented in this release of the compiler. The compiler scans to another verb, period, or word in Area A.

704 OCCURS ILLEGAL FOR 01
OR 77 ITEM. IGNORE.

An OCCURS clause is specified for an 01 or 77 level data-name. The compiler ignores the OCCURS clause.

705 .ACCEPT FROM. OBJECT NOT
IN SPECIALNAMES.

The mnemonic name used in the ACCEPT statement was not defined in the SPECIAL-NAMES paragraph. Fatal.

706 ACCEPT IDENTIFIER INVALID.

The word following the ACCEPT verb is not a data-name or is a data-name which has non-DISPLAY usage or invalid class. Fatal.

707 VERB OR COND. CLAUSE
CONFLICTS WITH FILE ACCESS.

There is a conflict between the ACCESS MODE of the referenced file and the I/O verbs and/or condition clauses which reference this file. Fatal.

710 DATANAME AFTER .GO
DEPENDING. INVALID.

The word following the DEPENDING ON phrase of the GO TO statement is not a data-name or is a data-name which has INDEX usage. This is illegal. Fatal.

711 INVALID CLASS OF DATANAME
AFTER .GO DEPENDING.

The data-name following the DEPENDING ON phrase of the GO TO statement is not a numeric data-name or is a numeric, non-integer data-name. This is illegal. Fatal.

712 .DISPLAY UPON. OBJECT
NOT IN SPECIALNAMES.

The mnemonic name used in the DISPLAY statement was not defined in the SPECIAL-NAMES paragraph. Fatal.

713 .DISPLAY. OPERAND IS INVALID.

A data item in the DISPLAY statement has invalid class or USAGE.

714 MISSING OR INVALID OPERAND.
OF .MULTIPLY..

One of the operands of the MULTIPLY statement either is missing or is invalid. Fatal.

715 ILLEGAL .MULTIPLY. DUE
TO MISSING .BY..

The keyword BY is omitted in the MULTIPLY statement. Fatal.

716 MISSING OR INVALID OPERAND
OF .DIVIDE..

One of the operands of the DIVIDE statement either is missing or is invalid. Fatal.

717 ILLEGAL .DIVIDE. DUE TO
MISSING .BY. OR .INTO..

One of the keywords BY or INTO is omitted in the DIVIDE statement. Fatal.

720 .GIVING. OPTION OF .DIVIDE.
MISSING.

The GIVING option must be specified in a DIVIDE statement when one of the following syntactic elements is present in the DIVIDE statement: (1) a numeric literal follows the keyword INTO or (2) the keyword BY is specified. In this DIVIDE statement, the GIVING option was omitted while one of the two aforementioned syntactic elements was present. Fatal.

721 MISSING OR INVALID OPERAND OF
.ADD. OR .COMPUTE.

One of the operands of an ADD or COMPUTE statement is either missing or is invalid. Fatal.

722 .TO. OR .GIVING. MISSING
FROM .ADD..

One of the keywords TO or GIVING is omitted in the ADD statement. Fatal.

723 MISSING OR INVALID
OPERAND OF SUBTRACT.

One of the operands in the SUBTRACT statement either is missing or is invalid. Fatal.

724 FILE NEEDS DYNAMIC ACCESS
FOR .READ NEXT..

In a READ NEXT statement, the referenced file must have ACCESS MODE IS DYNAMIC specified in the FILE-CONTROL paragraph. Fatal.

725 BAD PROCEDURE NAME IN
.PERFORM..

A missing or invalid
procedure-name is recognized
in the PERFORM statement.
Fatal.

726 ILLEGAL OPERAND OF .TIMES.
OPTION OF .PERFORM..

The TIMES operand of the
PERFORM statement is not
a numeric integer data-name
or numeric integer literal.
The compiler assumes a value
of 1 for the TIMES operand.

727 .TIMES. MISSING FROM
.PERFORM.. ASSUMED.

The PERFORM statement does
not contain the keyword TIMES
but does contain the
iteration value required to
execute the PERFORM
correctly. The keyword TIMES
is assumed present.

730 PROCEDURE NAME OMITTED
IN .ALTER..

A valid procedure-name was
not recognized in the ALTER
statement. Fatal.

731 ILLEGAL .ALTER. DUE
TO MISSING .TO..

The keyword TO was not
recognized in the ALTER
statement. Fatal.

732 FILE HAS VAR. SIZE RECS.
.READ INTO. ILLEGAL.

It is illegal for the READ
INTO statement to reference a
file which has multiple
record descriptions of
different lengths. Fatal.

733 FILE ACCESSED BY VERB
REQUIRING .LINAGE.

A file is accessed by an I/O
verb which did not have a
LINAGE clause in its
specification. Fatal.

734 .DELETE. OR .REWRITE.
WITHOUT INV. KEY OR USE.

A DELETE or REWRITE statement
references a file for which
there was no USE procedure
specified and for which the
INVALID KEY option was not
specified in that DELETE or
REWRITE statement. Fatal.

735 OPEN MODE OR NO READ
PROHIBITS REWRITE OR DELETE.

A DELETE or REWRITE statement
references a file which was
not OPENed in the proper mode
or which has no READ
statement referencing it in
the program. Fatal.

736 .START. CONFLICTS WITH OPEN
MODE.

A START statement references
a file which was not opened
in the proper mode. Fatal.

737 .WRITE. CONFLICTS WITH OPEN
MODE.

A WRITE statement references
a file which was not opened
in the proper mode. Fatal.

740 .READ. CONFLICTS WITH OPEN MODE.

A READ statement references a file which is only opened in OUTPUT or EXTEND mode. Fatal.

741 USE NOT IN DECLAR. OR NOT FOLLOWING SECTION NAME.

The USE statement is not in the DECLARATIVES section of the PROCEDURE DIVISION or is not immediately following a section name inside the DECLARATIVES. Fatal.

742 MORE THAN 255 ALTERNATE KEYS. IGNORED.

The maximum of 255 ALTERNATE KEYS has been exceeded. The clause is ignored.

743 INTEGER IN SWITCH CLAUSE INVALID OR OMITTED.

A SWITCH clause of the SPECIAL-NAMES paragraph either contains an invalid numeric integer or has omitted the integer in its specification. A SWITCH clause integer, for example, n must fall in the decimal range 1<=n<=16. The SWITCH clause is ignored.

744 .IS. OMITTED IN SPECIALNAMES. ASSUMED PRESENT.

The required keyword IS is omitted in a clause of the SPECIAL-NAMES paragraph. IS is assumed present and processing continues.

745 DEVICE MNEMONIC OMITTED IN SPECIALNAMES.

A valid device mnemonic-name is not recognized in one of the CONSOLE, LINE-PRINTER, CARD-READER, PAPER-TAPE-READER, or PAPER-TAPE-PUNCH clauses of the SPECIAL-NAMES paragraph. All source text is skipped up to the next recognizable keyword.

746 TERMINATOR OMITTED IN SPECIALNAMES.

The SPECIAL-NAMES paragraph is not terminated by a period. The period is assumed present and processing continues.

747 SUBJECT OF .ALTER. NOT .GO TO.. ALTER IGNORED.

The paragraph referenced by this ALTER statement does not contain a GO TO statement as its first statement. The ALTER statement is ignored.

750 KEYWORD OMITTED IN .SWITCH. CLAUSE.

One of the keywords OFF or ON is omitted in the SWITCH clause of the SPECIAL-NAMES paragraph. The SWITCH clause is ignored.

751 CONDITION NAME MISSING IN .SWITCH. CLAUSE.

A valid condition-name is not recognized in the SWITCH clause of the SPECIAL-NAMES paragraph. The SWITCH clause is ignored.

752 .CR. OR .DB. NOT AT RIGHT END OF PICTURE.

The PICTURE symbol CR or DB does not appear at the right end of the PICTURE string. The compiler ignores the user-supplied PICTURE and declares the data-name alphanumeric with a "PICTURE X" DECLARATION.

753 .CR. OR .DB. USED WITH SIGNED ITEM.

Both the PICTURE symbols, CR or DB, and a sign, + or -, appear in the same PICTURE. The compiler ignores the user-supplied PICTURE and declares the data-name alphanumeric with a "PICTURE X" declaration.

754 MULTIPLE DEFINITION OF SWITCH. FIRST USED.

Multiple definitions of a COBOL switch are detected in the SPECIAL-NAMES paragraph. All but the first definition of SWITCH are ignored.

755 .SENTENCE. ASSUMED AFTER .NEXT..

The keyword NEXT is not followed by the keyword SENTENCE. SENTENCE is assumed present and processing continues.

756 SUBSCRIPT NOT NUMERIC INTEGER.

A data-name used as a subscript is not numeric in class. A default value of 1 is assumed as the subscript.

760 ILLEGAL SYNTAX IN .DIVIDE. STATEMENT.

The compiler detects illegal syntax in the DIVIDE statement. Fatal.

761 INDEXED FILE REQUIRES .RECORD KEY. PHRASE.

Self explanatory.

762 RECORD KEY INVALID FOR THIS FILE.

The RECORD KEY clause is only valid for indexed files.

763 .ALT RECORD KEY. INVALID FOR FILE. IGNORED.

The ALTERNATE RECORD KEY clause is only valid for indexed files.

764 READ-AHEAD. OR. WRITE-BEHIND. NOT SUPPORTED.

The APPLY READ-AHEAD or APPLY WRITE-BEHIND clauses are not supported in this version of the compiler. The APPLY clause is ignored.

765 INTEGER INVALID IN. RESERVE
AREA. CLAUSE.

The number of buffer areas
reserved by the RESERVE clause
is invalid. The clause is
ignored and a default of one
area for SEQUENTIAL and
RELATIVE or two areas for
INDEXED is supplied.

766 BAD VALUE IN BLOCK CONTAINS
CLAUSE.

The numeric literal in the
BLOCK clause is less than the
sum of the record size, the
record header size, and the
bucket header size. The BLOCK
CONTAINS clause is ignored.

767 VALUE IN. BLOCK CONTAINS.
CLAUSE IS ROUNDED UP

The numeric literal in the
BLOCK clause is not a multiple
of 512. The value is rounded
up to the next even multiple
of 512.

770 EXPECTED .RECORD KEY.
DATANAME NOT DEFINED.

The data-name given in a
RECORD KEY clause has not been
defined in the DATA DIVISION.

771 .RECORD KEY. DATANAME HAS
INVALID CLASS.

A data-name referenced in a
RECORD KEY or ALTERNATE RECORD
KEY phrase of a SELECT clause
in the FILE-CONTROL paragraph
is defined in the FILE SECTION
with non-alphanumeric class.

772 .RECORD KEY. DATA ITEM
CANNOT BE VARIABLE LENGTH.

A data-name referenced in a
RECORD KEY or ALTERNATE RECORD
KEY phrase of a SELECT clause
in the FILE-CONTROL paragraph
is defined in the FILE SECTION
as an item whose size is
variable.

773 .RECORD KEY. ITEM NOT
DEFINED IN RECORD OF FILE

A data-name referenced in a
RECORD KEY or an ALTERNATE
RECORD KEY phrase of a SELECT
clause is not defined in the
record description of the
associated file.

774 FILE ACCESSED BY VERB
REQUIRING INDEXED ORG.

A file whose organization is
SEQUENTIAL or RELATIVE is
referenced by the READ verb
which has the KEY IS data-name
phrase specified. This is
illegal. The referenced file
must have INDEXED
organization. FATAL.

775 .KEY IS. PHRASE INVALID
FOR SEQUENTIAL .READ.

Either the file has ACCESS
SEQUENTIAL or the READ
statement contains the word
NEXT. In either case the KEY
IS data-name phrase is
illegal. FATAL.

776 INVALID DATANAME IN .KEY
IS. PHRASE.

The KEY IS phrase of the READ
statement was not followed by
a data-name. FATAL.

777 .KEY IS. PHRASE NOT FOLLOWED
BY RECORD KEY.

The data-name following the
KEY IS phrase of the READ
statement is not a RECORD KEY
or ALTERNATE RECORD KEY for
the referenced file. The
RECORD KEY data-name
is assumed.

1000 VARIABLE OCCURRENCES TABLE
MUST END RECORD.

A COBOL table declared with
the DEPENDING ON phrase may
only be followed, within the
record, by data description
entries whose level-numbers
are strictly greater than
the level-number of this
table entry. The compiler
ignores the remainder of the
record descriptor from the
point where the error is
detected. Fatal.

1001 .ASCENDING. OR .DESCENDING.
DATANAME EXPECTED.

A user-defined data-name was
expected, but not found, in
the ASCENDING KEY IS or
DESCENDING KEY IS phrase.

1002 RENAMED DATAITEMS NOT IN
CURRENT RECORD.

The data items specified
after the RENAMES keyword
(i.e., the data items being
renamed) are defined outside
of the current record
description. This is syntac-
tically invalid. The compiler
ignores the entire RENAMES
data description entry.

1003 MAXIMUM OCCURRENCES NOT
GREATER THAN MINIMUM.

In a variable occurrence
table declaration, the
integer following the
keyword TO (i.e., the
maximum) must be strictly
greater than the integer
following the keyword OCCURS
(i.e., the minimum). The
compiler assumes the maximum
value to be equal to the
minimum value plus one.

1004 .DEPENDING. IS OMITTED IN THE
.OCCURS. CLAUSE.

In a variable occurrence
table declaration, the
keyword DEPENDING has
been omitted. The compiler
ignores the remainder of the
OCCURS clause and treats the
table declaration as an
ordinary COBOL table.

1005 A DATANAME MUST FOLLOW THE
.DEPENDING. KEYWORD.

In a variable occurrence
table declaration, a valid
data-name is not found

following the keyword
DEPENDING. The compiler
ignores the remainder of the
OCCURS clause and treats the
table declaration as an
ordinary COBOL table.

1006 .OCCURS DEPENDING.
     SUBORDINATE TO AN .OCCURS.

The compiler detects a table
declaration with a DEPENDING
ON phrase subordinate to a
group item which has an
OCCURS clause. This is
syntactically illegal. The
compiler ignores the
DEPENDING ON phrase and
treats the declaration as an
ordinary COBOL table.

1007 MAXIMUM NO. TABLE OCCURRENCES
     MUST BE POSITIVE.

In a variable occurrence
table declaration, the
integer following the keyword
TO (i.e., the maximum) must
be greater than zero. The
compiler assumes the maximum
value to be equal to the
integer value following the
keyword OCCURS (i.e., the
minimum) plus one.

1010 EXPECTED .DEPENDING ON.
     DATANAME NOT DEFINED.

The data-name referenced in a
DEPENDING ON phrase was not
defined in the DATA DIVISION.
Fatal.

1011 EXPECTED .ASCENDING KEY.
     DATANAME NOT DEFINED.

The data-name referenced in
an ASCENDING KEY phrase was
not defined in the DATA
DIVISION. Fatal.

1012 EXPECTED .DESCENDING KEY.
     DATANAME NOT DEFINED.

The data-name referenced in a
DESCENDING KEY phrase was not
defined in the DATA DIVISION.
Fatal.

1013 .DEPENDING ON. DATANAME NOT A
     NUMERIC INTEGER.

The data-name referenced in a
DEPENDING ON phrase was not
declared as a numeric integer
in the DATA DIVISION. Fatal.

1014 .RENAMES. APPLIED TO AN
     INVALID LEVEL OF DATA.

The RENAMES clause specifies
the renaming of data items
whose level number is an 01,
66, 77, or 88. This is syn-
tactically invalid. The
compiler ignores the entire
RENAMES data description
entry.

1015 .DEPENDING ON. DATANAME
     DETECTED WITHIN TABLE.

The compiler detects a
data-name, which follows a
DEPENDING ON phrase and which
defines the current number of
occurrences in a variable
occurrence table, to have its

storage allocated within the range of the table. This is syntactically illegal. Fatal.

1016 .OCCURS. CLAUSE ON A TABLE KEY DATANAME.

The compiler detects the presence of an OCCURS clause on a data item which has been declared as an ASCENDING or DESCENDING KEY. This is syntactically illegal. Fatal.

1017 .SEARCH ALL. TABLE DOES NOT HAVE KEYS.

The table being searched by by SEARCH ALL statement must have the ASCENDING KEY or DESCENDING KEY phrase specified in its declaration. Fatal.

1020 IMPERATIVE STATEMENT EXPECTED DURING .SEARCH.

A period or a non-imperative statement was found where the SEARCH statement environment is expecting an imperative statement. Fatal.

1021 KEYS SPECIFIED FOR .SEARCH ALL. NOT DENSE.

When a key is referenced for the SEARCH ALL statement, all preceding keys in the KEY clause of the table declaration must also be referenced. Fatal.

1022 .WHEN. EXPECTED BUT NOT FOUND IN .SEARCH.

The compiler expected but failed to recognize the WHEN keyword while compiling the SEARCH statement. This SEARCH statement is considered syntactically invalid. Fatal.

1023 THE KEYWORD .WHEN. ILLEGAL IN THIS CONTEXT.

The compiler detects the presence of the keyword WHEN outside the environment of the SEARCH statement. This is syntactically invalid. Fatal.

1024 THE KEYWORD .SEARCH. ILLEGAL IN THIS CONTEXT.

While compiling a SEARCH statement, the compiler detects the presence of another SEARCH statement in the environment of the original SEARCH statement. The second SEARCH statement is detected at a point where an imperative statement is expected. This is syntactically invalid. Fatal.

1025 KEY MUST BE SUBSCRIPTED BY FIRST INDEX OF TABLE.

The SEARCH ALL statement requires that the key referenced on the left side of the simple condition must be subscripted by the first index-name of the table being searched. Fatal.

1026 THE KEYWORD .SENTENCE. EXPECTED AFTER .NEXT..

The keyword SENTENCE was not detected after the NEXT keyword during the compilation of a SEARCH statement. Fatal.

1027 TABLE NAME NOT FOUND AFTER .SEARCH. VERB.

The compiler failed to recognize a valid table data item after the keyword SEARCH or SEARCH ALL. Fatal.

1030 INVALID TABLE REFERENCE IN .SEARCH. STATEMENT.

The table data item reference following the SEARCH or SEARCH ALL verbs must have both the INDEXED BY and the OCCURS clauses specified in its declaration. Fatal.

1031 DATANAME EXPECTED AFTER .VARYING. IN .SEARCH.

No data-name reference was found after the VARYING keyword in the SEARCH statement being compiled. Fatal.

1032 .VARYING. ITEM MUST BE INDEX OR INTEGER.

The data-name reference following the VARYING keyword must be an index data item, an index-name, or an elementary numeric integer data-name reference. Fatal.

1033 .SEARCH ALL. DATA ITEM IS NOT A KEY.

The data item referenced on the left side of the SEARCH ALL simple condition must be declared as an ASCENDING or DESCENDING KEY. Fatal.

1034 DATA ITEM NOT A KEY FOR THIS .SEARCH. TABLE.

The data item referenced on the left side of the SEARCH ALL simple condition is not a key for the table being searched. This is considered illegal. Fatal.

1035 .RENAMES. SPECIFIES RENAMING OF A COBOL TABLE.

The RENAMES clause specifies the renaming of a datum which has an OCCURS clause in its declaration or is subordinate to another datum having an OCCURS clause. This is syntactically invalid. The compiler ignores the entire RENAMES data description entry.

1036 .RENAMES. APPLIED TO VARIABLE LENGTH DATAITEM.

The compiler detects an application of the RENAMES clause to a range of data items which contains a data item whose length is variable at run-time. This data item is variable in length because is has a subordinate data item whose data description entry contains an OCCURS

	DEPENDING ON clause. The application of the RENAMES clause to such a range of data items is syntactically invalid. The compiler ignores the entire RENAMES data description entry.
1037 DATANAME OMITTED AFTER 66 LEVEL NUMBER.	The data-name declaration is omitted after a 66 level number. The compiler ignores the entire RENAMES data description entry.
1040 .RENAMES. OMITTED IN LEVEL 66 DESCRIPTION ENTRY.	The RENAMES keyword is omitted in a level 66 data description entry. The compiler ignores the entire level 66 data description entry.
1041 SEARCH KEY NOT SUBORDINATE TO TABLE.	The compiler detects an ASCENDING or DESCENDING data-name key which is not defined as a data item subordinate to the associated SEARCH table. This is syntactically invalid.
1042 INVALID OR MISSING DATANAME AFTER .RENAMES..	The data-name is missing after the RENAMES keyword or, if present, is not recognized as a valid data item previously defined. The compiler ignores the entire RENAMES data description entry.
1043 .OCCURS. ITEM NOT ALLOWED BETWEEN TABLE AND KEY.	The compiler detects a data item declared with an OCCURS clause "sandwiched" between the declaration of another COBOL table and its associated SEARCH key. This is syntactically invalid.
1044 .RENAMES. SPECIFIES INVALID NOMENCLATURE RANGE.	In processing the RENAMES clause, the compiler detects an invalid nomenclature range specified by identical data-names following the RENAMES and THRU keywords, respectively. This is syntactically invalid. The compiler ignores the entire RENAMES data description entry.
1045 .RENAMES. SPECIFIES STORAGE OVERLAP ON LEFT END.	In processing the RENAMES clause, the compiler detects the condition in which the beginning of the storage allocated to the data-name

after the THRU keyword is positionally to the left of the beginning of the storage allocated to the data-name after the RENAMES keyword. This is syntactically invalid. The compiler ignores the entire RENAMES data description entry.

1046 INVALID OR MISSING DATANAME AFTER .THRU..

In specifying the RENAMES clause, a data-name is missing after the THRU keyword or, if present, is not recognized as a valid data item previously defined. The compiler ignores the entire RENAMES data description entry.

1047 INVALID OR MISSING DATANAME AFTER CORRESPONDING.

In the processing of an ADD, SUBTRACT, or MOVE CORRESPONDING statement, the compiler detects the omission of a valid data-name reference after the CORRE-SPONDING keyword. Fatal.

1050 .TO. OR .FROM. OMITTED IN .CORRESPONDING.

In the processing of an ADD, SUBTRACT, or MOVE CORRE-SPONDING statement, the compiler detects the omission of the TO or FROM keyword. Fatal.

1051 INVALID OR MISSING DATANAME AFTER .TO. OR .FROM.

In the processing of an ADD, SUBTRACT, or MOVE CORRESPONDING statement, the compiler detects the omission of a valid data-name reference after the keyword TO or FROM. Fatal.

1052 NO OBJECT CODE PRODUCED FOR .CORRESPONDING.

In the processing of an ADD, SUBTRACT, or MOVE CORRESPONDING statement, the compiler produced no object code. No object code is produced because no "correspondence" was found between the two group items referenced in the COBOL statement containing the CORRESPONDING option. This diagnostic is informational only.

1053 GROUP ITEM NOT REFERENCED IN
.CORRESPONDING.

In the processing of an ADD,
SUBTRACT, or MOVE
CORRESPONDING statement,
the compiler discovers that
one of the references is a
reference to an elementary
item. This is syntactically
invalid. Fatal.

1054 LEVEL 66 REFERENCE DISALLOWED
IN .CORRESPONDING.

In the processing of an ADD,
SUBTRACT, or MOVE
CORRESPONDING statement, the
compiler detects a reference
to a data-name declared at
level 66. This is an invalid
reference. Fatal.

1055 .FILE STATUS. ITEM DEFINED IN
.FILE SECTION.

A data-name referenced in a
FILE STATUS phrase of a
SELECT clause is defined in
the FILE SECTION of the
COBOL program. The compiler
ignores this error and
continues to process the
FILE STATUS data-name.

1056 INCOMPATIBLE OPERANDS FOUND
IN .CORRESPONDING.

In the processing of an ADD,
SUBTRACT, or MOVE
CORRESPONDING statement, the
compiler detects a pair of
CORRESPONDING data items
which are incompatible. This
diagnostic is informational
only.

1057 EMPTY .GO TO. WAS NOT THE
SUBJECT OF AN .ALTER..

A GO TO statement without a
procedure reference was
detected. The empty GO TO is
not the subject of an ALTER
statement. FATAL.

1060 QUALIFIER OMITTED IN
PROCEDURE REFERENCE.

A section name is omitted
after the keyword OF or IN in
a qualified procedure
reference of the COBOL
statement being compiled.
Fatal.

1061 INCONSISTENT NUMBER OF
ARGUMENTS IN .CALL..

The subprogram referenced in
this CALL statement has been
referenced before. The number
of arguments in the earlier
CALL differs from the number
in the current CALL.

1062 PARAGRAPH WITHOUT SECTION
PRECEDES THIS SECTION.

In a COBOL program, if one
paragraph is in a section,
then all paragraphs must be in
sections. In this source
program, a paragraph not
within a section has been
detected, preceding this
section in the source program.

1063 DUPLICATE PARAGRAPH
     NAME DETECTED.

In a section of the Procedure
Division, a paragraph name is
defined more than once which,
if referenced, is not
uniquely referenceable, even
with qualification.

1064 REFERENCE TO UNDEFINED
     PROCEDURE NAME.

The compiler detects a
reference to an undefined
procedure name in the
PROCEDURE DIVISION. This is
syntactically invalid.

1065 UNDEFINED PROCEDURE QUALIFIER
     REFERENCE.

The compiler detects a
qualified procedure reference
which contains an undefined
qualifier in the PROCEDURE
DIVISION. This is syntac-
tically invalid.

1066 ILLEGAL PROCEDURE NAME
     REFERENCE.

The compiler detects an
invalid procedure name
reference in the PROCEDURE
DIVISION. This is syntac-
tically invalid.

1067 AMBIGUOUS PROCEDURE
     NAME REFERENCE.

The compiler detects a
reference in the PROCEDURE
DIVISION to a procedure name
which is not uniquely
referenceable, even through
qualification. This is
syntactically invalid.

1070 PARAGRAPH NAME
     DISALLOWED AS QUALIFIER.

The compiler detects a
qualified procedure reference
in the PROCEDURE DIVISION
in which the qualifier is a
paragraph name. This is
syntactically invalid.

1071 SECTION NAME REFERENCE MAY
     NOT BE QUALIFIED.

The compiler detects a
qualified procedure reference
in the PROCEDURE DIVISION
in which a section name is
qualified by another section
name. This is syntactically
invalid.

1072 AMBIGUOUS PARAGRAPH
     NAME REFERENCE.

The compiler detects a
reference in the PROCEDURE
DIVISION to a paragraph name
which is not uniquely
referenceable, even through
qualification.

1073 POSSIBLE .PERFORM. RANGE VIOLATION.

The compiler detects a PERFORM THRU statement in which the procedure name following the THRU keyword is defined in the text of the Procedure Division before the procedure name following the PERFORM keyword. This condition may potentially represent a logic problem in the COBOL program being compiled, although not necessarily so.

1074 NUMERIC PROCEDURE NAME EXCEEDS 30 CHARACTERS.

A numeric string which appears to be a numeric procedure name exceeds 30 characters in length. The string is truncated on the right to 30 characters and processing of the numeric procedure name continues.

1075 NUMERIC PROCEDURE NAME CONTAINS DECIMAL POINT.

A numeric string which appears to be a numeric procedure name contains a decimal point. This is syntactically invalid. The compiler ignores the presence of the decimal point and proceeds with the processing of the numeric procedure name.

1076 .RELATIVE KEY. ITEM DEFINED IN RECORD OF FILE.

A data-name referenced in a RELATIVE KEY phrase of a SELECT clause is defined in the record description of the associated file. The compiler ignores this error and continues to process the RELATIVE KEY data-name.

1077 NO. OF AREAS DEFAULTS TO MAX. FOR FILE TYPE

The number of buffer areas reserved by the RESERVE clause is greater than the maximum allowed for the file organization. Instead, the clause will cause two areas to be allocated for a sequential file and one for a relative file.

APPENDIX I

RECORD MANAGEMENT SERVICES ERROR CODES

Any of the following I/O error conditions could occur during COBOL program execution. The codes appear in a COBOL message in the form shown below:

"RECORD MANAGEMENT SERVICES ERROR - nn"

(nn represents the Record Management Services error code).

These error codes are listed in Table I-1.

Table I-1
RMS System Standard Error Codes

Value	Meaning
-16	OPERATION ABORTED(STV=ER$STK/MAP)
-32	F11ACP COULD NOT ACCESS FILE(STV=SYS ERROR CODE)
-48	"FILE" ACTIVITY PRECLUDES OPERATION
-64	BAD AREA ID(STV=@XAB)
-80	ALIGNMENT OPTIONS ERROR(STV=@XAB)
-96	ALLOCATION QUANTITY TOO LARGE
-112	NOT ANSI "D" FORMAT
-128	ALLOCATION OPTIONS ERROR(STV=@XAB)
-144	INVALID(I.E. SYNCH) OPERATION AT AST LEVEL
-160	ATTRIBUTE READ ERROR(STV=SYS ERR CODE)
-176	ATTRIBUTE WRITE ERROR(STV=SYS ERR CODE)
-192	BUCKET SIZE TOO LARGE(FAB)
-208	BUCKET SIZE TOO LARGE(STV=@XAB)
-224	"BLN" LENGTH ERROR(RAB/FAB)
-232	BEGINNING OF FILE DETECTED
-240	PRIVATE POOL ADDRESS NOT MULTIPLE OF "4"
-256	PRIVATE POOL SIZE NOT MULTIPLE OF "4"
-272	INTERNAL RMS ERROR CONDITION DETECTED
-288	CAN'T CONNECT RAB
-304	$UPDATE-KEY CHANGE A KEY WITHOUT HAVING ATTRIBUTE OF XB$CHG SET
-320	BUCKET FORMAT CHECK-BYTE FAILURE
-336	RSTS/E CLOSE FUNCTION FAILED(STV=SYS ERR CODE)
-352	INVALID OR UNSUPPORTED "COD" FIELD(STV=@XAB)
-368	F11-ACP COULD NOT CREATE FILE(STV=SYS ERR CODE)
-384	NO CURRENT RECORD(OPERATION NOT PRECEDED BY GET/FIND)
-400	F11-ACP DEACCESS ERROR DURING "CLOSE"(STV=SYS ERR CODE)
-416	DATA "AREA" NUMBER INVALID(STV=@XAB)
-432	RFA-ACCESSED RECORD WAS DELETED
-448	BAD DEVICE, OR INAPPROPRIATE DEVICE TYPE
-464	ERROR IN DIRECTORY NAME

Table I-1 (Cont.)
RMS System Standard Error Codes

Value	Meaning
-480	DYNAMIC MEMORY EXHAUSTED
-496	DIRECTORY NOT FOUND
-512	DEVICE NOT READY
-520	DEVICE POSITIONING ERROR(STV=SYS ERR CODE)
-528	"DTP" FIELD INVALID(STV=@XAB)
-544	DUPLICATE KEY DETECTED, XB$DUP ATTRIBUTE NOT SET
-560	RSX-F11ACP ENTER FUNCTION FAILED(STV=SYS ERR CODE)
-576	OPERATION NOT SELECTED IN "ORG$" MACRO
-592	END-OF-FILE
-608	EXPANDED STRING AREA TOO SHORT
-616	FILE EXPIRATION DATE NOT YET REACHED
-624	FILE EXTEND FAILURE(STV=SYS ERR CODE)
-640	NOT A VALID FAB("BID" NOT=FB$BID)
-656	ILLEGAL FAC FOR REC-OP,0, OR FB$PUT NOT SET FOR "CREATE"
-672	FILE ALREADY EXISTS
-680	INVALID FILE-ID
-688	INVALID FLAG-BITS COMBINATION(STV=@XAB)
-704	FILE IS LOCKED BY OTHER USER
-720	PSX-F11ACP "FIND" FUNCTION FAILED(STV=SYS ERR CODE)
-736	FILE NOT FOUND
-752	ERROR IN FILENAME
-768	INVALID FILE OPTIONS
-784	DEVICE/FILE FULL
-800	INDEX "AREA" NUMBER INVALID(STV=@XAB)
-816	INDEX NOT INITIALIZED(STV ONLY,STS=ER$RNF)
-832	INVALID IFI VALUE
-848	MAX NUM(254) AREAS/KEY XABS EXCEEDED(STV=@XAB)
-864	$INIT MACRO NEVER ISSUED
-880	OPERATION ILLEGAL, OR INVALID FOR FILE ORG.
-896	ILLEGAL RECORD ENCOUNTERED(SEQ. FILES ONLY)
-912	INVALID ISI VALUE, ON UNCONNECTED RAB
-928	BAD KEY BUFFER ADDRESS(KBF=0)
-944	INVALID KEY FIELD(KEY=0/NEG)
-960	INVALID KEY-OF-REFERENCE($GET/$FIND)
-976	KEY SIZE TOO LARGE(IDX)/NOT=4(REL)
-992	LOWEST-LEVEL-INDEX "AREA" NUMBER INVALID(STV=@XAB)
-1008	NOT ANSI LABELED TAPE
-1024	LOGICAL CHANNEL BUSY
-1040	LOGICAL CHANNEL NUMBER TOO LARGE
-1048	LOGICAL EXTEND ERROR, PRIOR EXTEND STILL VALID(STV=@XAB)
-1056	"LOC" FIELD INVALID(STV=@XAB)
-1072	BUFFER MAPPING ERROR
-1088	F11ACP COULD NOT MARK FILE FOR DELETION(STV=SYS ERR CODE)
-1104	MRN VALUE=NEG/REL.KEY>MRN
-1120	MRS VALUE=0 FOR FIXED LENGTH RECS/=0 FOR REL. FILES
-1136	"NAM" BLOCK ADDRESS INVALID(NAM=0, OR NOT ACCESSIBLE)
-1152	NOT POSITIONED TO EOF(SEG. FILES ONLY)
-1168	CAN'T ALLOCATE INTERNAL INDEX DESCRIPTOR
-1184	INDEXED FILE-NO PRIMARY KEY DEFINED
-1200	RSTS/E OPEN FUNCTION FAILED(STV=SYS ERR CODE)
-1216	XAB'S NOT IN CORRECT ORDER(STV=@XAB)
-1232	INVALID FILE ORGANIZATION VALUE
-1248	ERROR IN FILE'S PROLOGUE(RECONSTRUCT FILE)
-1264	"POS" FIELD INVALID(POS>MRS,STV=@XAB)
-1280	BAD FILE DATE FIELD RETRIEVED(STV=@XAB)
-1296	PRIVILEGE VIOLATION(OS DENYS ACCESS)

RECORD MANAGEMENT SERVICES ERROR CODES

Table I-1 (Cont.)
RMS System Standard Error Codes

Value	Meaning
-1312	NOT A VALID RAB("BID" NOT=RB$BID)
-1328	ILLEGAL RAC VALUE
-1344	ILLEGAL RECORD ATTRIBUTES
-1360	INVALID RECORD BUFFER ADDR("ODD", OR NOT WORD-ALIGNED IF BLK-IO)
-1376	FILE READ ERROR(STV=SYS ERR CODE)
-1392	RECORD ALREADY EXISTS
-1408	BAD RFA VALUE(RFA=0)
-1424	INVALID RECORD FORMAT
-1440	TARGET BUCKET LOCKED BY ANOTHER STREAM
-1456	RSX-F11ACP REMOVE FUNCTION FAILED(STV=SYS ERR CODE)
-1472	RECORD NOT FOUND(STV=0/ER$IDX) RECORD NEVER WAS IN FILE, OR HAS BEEN DELETED
-1488	RECORD NOT LOCKED
-1504	INVALID RECORD OPTIONS
-1520	ERROR WHILE READING PROLOGUE(STV=SYS ERR CODE)
-1536	INVALID RRV RECORD ENCOUNTERED
-1552	RAB STREAM CURRENTLY ACTIVE
-1568	BAD RECORD SIZE(RSZ>MRS, OR NOT=MRS IF FIXED LENGTH RECS RSZ NOT=CURRENT REC.SIZE FOR $UPDATE TO SEQ.FILE
-1584	RECORD TOO BIG FOR USER'S BUFFER(STV=ACTUAL REC SIZE)
-1600	PRIMARY KEY OUT OF SEQUENCE(RAC=RB$SEQ FOR $PUT)
-1616	"SHR" FIELD INVALID FOR FILE(CAN'T SHARE SEQ FILES)
-1632	"SIZ" FIELD INVALID(STV=@XAB)
-1648	STACK TOO BIG FOR SAVE AREA
-1664	SYSTEM DIRECTIVE ERROR(STV=SYS ERR CODE)
-1680	INDEX TREE ERROR
-1696	ERROR IN FILE TYPE
-1712	INVALID USER BUFFER ADDR(0,ODD, OR IF BLK-IO NOT WORD ALIGNED)
-1728	INVALID USER BUFFER SIZE(USZ=0)
-1744	ERROR IN VERSION NUMBER
-1760	INVALID VOLUME NUMBER(STV=@XAB)
-1776	FILE WRITE ERROR(STV=SYS ERR CODE)
-1784	DEVICE IS WRITE-LOCKED
-1792	ERROR WHILE WRITING PROLOGUE(STV=SYS ERR CODE)
-1808	NOT A VALID XAB(@XAB=ODD,STV=@XAB)

APPENDIX J

OBJECT TIME SYSTEM ERROR MESSAGES

Table J-1
COBOL Object Time System Error Messages

Number	Message	Meaning
1	NON EXISTENT OTS ROUTINE INVOKED	The COBOL compiler has generated reference to a nonexistent OTS routine. This should never occur; (COBOL compiler error - notify your DEC Software Specialist).
3	DEPENDING DATA NAME OUT OF RANGE	The data item which defines the current number of elements in the table does not fall within the defined table size range.
4	ILLEGAL SUBROUTINE REENTRY	A COBOL subprogram may not be called while it is still processing a previous call.
5	INCORRECT NUMBER OF SUBROUTINE ARGUMENTS	The number of arguments expected by a COBOL subprogram does not agree with the number actually received.
6	FILE: NN... ATTEMPT TO OPEN 2 'MULTIPLE SAME AREA' FILES SIMULTANEOUSLY	The program tried to open a file that uses the same buffer area of another file that is still open. (NN... represents the file-name.)
7	FILE: NN... NOT OPEN	The program attempted to perform an I/O operation on a file that was not open. (NN... represents the file-name.)

Table J-1 (Cont.)
COBOL Object Time System Error Messages

Number	Message	Meaning
10	FILE: NN... ALREADY OPEN	The program attempted to open a file that was already open. (NN... represents the file-name.)
11	SUBSCRIPT TOO BIG	A subscript value used in a subscripted data item reference has exceeded the upper bounds of the number of items in the table.
12	PERFORM STACK OVERFLOW	The perform stack is used to process nested performs. The size of this stack is fixed at compile time. To increase the default size, specify the /PFM switch at compile time.
13	NULL ALTERABLE GO TO	An alterable GOTO statement has been reached, and no procedure name was assigned to it.
14	STOP, CR TO CONTINUE	The program executed a STOP statement. The OTS waits indefinitely. To continue, type carriage return.
15	STOP RUN	The program executed a STOP RUN statement. The program stops all activity and closes all open files.
16	SUBSCRIPT TOO SMALL	The subscript value of a data item is less than or equal to zero.
17	PERFORM END OF RANGE VIOLATION	The end-point of an active perform range has occurred. However, the perform range in question is not the most recent.
20	FILE: NN... OPTIONAL FILE MOUNTED? Y OR N?	The OTS is asking the operator to specify whether the file NN... is available to the running program. (NN... represents the file-name.) Type a Y for yes, or N (or some other character) for no.

Table J-1 (Cont.)
COBOL Object Time System Error Messages

Number	Message	Meaning
22	INDEX VALUE TOO SMALL OR TOO LARGE AT SOURCE LINE NNNNN	A value for an index name is being used in a SET statement that is outside the bounds of the table. (NNNNN represents the source program's page-line number.)
24	WRITE ERROR IN DISPLAY	A DISPLAY statement encountered a bad device or a record length of more than 132 characters.
25	ILLEGAL NESTED PERFORM	An attempt was made to invoke a perform range whose end-point is that of an active perform range.
26	UNKNOWN PROCEDURE	Self-explanatory; an appropriate diagnostic error message was produced by the compiler. See the compiler listing.
27	SPECIFY "ON" SWITCHES	See Section 2.8
30	ACCEPT-INPUT TOO LONG	A single ACCEPT statement has attempted to read more than 80 characters. The OTS currently imposes a limit of 80 characters on the ACCEPT statement.
31	FILE: NN... OPEN ERROR-XX	The program attempted to open file NN... but the open failed. The Record Management services error code specifies the kind of error. (See Appendix I for the RMS error codes.) (NNN.. represents the file-name. XX represents the error code.)
32	FILE: NN... CLOSE ERROR-XX	The program attempted to close file NN... but the close operation failed. The RMS error code specifies the kind of error. (See Appendix I for the RMS error codes.) (NN... represents the file-name. XX represents the error code.)
33	FILE: NN... NOT OPEN	The program attempted to close file NN... but file NN... is not open. (NN... represents the file-name.)

Table J-1 (Cont.)
COBOL Object Time System Error Messages

Number	Message	Meaning
34	FILE: NN... INVALID LINAGE	The LINAGE clause specified a page body size that has been calculated to be zero. (NN... represents the file-name.)
36	FILE: NN... REWRITE/ DELETE NOT LEGAL WITHOUT PRIOR READ	The program requested a REWRITE or a DELETE operation on a sequential file and the last I/O operation in the file was not a READ.
37	FILE: NN... NO USE PROCEDURE FOR I/O ERROR-XX	The OTS detected an I/O error for file NN... and no USE procedure is specified for the file (explicitly or implicitly). The RMS error code XX, specifies the kind of error. (See Appendix I for the RMS error codes.) This message results from a fatal error; the OTS executes a STOP RUN and closes all open files.
40	FILE: NN... LOCKED	The program previously closed the file with lock during this program execution. (NN... represents the file-name.)
41	FILE: NN... INVALID OPERATION	The program attempted to issue one of the following I/O statements on a file open in an incompatible mode: • A READ on a file open for output; • A WRITE on a file open for input or I-O; • A REWRITE or DELETE on a file open for input or output.
42	ABORT EXECUTION	Execution of the task has arrived at a point where a fatal diagnostic was detected by the compiler.

NOTE

The following errors indicate a fault condition within the task. See the Executive Reference Manual for your operating system.

OBJECT TIME SYSTEM ERROR MESSAGES

Table J-1 (Cont.)
COBOL Object Time System Error Messages

Number	Message	Meaning
43	ODD ADDRESS ERROR	
44	MEMORY PROTECTION VIOLATION	
45	T-BIT TRAP OR BPT INSTRUCTION	
46	IOT INSTRUCTION	
47	RESERVED INSTRUCTION	
50	NON-RSX EMT	
51	FLOATING POINT EXCEPTION	

READER'S COMMENTS

NOTE: This form is for document comments only. DIGITAL will
use comments submitted on this form at the company's
discretion. Problems with software should be reported
on a Software Performance Report (SPR) form. If you
require a written reply and are eligible to receive
one under SPR service, submit your comments on an SPR
form.

Did you find errors in this manual? If so, specify by page.

_____

_____

_____

_____

_____

_____

Did you find this manual understandable, usable, and well-organized?
Please make suggestions for improvement.

_____

_____

_____

_____

_____

_____

Is there sufficient documentation on associated system programs
required for use of the software described in this manual? If not,
what material is missing and where should it be placed?

_____

_____

_____

_____

_____

_____

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
☐ Non-programmer interested in computer concepts and capabilities

Name_____ Date_____

Organization_____

Street_____

City_____ State_____ Zip Code_____
                                                    or
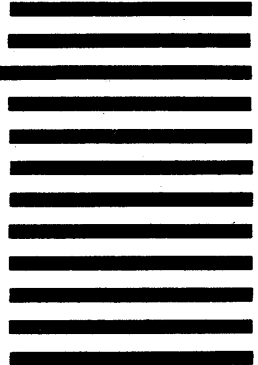   Country

Please cut along this line.

FIRST CLASS
PERMIT NO. 33
MAYNARD, MASS.

BUSINESS REPLY MAIL
NO POSTAGE STAMP NECESSARY IF MAILED IN THE UNITED STATES

Postage will be paid by:

d i g i t a l

Software Documentation
146 Main Street ML5-5/E39
Maynard, Massachusetts 01754

digital

digital equipment corporation