SPERRY UNIVAC

ATTN: CHARLIE GIBBS

01048
CAV208M45541 UP 8805-A

SPERRY UNIVAC UAS
SUITE 906
1177 WEST HASTINGS ST
VANCOUVER BC V6E 2K3

CAV

PUBLICATIONS

General

1974 American National Standard COBOL

Fundamentals

UP-8805-A

This Library Memo announces the release and availability of Updating Package A to "SPERRY UNIVAC 1974 American National Standard COBOL Fundamentals", UP-8805.

This update incorporates a minor change to the manual.

Copies of Updating Package A are now available for requisitioning. Either the updating package only or the complete manual with the updating package may be requisitioned by your local Sperry Univac representative. To receive only the updating package, order UP-8805—A. To receive the complete manual, order UP-8805.

LIBRARY MEMO ONLY	LIBRARY MEMO AND ATTACHMENTS	THIS SHEET IS
Mailing Lists BZ, CZ and MZ	Mailing Lists A00, B00, A11, A12, A13, B13, 10,11,18, 18U,19,19U,20,20U,21,21U,28U,29U,75,75U,76,76U, 77 and 78	Library Memo for UP-8805—A
		RELEASE DATE: August, 1982

SPERRY UNIVAC
COMPUTER SYSTEMS

SPERRY UNIVAC
SUITE 906
1177 WEST HASTINGS ST

VANCOUVER BC V6E 2K3

ATTN: CHARLIE GIBBS

00287
CAV208M45541

UAS

CAV

UP 8805

PUBLICATIONS RELEASE

General

1974 American National Standard COBOL

Fundamentals

UP-8805

This Library Memo announces the release and availability of "SPERRY UNIVAC® 1974 American National Standard COBOL Fundamentals", UP-8805.

This manual is an easy-to-read explanation of 1974 American National Standard COBOL based on the American National Standards Institute COBOL, ANSI X3.23-1974.

This manual describes:

- Structure of COBOL language
- Formats
- Rules
- Program organization
- Identification division
- Environment division
- Data division
- Procedure division
- Special features: table handling, data movement, sort/merge, interprogram communication, communications, debugging, library, segmentation, program execution methods

Additional copies may be ordered by your local Sperry Univac representative.

LIBRARY MEMO ONLY	LIBRARY MEMO AND ATTACHMENTS	THIS SHEET IS:
Mailing Lists BZ, CZ and MZ	Mailing Lists 10, 11, 18, 18U, 19, 19U, 20, 20U, 21, 21U, 28U, 29U, 75, 75U, 76, 76U, 77 and 78 (Covers and 273 pages)	Library Memo
		RELEASE DATE: January, 1981

1974 American National Standard COBOL

Fundamentals

SPERRY UNIVAC

This document contains the latest information available at the time of preparation. Therefore, it may contain descriptions of functions not implemented at manual distribution time. To ensure that you have the latest information regarding levels of implementation and functional availability, please consult the appropriate release documentation or contact your local Sperry Univac representative.

Sperry Univac reserves the right to modify or revise the content of this document. No contractual obligation by Sperry Univac regarding level, scope, or timing of functional implementation is either expressed or implied in this document. It is further understood that in consideration of the receipt or purchase of this document, the recipient or purchaser agrees not to reproduce or copy it by any means whatsoever, nor to permit such action by others, for any purpose without prior written permission from Sperry Univac.

Sperry Univac is a division of the Sperry Corporation.

This document was prepared by Systems Publications using the SPERRY UNIVAC UTS 400 Text Editor. It was printed and distributed by the Customer Information Distribution Center (CIDC), 555 Henderson Rd., King of Prussia, Pa., 19406.

PRINTED IN U.S.A.

PAGE STATUS SUMMARY
ISSUE: UPDATE A – UP-8805

Part/Section	Page Number	Update Level
Cover/Disclaimer		Orig.
PSS	1	A
Acknowledgment	1	Orig.
Contents	1 thru 7	Orig.
1	1 thru 6	Orig.
2	1 thru 21	Orig.
3	1 thru 15	Orig.
4	1,2	Orig.
5	1 thru 20	Orig.
6	1 thru 21 22 23 thru 36	Orig. A Orig.
7	1 thru 70	Orig.
8	1 thru 24	Orig.
9	1 thru 15	Orig.
10	1 thru 7	Orig.
11	1 thru 20	Orig.
12	1 thru 9	Orig.
13	1 thru 4	Orig.
14	1 thru 4	Orig.
Index	1 thru 9	Orig.
User Comment Sheet		

Part/Section	Page Number	Update Level

Part/Section	Page Number	Update Level

All the technical changes are denoted by an arrow (➔) in the margin. A downward pointing arrow (↓) next to a line indicates that technical changes begin at this line and continue until an upward pointing arrow (↑) is found. A horizontal arrow (➔) pointing to a line indicates a technical change in only that line. A horizontal arrow located between two consecutive lines indicates technical changes in both lines or deletions.

Acknowledgment

The following acknowledgment is reproduced from the *American National Standard COBOL, X3.23—1974* as requested in that publication:

"Any organization interested in reproducing the COBOL standard and specifications in whole or in part, using ideas from this document as the basis for an instruction manual or for any other purpose, is free to do so. However, all such organizations are requested to reproduce the following acknowledgment paragraphs in their entirety as part of the preface to any such publication (any organization using a short passage from this document, such as in a book review, is requested to mention "COBOL" in acknowledgement of the source, but need not quote the acknowledgment):

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

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

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

> FLOW-MATIC (trademark of Sperry Corporation), Programming for the UNIVAC® I and II, Data Automation Systems copyrighted 1958, 1959, by Sperry Corporation; IBM Commercial Translator Form No. F 28-8013, copyrighted 1959 by IBM; FACT, DSI 27A5260-2760, copyrighted 1960 by Minneapolis-Honeywell

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

Contents

3. LANGUAGE STRUCTURE

4. IDENTIFICATION DIVISION

5. ENVIRONMENT DIVISION

6. DATA DIVISION

7. PROCEDURE DIVISION

FIGURES

TABLES

1. Introduction

1.1. WHAT IS COBOL?

The COmmon Business Oriented Language (COBOL) is a high level programming language predominantly used for writing business and commercial application programs.

What is it about COBOL that makes it desirable to the business community? Most businesses come to COBOL for several reasons. Some of the reasons are because COBOL is:

- Standarized
- Easy to interpret
- Easy to learn
- Self documenting.

Probably the major reason for the wide acceptance of COBOL is that it is designed and updated by a committee of representatives of computer manufacturers and computer users. The standard COBOL has been in use for more than a decade; the latest update is *American National Standard COBOL, X3.23—1974.*

Any computer manufacturer that implements the standard COBOL compiler must adhere to rules stated in the standard. This means that the COBOL language has established itself in the data processing industry as a common programming language, understood from one computer manufacturer to another. It, therefore, provides stability and order to computer programming. Changing computers or programmers in a company that uses COBOL is simplified since COBOL is common across machines.

Another reason COBOL is so popular in business is that it uses an English-like language that is easily understood. Almost anyone can pick up a COBOL program and get an inkling as to what the program is supposed to do. For instance, a typical line from a COBOL program might be:

```
READ FILE-A, AT END GO TO END-ROUTINE.
```

Also contributing to the order rendered by COBOL is the structure of the language. It is broken into four easily identifiable parts called divisions. The order and content of each of these divisions is as follows:

- IDENTIFICATION DIVISION

  Here, you identify the source program, compilation listings, and other pertinent information about the program.

- ENVIRONMENT DIVISION

  In this division you describe the computer used to compile the source program and the computer and peripheral equipment used to execute the object program.

- DATA DIVISION

  This division describes all external data needed by your program as well as the constants and work areas required by the program.

- PROCEDURE DIVISION

  In this division, you specify the logical steps that the computer takes to solve your problem.

The inherent characteristic of this structure promotes orderly programming, the offspring of which is a program that is self-documented and thus easy to interpret.

1.2. STRUCTURE OF COBOL LANGUAGE SPECIFICATION

The *American National Standard COBOL, X3.23—1974* specification is structured into a nucleus and a number of functional processing modules. The nucleus contains language elements for internal processing. The functional processing modules are listed in Table 1–1. Each module contains either two or three levels. Those with three levels contain a null set at their lowest level, a low processing level (level 1), and a high processing level (level 2). In all cases, lower levels are subsets of higher levels within the same module.

Table 1—1. COBOL Language Processing Levels

Module	Level
Nucleus	2
Table handling	2
Sequential I-O	2
Relative I-O	2
Indexed I-O	2
Sort/merge	2
Segmentation	2
Library	2
Debug	2
Interprogram communication	2
Communication	2

1.2.1. Module Overview

- Nucleus

  The nucleus contains the language elements for internal processing. This module is divided into two levels. The Level 1 elements perform basic internal operations; i.e., elementary options of the various clauses and verbs. Level 2 provides more extensive and sophisticated internal processing capabilities.

- Table handling

  The table handling module contains the language elements necessary for:

  - the definition of tables;
  - the identification, manipulation, and use of indexes; and
  - reference to the items within tables.

  This module is divided into two levels. Level 1 provides the ability to define fixed-length tables of up to three dimensions and to refer to items within them using either a subscript or an index. Level 2 provides for the definition of variable-length tables. In addition, facilities for serial and nonserial lookup are provided by the SEARCH verb and its attendant data division clauses.

- Sequential I-O

  The sequential I-O module contains the language elements necessary for the definition and access of sequentially organized external files. The module is divided into two levels. Level 1 contains the basic facilities for the definition and access of sequential files and for the specification of checkpoints. Level 2 contains more complete facilities for defining and accessing these files.

- Relative I-O

  The relative I-O module provides the capability of defining and accessing mass storage files in which records are identified by relative record numbers. This module contains a null set as its lowest level and two processing levels. Level 1 provides basic facilities. Level 2 provides more complete facilities, including the capability of accessing the file both randomly and sequentially in the same COBOL program.

- Indexed I-O

  The indexed I-O module provides the capability of defining mass storage files in which records are identified by the value of a key and accessed through an index. This module contains a null set as its lowest level, and two processing levels. Level 1 provides basic facilities. Level 2 provides more complete facilities, including alternate keys, and the capability of accessing the file both randomly and sequentially in the same COBOL program.

- Sort/Merge

  The sort/merge module allows for the inclusion of one or more sorts in a COBOL program and consists of a null set and two processing levels. Level 1 contains facilities to sort a single file only; Level 2 provides extended sorting capabilities, including a merge facility.

- Segmentation

  The segmentation module provides for the overlaying at object time of procedure division sections. This module consists of a null set and two processing levels. Level 1 provides for section segment numbers and fixed segment limits; Level 2 adds the capability for varying the segment limit.

- Library

  The library module consists of a null set and two processing levels. It provides for the inclusion into a program of predefined COBOL text. Level 1 contains the basic COPY verb; Level 2 adds the REPLACING phrase.

- Debug

  The debug module provides a means by which the user can specify his debugging algorithm - the conditions under which data or procedure items are monitored during execution of the program. It consists of a null set and two processing levels. Level 1 provides a basic debugging capability, including the ability to specify selective or full paragraph monitoring. Level 2 provides the full COBOL debugging capability.

- Interprogram Communication

  The interprogram communication module provides a facility to which a program can communicate with one or more other programs. This module consists of a null set and two processing levels. Level 1 provides a capability to transfer control to another program known at compile time and the ability for both programs to have access to certain common data items. Level 2 adds the ability to transfer control to a program not identified at compile time as well as the ability to determine the availability of object time main storage for the called program. The high level also provides the capability for the release of main storage areas occupied by called programs.

- Communication

  The communication module provides the ability to access, process, and create messages or portions of messages, and to communicate through a COBOL message control system with local and remote communication devices. This module consists of a null set and two processing levels. Level 1 provides basic facilities to send or receive complete messages. Level 2 provides a more sophisticated facility including the capability to send or receive segments of a message.

1.3. SYMBOLS, RULES, AND NOTATIONS USED IN THIS MANUAL

1.3.1. Format

A format is the specific arrangement of the elements of a clause or a statement. A clause or a statement consists of elements as defined in 1.3.3. Throughout this document, a format is shown adjacent to information defining the clause or statement. When more than one specific arrangement is permitted, the format is separated into numbered formats. Clauses must be written in the sequence given except where specifically stated in the rules associated with a given format. (Clauses that are optional must appear in the sequence shown if they are used.) Applications, requirements, or restrictions are shown as rules.

1.3.2. Rules

Rules are used to define or clarify:

1. the syntax or arrangement of words or elements in a larger structure, such as a clause or statement; or

2. the meaning or relationship of meanings of an element or set of elements in a statement and the effect of the statement on compilation or execution.

1.3.3. Elements

Elements that make up a clause or a statement consist of uppercase and lowercase words, level-numbers, brackets, braces, connectives, and special characters.

- Words

  All underlined uppercase words are called key words and are required when the functions of which they are a part are used. Uppercase words that are not underlined are optional to the user and may or may not be present in the source program. Uppercase words, whether underlined or not, must be spelled correctly.

  Lowercase words, in a general format, are generic terms used to represent COBOL words, literals, PICTURE character-strings, comment-entries, or a complete syntactical entry that must be supplied by the user. Where generic terms are repeated in a general format, a number or letter appendage to the term serves to identify that term for explanation or discussion.

- Level-Numbers

  When specific level-numbers appear in data description entry formats, the specific level-numbers are required when such entries are used in a COBOL program. In this document, the form 01, 02, ..., 09 is used to indicate level-numbers 1 through 9.

- Brackets and Braces

  When a portion of a general format is enclosed in brackets, [ ], that portion may be included or omitted at the user's choice. Braces, { }, enclosing a portion of a general format mean a selection of one of the options contained within the braces must be made. In both cases, a choice is indicated by vertically stacking the possibilities. When brackets or braces enclose a portion of a format, but only one possibility is shown, the function of the brackets or braces is to delimit that portion of the format to which a following ellipsis applies. If an option within braces contains only reserved words that are not key words, then the option is a default option (implicitly selected unless one of the other options is explicityly indicated).

- Ellipsis

  In text, the ellipsis (...) may show the omission of a portion of a source program. This meaning becomes apparent in context. In the general formats, the ellipsis represents the position at which repetition may occur at the user's option. The portion of the format that may be repeated is determined as follows:

  > Given ... in a clause or statement format, scanning right to left, determine the] or } immediately to the left of the ... ; continue scanning right to left and determine the logically matching [or { . The ... applies to the words between the determined pair of delimiters.

- Format Punctuation

  The punctuation characters comma and semicolon shown in some formats are optional and may be included or omitted by the user. In the source program, these two punctuation characters are interchangeable and either one may be used wherever one of them is shown. Neither one may appear immediately preceding the first clause of an entry or paragraph. If desired, a semicolon or comma may be used between statements in the procedure division.

  Paragraphs within the identification and procedure divisions, and the entries within the environment and data divisions, must be terminated by the separator period.

- Use of Certain Special Characters

  When the special characters plus sign, minus sign, greater than, less than, and equal sign (+ - > < =) appear in a format, they are required although they are not underlined.

The following sections of this document discuss COBOL as it is presented in *American National Standard COBOL, X3.23—1974* with descriptions and examples. The characteristics of specific implementation are not discussed since each Sperry Univac computer system has separate COBOL documents for this purpose.

2. COBOL Program Organization

The divisional structure of a COBOL program is described in Section 1 and what, in general, each division does. To give you a clearer picture of how a COBOL program is organized and how the divisions interrelate, a basic COBOL program is examined in this section.

The program input is a deck of 80-column punched cards. The first card contains a date; all subsequent cards contain a number. After the last card is read, the program prints the date, a message, and the sum of the numbers on the cards. The program solution to this problem introduces you to many COBOL statements. No attempt is made to give you all the formats and rules that apply to these statements since they are discussed in individual sections throughout this document.

2.1. IDENTIFICATION DIVISION

Every COBOL program begins with the identification division, which names the program. As the first entry in the division, you simply code:

```
IDENTIFICATION DIVISION.
```

This marks the start of the program. The rest of the division consists of paragraphs headed by paragraph headers. Your program name belongs in the first paragraph headed by the name PROGRAM-ID. So, if you want to call the program FINDSUM, you need the following source code line:

```
PROGRAM-ID.  FINDSUM.
```

The other paragraphs in the identification division are optional; they allow you to make comments about the program. Two paragraphs are used to identify the program author and the installation:

```
AUTHOR.  J. PROGRAMMER.
INSTALLATION.  XYZ COMPANY
               NEW YORK.
```

The entries following the headers for the identification division's optional paragraphs are termed comment-entries. They can be any combination of characters from the computer character set.

The identification division of this program looks like this:

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  FINDSUM.
AUTHOR.  J. PROGRAMMER.
INSTALLATION.  XYZ COMPANY
               NEW YORK.
```

2.2. ENVIRONMENT DIVISION

The environment division follows the identification division. It provides machine-dependent information, so many of its entries are names specified by the manufacturer of the computer you are using.

The environment division, like the identification division, is divided into paragraphs. The paragraphs, in turn, are grouped into two sections: the CONFIGURATION SECTION and the INPUT-OUTPUT SECTION.

The configuration section contains two paragraphs (headed by the names SOURCE-COMPUTER and OBJECT-COMPUTER) that designate the computers used to compile the source program and execute the resultant object program. The computer manufacturer (the implementor) tells you what names belong in these paragraphs. Generally, you use the same computer name for both the SOURCE-COMPUTER and OBJECT-COMPUTER paragraphs. Assume you plan to use the same computer for both compilation and execution, and the implementor designates that the name COMPUT-001 represents that computer. The configuration section, then, contains the entries:

```
SOURCE-COMPUTER.  COMPUT-ØØ1.
OBJECT-COMPUTER.  COMPUT-ØØ1.
```

In the other section of the environment division, the INPUT-OUTPUT SECTION, we need a FILE-CONTROL paragraph that has two entries, one to identify the input card file and one to identify the output print file. Two clauses are required in a file control entry – a SELECT clause to name the file, and an ASSIGN clause to associate that file with a physical device. We code SELECT CARDIN to assign the name CARDIN to the input file. We then add ASSIGN TO CARD-READER to associate CARDIN with a card reader. That gives us a complete file control entry:

```
SELECT CARDIN ASSIGN TO CARD-READER.
```

Similarly, for the output file, we code

```
SELECT PRINTOUT ASSIGN TO PRINTER.
```

This assigns the name PRINTOUT to the print file and associates it with a printer.

The terms CARD-READER and PRINTER, like COMPUT-001, are names provided by the implementor, so they are not the same for each computer manufacturer. (Check the COBOL manual provided with your computer to find the appropriate implementor-name for each type of hardware device.)

Combining the configuration section and input-output section entries, we have a complete environment division:

```
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.  COMPUT-ØØ1.
OBJECT-COMPUTER.  COMPUT-ØØ1.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
   SELECT CARDIN ASSIGN TO CARD-READER.
   SELECT PRINTOUT ASSIGN TO PRINTER.
```

2.3. DATA DIVISION

We have named our program and described the equipment required for execution. Now, in the data division, we describe the data the program will manipulate.

First, we provide information about the files CARDIN and PRINTOUT, including a description of the records in each. This information belongs in the FILE SECTION, the first section in the data division.

Each file named in a SELECT clause in the environment division (2.2) must be described by a file description entry, identified by the letters FD in the FILE SECTION. Thus, in the data division, we need an entry that begins

```
FD CARDIN
```

to describe the input file, and another entry that begins

```
FD PRINTOUT
```

to describe the output file.

In any FD entry, we need a clause that tells whether the file has records containing labels that identify the file. Generally, the first record in a tape or disk file is a label record. Card and printer files, however, cannot have label records; thus, we include the following clause in each FD entry:

```
LABEL RECORDS ARE OMITTED
```

No other clauses are required in an FD entry but, in this example we have included two common optional clauses:

```
RECORD CONTAINS 80 CHARACTERS
```

and

```
RECORD CONTANS 132 CHARACTERS
```

This shows that each card record has 80 characters and each printer record has 132 print positions.

Another clause, DATA RECORD, names the records associated with the file. Let's use CARD-INPUT to name the records in CARDIN, and PRINTLINE for the records in PRINTOUT. The clauses, then, are

```
DATA RECORD IS CARD-INPUT.
```

in the FD CARDIN entry, and

```
DATA RECORD IS PRINTLINE.
```

in the FD PRINTOUT entry.

Immediately following each FD entry, we must describe each type of record. The record description for CARD-INPUT is:

```
01  CARD-INPUT.
    02  CARD-TYPE     PIC X.
    02  DATE.
        05  MONTH     PIC 9(2).
        05  DAY       PIC 9(2).
        05  YEAR      PIC 9(2).
    02  ADDEND        PIC 9(3).
    02  FILLER        PIC X(70).
```

Each name in the record description entry is preceded by a number, in this case 01, 02, or 05. These numbers, called level-numbers, show the hierarchical relationships of data in COBOL records. For instance, the name following the level-number 01, CARD-INPUT, is the most inclusive entry. You use it to reference the 80-character record as a whole. Names following the higher level-numbers are less inclusive; they reference only a portion of the record.

You specify the exact number of characters referenced with the PICTURE clause, abbreviated PIC. This clause tells the type of data referenced (X means alphanumeric, 9 means numeric) and the number of characters referenced. Thus, the entry

```
Ø2  CARD-TYPE  PIC X.
```

means that the name CARD-TYPE references one alphanumeric character; in this instance, the first character in the record. Similarly, the entry

```
Ø2  ADDEND  PIC 9(3).
```

means that the name ADDEND references three characters (the eighth, ninth, and tenth in the record), and the characters are numeric. The 3 in parentheses simplifies coding; you could have entered this source line as

```
Ø2  ADDEND  PIC 999.
```

Note how easy it is to make the names meaningful. You can use English words or groups of English words (connected by hyphens) that describe how the associated data item is used. Just be certain to limit the length of the data-names to 30 characters.

Names followed by a PIC clause, such as CARD-TYPE and MONTH, are called elementary items – they cannot be further subdivided. Names like CARD-INPUT and DATE, not followed by PIC clauses, are called group items, and are always subdivided (they are followed by group and elementary items with higher level-numbers).

Related elementary items are defined under one group item so you can reference the contents of these elementary items at the same time. DATE, for instance, references the 6-character field that contains three elementary items – MONTH, DAY, and YEAR.

The field named CARD-TYPE identifies which record contains a date, and which records contain a number. When CARD-TYPE equals A, the date is in character positions 2 through 7, referenced by DATE. When CARD-TYPE equals B, a number is in character positions 8 through 10, referenced by ADDEND. Thus, the first 10 character positions of the first input record might be

```
A1Ø2579△△△
```

(where △ is a blank) and the first 10 character positions of the next record might be

```
B△△△△△△255
```

You can see that in the first record, CARD-TYPE equals A and DATE equals 102579, but the ADDEND field is blank. In the next record, where CARD-TYPE equals B, the opposite occurs – ADDEND has a value (255) but DATE is equal to blanks.

We do not reference the last 70 characters of the input records because they do not contain any information we need. We have to account for these characters, however, by assigning them the special name FILLER. You can use this as many times as necessary to name data items you do not reference in the program.

But we need a second record description in our program. This one immediately follows the FD entry. We used

```
DATA RECORD IS PRINTLINE.
```

to assign a name to the records in the file. The record description for PRINTLINE, however, is going to be a little different than that for CARD-INPUT. Instead of defining each field, we code only one source line that defines the record as a whole, as in

```
01  PRINTLINE PIC X (132).
```

We still want to show the hierarchical structure of the record, but we are going to exercise our option to do it in the working-storage section rather than the file section. Before that, let's take a look at the complete file section.

```
FILE SECTION.
FD  CARDIN
    RECORD CONTAINS 80 CHARACTERS
    LABEL RECORDS ARE OMITTED
    DATA RECORD IS CARD-INPUT.
01  CARD-INPUT.
    02  CARD-TYPE     PIC X.
    02  DATE.
        05  MONTH     PIC 9(2).
        05  DAY       PIC 9(2).
        05  YEAR      PIC 9(2).
    02  ADDEND        PIC 9(3).
    02  FILLER        PIC X(70).
FD  PRINTOUT
    RECORD CONTAINS 132 CHARACTERS
    LABEL RECORDS ARE OMITTED
    DATA RECORD IS PRINTLINE.
01  PRINTLINE     PIC X(132).
```

The working-storage section is the second of the five sections that can be used in the data division. It can be advantageous to define I-O records (such as PRINTLINE) in working-storage. Of course, we already defined PRINTLINE when we coded

```
01  PRINTLINE     PIC X(132).
```

after the FD entry. That source line reserves 132 bytes of storage in an output record area. We use an additional 132 bytes to also describe the record in working-storage, but we benefit by being able to use the VALUE clause (which has restricted availability in the file section).

The VALUE clause allows you to assign an initial value to a data item. If we want to print a line that looks like this:

```
01/25/79          THE TOTAL IS  245,238
```

we can use the VALUE clause in the record description to include the words THE TOTAL IS before the number; to insert slashes (/) separating the month, day, and year; and to initialize the spaces between the items to blanks. We define the line this way:

```
01  SUM-LINE.
    02  FILLER      PIC X(27) VALUE SPACES.
    02  MONTH-OUT   PIC 9(2).
    02  FILLER      PIC X VALUE ''/''.
    02  DAY-OUT     PIC 9(2).
    02  FILLER      PIC X VALUE ''/''.
    02  YEAR-OUT    PIC 9(2).
    02  FILLER      PIC X(25) VALUE SPACES.
    02  MSG-OUT     PIC X(12) VALUE ''THE TOTAL IS''.
    02  FILLER      PIC X (3) VALUE SPACES.
    02  SUM-OUT     PIC 999,999.
    02  FILLER      PIC X(50) VALUE SPACES.
```

Although we are describing a line for the printer, we cannot use the same record name (PRINTLINE) as in the file section. In COBOL, each name used to reference a storage area must be unique. (FILLER is not a unique name; thus, it cannot be referenced.)

The record in working-storage, called SUM-LINE, occupies a different 132 bytes in storage than the record named PRINTLINE. Once we build the record in working-storage, we code (in the procedure division, 2.4) like this:

```
MOVE SUM-LINE TO PRINTLINE
```

and then

```
WRITE PRINTLINE
```

to get the output to the printer. Each entry in SUM-LINE named FILLER has the phrase VALUE SPACES following the PIC clause. This initializes those areas of the print line to blanks, ensuring that the gaps between the date (MSG-OUT and SUM-OUT) will not contain unwanted characters.

The VALUE clause associated with MSG-OUT makes MSG-OUT equal to the 12 characters (blank is a character) between the quotation marks – THE TOTAL IS. As might be expected, we use MONTH-OUT, DAY-OUT, and YEAR-OUT to store the date found on the first input cards, and SUM-OUT to receive the sum of the numbers that appear on all the other cards. The FILLER entries with VALUE "/" separate the month from the day and the day from the year.

Generally, when you write a COBOL program that produces a printed report, you want to include headings at the top of each printed page. If you do, you can describe headings in working-storage in record descriptions similar to SUM-LINE. For instance, to put SAMPLE REPORT at the top of the page, we include the following description in the working-storage section:

```
01  HEADING-LINE.
    02  FILLER     PIC X(60)  VALUE SPACES.
    02  FILLER     PIC X(13)
        VALUE    ''SAMPLE REPORT''.
    02  FILLER     PIC X(59)  VALUE SPACES.
```

Then, as with SUM-LINE, we print this line by coding

```
MOVE HEADING-LINE TO PRINTLINE
```

and

```
WRITE PRINTLINE.
```

The line consists of 60 blank characters, followed by SAMPLE REPORT, followed by 59 blanks.

While the working-storage section is commonly used to describe records associated with I-O files, you also use working-storage to describe records that are merely data elements and constants that have a hierarchic relationship to each other.

Only one running total is kept in this program, so we do not need a structured record to accumulate it. Instead, we use a single elementary data item. These totals also belong in working-storage and they are preceded by the level number 77. Thus, the accumulator field is described this way:

```
77  TOTAL     PIC 9(6).
```

There are two other sections – linkage and communication – that you can use in the data division. Both are for specialized types of applications. They are described in Sections 10 and 11, respectively.

Combining the working-storage and file sections gives us a complete data division:

```
DATA DIVISION.
FILE SECTION.
FD  CARDIN
    RECORD CONTAINS 80 CHARACTERS
    LABEL RECORDS ARE OMITTED
    DATA RECORD IS CARD-INPUT.
01  CARD-INPUT.
    02  CARD-TYPE     PIC X.
    02  DATE.
        05  MONTH     PIC 9(2).
        05  DAY       PIC 9(2).
        05  YEAR      PIC 9(2).
    02  ADDEND     PIC 9(3).
    02  FILLER     PIC X(70).
FD  PRINTOUT
    RECORD CONTAINS 132 CHARACTERS
    LABEL RECORDS ARE OMITTED
    DATA RECORD IS PRINTLINE.
01  PRINTLINE      PIC X(132).
WORKING-STORAGE SECTION.
77  TOTAL   PIC 9(6).
01  SUM-LINE.
    02  FILLER       PIC X(27)    VALUE SPACES.
    02  MONTH-OUT    PIC 9(2).
    02  FILLER       PIC X    VALUE ''/''.
    02  DAY-OUT      PIC 9(2).
    02  FILLER       PIC X    VALUE ''/''.
    02  YEAR-OUT     PIC 9(2).
    02  FILLER       PIC X(25)    VALUE SPACES.
    02  MSG-OUT      PIC X(12)  VALUE ''THE TOTAL IS''.
    02  FILLER       PIC X(3)     VALUE SPACES.
    02  SUM-OUT      PIC 999,999.
    02  FILLER       PIC X(50)    VALUE SPACES.
01  HEADING-LINE.
    02  FILLER       PIC X(60)    VALUE SPACES.
    02  FILLER       PIC X(13)    VALUE ''SAMPLE REPORT''.
    02  FILLER       PIC X(59)    VALUE SPACES.
```

2.4. PROCEDURE DIVISION

The data we want to act on is described in 2.3. In this executable portion of the program, we code the instructions that act upon that data.

Let's review what we intend to accomplish in this program: The input is a deck of punched cards. The first card has an A in card column 1 followed by a date, and all subsequent cards have a B in card column 1 and numbers in columns 8, 9, and 10. We want to calculate the sum of these numbers. Our output is a printed page that has two lines - the first line reads SAMPLE REPORT; the second line has the date, followed by the words THE TOTAL IS, followed by the sum we calculated.

The program logic is described as follows: We open the files and begin reading cards. If the card contains a date, we store it in the fields called MONTH-OUT, DAY-OUT, and YEAR-OUT and get the next card. If the card contains a number, we add it to the intermediate storage area TOTAL and read the next card. When all the cards have been read, we move TOTAL to SUM-OUT, write the heading line and sum line, close the files, and end the program. A flowchart of this program is given in Figure 2-1.

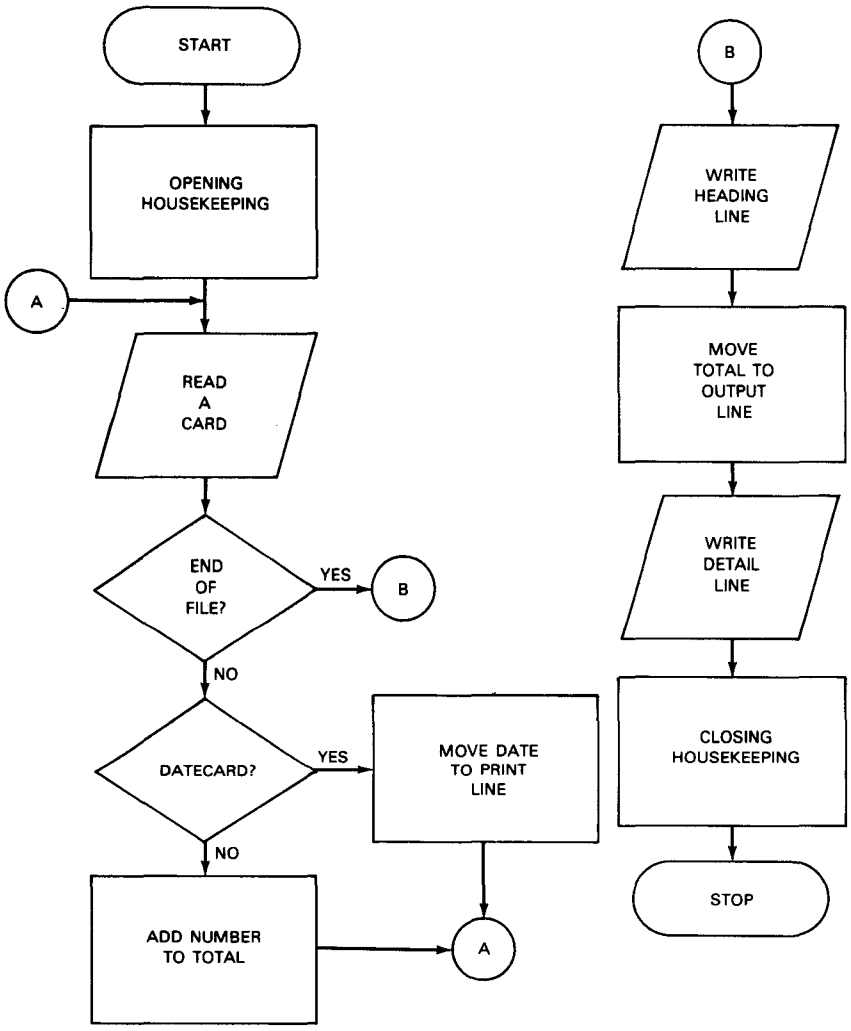


Figure 2—1. Flowchart for Sample COBOL Program

Now, we substitute COBOL statements for the blocks in the flowchart. We divide the procedure division into paragraphs headed by paragraph-names. The statements in the paragraphs are executed in the order you code them, unless you use a branching statement to redirect processing to a new paragraph.

To mark the start of the program (represented by START in the flowchart), all you need is:

```
PROCEDURE DIVISION.
```

and a name for the first paragraph such as

```
BEGIN-JOB.
```

There are two things to do as opening housekeeping. First, we use the OPEN statement to make the files CARDIN and PRINTOUT available for processing. You can't read from, or write to, a file until you open it. Thus, the statement

```
OPEN INPUT CARDIN OUTPUT PRINTOUT.
```

opens both input and output files.

Secondly, we need to initialize the accumulator field TOTAL to zero. A simple statement initiates the field:

```
MOVE ZEROS TO TOTAL.
```

Note that connector A is immediately above the flowchart block (READ A CARD) in Figure 2–1. The connector indicates we branch there from other points in the logic flow. Consequently, we must label that point of the program by including a new paragraph and paragraph header, such as

```
READ-CARD.
```

The first statement in READ-CARD

```
READ CARDIN AT END GO TO END-OF-JOB.
```

reads a record and tests for end-of-file. You use it to make the next record from CARDIN available in CARD-INPUT, and pass control to the next statement. If no record is available because that is the end of the file, control passes to the paragraph called END-OF-JOB (beginning at connector B in Figure 2–1).

The next step in the logic flow is to check to see whether CARD-INPUT contains a date or a number to be summed (DATECARD? in Figure 2-1). This is done by testing card column 1. If it is A, the record contains the date; if it is B, the record contains a number to be summed. The test for A or B, and the actions that result, are coded this way:

```
IF CARD-TYPE ='A'
   MOVE MONTH TO MONTH-OUT   MOVE DAY TO DAY-OUT   MOVE YEAR TO YEAR-OUT.
IF CARD-TYPE ='B'
   ADD ADDEND TO TOTAL.
```

When card column 1 is A, we move the date to the appropriate fields on the print line; when it's B, we add the number to the total and store the result in TOTAL.

After taking the information we need from the card, we read the next card. We branch by using the following statement:

```
GO TO READ-CARD.
```

This pattern of reading a card, checking for A or B, taking action, and returning for the next card continues until all cards have been read. At that point, control passes to the paragraph named in the AT END phrase, in this case:

```
END-OF-JOB.
```

In the end-of-job routine, we print the output, close the files, and end the program. We code it as follows:

```
END-OF-JOB.
   MOVE HEADING-LINE TO PRINTLINE.
   WRITE PRINTLINE.
   MOVE TOTAL TO SUM-OUT.
   MOVE SUM-LINE TO PRINTLINE.
   WRITE PRINTLINE.
   CLOSE CARDIN PRINTOUT.
   STOP RUN.
```

This routine builds the print lines in HEADING-LINE and SUM-LINE, the two record description areas we set up in working-storage; moves them to PRINTLINE (the output record area); and writes them to the printer. The CLOSE statement closes the files and the STOP RUN statement halts processing.

These are all the source code lines needed for this program. In 2.5, we discuss how to enter the source lines on a COBOL coding form. First, here's a look at the complete procedure division:

```
PROCEDURE DIVISION.
BEGIN-JOB.
     OPEN INPUT CARDIN OUTPUT PRINTOUT.
     MOVE ZEROS TO TOTAL.
READ-CARD.
     READ CARDIN AT END GO TO END-OF-JOB.
     IF CARD-TYPE ="A"
        MOVE MONTH TO MONTH-OUT
        MOVE DAY TO DAY-OUT
        MOVE YEAR TO YEAR-OUT.
     IF CARD-TYPE ="B"
        ADD ADDEND TO TOTAL.
     GO TO READ-CARD.
END-OF-JOB.
     MOVE HEADING-LINE TO PRINTLINE.
     WRITE PRINTLINE.
     MOVE TOTAL TO SUM-OUT.
     MOVE SUM-LINE TO PRINTLINE.
     WRITE PRINTLINE.
     CLOSE CARDIN PRINTOUT.
     STOP RUN.
```

2.5. COBOL CODING FORM

When you write a COBOL program, you generally enter the source code on a COBOL coding form to show how the code should look when it is input to the COBOL compiler. The American National Standard COBOL describes a format (called the reference format) for writing a COBOL source program. Because you have to follow that format when you put the source code lines onto the medium that will be input to the compiler, it makes sense to follow it when you write the source code onto a form.

The reference format (Figure 2–2) requires you to place certain source code in particular character positions of a source line. You can see that the character positions are grouped into four areas: (1) the Sequence Number Area, (2) the Indicator Area, (3) Area A, and (4) Area B. The areas are bounded by the various margins as shown.

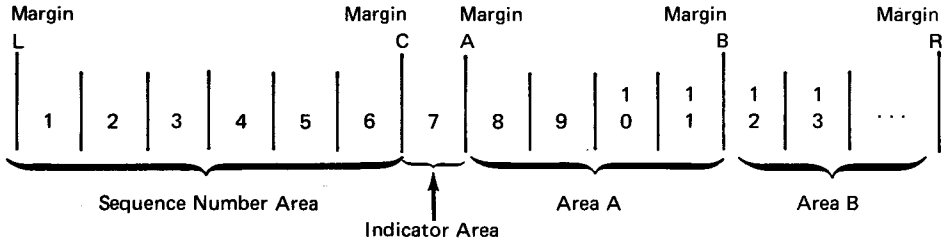


Figure 2—2. COBOL Reference Format

No upper limit of characters is specified; however, the most common COBOL coding form is designed to represent the source lines you enter on 80-column punched cards. Often, column 72 is designated as the last column in area B, and columns 73 through 80 are reserved for comments such as a program name or identification.

The COBOL compiler accepts the source lines as input, and produces, as output, an object program and a listing of your source lines.

Figure 2-3 is a typical COBOL coding form; it is designed for source code entered on 80-column punched cards. Each line on the form corresponds to one punched card. The first line in our program is the identification division header. A division header or section header must be the only entry on a line, and must begin in area A (columns 8 through 11).

On the first source card, we know we want the words IDENTIFICATION DIVISION beginning in column 8 (or 9 or 10 or 11). You may want to label the source lines with sequence numbers. If the program is on cards, the sequence numbers enable you to easily resequence a deck that was dropped or find where to add new cards.

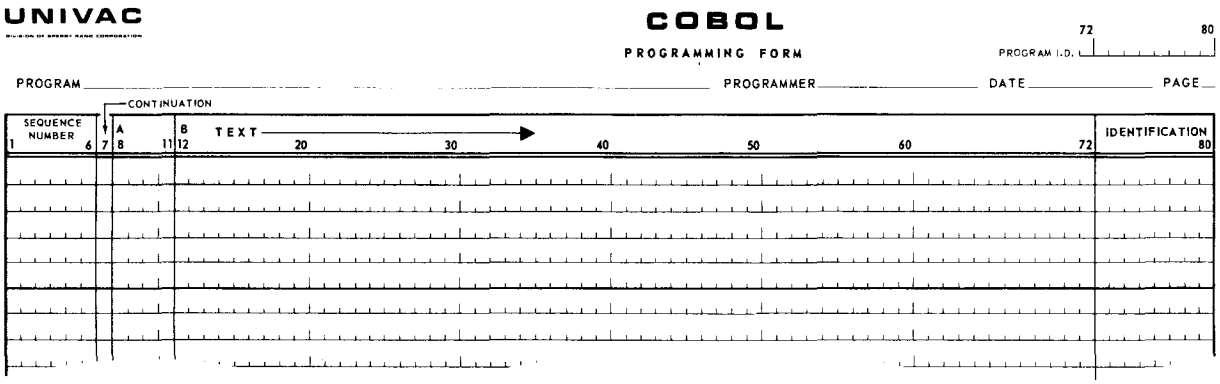


Figure 2—3. COBOL Coding Form

Generally, programmers assign sequence numbers in increments of 10 or 100, so numbers are available for lines they may insert later. If we code the first source line with the number 000010 in the sequence number area (columns 1–6), and if we use the program identification FINDSUM to the right of area B, the complete first entry is:

```
1       8   12                                                    80
000010 IDENTIFICATION DIVISION.                          FINDSUM
```

Although it is not shown in the other examples in this section, you can assume that FINDSUM will appear in columns 73 through 79 of all source lines. In addition, the following examples do not include the column numbers and area designations above the coding. You can assume each example in this section begins in column 1. In coding where sequence numbers are not used, you can assume column 8 is the leftmost column.

Paragraph headers (COBOL words that mark the start of paragraphs in the identification and environment divisions) and paragraph-names (user-defined words that identify the start of procedure division paragraphs) also must begin in area A. You can begin paragraph entries on the same line as the paragraph header or paragraph-name, as in the following coding:

```
000090 PROGRAM-ID.  FINDSUM.
000100 AUTHOR.  J. PROGRAMMER.
```

You can also begin the entries in area B on the next line:

```
000200 FILE-CONTROL.
000210     SELECT CARDIN
000220      ASSIGN TO CARD-READER.
000230     SELECT PRINTOUT
000240      ASSIGN TO PRINTER.
```

Following the rules of the reference format, you can also code these lines as

```
000090 PROGRAM-ID.
000100     FINDSUM.
000110 AUTHOR.
000120     J. PROGRAMMER.
```

and

```
000200 FILE-CONTROL. SELECT CARDIN
000210     ASSIGN TO CARD-READER.
000220     SELECT PRINTOUT ASSIGN
000230     TO PRINTER.
```

Note that in the first example the first SELECT entry begins on one line and continues on the next line. The reference format permits you to continue an entry on as many lines as you want or need, provided you start the continued lines in area B. In fact, to go to the extreme, you could code the SELECT entry this way:

```
000200 FILE-CONTROL.
000210     SELECT
000220     CARDIN
000230     ASSIGN
000240     TO
000250     CARD-READER.
```

Words also may be continued on the next line. If you code a hyphen in the indicator area (column 7), the first nonblank character in area B of the current line immediately follows the last nonblank character of the preceding line. Thus, the following source lines are equivalent:

```
000150     OPEN INPUT CARD
000160-    IN  OUTPUT PRIN
000170-    TOUT.


000150     OPEN INPUT CARDIN
000160          OUTPUT PRINTOUT.
```

Now you know how to code source lines in the identification, environment, and procedure divisions. Before explaining how to present data division entries, we will look at two types of source lines you can use to make the source program output listing more readable – comment lines and blank lines (2.5.1 and 2.5.2).

2.5.1. Comment Line

The first type is the comment line. On a comment line (identified by an asterisk in the indicator area, column 7) you can write explanatory notes to the reader of the source listing, possibly to explain a complex routine that follows or to make certain portions of the program easy to find. Comments can appear anywhere in the source program after the identification division header. You can write the comment in area A and area B of the line, and you can use any characters in the computer character set.

One common location to make comments is just after the identification division header, where it is appropriate to give information about the program as a whole. For example:

```
000010 IDENTIFICATION DIVISION.
000020* THIS PROGRAM READS A DATE CARD,
000030* THEN A VARIABLE NUMBER OF ADDITIONAL
000040* CARDS, EACH CONTAINING A NUMBER TO BE
000050* SUMMED. THE PROGRAM OUTPUT IS A
000060* PRINTED REPORT THAT GIVES THE DATE AND
000070* THE SUM OF THE NUMBERS ON THE CARDS.
000080
000090  PROGRAM-ID.    FINDSUM.
```

A special type of comment line allows you to print the comment on a new page in the program listing. To do this, code a stroke rather than an asterisk in the indicator area of the line:

```
000020/ THIS PROGRAM READS A DATE
000030* CARD, THEN A VARIABLE NUMBER
000040* OF ADDITIONAL CARDS, EACH...
```

Although similar in purpose, a comment line isn't the same as a comment-entry. A comment-entry must follow the header for an optional paragraph in the identification division; a comment line can be anywhere in the source program.

2.5.2. Blank Line

The second type of source line that can improve the readability of a program listing is the blank line – a line that has no characters from margin C through margin R. You might want to use blank lines (or comment lines) to separate groups of logically related source lines (such as paragraphs in the procedure division).

```
000100 AUTHOR.          J. PROGRAMMER.
000110 INSTALLATION.    XYZ COMPANY
000120                  NEW YORK.
000130
000140
000150 ENVIRONMENT DIVISION.
```

This shows how you can use a blank line to separate the identification and environment division in a program.

You can use a blank line anywhere in the source program, except preceding a continuation that has a hyphen character in column 7. Thus, the following is not legal:

```
000500 FILE-CONTROL.  SELECT STATUS
000510
000520       TICS ASSIGN TO DISC.
```

Comment lines and blank lines are for documentation only; they appear on the program listing but are not compiled.

2.5.3. Level Indicators/Numbers

Now, let us return to the data division. The division and section headers must, as in the other divisions, be on a line by themselves and must begin in area A. All other data division entries begin with one of the level indicators (FD, SD, and CD) or with a level-number (01 through 49, 66, 77, and 88).

If a data division entry begins with a level indicator or with level-number 01 or 77, you must code the indicator or number in area A, followed by a space, and followed in area B by the rest of the entry.

Entries are always ended by a period and a space. Thus, the "rest of the entry" in the first source line that uses level-number 01 is CARD-INPUT and it must begin in area B. The complete entry is coded this way:

```
000200 01  CARD-INPUT.
```

The full entry following the first FD extends for several lines. Still, the entire entry after the letters FD must start in area B, as follows:

```
000160 FD  CARDIN
000170     RECORD CONTAINS 80 CHARACTERS
000180     LABEL RECORDS ARE OMITTED
000190     DATA RECORDS IS CARD-INPUT.
```

If a data description entry begins with level-numbers 02 through 49, 66, or 88, you code it the same way you code 01 and 77 level entries or, to aid readability, you can indent it according to level-number. Thus, you might begin an 02 level entry in column 12, an 03 level entry in column 16, an 04 level entry in column 20, etc. The CARD-INPUT record description is written

```
000200 01  CARD-INPUT.
000210 02  CARD-TYPE  PIC X.
000220 02  DATE.
000230 05  MONTH  PIC 9(2).
000240 05  DAY    PIC 9(2).
000250 05  YEAR   PIC 9(2).
000260 02  ADDEND PIC 9(3).
000270 02  FILLER PIC X(70).
```

Or, using indentation of levels:

```
000200 01  CARD-INPUT.
000210     02  CARD-TYPE PIC X.
000220     02  DATE.
000230         05  MONTH PIC 9(2).
000240         05  DAY   PIC 9(2).
000250         05  YEAR  PIC 9(2).
000260     02  ADDEND  PIC 9(3).
000270     02  FILLER  PIC X(70).
```

As you can see, level-number 05 is used in place of 03 to represent the third level of the record. Actually, after the 01 level entry, you can use any level-number up to 49 for succeeding entries. Just be certain you use the numbers in ascending sequence. For example, you can use 18 and 37 in place of 02 and 05, respectively.

2.6. SAMPLE PROGRAM

We have described how to develop a COBOL program and how to submit it (in reference format) for processing. A complete program is shown as it appears on a coding form designed to represent source lines that are entered from 80-column punched cards. Note the use of comment lines, blank lines, and indentation of level-numbers to improve readability of the program listing. Each line starts in card column 1; the identification FINDSUM begins in column 73.

```
000010 IDENTIFICATION DIVISION.                                  FINDSUM
000020* THIS PROGRAM READS A DATE CARD,                          FINDSUM
000030* THEN A VARIABLE NUMBER OF ADDITIONAL                     FINDSUM
000040* CARDS, EACH CONTAINING A NUMBER TO BE                    FINDSUM
000050* SUMMED. THE PROGRAM OUTPUT IS A                          FINDSUM
000060* PRINTED REPORT THAT GIVES THE DATE AND                   FINDSUM
000070* THE SUM OF THE NUMBERS ON THE CARDS.                     FINDSUM
000080                                                           FINDSUM
000090 PROGRAM-ID.     FINDSUM.                                   FINDSUM
000100 AUTHOR.         J. PROGRAMMER.                             FINDSUM
000110 INSTALLATION.   XYZ COMPANY                                FINDSUM
000120                 NEW YORK.                                  FINDSUM
000130                                                           FINDSUM
000140                                                           FINDSUM
000150 ENVIRONMENT DIVISION.                                      FINDSUM
000160 CONFIGURATION SECTION.                                     FINDSUM
000170 SOURCE-COMPUTER.    COMPUT-001.                            FINDSUM
000180 OBJECT-COMPUTER.    COMPUT-001.                            FINDSUM
000190 INPUT-OUTPUT SECTION.                                      FINDSUM
000200 FILE-CONTROL.                                              FINDSUM
000210     SELECT CARDIN ASSIGN TO CARD-READER.                   FINDSUM
000220     SELECT PRINTOUT ASSIGN TO PRINTER.                     FINDSUM
000230                                                           FINDSUM
```

(continued)

```
000240                                                          FINDSUM
000250 DATA DIVISION.                                           FINDSUM
000260 FILE SECTION.                                            FINDSUM
000270******** INPUT FILE DESCRIPTION ********                  FINDSUM
000280 FD  CARDIN                                               FINDSUM
000290     RECORD CONTAINS 80 CHARACTERS                        FINDSUM
000300     LABEL RECORDS ARE OMITTED                            FINDSUM
000310     DATA RECORD IS CARD-INPUT.                           FINDSUM
000320 01  CARD-INPUT.                                         FINDSUM
000330     02  CARD-TYPE  PIC X.                               FINDSUM
000340     02  DATE.                                           FINDSUM
000350         05  MONTH  PIC 9(2).                            FINDSUM
000360         05  DAY    PIC 9(2).                            FINDSUM
000370         05  YEAR   PIC 9(2).                            FINDSUM
000380     02  ADDEND      PIC 9(3).                           FINDSUM
000385     02  FILLER        PIC X(70).                        FINDSUM
000390******** OUTPUT FILE DESCRIPTION ********                 FINDSUM
000400 FD  PRINTOUT                                             FINDSUM
000410     RECORD CONTAINS 132 CHARACTERS                       FINDSUM
000420     LABEL RECORDS ARE OMITTED                            FINDSUM
000430     DATA RECORD IS PRINTLINE.                            FINDSUM
000440 01  PRINTLINE      PIC X(132).                          FINDSUM
000450                                                          FINDSUM
000460 WORKING-STORAGE SECTION.                                 FINDSUM
000470 77  TOTAL    PIC 9(6).                                  FINDSUM
000480                                                          FINDSUM
000490 01  SUM-LINE.                                           FINDSUM
000500     02  FILLER       PIC X(27)   VALUE SPACES.          FINDSUM
000510     02  MONTH-OUT    PIC 9(2).                          FINDSUM
000511     02  FILLER       PIC X  VALUE ''/''.                FINDSUM
000512     02  DAY-OUT      PIC 9(2).                          FINDSUM
000513     02  FILLER       PIC X  VALUE ''/''.                FINDSUM
000514     02  YEAR-OUT     PIC 9(2).                          FINDSUM
000520     02  FILLER       PIC X(25)   VALUE SPACES.          FINDSUM
000530     02  MSG-OUT      PIC X(12)   VALUE ''THE TOTAL IS''.FINDSUM
000540     02  FILLER       PIC X(3)    VALUE SPACES.          FINDSUM
000550     02  SUM-OUT      PIC 999,999.                       FINDSUM
000560     02  FILLER       PIC X(50)   VALUE SPACES.          FINDSUM
000570                                                          FINDSUM
000580 01  HEADING-LINE.                                       FINDSUM
000590     02  FILLER       PIC X(60)   VALUE SPACES.          FINDSUM
000600     02  FILLER       PIC X(13) VALUE ''SAMPLE REPORT''. FINDSUM
000610     02  FILLER       PIC X(59)   VALUE SPACES.          FINDSUM
000620                                                          FINDSUM
000630                                                          FINDSUM
000640 PROCEDURE DIVISION.                                      FINDSUM
000650 BEGIN-JOB.                                               FINDSUM
000660     OPEN INPUT CARDIN OUTPUT PRINTOUT.                   FINDSUM
000670     MOVE ZEROS TO TOTAL.                                 FINDSUM
```

(continued)

```
000680                                                          FINDSUM
000690 READ-CARD.                                               FINDSUM
000700     READ CARDIN AT END GO TO END-OF-JOB.                 FINDSUM
000710****   CHECK TO SEE IF RECORD IS DATE CARD OR             FINDSUM
000720****   NUMBER TO BE SUMMED                                FINDSUM
000730     IF CARD-TYPE = ''A''                                 FINDSUM
000740         MOVE MONTH TO MONTH-OUT.                         FINDSUM
000741         MOVE DAY TO DAY-OUT.                             FINDSUM
000742         MOVE YEAR TO YEAR-OUT.                           FINDSUM
000750     IF CARD-TYPE = ''B''                                 FINDSUM
000760         ADD ADDEND TO TOTAL.                             FINDSUM
000770     GO TO READ-CARD.                                     FINDSUM
000780                                                          FINDSUM
000790 END-OF-JOB.                                              FINDSUM
000800*****   BUILD AND WRITE FIRST OUTPUT LINE                 FINDSUM
000810     MOVE HEADING-LINE TO PRINTLINE.                      FINDSUM
000820     WRITE PRINTLINE.                                     FINDSUM
000830*****   BUILD AND WRITE SECOND OUTPUT LINE                FINDSUM
000840     MOVE TOTAL TO SUM-OUT.                               FINDSUM
000850     MOVE SUM-LINE TO PRINTLINE.                          FINDSUM
000860     WRITE PRINTLINE.                                     FINDSUM
000870*** CLOSE FILES AND END JOB                               FINDSUM
000880     CLOSE CARDIN PRINTOUT                                FINDSUM
000890     STOP RUN.                                            FINDSUM
```

3. Language Structure

3.1. COBOL CHARACTER SET

The character is the basic unit of the language. You combine characters to form the words and separators that make up the source lines. Every column in every source line you code must contain one of the letters, digits, or special characters from the COBOL character set.

The 51-character COBOL set consists of the 26 letters of the alphabet, the digits 0–9, and the following 15 special characters:

Character	Definition
	Space (blank)
+	Plus sign
-	Minus sign or hyphen
*	Asterisk
/	Stroke (virgule)
=	Equal sign
$	Currency sign
,	Comma
;	Semicolon
.	Period
"	Quotation mark

Character	Definition
(	Left parenthesis
)	Right parenthesis
>	Greater than symbol
<	Less than symbol

The first line of a program is often

```
000001 IDENTIFICATION DIVISION.
```

You can see that every column contains one of the 51 members of the character set.

Sometimes you have more than 51 characters to choose from. If your computer system prints special characters not included in the COBOL character set, you can use those characters in a comment line, comment-entry, or nonnumeric literal (3.4). For instance, if the cent (¢) and percentage (%) symbols are in the computer's character set, then the coding statements

```
010350* THE RATE IS 40¢/LB.
```

and

```
07190 MOVE ''%'' TO PERCENT-SIGN.
```

are valid, even though neither ¢ or % is in the COBOL character set.

Continguous characters forming COBOL words (3.3), literals (3.4), PIC clause specifications, or comment-entries are called character-strings. For example, the coding

```
MOVE MONTH TO MONTH-OUT
```

has four character-strings (MOVE, MONTH, TO, MONTH-OUT). The character strings in

```
PIC 9(3)
```

are PIC and 9(3).

The comment-entry in this INSTALLATION paragraph

```
INSTALLATION. XYZ COMPANY NEW YORK.
```

is one long character string consisting of XYZ, COMPANY, NEW YORK, and the blanks between them. Note that MOVE, MONTH, TO, and MONTH-OUT are separate character-strings while X, Y, Z, C, O, M, P, etc. are merely characters in one character-sting. MONTH and MONTH-OUT are words representing particular storage areas. MOVE and TO are COBOL reserved words (3.3) used to cause a specific operation. XYZ, COMPANY, NEW, and YORK, on the other hand, are not COBOL words. They are part of a combination of letters and spaces that is documentation only; they do not represent storage or convey action.

3.2. SEPARATORS

You use separators to separate character-strings. The most common separator is the space. Anywhere you use a space as a separator, you can use more than one space, thus the following coding lines are equivalent:

```
MOVE MONTH TO MONTH-OUT
MOVE    MONTH      TO   MONTH-OUT.
```

The period also is a separator. You must use a period, followed by a space, to end a division, a paragraph, and section headers in the identification and procedure divisions and entries in the environment and data divisions. Since COBOL is an English-like language, you may also want to use a period to end each sentence in the identification and procedure divisions. Thus, it's proper to code

```
MOVE SUM-LINE TO PRINTLINE
WRITE PRINTLINE
CLOSE CARDIN PRINTOUT
STOP RUN.
```

or

```
MOVE SUM-LINE TO PRINTLINE.
WRITE PRINTLINE.
CLOSE CARDIN PRINTOUT.
STOP RUN.
```

You also may punctuate your sentences with commas and semicolons, as in

```
MOVE SUM-LINE TO PRINTLINE;
WRITE PRINTLINE;
CLOSE CARDIN, PRINTOUT;
STOP RUN.
```

Commas and semicolons are separators when immediately followed by a space. Most programmers don't use them because they add little to the clarity of the program; they just made coding and keypunch errors more likely.

Quotations marks and parentheses also are separators. Quotation marks, as shown in 3.4, delimit nonnumeric literals. Parentheses delimit subscripts, indexes, and arithmetic expressions. (These subjects are discussed separately.)

The parentheses in PIC 9(3) are not separators. They, like all punctuation characters in a PIC clause entry, are symbols that have a special meaning in the PIC character-string specification (6.2.2.)

3.3. COBOL WORDS

You use English words to communicate with other people; similarly, you use COBOL words (along with literals and other character-strings) to communicate with a COBOL compiler. There are three types of COBOL words: (1) words you supply, (2) words specified by the implementor (computer manufacturer), and (3) words predefined by the COBOL language.

In general, you supply words that vary from program to program. the names of files, records, and other data items (and the procedure division's paragraph-names) all are user-defined words.

The implementor supplies system-names. They identify the source and object computers, physical I-O devices, and other aspects of the operating environment.

Predefined COBOL words are called reserved words. You already know many of them – in Section 2 we used some in every COBOL division. Since a reserved word already has a specific meaning in COBOL, you can't use one where a user-defined word or system name is called for. The reserved words are listed in the current version of the programmer reference manual.

User-defined words and system names must be comprised of not more than 30 letters, digits, and hyphens (-) except that a hyphen may not be the first or last character. Thus, valid words include:

```
MONTH-OUT
PARA25-2
```

Invalid words include:

```
-MONTH-OUT
PARA25/2$
```

A statement from the environment division in Section 2 such as

```
SELECT PRINTOUT ASSIGN TO PRINTER
```

includes all three types of COBOL words:

- SELECT, ASSIGN, and TO are reserved words
- PRINTOUT is a user-defined word
- PRINTER is the implementor's name for a printer.

3.4. LITERALS

A literal is a character-string whose value is "literally" the characters which form it. That value may be numeric or nonnumeric.

The program described in Section 2 contains several nonnumeric literals. In the following statement:

```
Ø2 MSG-OUT PIC X (12) VALUE ''THE TOTAL IS''.
```

the literal is:

```
THE TOTAL IS
```

In

```
02 FILLER PIC X VALUE ''/''.
```

The literal is

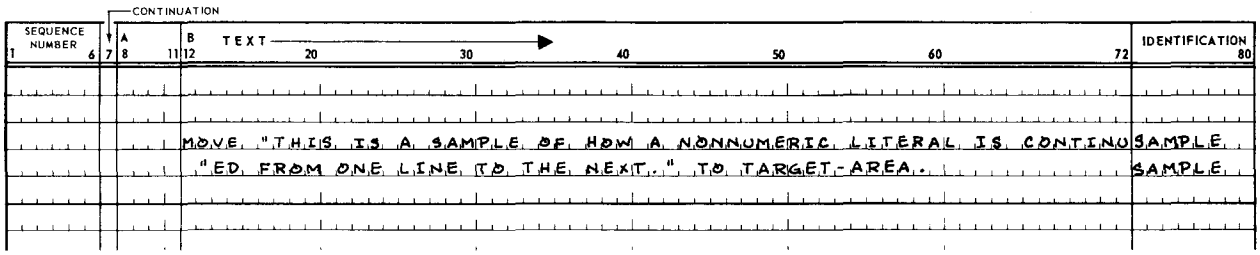
```
/
```

You must enclose all nonnumeric literals with quotation marks. If you want to make a quotation mark part of the literal, code two quotation marks together. Thus,

```
Ø2 LETTER PIC X (5) VALUE ''A''''A''''A''.
```

assigns the value A"A"A to the area referenced by LETTER.

A nonnumeric literal may consist of up to 120 characters and include any character in the computer's character set. If you code a nonnumeric literal that is too long for one source line, place a hyphen in the indicator area (column 7) of the next line and code a quotation mark as the first nonblank character in area B. The continuation starts with the character immediately following that quotation mark. Be certain to use another quotation mark to end the literal. For example:



Numeric literals consist of up to 18 digits and, optionally, a plus sign or minus sign, and one decimal point. If you include a decimal point, it must not be the rightmost character. If you do not include a decimal point, the literal is an integer. The sign must be the leftmost character of the literal; unsigned literals are positive.

Thus, in

```
ADD 1 TO COUNTER
```

the 1 is a numeric literal. Note that 1 is not enclosed within quotation marks. If it is, it would be a nonnumeric literal with a value of the character 1 (rather than the number).

The following examples are valid numeric literals:

```
MOVE +542.8 TO OUT-AREA.
DIVIDE -Ø.425 BY CURR-VALUE
ADD -21436 TO TOTAL
```

The following literals are invalid:

```
MOVE 542.8- TO OUT-AREA
DIVIDE -425. BY CURR-VALUE
ADD 2143.-6 TO TOTAL
```

3.5. FIGURATIVE CONSTANTS

A figurative constant is a reserved word that replaces a literal. You are never required to use one, but they can simplify coding. For instance, to initialize the 10-character field FIELD-OUT to blanks, you code

```
MOVE ''△△△△△△△△△△'' TO FIELD-OUT
```

Or, using the figurative constant spaces to replace the nonnumeric literal "△△△△△△△△△△", you code

```
MOVE SPACES TO FIELD-OUT.
```

The word ZEROS also is a figurative constant. In Section 2, we coded

```
MOVE ZEROS TO TOTAL
```

to initialize the field TOTAL. We could have coded

```
MOVE Ø TO TOTAL
```

but ZEROS is used more often – probably because it's more readable. The terms ZEROS and SPACES are the most common figurative constants, but several others are available.

The terms HIGH-VALUES and LOW-VALUES represent, respectively, the highest and lowest ordinal positions in a particular computer character set. If you code

```
Ø1 TABLE.
       Ø5 TABLE-ENTRY OCCURS 5ØØ TIMES.
                  PIC 9(5) VALUE HIGH-VALUES.
```

The highest values in the computer collating sequence are placed in the 2500 characters (500 occurrences of 5-character entries) referenced by TABLE.

You can use the word QUOTES to represent one or more of the quotation mark character ("). You cannot use this figurative constant, however, to bound nonnumeric literals. Thus,

```
QUOTE THE TOTAL IS QUOTE
```

is not a valid replacement for

```
''THE TOTAL IS''.
```

The figurative constant ALL followed by a literal represents one or more of the string of characters comprising the literal. The literal must be nonnumeric or a figurative constant other than ALL. If it is a figurative constant, the word ALL is redundant and therefore for readability only.

The entry

```
05 AST-OUT PIC X(5) VALUE '*****'.
```

can (using the ALL figurative constant) be coded

```
05 AST-OUT PIC X(5) VALUE ALL '*'.
```

Similarly, you might find it easier to code the following

```
02 LETTERS-OUT PIC X(12) VALUE 'ABCABCABCABC'.
```

as

```
02 LETTERS-OUT PIC X(12) VALUE ALL 'ABC'.
```

The length of a figurative constant is dependent upon the data item it is associated with. If you specify MOVE ZEROS TO TOTAL and TOTAL is defined

```
01 TOTAL PIC 9(9).
```

nine zeros are moved to TOTAL. If TOTAL is defined

```
01 TOTAL PIC 9(15).
```

fifteen zeros are moved to TOTAL. If no data item is associated with a figurative constant, the length of the constant is one. Thus

```
DISPLAY QUOTES UPON CONSOLE
```

displays one quotation mark on the console. The following listing summarizes the figurative constants and their values. The singular and plural forms of the constants are equivalent; you may use them interchangeably.

Constant	Value
SPACE SPACES	One or more blanks.
ZERO ZEROS ZEROES	The numeric value 0 or one or more of the character 0, depending on the context.
HIGH-VALUE HIGH-VALUES	One or more of the highest ordinal positions in the computer's character set collating sequence.
LOW-VALUE LOW-VALUES	One or more of the lowest ordinal positions in the computer's character set collating sequence.

Constant	Value
QUOTE QUOTES	One or more of the character ".
ALL literal	One or more of the string of characters comprising the literal.

3.6. GENERAL FORMATS

Once you know that COBOL source lines are comprised of words, literals, PIC character-strings, and comment-entries, general formats show you how to combine these elements into a clause or statement that is meaningful. There is a general format that tells you how to write every entry in a COBOL program.

We knew how to arrange the words

```
SELECT PRINTOUT ASSIGN TO PRINTER
```

because it is specified in the general format for file control entries. Let's look at that format:

```
SELECT [OPTIONAL] file-name
   ASSIGN TO implementor-name-1 [implementor-name-2] ...
   [RESERVE INTEGER-1 [AREA ]]
                      [AREAS]
   [ORGANIZATION IS SEQUENTIAL]
   [ACCESS MODE IS SEQUENTIAL]
   [FILE STATUS IS data-name-1].
```

The uppercase words are reserved words. If they are underlined (see SELECT and ASSIGN), they are called key words and must be included in the clause or statement. Reserved words not underlined (such as TO, IS, and MODE) are optional – you may include them to improve readability. The lowercase words such as file-name, implementor-name-1, integer-1, etc. are generic terms you must replace with a character-string. In the statement

```
SELECT PRINTOUT ASSIGN TO PRINTER
```

we replaced file-name with the word PRINTOUT, and implementor-name-1 with PRINTER.

Brackets [ ] enclose optional words, phrases, and clauses. Thus you may (as we did) omit OPTIONAL, implementor-name-2, and the last four clauses in the SELECT entry. When brackets enclose more than one choice, select one or none of the choices. Thus the following are all equivalent and valid:

```
RESERVE 2 AREA
RESERVE 2 AREAS
RESERVE 2
```

Note that the optional implementor-name-2 is followed by an ellipsis (...) An ellipsis indicates you may repeat the option as many times as necessary, in this case providing for an implementor-name-3, implementor-name-4, etc.

The ellipsis always refers to the option whose closing bracket or brace immediately precedes it. Thus, the first ellipsis in

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

refers to file-name-2 and means you may specify more than two file names. The second ellipsis refers to the brackets enclosing the entire clause, indicating you may repeat the clause as often as necessary.

Braces { } enclose a stack of options from which you must select one. For example:,

```
ENABLE {INPUT  [TERMINAL]} cd-name WITH KEY {identifier-1}
       {OUTPUT           }                  {literal-1   }
```

You must choose between INPUT and OUPUT and between identifier-1 and literal-1. (Note that TERMINAL is an option if you choose INPUT, but not if you choose OUTPUT).

If a single choice is enclosed in braces as in

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

the enclosed word or phrase is mandatory - the braces delimit the element to which the following ellipsis applies. Since an ellipsis refers to the immediately preceding brace, the braces in this format indicate that the first ellipsis refers to file-name-2, not to the choice of RECORD, SORT, or SORT-MERGE.

The leftmost margin of the general formats for the identification, environment, and data division entries is equivalent to margin A of a source line, and the first identation is equivalent to margin B. Statements in the procedure division begin in area B. The leftmost margin of a procedure division general format marks the beginning of the format for a new verb. The indented portions represent continuation of the verb.

3.7. SUBSCRIPTING AND INDEXING

Each COBOL word you specify must be unique; however, you do not have to assign individual data-names to data items that you can make unique by adding a subscript, an index, or a qualifier (3.8).

In many programs, you reference information that is in tabular form. In the following table, we assign a data-name to each table element:

```
01 MONTHS-TABLE.
   02  JAN  PIC X(9) VALUE  ''△JANUARY△''.
   02  FEB  PIC X(9) VALUE  ''△FEBRUARY''.
   02  MAR  PIC X(9) VALUE  ''△△MARCH△△''.
   02  APR  PIC X(9) VALUE  ''△△APRIL△△''.
   02  MAY  PIC X(9) VALUE  ''△△△MAY△△△''.
   02  JUN  PIC X(9) VALUE  ''△△△JUNE△△''.
   02  JUL  PIC X(9) VALUE  ''△△△JULY△△''.
   02  AUG  PIC X(9) VALUE  ''△△AUGUST△''.
   02  SEP  PIC X(9) VALUE  ''SEPTEMBER''.
   02  OCT  PIC X(9) VALUE  ''△OCTOBER△''.
   02  NOV  PIC X(9) VALUE  ''△NOVEMBER''.
   02  DEC  PIC X(9) VALUE  ''△DECEMBER''.
```

We could simplify coding by assigning the same name to all table elements, as in

```
01 MONTHS-TABLE.
   02 TAB-VAL  PIC  X(108).
          VALUE ''△JANUARY△△FEBRUARY△△MARCH△△
          ''△△APRIL△△△△△MAY△△△△△△JUNE△△△△△
          ''△△△△AUGUST△SEPTEMBER△OCTOBER△△NOVE
          ''MBER△DECEMBER''.
   02  REDF-VAL REDEFINES TAB-VAL.
       03 MONTH PIC X(9).
              OCCURS 12 TIMES.
```

Each table element still is unique, because we can refer to it as MONTH plus a subscript or index. The subscript or index tells which of the 12 occurrences of MONTH we are referencing. Thus, MONTH (3) equals △△MARCH△△ (the third occurrence of the 9-character field MONTH). The parentheses are separators that delimit a subscript or index from the data-name. A space between the last character of the data-name and the left parenthesis is optional. (Subscripting and indexing are also also discussed in Section 8.)

3.8. QUALIFICATION

Another way to make a name unique is to attach a qualifier to it. A qualifier identifies a name as being part of a particular hierarchy of names.

Qualifiers in the data division are associated with a level indicator (such as FD) or with a level-number.

If

```
FD  CARDIN
    LABEL RECORDS ARE OMITTED.
01  INPUT      PIX X(80).
```

and

```
FD  TAPEIN
    LABEL RECORDS ARE STANDARD.
01  INPUT      PIC X(80).
```

appear in the same program, a reference to INPUT is meaningless unless INPUT is associated with a qualifier (CARDIN or TAPEIN). Thus, you must reference INPUT as

```
INPUT OF CARDIN
```

or

```
INPUT OF TAPEIN
```

Names associated with level-indicators are the most significant qualifiers, names associated with level-number 01 are the next most significant, followed by level-number 02, and so on. Suppose your FD entries are:

Entry 1:

```
FD  CARDIN
    LABEL RECORDS ARE OMITTED.
01  INPUT-CARD.
    02  NAME-1.
        03  ADDRESS     PIC X(20).
        03  CITY        PIC X(18).
        03  STATE       PIC X(02).
    02  NAME-2.
        03  ADDRESS     PIC X(20).
        03  CITY        PIC X(18).
        03  STATE       PIC X(02).
```

Entry 2:

```
FD  TAPEIN
    LABEL RECORDS ARE STANDARD.
Ø1  INPUT-TAPE.
    Ø2  NAME-TAPE.
        Ø3  ADDRESS  PIC X(2Ø).
        Ø3  CITY     PIC X(18).
        Ø3  STATE    PIC X(Ø2).
    Ø2  NAME-2
        Ø3  ADDRESS  PIC X(2Ø).
        Ø3  CITY     PIC X(18).
        Ø3  STATE    PIC X(Ø2).
```

You can see that you have to qualify a reference to any of these 02 or 03-level data-names. To do so, you can use any qualifier in the same hierarchy; both

```
NAME-1  IN  INPUT-CARD
```

and

```
NAME-1  IN CARDIN
```

refer to the same data item. (Note the use of IN rather than OF – the two are equivalent.)

The term

```
NAME-1 OF INPUT-CARD OF CARDIN
```

also is valid, although it is not necessary, in this case, to include both qualifiers. You do need two qualifiers, however, to reference ADDRESS. Thus,

```
ADDRESS OF INPUT-CARD
```

or

```
ADDRESS OF CARDIN
```

do not show whether you mean ADDRESS subordinate to NAME-1 or ADDRESS subordinate to NAME-2.

The term

```
ADDRESS OF NAME-2.
```

also is insufficient.

You need to qualify both NAME-2 and ADDRESS:

```
ADDRESS OF NAME-2 OF INPUT-CARD
```

or

```
ADDRESS OF NAME-2 OF CARDIN
```

When you use more than one qualifier, be certain each is more significant and in the same hierarchy as the preceding qualifier.

You may qualify a data-name that is subscripted or indexed. For instance, a valid reference is:

```
MONTH IN MONTHS-TABLE(3)
```

In the procedure division, you can (if you group your paragraphs into sections) use section-names to qualify paragraph-names. Generally, you are not required to divide your procedure division into sections (note that this was not done in the program in Section 2) but sometimes you must, to take advantage of some COBOL features.

You cannot use the same paragraph-name more than once in a section, but you can use a paragraph-name in one section that is identical to a paragraph-name in a different section. So, if your program contains

```
PROCEDURE DIVISION.
REGION-A-PROCESSING SECTION.
.
.
.
ERROR-ROUTINE.
.
.
.
REGION-B-PROCESSING SECTION.
.
.
.
ERROR-ROUTINE.
.
.
.
```

you must code a branch to the paragraph ERROR-ROUTINE such as

```
GO TO ERROR-ROUTINE OF REGION-A-PROCESSING
```

or

```
GO TO ERROR-ROUTINE OF REGION-B-PROCESSING
```

Programmers often use qualification as a documentation tool. The term

```
FIRE-COVERAGE-AMT OF INSURANCE-RECD
```

is clearer than

```
INS-RECD-FIRE-COV-AMT.
```

In fact, you may qualify a name that does not even need to be qualified. Sometimes, however, qualification is not permitted; these instances are noted, when applicable.

Qualification is not permitted in the identification or environment divisions.

4. Identification Division

4.1. ORGANIZATION AND STRUCTURE

All COBOL programs must begin with an identification division. You use it to name the program and, at your option, tell when and where it was written, who wrote it, when it was last compiled, and who may see it.

The identification division consists of six paragraphs; only one – the PROGRAM-ID paragraph – is required. You must code the paragraphs in the order shown in the format.

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

In Section 2, we coded

```
PROGRAM-ID. FINDSUM.
```

to name the program FINDSUM. We could have selected any user-defined word; that is, any combination of letters, digits, and hyphens not exceeding 30 characters. You may prefer, however, to use a relatively short name. Although up to 30 characters is acceptable, many operating systems disregard all but the first six to eight characters of program-name, thus requiring those characters to be unique within your installation. If the implementor does limit the length of program-name, characters beyond the limit are treated as comments.

The other identification division paragraphs allow you to make comments about the program. While it is expected you will use the DATE-WRITTEN paragraph to establish when you wrote the program, the SECURITY paragraph to determine who may see the program, etc. it actually doesn't make any difference what you code as comment-entries; they are for documentation only and do not affect the compilation. Comment-entries consist of any characters in the computer (not just COBOL) character set.

The DATE-COMPILED paragraph is a little different than the other optional paragraphs – its comment-entry is replaced at compilation time by the system date. For example, if your program includes the source line

```
DATE-COMPILED. DATE WILL APPEAR HERE.
```

and you compile the program on 01/01/80, the actual line appears as

```
DATE-COMPILED. Ø1/Ø1/8Ø
```

Comment-entries in the other paragraphs appear on the listing exactly as you code them.

4.2. CODING EXAMPLE

Here is an example of a complete identification division:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. EDTINV.
AUTHOR. J.  PROGRAMMER
        L.  PROGRAMMER.
INSTALLATION.  XYZ CORP
               NEW YORK.
DATE-WRITTEN.  12/15/8Ø.
DATE-COMPILED. COMPILATION DATE.
SECURITY.      CLASS 2-C.
```

This assigns EDTINV as the program-name. Comment-entries specify the authors, the installation, the date the program was written, and the security classification. The system date is placed in the DATE-COMPILED paragraph each time the program is compiled.

5. Environment Division

5.1. ORGANIZATION AND STRUCTURE

In the environment division, you relate hardware to software; that is, you describe the physical devices needed to compile the source program and execute the object program. Since the environment division is hardware-related, the implementor defines much of its content. Consequently, you may have to revise it extensively to make a program transportable.

There are two sections in the environment division: (1) the CONFIGURATION SECTION that describes the source and object computers and name implementor-functions, character sets, and collating sequences; and (2) the INPUT-OUTPUT SECTION that names the files, describes the devices that contain them, and specifies special processing techniques. Each of the sections consists of the paragraphs shown:

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

5.2. SOURCE-COMPUTER AND OBJECT-COMPUTER PARAGRAPHS

The computer manufacturer specifies the computer-name entries for these paragraphs. While computer-name completes the source-computer-entry, the object-computer-entry has two optional clauses.

The formats are:

```
SOURCE-COMPUTER.   computer-name.
OBJECT-COMPUTER.   computer-name.
     MEMORY SIZE integer [WORDS     ]
                         [CHARACTERS]
                         [MODULES   ]
     PROGRAM COLLATING SEQUENCE IS alphabet-name.
```

Many compilers regard the MEMORY SIZE option as documentation only; you use it to specify the main storage capacity (expressed in WORDS, CHARACTERS, or MODULES) of the object computer. If the capacity is not sufficient to run the object program, the implementor defines what is done.

Each computer's native collating sequence determines the relative values of the members of its character set. It is used to find the truth value of the nonnumeric comparisons in your program. So, the expression

```
IF A IS LESS THAN B
```

is true if the value of A precedes B in the collating sequence. A collating sequence can be associated with an alphabet-name. If you don't want to use the native collating sequence for your program, code an alphabet-name in the PROGRAM COLLATING SEQUENCE clause, then use the SPECIAL-NAMES paragraph (5.3) to assign that alphabet-name to a new collating sequence. The new collating sequence replaces the native collating sequence for the duration of your program.

5.3. SPECIAL-NAMES PARAGRAPH

The SPECIAL-NAMES paragraph enables you to name implementor functions, identify character sets and collating sequences, select a replacement for the dollar sign, and provide for European-style currency punctuation.

The format is:

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

5.3.1. Implementor Function Clause

A common implementor function is vertical positioning of the lines on a printed page. Often 12 positions (called channels) are available; the top of the page is associated with channel 1, the row at the next lower page position is associated with channel 2, etc.

If the implementor names for the channels are CHAN-1, CHAN-2, etc., and you code

```
CHAN-1 IS TOP-OF-PAGE
```

in the SPECIAL-NAMES paragraph, you can use the name you selected (the mnemonic-name TOP-OF-PAGE) to position a print line.

Thus, when you code

```
WRITE HEADING-LINE AFTER ADVANCING TOP-OF-PAGE
```

this prints this heading-line on the row of the page associated with channel 1.

Similarly, if you code

```
CHAN-12 IS BOTTOM-OF-PAGE
```

in the SPECIAL-NAMES paragraph and then

```
WRITE FOOTER AFTER ADVANCING BOTTOM-OF-PAGE
```

this prints FOOTER on the row associated with channel 12.

Generally, many other implementor functions are available. Another common one receives an operator message from the console. If the implementor-name for this function is SYSCONSOLE and you code

```
SYSCONSOLE IS OP-CONSOLE
```

you access the operator console through the mnemonic-name OP-CONSOLE. For example:

```
DISPLAY MY-MESSAGE UPON OP-CONSOLE
```

This makes the message appear on the operator console.

If the implementor-name function is a switch, depending on the implementor, you can provide a mnemonic-name and a condition-name or just a condition-name. You must associate each switch with at least one condition-name – a name that represents the switch on or off status. If the implementor-name for a particular switch is SYSSWCH-5 and the MNEMONIC-NAME IS MYERROR, you could code

```
SYSSWCH-5 IS MYERROR, ON STATUS IS ERR-SWITCH-ON,
          OFF STATUS IS ERR-SWITCH-OFF
```

If you want to use condition-names without mnemonic-names:

```
SYSSWCH-5 ON STATUS IS ERROR-SWITCH-ON, OFF STATUS IS ERROR-SWITCH-OFF.
```

Then, when SYSSWCH-5 is on, the conditional expression coding IF ERROR-SWITCH-ON is true, and IF ERROR-SWITCH-OFF is false.

5.3.2. Alphabet Name Clause

The alphabet-name you use in the SPECIAL-NAMES paragraph must also appear elsewhere in the program. If it appears in a program collating sequence (5.2) or in the collating sequence phrase of a sort or merge statement (9.3), alphabet-name represents a collating sequence. If it appears in a code-set clause in a file description (FD) entry, alphabet-name represents a character code set. In the SPECIAL-NAMES paragraph, you associate alphabet-name with a particular collating sequence or character code set. Your choice must be:

- STANDARD-1

  The collating sequence or character code set defined in the *American National Standard Code for Information Interchange (ASCII), X3.4—1968*

- NATIVE

  The implementor-defined collating sequence or character code set associated with the computer named in the OBJECT-COMPUTER paragraph

- Another collating sequence or character code set defined by the implementor

- A collating sequence you create in the SPECIAL-NAMES paragraph (this doesn't apply to character code sets).

If you select STANDARD-1 or an implementor-defined character code set or collating sequence, the implementor defines the correspondence between the characters in the set you selected and the characters in the native character set. Suppose the implementor offers a special collating sequence to which it has assigned the name ALTERN-SEQ. You can code PROGRAM COLLATING SEQUENCE IS MY-COLL-SEQ in the OBJECT-COMPUTER paragraph and MY-COLL-SEQ IS ALTERN-SEQ in the SPECIAL-NAMES paragraph to assign the special collating sequence to your program. You create your own collating sequence by rearranging the characters in the native sequence.

Assume the native collating sequence includes the characters in the COBOL character set beginning with the special characters in the order listed in 3.1, followed by the letters of the alphabet, followed by the digits 0–9. This means the space is the lowest character in the collating sequence and 9 is the highest. (A computer character set generally is much more lengthy than the COBOL character set but, for this example, we will assume they are identical.)

You can form a new collating sequence by rearranging the characters any way you choose. For instance, to move the comma from eighth to first in the collating sequence and the semicolon from ninth to second, code

```
PROGRAM COLLATING SEQUENCE IS MY-COLL-SEQ
```

in the OBJECT-COMPUTER paragraph and

```
MY-COLL-SEQ IS ',;'
```

in the SPECIAL-NAMES paragraph. The order in which you code the literals (",," and ";") specifies, in ascending sequence, the order they assume in the collating sequence. The unspecified characters from the native collating sequence (in this example, every character except the comma and semicolon) move to the end of the collating sequence, but retain the same relative order. Thus, the space becomes the third lowest character, the plus sign becomes the fourth lowest character, etc.

To move the less than symbol (<) to 11th in the collating sequence rather than 15th, code

```
MY-COLL-SEQ IS '△+-*/=$,;.<'
```

The quotation mark, which was the 11th character, becomes 12th, and the other characters follow in order.

You may prefer to make this change by using a numeric literal to indicate the new position of the less than symbol:

```
MY-COLL-SEQ IS 1 THROUGH 1Ø 15
```

Another way to make the same change is:

```
MY-COLL-SEQ IS '△' THROUGH '.' '<'
```

The "△" THROUGH "." refers to the 10 characters in the collating sequence beginning with the space, ending with the period, and including all the characters between them. The "<" means the less than symbol immediately follows the 10 characters.

The THROUGH option may specify characters in ascending or descending sequence. Thus

```
MY-COLL-SEQ IS '9' THROUGH '△'
```

reverses the entire collating sequence. The ALSO option assigns the same position in the collating sequence to more than one character.

The coding

```
MY-COLL-SEQ IS ''△'' ALSO ''+'', ALSO ''-''
```

assigns the space, the plus sign, and the minus sign to the lowest position in the collating sequence. Note that in this sequence, the figurative constant LOW-VALUE has the value space (the first character specified of those assigned to the lowest position).

If more than one character has the highest position in the collating sequence, HIGH-VALUE equals the last character assigned to the highest position.

You can use more than one option at a time. Thus, the coding

```
MY-COLL-SEQ IS ''0'' THROUGH ''9'' ''△+-*/'')'' THRU
     ''='' ''AEFZ'' 35 ''Q'' 50 ''Y''
```

changes the collating sequence to

```
0,1,2,3,4,5,6,7,8,9,△,+,-,*,/,),(,'',*,;,',$,=,
A,E,F,Z,>,<,B,C,D,G,H,Q,I,J,K,L,M,N,O,P,R,S,T,U,V,W,Y,X
```

To build your character sequence, the COBOL compiler reads and substitutes all literal values including character ranges (THRU clauses) from your alphabet-name statement.

In this type of statement, the first THROUGH clause sets up the sequence for the first 10 characters of your character sequence (0–9). The next literal inserts the first five special characters of the COBOL character code set (space through stroke). Following this, the second THRU clause inserts special characters beginning with the right parenthesis and moving in reverse order through the equal sign.

The next value you give is the literal "AEFZ". The compiler inserts this followed by the literals "Q" and "Y" in positions 35 and 50. After inserting the literal "AEFZ", the compiler uses the next values in the COBOL character code sequence (>and<) and completes the character sequence by inserting all remaining unused characters, skipping over the values already specified for positions 35 and 50.

5.3.3. CURRENCY SIGN IS Clause

The CURRENCY SIGN IS clause allows you to name a literal to replace the currency symbol ($) in the PICTURE clause. The literal must be a single character. It may not be

0–9

a space

A,B,C,D,L,P,R,S,V,X,Z

*,+,-,,,.,;,(,),",/,=

If you code

```
CURRENCY SIGN IS ''F''
```

the F in

```
05 TOTAL PIC F99,999.99
```

acts as the currency sign.

5.3.4. DECIMAL-POINT IS COMMA Clause

The DECIMAL-POINT IS COMMA clause provides for European-style currency punctuation. If it is present, the function of the comma and period are exchanged in PICTURE character-strings and in numeric literals. For example, if you write 2,145.67 when the DECIMAL-POINT IS COMMA clause is not present, write 2.145.67 when it is present.

5.4. COBOL FILE ORGANIZATION AND ACCESS METHODS

When you write a program that creates a data file, you have to decide how to organize the records in the file. If it is on a tape or cards, the decision is easy – the file must be sequential. If it is on a mass storage device such as disk, your choice depends on how the file will be used. Sequential organization is best if most or all of the file's records are needed each time the file is processed, and if it is convenient to access the records in the order they were written. If only part of the file is needed each time it is processed, or if it is preferable to access the records in a random sequence, you must use relative or indexed organization.

The order of the records in a sequential file is established when the file is written (the first record written as output is the first record read when the file is input). The records in relative and indexed files, by comparison, can be written and retrieved in any order. Relative file records are assigned a relative ordinal position in the file indicated by the value of a data item you designate as the relative key. The records in indexed files also are located through use of a key field (it is called the record key and it is associated with an index that points to the record).

Records in a sequential file must be accessed in the order they were written. You may access the records in relative or indexed files sequentially (according to the ascending order of the relative key or record key values, or randomly), selecting a specific record by manipulating the value of the key field. Or you may use a third access method (the dynamic mode) that allows you to alternate between sequential and random processing.

For example, suppose you have a 100-record relative file, and you want to access records 50–75 and 95–100. In dynamic mode, you set the relative key to 50 to randomly access the 50th record, switch to sequential access (by using the NEXT phrase with the READ verb) to retrieve records 51–75, set the relative key to 95 to access the 95th record, and return to sequential processing to access records 96–100.

You can see why the dynamic access mode can be preferable to sequential or random mode. If you processed this file sequentially, you would have to access every record, starting with the first, even though you are not interested in records 1–49 and 76–94. In random mode, you would not have to access unwanted records, but you would have to reset the relative key before each access.

5.5. FILE-CONTROL PARAGRAPH

In the FILE-CONTROL paragraph, you name and describe each input-outpout file. The format for a file-control-entry varies slightly depending on the file's organization. Format 1 is for sequential files, Format 2 is for relative files, and Format 3 is for indexed files.

Format 1:

```
SELECT [OPTIONAL] file-name
  ASSIGN TO implementor-name-1 [,implementor-name-2] ...
  [; RESERVE integer-1 [AREA ]]
                       [AREAS]
  [; ORGANIZATION IS SEQUENTIAL]
  [; ACCESS MODE IS SEQUENTIAL]
  [; FILE STATUS IS data-name-1].
```

Format 2:

```
SELECT file-name
  ASSIGN TO implementor-name-1 [,implementor-name-2] ...
  [; RESERVE integer-1 [AREA ]]
                       [AREAS]
  ; ORGANIZATION IS RELATIVE
  [; ACCESS MODE IS { SEQUENTIAL [,RELATIVE KEY IS data-name-1]  }]
                    {{RANDOM  } , RELATIVE KEY IS data-name-1    }
                    {{DYNAMIC }                                  }
  [; FILE STATUS IS data-name-2].
```

Format 3:

```
SELECT file-name
  ASSIGN TO implementor-name-1 [,implementor-name-2] ...
  [; RESERVE integer-1 [AREA ]]
                       [AREAS]
  ; ORGANIZATION IS INDEXED
  [; ACCESS MODE IS {SEQUENTIAL}]
                    {RANDOM    }
                    {DYNAMIC   }
  ; RECORD KEY IS data-name-1
  [; ALTERNATE RECORD KEY IS data-name-2 [WITH DUPLICATES]] ...
  [; FILE STATUS IS data-name-3].
```

5.5.1. Sequential Files

For sequential files, you often name the file and associate it with a device:

```
SELECT TAPE-FILE ASSIGN TO TAPE.
```

You could include

```
ORGANIZATION IS SEQUENTIAL
```

and

```
ACCESS MODE IS SEQUENTIAL
```

as documentation, but it isn't necessary. When you describe a relative or indexed file, however, you must use the ORGANIZATION clause and, if you want to use random or dynamic access, the ACCESS MODE clause must be used.

5.5.2. Relative Files

For relative files, a RELATIVE KEY clause generally is required – you may omit it if the access mode is sequential and you do not plan to use the START verb (7.4.7).

Unlike the record key, the relative key field cannot be part of the file's record description; you must define it elsewhere in the data division (probably in the working-storage section) as an unsigned integer. The value of the relative key identifies the relative ordinal position of a record in the file (if the file has 100 records, the permissible relative key values are 1 through 100). If the file control paragraph includes

```
SELECT INVENTORY-FILE ASSIGN TO DISK
    ORGANIZATION IS RELATIVE
    ACCESS MODE IS RANDOM
    RELATIVE KEY IS REL-KEY
```

The REL-KEY is defined in the working-storage section as

```
77 REL-KEY PIC 9(3).
```

You retrieve the 100th record in INVENTORY-FILE by coding

```
MOVE 100 TO REL-KEY
READ INVENTORY-FILE
```

If the access mode is sequential and you do not use the START verb to select a starting record for processing, the record with relative position number 1 is the first accessed.

5.5.3. Indexed Files

For indexed files, a RECORD KEY clause completes the entry:

```
SELECT INVENTORY-FILE ASSIGN TO DISK
     ORGANIZATION IS INDEXED
     ACCESS MODE IS RANDOM
     RECORD KEY IS PART-NUMBER.
```

The RECORD KEY (in this example, PART-NUMBER) must be a field in the record description for the file (it may be qualified). So, if the FD and its associated record description are:

```
FD INVENTORY-FILE
   LABEL RECORDS ARE STANDARD
   DATA RECORD IS RECORD-DESC.
01 RECORD-DESC.
   02 PART-NUMBER PIC X(6).
   02 CLASS PIC X(4).
   02 NAME  PIC X(25).
   02 DESC  PIC X(25).
```

Then, the statements

```
MOVE ''123456'' TO PART-NUMBER
READ INVENTORY-FILE
```

retrieve the record from INVENTORY-FILE that has the value 123456 in the PART-NUMBER field. If the access mode was sequential, the coding

```
READ INVENTORY-FILE
```

would start the processing with the record whose PART-NUMBER field has the lowest value.

You may specify alternate record keys for an indexed file. If your file control paragraph includes

```
SELECT DIRECTORY-FILE ASSIGN TO DISK
    ORGANIZATION IS INDEXED
    RECORD KEY IS NAME
    ALTERNATE RECORD KEY IS PHONE
```

Then NAME and PHONE, which must be alphanumeric fields defined in the record description of DIRECTORY-FILE, are each associated with an index that points to the record. Thus, if the file is a list of employees and their phone extensions, you could access a record using either an employee's name or phone number as the key of reference. You identify that key in the START or READ statement.

For example:

```
MOVE ''SMITH'' TO NAME
```

and

```
READ DIRECTORY-FILE KEY IS NAME
```

retrieves the record with the name SMITH and enables you to find the corresponding phone extension, while

```
MOVE ''2315'' TO PHONE
```

and

```
READ DIRECTORY-FILE KEY IS NAME
```

retrieves the record with phone extension 2315 and enables you to find the corresponding employee name. If you add the phrase WITH DUPLICATES to the ALTERNATE RECORD KEY clause, as in

```
ALTERNATE RECORD KEY IS PHONE WITH DUPLICATES
```

the value of the alternate record key field may be duplicated among records in the file. So, in this example, WITH DUPLICATES means more than one employee may have the same phone extension. The records with duplicate key fields are accessed in the order they were written. If you do not use WITH DUPLICATES, the value of each record's alternate record key field must be unique. The value of the record key field always must be unique.

5.5.4. Additional Phrases and Clauses

There are other phrases and clauses you may want to use in a file-control-entry. If your program defines a sequential input file that isn't processed on each run – perhaps because it's a tape needed at the end of the month but not weekly – you can notify the operating system by inserting the word OPTIONAL before the file-name:

```
SELECT OPTIONAL TAPE-FILE ASSIGN TO TAPE.
```

If you do not code OPTIONAL before the file-name, the file must be present at run time.

The RESERVE clause is available with all file organizations. You use it if you want to change the number of input-output areas the implementor allocates for each file. So, if the implementor allocates two I-O areas for each file, you code

```
RESERVE 3 AREAS
```

to assign an extra I-O area to the file named in the corresponding SELECT statement. An extra I-O area requires more main storage but may speed up processing.

If you include the FILE STATUS clause in your file-control entry, you can find out whether a specific I-O operation for a file was successful. Immediately after the execution of an OPEN, CLOSE, READ, WRITE, REWRITE, DELETE, or START statement (and before any applicable USE procedure is executed), the operating system moves a code to the data-name you specify in the FILE STATUS clause. That data-name must be defined in the working-storage or linkage section as two alphanumeric characters. The leftmost digit of the code is known as status key 1; the rightmost digit is called status key 2.

Table 5–1 lists the possible status key values and explains what they mean. The value of status key 2 is dependent on the value of status key 1; it provides (if necessary) more information about the I-O operation. Often it is set to 0, indicating no further information is available.

All possible status key values are shown except that status key 1 may equal 9, indicating the I-O operation was unsuccessful as the result of a condition specified by the implementor. If status key 1 is 9, the value of status key 2 is defined by the implementor.

The table indicates that if status key 1 is 0 and status key 2 is 2, the I-O operation was successful, even though the record accessed has the same key field or another record in the file. The duplicate record key is valid either because a READ statement was executed and the key of reference equals the value of the same field in the next record (in this example the key would have to be an alternate record key), or a WRITE or REWRITE statement was executed and the record just written created a duplicate key value in an alternate record key field for which the WITH DUPLICATES phrase was specified.

Table 5—1. File Status Key Values

File	Status Key		Explanation
	1	2	
Sequential	0	0	Successful I-O
	1	0	READ not completed due to AT END condition or because file described as OPTIONAL is unavailable
	3	0	Unsuccessful I-O due to permanent error (data check, parity error, or transmission error)
	3	4	Unsuccessful I-O due to boundary violation
Relative	0	0	Successful I-O
	1	0	READ not completed due to AT END condition
	2	2	Unsuccessful I-O due to attempt to write a record that would create a duplicate relative key
	2	3	Unsuccessful I-O due to attempt to access a record using a relative key not associated with any record in the file
	2	4	Unsuccessful I-O due to attempt to write beyond the implementor-defined external boundaries of the file
	3	0	Unsucessful I-O due to permanent error (data check, parity error, or transmission error)
Indexed	0	0	Successful I-O
	0	2	Successful I-O (record accessed has duplicate key with another record in the file) (see text)
	1	0	READ not completed due to AT END condition
	2	1	Unsuccessful I-O because (in sequential mode) attempt was made to write a record with RECORD KEY not greater than that of previous record; or the value of RECORD KEY was changed between execution of a READ and corresponding REWRITE statement
	2	2	Unsuccessful I-O due to attempt to write or rewirte a record that would create a duplicate RECORD KEY
	2	3	Unsuccesful I-O because no record in the file has a RECORD KEY field that matches value specified
	2	4	Unsuccessful I-O due to attempt to write beyond the implementor-defined external boundaries of the file
	3	0	Unsuccessful I-O due to permanent error (data check, parity error, or transmission error)

5.6. I-O CONTROL PARAGRAPH

If some of your COBOL programs require longer execution times, you will want to avoid having to rerun them from the beginning if a hardware failure, operating system error, or some other problem halts processing before a run is completed. The I-O paragraph allows you to write checkpoint records (information about the status and environment of a program) at specified intervals during the run. Then, if something goes wrong, you can restart the program from the point where a checkpoint was taken. The I-O paragraph also provides a way to assign the same main storage area to several files, and a clause needed for processing tapes that contain more than one file.

The format for the I-O CONTROL paragraph is:

```
I-O-CONTROL.
 [ ; RERUN [ON { file-name-1       }]                                  ] ...
                { implementor-name }
   EVERY { { [END OF] { REEL } }                  OF file-name-2 }
         { {          { UNIT } }                               }
         { { integer-1 RECORDS }                               }
         { integer-2 CLOCK-UNITS                               }
         { condition-name                                      }
 [ ; SAME [RECORD] AREA FOR file-name-3 {,file-name-4} ...] ...
 [ ; MULTIPLE FILE TAPE CONTAINS file-name-5 [POSITION integer-3]  ] ...
   [,file-name-6 [POSITION integer-4]] ...                           ]
```

5.6.1. RERUN Clause

The RERUN clause specifies when and where checkpoint records are taken. The records provide the operating system with the information it needs to restart a program at a checkpoint. There are seven valid forms of the RERUN clause; the implementor must provide at least one of them. Following are examples of each form.

Example 1:

```
RERUN EVERY END OF REEL OF TAPE-FILE
```

or

```
RERUN EVERY END OF UNIT OF DISK-FILE
```

A checkpoint record is written on TAPE-FILE or DISK-FILE at the end of each reel of tape containing TAPE-FILE or at the end of each unit containing DISK-FILE (the definition of UNIT is determined by the implementor). TAPE-FILE and DISK-FILE must be output files with sequential organization.

Example 2:

```
RERUN ON OUTPUT-FILE EVERY END OF REEL OF TAPE-FILE
```

or

```
RERUN ON OUTPUT-FILE EVERY END OF UNIT OF DISK-FILE
```

This coding writes a checkpoint record on OUTPUT-FILE (not on TAPE-FILE or DISK-FILE) at the end of each reel of tape containing TAPE-FILE or at the end of each unit containing DISK-FILE. TAPE-FILE and DISK-FILE still must be sequential, but they may be either input or output files. OUTPUT-FILE (or whatever file to which you write checkpoint records) must be a sequential file.

Example 3:

```
RERUN ON IMP-UNIT EVERY END OF REEL OF TAPE-FILE
```

or

```
RERUN ON IMP-UNIT EVERY END OF UNIT OF DISK-FILE
```

These examples work the same as Example 2, except that the checkpoint record is written on a device specified by the implementor (IMP-UNIT) instead of on a file you define (OUTPUT-FILE).

Example 4:

```
RERUN ON IMP-UNIT EVERY 1000 RECORDS OF INV-FILE
```

This coding writes a checkpoint record on the implementor-specified device (IMP-UNIT) after each 1000 records of INV-FILE are processed. INV-FILE may be either an input or output file with any organization or access. Whenever you select the RECORDS or CLOCK-UNITS option, you must use implementor-name.

Example 5:

```
RERUN ON IMP-UNIT EVERY 60 CLOCK-UNITS
```

A checkpoint record is written on IMP-UNIT whenever an interval of time represented by 60 CLOCK-UNITS (defined by implementor) has elapsed. You may use only one RERUN clause that specifies the CLOCK-UNITS option.

Example 6:

```
RERUN ON IMP-UNIT EVERY CHECKPT-SWITCH-ON
```

A checkpoint record is written on IMP-UNIT when a switch defined in the SPECIAL-NAMES paragraph such as

```
CHECKPT-SWITCH ON STATUS IS CHECKPT-SWITCH-ON
               OFF STATUS IS CHECKPT-SWITCH-OFF
```

is on. The implementor defines when the switch's status is interrogated.

Example 7:

```
RERUN ON CHECKPT-FILE EVERY CHECKPT-SWITCH-ON
```

This is the same as Example 6, except that the checkpoint record is written on CHECKPT-FILE, which must be a sequential output file.

You may specify more than one RERUN clause for a given file-name-2; however, if you use multiple clauses with the RECORDS option or multiple clauses with the END OF REEL or END OF UNIT option, no two of them may specify the same file-name-2.

5.6.2. SAME AREA Clause

If you want to reduce the amount of main storage needed by your program, you can use the SAME AREA clause to assign the same main storage area to two or more files, regardless of their organization or access. You can include more than one SAME AREA clause in your program, but a file-name may appear in only one clause.

The coding

```
SAME AREA FOR TAPE-FILE DISK-FILE CARD-FILE
```

indicates that TAPE-FILE, DISK-FILE, and CARD-FILE share a main storage area. Only one of the files may be open at the same time. The SAME [RECORD] AREA clause specifies that two or more files share the same main storage area for processing the current logical record. All of the files may be open at the same time; thus, a record in the shared area is considered to be a record in each opened output file and in the most recently read input file named in the clause. For example, if your program has two files defined as

```
FD  INPUT-FILE
    LABEL RECORDS ARE OMITTED.
Ø1  IN-RECD PIC X(8Ø).
FD  OUTPUT-FILE
    LABEL RECORDS ARE OMITTED.
Ø1  OUT-RECD PIC X(8Ø)
```

and you code

```
SAME RECORD AREA FOR INPUT-FILE OUTPUT-FILE
```

then

```
READ INPUT-FILE AT END GO TO EOJ.
WRITE OUTPUT-FILE.
```

is equivalent to

```
READ INPUT-FILE AT END GO TO EOJ.
MOVE IN-RECD TO OUT-RECD.
WRITE OUTPUT-FILE.
```

The MOVE statement is not needed because IN-RECD and OUT-RECD refer to the same record area in main storage. Do not use the the same file-name in more than one SAME RECORD AREA clause. You can include file-names that appear in a SAME AREA clause; however, if one file named in a SAME AREA clause is included in the SAME RECORD AREA clause, all the files named in that SAME AREA clause must appear in the SAME RECORD AREA clause. In this situation, it is still a rule that only one file named in a SAME AREA clause may be open, but all files named in the SAME RECORD AREA clause that are not in the SAME AREA clause may be open.

5.6.3. MULTIPLE FILE Clause

If you process files on a tape that contains more than one file, you must use the MULTIPLE FILE clause to specify the relative positions on the tape of the files you need. If a tape contains four files called FILE-1, FILE-2, FILE-3, and FILE-4, and your program processes FILE-1 and FILE-3, you must code

```
MULTIPLE FILE TAPE CONTAINS FILE-1 POSITION 1
                           FILE-3 POSITION 3.
```

You may omit the POSITION clause if the files you need can be listed in consecutive order beginning with the first on the tape. Thus, in the following example, if your program processes FILE-1, FILE-2, and FILE-3, you can code

```
MULTIPLE FILE TAPE CONTAINS FILE-1,FILE-2 FILE-3.
```

However, if your program processes the last three files on the tape, you must code

```
MULTIPLE FILE TAPE CONTAINS FILE-2 POSITION 2
                           FILE-3 POSITION 3
                           FILE-4 POSITION 4
```

Only one file on the same tape reel may be open at the same time.

5.7. CODING EXAMPLE

Following is a sample environment division that includes many of the clauses and phrases discussed in this section (partial data division is included to show how it relates to some of the environment division entries):

```
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.     COMPUT-3.
OBJECT-COMPUTER.     COMPUT-3.
     MEMORY SIZE 50 MODULES
     PROGRAM COLLATING SEQUENCE IS PROG-COLL-SEQ.
SPECIAL-NAMES.
     CHAN-1 IS TO-NEXT-PAGE.
     CHAN-10 IS SUBTOT-LINE.
     CHAN-12 IS BOTTOM-PAGE.
     SYSCONSOLE IS OP-RESPONSE.
     SYSRERUN ON STATUS IS TAKE-CKPOINT OFF STATUS IS NO-CKPOINT.
     PROG-COLL-SEQ IS STANDARD-1.
     CURRENCY SIGN IS W.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
     SELECT OPTIONAL CARD-DECK ASSIGN TO READER.
     SELECT PRINTOUT ASSIGN TO PRINTER.
     SELECT MASTER-FILE ASSIGN TO DISK.
          ORGANIZATION IS INDEXED.
          RECORD KEY IS REC-KEY.
          ACCESS MODE IS DYNAMIC.
          FILE STATUS IS CHECK-MASTER.
I-O-CONTROL.
     RERUN ON COMPUT-FILE EVERY TAKE-CHPOINT.
DATA DIVISION.
FILE SECTION.
FD  CARD-DECK
     .
     .
     .
FD  PRINTOUT
     .
     .
     .
```

(continued)

```
FD  MASTER-FILE
    .
    .
    .
WORKING-STORAGE SECTION.
77 CHECK-MASTER      PIC X(2).
    .
    .
    .
```

The specific effects of many of the clauses and phrases are implementor-defined. COMPUT-3, CHAN-1, CHAN-10, CHAN-12, SYSCONSOLE, SYSRERUN, READER, DISK, and COMPUT-FILE all are implementor-names; you will have to check the operating system documentation to see what names you must use for similar entries.

The terms TO-NEXT-PAGE, SUBTOT-LINE, PAGE-BOTTOM, and OP-RESPONSE are mnemonic-names. Mnemonic-names are defined only in the SPECIAL-NAMES paragraph; their meanings are implementor-defined.

The program that contains this environment division is compiled and executed on a computer called COMPUT-3 (that computer has a main storage capacity of 50 modules). The ASCII collating sequence replaces the COMPUT-3 native collating sequence in this program.

Mnemonic-names are provided to allow the programmer to receive a message from the console and to position printed output at three locations. This program takes a checkpoint record whenever the switch SYSRERUN is tested and found to be on. In the program PICTURE clauses, the character W represents the currency sign.

Three I-O files are described in the FILE-CONTROL paragraph; two of the files are always processed; the third (CARD-DECK) may or may not be processed in a given run. CARD-DECK and PRINTOUT are sequential files. MASTER-FILE is an indexed file that is accessed dynamically. After the execution of an I-O verb that references MASTER-FILE, a 2-digit code is moved to CHECK-MASTER. That code indicates the status of the I-O operation just completed for the file.

6. Data Division

The data division describes the data your program processes. Data elements originate from external files, are developed in the program, or appear in the program as constants. You describe the data in sections, entered in this order:

```
DATA DIVISION.
[FILE SECTION.                                                              ]
[     [file-description-entry  [record-description-entry] ...              ] ...]
[     [sort-merge-file-description-entry {record-description-entry} ...    ]   ]
[WORKING-STORAGE SECTION.                                                   ]
[     [77-level-description-entry] ...                                      ]
[     [record-description-entry  ]                                          ]
[LINKAGE SECTION.                                                           ]
[     [77-level-description-entry] ...                                      ]
[     [record-description-entry  ]                                          ]
[COMMUNICATION SECTION.                                                     ]
[     [communication-description-entry [record-description-entry] ...] ...  ]
```

The file section is where you describe your data files; these include file-description-entries, record-description-entries, and sort-merge-file-descriptions. The latter entries are explained in Section 9.

The working-storage section consists of 77-level-description-entries (single data items not related to other data items) and record-description-entries (records like those in the file section, but not associated with a file).

The linkage and communication entries are examined in Sections 10 and 11, respectively.

6.1. FILE DESCRIPTION ENTRY

File description (FD) entries consist of a level indicator FD, a file-name, and a series of clauses that specify the size of the file's logical and physical records, the type of file labels used, the names of the records that comprise the file, and other related data.

The FD format is:

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

In the environment division, SELECT statements are used to name the files and associate them with devices. Each file-name that is the subject of a SELECT statement must also be the subject of an FD. If you code

```
SELECT MASTER ASSIGN TO TAPE
```

in the environment division, you must code an entry that begins

```
FD MASTER
```

in the data division. The letters FD belong in area A; the rest of the entry must be in area B.

6.1.1. LABEL Clause

The LABEL clause is required, and for good reason. It allows the operating system to check labels to make certain you are processing the correct file. It also specifies the label you want written on your output files. Coding

```
LABEL RECORDS ARE STANDARD
```

can complete an FD entry – nothing else is required. The STANDARD option means that you have labeled your files and the labels conform to the implementor's standard format. Unit record file (card reader, printer, etc.) are never labeled; most programmers label the tape, disk, and diskette files. Depending on your operating system, the actual label is specified in job control statements or the VALUE OF clause in the FD entry.

The entry

```
FD   MASTER
     LABEL RECORDS ARE STANDARD
     VALUE OF FILE-ID IS ''PAYROLL-FILE''
     VALUE OF CALC-FREQ IS PAYROLL-CALC
```

means the file named MASTER is labeled and the implementor assigned the names FILE-ID and CALC-FREQ to particular fields in its standard label records. In this example, the fields contain a file label and a file payroll computation time.

When MASTER is opened as an input file, the operating system compares the label in FILE-ID to PAYROLL-FILE. If they match, processing proceeds. The operating system also compares the report frequency (PAYROLL-CALC in working storage) with the value in CALC-FREQ. If these values match, processing again proceeds. Standard error procedures are specified by the implementor.

When MASTER is opened as an output file, the operating system writes a label record for the file. It writes PAYROLL-FILE into the FILE-ID field and the report frequency value PAYROLL-CALC (i.e., weekly or monthly) into the CALC-FREQ field.

You can qualify the data-names (like PAYROLL-CALC) that you use in VALUE OF clauses, but you cannot subscript or index them and you cannot describe them with the USAGE IS INDEX clause.

6.1.2. BLOCK CONTAINS Clause

In most FD entries, you need a BLOCK CONTAINS clause; it specifies the size of a physical record. The term *record* refers to *logical records* (the groups of related data you define as hierarchical structures in the file or working-storage sections, and treat as a unit). A *physical record* (also known as a *block*) is a unit of data (generally one or more logical records) as it appears on an I-O device. Operating systems work with physical records; programmers work with logical records.

A card file, because a card is a fixed length, is always unblocked – its logical and physical records are the same size. Thus, you do not need a BLOCK CONTAINS clause, and the coding

```
FD   A-CARD-FILE
     LABEL RECORDS ARE OMITTED.
```

is a complete entry. When tape or mass storage files are unblocked, you again omit the BLOCK CONTAINS clause or, depending on the operating system, you code

```
BLOCK CONTAINS 1 RECORD
```

Generally, computers process tape and disk I-O files more efficiently when the records are blocked; that is, when each physical record contains more than one logical record. Then, fewer physical I-O operations – the movement of blocks of records (by the operating system) from I-O devices to main storage – are needed. If you code

```
BLOCK CONTAINS 5 RECORDS
```

for an input file, the operating system moves five records at a time into main storage. The file is said to have a blocking factor of five. Program logic isn't affected; a READ statement still retrieves only one record – just as if the file was unblocked. But the number of physical I-O operations is reduced by a factor of five (five READ statements are executed before the operating system moves the next block of five records into main storage). The blocking factor is limited by the amount of main storage available to execute the program; the greater the blocking factor, the greater the amount of main storage needed.

You may specify the number of characters, rather than records, in a block. In fact, you *must* use characters if

- in a mass storage file, more than one physical record is needed to contain a logical record;
- the physical record contains padding – areas not part of a logical record (this depends on the implementor); or
- the grouping of logical records implies an inaccurate physical record size.

For example, if two 100-character records always follow one 50-character record in a file, use

```
BLOCK CONTAINS 250 CHARACTERS
```

rather than

```
BLOCK CONTAINS 3 RECORDS
```

You can specify a range for the block size by making integer-1 a minimum and integer-2 a maximum size. Thus,

```
BLOCK CONTAINS 500 TO 4000 CHARACTERS
```

The implementor determines how the specific size of each block is calculated.

6.1.3. RECORD CONTAINS Clause

You also can specify the size of a file's logical records. The RECORD CONTAINS clause is never required, because the size of each data record is determined by the record description entry. But you may want to use the clause as documentation. If all records have the same number of characters, specify that number in the clause:

```
RECORD CONTAINS 80 CHARACTERS
```

If the record length is variable, specify the minimum and maximum size. For instance, if the record description is

```
01   RECD-DESC.
     02  A     PIC X(10).
     02  B     OCCURS 1 TO 5 TIMES
                  DEPENDING ON THE-COUNT    PIC X(25).
```

then the coding is

```
RECORD CONTAINS 35 TO 135 CHARACTERS
```

6.1.4. DATA RECORDS Clause

The DATA RECORDS clause also is for documentation only. It names the data records associated with the file. The names are the same as those that follow level-number 01 in the file's record description entries. Thus, if the record description entries are

```
01   NAME-RECD.
     02  ...
      .
      .
      .
01   ADD-RECD.
     02  ...
      .
      .
      .
```

The coding is

```
DATA RECORDS ARE NAME-RECD ADD-RECD
```

6.1.5. CODE-SET Clause

The CODE-SET clause is similar to the PROGRAM COLLATING SEQUENCE clause (5.2) except it pertains to a character code set rather than a collating sequence. When you use the CODE-SET clause, data in the file is represented by the character code set specified by alphabet-name, rather than the native character code set. You use the SPECIAL-NAMES paragraph to associate alphabet-name with a specific character code set. For instance, if you code

```
CODE-SET IS OUR-CODE-SET
```

in the FD and

```
OUR-CODE-SET IS STANDARD-1
```

in the SPECIAL-NAMES paragraphs in the environment division, the ASCII character code set (STANDARD-1 represents ASCII) is in effect for the file.

You may not specify the CODE-SET clause for mass storage files. You must describe all data in a file that has the CODE-SET clause as USAGE IS DISPLAY (the usage always is displayed unless you specify otherwise) and you must describe any signed numeric data with the SIGN IS SEPARATE clause.

6.1.6. LINAGE Clause

When you write a program that produces printed (or display) output, you may code logic to count print lines and determine when to advance to a new page, or you may code a LINAGE clause to do that work for you. The LINAGE clause specifies the number of lines on a logical page, the size of page margins, and the location of a page footing area (Figure 6–1). If you code

```
FD   PRINTOUT
     LABEL RECORDS ARE OMITTED
     LINAGE IS 54 LINES
        WITH FOOTING AT 51
        LINES AT TOP 3
        LINES AT BOTTOM.3
01   PRINTLINE   PIC  X(132).
```

a logical page for PRINTOUT consists of 60 lines – a page body of 54 lines plus 3-line margins at the top and bottom of the page. In addition, lines 51-54 of the page body are considered a footing area.

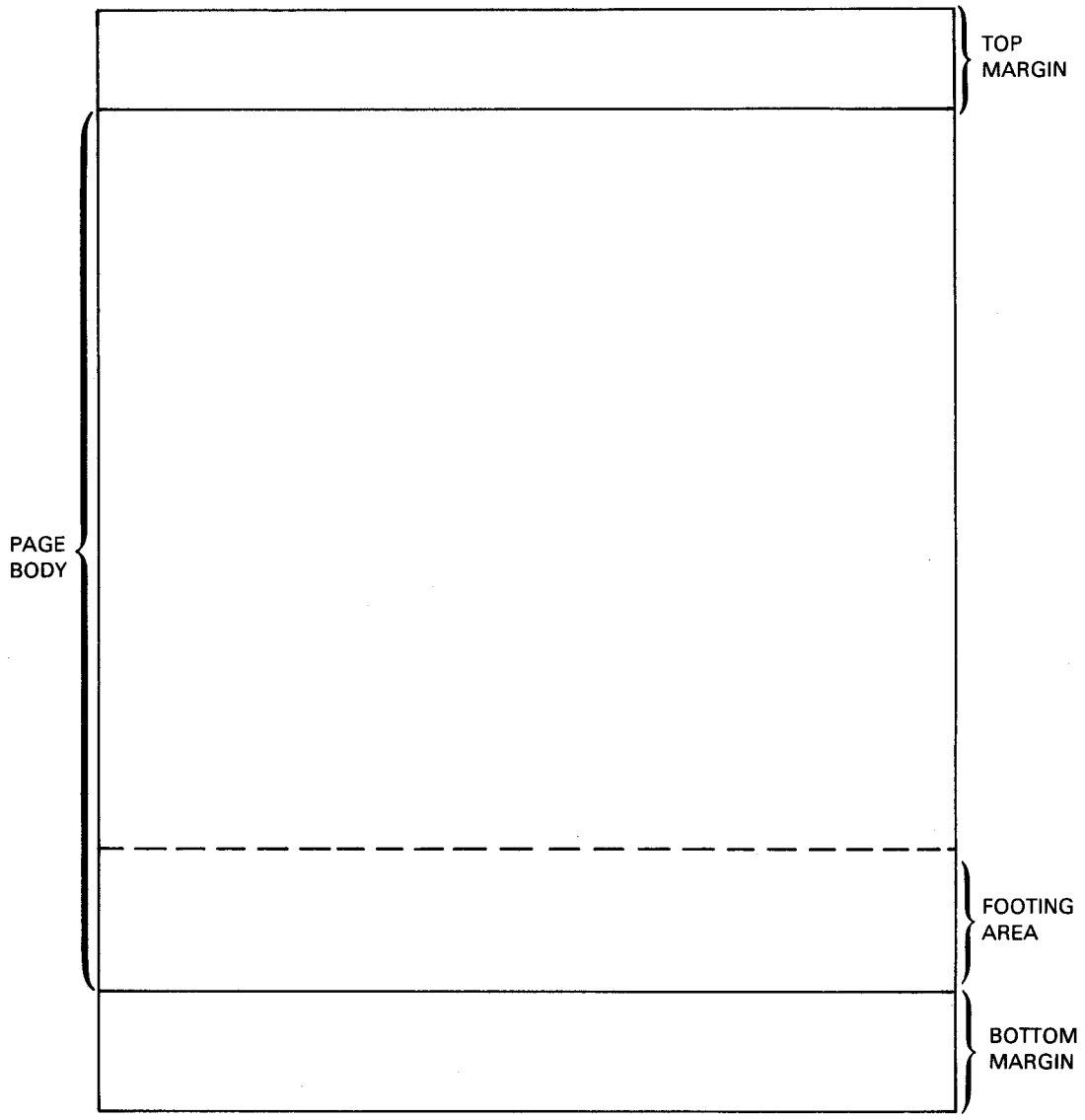


Figure 6—1. Page Defined by LINAGE Clause

For every file that has a LINAGE clause, the operating system generates a data item represented by the reserved word LINAGE-COUNTER. Its value indicates the vertical position of a print line; i.e., it counts lines for you. When the file is opened, and with each page advance, LINAGE-COUNTER is set to 1. It is incremented during the execution of WRITE statements for the file, until the end of the page is reached.

If you assign a footing area, and you use the END-OF-PAGE phrase with the WRITE statement, that end of the page occurs when LINAGE-COUNTER equals or exceeds the value in the WITH FOOTING phrase. So, if the execution of

```
WRITE PRINTLINE AFTER ADVANCING 2 LINES
      AT END-OF-PAGE PERFORM HEADING-ROUTINE
```

causes LINAGE-COUNTER for PRINTOUT to equal or exceed 51,

```
PERFORM HEADING-ROUTINE
```

is executed.

Generally, the imperative statement (like PERFORM HEADING-ROUTINE) includes logic that causes a page advance, thus resetting LINAGE-COUNTER to 1. If this logic does not exist, or if there is no footing area, the end of the page (and an automatic page advance) occurs when there is a page overflow condition; i.e., the value in LINAGE-COUNTER is greater than the length of the page body (in this case, 54). When automatic page overflow occurs, the line to be printed is presented at the top of the next page.

Figures 6-2 and 6-3 show portions of similar sample programs. In Figure 6-2, programmer logic sets up page advances; in Figure 6-3, the LINAGE clause replaces that logic.

You can reference but not modify LINAGE-COUNTER in the procedure division. If there is more than one LINAGE-COUNTER in the program (because more than one file is described with a LINAGE clause), you must qualify the references by file-name:

```
LINAGE-COUNTER OF PRINTOUT
```

You do not need to specify a footing area or top and bottom margin areas. The entry

```
FD   PRINTOUT
     LABEL RECORDS ARE OMITTED
     LINAGE IS 60 LINES
```

means a logical page for PRINTOUT has 60 lines and no footing area or top or bottom margins. There is an automatic page advance – and LINAGE-COUNTER OF PRINTOUT is set to 1 – whenever the value of LINAGE-COUNTER equals or exceeds 60.

```
DATA DIVISION.
 .
 .
 .
FD  CARDFILE
    LABEL RECORDS ARE OMITTED.
01  CARDRECD      PIC X(80).
FD  PRINTFILE
    LABEL RECORDS ARE OMITTED.
01  PRINTLINE    PIC X(132).
 .
 .
 .
77  LINECOUNT      PIC 9(2).
 .
 .
 .
PROCEDURE DIVISION.
BEGIN.
     OPEN INPUT CARDFILE OUTPUT PRINTFILE.
     PERFORM WRITEHEADINGS.
MAIN.
     IF LINECOUNT GREATER THAN 59 PERFORM WRITEHEADINGS.
     READ CARDFILE AT END GO TO EOJ.
      .
      .
      .
     WRITE PRINTLINE FROM DETAIL-LINE AFTER ADVANCING 2 LINES.
     ADD 2 TO LINECOUNT.
     GO TO MAIN.
      .
      .
      .
WRITEHEADINGS.
     MOVE ZEROS TO LINECOUNT.
     WRITE PRINTLINE FROM HEAD1 AFTER ADVANCING PAGE.
     WRITE PRINTLINE FROM HEAD2 AFTER ADVANCING 1 LINE.
     ADD 2 TO LINECOUNT.
      .
      .
      .
EOJ.
     WRITE PRINTLINE FROM LAST-LINE AFTER ADVANCING 3 LINES.
      .
      .
      .
```

Figure 6—2. Page Advance Controlled by Counting Lines

```
DATA DIVISION.
 .
 .
 .
FD  CARDFILE
    LABEL RECORDS ARE OMITTED.
01  CARDRECD      PIC X(80).
FD  PRINTFILE
    LABEL RECORDS ARE OMITTED
    LINAGE IS 60 LINES
          WITH FOOTING AT 56
          LINES AT TOP 5
          LINES AT BOTTOM 5.
01  PRINTLINE     PIC X(132).
 .
 .
 .
PROCEDURE DIVISION.
BEGIN.
     OPEN INPUT CARDFILE OUTPUT PRINTFILE.
     PERFORM WRITEHEADINGS.
MAIN.
     READ CARDFILE AT END GO TO EOJ.
      .
      .
      .
     WRITE PRINTLINE FROM DETAIL-LINE AFTER ADVANCING 2 LINES
          AT END-OF-PAGE PERFORM WRITEHEADINGS.
     GO TO MAIN.
      .
      .
      .
WRITEHEADINGS.
     WRITE PRINTLINE FROM HEAD1 AFTER ADVANCING PAGE.
     WRITE PRINTLINE FROM HEAD2 AFTER ADVANCING 1 LINE.
      .
      .
      .
EOJ.
     WRITE PRINTLINE FROM LAST-LINE AFTER ADVANCING 3 LINES.
      .
      .
      .
```

Figure 6—3. Page Advance Controlled by LINAGE Clause

6.2. RECORD DESCRIPTION ENTRY

Record description entries describe records. They are composed of data description entries that characterize the particular data items (or fields) in the record. At least one record description entry must follow each file description entry. In the working-storage section, record description entries describe groups of related data items.

The file description entry is as follows:

```
FD  CARDIN
    LABEL RECORDS ARE OMITTED
    DATA RECORDS IS CARD-RECD.
```

It is immediately followed by a record description entry beginning with

```
01  CARD-RECD
```

If the record is 80 characters long

```
01  CARD-RECD      PIC X(80).
```

can be a complete record description entry. Generally, you are more specific in describing the 80 characters. If the record contains a name, address, and phone number, the code is:

```
01  CARD-RECD.
    02   NAME      PIC X(30).
    02   ADDRESS.
         03   STREET      PIC X(25).
         03   CITY        PIC X(15).
         03   STATE       PIC X(2).
    02   PHONE.
         03   EXCHANGE    PIC X(3).
         03   EXTENSION   PIC X(4).
    02   FILLER       PIC X(1).
```

You can now reference the various fields in the record. If the file contains more than one type of record, a record description for each type is coded as follows:

```
FD  CARDIN
    LABEL RECORDS ARE OMITTED
    DATA RECORDS ARE CARD-RECD CARD-RECD-2.
01  CARD-RECD.
    02  TYPE    PIC X(1).
    02  NAME    PIC X(39).
    02  ST-ADDR-1  PIC X(20).
    02  ST-ADDR-2  PIC X(20).
01  CARD-RECD-2.
    02  FILLER   PIC X(1).
    02  ST-ADDR-3   PIC X(20).
    02  FILLER    PIC X(59).
```

The CARD-RECD and CARD-RECD-2 entries share the same area of storage; thus, if you read CARDIN and in fact read the CARD-RECD record but reference ST-ADDR-3, you get the first 20 characters of NAME. As mentioned, record description entries are composed of data description entries that characterize the particular fields in the record. The format for a data description entry is:

```
level-number  {data-name-1}
              {FILLER     }
  [; REDEFINES data-name-2]
  [; {PICTURE} IS character-string]
     {PIC    }
  [; [USAGE IS] {COMPUTATIONAL}]
                {COMP        }
                {DISPLAY     }
  [; [SIGN IS] {LEADING } [SEPARATE CHARACTER]]
               {TRAILING}
  [; {SYNCHRONIZED} [LEFT ]]
     {SYNC        }  [RIGHT]
  [; {JUSTIFIED} RIGHT]
     {JUST     }
  [; BLANK WHEN ZERO]
  [; VALUE IS literal] .
66   data-name-1; RENAMES data-name-2 [{THROUGH} data-name-3 ] .
                                      {THRU   }
88   condition-name; {VALUE IS } literal-1 [{THROUGH} literal-2 ]
                     {VALUES ARE}          {THRU   }
[, literal-3 [{THROUGH} literal-4 ] ] ... .
             {THRU   }
```

6.2.1. REDEFINES Clause

Level 01 entries for the same FD implicitly redefine each other. Except for 01-, 66-, and 88-level entries in the file section for which REDEFINES clauses are not permitted, you use the REDEFINES clause to assign more than one data description to the same storage area. If you code

```
05  A   PIC X(5).
05  B  REDEFINES  A  PIC 9(5).
```

A and B reference the same five characters of storage. The item that has the REDEFINES clause (the B entry) must immediately follow the object of the REDEFINES clause (the A entry). Both items must have the same level-number and no entry with a numerically lower level-number may come between them. Thus, the following is not allowed:

```
05  A   PIC X(5).
02  C.
05  B  REDEFINES  A  PIC X(5).
```

But the following is legal:

```
05  A    PIC X(5).
05  B  REDEFINES  A  PIC 9(5).
```

The REDEFINES clause often is used when describing fields that may be either numeric or alphanumeric, depending on the record type. A storage area that may be an alphanumberic invoice number or a numeric order number can be described as

```
02  INVOICE-NUMBER     PIC X(6).
02  ORD-NO  REDEFINES INVOICE-NUMBER  PIC 9(6).
```

You use INVOICE-NUMBER to reference the area when it is alphanumeric and ORD-NO to reference it when it is numeric. Both data-names in the REDEFINES clause (except level 01 entries in working storage section) must reference the same number of characters. Thus, if the invoice number is six characters but the order number is only four, you code:

```
02  INVOICE-NUMBER    PIC X(6).
02  ORD-NO    REDEFINES    INVOICE-NUMBER.
    03  ORDER-NUMBER     PIC 9(4).
    03  FILLER     PIC X(2).
```

The FILLER entry is used to account for the two character positions not used when the field is numeric.

You may redefine a storage area as many times as you like, but the object of the REDEFINES clause must be the same in each definition. if A, B, C, and D reference the same area, the code is:

```
02  A    PIC X(5).
02  B  REDEFINES A    PIC X(5).
02  C  REDEFINES A    PIC X(5).
02  D  REDEFINES A    PIC X(5).
```

not

```
02  A    PIC X(5).
02  B  REDEFINES A    PIC X(5).
02  C  REDEFINES B    PIC X(5).
02  D  REDEFINES C    PIC X(5).
```

You may not use the VALUE clause in the description of a data item that has a REDEFINES clause, but you may use it in the description of the object of the clause. Thus, the following is valid:

```
05  ITEM    PIC X(2) VALUE ''A2''.
05  RED-ITEM  REDEFINES  ITEM  PIC 9(2).
```

The following is not valid:

```
05  ITEM    PIC X(2).
05  RED-ITEM  REDEFINES  ITEM  PIC 9(2)   VALUE ''15''.
```

The object of the REDEFINES clause cannot be described with an OCCURS or REDEFINES clause, so

```
05  ITEM    OCCURS 5 TIMES  PIC 9(2).
05  RED-ITEM REDEFINES ITEM  PIC 9(10).
```

is not legal. It may, however, be subordinate to an entry that contains an OCCURS or REDEFINES clause, as in

```
02  CLASS OCCURS 5 TIMES.
    05  ITEM-1    PIC X(2).
    05  ITEM-2    PIC X(2).
    05  NUM-ITEM-2  REDEFINES  ITEM-2    PIC 9(2).
```

This is provided the OCCURS clause does not define an item whose size is variable (this happens when the DEPENDING ON phrase is used with OCCURS).

In the example, ITEM-2 occurs five times, so procedure division references to it must be subscripted or indexed. In the data division, however, ITEM-2 may not be subscripted or indexed, so each (not just one) occurrence of ITEM-2 is redefined by NUM-ITEM-2, and procedure division references to NUM-ITEM-2 also must be subscripted or indexed.

6.2.2. PICTURE Clause

The PICTURE clauses describe program elementary data items and provide for special editing of items such as inserting signs or other characters or the suppression of leading zeros. A PICTURE clause may only be specified for elementary items, and one clause must be specified for each item.

A PICTURE clause consists of up to 30 characters used as symbols. Symbols are discussed in previous sections and the PICTURE clause is used in many coding examples. (Note that X in a PICTURE character-string represents any alphanumeric character from the computer's character set; 9 represents a character that is one of the numeric digits 0–9; while the A represents one of the 26 letters of the alphabet or the space.)

There are five categories of data items: alphabetic, alphanumeric, numeric, alphanumeric edited, and numeric edited. These are discussed in 6.2.2.1 through 6.2.2.4.

6.2.2.1. Alphabetic and Alphanumeric Fields

An item that is alphabetic must be described with a PICTURE that is all A's and B's (for blanks).

A PICTURE that describes an alphanumeric data item consists of all X's, or of a combination of the X, 9, and A symbols.

6.2.2.2. Numeric Fields

Descriptions of numeric fields consist of the symbols V, P, and S in addition to 9's. The symbol V in a PICTURE clause represents an assumed decimal point; it is not counted in the size of the item and does not appear if the item is printed or displayed.

The coding

```
05  ITEM  PIC  999V99
```

describes a 5-digit item that has two decimal places. If you omit the V, it is assumed the decimal point is to the right of the rightmost digit; thus, it is redundant to code

```
05  ITEM  PIC  99999V
```

The symbol P also indicates there is an assumed decimal point; the difference is that the decimal point is not within the data item. Although only one V is permissible in a PICTURE, you may use as many P's as are needed. (The use of both V and P in the same PICTURE is redundant.)

The P symbol must appear as a continuous string to the right or left of the rest of the PICTURE; the assumed decimal point is to the right or left of the string. Each digit position described by a P is considered to contain the value zero. So, if you move .000012345 to an item coded as

```
05  ITEM  PIC  PPPP99999
```

the ITEM will contain

```
1 2 3 4 5
```

which represents a value of

```
.000012345
```

If you move 12345 to an item described as

```
05  ITEM  PIC  99999PPPP
```

the ITEM will contain

```
1 2 3 4 5
```

which represents a value of

```
123450000
```

The symbol S in a PICTURE means the data item is signed. The S must be the leftmost character in the PICTURE and (like V) is not counted in the size of the data item (6.2.4). The PICTURE in

```
05  ITEM  PIC  S9999V99
```

means the item is a 6-digit signed number with two decimal places.

There is no such thing as an alphabetic edited data item, but you may edit alphabetic fields by inserting blanks. The symbol B in a PICTURE is replaced by a blank. If you move

```
ABCDEF
```

to

```
05  ALPHA-ITEM  PIC  AAABBAAA
```

the result is

```
ABC△△DEF
```

The edited item still is categorized as alphabetic.

6.2.2.3. Alphanumeric Edited Fields

To edit alphanumeric fields, you specify that blanks (B), zeros (0), or strokes (/) are to be inserted in the data item. If you code

```
05  ITEM   PIC  XX/XX/XX
```

strokes are inserted in the positions shown. If the value of DATE in

```
05  DATE   PIC 9(6)
```

is 012181, and you code

```
MOVE DATE TO ITEM
```

ITEM equals

```
Ø1/21/81
```

The DATE field, which is numeric, is known as the sending field; ITEM is alphanumeric edited which is known as the receiving field.

You may insert all three characters (B,0,/) into the same item. If a sending field described as PIC X(8) equals

```
A1B2C3D4
```

and the receiving field's PICTURE is

```
PIC  XBOXXBOX/XBOXXBOX
```

the edited result is

```
A Ø1B Ø2/C Ø3D Ø4
```

6.2.2.4. Numeric Edited Fields

You edit numeric fields so they are printed or displayed in a more readable form. For instance, you may prefer to print 43679216 and 00000025 as

```
43,679,216
```

and

```
25
```

Or you may want to insert a dollar sign and decimal point so that a field prints as one of the following formats:

```
$329.95
32995
```

You cannot, however, use numeric edited items in arithmetic operations.

Many more symbols are available for editing numeric fields than for alphabetic or alphanumeric fields. Again, you may use B, 0, and / to insert blanks, zeros, and strokes into data items. Additionally, you may insert commas, a currency symbol, a sign (+ or –), the letters DB (for debit), the letters CR (for credit), or a period.

- Decimal Point (Period)

  If you move

```
54319.27
```

  to a field described as

```
05  NUM  PIC  999,999.99
```

  the edited result is

```
054,319.27
```

  The period represents the decimal point. Do not use the period or comma as the last character in a PICTURE character-string.

- Z and Asterisk

  The Z and * symbols allow you to suppress leading zeros. If you move

```
54319.27
```

  to a field described as

```
05  NUM  PIC  Z99,999.99
```

  the edited result is

```
54,319.27
```

  Since the value represented by Z is zero, it is replaced by a blank. The * replaces leading zeros with an asterisk, so this change in the receiving item

```
05  NUM  PIC  *99,999.99
```

  changes the result to

```
*54,319.27
```

The Z and * substitute a space or asterisk only if the corresponding position in the data item is zero; otherwise, they have the same effect as the symbol 9. If the sending field equals

```
943567.24
```

and the receiving field is described as

```
05  REC-FIELD  PIC  ZZZ,ZZZ.99
```

The REC-FIELD equals

```
943,567.24
```

- Zero Suppression

When you edit a field with zero suppression, unwanted commas are dropped. Thus, if the receiving item is described as

```
05  REC-FIELD  PIC  ZZZ,ZZZ.99
```

and the sending field equals

```
129.37
```

the edited result is

```
129.37
```

not

```
,129.37
```

Special rules are in effect if the PICTURE is all Z's or all *'s.

If you describe a field as

```
05  ITEM  PIC  ZZZ.ZZ
```

or

```
05  ITEM  PIC  ***.**
```

and the value is zero, the entire data item is spaces or is equal to ***.**. If the value is .05, however, the data item equals .05 or ***.05, just as if the PICTURE was

```
05  ITEM  PIC  ZZZ.99
```

or

```
05  ITEM  PIC  ***.99
```

You may not use both Z and * in the same PICTURE. If asterisk is the zero suppression symbol, you can't use BLANK WHEN ZERO in the same data description.

- DB and CR

  You include DB or CR in a PICTURE if you want those letters to appear in the rightmost character postions of negative data items. If you move

  ```
  -1,543.97
  ```

  to an item described as

  ```
  05  ITEM  PIC  9,999.99CR
  ```

  ITEM prints as

  ```
  1,543.97CR
  ```

  Move +1,543.97 to the same data item and CR is replaced by spaces.

- Plus and Minus Signs

  The symbols + and - are similar to DB and CR, except they may occupy the rightmost or the leftmost character position. If a sending data item is negative, the minus sign always appears in the receiving item; if it's positive or zero, the plus sign appears only if + is the editing symbol. For example, if you move

  ```
  -34.51
  ```

  to

  ```
  05  ITEM  PIC  -Z,ZZZ.99
  ```

  or

  ```
  05  ITEM  PIC  +Z,ZZZ.99
  ```

  the edited result is

  ```
  -△△△34.51
  ```

But move

```
+34.51
```

to the same data items and the edited results are

```
△△△△34.51
```

and

```
+△△△34.51
```

Note that insertion of DB, CR, +, AND - depend on a number's sign, thus; the PICTURE of the sending field should include an S.

- Currency Symbol

You can insert the currency symbol as the leftmost character of a data item. However, a + or - may precede it. If the sending field has the value

```
545.29
```

and the receiving field is described as

```
05  TOTAL  PIC  $Z,ZZZ.99
```

the edited result is

```
$△△545.29
```

Two or more currency symbols, plus signs, or minus signs represent floating insertion characters; you use this type of editing if you want the insertion character to appear immediately to the left of the first nonzero character, rather than in a fixed position that may be several spaces from the leftmost digit of the number. For instance, if the sending item equals

```
125.39
```

and you describe the receiving item as

```
05  SALES  PIC  $ZZZ,ZZ9.99
```

or

```
05  SALES  PIC  -ZZZ,ZZ9.99
```

the edited result is

```
$△△△△125.39
```

or

```
-△△△△125.39
```

But if you describe the receiving item as

```
05  SALES  PIC  $$$$,$$9.99
```

or

```
05  SALES  PIC  ----,--9.99
```

the edited result is

```
$125.39
```

or

```
-125.39
```

- Floating Symbol

Include one more floating symbol than is needed to represent all the characters in the data item. If the sending field equals

```
725.44
```

and the receiving item is

```
05  COMM  PIC  $$$.99
```

the most significant digit is truncated and the edited result is

```
$25.44
```

The data description should be

```
05  COMM PIC  $$$$.99
```

Then, the edited result is

```
$725.44
```

As with Z, if you use a floating insertion character for all character positions, and the value is zero, the data item contains spaces. But if the value is a number less than 1, the edited result is the same as if the PICTURE was

```
05  ITEM  PIC  $$$.99
05  ITEM  PIC  ---.99
```

or

```
05  ITEM  PIC  +++.99
```

Table 6-1 summarizes the PICTURE symbols and what they represent; Table 6-2 includes examples of the various types of editing; and Table 6-3 lists the order of precedence of the characters used as symbols in a PICTURE clause. An X at an intersection indicates the symbol at the top of the column may precede the symbol at the left of the row. When a symbol appears twice, the first appearance represents its use to the left of the decimal point, and the second appearance represents its use to the right of the decimal point.

Table 6—1. Summary of PICTURE Symbols

Picture Symbol	Represents
A	An alphabetic character or space
B	Insert a blank or space
P	Assumed decimal point to left or right of data item
S	An operational sign is associated with data item
V	Assumed decimal point in data item
X	An alphanumeric character
Z	Suppression of leading 0's (replaced by blanks or spaces)
Asterisk (*)	Check protection (replaces leading 0's with asterisks)
Zero (0)	Insert 0
9	A numeric character
Stroke (/)	Insert stroke character
Comma (,)	Insert comma
Period (.)	Insert actual decimal point

Picture Symbol	Represents
CR	Insert character CR if data item is negative; insert two blanks or spaces if value is positive or zero
DB	Insert character DB if data item is negative; insert two blanks if value is positive or zero
Plus (+)	Insert in character position if data item value is positive or zero
+++...+	If more than one consecutive plus is present, indicates floating sign.
Minus (-)	Insert in charcter position if data item value is negative
---...-	If more than one consecutive minus is present, indicates floating sign.
Currency sign ($)	Insert in character position
$$$...$	If more than one consecutive $ is present, indicates floating currency sign.

Table 6—2. Examples of PICTURE Clause Editing

Edit Procedure	Sending Field		Receiving Field	
	Picture	Data	Picture	Data
Alphabetic editing				
Blank insertion	A(5)	ALPHA	AAABABA	ALP△H△A
Alphanumeric editing				
Zero insertion	X(5)	GLF01	00XXX0XX	00GLF001
Stroke insertion	X(6)	112481	XX/XX/XX	11/24/81
Blank insertion	X(9)	159440701	XXXBXXBXXXX	159△44△0701
Numeric editing				
Zero insertion	9(4)	4315	999900	431500
Stroke insertion	9(5)	32647	99/999	32/647
Blank insertion	9(5)	29141	99B999	29141
Comma insertion	9(4)	3419	9,999	3,419
Decimal point insertion	9(4)V9(2)	416375	9,999.99	4,163.75
CR insertion	S9(5)V9(2)	+5935198	ZZ,ZZZ.99CR	59,351.98
	S9(5)V9(2)	-5935198	ZZ,ZZZ.99CR	59,351.98CR
DB insertion	S9(5)V9(2)	+5935198	ZZ,ZZZ.99DB	59,351.98
	S9(5)V9(2)	-5935198	ZZ,ZZZ.99DB	59,351.98DB
Fixed plus sign	S9(4)V9(2)	+003978	+ZZZZ.99	+△△39.78
	S9(4)V9(2)	-003978	+ZZZZ.99	-△△39.78
	S9(4)V9(2)	+003978	ZZZZ.99+	39.78+
	S9(4)V9(2)	-003978	ZZZZ.99+	39.78-
Fixed minus sign	S9(4)V9(2)	+003978	-ZZZZ.99	39.78
	S9(4)V9(2)	-003978	-ZZZZ.99	-△△39.78
	S9(4)V9(2)	+003978	ZZZZ.99-	39.78
	S9(4)V9(2)	-003978	ZZZZ.99-	39.78-
Fixed currency sign	9(6)V9(2)	04124578	$ZZZ,ZZZ.99	$△41,245.78
Floating plus sign	S9(4)V9(2)	+000009	++,+++.++	+.09
	S9(4)V9(2)	000000	++,+++.++	△
	S9(4)V9(2)	+002435	++,+++.99	+24.35
	S9(4)V9(2)	-002435	++,+++.99	-24.35
Floating minus sign	S9(4)V9(2)	+000009	--,---.--	.09
	S9(4)V9(2)	-000009	--,---.--	-.09
	S9(4)V9(2)	+002435	--,---.99	24.35
	S9(4)V9(2)	-002435	--,---.99	-24.35
Floating currency symbol	S9(4)V9(2)	000147	$$,$$$.99	$1.47

Table 6—3. PICTURE Character Precedence Chart

Second Symbol		First Symbol																				
		Nonfloating Insertion Symbols									Floating Insertion Symbols						Other Symbols					
		B	O	/	,	.	+ or −	+ or −	CR or DB	cs	* or Z	* or Z	+ or −	+ or −	cs	cs	9	A or X	S	V	P	P
Nonfloating Insertion Symbols	B	x	x	x	x	x	x			x	x	x	x	x	x	x	x	x		x		x
	O	x	x	x	x	x	x			x	x	x	x	x	x	x	x	x		x		x
	/	x	x	x	x	x	x			x	x	x	x	x	x	x	x	x		x		x
	,	x	x	x	x	x	x			x	x	x	x	x	x	x	x			x		x
	.	x	x	x	x		x			x	x		x		x		x					
	+ or −																					
	+ or −	x	x	x	x	x				x	x	x			x	x	x			x	x	x
	CR or DB	x	x	x	x	x				x	x	x			x	x	x			x	x	x
	cs						x															
	Z or *	x	x	x	x		x			x	x											
	Z or *	x	x	x	x	x	x			x	x	x								x		x
Floating Insertion Symbols	+ or −	x	x	x	x					x			x									
	+ or −	x	x	x	x	x				x			x	x						x		x
	cs	x	x	x	x		x								x							
	cs	x	x	x	x	x	x								x	x				x		x
	9	x	x	x	x	x	x			x	x		x		x		x	x	x	x		x
	AX	x	x	x													x	x				
Other Symbols	S																					
	V	x	x	x	x		x			x	x		x		x		x		x		x	
	P	x	x	x	x		x			x	x		x		x		x		x		x	
	P																					

6.2.3. USAGE Clause

The USAGE clause specifies the form in which a data item is stored. In the format, COMPUTATIONAL and COMP are equivalent. If a numeric data item is used in computations, you should describe it as COMP. If you do not, the object program must convert it to COMP format each time it is needed for a computation.

Programmers seldom code USAGE IS DISPLAY (which indicates standard data format) because all data items not specified as COMP are assumed to be DISPLAY.

You may write the USAGE clause at the elementary or group level. At the group level, the USAGE clause applies to each elementary item in the group and may not be contradicted by any elementary item in the group. Thus, the following coding is not valid:

```
01  TOTALS  USAGE  IS COMPUTATIONAL.
    02  A   PIC   9(5).
    02  B   PIC   9(3)  USAGE IS DISPLAY.
    02  C   PIC  9(8).
```

This coding is valid, however:

```
01  TOTALS.
    02  A   PIC   9(5)  USAGE IS COMP.
    02  B   PIC   9(3)  USAGE IS DISPLAY.
    02  C   PIC   9(8)  USAGE IS COMP.
```

6.2.4. SIGN Clause

The S in a PICTURE clause indicates a numeric data item is signed. Normally, the position and method of representation of that sign are defined by the implementor, but the SIGN clause allows you to specify the sign's location and, if you use the SEPARATE CHARACTER phrase, its form.

If you use a SIGN clause, but not the SEPARATE CHARACTER phrase, the sign is associated with the first or last (LEADING or TRAILING) digit in the item, and the implementor defines how the sign is represented. If you use the SEPARATE CHARACTER phrase, the sign is not associated with a digit position; it's a separate character (either – or +) preceding or following (LEADING or TRAILING) the number.

Suppose a card record has a 5-digit numeric field. If a + or – is punched preceding the number to indicate its sign, you should describe the field as

```
05  AMOUNT    PIC  S9(5)
              SIGN IS LEADING SEPARATE CHARACTER.
```

In this example, the S is counted in the size of the item; i.e., AMOUNT has six rather than five character positions.

You cannot use the SIGN clause in the description of a COMPUTATIONAL item.

6.2.5. SYNCHRONIZED Clause

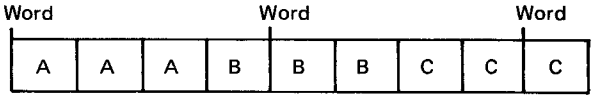
Main storage in some computers is organized so that it contains natural addressing boundaries. The SYNCHRONIZED clause aligns elementary data items on these natural boundaries. That is important, because some computers cannot perform certain operations on data not so aligned. You never have to use the SYNCHRONIZED clause, but when you don't, the program executes less efficiently – the compiler must generate extra instructions that align data items not stored on proper boundaries.

The SYNCHRONIZED clause is implementor-dependent (it does not work the same way in all operating environments). Commonly, it's included only in data descriptions of items used in arithmetic operations or as subscripts.

You specify the SYNCHRONIZED clause (SYNC is equivalent) to ensure that an elementary data item is aligned so that no other data item occupies any of the character positions between the leftmost and rightmost natural boundaries containing the item. The natural boundaries are not the same in each operating environment. Assume the word, which consists of four character positions, represents the natural boundaries. If you code

```
01  GROUP.
    02  A   PIC   999  COMP.
    02  B   PIC   999  COMP.
    02  C   PIC   999  COMP.
```

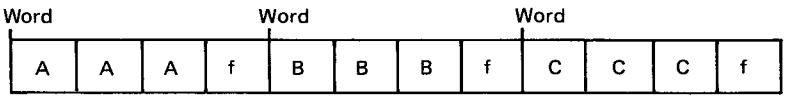
the items may be stored as follows:



But if you code

```
01  GROUP.
    02  A   PIC   999   COMP   SYNC   LEFT.
    02  B   PIC   999   COMP   SYNC   LEFT.
    02  C   PIC   999   COMP   SYNC   LEFT.
```

the items are stored this way:

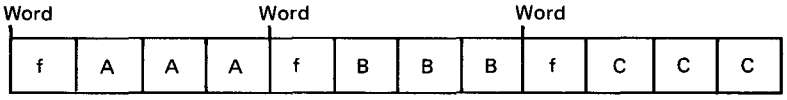


FILLER slack bytes (f) are inserted to fill character positions that are part of a word, but are not needed by the data item. Thus, each item may begin on a word boundary.

As you might expect, the LEFT option positions the item beginning at the leftmost position within the boundary, and the RIGHT option positions the item so it ends at the rightmost character position within the boundary. Thus,

```
01  GROUP.
    02  A   PIC   999   SYNC  RIGHT  COMP.
    02  B   PIC   999   SYNC  RIGHT  COMP.
    02  C   PIC   999   SYNC  RIGHT  COMP.
```

is stored as



In some operating environments, you use SYNC without the LEFT or RIGHT option (the specific positioning of the synchronized item is determined by the implementor).

If you use the SYNCHRONIZED clause in the description of an item that contains or is subordinate to an item that contains an OCCURS clause, each occurrence of the data item is synchronized.

6.2.6. JUSTIFIED Clause

You use the JUSTIFIED clause to make an alphabetic or alphanumeric data item right justified. When you describe a receiving item as JUSTIFIED, the rightmost character of the sending item is aligned with the rightmost position of the receiving item.

Normally, characters are moved one at a time, from left to right, from the sending item to the receiving item, until the receiving item is filled.

If the sending item equals SENDING and the receiving item is described as

```
02  REC   PIC   X(9).
```

the the value is stored in REC as

```
SENDING△△
```

Now, if the REC description is changed to

```
02  REC   PIC   X(5).
```

the the value stored in A is

```
SENDI
```

The JUSTIFIED clause has the reverse effect (it moves characters to the receiving item from right to left).

If

```
SENDING
```

is sent to an item described as

```
02   REC   PIC   X(9)   JUST   RIGHT.
```

it is stored as

```
△△SENDING
```

Now, if the description is changed to

```
02   REC   PIC   X(5)   JUST   RIGHT.
```

the value is stored as

```
NDING
```

Do not specify the JUSTIFIED clause for a data item that is numeric or for which editing is specified.

6.2.7. BLANK WHEN ZERO Clause

The BLANK WHEN ZERO clause provides for the blanking of an item when its value is zero. It may be specified only for an elementary item whose PICTURE is numeric or numeric edited.

If you move zeros to

```
05   TOTAL   PIC   999   BLANK   WHEN   ZERO.
```

the value of TOTAL becomes spaces.

6.2.8. VALUE Clause

The VALUE clause assigns a value to a data item. We used the VALUE clause in the working-storage section in the program in Section 2 to define constants and to initialize data items used in computations.

It has two formats:

Format 1:

```
VALUE IS literal
```

Format 2:

```
88   condition-name;  {VALUE IS  }  literal-1  [{THROUGH}  literal-2 ]
                      {VALUES ARE}              {THRU    }
   [, literal-3  [{THROUGH}  literal-4 ] ] ...  .
                 {THRU    }
```

Format 1 is not valid in the file or linkage sections; Format 2 is only valid in condition-name entries (6.2.9).

Make certain the literal in the VALUE clause is compatible with the data item PICTURE (i.e., if the category of the item is numeric, the literal must be numeric).

Note that nonnumeric literals are enclosed by quotation marks, as in

```
02  CLASS    PIC   XX   VALUE   ''1A''.
```

and numeric literals are not:

```
02  CLASS    PIC   99   VALUE  25.
```

A figurative constant may replace the literal. The entry initializes TOTAL to zeros:

```
02  TOTAL  PIC  9(9)V99   VALUE  ZEROS.
```

Do not use the VALUE clause (except in condition-name entries) in the description of an item that contains, or is subordinate to, an item that contains an OCCURS or a REDEFINES clause. Thus, the following is invalid:

```
02  CLASS    OCCURS  5  TIMES.
    03  DIV    PIC   X(2)   VALUE  ''10''.
    03  TERR   PIC   X(2)   VALUE  ''2C''.
```

If you wish to assign values to a table, you code

```
02  CL-VAL     PIC   X(20)   VALUE  ''102C142B183E1AF02C52''.
02  CL-REDF  REDEFINES  CL-VAL.
    04   CLASS  OCCURS 5 TIMES.
         06   DIV   PIC   X(2).
         06   TERR  PIC   X(2).
```

You may use the VALUE clause in the description of a group item. When you do, the literal must be nonnumeric or a figurative constant, and you may not use JUSTIFIED, SYNCHRONIZED, COMPUTATIONAL, or VALUE in the description of any elementary item in the group. Thus,

```
02  POLICY-NO   VALUE   ''003124AA21''.
    03  DIV     PIC  X(6).
    03  CLASS    PIC  X(4).
```

the DIV is assigned an initial value of 003124 and CLASS equals AA21.

6.2.9. Conditional Variables

Conditional variables help make a program self-documenting. They allow you to assign meaningful names to the possible values of a data item; then, in the procedure division, use those names (called condition-names) to test its value.

You assign the names by using the VALUE clause in an 88-level entry immediately following the conditional variable. For instance, the coding

```
02  MARITAL-STATUS     PIC  X(1).
```

may have a value of 1 of 2, for single or married, respectively. You can test its value by coding

```
IF MARITAL-STATUS IS EQUAL TO ''1'' ...
IF MARITAL-STATUS IS EQUAL TO ''2'' ...
```

If you make MARITAL-STATUS a conditional variable, the test can be more readable. You use 88-level entries to assign the condition-names SINGLE to the value "1" and MARRIED to "2":

```
02  MARITAL-STATUS   PIC  X(1).
    88  SINGLE   VALUE  ''1''.
    88  MARRIED  VALUE  ''2''.
```

Then, testing MARITAL-STATUS is simplified:

```
IF SINGLE ...
IF MARRIED ...
```

when MARITAL-STATUS equals 1, the first statement is true; when it equals 2, the second is true.

You may use Format 2 of the VALUE clause to assign a range of values to a conditional variable. If you code

```
02  MONTH     PIC  X(2).
    88  FIRST-QUARTER      VALUES ARE ''01'' THRU ''03''.
    88  SECOND-QUARTER     VALUES ARE ''04'' THRU ''06''.
    88  THIRD-QUARTER      VALUES ARE ''07'' THRU ''09''.
    88  FOURTH-QUARTER     VALUES ARE ''10'' THRU ''12''.
```

and MONTH equals 05, the following is true and the range of values includes the end values.

```
IF SECOND-QUARTER ...
```

When you use the THRU option, the literals must be in ascending sequence, and the following is invalid:

```
88  FIRST-QUARTER      VALUES ARE ''03'' THRU ''01''.
```

A condition-name, like a data-name, must be unique in the program or be made unique through qualification and indexing or subscripting. If you code

```
02  CLASS  OCCURS  10  TIMES   PIC  X(4).
    88  SPECIAL-GROUP      VALUE    ''C141''.
```

SPECIAL–GROUP, like CLASS, must be subscripted, as in

```
IF SPECIAL-GROUP (4)  ...
```

This is equivalent to coding

```
IF CLASS (4) IS EQUAL TO ''C141'' ...
```

If you code

```
02  CLASS    OCCURS  10  TIMES   PIC  X(4).
    88  SPECIAL-GROUP     VALUE    ''C141''.
02  TERR     OCCURS  12  TIMES   PIC  X(4).
    88  SPECIAL-GROUP     VALUE    ''6B2K''.
```

SPECIAL-GROUP must be qualified and subscripted, as in

```
IF SPECIAL-GROUP OF TERR (6)  ...
```

This is the same as

```
IF TERR (6) IS EQUAL TO ''6B2K''  ...
```

6.2.10. RENAMES Clause

You use the RENAMES clause to assign alternate names to elementary items within a record. It is similar to the REDEFINES clause (it allows you to reference data items in more than one way) but, since it has no PICTURE clause associated with it, it does not provide an alternate description of the items.

A REDEFINES clause may redefine an entire record. The RENAMES clause may not – it renames only elementary items or groups of elementary items within a record. It may not rename another entry that has a RENAMES clause, or a 77- or 88-record entry.

You write the RENAMES clauses (there may be more than one associated with each record) immediately following the record that contains the elementary items being renamed. It begins with level-number 66. If you code

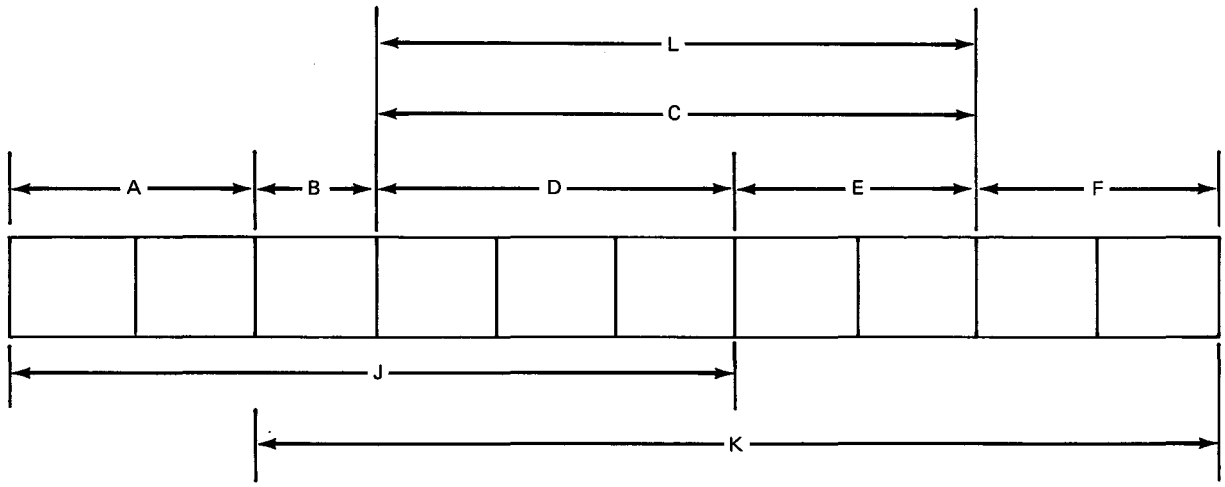
```
01  RECD.
    02  A    PIC  X(2).
    02  B    PIC  X(1).
    02  C    PIC  X(2).
    66  D RENAMES B.
```

D references the same area as B.

You use the THROUGH option (THROUGH and THRU are equivalent) to rename a group of elementary items. If you code

```
01  RECD.
    02  A    PIC  X(2).
    02  B    PIC  X(1).
    02  C.
        03  D  PIC  X(3).
        03  E  PIC  X(2).
    02  F    PIC  X(2).
    66  J  RENAMES A THRU D.
    66  K  RENAMES B THRU F.
    66  L  RENAMES C.
```

RECD is structured this way:



J and K are group items that include elementary items A, B, and D, and B, D, E, and F, respectively. L also is a group item, because it renames a group item (C).

When you use THRU, the second data-name specified in the clause may not precede the first; thus, the RENAMES clauses in the following are invalid:

```
01  RECD.
    02  A  PIC   X(2).
    02  B  PIC  X(1).
    02  C.
        03  D   PIC   X(3).
        03  E   PIC   X(2).
    02  F  PIC  X(2).
    66  J RENAMES D THRU B.
    66  K RENAMES E THRU C.
```

In addition, neither of the data-names specified in the THRU clause may have, or be subordinate to, an item that has an OCCURS clause. Either data-name may be qualified.

The subject of the RENAMES clause (such as J or K in the previous example) may not be used as a qualifier. It may be qualified only by the associated level 01, FD, CD, or SD entries (in this case, RECD).

6.3. LEVEL 77 ENTRY

Often you need to define data items (perhaps counters, constants, or subscripts) that have no logical relationship to other data items and do not need to be subdivided. These belong in the working-storage section and are called noncontiguous elementary items (level 77 entries).

These entries consist of the level-number 77, a data-name, and a PICTURE clause or USAGE IS INDEX clause. The other data description clauses may be used if needed. For example, a subscript may be defined as

```
77   DIV-SS   PIC  9(2)   COMP   VALUE  1.
```

6.4. CODING EXAMPLE

This sample data division is used in a program that reads transactions from a tape, updates a master inventory file, and prints a report. The bulk of the division is descriptions (in the file section) of three files – INVENTORY–MASTER, INVENTORY–TRANS, and PRINT–FILE. Print lines for the report are built in working-storage record areas (PRINT-AREA and TOTAL-PRINT-AREA). The working-storage section also includes two level 77 entries that accumulate totals.

The RECORD CONTAINS and DATA RECORD IS clauses are included in each FD as documentation. The records in INVENTORY-MASTER and INVENTORY-TRANS are blocked for efficiency. Both files have standard label records and VALUE OF clauses tell what is expected in particular fields of the record. Printer files may not be blocked or labeled; therefore, the FD for PRINT-FILE has no BLOCK CONTAINS clause, and specifies LABEL RECORDS ARE OMITTED. It does include a LINAGE clause that defines the dimensions of a logical page. The page body is 75 lines plus 5-line margins at the top and bottom. Lines 70 to 75 form a footing area.

The print areas are set up in working-storage so we can use the VALUE clause to include literals such as PRINT-AREA and TOTAL ON HAND in TOTAL-PRINT-AREA. The VALUE clauses in the file section are in the condition-name entries associated with the conditional variable TRANS-TYPE. The numeric fields in PRINT-AREA and TOTAL-PRINT-AREA are described with editing symbols in the PICTURE clause to suppress printing of leading zeros.

Data Division Coding:

```
DATA DIVISION.
FILE SECTION.
FD   INVENTORY-MASTER
     BLOCK CONTAINS 50 RECORDS
     RECORD CONTAINS 68 CHARACTERS
     LABEL RECORDS ARE STANDARD
     VALUE OF FILE-ID IS ''INVEN-MSTR''
     DATA RECORD IS MASTR-RECD.
01   MASTR-RECD.
     02  NAME            PIC  X(10).
     02  PART-NO         PIC  X(6).
     02  DESC            PIC  X(24).
     02  QTY-ON-HAND     PIC  9(9).
     02  QTY-ON-ORDER    PIC  9(9).
     02  UNIT-PRICE      PIC  9(8)V9(2).
```

(continued)

```
FD  INVENTORY-TRANS
    BLOCK CONTAINS 150 RECORDS
    RECORD CONTAINS 10 CHARACTERS
    LABEL RECORDS ARE STANDARD
    VALUE OF TAPE-IDENT IS ''INVEN-TRANS''
    DATA RECORD IS INVENTORY-TAPE.
01  INVENTORY-TAPE.
    02  TRANS-TYPE    PIC X(1).
        88  ADD-TO-INVENTORY    VALUE    ''A''.
        88  SUBTRACT-FROM-INVENTORY    VALUE  ''B''.
    02  TRANS-PART-NO     PIC X(6).
    02  TRANS-QTY     PIC 9(3).
FD  PRINT-FILE
    RECORD CONTAINS 132 CHARACTERS
    LABEL RECORDS ARE OMITTED
    DATA RECORD IS PRINTLINE
    LINAGE IS 75 LINES
         WITH FOOTING AT 70
         LINES AT TOP 5
         LINES AT BOTTOM 5.
01  PRINTLINE     PIC X(132).
WORKING-STORAGE SECTION
77  TOT-ON-HAND    PIC 9(10)    COMP.
77  TOT-ON-ORDER   PIC 9(10)    COMP.
01  PRINT-AREA.
    02  FILLER    PIC X(4)   VALUE ''PART''.
    02  FILLER    PIC X(5)   VALUE SPACES.
    02  PART-OUT    PIC X(6).
    02  FILLER    PIC X(8)   VALUE SPACES.
    02  NAME-OUT PIC X(10).
    02  FILLER    PIC X(8)   VALUE SPACES.
    02  DESC-OUT    PIC X(24).
    02  FILLER    PIC X(8)   VALUE SPACES.
    02  ON-HAND-OUT    PIC ZZZ,ZZZ,ZZ9.
    02  FILLER    PIC X(8)   VALUE SPACES.
    02  ON-ORDER-OUT   PIC ZZZ,ZZZ,ZZ9.
    02  FILLER    PIC X(8)   VALUE SPACES.
    02  PRICE-OUT     PIC *,***,***.99.
    02  FILLER    PIC X(9).
01  TOTAL-PRINT-AREA.
    02  FILLER    PIC X(30)   VALUE SPACES.
    02  FILLER    PIC X(15)   VALUE ''TOTAL ON HAND△△''.
    02  ON-HAND-TOT   PIC  Z,ZZZ,ZZZ,ZZ9.
    02  FILLER    PIC X(15)   VALUE SPACES.
    02  FILLER    PIC X(16)   VALUE ''TOTAL ON ORDER△△''.
    02  ON-ORDER-TOT    PIC    Z,ZZZ,ZZZ,ZZ9.
    02  FILLER    PIC X(30)   VALUE SPACES.
```

7. Procedure Division

7.1. FORMATS

In the procedure division, you write the instructions needed to solve your problem. The instructions manipulate data (the data described in the data division and literals).

The procedure division consists of:

1. Statements - Syntactically correct combinations of words and symbols beginning with a COBOL verb. The statements are joined to form:

2. Sentences - One or more statements ending with a period and space.

3. Paragraphs - One or more sentences from a paragraph (although, technically a paragraph may consist of a paragraph-name and no sentences).

4. Sections - Not required but in the procedure division you can group paragraphs into sections. In fact, to take advantage of such features as declaratives and segmentation, you must use sections.

The procedure division is organized based on one of the following two formats.

Format 1:

```
PROCEDURE DIVISION [USING data-name-1 [,data-name-2] ... ].
[ DECLARATIVES.                                                    ]
[ { section-name SECTION [segment-number]. declarative-sentence } ... ]
[ {[paragraph-name. [sentence] ... ] ...                         } ]
[ END DECLARATIVES.                                                ]
{ section-name SECTION [segment-number]. } ...
{ [paragraph-name. [sentence] ... ] ...  }
```

Format 2:

```
PROCEDURE DIVISION [USING data-name-1 [,data-name-2] ... ].
{paragraph-name. [sentence] ... } ...
```

You cannot mix formats. If you group some paragraphs into sections, you must group all paragraphs into sections.

Declarative sections are executed only in special situations that may or may not arise during program execution. USE statements (7.8.1) within the sections define those situations.

When you use declaratives, they must be at the beginning of the procedure division, preceded by the key word DECLARATIVES and followed by the key words END DECLARATIVES. Program execution begins with the first statement of the procedure division after the declaratives.

7.2. EXPRESSIONS

COBOL expressions are meaningful combinations of data-names, literals, and operators that reduce to a single value. There are two types of expressions: arithmetic and conditional. Arithmetic expressions reduce to a single numeric value; conditional expressions reduce to a single truth value (true or false).

7.2.1. Arithmetic Expressions

You use arithmetic expressions as operands in arithmetic statements or conditional expressions. There are five types:

1. Identifiers of numeric elementary items. An identifier is a data-name plus any qualifiers, subscripts, or indexes needed to make that data-name unique. For example:

```
01   IN-RECD.
     05  SUBTOT      PIC 9(9).
     05  FINTOT      PIC 9(9).
     05  TYPE        PIC 9.
01   OUT-RECD.
     05  SUBTOT      PIC 9(9).
     05  FINTOT      PIC 9(9).
```

   The data-name TYPE is unique and this is an identifier. The data-names SUBTOT and FINTOT, however, are not unique; thus, to be identifiers they must be qualified as in

```
SUBTOT OF IN-RECD
```

   or

```
FINTOT OF OUT-RECD
```

2. Numeric literals. For example:

```
75
```

3. Identifiers and literals separated by arithmetic operators:

```
TRANS-AMOUNT + 15
```

4. Arithmetic expressions enclosed in parentheses:

```
(TRANS-AMOUNT + 15)
```

5. Two or more arithmetic expressions separated by arithmetic operators:

```
(TRANS-AMOUNT + 15) * 2.5
```

Five binary and two unary arithmetic operators are valid operators in arithmetic expressions.

The binary operators are:

+	Addition
-	Subtraction
*	Multiplication
/	Division
**	Exponentiation

The unary operators are:

+	Multiplication by +1
-	Multiplication by -1

A space must precede and follow an arithmetic operator.

A unary operator may precede any arithmetic expression.

Arithmetic expressions are evaluated, from left to right, in this order:

1st - Unary plus and minus

2nd - Exponentiation

3rd - Multiplication and division

4th - Addition and subtraction

Thus:

```
4 + 5 * 3 - 2 ** 3
```

Equals

```
4 + 5 * 3 - 8
```

Which equals

```
4 + 15 - 8
```

Which equals

```
19 - 8
```

Which equals

```
11
```

You can clarify arithmetic expressions (or change the order in which their elements are evaluated) by using parentheses. Elements within parentheses are evaluated first. Although it is evaluated in the same way, the expression in the previous example is easier to read when written as

```
4 + (5 * 3) - (2 ** 3)
```

When you change the location of the parentheses, you change the order in which the elements are evaluated, thus changing the value of the expression.

Thus:

```
(4 + 5) * (3 - 2 ** 3)
```

Equals

```
9 * (3 - 8)
```

Which equals

```
9 * - 5
```

Which equals

```
- 45
```

When parentheses are nested within other parentheses, the evaluation proceeds from the innermost to the outermost set. Note the order of evaluation of this expression:

```
22 + (18 - 12) + (6 ** 2) * 5) - 40
22 + (18 - 12) + (36 * 5) - 40
22 + 6 + (36 * 5) - 40
22 + 6 + 180 - 40
28 + 180 - 40
208 - 40
168
```

Expressions must begin with a left parenthesis, a unary operator, or a variable (an identifier or literal), and end with a right parenthesis or a variable. Each left parenthesis must have, and be to the left of, a corresponding right parenthesis.

Table 7-1 summarizes how operators, variables, and parentheses are combined to form arithmetic expressions. P indicates a permissible combination; an I indicates an invalid combination.

Table 7—1. Combination of Symbols in Arithmetic Expressions

First Symbol	Second Symbol				
	Variable	* / ** - +	Unary + or -	(	)
Variable	I	P	I	I	P
* / ** + -	P	I	P	P	I
Unary + or -	P	I	I	P	I
(	P	I	P	P	I
)	I	P	I	I	P

Thus, a variable such as SUBTOT may be followed only by *, /, **, -, or + or (provided there is a corresponding left parenthesis) a right parenthesis. Thus, the following are valid arithmetic expressions:

```
SUBTOT + (.25 * RATE)
(- FINTOT / .05)
- .06 - - .06 * RATE
A**B
```

Invalid expressions are as follows:

```
+ - .06 * 5
(SUBTOT + (.25 * RATE)
SUBTOT FINTOT + 3
- 21 (SUBTOT / .05)
```

7.2.2. Conditional Expressions

You use conditional expressions in IF, PERFORM, and SEARCH statements to control program branching (control goes one way if the expression is true, another way if it is false). There are two categories of conditional expressions: simple and complex.

7.2.2.1. Simple Conditions

One simple condition is called a *relation condition;* it is the comparison of two operands, each represented by an identifier, a literal, or the value resulting from an arithmetic expression. The format is:

```
{identifier-1                  }{IS [NOT] GREATER THAN}{identifier-2                  }
{literal-1                     }{IS [NOT] LESS THAN   }{literal-2                     }
{arithmetic-expression-1       }{IS [NOT] EQUAL TO    }{arithmetic-expression-2       }
                                {IS [NOT] >           }
                                {IS [NOT] <           }
                                {IS [NOT] =           }
```

NOTE:

The required relational characters >, <, and = are not underlined to avoid confusion with other symbols such as ⩾ (greater than or equal to).

The first operand is the subject of the condition; the second is the object. The subject and object may not both be literals.

The operator between the operands (called *relational operator*) specifies the type of comparison. Each reserved word comprising the relational operator must be preceded and followed by a space.

When you use NOT in relational operators, the condition tested for is exactly opposite to what is true when NOT is not present. For instance:

```
IF DAMAGE NOT = 1000
```

then

```
DAMAGE > or < 1000
```

is true, but

```
DAMAGE = 1000
```

is false.

When both operands are numeric, their algebraic values are compared; the number of digits in the operands and their USAGE need not be the same. Thus,

```
12
```

is equal to

```
12.00
```

Unsigned numeric operands are considered positive. Zero is considered a unique value regardless of the sign.

When both operands are nonnumeric (alphabetic, alphanumeric, alphanumeric edited, numeric edited) or when just one is numeric (it must be an integer), the comparison is based on the program collating sequence. A character that precedes another in the collating sequence is considered less than that character.

When the operands are the same size, the comparison begins with the leftmost characters and continues until a pair of unequal characters are encountered (then the operand containing the character that is higher in the collating sequence is considered greater). If no unequal characters are found, the operands are equal. The expression

```
''BIT'' GREATER THAN ''BIG''
```

is true if T follows G in the collating sequence.

When the operands are of unequal size, spaces are added to the smaller operand until the operands are the same size. For instance, the expression

```
''FORT'' LESS THAN ''FORTH''
```

is equivalent to

```
''FORT△'' LESS THAN ''FORTH''
```

and is true if △ precedes H in the collating sequence.

Another simple condition is the condition-name condition which is another way of expressing a relational condition. The rules for both are the same. The difference is that instead of using two operands, as you do in a relational condition comparison, you only have to give the condition-name in a condition-name comparison, such as

```
IF condition-name
```

or

```
IF NOT condition-name
```

When the comparison takes place, the condition-name is compared to a conditional variable and, if the two are equal, the comparison is true. To use this type of comparison you must assign a value to the condition-name and indicate the conditional variable associated with the condition-name. This is done by using 88-level entries under a data item. The 88-level entries, assigned a value with the VALUE clause, are the condition-names and the data item is the conditional variable. The condition-name comparison is a good way to assign descriptive names to some otherwise unclear test conditions making a program easier to read. (For more detail and examples, see 6.2.9).

Another condition that can be tested in the procedure division is the switch-status condition. This tests the status of an implementor-defined switch which is either on or off. The implementor-defined switch is identified in the SPECIAL-NAMES paragraph of the ENVIRONMENT DIVISION by an implementor-name and a condition-name (see 5.3). The result of a switch-status condition test is true if the switch is set to the specified position corresponding to the condition-name.

Another type of simple condition is the class condition; it determines whether an operand is numeric (consists entirely of the characters 0–9) or alphabetic (consists entirely of the letters of the alphabet and the space). The format is:

```
identifier IS [NOT] { NUMERIC    }
                    { ALPHABETIC }
```

You cannot use the NUMERIC test for an item described in its PICTURE clause as alphabetic, or for a group item composed of elementary items whose PICTURE clauses contain the S symbol. Similarly, you cannot use the ALPHABETIC test for an item described as numeric.

When you don't use S in the PICTURE clause of an item, it tests as numeric only if its contents are numeric and no operational sign is present. When S is in the PICTURE clause, the item tests as numeric only when its contents are numeric and a valid operational sign is present. The + and – symbols are the valid signs for items described with the SIGN IS SEPARATE clause; otherwise, the implementor defines what constitutes a valid sign.

Another simple condition (the sign condition) determines whether an arithmetic expression is less than (NEGATIVE), greater than (POSITIVE), or equal to zero. The format is:

```
arithmetic-expression is [NOT] { POSITIVE }
                               { NEGATIVE }
                               { ZERO     }
```

The arithmetic expression must reference at least one variable; thus,

```
2 + 3 * 5 POSITIVE
```

is not a valid expression, but

```
2 + 3 * COUNTER POSITIVE
```

is valid.

7.2.2.2. Complex Conditions

Complex conditions use the logical connectors AND or OR to combine conditions, or the logical operator NOT to negate conditions. An expression that combines conditions with the AND operator is true if both conditions are true, and false if one or both of the conditions are false. If you use OR, the expression is true if one or both of the conditions are true, and false if both conditions are false. The NOT operator reverses truth values; the expression is true if the condition that is negated is false, and false if it is true.

In other words:

```
TRUE AND TRUE = TRUE
TRUE AND FALSE = FALSE
FALSE AND FALSE = FALSE
```

and

```
TRUE OR TRUE = TRUE
TRUE OR FALSE = TRUE
FALSE OR FALSE = FALSE
```

and

```
NOT (TRUE AND TRUE) = FALSE
NOT (TRUE AND FALSE) = TRUE
NOT (TRUE OR FALSE) = FALSE
```

One type of complex condition, negated simple conditions, effects the opposite truth value for simple conditions. Its format is:

```
NOT simple-condition
```

So, if the simple condition

```
A = B
```

is true, the negated simple condition

```
NOT A = B
```

is false

When conditions include AND or OR, they are called combined conditions. The format is:

```
condition {{AND} condition} ...
          {{OR }
```

In the format, condition may be a

- simple condition;
- negated simple condition;
- combined condition;
- negated combined condition (NOT followed by a combined condition enclosed within parentheses); or
- permissible combinations of the preceding as defined in Table 7-2. (The table also shows the ways you may parenthesize conditions and logical operators.)

Table 7—2. Combinations of Conditions, Logical Operators, and Parentheses

Given the following element:	Location in conditional expression		In a left-to-right sequence of elements:	
	First	Last	Element (when not first) may be immediately preceded by only:	Element (when not last) may be immediately followed by only:
Simple condition	Yes	Yes	OR, NOT, AND, (	OR, AND,)
OR or AND	No	No	simple-condition,)	Simple condition, NOT, (
NOT	Yes	No	OR, AND, (	Simple condition, (
(	Yes	No	OR, NOT, AND, (	Simple condition, NOT,(
)	No	Yes	Simple condition,)	OR, AND,)

Often you can abbreviate combined relation conditions. If simple relation conditions combined with AND or OR have the same subject, you omit the subject in all but the first condition. For instance, you can write either

```
A = B AND A > C
A = B AND > C
```

And you can write either

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

If both the relational operator and the subject are repeated you may drop all but the first occurrence of both of them:

```
A = B AND A = C
```

This equals

```
A = B AND C
```

Also:

```
A > B AND A > C AND A > D AND A > E
```

This equals

```
A > B AND C AND D AND E
```

Note that you can omit the relational operator only when the immediately preceding operator is the same:

```
A > B OR A > C AND A = D OR A = E OR A NOT > F
```

The abbreviation is:

```
A > B OR C AND = D OR E OR NOT > F
```

The word NOT can be confusing in an expression, because sometimes it is part of a relational operator and sometimes it is a logical operator:

```
NOT A NOT = B
```

The first NOT is a logical operator and the second is part of the relational operator NOT =. The rule is that if NOT is followed by EQUAL (=), GREATER (>), or LESS (<) it is part of a relational operator; otherwise, it is a logical operator indicating an opposite truth value for the condition.

When you negate combined conditions, use parentheses to show that the negation applies to more than the immediately following simple condition. To negate

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

you must code

```
NOT (A NOT = B AND = C)
```

rather than

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

And the latter is equivalent to

```
NOT (A NOT = B) AND (A = C)
```

Note the difference in the evaluation of the expressions.

If the simple conditions A NOT = B and A = C are both false, the expression

```
NOT (A NOT = B AND A = C)
```

is evaluated as

```
NOT ((false) AND (false))
```

or

```
NOT (false)
```

or true.

But the expression

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

is evaluated as

```
NOT (false) AND (false)
```

or

```
true AND false
```

or false.

As with arithmetic expressions, you can change the order of evaluation of complex conditions by using parentheses. Conditions within parentheses are evaluated first, and, within nested parentheses, the evaluation proceeds from the innermost to the outermost sets of parentheses.

When you do not use parentheses, or when parenthesized conditions are at the same level of inclusiveness, complex conditions are evaluated in this order:

1. Values are established for arithmetic expressions.
2. Truth values for simple conditions are established in the following order:
   a. Relation (following the expansion of any abbreviated conditions)
   b. Class
   c. Condition-name (88-level entries)
   d. Switch-status (switch defined in SPECIAL-NAMES paragraph)
   e. Sign
3. Truth values are established for negated simple conditions.
4. Truth values are established for combined conditions, with AND logical operators taking precedence over OR.
5. Truth values are established for negated combined conditions.

When parentheses do not completely specify the sequence of evaluation, the order of evaluation of operations on the same hierarchical level is from left to right.

Following are examples of abbreviated combined relation conditions and their expanded equivalents, evaluated to arrive at truth values. Assume for all examples that A = 8, B = 5, C = 12, and D = 10.

Example 1:

If the abbreviated condition is

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

the expanded equivalent is

```
((A > B) AND (A NOT < C)) OR (A NOT < D)
```

This equals

```
(true AND false) OR (false)
false OR false
false
```

Example 2:

If the abbreviated condition is

```
A NOT EQUAL B OR C
```

The expanded equivalent is

```
(A NOT EQUAL B) OR (A NOT EQUAL C)
```

This equals

```
true OR true
true
```

Example 3:

If the abbreviated condition is

```
NOT A = B OR C
```

The expanded equivalent is

```
(NOT (A = B)) OR (A = C)
```

This equals

```
(NOT false) OR false
true OR false
true
```

Example 4:

If the abbreviated condition is

```
NOT (A > B OR < C)
```

The expanded equivalent is

```
NOT ((A > B) OR (A < C))
```

This equals

```
NOT (true OR true)
NOT (true)
false
```

Example 5:

If the abbreviated condition is

```
NOT (A NOT > B AND C AND NOT D)
```

The expanded equivalent is

```
NOT ((((A NOT > B) AND (A NOT > C)) AND (NOT (A NOT > D))))
```

This equals

```
NOT (((false AND true) AND (NOT true)))
NOT ((false AND false))
NOT (false)
true
```

7.3. STATEMENTS AND SENTENCES

Statements are the basic functional components of COBOL procedures. Just as clauses make up sentences in normal English, statements make up COBOL sentences. There are three types of statements: conditional, compiler-directing, and imperative.

7.3.1. Conditional Statements

Conditional statements specify that truth values of conditional expressions are to be found, and the truth values determine how the program proceeds. For instance, when the following is executed:

```
IF LINE-COUNT > 56 PERFORM HEADING-ROUTINE
     ELSE GO TO MAIN
```

the truth value of the expression

```
LINE-COUNT > 56
```

determines where control passes. When the expression is true, control passes to

```
PERFORM HEADING-ROUTINE
```

When the expression is false, control passes to

```
GO TO MAIN
```

Within the IF statement, the PERFORM HEADING-ROUTINE and GO TO MAIN are imperative statements. (See 7.3.3.)

Table 7-3 lists the COBOL verb and required phrases that constitute valid conditional statements. Note that except for the IF and SEARCH verbs, these verbs form conditional statements only when used with the phrases listed in the table. Thus, if you use ADD without the SIZE ERROR option, the statement is not conditional.

Table 7—3. Valid Conditional Statements

Verb	Required Phrase
ADD	SIZE ERROR
CALL	OVERFLOW
COMPUTE	SIZE ERROR
DELETE	INVALID KEY
DIVIDE	SIZE ERROR
IF	
MULTIPLY	SIZE ERROR
READ	AT END or INVALID KEY
RECEIVE	NO DATA
RETURN	AT END
REWRITE	INVALID KEY
SEARCH	
START	INVALID KEY
STRING	OVERFLOW
SUBTRACT	SIZE ERROR
UNSTRING	OVERFLOW
WRITE	INVALID KEY or END-OF-PAGE

7.3.2. Compiler Directing Statements

Compiler directing statements direct the compiler to take specific actions during compilation. They consist of a compiler directing verb (COPY or USE) and its operands. For example:

```
COPY MASTR-RECORD OF POLICY-LIB
```

This is a compiler directing statement. It directs the compiler to insert source code (MASTR-RECORD) taken from an external library named POLICY-LIB.

7.3.3. Imperative Statements

Statements that are not conditional or compiler directing are imperative statements. They indicate specific unconditional actions to be taken by the object program.

Table 7–4 lists the COBOL verbs used for constructing valid imperative statements and shows the verb/phrase combinations that are invalid. The phrases shown must be excluded from the statement if the associated verb is also in the statement.

Table 7—4. Valid Imperative Statements

Verb	Excluded Phrase
ACCEPT	
ADD	SIZE ERROR
ALTER	
CALL	ON OVERFLOW
CANCEL	
CLOSE	
COMPUTE	SIZE ERROR
DELETE	INVALID KEY
DISABLE	
DISPLAY	
DIVIDE	SIZE ERROR
ENABLE	
EXIT	
GO	
INSPECT	
MERGE	
MOVE	
MULTIPLY	SIZE ERROR
OPEN	
PERFORM	
READ	AT END or INVALID KEY
RECEIVE	NO DATA
RELEASE	
REWRITE	INVALID KEY
SEND	
SET	
SORT	
START	INVALID KEY
STOP	
STRING	ON OVERFLOW
SUBTRACT	SIZE ERROR
UNSTRING	ON OVERFLOW
WRITE	INVALID KEY or END-OF-PAGE

An imperative statement may consist of a sequence of imperative statements:

```
IF INDICATOR = ''YES''
     ADD 1 TO TOTAL
     MOVE ''NO'' TO INDICATOR
     GO TO MAIN.
```

The last three lines of code are considered one imperative statement.

7.4. INPUT-OUTPUT VERBS

FD entries in the data division describe the input-output files and the SELECT statement in the environment division associates the files with external devices. In the procedure division, input-output verbs control the flow of data between main storage and the devices; that is, to read data from, or put data into, the files.

The input-output verbs described in 7.4.1 through 7.4.10 are OPEN, CLOSE, READ, WRITE, REWRITE, DELETE, START, DISPLAY, ACCEPT and STOP (literal). Other input-output verbs are discussed in Section 11: DISABLE (11.8), ENABLE (11.9), RECEIVE (11.10), and SEND (11.11).

7.4.1. OPEN Statement

OPEN statements prepare files for processing. The preparation needed (checking labels, ensuring physical presence of files, etc.) varies from implementor to implementor. What is important is that you must open a file before you use input-output verbs such as READ, WRITE, and REWRITE to access it.

OPEN statements also allow you to access the file's record areas (the areas defined by the 01-level records that follow FD entries). Note that making a file record area available does not release or obtain its first data record – to do that you need an input-output verb such as READ.

The format for the OPEN statement is:

```
OPEN {INPUT file-name-1 [REVERSED          ] [,file-name-2 [REVERSED          ]] ... } ...
     {                  [WITH NO REWIND    ]               [WITH NO REWIND    ]]    }
     {OUTPUT file-name-3 [WITH NO REWIND] [,file-name-4 [WITH NO REWIND]] ...        }
     {I-O file-name-5 [,file-name-6] ...                                             }
     {EXTEND file-name-7 [,file-name-8] ...                                          }
```

When you open files, you must specify whether they are used as input, output, I-O, or extend.

Opening files as I-O allows you to access or replace existing records on mass storage devices. I-O files act as input files when you read records and output files when you store the updated records (using the REWRITE verb) in their original record positions in the files.

In the following coding sequence DISK-FILE is opened as I-O, and the first record is read and moved to NEW-RECD. During processing, appropriate changes are made to NEW-RECD, and the updated record is written to the same location on the disk as the record just read.

```
OPEN I-O DISK-FILE.
READ DISK-FILE INTO NEW-RECD AT END GO TO EOJ.
      .
      .}processing
      .
REWRITE DISK-FILE FROM NEW-RECD.
```

The file descriptions for files you open as input or I-O must be the same as when the files were created.

When you use the OUTPUT option, successful completion of an OPEN statement prepares for new file creation. At that point, the file contains no data records. You move records into the file using input-output verbs such as WRITE.

Normally, OPEN statements position files at their beginning. Options available for tape files, however, allow you to position files at their end.

The EXTEND option is for adding records at the end of existing sequential files:

```
OPEN EXTEND MASTER-FILE
```

This positions MASTER-FILE immediately following its last record. Then, use WRITE statements to add records as if the file was opened in output mode. Do not extend a file that is on a tape containing more than one file.

You can read sequential files backwards (that is, beginning with the last record and ending with the first) by opening them reversed. You cannot use the REVERSED option for files contained on more than one reel or unit, or on devices for which the term rewinding is not applicable. (The specific definition of a reel or unit in your environment is given by the implementor.) If you code the following

```
OPEN INPUT MASTER REVERSED
```

the first READ statement for MASTER retrieves the last record in the file, and subsequent READ statements obtain the file's records in reverse order.

The NO REWIND option also is valid only for tape files. You use it to open files already positioned at their beginning, perhaps because the preceding file on the same reel has just been closed with NO REWIND option. When you code:

```
OPEN INPUT MASTER WITH NO REWIND
```

the OPEN statement does not reposition the file (MASTER).

With the OPEN statement, you can open as many files as you want regardless of their use, organization, or access. So, if your program uses five files you might code

```
OPEN INPUT FILE-1 FILE-2
     OUTPUT FILE-3 FILE-4
     I-O FILE-5.
```

7.4.2. CLOSE Statement

When you want to access files, you open them; similarly, when you finish using files, you must close them. CLOSE statements initiate closing operations specified by the implementor, including any applicable ending label processing. The format is:

```
CLOSE file-name-1 [ {REEL} [WITH NO REWIND]  ]
                  [ {UNIT} [FOR REMOVAL     ]  ]
                  [ WITH {NO REWIND}           ]
                  [      {LOCK     }           ]
                  [ ,file-name-2 [ {REEL} [WITH NO REWIND]  ] ]
                                 [ {UNIT} [FOR REMOVAL     ]  ]
                                 [ WITH {NO REWIND}           ]
                                 [      {LOCK     }           ]
                                                                ...
```

As in the OPEN statement, one CLOSE statement may close more than one file. Unlike the OPEN statement, you do not have to specify whether the files are input, output, I-O, or extend.

Once you close a file, you may not use any of the input-output verbs that access it without first using an OPEN statement to reopen it. You may open and close a file as often as you like. Just be certain you only access files that are in open mode and that all files are in close mode when program execution ends.

If you close a file and you want to be certain that you do not inadvertently reopen it during the same run unit, you close it WITH LOCK:

```
CLOSE MASTER WITH LOCK
```

When this is executed, the statement OPEN INPUT MASTER is invalid. Generally, statements like either CLOSE THE-FILE or CLOSE THE-FILE WITH LOCK are sufficient for closing files; however, when the files are on reel or unit devices, special options allow you to control reel switching and rewinding.

The CLOSE or CLOSE WITH LOCK statement, as part of closing operations, normally repositions files at their beginning. If you want the file to remain in its current position, use the CLOSE verb with the NO REWIND option.

When you process sequential files contained on more than one reel or unit (tape files as an example), the implementor makes certain that when one reel ends, the next record is read from, or written to, another reel. This switch to the next reel is automatic (you don't have to include extra coding). You can use the REEL option (REEL and UNIT are equivalent) of the CLOSE statement to control the switching yourself.

For instance, suppose you are reading a file called MASTER that is contained on two reels. Before completing processing of the first reel, you want to skip to the second reel. If you code

```
CLOSE MASTER REEL
```

the reel swap procedures are executed and the next statement

```
READ MASTER
```

retrieves the first record on the second reel. The CLOSE statement with the REEL option does not close a file; MASTER remains in open mode, but the first reel of MASTER is in effect closed (it is repositioned at its beginning and cannot be accessed again unless the file is closed and then reopened).

If you don't plan to reaccess the first reel, you can add FOR REMOVAL to the CLOSE REEL statement:

```
CLOSE MASTER REEL FOR REMOVAL.
```

This logically removes the reel from the run unit, thus freeing a tape drive for use in other run units.

If you close a reel WITH NO REWIND, it is not repositioned.

Remember, when you have finished processing the last reel of a multireel file, you must close the file using a CLOSE statement without the REEL option.

7.4.3. READ Statement

READ statements retrieve records from files. For sequential access, the READ statement retrieves the next record in sequence. For random access, the READ statement retrieves a record specified by the value of a key field. The format is:

```
READ file-name [NEXT] RECORD [INTO identifier]
  [; AT END imperative-statement]
  [; KEY IS data-name]
  [; INVALID KEY imperative-statement]
```

Successful execution of a READ statement releases a record into the file's record area. The INTO option, provided all the records are the same length, moves the record to a work area, usually in working-storage. If you define MASTER as follows

```
FD  MASTER-FILE
    LABEL RECORDS ARE OMITTED.
    01   MASTER-RECD    PIC X(100).
```

And an area in working-storage as

```
01  MASTER-WORK       PIC X(100).
```

The statement

```
READ MASTER-FILE INTO MASTER-WORK
```

puts the record in both the record area (MASTER-RECD) and the work area (MASTER-WORK) just as if you coded

```
READ MASTER-FILE.
MOVE MASTER-RECD TO MASTER-WORK.
```

The record is moved from the record area to the work area according to the rules for the MOVE statement (7.6.1) without the CORRESPONDING phrase. If execution of a READ statement is unsuccessful, the move does not occur and the contents of the record area are undefined.

When you access files sequentially, you must provide a path of control for the program to take when you execute a READ statement and no records remain in the file. You may use the AT END option to specify that path. For example, execute

```
READ MASTER-FILE INTO MASTER-WORK
     AT END GO TO EOJ
```

This will, when no next record exists in MASTER-FILE, transfer control to the paragraph labeled EOJ. Once the AT END condition is recognized, you cannot read the file unless you first close and then reopen it.

If you do not use the AT END option, you must provide, in the declaratives portion of the procedure division, a USE procedure (7.8.2) that is executed when input-output operations for the file are unsuccessful such as an attempt to sequentially read a file that is at its end.

When you read files randomly, records are retrieved according to the values of RELATIVE KEY fields associated with relative files, or RECORD KEY values associated with indexed files. You must use the INVALID KEY option of the READ statement to provide a path of control for the program to take when you execute a READ statement and the value of the applicable key does not match any record in the file. For example, the relative key for a relative file is defined as follows:

```
77  REL-KEY   PIC 9(5).
```

When

```
READ MASTER-FILE INTO MASTER-WORK
     INVALID KEY GO TO NO-RECD-FOUND
```

is executed, no record in MASTER-FILE has a key value that matches the value of REL-KEY, control passes to the paragraph labeled NO-RECD-FOUND.

The dynamic access method described in 5.4 allows you to alternate between sequential and random processing of relative or indexed files. When you want to read randomly, you specify the desired record by placing a value in the relative key data item (for a relative fiel) or in the record key data item (for an indexed file) and then issue the READ statement with the INVALID KEY option (unless there is an applicable USE procedure). For example, if the value of the relative key associated with MASTER is 5, then

```
READ MASTER INVALID KEY PERFORM ERROR-PARA
```

retrieves the fifth record in the file. If the next READ statement executed for the file is coded with the NEXT option,

```
READ MASTER NEXT AT END GO TO EOF-MASTER
```

the value of the relative key automatically is incremented by 1 and the next record in sequence is retrieved (in this case, the sixth record in the file).

You use the KEY IS phrase for random access of indexed files. The data-name you used in the KEY IS phrase of the READ statement may be a RECORD KEY or an ALTERNATE RECORD KEY. If you code

```
ALTERNATE RECORD KEY IS POLICY-NO
```

in the FILE-CONTROL paragraph in the environment division, the code

```
READ MASTER KEY IS POLICY-NO
     INVALID KEY GO TO ERROR-PARA
```

retrieves the record whose POLICY-NO field equals the current value of POLICY-NO. When you don't use the KEY IS phrase, records are retrieved from indexed files based on the current value of the record key.

7.4.4. WRITE Statement

Write statements put records into output or I-O files. Format:

```
WRITE record-name[FROM identifier][; INVALID KEY imperative-statement]
      [{BEFORE} ADVANCING{{identifier-2}[LINE ]}]
       {AFTER }          {{integer   }[LINES]}
                         {{mnemonic-name}    }
                         {{PAGE          }    }
      [;AT{END-OF-PAGE}imperative-statement]
          {EOP        }
```

The FROM option for the WRITE statement is similar to the INTO option for the READ statement. While the INTO option moves records from record areas to work areas, the FROM option moves records from work areas to record areas. If you define an output file as

```
FD  MASTER-OUT
    LABEL RECORDS ARE OMITTED.
01  MASTR-RECD   PIC X(90).
```

And a work area as

```
01  WORK-OUTPUT   PIC X(90).
```

The statement

```
WRITE MASTR-RECD FROM WORK-OUTPUT
```

has the same effect as

```
MOVE WORK-OUTPUT TO MASTR-RECD
WRITE MASTR-RECD
```

After successful execution of a WRITE statement, the record is no longer available in the record area (unless the file is named in a SAME RECORD AREA clause). But, if you use the FROM option, the record remains available in the work area.

For printer-destined files, WRITE statements control the vertical positioning of lines. You can write a line before or after advancing either to a certain number of lines, to a specific spot on a page, or to a new page. If you use the BEFORE option, the line is written before the output device is repositioned; if you use the AFTER option, the line is written after the device is repositioned. If you omit the ADVANCING phrase, the WRITE statement is executed as if you coded ADVANCING 1 LINE.

In the format, integer or identifier-2 (which must represent an integer) specifies the number of lines the device is to be repositioned. It may be zero.

You define mnemonic-name in the SPECIAL-NAMES paragraph (5.3) of the environment division. It is associated with a particular feature specified by the implementor.

You use the PAGE option to write records before or after the device is repositioned to a new page. A LINAGE clause (6.1.6), if present, specifies where the next page begins. If you do not use the LINAGE clause, the implementor defines where the next page begins.

The END OF PAGE (EOP is equivalent) option is valid only if a LINAGE clause is associated with the file. It specifies an imperative statement executed at the end of a page.

When you write records to relative files, and the access is sequential, the first record is assigned relative value 1, the second record relative value 2, etc. There is no need to specify a relative key, but if you do, it's updated appropriately as WRITE statements are executed.

When the access mode is random or dynamic, relative file records are associated with relative key values that specify their ordinal position in the file. If

```
77  REL-KEY     PIC 9(4).
```

is the relative key associated with

```
FD  MSTR-OUT
    LABEL RECORDS ARE STANDARD.
    01  MSTR-RECD     PIC X(90).
```

you code

```
MOVE 12 TO REL-KEY.
WRITE MSTR-RECD INVALID KEY GO TO ERROR-PARA.
```

This stores the record in the twelfth ordinal position in the file.

If you do not provide an appropriate USE procedure, the INVALID KEY option is required on a WRITE statement referencing a relative file. In the absence of the USE procedure, the INVALID KEY option indicates what should be done when a WRITE statement is unsuccessful.

In this situation, the statement GO TO ERROR-PARA is executed. If you use the FILE STATUS clause in the environment division, the status key values identify the specific cause of the INVALID KEY condition.

For indexed files, when the access is sequential, you must write records in the ascending order of their RECORD KEY values. Remember that unlike a relative key, a record key is a field within the record. You must make sure the key field value in each new record is greater than the key field value in the previous record.

When the access for an indexed file is random or dynamic, you can write the records in any order. Again, you must use the INVALID KEY option to specify the action taken if a WRITE statement is unsuccessful, and again the status key values associated with the file indicate the cause of unsuccessful WRITE statements. If you code RECORD KEY IS POL-NO in the environment division, and define the corresponding indexed file in the data division as

```
FD  MASTER-OUT
    LABEL RECORDS ARE STANDARD.
    01  MASTER-RECD.
        02  POL-NO     PIC 9(6).
        02  FILLER     PIC X(144).
```

Then,

```
WRITE MASTER-RECD INVALID KEY GO TO ERRORS
```

writes a record with a record key equal to the current value of POL-NO.

In addition to putting records into new files, WRITE statements add records to existing sequential files opened in EXTEND mode, and existing relative and indexed files opened in I-O mode. Be certain the records you add to files opened as I-O have unique relative key or record key values. If you want to replace existing records in files opened as I-O, use the REWRITE statement (7.4.5).

7.4.5. REWRITE Statement

While WRITE statements allow you to put new records into files, REWRITE statements allow you to replace records already existing in files. The format shows that the REWRITE statement is similar to the WRITE statement. It has the FROM option to permit writing from a work area and the INVALID KEY option to pass control to an imperative statement when it is unsuccessful:

```
REWRITE record-name[FROM identifier][;INVALID KEY imperative-statement]
```

You can rewrite records only to mass storage files opened in I-O mode. If the access is sequential, you must read the record to be replaced, then rewrite the new record without executing any other input-output statements for the file between the READ and REWRITE statements. If the indexed file is opened as I-O:

```
FD  PAYROLL
    LABEL RECORDS ARE STANDARD.
    01   PAY-RECD    PIC X(100).
```

You define a work area as follows:

```
01  WORK-PAY.
    02  A    PIC  X(25).
    02  B    PIC  X(25).
    02  C    PIC  X(25).
    02  D    PIC  X(25).
```

Then you code

```
READ PAYROLL INTO WORK-PAY AT END GO TO EOJ
```

to obtain the record with the current RECORD KEY value, then you make changes in WORK-PAY.

Finally, you code

```
REWRITE PAY-RECD FROM WORK-PAY
     INVALID KEY GO TO ERROR
```

to replace the record in the file. The new record and the record being replaced must have the same number of characters.

The INVALID KEY phrase is required for indexed files in any access mode, and for relative files accessed randomly or dynamically, if no associated USE procedure is specified. The INVALID KEY phrase must not be specified for relative files accessed sequentially and for sequential files.

If the access is random or dynamic, you do not necessarily have to read a specific record before you use the REWRITE statement to replace it. The REWRITE statement replaces the record with the key value equal to the current value of the relative key or record key associated with the file.

7.4.6. DELETE Statement

DELETE statements are similar to REWRITE statements; the difference is they logically delete rather than replace records on mass storage files. The format:

```
DELETE file-name RECORD[;INVALID KEY imperative-statement]
```

The file affected must be open in I-O mode, and its organization must be relative or indexed.

As with the REWRITE statement, if the access is sequential, you must first read the file, and then (without executing any intervening input-output statements that reference the file) use the DELETE statement to remove the record just read. You must not use the INVALID KEY phrase when you access files sequentially.

If the access is random or dynamic, the DELETE statement removes the record with the key value equal to the current value of the relative key or record key associated with the file. You must use the INVALID KEY phrase (again, unless you specify an applicable USE procedure). If you execute a DELETE statement and the file does not contain the record specified by the key, the imperative statement associated with the INVALID KEY phrase is executed.

If the relative key associated with the file MASTER is called REL-KEY, the following statements delete relative record number 215 from the file. If MASTER does not have a relative record number 215, PERFORM ERRORS is executed.

```
MOVE 215 TO REL-KEY.
DELETE MASTER INVALID KEY PERFORM ERRORS.
```

7.4.7. START Statement

The START statement is used to position relative or indexed files somewhere other than at their beginning. The OPEN statement positions files so that the first read accesses relative record number 1 (if the file has relative organization) or the record with the lowest RECORD KEY value (if the file has indexed organization). When you want to read a file sequentially and do not want to begin with the file's first record, use the START statement to reposition the file. The START statement does not access a file; it determines which record is retrieved by the next READ statement.

For example, if you want to begin reading a relative file called INVEN-FILE at record 25, and RELATIVE KEY is REL-KEY you code

```
OPEN INPUT INVEN-FILE.
MOVE 25 TO REL-KEY.
START INVEN-FILE.
READ INVEN-FILE AT END GO TO EOJ.
```

When RECORD KEY is EMPL-NO and you want to begin reading an indexed file called PAYROLL at the record whose key value is 413697, you code

```
OPEN INPUT PAYROLL.
MOVE 413697 TO EMPL-NO.
START PAYROLL.
READ PAYROLL AT END GO TO EOJ.
```

The format for the START statement is:

```
START file-name [KEY { IS EQUAL TO      } data-name]
                     { IS =             }
                     { IS GREATER THAN  }
                     { IS >             }
                     { IS NOT LESS THAN }
                     { IS NOT <         }
  [; INVALID KEY imperative-statement]
```

The file must be open in input or I-O mode when the START statement is executed. The access must be sequential or dynamic. Data-name, if used, must be the relative key associated with relative files, or one of the record keys associated with indexed files. Thus, the key of the first record you read is equal to, greater than, or not less than the current value of the relative key or one of the record keys.

If you do not use the KEY phrase, as in our examples, the relational operator "IS EQUAL TO" is assumed.

You must use the KEY IS phrase to sequentially read an indexed file based on an alternate record key. If you code

```
ALTERNATE RECORD KEY IS PHONE-NO
```

in the FILE-CONTROL paragraph, and you want to begin reading MASTER at the record with PHONE-NO equal to 2314, code

```
OPEN INPUT MASTER.
MOVE 2314 TO PHONE-NO.
START MASTER KEY IS EQUAL TO PHONE-NO.
READ MASTER AT END GO TO EOJ.
```

For indexed file, the data-name in the KEY phrase of a START statement may be a data item that is subordinate to a record key, provided it has the same leftmost character as the record key. For example, if

```
RECORD KEY IS EMPL-NO
```

and the record is defined in the FD as

```
01   PAY-RECD.
     02  EMPL-NO.
         03  EMPL-CLASS     PIC X(4).
         03  EMPL-CODE      PIC X(6).
     02  FILLER      PIC X(90).
```

Then the following is valid:

```
START PAYROLL KEY GREATER THAN EMPL-CLASS
```

7.4.8. DISPLAY Statement

DISPLAY statements move small amounts of data to hardware devices such as printers, terminals, or console devices. The format:

```
DISPLAY {identifier-1} [, identifier-2] ... [UPON mnemonic-name]
        {literal-1   }  [  literal-1   ]
```

The output of a DISPLAY statement is not associated with a file. You name the specific data you want to appear on the device. One common use of the DISPLAY statement is to send a message to the computer operator. For instance, if the run unit processes the same tape twice, the code

```
DISPLAY ''TAPE 253 RUNS TWICE'' UPON CONSOLE
```

alerts the operator not to remove the tape the first time it rewinds.

Literals in the format can be nonnumeric literals, unsigned integers, or any figurative constant except ALL.

The implementor specifies what hardware devices receive data and how much data each receives. The implementor also specifies, in the SPECIAL-NAMES paragraph in the environment division, the names used to associate mnemonic-names with hardware devices. If the implementor's name for a specific terminal is TXR-343, and you code

```
TXR-343 IS TERM-1
```

in the SPECIAL-NAMES paragraph, then

```
DISPLAY ''END OF PROGRAM'' UPON TERM-1
```

makes END OF PROGRAM appear on that terminal.

If you use the DISPLAY statement without the UPON phrase, the data appears on the implementor's standard display device.

7.4.9. ACCEPT Statement

ACCEPT statements retrieve small amounts of data from specified hardware devices. They also provide a way for you to retrieve the system date and time. Two formats are available:

Format 1:

```
ACCEPT identifier [FROM mnemonic-name]
```

Format 2:

```
ACCEPT identifier-FROM { DATE }
                       { DAY  }
                       { TIME }
```

You use the SPECIAL-NAMES paragraph in the environment division to associate a mnemonic-name with a device. As with the DISPLAY statement, the implementor defines the devices that send data and the amount of data each may send.

It's common to use ACCEPT statements to receive data from the computer operator. For instance, you might code

```
77  OPR-REPLY    PIC X.
```

in the data division, and the following sequence of instructions in the procedure division:

```
DISPLAY ''KEY IN RUN CYCLE, W FOR WEEKLY, M FOR MONTHLY'' UPON CONSOLE.
ACCEPT OPR-REPLY FROM CONSOLE.
IF OPR-REPLY = ''W'' GO TO WEEKLY.
IF OPR-REPLY = ''M'' GO TO MONTHLY.
```

If you use the ACCEPT statement without the FROM phrase, data is received from the implementor's standard device.

Format 2 allows you to accept from the system date, day, and time.

- DATE consists of six digits that express the date in year, month, day order. Thus, July 1, 1968 is 680701.
- DAY consists of five digits (a two-digit year and a three-digit ordinal day). July 1, 1968 is expressed as 68183.
- TIME consists of eight digits representing the hours, minutes, seconds, hundredths of seconds past midnight. For instance, midnight is expressed as 00000000. Thus, 2:41 p.m. is 14410000 and the maximum value is 23595999 (one-hundredth of a second before midnight).

To receive DATE, DAY, and TIME you may code the following in the data division

```
77  THE-DATE     PIC 9(6).
77  THE-DAY      PIC 9(5).
77  THE-TIME     PIC 9(8).
```

and the following in the procedure division

```
ACCEPT THE-DATE FROM DATE.
ACCEPT THE-DAY FROM DAY.
ACCEPT THE-TIME FROM TIME.
```

7.4.10. STOP Statement

You use STOP statements to temporarily or permanently stop execution of the object program. The format is:

```
STOP { RUN     }
     { literal }
```

The STOP RUN option permanently stops execution of the object program. If you use it in a series of imperative statements, it should be the last statement:

```
IF:COUNT = 50
     MOVE ''E'' TO IND
     WRITE FINAL-LINE
     STOP RUN.
```

The STOP literal option works like a DISPLAY statement, except it requires a response by the operator, thus temporarily suspending processing.

The literal, as with the DISPLAY statement, may be a nonnumeric literal, an unsigned integer, or any figurative constant except ALL. Generally, the literal gives the operator some information or requests some action from the operator. For example:

```
STOP ''PLEASE MOUNT MONTHLY REPORT FORM ON PRINTER'' UPON CONSOLE.
WRITE ''MONTHLY REPORT'' AFTER ADVANCING PAGE.
```

The STOP statement displays a message requesting the operator to mount a preprinted form on the printer. When the operator resumes execution, the next statement, which is the WRITE statement, is executed.

7.5. ARITHMETIC VERBS

The arithmetic verbs (ADD, SUBTRCT, MULTIPLY, DIVIDE, and COMPUTE) allow you to perform basic calculations on data. Operands for the calculations may be numeric literals or elementary data items defined in the data division as numeric. You may not use numeric edited items as operands in calculations; however, data items that store results, such as those following the word GIVING in arithmetic statements or preceding the equal sign in COMPUTE statements, may be numeric edited.

The data descriptions of operands do not have to be the same; any adjustments needed are made for you. For example, if you add two lines

```
05  A  PIC  999.99.
05  B  PIC  99.9  COMP.
```

A is converted to COMP usage and decimal point alignment is supplied.

No operand may consist of more than 18 digits. In addition, the composite of operands (the hypothetical data item resulting from superimposing all a statement's operands on their decimal points) must not be more than 18 digits. For example, the composite of operands of 2.15 and 37.6 (39.75) has four digit positions (3; 2 and 7; 1 and 6; and 5).

7.5.1. ROUNDED Phrase

The ROUNDED phrase rounds off the results of arithmetic statements. When you specify rounding, the last fractional digit is increased by 1 if the excess digit is 5 or greater.

Of course, results are rounded only when the fraction requires more decimal places than you specified in the PICTURE clause of the resultant data item. If the result of a calculation is 6.38 and is to be stored in a data item whose PICTURE clause is

```
PIC   999.9
```

It is stored as follows if rounding is specified:

```
006.4
```

But if the PICTURE clause is

```
PIC   999.99
```

it is stored as

```
ØØ6.38
```

If the PICTURE clause is

```
PIC   999.9
```

and you do not specify the ROUNDED phrase, the last digit is truncated. Thus, in the example, the result is stored as

```
ØØ6.3
```

7.5.2. SIZE ERROR Phrase

When the result of a calculation exceeds the largest value that can be contained in the resultant data item, a size error condition exists. If you use the SIZE ERROR phrase following an arithmetic statement, the object program won't store results in areas that are too small. Thus, if the data item you designate for storing a result is defined as

```
A   PIC   99.
```

and the result is 347, there is a size error. If you used the SIZE ERROR phrase, the value of A is not changed, and the imperative statement associated with the SIZE ERROR phrase is executed.

If a calculation has more than one resultant data item, arithmetic operations are completed for results that can be stored, but not for results that cause a size error condition. Assume your working-storage section includes:

```
77   A   PIC 999 COMP.
77   B   PIC 999 COMP.
77   C   PIC 999.
77   D   PIC 9999.
```

You want to add A to B and store the result in both C and D. Also assume A = 800, B = 300, C = 0, and D = 0. If you code

```
ADD A, B GIVING C, D
      ON SIZE ERROR PERFORM TOO-BIG.
```

the result (1100) is too big to be stored in C, but may be stored in D. Since you used the SIZE ERROR phrase, C is unaffected (it remains equal to 0) and the operation is completed with respect to D (it equals 1100). Then, the imperative statement PERFORM TOO-BIG is executed. If no SIZE ERROR phrase is specified, D still equals 1100, but the value of C is unpredictable. Also, division by zero can cause the SIZE ERROR condition.

7.5.3. ADD Statement

You use ADD statements to sum two or more operands and store the results. There are three formats:

Format 1:

```
ADD {identifier-1} [, identifier-2] ... TO identifier-m [ROUNDED]
    {literal-1   } [, literal-2   ]
  [, identifier-n [ROUNDED]] ... [; ON SIZE ERROR imperative-statement]
```

Format 2:

```
ADD {identifier-1} , {identifier-2} , [identifier-3] ...
    {literal-1   }   {literal-2   }   [literal-3   ]
  GIVING identifer-m [ROUNDED] [, identifier-n [ROUNDED]] ...
  [; ON SIZE ERROR imperative-statement]
```

Format 3:

```
ADD {CORRESPONDING} identifier-1 TO identifer-2 [ROUNDED]
    {CORR          }
  [; ON SIZE ERROR imperative-statement]
```

Formats 1 and 2 work almost the same way; the difference is that the result of a Format 1 ADD is added to the current value of identifier-m, but the result of a Format 2 ADD replaces the current value of identifier-m.

If A, B, and C all equal 5, and you code

```
ADD A B TO C
```

the sum of A and B (10) is added to the value of C (5). Thus, the value 15 is stored in C.

But if you code

```
ADD A B GIVING C
```

the sum of A and B (10) replaces the value of C (5). Thus, the value 10 is stored in C.

Format 2 requires at least two operands to the left of GIVING. Thus,

```
ADD 5 GIVING TOTAL
```

is invalid (and illogical). But you can set TOTAL to 5 by coding

```
MOVE 5 TO TOTAL
```

You can, however, use Format 1 and code

```
ADD 5 TO TOTAL
```

This increases the current value of TOTAL by 5.

You can use ADD statements to sum more than two operands, and you can store the result in more than one identifier:

```
ADD 3 4 5 6 TO SUBTOT FINTOT
```

This adds 18 to the current value of SUBTOT and to the current value of FINTOT.

The coding

```
ADD 3 4 5 6 GIVING SUBTOT FINTOT
```

replaces the values of SUBTOT and FINTOT with 18.

In Format 3, identifier-1 and identifier-2 refer to group items. The ADD CORRESPONDING (CORR is equivalent) statement adds the values of elementary numeric items subordinate to identifier-1 to similar items of identifier-2 that have the same name (excluding FILLER items). If you code the following in the data division

```
01  SUBTOT.
    02  A      PIC 9(5)    COMP.
    02  C      PIC 9(5)    COMP.
    02  FILLER  PIC X(5).
    02  E      PIC  9(5)   COMP.
01  FINTOT.
    02  A      PIC 9(5)    COMP.
    02  B      PIC 9(5)    COMP.
    02  C      PIC 9(5)    COMP.
    02  FILLER  PIC X(5).
```

the statement

```
ADD CORRESPONDING SUBTOT TO FINTOT
```

adds the current value of A of SUBTOT to A of FINTOT, and C of SUBTOT to C of FINTOT. But E of SUBTOT is not affected because there is no E in FINTOT. Similarly, B of FINTOT is not affected because there is no B in SUBTOT.

You can use the ROUNDED phrase (7.5.1) following selected results and the SIZE ERROR phrase (7.5.2) following selected statements. When you round one result in a statement, you do not necessarily have to round all the results. For example, the following is a valid statement:

```
ADD A B C TO SUBTOT ROUNDED FINTOT
     ON SIZE ERROR GO TO TOO-BIG
```

The sum of A, B, and C is added to SUBTOT, and the result is rounded. The same sum is added to FINTOT, but any decimal places not specified in the PICTURE clause are truncated rather than rounded. If the result is too big to be stored in SUBTOT or FINTOT, control is transferred to the paragraph labeled TOO-BIG.

7.5.4. SUBTRACT Statement

You use SUBTRACT statements to perform subtraction operations and store the results. As with the ADD statement, there are three formats:

Format 1:

```
SUBTRACT {identifier-1} [, identifier-2] ... FROM identifer-m [ROUNDED]
         {literal-1   }  [, literal-2   ]
  [, identifier-n[ROUNDED]] ... [; ON SIZE ERROR imperative-statement]
```

Format 2:

```
SUBTRACT {identifer-1} [, identifer-2] ... FROM {identifer-m} ...
         {literal-1   }  [, literal-2   ]      {literal-m   }
  GIVING identifer-n[ROUNDED][,identifer-o[ROUNDED]] ...
  [; ON SIZE ERROR imperative-statement]
```

Format 3:

```
SUBTRACT {CORRESPONDING} identifer-1 FROM identifier-2[ROUNDED]
         {CORR         }
  [; ON SIZE ERROR imperative-statement]
```

The results in Format 1 of the subtractions are stored in the data items from which the operands are subtracted. The operands preceding FROM are added, and the sum is subtracted from the current value of identifier-m.

In Format 2, the results are stored in separate data items. Again, the operands preceding FROM are added, and the result is subtracted from identifier-m (or literal-m), but the result is stored in identifier-n and the value of identifier-m is unchanged.

Assume A = 5, B = 5, and C = 15:

```
SUBTRACT A B FROM C
```

This subtracts 10 from 15 and stores the result (5) in C. Executing the following stores the result (5) in D and leaves C equal to 15.

```
SUBTRACT A B FROM C GIVING D
```

The Format 3 SUBTRACT statement is similar to the Format 3 ADD statement; it subtracts items subordinate to identifier-1 from items with the same name in identifier-2, storing the result in the items subordinate to identifier-2. If you code the following in the data division

```
01  TOTALS.
    02  A  PIC 9(7)  COMP.
    02  B  PIC 9(7)  COMP.
01  TOTALS-2.
    02  A  PIC 9(7)  COMP.
    02  B  PIC 9(7)  COMP.
```

the statement

```
SUBTRACT CORR TOTALS FROM TOTALS-2
```

subtracts A of TOTALS from A of TOTALS-2, storing the result in A of TOTALS-2, and subtracts B of TOTALS from B of TOTALS-2, storing the result in B of TOTALS-2.

7.5.5. MULTIPLY Statement

MULTIPLY statements multiply data items and store the results. Two formats are applicable:

Format 1:

```
MULTIPLY {identifier-1} BY identifier-2[ROUNDED]
         {literal-1   }
  [, identifier-3[ROUNDED]] ... [; ON SIZE ERROR imperative-statement]
```

Format 2:

```
MULTIPLY {identifier-1} BY {identifier-2} GIVING identifier-4[ROUNDED]
         {literal-1   }    {literal-2   }
  [, identifier-5[ROUNDED]] ... [; ON SIZE ERROR imperative-statement]
```

The products can be stored in place of the multipliers or in separate data items. For example, if A = 3, B = 5, and C = 7, then

```
MULTIPLY A BY B C
```

stores the value 15 in B, and 21 in C; however,

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

stores the value 15 in both C and D and B remains equal to 5.

7.5.6. DIVIDE Statement

DIVIDE statements are used to divide one data item into others, and store the results including (at your option) any remainder. The five formats are:

Format 1:

```
DIVIDE {identifier-1} INTO identifier-2[ROUNDED]
       {literal-1   }
  [, identifier-3[ROUNDED]] ... [; ON SIZE ERROR imperative-statement]
```

Format 2:

```
DIVIDE {identifier-1} INTO {identifier-2} GIVING identifier-3[ROUNDED]
       {literal-1   }      {literal-2   }
  [, identifier-4[ROUNDED]] ... [; ON SIZE ERROR imperative-statement]
```

Format 3:

```
DIVIDE {identifier-1} BY {identifier-2} GIVING identifer-3[ROUNDED]
       {literal-1   }    {literal-2   }
  [, identifer-4[ROUNDED]] ... [; ON SIZE ERROR imperative-statement]
```

Format 4:

```
DIVIDE {identifier-1} INTO {identifier-2} GIVING identifier-3[ROUNDED]
       {literal-1   }      {literal-2   }
  REMAINDER identifier-4[; ON SIZE ERROR imperative-statement]
```

Format 5:

```
DIVIDE {identifier-1} BY {identifier-2} GIVING identifier-3[ROUNDED]
       {literal-1   }    {literal-2   }
  REMAINDER identifier-4[; ON SIZE ERROR imperative-statement]
```

Formats 1 and 2 divide the first operand into the second operand; Format 1 stores the quotient in the dividend and Format 2 stores it in a separate data item following GIVING. If A = 5 and B = 10

```
DIVIDE A INTO B
```

stores the result (2) in B, while

```
DIVIDE A INTO B GIVING C
```

stores the 2 in C, leaving B equal to 10.

Format 3 divides the second operand into the first operand, and stores the result in a separate data item. If A = 5 and B = 10

```
DIVIDE A BY B GIVING C
```

stores .5 in C. A remains equal to 5.

Formats 4 and 5 are identical to Formats 2 and 3, respectively, except they permit only one GIVING operand and provide for storing a remainder. You find the remainder of a divide operation by subtracting the product of the quotient and the divisor from the dividend. If you code

```
77  A   PIC 999 COMP.
77  B   PIC 999 COMP.
77  C   PIC 999 COMP.
77  REM  PIC 99.
```

in the data division, and A = 5, B = 9, and C = 0, execution of

```
DIVIDE A INTO B GIVING C REMAINDER REM
```

stores 1 in C and 4 in REM.

You may round a result and still store a remainder:

```
77  A   PIC  99  COMP.
77  B   PIC  99  COMP.
77  C   PIC  99  COMP.
77  REM  PIC  99.
```

This is coded in the data division, and A = 3, B = 11, and C = 0. The statement

```
DIVIDE B BY A GIVING C ROUNDED REMAINDER REM
```

rounds the result (3.666) to 4, and stores it in C; but, when calculating the remainder, truncates the result to 3 (just as if you omitted the ROUNDED phrase). Thus, the remainder is the product of the result (3) and the divisor (3) subtracted from the dividend (11). So, after execution of the statement, A = 3, B = 11, C = 4, and REM = 2.

If you use the SIZE ERROR phrase with Formats 4 and 5, and a size error occurs on the quotient, both identifier-3 and identifier-4 remain unchanged. If the size error occurs on the remainder, the quotient is stored in identifier-3, but identifier-4 remains unchanged.

7.5.7. COMPUTE Statement

You use COMPUTE statements to assign the values of arithmetic expressions to one or more data items. COMPUTE statements allow you to add, subtract, multiply, and divide in one statement, thus reducing the number of lines of code needed. The format:

```
COMPUTE identifier-1[ROUNDED][, identifier-2[ROUNDED]] ... =
  arithmetic-expression[; ON SIZE ERROR imperative-statement]
```

The identifiers that represent results may be elementary numeric or elementary numeric edited data items. If you code

```
77  A  PIC 99 COMP.
77  B  PIC 99 COMP.
77  C  PIC 99 COMP.
77  D  PIC 999.
77  E  PIC ZZ9.
```

in the data division, if A = 5, B = 6, and C = 15, then

```
COMPUTE D E = (A * B) - C
```

stores 015 in D and 15 in E.

Note that coding this statement using simple arithmetic verbs requires two statements:

```
MULTIPLY A BY B GIVING TEMP.
SUBTRACT C FROM TEMP GIVING D, E.
```

7.6. DATA MOVEMENT VERBS

The data movement verbs are MOVE, INSPECT, STRING, and UNSTRING. They move and manipulate data. Descriptions for each are included in 7.6.1 through 7.6.4.

7.6.1. MOVE Statement

MOVE statements transfer data from one data item to one or more others. Two formats are used:

Format 1:

```
MOVE {identifier-1} TO identifier-2 [, identifier-3] ...
     {literal     }
```

Format 2:

```
MOVE {CORRESPONDING} identifier-1 TO identifier-2
     {CORR         }
```

The identifier or literal to the left of TO is the sending item and the identifiers to the right of TO are receiving items. A successful MOVE makes the value of the receiving items equal to the value of the sending item, without affecting the value of the sending item. So, if A = 5, execution of the following coding makes A, B, and C all equal to 5.

```
MOVE A TO B, C.
```

As described in the discussion of the PICTURE clause (6.2.2), there are five types of elementary data items: alphabetic, alphanumeric, numeric, alphanumeric edited, and numeric edited. Table 7–5 lists the valid combinations of sending and receiving elementary items in MOVE statements. The figurative constant ZERO is category numeric, SPACE is category alphabetic, and the other figurative constants are alphanumeric.

Table 7—5. Permissible Combinations of Operands in MOVE Statements

Category of Sending Data Item		Category of Receiving Data Item		
		Alphabetic	Alphanumeric Edited Alphanumeric	Numeric Integer Numeric Noninteger Numeric Edited
Alphabetic		Yes	Yes	No
Alphanumeric		Yes	Yes	Yes
Alphanumeric Edited		Yes	Yes	No
Numeric	Integer	No	Yes	Yes
	Noninteger	No	No	Yes
Numeric Edited		No	Yes	No

Alphanumeric items are mostly moved to alphanumeric or alphanumeric edited items, and numeric items to numeric or numeric edited items. But, as shown in Table 7-5, other types of MOVE statements are permissible.

If the receiving item is described as alphanumeric or alphanumeric edited as in

```
A     PIC     X(5).
```

and the sending item is signed numeric

```
B     PIC     S9(5).
```

the sign is not moved. So, if B equals +43775

```
MOVE B TO A
```

makes A equal to 43775, with no sign, regardless of whether the sign of B is stored as part of another digit or as a separate character.

When the receiving item is a signed numeric item, and the sending item is an unsigned numeric item or an alphanumeric item, a positive sign is generated for the receiving item.

When the receiving item is unsigned numeric, the absolute value of the sending item is moved and any sign is dropped.

If either the sending item, the receiving item, or both are group items, the MOVE is treated as if it were an alphanumeric to alphanumeric elementary move and no consideration is given to the types of individual elementary items within the group.

Format 2 MOVE moves are similar to ADD or SUBTRACT actions that use the CORRESPONDING phrase. Identifier-1 and identifier-2 must be group items. You move items of one group to items of the other group that have the same name. If you code

```
01  GROUP-1.
    02  A  PIC  X(7).
    02  B  PIC  X(5).
    02  C  PIC  X(5).
01  GROUP-2.
    02  A  PIC  X(7).
    02  E  PIC  X(9).
    02  C  PIC  X(5).
```

in the data division, the statement

```
MOVE CORR GROUP-1 TO GROUP-2
```

is equivalent to

```
MOVE A OF GROUP-1 TO A OF GROUP-2.
MOVE C OF GROUP-1 TO C OF GROUP-2.
```

7.6.2. INSPECT Statement

INSPECT statements tally and replace occurrences of characters in a data item. There are three formats: the first tallies characters, the second replaces characters, and the third both tallies and replaces characters.

Format 1:

This format counts occurrences of specified characters in identifier-1 and stores the count in identifier-2. You can count all the characters in identifier-1 or just specific characters.

```
INSPECT THE-WORD TALLYING
    THE-COUNT FOR CHARACTERS
```

This tallies the number of characters in THE-WORD, and stores that number in THE-COUNT.

```
INSPECT THE-WORD TALLYING
    THE-COUNT FOR ALL ''R''
```

This coding tallies the number of R's in THE-WORD and stores that number in THE-COUNT. Thus, if THE-WORD equals TERRITORY, THE-COUNT equals 3.

The LEADING option is for counting contiguous occurrences of a specific character or characters at the beginning of a word. For instance, the number of leading R's in TERRITORY is 0, because the first letter is not an R. The number of leading T's in TERRITORY is 1. Although there is another T in TERRITORY, it is not contiguous (immediately adjacent) to the first T.

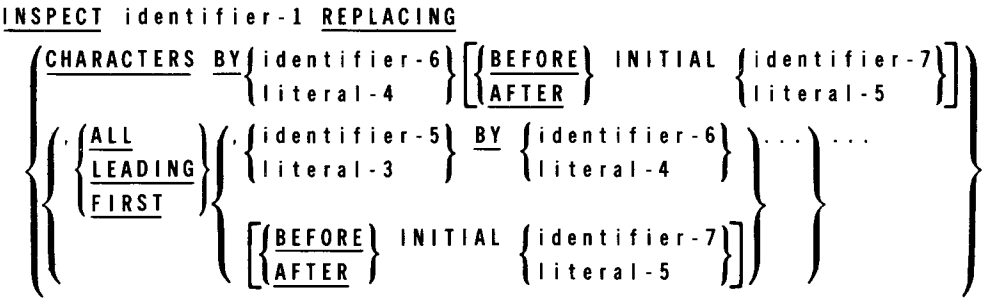
The BEFORE and AFTER options allow you to inspect portions of a word, such as all the characters before the first (INITIAL) L or after the first X. For instance, if you code

```
INSPECT  THE-WORD TALLYING
    THE-COUNT  FOR LEADING ''XLO'' AFTER INITIAL ''Q''
```

If THE-WORD equals XLOLQXLOXLOQR, THE-COUNT equals 2 (representing the two contiguous occurrences of XLO immediately following the first Q).

Format 2:

Note that in Format 2 there are two ways to use the REPLACING option.



The first method replaces all characters in identifier-1 with the character represented by identifier-6 or literal-4. Generally, you use the BEFORE or AFTER phrase with this option; otherwise, it's just like the MOVE statement.

Note that in the following coding the shorter second line has the same effect as the first.

```
INSPECT NAME REPLACING CHARACTERS BY SPACE
MOVE SPACES TO NAME
```

The next statement, however, is more common:

```
INSPECT NAME REPLACING
    CHARACTERS BY SPACE AFTER INITIAL ''''
```

It affects only the characters in NAME to the right of the first comma.

You also can replace specific characters in identifier-1. The ALL and LEADING options work the same way as in Format 1. The FIRST option refers to the first occurrence of a character in identifier-1. If A = QPLPLZ459, the coding

```
INSPECT A REPLACING FIRST ''PL'' BY ''XR''
```

changes A to QXRPLZ459. Again, you can use the BEFORE or AFTER phrase to indicate the inspection begins before or after the initial occurrence of a character or characters.

If the literal named in a BEFORE phrase does not occur at all in identifier-1, the inspection continues as if the BEFORE phrase was not in the statement. If there is no occurrence in identifier-1 of the literal named in the AFTER phrase, no changes are made in identifier-1. Thus, if A = XXXZYR, the following coding changes A to ZZZZYR.

```
INSPECT A REPLACING
    ALL ''X'' BY ''Z'' BEFORE INITIAL ''L''
```

Also, the following leaves A equal to XXXZYR:

```
INSPECT A REPLACING
    ALL ''X'' BY ''Z'' AFTER INITIAL ''L''
```

Some other examples:

```
INSPECT WORK REPLACING
    ALL ''A'' BY ''G'' BEFORE INITIAL ''X''
```

If WORD = ARXAX, it's changed to GRXAX.

If WORD = HANDAX, it's changed to HGNDGX.

```
INSPECT WORD REPLACING
    CHARACTERS BY ''B'' BEFORE INITIAL ''A''
```

If WORD = 12△XZABCD, it's changed to BBBBBABCD.

If WORD = 1439C, it's unchanged because there's no A in WORD.

```
INSPECT WORD REPLACING
    LEADING ''AAA'' BY ''BBB'' AFTER INITIAL ''F''
```

If WORD = AFAAAABBCD, it's changed to AFBBBABBCD.

If WORD = AAAFAAABB, it's changed to AAAFBBBBB.

More than one character or characters can be replaced in a word. For instance:

```
INSPECT A REPLACING ALL ''Q'' BY ''Z'', ''X'' BY ''R''
```

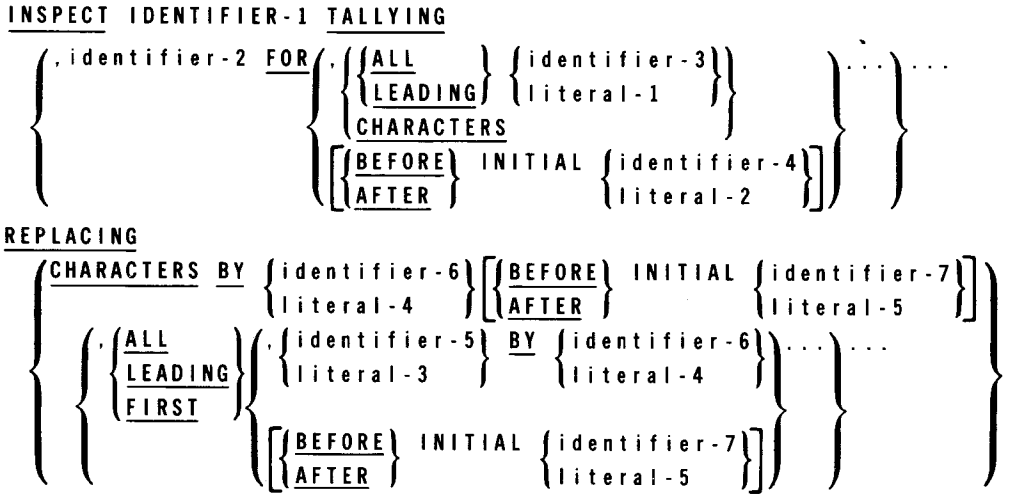
This changes A from LMQZXPQRX to LMZZRPZRR.

If you use the BEFORE or AFTER phrase when replacing more than one literal, the phrase applies only to one change:

```
INSPECT WORD REPLACING ALL ''X'' BY ''Y''
    ''B'' BY ''Z'', ''W'' BY ''Q'' AFTER INITIAL ''R''
```

In this example, the AFTER INITIAL "R" applies only to replacing "W" by "Q". Thus, if WORD = YZACDWBR, it is changed to YZACDWZR. If WORD = XMWXMBRXW, the change is to YMWYMZRYQ.

Format 3:



This format is simply a combination of Formats 1 and 2; you can both tally and replace characters in identifier-1. Here are some examples:

Example 3a:

```
INSPECT WORD TALLYING COUNT FOR CHARACTERS
    AFTER INITIAL ''J'', REPLACING ALL ''A'' BY ''B''
```

1. WORD = ADJECTIVE to BDJECTIVE and COUNT = 6.
2. WORD = JACK to JBCK and COUNT = 3.
3. WORD = JUJMAB to JUJMBB and COUNT = 5.

Example 3b:

```
INSPECT WORD TALLYING COUNT FOR ALL ''L''
    REPLACING LEADING ''A'' BY ''E'' AFTER INITIAL ''L''
```

1. WORD = CALLAR remains CALLAR and COUNT = 2.
2. WORD = SALAMI to SALEMI and COUNT = 1.
3. WORD = LATTER to LETTER and COUNT = 1.

Example 3c

```
INSPECT WORD TALLYING COUNT FOR LEADING ''XX''
    REPLACING FIRST ''2'' BY ''3'' AFTER INITIAL ''TOT''
```

1. WORD = ALXX2TOT323 to ALXX2TOT333 and COUNT = 0.

2. WORD = XXXXXXX23TOT remains XXXXXXX23TOT and COUNT = 3.

3. WORD = X12TOT2TOTX12 to X12TOT3TOTX12 and COUNT = 0.

The TALLYING data item must be a numeric data item. The usage of all other data items must be DISPLAY. Literals in the formats must be nonnumeric.

7.6.3. STRING Statement

STRING statements combine characters from two or more data items into a single data item.

Format:

```
STRING {identifier-1} [, {identifier-2}] ... DELIMITED BY {identifier-3}
       {literal-1   }  [, {literal-2   }]                 {literal-3   }
                                                          {SIZE        }
    [, {identifier-4} [, {identifier-5}] ... DELIMITED BY {identifier-6}] ...
       {literal-4   }  [, {literal-5   }]                 {literal-6   }
                                                          {SIZE        }]
    INTO identifier-7 [WITH POINTER identifier-8]
    [,ON OVERFLOW imperative-statement]
```

Literals and identifiers left of DELIMITED are sending items; characters from sending items are combined to form the receiving item following INTO.

You can move all the characters (or selected characters) from the sending items. The SIZE option moves all the characters. If A = HIGH and B = WAY, the following coding makes C equal HIGHWAY.

```
STRING A, B DELIMITED BY SIZE INTO C
```

To move selected characters, you specify a literal or identifier that delimits the move. For instance, if the delimiter is G, only characters in each sending item to the left of G are included in the move. Thus, if A = HIGH and B = LIGHT, the following makes C equal HILI.

```
STRING A, B DELIMITED BY ''G'' INTO C
```

Sending items represented by identifiers, such as A and B in these examples, must be described as USAGE IS DISPLAY and, if numeric, must be described as integers with no P's in the PICTURE clause. Literals that are sending items must be nonnumeric. Receiving items, such as C in the examples, must be elementary alphanumeric data items described without editing symbols or the JUSTIFIED clause.

If the character-string formed from the sending items has fewer character positions than the receiving item, the unaffected character positions keep the value they had before execution of the STRING statement. So, if A = XXX, B = YYY, and C = ZZZZZZZZZ, the following code changes C to XXXYYYZZZ.

```
STRING A B DELIMITED BY SIZE INTO C
```

If the character-string formed from the sending items has more character positions than the receiving item, the move ends when the receiving item is filled; then the imperative statement associated with the ON OVERFLOW phrase is executed. If there is no ON OVERFLOW phrase, control passes to the next statement.

You use the POINTER phrase to indicate the character position in the receiving item into which data is moved. Thus, just as the values of some rightmost characters in the receiving item may not change, the values of a given number of leftmost character positions also may not change. If A = XXX, B = YYY, C = ZZZZZZZZZ, and CHAR-POS = 3, the following code changes C to ZZXXXYYYZ, and CHAR-POS = 9.

```
STRING A B DELIMITED BY SIZE
    INTO C WITH POINTER CHAR-POS
```

Pointers, such as CHAR-POS, must be described as elementary integers large enough to contain a value equal to the number of character positions in the receiving item, plus 1. The value of a pointer is incremented by 1 each time a character is moved from a sending item to the receiving item. If you don't use the WITH POINTER phrase, the first character from the first sending item is moved to the leftmost character position of the receiving item.

If you set a pointer improperly, that is, if you give it a value that is less than 1 or greater than the number of character positions in the receiving item, execution of a STRING statement causes an overflow condition. Control passes to the next statement or, if specified, to the imperative statement associated with the ON OVERFLOW phrase.

7.6.4. UNSTRING Statement

UNSTRING statements, as you might expect, have an effect opposite to that of STRING statements; they move characters from a single sending field to multiple receiving fields. The format is:

```
UNSTRING identifier-1
  [DELIMITED BY [ALL] {identifier-1} [, OR [ALL] {identifier-3}] ... ]
                      {literal-1   }            {literal-2   }
  INTO identifier-4 [, DELIMITER IN identifier-5][, COUNT IN identifier-6]
  [, identifier-7 [, DELIMITER IN identifier-8][, COUNT IN identifier-9]] ...
  [WITH POINTER identifier-10][TALLYING IN identifier-11]
  [, ON OVERFLOW imperative-statement]
```

7.6.4.1. Sending Field

The sending field, represented by identifier-1, immediately follows the UNSTRING verb. Its characters are moved to the receiving fields (identifier-4, identifier-7, etc.) following INTO. The number of characters moved depends on the number of characters in the receiving fields. If A is described as PIC X(4) and B as PIC X(5), and WORD = HIGHLIGHT, then

```
UNSTRING WORD INTO A B
```

moves four characters (HIGH) to A, and five characters (LIGHT) to B. If A description is PIC X(6) and B is PIC X(3), the same statement makes A = HIGHLI and B = GHT.

You must describe the sending field as alphanumeric. You describe the receiving fields as alphabetic (without the B symbol), alphanumberic, or numeric (without the P symbol). The USAGE of the receiving field must be DISPLAY. Each literal must be a nonnumeric literal.

As with the STRING statement, you may specify characters that delimit the move. Then, characters in the sending field preceding the first occurrence of the delimiter are moved to the first receiving field (unless the receiving field is filled before a delimiter is encountered) and characters between the first and second occurrence of the delimiter are moved to the second receiving field, etc.

Thus, if WORD = HIGHLIGHT, and A and B are described as

```
02  A  PIC X(3).
02  B  PIC X(3).
```

the statement

```
UNSTRING WORD DELIMITED BY ''GH'' INTO A, B
```

makes A = HI△ and B = LI△.

If the number of characters preceding the first delimiter is less than the number of characters in the first receiving field, the unaffected character positions in the receiving field are zero or space filled, depending on whether the item is numeric or nonnumeric. If WORD = HIGHLIGHT and A and B are described as

```
Ø2  A  PIC X(4) VALUE ''XXXX''.
Ø2  B  PIC X(5) VALUE ''ZZZZZ''.
```

then

```
UNSTRING WORD DELIMITED BY ''G'' INTO A, B
```

makes A = HI△△ and B = LI△△. Similarly, if WORD = "45X9732X43", and the data division includes

```
77  A  PIC 9(5) VALUE ZEROS.
77  B  PIC 9(5) VALUE ZEROS.
77  C  PIC 9(5) VALUE ZEROS.
```

then

```
UNSTRING WORD DELIMITED BY ''X'' INTO A, B, C
```

makes A = 00045, B = 09732, and C = 00043.

You may specify more than one delimiter. For instance,

```
UNSTRING WORD DELIMITED BY ''X'' OR ''Y'' OR ''Z'' INTO A, B
```

This stops the examination of WORD whenever an X, Y, or Z is encountered.

So, if WORD = ABCYDZ, A = ABC and B = D.

7.6.4.2. ALL Word

The word ALL before a delimiter means one or more contiguous occurrences of the delimiter are considered to be one occurrence. Suppose WORD = MMMXXNNN and A and B are defined as

```
Ø2  A  PIC X(3).
Ø2  B  PIC X(3).
```

If you code

```
UNSTRING WORD DELIMITED BY ALL ''X'' INTO A B
```

the XX is considered to be one occurrence of the delimiter, and A = MMM and B = NNN. But if you do not use ALL, as in

```
UNSTRING WORD DELIMITED BY ''X'' INTO A B
```

XX is two occurrences of the delimiter; thus, after MMM is moved to A, there are no characters before the next delimiter to be moved to B and B is space (or zero) filled according to the description of the field.

7.6.4.3. DELIMITED and COUNT Phrases

You may use the DELIMITER IN and COUNT IN phrases only if you used the DELIMITED BY phrase. Identifiers associated with DELIMITER IN phrases receive the delimiting characters that caused data to be moved to the current receiving item. Identifiers associated with COUNT IN phrases show the number of characters moved to the current receiving item. If WORD = HIGHWAYS

```
UNSTRING WORD DELIMITED BY ''GH'' OR ''S''
    INTO A DELIMITER IN A-DELS  COUNT IN A-COUNT
    INTO B DELIMITER IN B-DELS  COUNT IN B-COUNT
```

This makes A = HI, A-DELS = GH, A-COUNT = 2, B = WAY, B-DELS = S, and B-COUNT = 3. If you specify DELIMITER IN, and the delimiting condition is the end of the sending item, the identifier associated with DELIMITER IN equals spaces. So, if WORD = HIGHWAYS

```
UNSTRING WORD DELIMITED BY ''GH''
    INTO A DELIMITER IN A-DELS  COUNT IN A-COUNT
    INTO B DELIMITER IN B-DELS  COUNT IN B-COUNT
```

Thus, A = HI, A-DELS = GH, A-COUNT = 2, B = WAYS, B-DELS = blanks, and B-COUNT = 4.

You must describe identifiers associated with DELIMITER IN as alphanumeric; identifiers associated with COUNT IN must represent integers.

7.6.4.4. POINTER Phrase

The POINTER phrase allows you to begin examination of the sending field at any character position. For example, if WORD = XL459S243 and the data division includes the following coding

```
Ø2  A  PIC 9(5).
Ø2  B  PIC 9(5).
Ø2  POINT  PIC 9(2).
```

then

```
MOVE 3 TO POINT.
UNSTRING WORD DELIMITED BY ''S'' INTO A, B
     WITH POINTER POINT
```

makes the examination of WORD begin at the third character position; thus, A = 00459 and B = 00243. The pointer, which must be an integer, is incremented by 1 as each character in the sending item is examined.

If you set the pointer improperly (if you make it less than 1 or greater than the number of characters in the sending item) or if characters remain in the sending item after all the receiving items are filled, an overflow condition exists. Then, the imperative statement associated with the ON OVERFLOW phrase is executed or, if the ON OVERFLOW phrase is not specified, control passes to the next statement. If WORD = HIGHWAYS and the data division includes

```
Ø2  A  PIC  X(4).
Ø2  B  PIC  X(3).
```

then

```
UNSTRING WORD INTO A, B
     ON OVERFLOW GO TO SEND-LARGE
```

makes A = HIGH and B = WAY. Since the S from the sending field isn't sent, an overflow condition exists, and control passes to SEND-LARGE.

7.6.4.5. TALLYING Phrase

The TALLYING phrase provides a count of the receiving items into which data is moved. You must initialize the TALLYING data item before the execution of an UNSTRING statement. Thus, if word = PASSWORD, and the data division includes

```
77  A  PIC  X(4).
77  B  PIC  X(4).
77  C  PIC  X(3)  VALUE ''XYZ''.
77  D  PIC  9(2).
```

then

```
MOVE ZEROS TO D
UNSTRING WORD INTO A, B, C
    TALLYING IN D
```

makes A = PASS, B = WORD, C = XYZ (unchanged), and D = 2.

7.7. PROCEDURE BRANCHING VERBS

Normally, procedure division statements are executed consecutively in the order of appearance. The procedure branching verbs (EXIT, IF, GO TO, ALTER, and PERFORM) change the normal sequence of operation; they cause the control to branch to a line in the program other than the next executable statement.

7.7.1. EXIT Statement

EXIT statements are not executable; they're used only to serve as common end points for a series of procedures. For example:

```
AA-MAIN.
      .
      .
      .
      IF ... GO TO AA-EXIT.
AA-TYPE-1.
      .
      .
      .
      IF ... GO TO AA-EXIT.
```

(continued)

```
AA-TYPE-1.
     .
     .
     .
     IF ... GO TO AA-EXIT.
.
.
.
AA-EXIT.
     EXIT.
```

Several paragraphs may branch to the same point – a paragraph called AA-EXIT that serves solely as an exit point for a series of related procedures. Of course, the paragraphs could branch to another paragraph elsewhere in the program, or even to a paragraph that contains no statements. Thus, the EXIT statement isn't needed; it's normally used for a common exit of a subroutine.

When you use EXIT, it must, as in the example, be the only statement in a sentence that is the only sentence in a paragraph.

7.7.2. IF Statement

IF statements call for the evaluation of conditional expressions. Subsequent action of the program depends on whether the value of an expression is true or false. The format is:

```
IF condition;{statement-1   }{; ELSE statement-2    }
             {NEXT SENTENCE }{; ELSE NEXT SENTENCE  }
```

The IF statement is used in examples throughout this manual, particularly in the discussion of conditional expressions (7.2.2). In those examples, imperative statements immediately following the condition provide the actions taken when the expressions are true. In

```
IF A > B ADD A TO TOTAL.
```

the value of A is added to the value of TOTAL, provided A is greater than B. Otherwise, control passes immediately to the next sentence.

You use the ELSE phrase to specify actions taken when conditional expressions are false. Adding the ELSE phrase to the sentence

```
IF A > B ADD A TO TOTAL
ELSE MOVE A TO C.
```

means that if A is not greater than B, A is not added to TOTAL; instead, MOVE A TO C is executed.

You can use the NEXT SENTENCE phrase rather than an imperative statement for either the true or false condition:

Example 1:

```
IF A > B NEXT SENTENCE
    ELSE MOVE A TO C.
```

Example 2:

```
IF A > B ADD A TO TOTAL
    ELSE NEXT SENTENCE.
```

The first example transfers control to the next executable sentence when the condition is true; the second example does so when the condition is false.

Of course, in the second example, the NEXT SENTENCE phrase really is redundant, and you may omit the phrase, if you end the first line with a period. You generally associate the NEXT SENTENCE phrase with false conditions only when IF statements are nested; that is, when statement-1 or statement-2 in the format contains another IF statement:

```
IF A > B
    IF C > D GO TO MAIN
    ELSE NEXT SENTENCE
ELSE GO TO EOJ.
```

In this example, when IF statements are nested, each ELSE phrase, from left to right, is paired with the closest preceding IF phrase that isn't already paired with an ELSE. Thus, the first (from the left) ELSE phrase (ELSE NEXT SENTENCE) is paired with the closest preceding IF (IF C > D) and the second ELSE phrase (ELSE GO TO EOJ) is paired with the closest preceding IF not already paired with an ELSE (IF A > B). So, if A > B is false, the corresponding ELSE phrase (GO TO EOJ) is executed; but if A > B is true, then C > D must be evaluated. If it is also true, GO TO MAIN is executed. But if it is false, control passes to the next sentence.

Many levels of nesting are possible. Programmers generally use indentation of statements to make the IF-ELSE pairings easy to follow:

```
IF A > B
    IF C > D
        IF E > F
            IF G > H MOVE 1 TO IND
            ELSE GO TO Z
        ELSE NEXT SENTENCE
    ELSE PERFORM Q1
ELSE GO TO EOJ.
```

Each IF begins in the same source code column as its corresponding ELSE phrase. The IF phrases are evaluated, in order, until one is found to be false. Then, the corresponding ELSE phrase is executed, and control passes to the next sentence. In this example, MOVE 1 TO IND is executed only if A > B, C > D, E > F, and G > H all are true.

7.7.3. GO TO Statement

You use GO TO statements to transfer control to paragraphs or sections in the procedure division. There are two formats:

Format 1:

```
GO TO [procedure-name-1]
```

Format 2:

```
GO TO procedure-name-1 [, procedure-name-2] ... , procedure-name-n
      DEPENDING ON identifier
```

Procedure-names are section-names or paragraph-names (which may be qualified by section-names).

Format 1 transfers control to the paragraph or section named in the statement. For example:

```
GO TO MAIN
```

This transfers control to the paragraph or section called MAIN.

Format 2 transfers control to a procedure-name based on the value of an identifier. The identifier should have an integer value that corresponds to the position in the GO TO statement of one of the procedure-names. Procedure-name-1 is in position 1, procedure-name-2 is in position 2, etc. Thus

```
GO TO A B C DEPENDING ON NUM
```

transfers control to A if NUM = 1, to B if NUM = 2, and to C if NUM = 3. If NUM isn't equal to 1, 2, or 3, control passes to the next statement.

You can name the same procedure more than once in a Format 2 GO TO statement:

```
GO TO A B B A A DEPENDING ON NUM
```

This transfers control to A if NUM = 1, 4, or 5, and to B if NUM = 2 or 3.

A Format 1 GO TO statement that does not specify a procedure-name is valid only if it is executed after an ALTER statement (7.7.4) that references it. This type of GO TO statement must be the only statement in a paragraph.

7.7.4. ALTER Statement

ALTER statements change procedure-names referenced by Format 1 GO TO statements. The format is:

```
ALTER procedure-name-1 TO [PROCEED TO] procedure-name-2
  [, procedure-name-3 TO [PROCEED TO] procedure-name-4] ...
```

The procedure-names must be paragraph-names. The paragraph named to the left of TO PROCEED TO must consist solely of a Format 1 GO TO statement. The ALTER statement modifies that GO TO statement so that it refers to the paragraph named to the right of TO PROCEED TO:

```
ALTER MAIN TO PROCEED TO B
```

This in effect changes the paragraph

```
MAIN.
    GO TO A.
```

to

```
MAIN.
    GO TO B.
```

This stands for all subsequent executions of MAIN, unless MAIN is referenced by another ALTER statement that changes the destination again.

The GO TO statement in the paragraph referenced by ALTER does not have to specify a procedure-name. Thus, in the example, MAIN can be coded

```
MAIN.
    GO TO.
```

The coding

```
ALTER MAIN TO PROCEED TO B
```

still, in effect, changes the paragraph to

```
MAIN.
    GO TO B.
```

7.7.5. PERFORM Statement

PERFORM statements transfer control to a series of procedures and, once the procedures are executed, return control to the statement following the PERFORM statement. There are four formats:

Format 1:

```
PERFORM procedure-name-1 [{THROUGH} procedure-name-2]
                         {THRU   }
```

Format 2:

```
PERFORM procedure-name-1 [{THROUGH} procedure-name-2] {identifier-1} TIMES
                         {THRU   }                    {integer-1   }
```

Format 3:

```
PERFORM procedure-name-1 [{THROUGH} procedure-name-2] UNTIL condition-1
                         {THRU   }
```

Format 4:

```
PERFORM procedure-name-1 [{THROUGH} procedure-name-2]
                         {THRU   }
    VARYING {identifier-2} FROM {identifier-3}
            {index-name-1}      {index-name-2}
                                {literal-1   }
       BY {identifier-4} UNTIL condition-1
          {literal-3   }
   [AFTER {identifier-5} FROM {identifier-6}
          {index-name-3}      {index-name-4}
                              {literal-3   }
      BY {identifier-7} UNTIL condition-2
         {literal-4   }
   [AFTER {identifier-8} FROM {identifier-9}
          {index-name-5}      {index-name-6}
                              {literal-5   }
      BY {identifier-10} UNTIL condition-3
         {literal-6    }  ]]
```

The simplest PERFORM executes one paragraph or section.

For example:

```
PERFORM A
```

If A is a section, this executes all the paragraphs in the section. If A is a paragraph, it executes only the sentences in the paragraph. After the procedures are executed, control passes to the statement following the PERFORM statement. For example, in the code

```
MAIN-PARA  SECTION.
CARDIN.
      READ CARDFILE AT END GO TO EOJ.
      PERFORM FIND-CARD-TYPE.
      GO TO MAIN-PARA.
FIND-CARD-TYPE  SECTION.
CHECK-IND.
      IF IND EQUAL TO ''A'' GO TO A.
      IF IND EQUAL TO ''B'' GO TO B.
      IF IND EQUAL TO ''C'' GO TO C.
A.
      .........
      .........
      .........
B.
      .........
      .........
      .........
C.
      .........
      .........
      .........
FIND-CARD-TYPE-EXIT.
      EXIT.
END-JOB  SECTION.
EOJ.
      .........
```

the sentence

```
PERFORM FIND-CARD-TYPE
```

executes the paragraphs CHECK-IND, A, B, C, and FIND-CARD-TYPE-EXIT, then returns control to GO TO MAIN-PARA. Of course, it is possible not all the paragraphs (or all the statements in each paragraph) will be executed. If, in CHECK-IND, the conditional statement

```
IF IND EQUAL TO ''C''
```

is true, control branches to C and A and B aren't executed.

The procedures performed are executed beginning with the first statement in the first paragraph. You must make certain execution of the procedures ends with the last statement in the last paragraph; otherwise, control is never returned to the statement following the PERFORM statement.

The THROUGH option allows you to specify a series of paragraphs or sections to be executed:

```
PERFORM FIND-CARD-TYPE
```

This could be written as

```
PERFORM CHECK-IND THROUGH FIND-CARD-TYPE-EXIT
```

If you use the THROUGH option and either procedure is in the declaratives portion of the program, the other procedure must also be in the declaratives portion.

Format 2 executes the paragraphs or sections a specified number of times before returning control to the statement following the PERFORM statement. You code

```
PERFORM READ-ROUTINE COUNTER TIMES
```

to consecutively execute READ-ROUTINE the number of times indicated by the value of COUNTER. COUNTER must represent a positive integer; if it is negative, or zero, control passes immediately to the statement following the PERFORM statement, without executing READ-ROUTINE.

If the value of COUNTER changes during execution of READ-ROUTINE, the operation of the PERFORM statement is not affected; if COUNTER equals 5 when control passes to the PERFORM statement, READ-ROUTINE is executed five times regardless of any change in COUNTER.

Format 3 executes the procedures until a specified condition is true. The paragraphs or sections are executed once each time that condition is tested and found to be false. If you code

```
PERFORM MAIN THROUGH FIND-COST UNTIL TOTAL > LIMIT
```

and the condition TOTAL > LIMIT is false, the procedures MAIN THROUGH FIND-COST are executed once, and the condition is tested again. This cycle continues until the condition is true. If the condition is true the first time it is tested, control passes immediately to the statement following the PERFORM statement.

Format 4 PERFORM statements generally are used for table handling routines; they allow you to vary the values of indexes and subscripts before each execution of a procedure. For instance, to find the average value of the items in this single-level table:

```
01  PREMIUM-TABLE.
    05  PREM OCCURS 15 TIMES  PIC 9(7) COMP.
```

You could code as follows:

```
PREM-LOOP-SETUP.
      MOVE 1 TO PREM-SS.
PREM-LOOP.
      ADD PREM (PREM-SS) TO TOTAL.
      ADD 1 TO PREM-SS.
      IF PREM-SS > 15 GO TO PREM-CALC
           ELSE GO TO PREM-LOOP.
PREM-CALC.
      DIVIDE TOTAL BY 15 GIVING AVERGE.
```

The term PREM-SS is a subscript. This routine adds the value of each occurrence of PREM, stores the total value in TOTAL, and divides by 15 to find the average value, which is stored in AVERGE. Using a Format 4 PERFORM statement, you could code the routine as follows:

```
PERFORM PREM-LOOP VARYING PREM-SS FROM 1 BY 1
      UNTIL PREM-SS > 15.
      GO TO PREM-CALC.
PREM-LOOP.
      ADD PREM (PREM-SS) TO TOTAL.
PREM-CALC.
      DIVIDE TOTAL BY 15 GIVING AVERGE.
```

Execution proceeds as follows:

1. The identifier in the VARYING phrase (PREM-SS) is set to the value specified in the FROM phrase.

2. The condition (PREM-SS > 15) is tested. Because it's false, the procedure (PREM-LOOP) is executed once. Thus, the first occurrence of PREM is added to TOTAL.

3. PREM-SS is incremented by 1, the value specified in the BY phrase.

4. The condition is tested again and (because it is still false) PREM-LOOP is executed again, this time adding the value of the second occurrence of PREM to TOTAL.

5. The loop is repeated 15 times until PREM-SS > 15 is true; then, control passes to GO TO PREM-CALC.

Note that we tested the subscript for a value greater than (rather than equal to) 15. That's because the condition is tested immediately after the identifier is incremented. If EQUAL TO were used, the procedure would be executed only 14 times.

Figure 7-1 is a flowchart that summarizes the execution of a Format 4 PERFORM statement that has one condition.

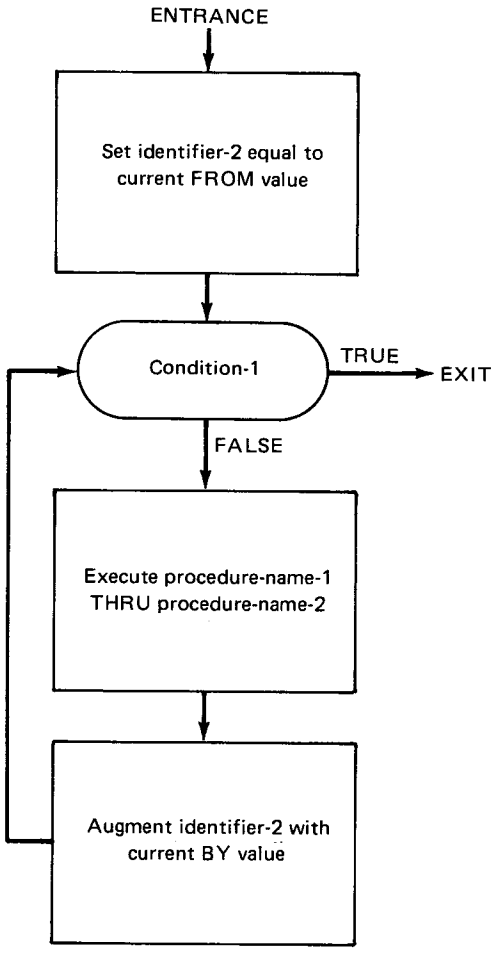


Figure 7—1. Format 4 Flowchart for PERFORM Statement with 1 Condition

When you work with 2-level tables, you vary two identifiers or indexes. To find the average of the values in the following

```
01  PREM-TABLE.
    05  PREM-CLASS  OCCURS 3 TIMES
                     INDEXED BY CLASS-INDEX.
        10  PREM  OCCURS 15 TIMES  PIC 9(7)
                     INDEXED BY PREM-INDEX.
```

you code

```
PERFORM PREM-LOOP VARYING CLASS-INDEX FROM 1 BY 1
    UNTIL CLASS-INDEX > 3  AFTER PREM-INDEX FROM 1
    BY 1 UNTIL PREM-INDEX > 15.
      GO TO PREM-CALC.
PREM-LOOP.
      ADD PREM (CLASS-INDEX, PREM-INDEX) TO TOTAL.
PREM-CALC.
      DIVIDE TOTAL BY 45 GIVING AVERGE.
```

Execution proceeds this way:

1. CLASS-INDEX and PREM-INDEX are set to their FROM values (1).

2. Condition-1 and condition-2 are tested, in order. Because they are false, PREM-LOOP is executed once; thus, PREM (1, 1) is added to TOTAL.

3. PREM-INDEX, but not CLASS-INDEX, is incremented by its BY value (1).

4. Condition-2 (PREM-INDEX > 15) is tested. Because it's false, PREM-LOOP is executed again, this time adding PREM (1, 2) to TOTAL.

5. The incrementing of PREM-INDEX and testing of condition-2 continues until condition-2 is true. At that time, PREM (1, 15) has just been added to TOTAL.

6. PREM-INDEX is reset to its FROM value (1) and CLASS-INDEX is incremented by its BY value (1). Condition-1 is tested and, because it's false, PREM-LOOP is executed. This adds PREM (2, 1) to TOTAL.

7. The cycle continues as before, until condition-2 is true again. Thus, PREM (2, 2), PREM (2, 3), PREM (2, 4) ... PREM (2, 15) are added to TOTAL.

8. The cycle repeats for PREM (3, 1), PREM (3, 2) ... PREM (3, 15).

9. Control passes to GO TO PREM-CALC.

Thus you can see that identifier-5 or index-name-3 in the format goes through a complete cycle each time identifier-2 or index-name-1 is varied. Also note that Format 4 works the same way whether you use subscripting or indexing. In the first example, when subscripting is used, the setting of the identifier to its FROM value is the equivalent of

```
MOVE 1 TO PREM-SS
```

However, when indexing is used, it is the same as

```
SET CLASS-INDEX PREM-INDEX TO 1
```

Similarly, incrementing subscripts by the BY value is the same as

```
ADD 1 TO PREM-SS
```

but incrementing PREM-INDEX and CLASS-INDEX is equivalent to

```
SET PREM-INDEX UP BY 1
```

and

```
SET CLASS-INDEX UP BY 1
```

Figure 7–2 summarizes execution of a Format 4 PERFORM statement that has two conditions.

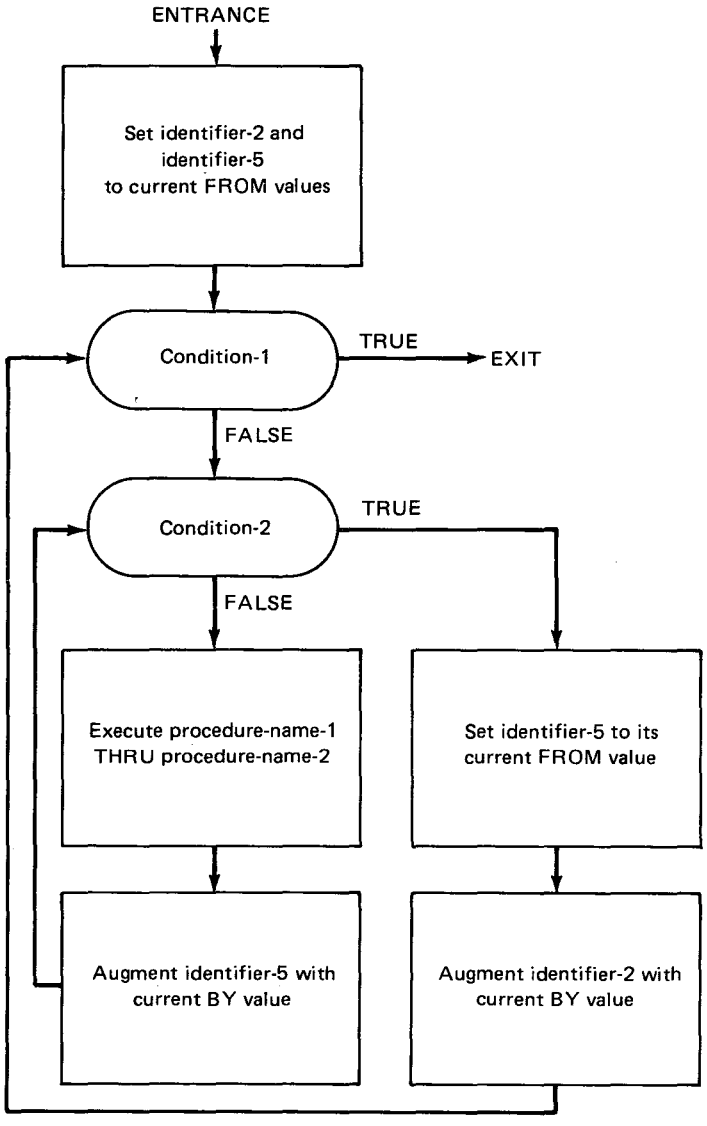


Figure 7—2. Format 4 Flowchart for PERFORM Statement with 2 Conditions

Using PERFORM statements to manipulate 3-level tables works the same way as for 2-level tables, except that identifier-8 in the format goes through a complete cycle each time identifier-5 is augmented by identifier-7 or literal-4, which in turn goes through a complete cycle each time identifier-2 is varied. The operation of a Format 4 PERFORM having three conditions is summarized in Figure 7-3.

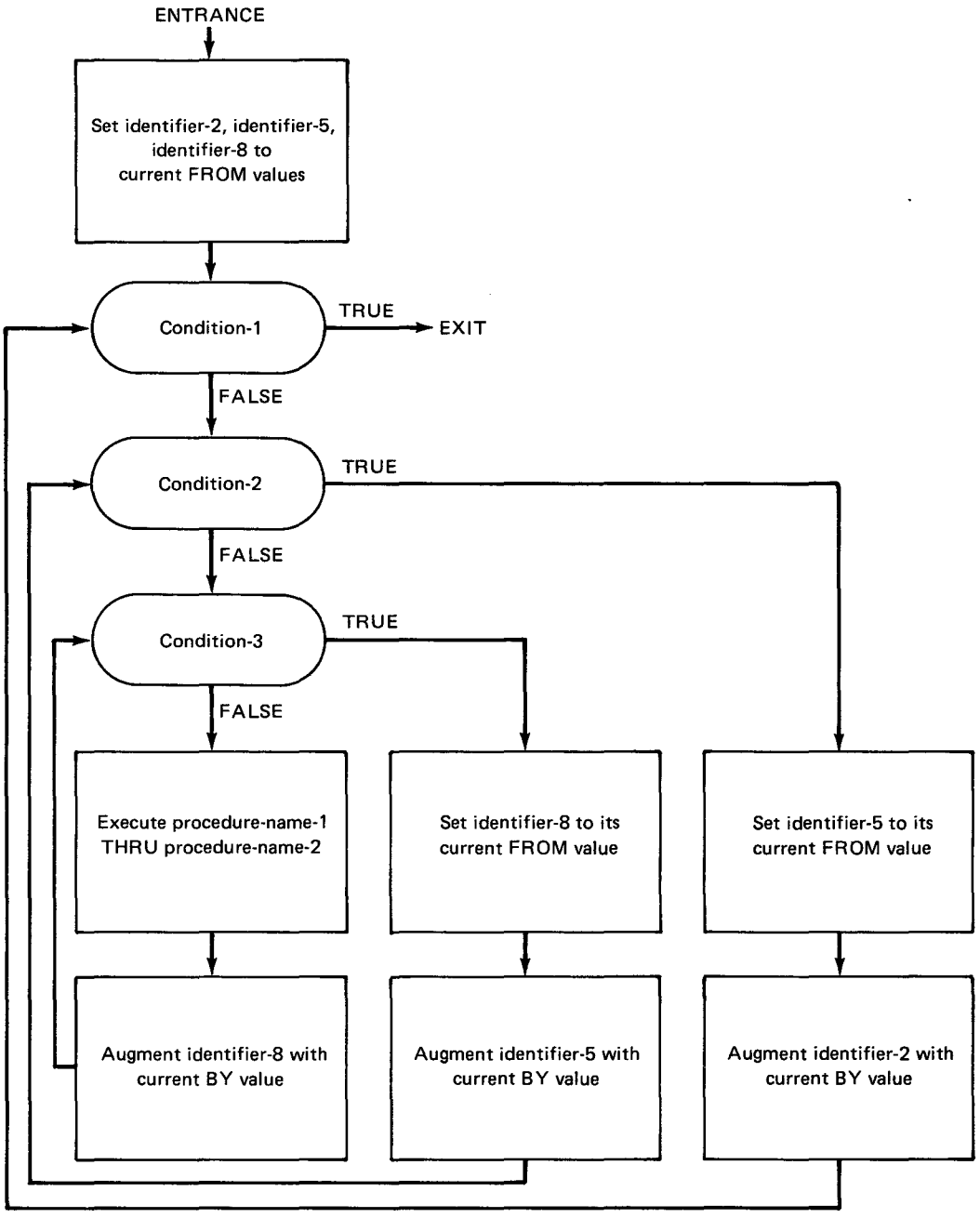


Figure 7—3. Format 4 Flowchart for PERFORM Statement with 3 Conditions

The sequence of statements a PERFORM statement refers to may include another PERFORM statement, but the sequence of procedures associated with the included PERFORM statement must either be totally included in, or totally excluded from, the logical sequence referred to by the first PERFORM statement. Additionally, two such perform statements have the same exit point. For example:

```
X.
    PERFORM A THRU M.
A.
D.
    PERFORM F THRU J.
F.
J.
M.
```

The PERFORM statement in D is valid because it references procedures totally within the range of procedures referenced by the perform in X. Changing the PERFORM statement in D to

```
PERFORM F THRU M
```

is invalid because the included PERFORM statement shares an exit point with the first PERFORM statement.

The following example

```
X.
    PERFORM A THRU M.
A.
D.
    PERFORM F THRU J.
H.
M.
F.
J.
```

is valid because it references procedures totally outside of the range of procedures referenced by the first PERFORM statement.

Procedures may be referenced by more than one PERFORM statement in the program. For instance, the following is a valid use of PERFORM statements:

```
X.
    PERFORM H THRU J.
A.
    PERFORM M THRU J.
D.
    PERFORM H THRU F.
H.
M.
F.
J.
```

7.8. COMPILER DIRECTING VERB

The compiler directing verb USE directs the compiler to take a specific action at compile time.

7.8.1. USE Statement

USE statements allow you to specify procedures that are executed when there are system input-output errors. These procedures are in addition to the standard ones provided by your operating system's input-output control system.

Additionally, USE statements may provide procedures to process end-of-file conditions or file key specification errors, if the AT END or INVALID KEY phrase is not specified in the associated input-output statement. The format:

```
USE AFTER STANDARD {EXCEPTION} PROCEDURE ON {file-name-1 [,file-name-2] ... }.
                   {ERROR    }             {INPUT                           }
                                           {OUTPUT                          }
                                           {I-O                             }
                                           {EXTEND                          }
```

In this format, ERROR and EXCEPTION are equivalent.

The USE statements belong in the declaratives section, immediately following section headers. The error handling procedures follow the USE statements. Note that USE statements themselves are never executed; they merely define the conditions under which the procedures that follow are executed. The procedures may apply to specific files, or to all files opened in either INPUT, OUTPUT, I-O, or EXTEND mode.

For instance, you code

```
DECLARATIVES.
OUT-ERR SECTION. USE AFTER STANDARD ERROR PROCEDURE ON OUTPUT.
PARA.
     DISPLAY ''OUTPUT ERROR ON'' ID-FIELD UPON CONSOLE.
END DECLARATIVES.
```

If, during execution of your program, there is an error associated with any of your program's output files, the system's standard input-output procedures are executed, then the paragraph called PARA is executed (the message OUTPUT ERROR ON followed by the value of the field called ID-FIELD appears on the console).

Statements in USE procedures may not reference nondeclarative procedures, and no statements (except PERFORM) in the nondeclarative portion may reference procedures associated with a USE statement.

After execution of a USE procedure, control is returned to the invoking routine.

7.9. CODING EXAMPLE

Some of the features of a procedure division taken from a payroll program are included in this sample coding. The program reads a file of employees, prints a payroll check for each employee, and produces a payroll report.

The section in the declaratives portion of the procedure division defines what is done if there is a system error during processing of PAY-FILE. The message PAYROLL FILE ERROR appears on the system console.

The BEGIN-JOB section opens files, initializes data fields, and retrieves the system date.

The MAIN-LOOP section reads employee records and determines whether the employees are part time or full time. When employees are full time, the routine determines whether they're exempt or non-exempt. And when employees are non-exempt, the routine determines whether they worked overtime. The routine then directs execution of sections appropriate to the employee's status, writes a check and a line on the report, and returns for the next record.

The end of job section (EOJ) adds the totals line to the report, closes files, and ends processing.

Procedure Division Coding:

```
PROCEDURE DIVISION.
DECLARATIVES.
PAY-FILE-ERROR  SECTION.
     USE AFTER STANDARD ERROR PROCEDURE ON PAY-FILE.
ERR-ROUT.
     DISPLAY ''PAYROLL FILE ERROR'' UPON CONSOLE.
END DECLARATIVES.
BEGIN-JOB  SECTION.
HOUSEKEEP-ROUT.
     OPEN INPUT PAY-FILE  OUTPUT CHECK-FILE REPORT-FILE.
     MOVE ZEROS TO PT-TOTAL EXEMPT-TOTAL NON-EXEMPT-TOTAL
          OT-TOTAL PT-HOURS OT-TOT-HOURS TOT-CHECKS.
     MOVE SPACES TO PAY-WORK CHECK-WORK PAY-REPT-WORK
          PAY-REPT-WORK-2.
     ACCEPT RUN-DATE FROM DATE.
MAIN-LOOP  SECTION.
FULL-ROUT.
     READ PAY-FILE INTO PAY-WORK AT END GO TO EOJ.
     IF FULL-TIME
          IF NON-EXEMPT
               IF OVERTIME-PAY PERFORM CALC-NON-EXEMPT
                               PERFORM CALC-OVERTIME
               ELSE PERFORM CALC-NON-EXEMPT
          ELSE PERFORM CALC-EXEMPT
     ELSE PERFORM CALC-PART-TIME.
     PERFORM WRITE-REPORT THRU WRITE-CHECK.
     GO TO MAIN-LOOP.
CALC-PART-TIME  SECTION.
PART-ROUT.
     COMPUTE PT-PAY = PT-RATE * PT-HRS-WORKED.
     MOVE PT-PAY TO CHECK-AMOUNT.
     ADD PT-PAY TO PT-TOTAL.
     ADD PT-HRS-WORKED TO PT-HOURS.
CALC-EXEMPT  SECTION.
EXEMPT-ROUT.
     MOVE WEEKLY-PAY TO-CHECK-AMOUNT.
     ADD WEEKLY-PAY TO EXEMPT-TOTAL.
CALC-NON-EXEMPT  SECTION.
NON-EXEMPT-ROUT.
     COMPUTE REG-PAY = REG-RATE * HOURS-WORKED.
     MOVE REG-PAY TO CHECK-AMOUNT.
     ADD REG-PAY TO NON-EXEMPT-TOTAL.
CALC-OVERTIME  SECTION.
```

(continued)

```
OVERTIME-ROUT.
     SUBTRACT 40 FROM HOURS-WORKED GIVING OT-HOURS.
     DIVIDE 2 INTO REG-RATE GIVING OT-RATE.
     COMPUTE OT-PAY = OT-RATE * OT-HOURS.
     ADD OT-PAY TO OT-TOTAL.
     ADD OT-HOURS TO OT-TOT-HOURS.
     ADD OT-PAY TO REG-PAY, NON-EXEMPT-TOTAL.
     MOVE REG-PAY TO CHECK-AMOUNT.
WRITE-REPORT  SECTION.
REPORT-ROUT.
     MOVE LAST-NAME TO REPORT-NAME.
     MOVE CHECK-AMOUNT TO REPT-CHECK-AMOUNT.
     ADD CHECK-AMOUNT TO TOT-CHECKS.
     WRITE PAY-REPT FROM PAY-REPT-WORK.
WRITE-CHECK  SECTION.
CHECK-ROUT.
     MOVE NAME TO NAME-OUT.
     MOVE CHECK-AMOUNT TO CHECK-AMOUNT-OUT.
     WRITE CHECK-RECD FROM CHECK-WORK.
EOJ  SECTION.
FINISH-ROUT.
     MOVE PT-TOTAL TO REPT-PT-TOTAL.
     MOVE EXEMPT-TOTAL TO REPT-EXEMPT-TOTAL.
     MOVE NON-EXEMPT-TOTAL TO REPT-NON-EXEMPT-TOTAL.
     MOVE OT-TOTAL TO REPT-OT-TOTAL.
     MOVE PT-HOURS TO REPT-PT-HOURS.
     MOVE OT-TOT-HOURS TO REPT-OT-TOT-HOURS.
     MOVE RUN-DATE TO REPT-RUN-DATE.
     WRITE PAY-REPT FROM PAY-REPT-WORK-2.
     CLOSE PAY-FILE CHECK-FILE REPORT-FILE.
     DISPLAY ''END OF PAYROLL RUN'' UPON CONSOLE.
     STOP RUN.
```

8. Table Handling

8.1. DEFINING TABLES

Tables consist of functionally related, contiguous data items arranged in some logically meaningful order. For example, the following listing of the months of the year is a table.

1	JANUARY
2	FEBRUARY
3	MARCH
4	APRIL
5	MAY
6	JUNE
7	JULY
8	AUGUST
9	SEPTEMBER
10	OCTOBER
11	NOVEMBER
12	DECEMBER

8.1.1. Table Elements

Each month is contained in a table element associated with a relative position in the table. The month table has 12 elements, one for each month, and each element has its own unique position within the table.

To create the month table in your COBOL program you assign a data-name which is common to all the elements in the table. This data-name is then used along with the relative number of the element you wish to access. For instance, data-name (6) would be used to access the month of JUNE. You can use a subscript or index to indicate the relative position of the desired element within the table.

To define tables in a COBOL program, you use OCCURS clauses (8.1.2) to specify the number of times the table items (in this example, the months of the year) are repeated. You include the specific table data (i.e., JANUARY, FEBRUARY,...) in your COBOL source code or (particularly if your tables are long or likely to need changes frequently) you read the data from an external device when the program is executed.

You use VALUE clauses to include the data in the COBOL source code:

```
01  TABLE-OF-MONTHS.
    03  TABLE-ENTRIES.
        05  FILLER     PIC X(9)     VALUE  ''JANUARY  ''.
        05  FILLER     PIC X(9)     VALUE  ''FEBRUARY ''.
        05  FILLER     PIC X(9)     VALUE  ''MARCH    ''.
        05  FILLER     PIC X(9)     VALUE  ''APRIL    ''.
        05  FILLER     PIC X(9)     VALUE  ''MAY      ''.
        05  FILLER     PIC X(9)     VALUE  ''JUNE     ''.
        05  FILLER     PIC X(9)     VALUE  ''JULY     ''.
        05  FILLER     PIC X(9)     VALUE  ''AUGUST   ''.
        05  FILLER     PIC X(9)     VALUE  ''SEPTEMBER''.
        05  FILLER     PIC X(9)     VALUE  ''OCTOBER  ''.
        05  FILLER     PIC X(9)     VALUE  ''NOVEMBER ''.
        05  FILLER     PIC X(9)     VALUE  ''DECEMBER ''.
    03  FILLER REDEFINES TABLE-ENTRIES.
        05  MONTH        OCCURS 12 TIMES
                         INDEXED BY MONTH-INDEX  PIC X(9).
```

Since VALUE clauses are invalid in data description entries that either contain an OCCURS clause or are subordinate to entries containing an OCCURS clause, we cannot use an OCCURS clause with TABLE-OF-MONTHS or TABLE-ENTRIES. Instead, we redefine that table area and include an OCCURS clause in the description of the data-name used in the redefinition. That data-name (MONTH) plus a subscript (8.2.1) or index (8.2.2) that defines the element's relative position in the table (1 through 12) references an individual table element. Note that all table items must be the same length; thus, all months are space-filled to the length of the longest name, SEPTEMBER.

To retrieve the table of months from an external device such as disk, you define a storage area to contain it:

```
01  TABLE-OF-MONTHS.
    03  TABLE-ENTRIES     OCCURS 12 TIMES
                           INDEXED BY MONTH-INDEX.
        05  MONTH     PIC X(9).
```

This coding describes 12 contiguous occurrences of the 9-character data-name called MONTH, just as in the previous table. This time, however, you do not include the specific table values. If the table is in a file called DISK-FILE, a READ statement such as

```
READ DISK-FILE INTO TABLE-OF-MONTHS
```

transfers the values – the names of the months – from the disk to your storage area. Again, you reference the individual table elements by using MONTH plus a subscript or index.

8.1.2. OCCURS Clause

The OCCURS clause specifies the number of times individual table elements are repeated, and provides information needed for subscripting and indexing. There are two formats for the OCCURS clause:

Format 1:

```
OCCURS integer-1 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] ...]
```

Format 1 defines fixed-length tables. If you code as follows

```
05  RATE  OCCURS 50  TIMES  PIC 9(5).
```

the table element is repeated 50 times; thus, the table has 250 characters (50 occurrences of the 5-character elementary item called RATE).

You may use the OCCURS clause with group items:

```
05  RATE  OCCURS 50 TIMES.
    07  A-RATE    PIC 9(5).
    07  B-RATE    PIC 9(5).
```

This describes 500 characters (50 occurrences of A-RATE followed by B-RATE).

Format 2 defines variable-length tables. The values (integer-1 and integer-2) specified preceding the word DEPENDING define the limits of the table; the value of the data item immediately following DEPENDING (data-name-1) defines the specific size of the table at a given time. Thus:

```
01  TABLE.
    05  LENGTH-INDICATOR    PIC 9(2).
    05  ENTRIES      OCCURS 1 TO 50 TIMES
                      DEPENDING ON LENGTH-INDICATOR
                      PIC X(5).
```

This coding defines a table containing from 1 to 50 five-character entries, depending on the value of LENGTH-INDICATOR. So, when LENGTH-INDICATOR equals 20, the table has 20 entries, when LENGTH-INDICATOR equals 40, the table has 40 entries, etc. You must be certain that LENGTH-INDICATOR is never less than the table's lower limit (integer-1) or greater than its upper limit (integer-2).

An important item to note is that when the group item TABLE in the example is referenced, the *current* value of LENGTH-INDICATOR is used to determine the actual length of TABLE. This means it is the programmer's responsibility to set data-name-1 to the correct value before accessing a table or the previous value of data-name-1 is used. So far as this example is concerned, before any operation on the group item TABLE takes place, the programmer should be certain the value of LENGTH-INDICATOR represents the current number of occurences of the table elements.

To further clarify this point, this second example is presented:

```
01  WORK-AREA.
    02  COUNT    PIC  9(2).
    02  ELEMENT-1     PIC X(5).
    02  ELEMENT-2     PIC X(5).
    02  ELEMENT-3     PIC X(5).
01  TABLE.
    05  LENGTH-INDICATOR     PIC 9(2).
    05  ENTRIES      PIC X(5)
           OCCURS 1 TO 50 TIMES
           DEPENDING ON LENGTH-
               INDICATOR.
```

In this example if you want to move WORK-AREA to TABLE, you must first initialize the LENGTH-INDICATOR to the current number of occurrences of ENTRIES before you issue a MOVE. This could be done in the following manner:

```
MOVE 3 TO LENGTH-INDICATOR
MOVE WORK-AREA TO TABLE
```

If the first MOVE statement is omitted, which means LENGTH-INDICATOR is not initialized, the length of the receiving data item is determined by whatever the value of LENGTH-INDICATOR happens to be. This could jeopardize the original intent of the MOVE operation.

You do not have to define the data-name following DEPENDING within the same record as the table it is controlling (as in the example). But when you do, it must precede the data-name that has the OCCURS clause. Thus, the following coding is invalid:

```
01  TABLE.
    05  ENTRIES     OCCURS 1 TO 50 TIMES
                    DEPENDING ON LENGTH-INDICATOR
                    PIC X(5).
    05  LENGTH-INDICATOR  PIC 9(2).
```

When you use a Format 2 OCCURS clause in a data description entry, the entry may only be followed (within the same record description) by data description entries that are subordinate to it. Thus, the following coding is valid:

```
01  TABLE.
    05  A-TOTS  OCCURS 10 TO 25 TIMES
                DEPENDING ON COUNT.
        10  A-SUB-1    PIC 9(8).
        10  A-SUB-2    PIC 9(8).
```

The following coding, however, is invalid:

```
01  TABLE.
    05  A-TOTS  OCCURS 10 TO 25 TIMES
                DEPENDING ON COUNT.
        10  A-SUB-1   PIC 9(8).
        10  A-SUB-2   PIC 9(8).
    05  B-TOTS  PIC 9(16).
```

If you want to reference a table with a SEARCH ALL statement (8.4.3.3) (it executes faster than a SEARCH statement), you must arrange the table's elements in sequence according to the values of a key field. The ASCENDING/DESCENDING KEY phrase describes that sequence; it indicates the table elements are in ascending (if you code ASCENDING) or descending (if you code DESCENDING) sequence according to the values of specific fields (represented by data-name-2, data-name-3, etc.). Since the fields must be part of the table, they must be the subject of the OCCURS clause or subordinate to the subject of the OCCURS clause.

The ASCENDING KEY phrase in the following coding means the 50 values of TYPE-1 are arranged in ascending sequence:

```
01  A-RECD.
    02  CLASS-A    PIC 9(10).
    02  RATE   OCCURS 50 TIMES
                ASCENDING KEY IS TYPE-1.
        03  TYPE-1   PIC 9(5).
        03  TYPE-2   PIC 9(5).
```

In the coding, TYPE-1 (1) is less than TYPE-1 (2), TYPE-1 (2) is less than TYPE-1 (3), etc.

The data-name TYPE-1 is valid in the KEY phrase because it is subordinate to RATE – the subject of the OCCURS clause. You could not use CLASS-A in the KEY phrase, because it is not a table element.

In this example,

```
Ø1  SAMPLE
     Ø3  TYPE: OCCURS 25 TIMES
                ASCENDING KEY IS TYPE.
          Ø5  DIVISION  PIC X(5).
          Ø5  CLASS     PIC X(3).
          Ø5  CATEGORY  PIC X(7).
```

the subject of the OCCURS clause (TYPE) is the data-name used in the KEY phrase. Note that

```
ASCENDING KEY IS DIVISION, CLASS, CATEGORY
```

is equivalent to

```
ASCENDING KEY IS TYPE
```

Some other valid KEY phrases for this example, depending on the arrangement of your data, include

```
ASCENDING KEY IS CLASS
ASCENDING KEY IS CATEGORY, DIVISION
DESCENDING KEY IS DIVISION, CLASS
DESCENDING KEY IS CATEGORY
```

When the data-names in the KEY phrase are subordinate to the subject of the OCCURS clause (such as DIVISION, CLASS, and CATEGORY), none of them may contain an OCCURS clause and there must not be any entry containing an OCCURS clause between any of them and the subject (TYPE) of the OCCURS clause.

If you want to use indexing to reference the subject of an OCCURS clause and its subordinante items, you must use the INDEXED BY phrase (it associates an index with a specific table). Thus, in the TABLE-OF-MONTHS (8.1.1), the coding

```
INDEXED BY MONTH-INDEX
```

associates the index named MONTH-INDEX with the table.

Do not use OCCURS clauses to describe data items that have level-numbers 01, 66, 77, or 88; and do not use them to describe an item whose size is variable (that is, an item that has a subordinate item described with a Format 2 OCCURS clause).

8.2. REFERENCING TABLE ITEMS

Individual table items are referenced by adding subscripts or indexes to data-names described with OCCURS clauses. The subscripts or indexes represent occurrence numbers of table elements. For example, to reference a specific entry in the TABLE-OF-MONTHS, you code

```
MONTH (9)
```

where 9 is a subscript representing the ninth occurrence of MONTH in TABLE-OF-MONTHS. Thus, MONTH (9) equals

```
SEPTEMBER.
```

(Note that subscripts and indexes must be enclosed in parentheses.) Since an index (MONTH-INDEX) is associated with TABLE-OF-MONTHS, you also get the value SEPTEMBER by referencing MONTH (MONTH-INDEX), provided you set the value of MONTH-INDEX to 9. Methods of setting indexes are described in (8.4.3.2).

8.2.1. Subscripting

Subscripts reference individual elements from tables of like elements that do not have individual data-names. The format for subscripting is:

```
{data-name     } (subscript-1 [,subscript-2 [,subscript-3]])
{condition-name}
```

Subscripts must be integers – either a numeric literal that is an integer or a data-name that represents an integer. So, when SS equals 9,

```
MONTH (9)
```

and

```
MONTH (SS)
```

reference the same table element.

Data-names used as subscripts are not associated with particular tables; you can use them as subscripts when referencing any table in the program. Such data-names may be qualified, but not subscripted.

The lowest possible subscript value is 1; the highest is equivalent to the maximum number of occurrences of the item as specified in the associated OCCURS clause. Thus, MONTH in the TABLE-OF-MONTHS may be subscripted by the integers 1 through 12.

You can sign a subscript, but the sign must be positive.

8.2.2. Indexing

You use indexing, like subscripting, to reference individual elements within tables of like elements that do not have individual data-names. You associate an index-name (like MONTH-INDEX in the TABLE-OF-MONTHS) with a table by specifying it in an INDEXED BY phrase of the OCCURS clause. The value of an index corresponds to an occurrence number of an element in the associated table; the manner of that correspondence is determined by the implementor.

Here is the general format for indexing:

```
{data-name     }({index-name-1[{+} literal-2]}
{condition-name}({literal-1                  }
    [,{index-name-2 [{±} literal-4]}[,{index-name-3 [{±} literal-6]}]])
      {literal-3                   }  {literal-5                   }
```

Before you use an index-name to reference a table element, you must initialize it with a SET, SEARCH ALL, or PERFORM statement:

```
SET MONTH-INDEX TO 1
```

This initializes MONTH-INDEX to a value that corresponds to an occurrence number of 1. Then you can reference the table element with direct indexing (using an index-name like a subscript):

```
MONTH (MONTH-INDEX)
```

Another format is relative indexing (using an index-name plus a displacement value):

```
MONTH (MONTH-INDEX + 3)
```

The displacement value, which must be an unsigned integer, increases or decreases the occurrence number associated with the index. Thus, if MONTH-INDEX is set to 1,

```
MONTH (MONTH-INDEX + 3)
```

references the fourth occurrence of MONTH. Similarly, if MONTH-INDEX is set to 12,

```
MONTH (MONTH-INDEX - 4)
```

references the eighth occurrence of MONTH.

Make certain the occurrence value resulting from relative indexing is at least 1 and is no more than the maximum number of occurrences of the item as specified in the OCCURS clause.

Using literal-1, literal-3, and literal-5 in the format is functionally equivalent to using subscripting.

8.3. MULTIDIMENSIONAL TABLES

The tables described so far have one dimension. You can, however, code tables that have up to three dimensions. For example:

```
01  TABLE.
    03  DIVISION OCCURS 2 TIMES
        INDEXED BY DIV-INDEX.
         05  DEPARTMENT  OCCURS 3 TIMES
                  INDEXED BY DEPT-INDEX.
              07  CLASS  PIC X(1).
              07  CATEGORY  OCCURS 5 TIMES
                       INDEXED BY CAT-INDEX PIC X(2).
```

A reference to the third occurrence of CATEGORY, or CATEGORY (3), is meaningless without an indication of the DEPARTMENT and DIVISION with which CATEGORY is associated. So, to reference an occurrence of CATEGORY, you must include three subscripts or indexes:

```
CATEGORY (2, 1, 3)
```

Thus, the third occurrence of CATEGORY within the first occurrence of DEPARTMENT within the second occurrence of DIVISION. Similarly, a reference to DEPARTMENT includes two levels of subscripting or indexing:

```
DEPARTMENT (DIV-INDEX, DEPT-INDEX)
```

A reference to CLASS also requires two levels of subscripting or indexing to indicate the DEPARTMENT and DIVISION it is associated with.

You must write multiple subscripts or indexes in the order of successively less inclusive dimensions of the table. Thus,

```
CATEGORY (3, 1, 2)
```

is invalid because it references a nonexistent third occurrence of DIVISION.

The commas between the indexes or subscripts are optional, as is the space between the data-name and the left or right parenthesis.

When you use subscripting to reference multidimensional table elements, you may mix subscripts that are literals with subscripts that are data-names.

Suppose you want to use subscripting to reference items from the following table:

```
01  TABLE.
    03  CLASS  OCCURS  12  TIMES.
         05  RATE  OCCURS 30 TIMES.
              07  LIMIT  OCCURS 10 TIMES PIC 9(7).
```

You can reference the third occurrence of LIMIT within the 15th occurrence of RATE within the sixth occurrence of CLASS several ways. For example (assume CLASS-SS, RATE-SS, and LIMIT-SS are elementary data items with the integer values 6, 15, and 3, respectively):

```
LIMIT (6, 15, 3)
LIMIT (CLASS-SS, 15, 3)
LIMIT (CLASS-SS, 15, LIMIT-SS)
LIMIT (CLASS-SS, RATE-SS, LIMIT-SS)
```

As you can see, all references to LIMIT have three subscripts, because LIMIT is the innermost level of a 3-level table. Similarly, references to RATE contain two subscripts, as in RATE (5, 18) or RATE (CLASS-SS, RATE-SS). References to CLASS have just one subscript, as in CLASS (4) or CLASS (CLASS-SS).

In the next example, table elements are referenced with the indexes CLASS-INDEX, RATE-INDEX, and LIMIT-INDEX.

```
01  TABLE.
    03  CLASS OCCURS 12 TIMES
        INDEXED BY CLASS-INDEX.
        05  RATE OCCURS 30 TIMES
            INDEXED BY RATE-INDEX.
            07  LIMIT OCCURS 10 TIMES
                INDEXED BY LIMIT-INDEX PIC 9(7).
```

You reference an occurrence of LIMIT as:

```
LIMIT (CLASS-INDEX, RATE-INDEX, LIMIT-INDEX)
```

An occurrence of RATE as:

```
RATE (CLASS-INDEX, RATE-INDEX)
```

An occurrence of CLASS as:

```
CLASS (CLASS-INDEX)
```

Or, you can mix index-names and literals:

```
LIMIT (CLASS-INDEX, 5, 7)
```

8.4. TABLE LOOKUP

8.4.1. Coding Specific Elements

In the data division, you define tables and identify index-names associated with levels of tables. Now, in the procedure division, you code the statements that access the specific elements you need. Often you only need one statement. For instance, if your program includes the TABLE-OF-MONTHS and you want your program to print the name of the current month as the last word in a heading line such as:

```
PRODUCTION REPORT FOR
```

you code

```
MOVE MONTH (MONTH-IN) TO MONTH-OUT.
```

Assume MONTH-OUT represents the last data item in the heading line and MONTH-IN is an integer from 1 to 12 that is read from an input record. Then, if the value of MONTH-IN is 11, the heading line reads

```
PRODUCTION REPORT FOR NOVEMBER
```

You can see that using MONTH-IN as a subscript is simpler than testing for each month separately:

```
IF MONTH-IN =  1 MOVE ''JANUARY'' TO MONTH-OUT.
IF MONTH-IN =  2 MOVE ''FEBRUARY'' TO MONTH-OUT.
IF MONTH-IN =  3 MOVE ''MARCH'' TO MONTH-OUT.
     .
     .
     .
IF MONTH-IN = 12 MOVE ''DECEMBER'' TO MONTH-OUT.
```

When you use subscripting in routines that search a table for a specific element, you generally code a loop that accesses a table element and checks to see if it is the one needed; if it isn't, the routine increments the subscript and branches back to access the next element. When you use indexing, however, the SEARCH statement does the loop processing for you.

8.4.2. Table Lookup with Subscripting

Suppose your data division contains the following entries:

```
77  COUNT  PIC 9(2).
01  TABLE.
     02  INV-TABLE.
          03  FILLER PIC 9(5) VALUE ''15095''.
          03  FILLER PIC 9(5) VALUE ''26418''.
          03  FILLER PIC 9(5) VALUE ''09901''.
          03  FILLER PIC 9(5) VALUE ''18123''.
          03  FILLER PIC 9(5) VALUE ''42008''.
          03  FILLER PIC 9(5) VALUE ''12221''.
          03  FILLER PIC 9(5) VALUE ''24068''.
          03  FILLER PIC 9(5) VALUE ''51595''.
          03  FILLER PIC 9(5) VALUE ''72219''.
          03  FILLER PIC 9(5) VALUE ''77645''.
     02  FILLER REDEFINES INV-TABLE.
          03  ENTRIES OCCURS 10 TIMES
              INDEXED BY INV-INDEX.
               04  PART  PIC 9(2).
               04  STOCK PIC 9(3).
```

Then the following routine finds the amount of stock (STOCK) for a particular part (PART):

```
     MOVE 1 TO COUNT.
LOOP.
     IF INPUT-PART IS EQUAL TO PART (COUNT)
          MOVE STOCK (COUNT) TO PRINTLINE
          WRITE PRINTOUT
          GO TO ...
     ELSE ADD 1 TO COUNT.
     IF COUNT IS GREATER THAN 10 GO TO NO-MATCH
     ELSE GO TO LOOP.
```

The table elements, beginning with the first, are examined until the part number from an input record (INPUT-PART) matches the part number (PART) of a table entry. When it does, the corresponding stock amount (STOCK) is printed. Thus, if INPUT-PART equals 15, 095 appears on the printout. If all 10 table entries are examined and INPUT-PART and PART (COUNT) never match, control passes to an error routine called NO-MATCH.

8.4.3. Table Lookup with Indexing

To do the same processing as in 8.4.2 using indexing, you use the SET statement (8.4.3.2) to initialize INV-INDEX, then a SEARCH statement (8.4.3.3) to search the table for the element that has a part number equal to the one in the input record. Thus, if INPUT-PART equals 15

```
SET INV-INDEX TO 1.
SEARCH ENTRIES AT END GO TO NO-MATCH.
     WHEN INPUT-PART IS EQUAL TO PART (INV-INDEX)
     MOVE STOCK (INV-INDEX) TO PRINTLINE
     WRITE PRINTOUT.
```

Executing this coding prints 095 on the report. Again, if INPUT-PART never equals PART (INV-INDEX), control passes to NO-MATCH.

8.4.3.1. USAGE IS INDEX Clause

The USAGE IS INDEX clause allows the definition of elementary items that can be used to save index values. The items are called index data items. When you describe an index data item such as

```
77  SAVE-INDEX  USAGE IS INDEX.
```

you cannot include the PICTURE, VALUE, SYNCHRONIZED, JUSTIFIED, or BLANK WHEN ZERO clauses. The item (in this example SAVE-INDEX) simply contains a value that corresponds to an occurrence number of a table element. You establish that value by setting the index data items equal to the value of an index named in an INDEXED BY phrase of an OCCURS clause. For instance, if MONTH-INDEX is associated with a table, then

```
SET SAVE-INDEX TO MONTH-INDEX
```

makes SAVE-INDEX equal to the current value of MONTH-INDEX. Thus, the actual value and method of representing an index is determined by the implementor. Then you can change the value of MONTH-INDEX and eventually, if necessary, restore it to its original value by executing the coding

```
SET MONTH-INDEX TO SAVE-INDEX
```

You can write the USAGE IS INDEX clause at any level. If you write it at the group level, each elementary item in the group is an index data item, but the group itself cannot be referenced as an index data item. For example, if you code

```
01  A  USAGE IS INDEX.
     02  B.
     02  C.
     02  D.
```

then B, C, and D are index data items. But A is not; thus, the following is invalid:

```
SET A TO MONTH-INDEX
```

Index data items can be part of groups referred to in a MOVE statement. No conversion of the item occurs.

8.4.3.2. SET Statement

SET statements manipulate the values of indexes. There are two formats:

Format 1:

```
SET  { identifier-1 [, identifier-2] ... }  TO  { identifier-3 }
     { index-name-1 [, index-name-2] ... }      { index-name-3 }
                                                { integer-1    }
```

Format 2:

```
SET index-name-1 [, index-name-2] ...  { UP BY   }  { identifier-1 }
                                       { DOWN BY }  { integer-1    }
```

You use Format 1 to initialize the values of index-names, index data items, or integer data items. Index-names represent indexes defined in the INDEXED BY clause for a table; index data items are elementary data items described with the USAGE IS INDEX clause; and integer data items are elementary data items defined as integers.

The items initialized by SET statements are called receiving items and are coded to the left of the word TO in the format. The items to the right of TO are called sending items because their values are transferred to the receiving items, either directly, as if a simple MOVE statement were in effect, or after a conversion takes place to put the value into a format appropriate to the receiving item.

Sending items can be index-names, index data items, and integer data items (same as for receiving items) or they can be integers. However, you may or may not be able to use all four types of sending items, depending on the type of receiving item. The valid combinations of sending and receiving items are identitied in the following listing:

If Receiving Item (item to left of TO) is	Combinations of Receiving and Sending Items			
	Can Sending Item (item to right of TO) be:			
	index-name	index data item	integer data item	integer
index-name	Yes	Yes	Yes	Yes
index data items	Yes	Yes	No	No
integer data item	Yes	No	No	No

As you can see, when an index-name is the receiving item, you can use any of the four types of sending items. If the sending item is an index data item or if it is an index-name related to the same table as the receiving index-name, no conversion takes place as the value of the sending item is transferred to the receiving item (otherwise, a conversion is necessary). For example, if the sending item is an integer, the index-name is converted to a value that refers to the table element that corresponds to the occurrence number referenced by the integer (if the integer is signed, it must be positive). If MONTH-INDEX is associated with a table, the coding

```
SET MONTH-INDEX TO 1
```

sets it to a value corresponding to an occurrence number of 1.

When an index data item is the receiving item, you can set it equal to either the contents of an index-name or of another index data item; no conversion takes place in either case. You cannot set the value of an index data item to an integer or an integer data item. This statement

```
SET HOLD-INDEX TO RATE-INDEX
```

stores the current value of RATE-INDEX in the index data item called HOLD-INDEX.

If you use an integer data item as the receiving item, the sending item must be an index-name. The value of the integer data item is converted to an occurrence number that corresponds to the value of the index-name. In this example,

```
SET TAB-NUMB TO RATE-INDEX
```

the integer data item TAB-NUMB is set to an occurrence value that corresponds to the value of the index-name RATE-INDEX.

The Format 2 SET statements increment (if UP BY is used) or decrement (if DOWN BY is used) index-names by a value that corresponds to the number of occurrences represented by the value of identifier-1 or integer-1.

Identifier-1 must be described as an elementary numeric integer. Integer-1 may be signed (you may set an index up or down by a negative number).

In the following example, the index-names RATE-INDEX and CLASS-INDEX are initialized by a Format 1 SET statement, then incremented by a Format 2 SET statement.

```
SET RATE-INDEX TO 1.
SET CLASS-INDEX TO 3.
SET RATE-INDEX CLASS-INDEX UP By 1.
```

After these statements are executed, RATE-INDEX corresponds to an occurrence value of 2, and CLASS-INDEX corresponds to an occurrence value of 4.

8.4.3.3. SEARCH Statement

SEARCH statements examine individual table elements until predetermined conditions are true or until all table elements have been examined. There are two formats.

Format 1:

```
SEARCH identifier-1 [VARYING {identifier-2}]
                             {index-name-1}
   [; AT END imperative-statement-1 ]
    ; WHEN condition-1 {imperative-statement-2}
                       {NEXT SENTENCE         }
   [; WHEN condition-2 {imperative-statement-3}] ...
                       {NEXT SENTENCE         }
```

Format 2:

```
SEARCH ALL identifier-1 [; AT END imperative-statement-1]
   ;WHEN {data-name-1      {IS EQUAL TO} {identifier-3           }}
         {                {IS =       } {literal-1              }}
         {                              {arithmetic-expression-1}}
         {condition-name-1                                       }
     [AND {data-name-2      {IS EQUAL to} {identifier-4           }}] ...
          {                {IS =       } {literal-2              }}
          {                              {arithmetic-expression-2}}
          {condition-name-2                                       }
    {imperative-statement-2}
    {NEXT SENTENCE         }
```

If you code the following:

```
05  EMPLOYEE OCCURS 100 TIMES INDEXED BY EMP-INDEX.
    06  EMP-NO PIC 9(3).
    06  EMP-NAME PIC X(25).
```

A basic SEARCH statement such as follows examines the table elements in EMPLOYEE.

```
SEARCH EMPLOYEE
    WHEN NUM = EMP-NO (EMP-INDEX)
    MOVE EMP-NAME (EMP-INDEX) TO NAME-OUT.
```

The search begins with the element referenced by the current setting of the first (or only) index-name in the associated INDEXED BY phrase (in this case, EMP-INDEX). So, if EMP-INDEX is set to 1, the search begins with EMPLOYEE (1). The table elements are examined one at a time, in sequence; i.e., EMPLOYEE (1), EMPLOYEE (2), ... EMPLOYEE (100), until NUM = EMP-NO (EMP-INDEX) or until EMPLOYEE (100) is tested and found not to satisfy the condition NUM = EMP-NO (EMP-INDEX).

The conditions you use can be any conditions described as conditional expressions (7.2.2). You can specify more than one condition.

After each table element is examined, the occurrence number represented by the index-name is incremented by 1. Other index-names assigned in the associated INDEXED BY phrase are not incremented. After the search, the index-name remains set at the occurrence that caused the condition to be true.

Thus, if

```
NUM = EMP-NO (EMP-INDEX)
```

is true when EMP-INDEX is 60, EMP-INDEX equals 60 when the search is completed.

The search ends when one of the specified conditions is true or when the at-end-condition occurs. The appropriate imperative statement is executed and control passes to the next executable statement following the SEARCH statement, unless you direct control elsewhere by ending your imperative statement with a GO TO instruction. If an at-end-condition occurs and you did not use the AT END phrase in your SEARCH statement, control passes to the next executable statement following the SEARCH statement.

The VARYING phrase manipulates indexes. If the index-name in the VARYING phrase is the same as one that appears in the table's INDEXED BY phrase, then that index-name is used in the search operation. So, if your table is described as

```
05  EMPLOYEE OCCURS 100 TIMES
    INDEXED BY EMP-INDEX NAME-INDEX.
    06  EMP-NO PIC 9(3).
    06  EMP-NAME PIC X(25).
```

and you want to increment NAME-INDEX rather than EMP-INDEX during the search, you code

```
SEARCH EMPLOYEE VARYING NAME-INDEX...
```

and the current value of EMP-INDEX is unchanged by the search operation.

If the index-name in the VARYING phrase is associated with a different table, the first (or only) index-name associated with the table to be searched (EMP-INDEX in the example) is used in the search operation (just as if you did not use the VARYING phrase), and the occurrence number represented by the index-name specified in the VARYING phrase is incremented by the same amount as (and at the same time as) the index-name used in the search.

Thus, if you code

```
SEARCH EMPLOYEE VARYING TABLE-INDEX
```

to search the table described in the previous example, EMP-INDEX is used for the search and TABLE-INDEX is incremented by the same amount as, and at the same time as, EMP-INDEX. Thus, if EMP-INDEX and TABLE-INDEX are equal before the search, they are still equal after the search.

If you specify identifier-2 in the VARYING phrase, the first (or only) index-name associated with the table to be searched is again used in the search operation and, if identifier-2 is an index data item, the index-name and the index data item are incremented by the same amount and at the same time. If identifier-2 represents an elementary item that is an integer, it is incremented by the value 1 at the same time as index-name is incremented.

If your data division includes

```
77  HOLD-INDEX  USAGE IS INDEX.
01  TABLE-1.
    03  RATE-TABLE OCCURS 75 TIMES
        INDEXED BY RATE-INDEX.
        05  RATE-CAT  PIC X(5).
        05  RATE-NUM  PIC 9(7)V9(2).
```

you can search RATE-TABLE by coding

```
SEARCH RATE-TABLE VARYING HOLD-INDEX
    AT END GO TO CALC-ROUTINE
    WHEN RATE-CAT (RATE-INDEX) = CARD-CAT
        MOVE RATE-NUM (RATE-INDEX) TO CURRENT-RATE
    WHEN RATE-CAT (RATE-INDEX) = CARD-CANC-CAT
        PERFORM CANC-ROUTINE.
```

The entries in RATE-TABLE are examined beginning with RATE-TABLE (1) and continue in a serial manner until either the end of the table is reached, causing execution of the statement associated with AT END (GO TO CALC-ROUTINE), or until the condition specified for either of the WHEN phrases is true, causing (in this case) execution of MOVE RATE-NUM (RATE-INDEX) TO CURRENT-RATE or PERFORM CANC-ROUTINE.

As each successive entry in the table is examined, the occurrence numbers associated with both RATE-INDEX and HOLD-INDEX are incremented. HOLD-INDEX is an elementary data item described with the USAGE IS INDEX clause. After execution of the statement associated with either of the WHEN phrases, control passes to the next executable statement following the SEARCH statement.

Format 2 SEARCH statements provide for nonserial searches; that is, the table elements are not examined in sequence beginning with the element referenced by the current index setting. Instead, the initial setting of the index used in the search operation is ignored and the setting is varied during the search in a manner specified by the implementor. The purpose is to reduce the number of table accesses needed to find the element that makes the specified conditions true.

The nonserial search is possible only if table elements are arranged in sequence according to the values of key fields. You identify these fields in the ASCENDING/DESCENDING KEY phrase of the OCCURS clause that describes the table being searched.

The WHEN phrase differs from Format 1 in that instead of providing several conditions to test, any of which would end the search if true, it includes only one WHEN phrase that may test several conditions, all of which must be true to end the search.

If you describe a table as

```
05  RATE-TABLE OCCURS 50 TIMES
      ASCENDING KEY IS RATE-CAT RATE-DIV
      INDEXED BY RATE-INDEX.
      10  RATE-CAT  PIC X(3).
      10  RATE-DIV  PIC X(5).
```

you can search it with a SEARCH ALL statement such as

```
SEARCH ALL RATE-TABLE
      AT END GO TO NOHIT-ROUTINE
      WHEN RATE-CAT (RATE-INDEX) = CARD-CAT
      AND RATE-DIV (RATE-INDEX) = CARD-DIV
      PERFORM TABLE-HIT-ROUTINE.
```

Since the initial value of RATE-INDEX is ignored, you do not have to set it before the search. The search continues until both conditions specified in the WHEN phrase are true for a specific setting of RATE-INDEX; then, the imperative statement PERFORM TABLE-HIT-ROUTINE is executed and control passes to the statement following the SEARCH statement (again, unless you direct control elsewhere by ending your imperative statement with a GO TO statement).

If no index setting makes the conditions true, the imperative statement specified in the AT END phrase is executed (in this case passing control to NOHIT-ROUTINE). If no AT END phrase is specified, control passes to the statement following the SEARCH statement.

In the WHEN phrase, the data-names (or condition-names associated with the data-names) you use must be named in the ASCENDING/DESCENDING KEY phrase associated with the table. Any condition-names referenced must have only a single value. Identifier-2, identifier-3, or identifiers specified in arithmetic-expression-1 or arithmetic-expression-2 must not be referenced in the KEY phrase or be indexed by the first index-name associated with the table.

You cannot use a data-name from an ASCENDING/DESCENDING KEY phrase unless you also use all the preceding data-names in the same KEY phrase. Thus, in the example, RATE-CAT must be included in the WHEN phrase if RATE-DIV. is used. The reverse, however, is not true; you may use RATE-CAT without RATE-DIV.

You cannot use the VARYING phrase with Format 2; thus, the index-name used in the search operation always is the first (or only) index-name that appears in the INDEXED BY phrase for the table. As the value of that index-name changes during the search, the values of other index-names associated with the table remain unchanged. After the search operation, the index-name remains set at the occurrence that makes the conditons true. If no setting makes the conditions true, the final setting of the index-name is unpredictable.

The results of SEARCH ALL operations are predictable only when the table data is arranged in the order described by its assoicated ASCENDING/DESCENDING KEY phrase, and the contents of the keys referenced in the WHEN phrase are sufficient to identify a unique table element.

8.5. TABLE HANDLING EXAMPLES

The two table handling examples included accomplish the same task - the first using indexing; the second using subscripting.

A table (STUDENT-TABLE) in the program is accessed two ways. First, a routine (GPA-LOOP) checks each entry in the table and prints the social security number field (STUDENT-SS-NO) of any entry that has a grade point average field (STUDENT-GPA) equal to 3.50 or more. A second routine (READ-CARD) reads a card that has a social security number field (CARD-SS-NO) and, using the SEARCH verb, looks in the table for a matching social security number (STUDENT-SS-NO). When there is a match, the social security number (STUDENT-SS-NO) and grade point average (STUDENT-GPA) are printed. Otherwise, the words INVALID INPUT are printed.

A card deck used as input to these programs might include the following data:

```
151325164
159440701
161250524
146325214
158413022
132546589
162054785
144320205
```

The printed output then, is as follows:

```
159440701
161250524
168232051
162054785
132458797

* * * * *

151325164       3.40
159440701       3.67
161250524       3.85
146325214       3.00
INVALID INPUT
132546589       2.47
162054785       3.65
144320205       3.12
```

The table handling routines using indexing and subscripting follow:

Table handling using indexing:

```
00001          IDENTIFICATION DIVISION.
00002          PROGRAM-ID.     XINDEX.
00003          AUTHOR.         FUNDS OF COBOL.
00004          ENVIRONMENT DIVISION.
00005          CONFIGURATION SECTION.
00006          SOURCE-COMPUTER.  UNIVAC-OS3.
00007          OBJECT-COMPUTER.  UNIVAC-OS3.
00008          INPUT-OUTPUT SECTION.
00009          FILE-CONTROL.
00010              SELECT CARD-FILE, ASSIGN TO CARDREADER-CDRDR-F.
00011              SELECT PRINT-FILE, ASSIGN TO PRINTER-PTR-FC.
00012          DATA DIVISION.
00013          FILE SECTION.
00014          FD  CARD-FILE
00015              LABEL RECORDS ARE OMITTED
00016              RECORD CONTAINS 80 CHARACTERS
00017              DATA RECORD IS CARD-RECORD.
00018          01  CARD-RECORD.
00019              05  CARD-SS-NO     PIC X(9).
00020              05  FILLER         PIC X(71).
00021          FD  PRINT-FILE
00022              LABEL RECORDS ARE OMITTED
00023              RECORD CONTAINS 20 CHARACTERS
00024              DATA RECORD IS PRINT-LINE.
```

(continued)

```
00025        01  PRINT-LINE.
00026            05  PRINT-SS-NO    PIC X(9).
00027            05  FILLER         PIC X(7).
00028            05  PRINT-GPA      PIC 9.99.
00029        WORKING-STORAGE SECTION.
00030        77  STUDENT-SUB        PIC 9(2)  USAGE  COMPUTATIONAL.
00031        01  STUDENT-TABLE.
00032            05  TABLE-ENTRIES.
00033                10  FILLER  PIC 9(12)   VALUE  159440701367.
00034                10  FILLER  PIC 9(12)   VALUE  144320205312.
00035                10  FILLER  PIC 9(12)   VALUE  132546589247.
00036                10  FILLER  PIC 9(12)   VALUE  161250524385.
00037                10  FILLER  PIC 9(12)   VALUE  168232051375.
00038                10  FILLER  PIC 9(12)   VALUE  162054785365.
00039                10  FILLER  PIC 9(12)   VALUE  158423021247.
00040                10  FILLER  PIC 9(12)   VALUE  151325164340.
00041                10  FILLER  PIC 9(12)   VALUE  132458797376.
00042                10  FILLER  PIC 9(12)   VALUE  146325214300.
00043            05  FILLER REDEFINES TABLE-ENTRIES.
00044                10  TABLE-OF-STUDENTS   OCCURS 10 TIMES
00045                                        INDEXED BY STUDENT-INDEX.
00046                    15  STUDENT-SS-NO   PIC 9(9).
00047                    15  STUDENT-GPA     PIC 9(1)V9(2).
00048        PROCEDURE DIVISION.
00049        START-PROG.
00050            OPEN INPUT CARD-FILE, OUTPUT PRINT-FILE.
00051            MOVE SPACES TO PRINT-LINE.
00052        FIND-HIGH-GPA.
00053            SET STUDENT-INDEX TO 1.
00054        GPA-LOOP.
00055            IF STUDENT-GPA (STUDENT-INDEX) IS NOT LESS THAN 3.50
00056            MOVE STUDENT-SS-NO (STUDENT-INDEX) TO PRINT-SS-NO
00057            WRITE PRINT-LINE AFTER ADVANCING 2 LINES
00058            MOVE SPACES TO PRINT-LINE.
00059        IF STUDENT-INDEX IS EQUAL TO 10 NEXT SENTENCE
00060        ELSE SET STUDENT-INDEX UP BY 1, GO TO GPA-LOOP.
00061        MOVE ''* * * * *'' TO PRINT-LINE.
00062            WRITE PRINT-LINE AFTER ADVANCING 3 LINES.
00063            MOVE SPACES TO PRINT-LINE.
00064        READ-CARDS.
00065            READ CARD-FILE, AT END GO TO END-OF-JOB.
00066            SET STUDENT-INDEX TO 1.
00067            SEARCH TABLE-OF-STUDENTS, AT END PERFORM NOHIT
00068                WHEN STUDENT-SS-NO (STUDENT-INDEX) IS EQUAL TO CARD-SS-NO
00069                MOVE STUDENT-SS-NO (STUDENT-INDEX) TO PRINT-SS-NO
00070                MOVE STUDENT-GPA (STUDENT-INDEX) TO PRINT-GPA
00071                WRITE PRINT-LINE AFTER ADVANCING 2 LINES
00072                MOVE SPACES TO PRINT-LINE.
00073            GO TO READ-CARDS.
```

(continued)

```
00074          NOHIT.
00075              MOVE ''INVALID INPUT'' TO PRINT-LINE.
00076              WRITE PRINT-LINE AFTER ADVANCING 2 LINES.
00077              MOVE SPACES TO PRINT-LINE.
00078          END-OF-JOB.
00079              CLOSE CARD-FILE, PRINT-FILE.
00080              STOP RUN.
```

Table handling using subscripting:

```
00001          IDENTIFICATION DIVISION.
00002          PROGRAM-ID.  SSCRIPT.
00003          AUTHOR.      FUNDS OF COBOL.
00004          ENVIRONMENT DIVISION.
00005          CONFIGURATION SECTION.
00006          SOURCE-COMPUTER.  UNIVAC-OS3.
00007          OBJECT-COMPUTER.  UNIVAC-OS3.
00008          INPUT-OUTPUT SECTION.
00009          FILE-CONTROL.
00010              SELECT CARD-FILE, ASSIGN TO CARDREADER-CDRDR-F.
00011              SELECT PRINT-FILE, ASSIGN TO PRINTER-PTR-FC.
00012          DATA DIVISION.
00013          FILE SECTION.
00014          FD CARD-FILE
00015              LABEL RECORDS ARE OMITTED
00016              RECORD CONTAINS 80 CHARACTERS
00017              DATA RECORD IS CARD-RECORD.
00018          01  CARD-RECORD.
00019              05  CARD-SS-NO     PIC X(9).
00020              05  FILLER         PIC X(71).
00021          FD  PRINT-FILE
00022              LABEL RECORDS ARE OMITTED
00023              RECORD CONTAINS 20 CHARACTERS
00024              DATA RECORD IS PRINT-LINE.
00025          01  PRINT-LINE.
00026              05  PRINT-SS-NO    PIC X(9).
00027              05  FILLER         PIC X(7).
00028              05  PRINT-GPA      PIC 9.99.
00029          WORKING-STORAGE SECTION.
00030          77  STUDENT-SUB        PIC 9(2)  USAGE  COMPUTATIONAL.
00031          01  STUDENT-TABLE.
00032              05  TABLE-ENTRIES.
00033                  10  FILLER  PIC 9(12)   VALUE   159440701367.
00034                  10  FILLER  PIC 9(12)   VALUE   144320205312.
00035                  10  FILLER  PIC 9(12)   VALUE   132546589247.
00036                  10  FILLER  PIC 9(12)   VALUE   161250524385.
00037                  10  FILLER  PIC 9(12)   VALUE   168232051375.
00038                  10  FILLER  PIC 9(12)   VALUE   162054785365.
```

(continued)

```
00039                10  FILLER  PIC 9(12)   VALUE   158423021247.
00040                10  FILLER  PIC 9(12)   VALUE   151325164340.
00041                10  FILLER  PIC 9(12)   VALUE   132458707376.
00042                10  FILLER  PIC 9(12)   VALUE   146325214300.
00043            05  FILLER REDEFINES TABLE-ENTRIES.
00044                10  TABLE-OF-STUDENTS   OCCURS  10 TIMES
00045                                        INDEXED BY STUDENT-INDEX.
00046                    15  STUDENT-SS-NO   PIC 9(9).
00047                    15  STUDENT-GPA     PIC 9(1)V9(2).
00048        PROCEDURE DIVISION.
00049        START-PROG.
00050            OPEN INPUT CARD-FILE, OUTPUT PRINT-FILE.
00051            MOVE SPACES TO PRINT-LINE.
00052            MOVE 1 TO STUDENT-SUB.
00053        GPA-LOOP.
00054            IF STUDENT-GPA (STUDENT-SUB) IS NOT LESS THAN 3.50
00055            MOVE STUDENT-SS-NO (STUDENT-SUB) TO PRINT-SS-NO
00056                WRITE PRINT-LINE AFTER ADVANCING 2 LINES
00057                MOVE SPACES TO PRINT-LINE.
00058            IF STUDENT-SUB IS EQUAL TO 10 NEXT SENTENCE
00059            ELSE ADD 1 TO STUDENT-SUB, GO TO GPA-LOOP.
00060            MOVE ''* * * * *'' TO PRINT-LINE.
00061            WRITE PRINT-LINE AFTER ADVANCING 3 LINES.
00062            MOVE SPACES TO PRINT-LINE.
00063        READ-CARDS.
00064            READ CARD-FILE, AT END GO TO END-OF-JOB.
00065            MOVE 1 TO STUDENT-SUB.
00066        LOOP.
00067            IF STUDENT-SUB IS EQUAL TO 11 GO TO NOHIT.
00068            IF STUDENT-SS-NO (STUDENT-SUB) IS EQUAL TO CARD-SS-NO
00069            MOVE STUDENT-SS-NO (STUDENT-SUB) TO PRINT-SS-NO
00070            MOVE STUDENT-GPA (STUDENT-SUB) TO PRINT-GPA
00071            WRITE PRINT-LINE AFTER ADVANCING 2 LINES
00072            ELSE ADD 1 TO STUDENT-SUB, TO TO LOOP.
00073            GO TO READ-CARDS.
00074        NOHIT.
00075            MOVE ''INVALID INPUT'' TO PRINT-LINE.
00076            WRITE PRINT-LINE AFTER ADVANCING 2:LINES.
00077            MOVE SPACES TO PRINT-LINE.
00078            GO TO READ-CARDS.
00079        END-OF-JOB.
00080            CLOSE CARD-FILE, PRINT-FILE.
00081            STOP RUN.
```

9. Sort/Merge

When solving data processing problems, you often need to rearrange records into a particular order. Many operating systems provide you with a utility program to accomplish this task outside of your COBOL program. By using the COBOL sort/merge feature, however, you can solve this problem with instructions included inside your program. The instructions allow you to sort (rearrange the sequence of records in a file or files) or merge (combine two or more identically sequenced files) one or more times within a given execution of the program.

You can code your sort/merge operation in one of two ways:

1. Code one statement that
   - provides the input records;
   - sorts or merges them; and
   - creates a sorted or merged output file.
2. Specify your own sort/merge input and output procedures, applying some special processing to the individual records before and after sorting or after merging.

When you use the sort/merge feature, you must define a sort/merge file that functions as a work file; it receives the records from the file or files that are to be sorted or merged and (when the sort/merge operation is completed) is the source of the rearranged records passed to an output file.

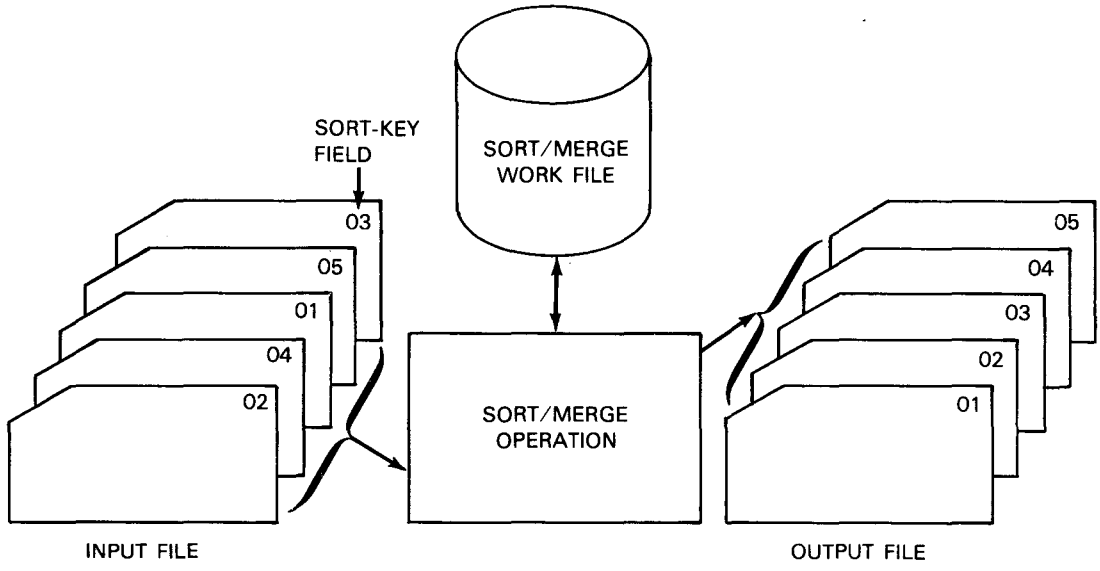
9.1. SORT/MERGE OPERATION

SORT or MERGE statements in the procedure division identify the sort/merge work files used, the key fields on which records are sorted or merged, the collating sequences used, and either (1) the names of the files that contain the records you want to sort or merge and the file that is to receive the reordered records, or (2) the names of the paragraphs that contain your input and output procedures.

When SORT or MERGE statements are executed, the following general sequence of procedures occurs:

- The records become input for the sort/merge operation, either from the file or files you specified in the USING option of the SORT or MERGE statement, or as a result of the input procedure you coded for the sort statement. The sort/merge operation moves the records to the sort/merge work file.
- The records are sorted or merged according to the keys and collating sequence you selected.
- The records, in their new sequence, are moved from the sort/merge work file to the output file you designated either in the GIVING option of the SORT or MERGE statement, or as a result of output procedure.
- The operation of the SORT or MERGE statement terminates and control passes to the next statement in the program.

Suppose you want to resequence an input card file based on ascending values of card columns 1 and 2 (call that field SORT-KEY). The sort operation, triggered by the SORT statement, looks like the following:



The sort/merge operation moves the input records to the sort/merge work file, sorts them in ascending SORT-KEY sequence, and punches output records in the new sequence.

9.2. DEFINING SORT/MERGE FILE

You must identify sort/merge files in the FILE-CONTROL paragraph of the environment division, and describe them in the FILE SECTION of the data division – just as you did for the input-output files. The entries are a little different, however, because sort/merge files are work files; thus, the optional clauses that apply specifically to input and output processing are invalid.

In the FILE-CONTROL paragraph, the sort/merge files are named and associated with storage mediums. The format is:

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

The SELECT clause names the file; the ASSIGN clause associates the file with a storage medium. Each file-name must be unique and must be named in a corresponding sort or merge file description (SD) entry in the data division. That entry is similar to the file description (FD) entries you code for input-output files. The format is:

```
SD file-name
    [;RECORD CONTAINS[integer-1 TO] integer-2 CHARACTERS]
    [;DATA{RECORD IS  } data-name-1 [,data-name-2] ... ].
          {RECORDS ARE}
```

The level indicator SD identifies the beginning of the sort/merge work file description and is immediately followed by a file-name that must match a file-name in a SELECT clause. One or more record description entries always must follow the sort/merge file description entry.

The two optional clauses shown are for documentation only; however, if you use the RECORD CONTAINS clause to specify record size, the size you specify must match the actual size of the records as described in the record descriptions associated with the SD. Use integer-2 by itself to specify the record size if all the records in the file are the same size. If the file consists of variable length records, specify the smallest (integer-1) and largest (integer-2) size record.

The DATA RECORDS clause names the records in the file. The names must be the same as in the 01 level number record descriptions associated with the SD.

The following is coded in the environment division and the data division:

1. Environment Division

```
FILE-CONTROL.
    SELECT MY-SORT-FILE ASSIGN TO DISC.
    SELECT BEFORE-THE-SORT ASSIGN TO TAPE.
    SELECT AFTER-THE-SORT ASSIGN TO TAPE.
```

2. Data Division

```
FILE SECTION.
SD MY-SORT-FILE
     RECORD CONTAINS 95 CHARACTERS
     DATA RECORD IS MY-SORT-RECORD.
Ø1 MY-SORT-RECORD.
     Ø2  SORT-KEY   PIC X(5).
     Ø2  FILLER     PIC X(9Ø).
FD BEFORE-THE-SORT
   LABEL RECORDS ARE OMITTED.
Ø1 BEFORE-SORT-RECORD PIC X(95).
FD AFTER-THE-SORT
   LABEL RECORDS ARE OMITTED.
Ø1 AFTER-SORT-RECORD PIC X(95).
```

The sort/merge work file is called MY-SORT-FILE and is assigned to a device with implementor-name DISC. It consists of 95-character records called MY-SORT-RECORD. An input-output file containing the unsorted records (BEFORE-THE-SORT) and another file to receive the sorted records (AFTER-THE-SORT) also are identified. Note that the file records of the input-output and the sort/merge work file records must be the same size. Both input-output files are assigned to a device with implementor-name TAPE.

9.3. SORT/MERGE STATEMENTS

There are four statements you can use when you code a sort/merge operation in the procedure division. You always use either a SORT or MERGE statement and, if you specify your own input and output procedures, you use the RELEASE and RETURN statements. These statements are described in 9.3.1 through 9.3.4.

9.3.1. SORT Statement

SORT statements rearrange the sequence of records in a file or files. The format is:

```
SORT file-name-1 ON {ASCENDING } KEY data-name-1 [,data-name-2] ...
                    {DESCENDING}
                 [ ON {ASCENDING } KEY data-name-3 [,data-name-4] ...]...
                      {DESCENDING}
    [COLLATING SEQEUNCE IS alphabet-name]
    {INPUT PROCEDURE IS section-name-1[{THROUGH} section-name-2]}
    {                                 {THRU   }                 }
    {USING file-name-2 [,file-name-3]...                        }
    {OUTPUT PROCEDURE IS section-name-3[{THROUGH} section-name-4]}
    {                                  {THRU   }                }
    {GIVING file-name-4                                         }
```

File-name-1 is the name of the sort/merge work file described in a sort/merge file description (SD) entry. File-name-2, file-name-3, etc. represent the files to be sorted; file-name-4 receives the reordered records.

Although the resequencing of records is based on the values of fields in input files records, sort keys are identified as fields in sort/merge work file records. So, to sort the file

```
FD  PAY-FILE
    LABEL RECORDS ARE OMITTED.
Ø1  PAY-RECORD.
    Ø2  EMPL-NO  PIC X(5).
    Ø2  FILLER   PIC X(9Ø).
```

into EMPL-NO sequence, a field must be described in the sort/merge work file record that corresponds to EMPL-NO:

```
SD  SORT-FILE
Ø1  SORT-RECD.
    Ø2  EMPL-SORT  PIC X(5).
    Ø2  FILLER     PIC X(9Ø).
```

Now you can use EMPL-SORT as the sort key.

Sort keys are named in the ASCENDING/DESCENDING KEY clause of SORT statements. As expected, the ASCENDING phrase is for sequencing records from the lowest to the highest values of the key fields and the DESCENDING phrase is for sequencing records from the highest to the lowest values of the key fields.

If more than one key is used, they must be specified in decreasing order of significance. If you describe the sort/merge work file as

```
SD  MY-SORT FILE.
Ø1  SORT-RECD.
    Ø2  A  PIC X(7).
    Ø2  B  PIC X(3).
    Ø2  C  PIC X(5).
    Ø2  FILLER  PIC X(8Ø).
```

and you want the sort to be based first on the ascending values of the field B, second on the descending values of field A, and third on the ascending values of field C, code as follows:

```
SORT MY-SORT-FILE
     ON ASCENDING KEY B
     ON DESCENDING KEY A
     ON ASCENDING KEY C ...
```

If the SD file has more than one record description, you need to describe the keys in only one of them. The keys cannot be variable length data items.

You can write input and output procedures that transfer records to and from the sort/merge work file, or you can code USING and GIVING phrases that transfer them automatically. Remember that you must describe the input-output files in data division file description (FD) entries, not sort/merge file description (SD) entries, and the input-output file records must be the same size as the sort/merge work file records.

When you use the USING or GIVING phrases, make sure the associated input-output files are not open when the SORT statement is executed. The sort/merge operation automatically opens and closes the input files as needed, including performing all implicit functions such as the execution of any associated USE procedures. When the sort/merge operation is completed, the files are closed as if a CLOSE statement, without optional phrases, was executed for each file.

If the input and output files are

```
FD  BEFORE-THE SORT
    LABEL-RECORDS ARE OMITTED.
Ø1  BEFORE-SORT-RECORD  PIC X(95).
FD  AFTER-THE-SORT
    LABEL RECORDS ARE OMITTED.
Ø1  AFTER-SORT-RECORD  PIC X(95).
```

and the sort/merge work file is

```
SD  MY-SORT-FILE.
Ø1  MY-SORT-RECORD.
    Ø2  SORT-KEY  PIC X(5).
    Ø2  FILLER    PIC X(9Ø).
```

then

```
SORT MY-SORT-FILE
     ON ASCENDING KEY SORT-KEY
     USING BEFORE-THE SORT
     GIVING AFTER-THE-SORT.
```

thus opens the files, moves the records from BEFORE-THE-SORT to MY-SORT-FILE, sorts them, moves the sorted record to AFTER-THE-SORT, and closes the files.

Note that none of the key data items can be described by an entry that contains an OCCURS clause, or is subordinate to an entry that contains an OCCURS clause.

You code input procedures in one or more contiguous sections that are used exclusively as an input procedure for an associated sort operation. You identify these sections in the INPUT PROCEDURE phrase. Do not pass control to any point in the input procedure except via a SORT statement and do not code any statement, including SORT and MERGE statements, within the input procedure that passes control to a point outside the procedure.

In the input procedure, use OPEN and READ statements to obtain the records from input files. Process the records as needed, and then use the RELEASE statement to transfer the records to the sort operation. When all input records are released, close the input files with a CLOSE statement, which normally ends the input procedure. The actual sorting does not begin until the last statement in the input procedure is executed. At that point, all of the records transferred by the execution of the RELEASE statement, are contained in the sort/merge work file.

When coding the output procedure, use the OUTPUT PROCEDURE phrase to indicate the section or sections that contain it. As with input procedures, the sections must be contiguous and must be used only for the sort operation; you cannot pass control to the sections containing the output procedure except by execution of an associated SORT statement, and the output procedure cannot transfer control to points outside the procedure.

NOTE:

In both input and output procedures, COBOL statements that may cause a transfer of control to the declaratives (OPEN, READ, WRITE, etc.) are permitted.

Output procedures normally include OPEN, RETURN (9.3.4), WRITE, and CLOSE statements to open the output file, obtain the ordered records from the sort operation, write the records, and when all records are written, close the output file. As you obtain the records from the sort/merge work file, you can apply special processing as needed. You cannot use SORT or MERGE statements in output procedures.

SORT operations may have multiple input files, but only one is required. You may name only one output file. In the SORT statement, you cannot repeat a file-name or specify more than one file from a multiple file reel.

If you want your sort operation to be guided by a collating sequence other than the program collating sequence (5.2), specify a collating sequence in the alphabet-name clause of the SPECIAL-NAMES paragraph and reference the alphabet-name in the COLLATING SEQUENCE phrase of the SORT statement. The new collating sequence is in effect only for that SORT statement.

9.3.2. MERGE Statement

MERGE statements combine two or more files that already are identically sequenced. Although you can use SORT statements to merge files, MERGE statements execute faster. The format is:

```
MERGE file-name-1 ON {ASCENDING } KEY data-name-1 [,data-name-2]...
                     {DESCENDING}
                   [ON {ASCENDING } KEY data-name-3 [,data-name-4]...]...
                       {DESCENDING}
    [COLLATING SEQUENCE IS alphabet-name]
     USING file-name-2, file-name-3 [,file-name-4]...
    {OUTPUT PROCEDURE IS section-name-1 [{THROUGH} section-name-2]}
    {                                   {THRU   }                 }
    {GIVING file-name-5                                           }
```

The coding for MERGE statements is almost identical to that for SORT statements. There are two differences:

1. MERGE statements require more than one input file.

2. When using merge statements, you cannot code your own input procedure; you must name the input files in a USING phrase.

If you code

```
MERGE MY-SORT-FILE
    ON ASCENDING KEY MY-SORT-KEY-1 MY-SORT-KEY-2
    USING INFILE-1 INFILE-2
    GIVING OUT-FILE.
```

the records in INFILE-1 and INFILE-2 are merged together into a new file – OUT-FILE. INFILE-1, INFILE-2, and OUT-FILE are described in FD entries in the data division; MY-SORT-FILE is described in a data division SD entry; and MY-SORT-KEY-1 and MY-SORT-KEY-2 are data items named in the record description associated with MY-SORT-FILE.

9.3.3. RELEASE Statement

You use RELEASE statements in input procedures associated with SORT statements to transfer records to the initial phase of a sort operation. The format is:

```
RELEASE record-name[FROM identifier]
```

The execution of a RELEASE statement releases the record named by record-name to the sorting operation. Record-name, which may be qualified, must be associated with a sort/merge work file description (SD) entry. Note, however, that it is your responsibility to move what you want to sort – usually a record from an input file – to record-name.

For example, suppose you want to read a card file. Select all records that have a 1 in card column 80 (we'll assign card column 80 the data-name CC-80), and include the records in an output file sorted according to the keys specified in the SORT statement (not shown). For example:

```
INPUT-PROCEDURE SECTION.
IN-PROC.
    READ CARD-FILE AT END GO TO END-IN-PROC.
    IF CC-80 IS EQUAL TO "1"
        MOVE CARD-RECORD TO SORT-RECORD
    ELSE GO TO IN-PROC.
    RELEASE SORT-RECORD.
    GO TO IN-PROC.
END-IN-PROC.
```

As you can see, the contents of the record area (CARD-RECORD) for the card file are moved to the record area (SORT-RECORD) associated with the sort/merge work file, then released to the sort operation.

You can avoid coding MOVE statements in input procedures by using the FROM option of the RELEASE statement. When you use the FROM phrase, the contents of an identifier are moved to record-name, then the contents of record-name are released to the sort operation. Record-name and identifier in the format must not refer to the same storage area. The move takes place according to the rules specified for the MOVE statement without the CORRESPONDING phrase. Coding the example using the FROM option looks like this:

```
INPUT-PROCEDURE SECTION.
IN-PROC.
    READ CARD-FILE AT END GO TO END-IN-PROC.
    IF CC-80 IS EQUAL TO "1"
        RELEASE SORT-RECORD FROM CARD-RECORD.
    GO TO IN-PROC.
END-IN-PROC.
```

After execution of a RELEASE statement with the FROM option, you can still access the data area associated with identifier, but you cannot access the data area associated with record-name unless you named the associated sort/merge work file in a SAME RECORD AREA clause (9.4). In that case, you can still access the data in record-name and in the record area of other files referenced in your SAME RECORD AREA clause.

When control passes from an input procedure, the associated sort/merge work file consists of all those records placed in it by the execution of RELEASE statements.

9.3.4. RETURN Statement

RETURN statements are similar to RELEASE statements, but they are associated with output procedures rather than input procedures. You use them to return records one by one, in sequence, from the storage area where the sorting or merging takes place to the record area associated with the sort/merge work file (SD) entry. Then you can move the record from the SD file to your output file. The format is:

```
RETURN file-name RECORD [INTO identifier] ; AT END imperative-statement
```

You must describe file-name in a sort/merge file description (SD) entry.

You can use MOVE statements to move your records from the SD file to your output file, or you can use the INTO phrase of the RETURN statement to move them automatically. (Again, the records are moved according to the rules for the MOVE statement without the CORRESPONDING phrase.)

The following example uses the RETURN statement without the INTO option:

```
OUTPUT-PROCEDURE SECTION.
OUT-PROC.
    RETURN THE-SORTED-RECORD AT END GO TO END-OUT-PROC.
    MOVE THE-SORTED-RECORD TO PRINT-RECORD.
    WRITE PRINT-RECORD.
    GO TO OUT-PROC.
END-OUT-PROC.
```

The term THE-SORTED-RECORD is the name of the record area associated with the sort/merge work file. The sorted record is moved from the sort/merge work file to an output area called PRINT-RECORD, and then printed. When a RETURN statement is executed and no more records remain in the sort/merge area, the AT END statement is executed and control passes to the SORT or MERGE statement that initiated the output procedure.

To do the same processing using the INTO option, you code the output procedure as follows:

```
OUTPUT-PROCEDURE SECTION.
OUT-PROC.
    RETURN THE-SORTED-RECORD INTO PRINT-RECORD
        AT END GO TO END-OUT-PROC.
    WRITE PRINT-RECORD.
    GO TO OUT-PROC.
END-OUT-PROC.
```

In this example, each record returned to the sort/merge work file record area automatically is moved to the output area called PRINT-RECORD.

When a RETURN statement is executed, the next record, in the order specified by the keys listed in the SORT or MERGE statement, is moved to the sort/merge work file record area. As mentioned, if no next logical record exists when the RETURN statement is executed, the AT END condition occurs. At this point, the contents of the SD record area are undefined. You must code your output procedure so that no attempt is made to execute a RETURN statement after an AT END phrase is executed during the same output procedure.

If you define the records of the sort/merge work file with more than one record description, the records automatically share the same area in storage. After a RETURN statement is executed, the contents of any data items which lie beyond the range of the record just returned are undefined. Do not use the INTO phrase when the SD file contains variable length records.

9.4. SAME SORT AREA CLAUSE

The SAME SORT AREA clauses, like SAME RECORD AREA clauses (5.6.2), allow you to assign the same main storage area to more than one file, regardless of their orgnaization or access. The format is:

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

The clauses SORT and SORT-MERGE are equivalent.

You may select both sort/merge work files and input-output files to share the main storage area, with these restrictions:

- at least one of the files must be a sort/merge work file;
- more than one sort/merge work file may appear in the same SAME SORT AREA clause;
- the same sort/merge work file may not appear in more than one SAME SORT AREA clause; and
- input-output files may appear in more than one SAME SORT AREA clause.

Sorting or merging operations for any of the sort/merge files you name in a given SAME SORT AREA clause take place in the same area of main storage, and that area may be reused for an indefinite number of sorts or merges.

Additionally, storage assigned to input-output files you named in that given SAME SORT AREA clause is allocated for the sorting and merging operation on an as-needed basis by the implementor. Be certain, though, that your input-output files are not open during execution of a sort or merge operation that may use that storage.

Note that any input-output files you name in a SAME SORT AREA clause do not share storage with each other unless named in a SAME AREA or SAME RECORD AREA (5.6.2) clause. If you do use a SAME AREA clause and one or more of the files you name in it also appears in a SAME SORT AREA clause, then all the files in the SAME AREA clause must be included in the SAME SORT AREA clause.

You may name a sort/merge work file in a SAME RECORD AREA clause. When you do, the rules for the clause still apply as described in 5.6.2.

In this example, any sort or merge procedures using either SORT-FILE-1 or SORT-FILE-2 are executed in an area of main storage reserved for the two sort files. If more main storage is needed to execute the procedure, storage assigned to INPUT-FILE-1 is used:

```
I-O-CONTROL.
    SAME SORT AREA FOR SORT-FILE-1 SORT-FILE-2 INPUT-FILE-1.
```

9.5. SAMPLE SORT PROGRAM

This sample program reads a name and address card file, selects all records that have a state field equal to PA, sorts the selected records in ascending sequence according to last name, and prints an output file. Input and output procedures transfer the records to and from the sort operation.

Here is a listing of the cards in CARD-FILE:

```
ANDERSON  ALLEN     452 MAIN STREET      PHILA      PA
SMITH     ROBERT    21 LINCOLN DRIVE     ALTOONA    PA
JONES     FRED      547 HIGH STREET      TRENTON    NJ
BROWN     ROGER     11 OLD FORGE DRIVE   WAYNE      PA
ROGERS    THEODORE  66 SUNSET DRIVE      JOHNSTOWN  PA
THOMPSON  FRANKLIN  413 APPLE STREET     HARRISBURGPA
LEONARD   GEORGE    MARKET STREET        CHESTER    PA
GRANT     HOWARD    34 ELMWOOD DRIVE     CAMDEN     NJ
HENDERSON ISSAC J.  21 AVON GROVE LANE   BLUE BELL  PA
THOMAS    JOHN      16 GRANT BLVD.       DEVON      PA
TERRY     KEVIN     219 ADAMS DRIVE      BERWICK    PA
THORN     LAWRENCE  29 VAIRO BLVD.       BRYN MAWR  PA
CARTER    MICHAEL   1405 SECOND AVE.     PAOLI      PA
DRAKE     EDWARD    451 WASHINGTON ST.   PATERSON   NJ
FRANKS    CARL      25 FIRST AVENUE      NORRISTOWNPA
/*
```

Here is the printed output of the program:

```
ANDERSON  ALLEN      452 MAIN STREET      PHILA       PA
BROWN     ROGER      11 OLD FORGE DRIVE   WAYNE       PA
CARTER    MICHAEL    1405 SECOND AVE.     PAOLI       PA
FRANKS    CARL       25 FIRST AVENUE      NORRISTOWNPA
HENDERSON ISSAC J.   21 AVON GROVE LANE   BLUE BELL PA
LEONARD   GEORGE     MARKET STREET        CHESTER     PA
ROGERS    THEODORE   66 SUNSET DRIVE      JOHNSTOWN PA
SMITH     ROBERT     21 LINCOLN DRIVE     ALTOONA     PA
TERRY     KEVIN      219 ADAMS DRIVE      BERWICK     PA
THOMAS    JOHN       16 GRANT BLVD.       DEVON       PA
THOMPSON  FRANKLIN   413 APPLE STREET     HARRISBURGPA
THORN     LAWRENCE   29 VAIRO BLVD.       BRYN MAWR PA
```

Here is the complete SORT program:

```
00001      001010 IDENTIFICATION DIVISION.
00002      001020 PROGRAM-ID.     SRTEXAMP.
00003      001030 AUTHOR.         FUNDS OF COBOL.
00004      001040 ENVIRONMENT DIVISION.
00005      001050 CONFIGURATION SECTION.
00006      001060 SOURCE-COMPUTER. UNIVAC-OS3.
00007      001070 OBJECT-COMPUTER. UNIVAC-OS3.
00008      001080 INPUT-OUTPUT SECTION.
00009      001090 FILE-CONTROL.
00010      001100     SELECT CARD-FILE,ASSIGN TO CARDREADER-CDRDR-F.
00011      001110     SELECT MY-SORT-FILE, ASSIGN TO DISC-MYSRT-F.
00012      001120     SELECT PRINT-FILE, ASSIGN TO PRINTER-PTR-FC.
00013      001130 DATA DIVISION.
00014      001140 FILE SECTION.
00015      001150 FD  CARD-FILE
00016      001160     LABEL RECORDS ARE OMITTED
00017      001170     RECORD CONTAINS 80 CHARACTERS
00018      001180     DATA RECORD IS CARD-RECORD.
00019      001190 01  CARD-RECORD          PIC X(80).
00020      001200 SD  MY-SORT-FILE
00021      001210     RECORD-CONTAINS 52 CHARACTERS
00022      001220     DATA RECORD IS MY-SORT-RECORD.
00023      001230 01  MY-SORT-RECORD.
00024      001240     05  MY-SORT-KEY-LAST-NAME          PIC X(10).
00025      001250     05  FILLER                         PIC X(42).
00026      001260 FD  PRINT-FILE
00027      001270     LABEL RECORDS ARE OMITTED
00028      001280     RECORD CONTAINS 52 CHARACTERS
00029      001290     DATA RECORD IS PRINT-LINE.
00030      001300 01  PRINT-LINE           PIC X(52).
00031      003010 WORKING-STORAGE SECTION.
00032      003020 01  HOLD-CARD-AREA.
```

(continued)

```
00033      003030      05  PRINT-CARD-AREA.
00034      003040          10  PRINT-CARD-LAST-NAME        PIC X(10).
00035      003050          10  PRINT-CARD-FIRST-NAME       PIC X(10).
00036      003060          10  PRINT-CARD-STREET-ADDRESS   PIC X(20).
00037      003070          10  PRINT-CARD-CITY             PIC X(10).
00038      003080          10  PRINT-CARD-STATE            PIC X(02).
00039      003090      05  FILLER                          PIC X(28).
00040      005010 PROCEDURE DIVISION.
00041      005020 START-PROG.
00042      005030      OPEN INPUT CARD-FILE, OUTPUT PRINT-FILE.
00043      005040      SORT MY-SORT-FILE
00044      005050          ON ASCENDING KEY MY-SORT-KEY-LAST-NAME
00045      005060          INPUT PROCEDURE IS IN-PROC THRU IN-PROC-EXIT
00046      005070          OUTPUT PROCEDURE IS OUT-PROC THRU OUT-PROC-EXIT.
00047      005080      GO TO END-PROG.
00048      005090 IN-PROC SECTION.
00049      005095 IN-PROC-ROUTINE.
00050      005100      READ CARD-FILE INTO HOLD-CARD-AREA
00051      005110          AT END GO TO IN-PROC-EXIT.
00052      005120      IF PRINT-CARD-STATE IS EQUAL TO "PA"
00053      005130          MOVE PRINT-CARD-AREA TO MY-SORT-RECORD
00054      005140          ELSE GO TO IN-PROC-ROUTINE.
00055      005150      RELEASE MY-SORT-RECORD.
00056      005160      GO TO IN-PROC-ROUTINE.
00057      005170 IN-PROC-EXIT SECTION.
00058      005175 END-OF-IN-PROC.
00059      005180      EXIT.
00060      005190 OUT-PROC SECTION.
00061      005195 OUT-PROC-ROUTINE.
00062      005200      RETURN MY-SORT-FILE RECORD AT END GO TO OUT-PROC-EXIT.
00063      005210      MOVE MY-SORT-RECORD TO PRINT-LINE.
00064      005220      WRITE PRINT-LINE AFTER ADVANCING 1 LINES.
00065      005230      MOVE SPACES TO PRINT-LINE.
00066      005240      GO TO OUT-PROC-ROUTINE.
00067      005250 OUT-PROC-EXIT SECTION.
00068      005255 END-OF-OUT PROC.
00069      005260      EXIT.
00070      005270 END-PROG.
00071      005280      CLOSE CARD-FILE, PRINT-FILE.
00072      005290      STOP RUN.
```

9.6. SAMPLE MERGE PROGRAM

This sample program takes two name and address files similar to the one created in the sort program example (9.5) and merges them together to create one combined file. The records in the files already are in sequence before the program executes.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. MERGEXMP.
```

(continued)

```
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. UNIVAC-OS3.
OBJECT-COMPUTER. UNIVAC-OS3.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT NAMEFIL-1 ASSIGN TO TAPE-FIL1-F.
    SELECT NAMEFIL-2 ASSIGN TO TAPE-FIL2-F.
    SELECT MY-SORT-FILE ASSIGN TO DISC-SRT1-F.
    SELECT TAP-OUT-FILE ASSIGN TO TAPE-POUT-F.
DATA DIVISION.
FILE SECTION.
FD  NAMEFIL-1
    LABEL RECORDS ARE STANDARD
    RECORD CONTAINS 52 CHARACTERS
    DATA RECORD IS NAMEREC-1.
01  NAMEREC-1.
    02  LAST-NAME-1          PIC X(10).
    02  FIRST-NAME-1         PIC X(10).
    02  STREET-ADDRESS-1     PIC X(20).
    02  CITY-1               PIC X(10).
    02  STATE-1              PIC X(2).
FD  NAMEFIL-2    LABEL RECORDS ARE STANDARD
    RECORD CONTAINS 52 CHARACTERS
    DATA RECORD IS NAMEREC-2.
01  NAMEREC-2.
    02  LAST-NAME-2          PIC X(10).
    02  FIRST-NAME-2         PIC X(10).
    02  STREET-ADDRESS-2     PIC X(20).
    02  CITY-2               PIC X(10).
    02  STATE-2              PIC X(2).
SD  MY-SORT-FILE
    RECORD CONTAINS 52 CHARACTERS
    DATA RECORD IS MY-SORT-RECORD.
01  MY-SORT-RECORD.
    02  MY-SORT-KEY-LAST-NAME    PIC X(10).
    02  FILLER    PIC X(42).
FD  TAP-OUT-FILE
    LABEL RECORDS ARE STANDARD
    RECORD CONTAINS 52 CHARACTERS
    DATA RECORD IS TAP-OUT-RECORD.
01  TAP-OUT-RECORD    PIC X(52).
PROCEDURE DIVISION.
START-PROG.
    MERGE MY-SORT-FILE
        ON ASCENDING KEY MY-SORT-KEY-LAST-NAME
        USING NAMEFIL-1 NAMEFIL-2
        GIVING TAP-OUT-FIL.
END-PROG.
    STOP RUN.
```

10. Interprogram Communication

One way to solve a complex data processing problem is to break it down into several smaller, more manageable problems. The interprogram communication module allows you to combine separately-compiled but logically-coordinated programs for execution as a single run unit. The programs communicate with each other, initiating transfers of control and providing references for common data.

To use interprogram communication, you need to understand the terms *calling program* and *called program.* A calling program contains a CALL statement instruction (see 10.2) that transfers control to another program. That other program is referred to as the called program. For example, if program A transfers control to program B, A is a calling program and B is a called program. If program B, in turn, transfers control to program C, B is still a called program with respect to A, but is a calling program with respect to C.

You may, in your CALL statements, identify data items to be passed from the calling to the called program. In the called program, you identify those data items in the procedure division header (10.3) and describe them in the linkage section (10.1).

10.1. LINKAGE SECTION

You must include a linkage section in called programs that need to reference data passed from a calling program. The linkage section follows the file and working-storage sections in the data division. It consists of record description entries and noncontiguous elementary items originally described in the calling program in the file, working-storage, communication, or linkage section (only if the calling program is also a called program). You include only those items from the calling program that are needed by the called program.

In the procedure division, you can reference these data items only if you name them as operands in the USING phrase of the procedure division header (or if they are subordinate to such operands). Moreover, the CALL statement in the calling program must specify a USING phrase.

No space is allocated in the called program for the data items you name in the linkage section. Procedure division references to the items are resolved at object time by equating the reference in the called program to the location used in the calling program. In the case of index-names, no such correspondence is established. Index-names in the called and calling programs always refer to separate indexes.

The format for the linkage section is:

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

Note that its structure is the same as that for the working-storage section. As in working-storage, you use 77-level description entries to describe noncontiguous elementary items. Each description begins with level-number 77 and is followed by a data-name and either a PICTURE clause or a USAGE IS INDEX clause. Other data description clauses, except the initial VALUE clause, are optional; you use them, if needed, to complete the item description.

Again, you group data elements that have a definite hierarchic relationship to each other into records according to the rule for the formation of record descriptions (6.2). You can use any record description clause except the VALUE clause; it is invalid in the linkage section except in condition-name (level 88) entries.

Each record-name or 77-level data-name in the linkage section must be unique within the program. They cannot be qualified.

10.2. CALL STATEMENT

You use CALL statements in calling programs to pass control to called programs. The format is:

```
CALL {identifier-1}[USING data-name-1 [,data-name-2] ... ]
     {literal-1   }
     [;ON OVERFLOW imperative-statement]
```

Literal-1 or identifier-1 represents the name of the called program. Literal-1, if used, must be a nonnumeric literal. Identifier-1, if used, must be an alphanumeric data item whose value can be a program name.

If you want some data from the calling program to be available to the called program, you must name that data in the USING phrase. Furthermore, in the procedure division header in the called program, you must include a corresponding USING phrase that defines the same number of operands and the same number of character positions as the USING phrase in the CALL statement.

Each of the operands you include in the CALL statement's USING phrase must be defined in the calling program's file, working-storage, communication, or linkage section (only if the calling program is also a called program), and must have a level number of 01 or 77. You can qualify data-name-1, data-name-2, etc, only when they reference data items defined in the file or communication section.

You use the ON OVERFLOW phrase if you are not certain whether enough object time main storage will be available for the program you call. If you use the ON OVERFLOW phrase, and there is insufficient main storage when the CALL statement is executed, no action is taken, and the associated imperative statement is executed. If there is insufficient main storage and you did not specify the ON OVERFLOW phrase, the effects of the CALL statement are defined by the implementor.

A called program is in its initial state the first time it is called within a run unit and the first time it is called after execution of a CANCEL (10.4) that names the called program. On all other calls, the called program is in its last-used state. This incudes all data fields, the status and postiioning of all files, and all alterable switch settings.

You can use CALL statements anywhere within a segmented program (Section 14); the implementor makes certain the program logic flow is maintained. Thus, when a CALL statement appears in an independent segment, that segment is in its last-used state when the EXIT PROGRAM statement (10.5) of the called program returns control to it.

You can include CALL statements in called programs; however, the called progam must not contain a CALL statement that directly or indirectly calls the calling program.

In this CALL statement:

```
CALL COBPROG USING REC-COUNT COBRECD
     ON OVERFLOW PERFORM OVERFLOW-ROUTINE
```

the calling program transfers control to the program called COBPROG. The data-names REC-COUNT and COBRECD reference data items in the calling program that will be available to the called program. If there is not enough main storage for COBPROG at execution time, the imperative statement PERFORM OVFLOW-ROUTINE is executed.

10.3. PROCEDURE DIVISION HEADER

The format for a procedure division header in a called program is:

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

If the CALL statement in the calling program contains a USING phrase and you want the called program to access data from the calling program, you need a USING phrase in the procedure division header.

You must define data-name-1, data-name-2, etc, in the linkage section of the called program, and you must assign them a 01 or 77 level number. The procedure division of the called program may reference data-name-1, data-name-2, etc; any data items subordinate to these data-names; and any associated condition-names and index-names.

When you include the USING phrase in both the CALL statement of the calling program and the procedure division header of the called program, the corresponding operands of the two phrases refer to a single set of data that is equally available to both the called and calling programs. The operands you name in the USING phrase of the called program functionally redefine the storage area referenced by the operands named in the USING phrase of the calling program. Each group of operands must define the same number of character positions, but the names of the operands can be different.

In the following example, A and X refer to the same set of data, B and Y refer to the same set of data, and C and 2 refer to the same set of data:

In the calling program:

```
CALL PROGA USING A,B,C.
```

In the called program (PROGA):

```
PROCEDURE DIVISION USING X,Y,Z.
```

You cannot use a data-name more than once in the USING phrase of the procedure division header of the called program; however, you can use a given data-name more than once in the same USING phrase of a CALL statement.

10.4. CANCEL STATEMENT

You use CANCEL statements to release main storage areas occupied by called programs. The format is:

```
CANCEL { identifier-1 } [ , identifier-2 ] ...
       { literal-1    }   [ , literal-2    ]
```

Literal-1, literal-2, etc, represent nonnumeric literals. Identifier-1. identifier-2, etc, are alphanumeric data items whose values can be program names. The literals or identifiers you specify must not refer to any program that has been called and has not yet executed an EXIT PROGRAM statement.

Once a CANCEL statement is executed, the named program ceases to have any logical relationship to the run unit, and the main storage area it occupied is made available to the operating system. You can reestablish a logical relationship to canceled programs only by executing a subsequent CALL statement. In such an instance, the called program is in its initial state.

CANCEL statements are ignored and control passes to the next statement if the CANCEL statement names a program that has not been called in the run unit or names a program that has already been canceled (and has not been named in a subsequent CALL statement).

The statement

```
CANCEL CALDPRO
```

in a calling program that releases the main storage reserved for the called program, CALDPRO. If a CALL CALDPRO statement is executed subsequently in the calling program, CALDPRO is available in the initial state.

10.5. EXIT PROGRAM STATEMENT

EXIT PROGRAM statements are used to mark the logical end of called programs. The format is:

```
EXIT PROGRAM.
```

You must code the EXIT PROGRAM statement in a sentence by itself, and that sentence must be the only sentence in the paragraph. When an EXIT PROGRAM statement is executed in a called program, control passes to the calling program. An EXIT PROGRAM statement in a program that is not called is treated as if it were an EXIT statement (7.7.1).

10.6. SAMPLE PROGRAM

In this sample program, the calling program, PROGA, reads a deck of computer cards and, depending on the value of card column 1, may transfer control to the called program, PROGB.

If card column 1 is equal to A, PROGA adds 1 to a counter and then reads the next card. If card column 1 is not equal to A, PROGA transfers control to PROGB. As part of the transfer, two storage areas defined in PROGA (the card record and an end of file indicator) are made available to PROGB. PROGB checks card column 1, adds 1 to a counter if it finds the value B, and then returns control to PROGA.

When the last input card is read in PROGA, an X is moved to the end-of-file indicator, and PROGA prints a total of the number of input cards that had an A in card column 1; then transfers control to PROGB, which prints the number of cards that had a B in column 1. Note that PROGB checks the end-of-file indicator passed from PROGA and, finding the value X (indicated by the condition-name THIS-IS-LAST-ENTRY), branches to the end-of-job routine in PROGB. Control returns to PROGA to stop the run. Here are the programs:

Calling Program (PROGA):

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  PROGA.
ENVIRONMENT DIVISION.
CONFIGURATION.SECTION.
SOURCE-COMPUTER.  UNIVAC-OS3.
OBJECT-COMPUTER.  UNIVAC-OS3.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT CARDFILE, ASSIGN TO CARDREADER-CDR-F.
    SELECT PRINTFILE, ASSIGN TO PRINTER-PTR-FC.
DATA DIVISION.
FILE SECTION.
FD  CARDFILE
    LABEL RECORDS ARE OMITTED
    RECORD CONTAINS 80 CHARACTERS
    DATA RECORD IS CARDREC.
Ø1  CARDRECD.
    Ø3  CARD-IND    PIC X(1).
    Ø3  FILLER      PIC X(79).
FD  PRINTFILE
    LABEL RECORDS ARE OMITTED
    RECORD  CONTAINS 15 CHARACTERS
    DATA RECORD IS PRINTLINE.
Ø1  PRINTLINE.
    Ø5  PRINT-MSG  PIC X(15).
    Ø5  PRINT-TOTAL  PIC ZZZZ9.
WORKING-STORAGE SECTION.
77  EOJ-IND       PIC X(1).
77  A-COUNTER     PIC 9(5).
PROCEDURE DIVISION.
HOUSEKEEPING.
    OPEN INPUT CARDFILE, OUTPUT PRINTFILE.
    MOVE ZEROS TO A-COUNTER.
    MOVE SPACES TO EOJ-IND.
READ-ROUTINE.
    READ CARDFILE, AT END GO TO END-OF-JOB.
    IF CARD-IN = 'A'.
          ADD 1 TO A-COUNTER
          ELSE CALL PROGB USING EOJ-IND, CARDRECD.
    GO TO READ-ROUTINE.
END-OF-JOB.
    MOVE A-COUNTER TO PRINT-TOTAL.
    MOVE ' A TOTAL  ' TO PRINT-MSG.
    WRITE PRINTLINE.
    MOVE 'X' TO EOJ-IND.
    CALL PROGB USING EOJ-IND, CARDRECD.
    CLOSE CARDFILE, PRINTFILE.
    DISPLAY '**** END OF PROGRAM ****'.
    STOP RUN.
```

Called Program (PROGB):

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  PROGB.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.  UNIVAC-OS3.
OBJECT-COMPUTER.  UNIVAC-OS3.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT PR-FILE, ASSIGN TO PRINTER-PTR-FC.
DATA DIVISION.
FILE-SECTION.
FD  PR-FILE
    RECORD CONTAINS 15 CHARACTERS
    LABEL RECORDS ARE OMITTED
    DATA RECORD IS PRINTLINE.
Ø1  PR-LINE.
    Ø5  PRINT-MSG     PIC X(1Ø).
    Ø5  PRINT-TOTAL   PIC ZZZZ9.
WORKING-STORAGE SECTION.
77  B-COUNTER    PIC 9(5) VALUE ZERO.
LINKAGE SECTION.
77  END-OF-JOB-IND    PIC X(1).
    88  THIS-IS-LAST-ENTRY  VALUE 'X'.
Ø1  CARDRECORD.
    Ø3  CARD-IND     PIC X(1).
    Ø3  FILLER       PIC X(79).
PROCEDURE DIVISION USING END-OF-JOB-IND, CARDRECORD.
START-PROG.
    IF THIS-IS-LAST-ENTRY-GO TO EOJ.
    IF CARD-IND = 'B'
        ADD 1 TO B-COUNTER.
    GO TO RETURN-PARA.
EOJ.
    OPEN OUTPUT PR-FILE.
    MOVE B-COUNTER TO PRINT-TOTAL.
    MOVE ' B-TOTAL   ' TO PRINT-MSG.
    WRITE PR-LINE.
    CLOSE PR-FILE.
RETURN-PARA.
    EXIT PROGRAM.
```

11. Communication

The COBOL communication allows you to access, process, and create messages or portions of messages by communicating, through a message control system (MCS), with local and remote communication devices.

11.1. MESSAGE CONTROL SYSTEM

If you write COBOL programs that use the communication facility, your operating environment includes a message control system that has three main functions:

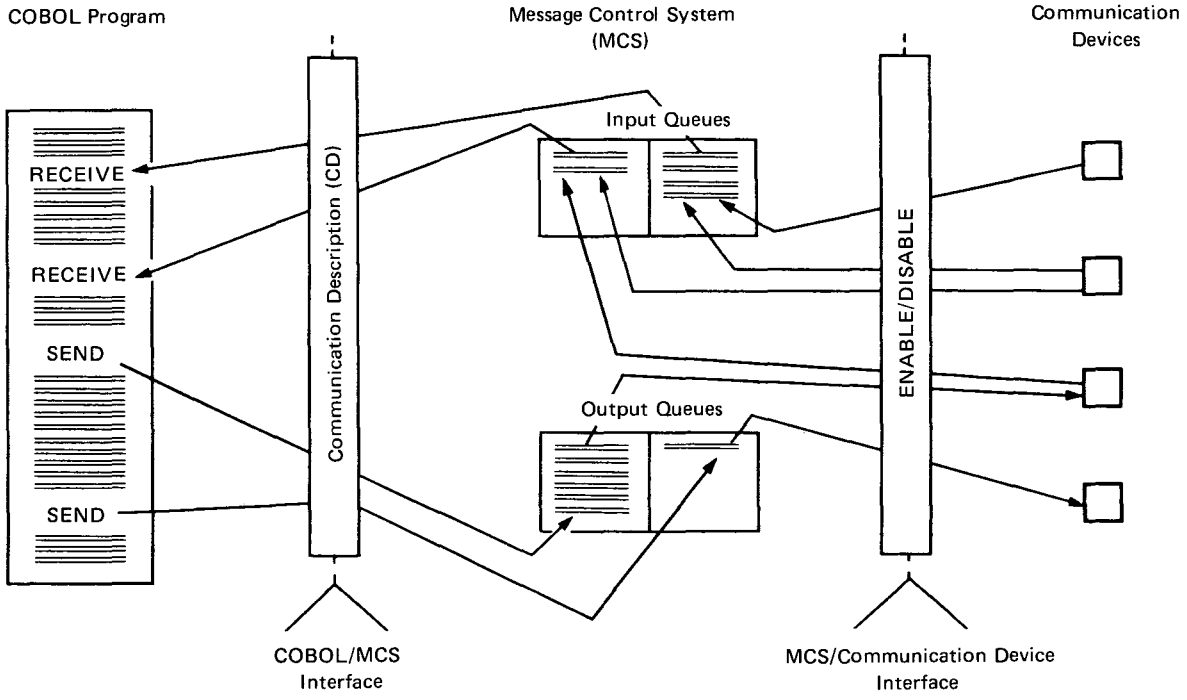
- to act as an interface between COBOL object programs and a network of communication devices (in much the same manner as an operating system acts as an interface between COBOL programs and peripheral devices such as card readers, magnetic tape, mass storage devices, and printers);
- to perform communication line discipline (including such tasks as dial-up, polling, and synchronization); and
- to perform device-dependent tasks (such as character translation and insertion of control characters, so that you can write COBOL programs that are device-independent).

You, as a COBOL programmer, do not have to be concerned with the latter two functions of the MCS. The MCS places messages from communication devices into input queues to await disposition by your COBOL object program, and places output messages from your program into output queues to await transmission to communication devices. The symbolic names of the queues, the message sources, and the message destinations all are predefined to the MCS so you can reference them in your programs. When your programs are executed, the MCS performs all actions to update the various queues as required.

11.2. COBOL COMMUNICATION ENVIRONMENT

You establish the interface between your COBOL object program and the MCS and between the MCS and the communication devices by including a communication description (CD) and associated clauses in the communication section of the data division. You control those interfaces using statements in the procedure division. SEND statements, RECEIVE statements, and ACCEPT statements with the COUNT phrase send data, receive data, or interrogate the status of queues. ENABLE and DISABLE statements control the interface between the MCS and the communication devices, directing the MCS to logically establish or break the connection between the device and a specified portion of the MCS queue structure. The method of handling the physical connection is a function of the MCS.

The COBOL communication environment is summarized in the following diagram:



11.3. PROGRAM EXECUTION METHODS

There are two ways to schedule COBOL communication object programs for execution: (1) through the normal means available in the program's operating environment, such as job control language; or (2) by the MCS. The difference between the two methods is that scheduling by the MCS moves the symbolic queue and subqueue names into the appropriate fields in the CD; scheduling by job control moves spaces to those fields.

If you plan to schedule communication programs through the normal means available in your operating environment, you can use three statements in your program to determine what messages, if any, are available in the input queues:

- the ACCEPT statement with the COUNT phrase;
- the RECEIVE statement with a NO DATA phrase; and
- the RECEIVE statement without a NO DATA phrase.

In the latter case, a program wait is implied if no data is available.

You use MCS invocation if you want to schedule communication programs only when there is work available for it to do. For this type of scheduling, your program must include a CD that specifies the FOR INITIAL INPUT clause.

The MCS determines the COBOL object program needed to process an available message, then schedules that program for execution. Before the program is executed, the MCS places symbolic queue and subqueue names into the appropriate fields in the CD associated with the FOR INITIAL INPUT clause.

Note that you can test these fields to determine how the program was scheduled. If the fields contain spaces, job control statements scheduled the program. If they do not contain spaces, the MCS scheduled the COBOL object program and replaced the spaces with the symbolic name of the queue containing the message to be processed. You can then retrieve that message by executing a RECEIVE statement directed to the CD associated with the FOR INITIAL INPUT clause.

11.4. MESSAGES AND MESSAGE SEGMENTS

Messages comprise the fundamental, but not necessarily the most elementary, units of data to be processed in a COBOL communication environment. They consist of some arbitrary amount of information, usually character data, whose beginning and end are defined or implied.

You may divide messages into smaller units of data called message segments, which are separated within messages by end of segment indicators (ESI). Messages consisting of one or more segments are separated from other messages by end of message indicators (EMI). Similarly, groups of messages are separated from each other by end of group indicators (EGI).

When your COBOL object program receives a message or message segment, the MCS updates a specified area of the associated CD to indicate if an ESI, EMI or EGI was sent with the messge (but not as part of the message text). When your COBOL object program sends a message or message segment, you must specify ESI, EMI or EGI, if needed, in the SEND statement. Thus, the presence of these logical indicators is recognized and specified by both the MCS and the COBOL object program.

Of the indicators, EGI is the most inclusive and ESI is the least inclusive. Thus, the existence of an EGI implies the existence of EMI and ESI, and the existence of an EMI implies the existence of ESI.

11.5. QUEUES

Queues form the data buffers between your COBOL object program and the MCS. They consist of the messages being sent to or being received from communication devices. Input queues are logically separate from output queues.

The MCS places or removes only complete messages in or from queues. So, for example, if your program executes a RECEIVE statement with the SEGMENT phrase, the MCS does not pass the message segment to your program until the entire message is available in the input queue. Similarly, the MCS does not transfer messages or message segments to communication devices until the complete message is in the output queue. The number of messages that exist in a given queue reflects only the number of complete messages that exist in the queue.

The process that places messages into queues is called enqueueing. Dequeueing is the process that removes messages from queues.

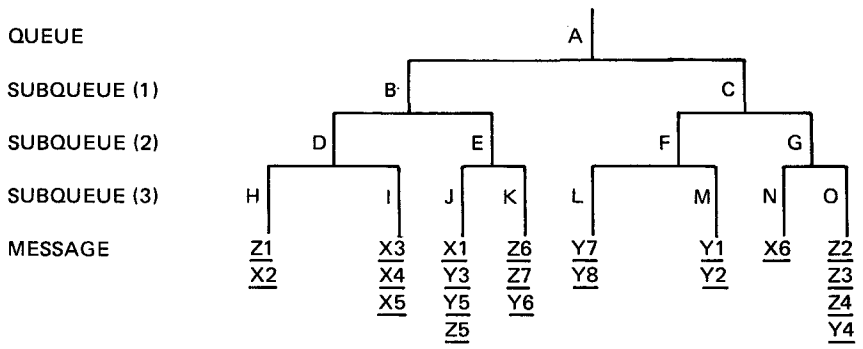
It is possible that the MCS will receive a message for your program before it is executed. When that happens, the MCS enqueues the message in the proper input queue, where it remains until your program requests dequeueing (by issuance of a RECEIVE statement).

It is also possible that your program will send messages that will not be transmitted to a communication device until after the program terminates. This could happen either because the data transfer between the queue and the device was inhibited, or because your program created messages faster than the device could receive them.

Depending on your operating environment, you may, before you run a program, be able to tell the MCS how to select the queues that will hold your messages. You might, for example, specify that all messages from a given source be placed in a given input queue, and that all messages to be sent to a given destination be placed in a given output queue. Dequeueing often is done on a first in, first out basis. However, through prior specifications to the MCS, you can set up a priority system for dequeueing.

You can exercise even more control of enqueueing and dequeueing through use of a hierarchy of input queues defined to the MCS. The available queues, in order of decreasing significance, are named queue, subqueue-1, subqueue-2, and subqueue-3.

Following is an example of a queue hierarchy. Queues and subqueues are named with letters A through O. Messages are named with letters according to their source (X, Y, or Z), and with a sequential number.



Let's assume messages are placed in the various queues according to the contents of a specified data field in each message. Also assume that when RECEIVE statements that dont't specify a given subqueue level are executed, the MCS chooses the subqueue from that level in alphabetical order. For example, if your program executes a RECEIVE statement associated with a CD that specifies queue A and no subqueues, message Z1 is retrieved.

That's because the MCS chose (in alphabetical order) B for subqueue-1, D for subqueue-2 and H for subqueue-3. Other examples:

1. A RECEIVE statement associated with a CD that specifies queue A and subqueue-C retrieves message Y7.

2. A RECEIVE statement associated with a CD that specifies queue A, subqueue-1B and subqueue-2E returns message X1.

3. A RECEIVE statement associated with a CD that specefies queue A, subqueue-1C, subqueue-2G, and subqueue-3N yields message X6.

Of course, the MCS in your operating environment may perform dequeueing based on some criteria other than alphabetical order.

For output, you specify only the destination or destinations of the message, and the MCS places the message in the proper output queue structure. Note there is no one-to-one relationship between communication devices and sources or destinations; i.e., a source or destination may consist of one or more physical devices. The devices that comprise a source or destination are predefined to the MCS.

11.6. DATA DIVISION ENTRIES

When you use the communication module, you have to include communication descriptions (CDs) in the communication section of your program's data division to define the interface area between the program and the MCS. This interface area gives you informtion about the message being handled; however, the information is provided by the MCS (it does not come from the terminal as part of the message). There are separate formats for input and output CDs.

11.6.1. Input Communication Description (CD)

You define input CDs by coding selected clauses that associate data-names with fields in the CD area, or by coding a string of 11 data-names which, taken in order, correspond to the 11 fields in a CD area. Here is the format:

```
CD cd-name;
    FOR [INITIAL] INPUT  [[; SYMBOLIC QUEUE IS data-name-1]
                         [; SYMBOLIC SUB-QUEUE-1 IS data-name-2]
                         [; SYMBOLIC SUB-QUEUE-2 IS data-name-3]
                         [; SYMBOLIC SUB-QUEUE-3 IS data-name-4]
                         [; MESSAGE DATE IS data-name-5]
                         [; MESSAGE TIME IS data-name-6]
                         [; SYMBOLIC SOURCE IS data-name-7]
                         [; TEXT LENGTH IS data-name-8]
                         [; END KEY IS data-name-9]
                         [; STATUS KEY IS data-name-10]
                         [; MESSAGE COUNT IS data-name-11]]
                        [data-name-1, data-name-2, ..., data-name-11]]
```

If your CD definition is written as a string of 11 data-names, each data-name must be unique within the CD. Within the string, you may replace any data-name with the reserved word FILLER.

If your CD area isn't defined as clauses or as a string of 11 data-names, you must define it using a level 01 data description entry that describes an 87-character record.

No matter how you define your CD area, you may redefine it as many times as you want using 01 level record descriptions, each defining 87-character records. However, you can use VALUE clauses only in the first redefinition.

If you want the MCS to schedule your program, your input CD must include the INITIAL option. As explained in 11.3, MCS invocation moves the symbolic names of the queue structure that demanded the activity to the fields referenced by data-name-1 through data-name-4 of the CD associated with the INITIAL clause; otherwise, those fields are initialized to spaces. The symbolic names are inserted or the initialization-to-spaces is completed before the first procedure division statement is executed.

When your program is scheduled by the MCS, data-name-1 through data-name-4 are filled, but the rest of the CD (represented by data-name-5 through data-name-11) is not updated until the execution of a RECEIVE statement for the CD.

Only one CD in your program may specify the INITIAL option. Do not include a CD with the INITIAL option in programs that specify the USING phrase of the procedure division header (i.e., in a called program). The results are undefined if the MCS attempts to schedule a program that does not have an input CD with the INITIAL option.

The CD area, whether you define it with clauses or with the list of 11 data-names, consists of 87 contiguous standard data format characters and has an implicit description equivalent to the following:

```
Implicit Description                                    Comment

01  data-name-0.                                        CD NAME
    02  data-name-1.     PICTURE  X(12).               SYMBOLIC QUEUE
    02  data-name-2      PICTURE  X(12).               SYMBOLIC SUB-QUEUE-1
    02  data-name-3      PICTURE  X(12).               SYMBOLIC SUB-QUEUE-2
    02  data-name-4      PICTURE  X(12).               SYMBOLIC SUB-QUEUE-3
    02  data-name-5      PICTURE  9(06).               MESSAGE DATE
    02  data-name-6      PICTURE  9(08).               MESSAGE TIME
    02  data-name-7      PICTURE  X(12).               SYMBOLIC SOURCE
    02  data-name-8      PICTURE  9(04).               TEXT LENGTH
    02  data-name-9      PICTURE  X.                   END TEXT
    02  data-name-10     PICTURE  XX.                  STATUS KEY
    02  data-name-11     PICTURE  9(06).               MESSAGE COUNT
```

NOTE:

The comment entry is not part of the description.

Data-name-1 through data-name-4 represent the hierarchy of queues as described in 11.5. Before you code RECEIVE statements in your program, you tell the MCS where to find the desired message by moving symbolic names for the appropriate queues to these data-names. The symbolic names must follow the rules for the formation of system-names, and must be previously defined to the MCS. If you do not specify all the subqueues (data-name-2 through data-name-4), the MCS determines the message or message segment accessed. The field representing any subqueue not specified must contain spaces before the RECEIVE statement is executed. When you do specify subqueues, you must also specify all higher queue levels.

After the execution of a RECEIVE statement, data-name-1 through data-name-4 contain the symbolic names of all the levels of the queue structure that contained the message.

Given the following partial CD description,

```
CD EXAMP; FOR INPUT
    SYMBOLIC QUEUE IS SYM-QUEUE
    SYMBOLIC SUB-QUEUE-1  IS SYM-Q-1
    SYMBOLIC SUB-QUEUE-2  IS SYM-Q-2
```

you might, for example, code

```
MOVE  Q-A TO SYM-QUEUE
MOVE  SUB-Q-A TO SYM-Q-1
MOVE  SPACES TO SYM-Q-2, SYM-Q-3
```

before the RECEIVE statement. The MCS, then, determines the appropriate subqueue-2 and subqueue-3 from which to retrieve the message and, after the RECEIVE statement is executed, moves the symbolic names for those subqueues to SYM-Q-2 and SYM-Q-3.

As with data-name-1 through data-name-4, data-name-5 through data-name-9 are updated by the MCS only when RECEIVE statements are executed.

Data-name-5 and data-name-6 contain the date and time, respectively, that the MCS recognizes that the message is complete. The date is in YYMMDD (year, month, day) format; the time is in HHMMSSTT (hours, minutes, seconds, hundredths of a second) format.

In data-name-7, you find the symbolic name of the communication terminal that sent the message. If the name is not known to the MCS, data-name-7 equals spaces.

You test data-name-8 to learn the number of character positions filled by the execution of the RECEIVE statement.

Data-name-9, the end key indicator, is set as follows:

- to 3, if an end of group indicator (EGI) is detected;
- to 2, if an end of message indicator (EMI) is detected;
- to 1, if an end of segment indicator (ESI) is detected (and the RECEIVE SEGMENT phrase is specified); or
- to 0, if less than a message (if RECEIVE MESSAGE is specified) or less than a message segment (if RECEIVE SEGMENT is specified) is transferred.

When more than one of these conditions is satisfied simultaneously, data-name-9 is set according to the condition with the highest number. Thus, if both EMI and EGI are detected, data-name-9 equals 3.

You test data-name-10 to find the status condition of the previously executed RECEIVE, ACCEPT MESSAGE COUNT, ENABLE INPUT, or DISABLE INPUT statement. The following table explains possible status key codes and their meanings:

RECEIVE	ACCEPT MESSAGE COUNT	ENABLE INPUT (without TERMINAL)	ENABLE INPUT (with TERMINAL)	DISABLE INPUT (without terminal)	DISABLE INPUT (with terminal)	STATUS KEY CODE	FUNCTION
X	X	X	X	X	X	00	No error detected. Action completed.
X	X	X		X		20	One or more queues or subqueues unknown. No action taken.
			X		X	20	The source is unknown. No action taken.
		X	X	X	X	40	Password invalid. No enabling disabling action taken.

NOTE:

An X on a line in a statement column indicates the associated code shown for that line is possible for that statement.

The MCS updates data-name-11 only when ACCEPT statements with the COUNT phrase are executed. You use it to find out how many complete messages are in the queue structure specified in data-name-1 through data-name-4.

Following is an example of how you can code a Format 1 CD:

```
CD  CDEXAM; FOR INITIAL INPUT
    SYM-Q, SYM-Q-1, SYM-Q-2, SYM-Q-3, MSG-DATE
    MSG-TIME, SYM-SRG, FILLER, FILLER, STAT-KEY, MSG-CT
```

The name of this CD is CDEXAM. Since the INITIAL option is used, the program that includes this CD definition may be scheduled for execution by the MCS. The queue structure for this CD is represented by the names SYM-Q, SYM-Q-1, SYM-Q-2, and SYM-Q-3.

The message date and time are in fields referenced by the data-names MSG-DATE and MSG-TIME. The symbolic name of the terminal that sends a message received from the queue structure defined in this CD is referenced by SYM-SRC.

The reserved word FILLER is used for text length and end key. Since complete input messages of fixed length are always received, it is not necessary to reference these fields. The data-names STAT-KEY and MSG-CT reference the status key and the message count.

11.6.2. Output Communication Description (CD)

You use this format to describe CD areas used for output. The CD information is not sent to the terminal; it constitutes communication between your program and the MCS about the message being handled. The format is:

```
CD cd-name; FOR OUTPUT
   [; DESTINATION COUNT IS data-name-1]
   [; TEXT LENGTH IS data-name-2]
   [; STATUS KEY IS data-name-3]
   [; DESTINATION TABLE OCCURS integer-2 TIMES
      [; INDEXED BY index-name-1 [,index-name-2]...]]
   [; ERROR-KEY IS data-name-4]
   [; SYMBOLIC DESTINATION IS data-name-5].
```

The size of the CD output area depends on the number of message destinations. The first 10 characters of the area (the destination count, text length, and status key) are followed by 13-character destination table entries. So, if there are three destinations in the destination table, the CD area has 10 plus 13 times 3 or 49 characters. If you don't use the DESTINATION TABLE OCCURS clause, one ERROR KEY and one SYMBOLIC DESTINATION area are assumed. This produces a 23-character area.

Use of the clauses gives you a CD record with an implicit description equivalent to the following:

Implicit Description			Comment
01	data-name-0.		CD NAME
02	data-name-1	PICTURE 9(04).	DESTINATION COUNT
02	data-name-2	PICTURE 9(04).	TEXT LENGTH
02	data-name-3	PICTURE XX.	STATUS KEY
02	data-name	OCCURS integer-2 TIMES.	DESTINATION TABLE
03	data-name-4	PICTURE X.	ERROR KEY
03	data-name-5	PICTURE X(12).	SYMBOLIC DESTINATION

NOTE:

The comment entry is not part of the description.

If you don't use any of the optional clauses, you must describe the CD area with a level 01 data description entry. As with input CDs, you can redefine output CDs as often as you want (even if clauses are used for the orginal definition) using level 01 record descriptions. Again, you can use VALUE clauses in only the first redefinition.

You use data-name-1 to tell the MCS how many of the destinations represented by data-name-5 are affected when SEND, ENABLE OUTPUT, or DISABLE OUTPUT statements are executed. So, for example, if data-name-1 equals 4 and the CD has 15 occurrences of data-name-5, only the first 4 of these occurrences (or destinations) are affected by the execution of, for example, a SEND statement. You must make certain that, when one of the three statements mentioned is executed, the value of data-name-1 is within the range of 1 and the number of occurrences of destination table entries.

When a SEND, ENABLE OUTPUT, or DISABLE OUTPUT statement is executed, and the value of data-name-1 is not within that range, an error condition is indicated and execution of the statement is terminated. You use data-name-2 to tell the MCS the number of leftmost character positions being transferred from the identifier associated with a SEND statement.

The status key, represented by data-name-3, is similar to the status key in an input CD. You test it to find the status condition of the previously executed SEND, ENABLE OUTPUT, or DISABLE OUTPUT statement. The possible status key codes are explained in the following chart. An X on a line in a statement column indicates the associated code shown for that line is possible for that statement.

SEND	ENABLE OUTPUT	DISABLE OUTPUT	STATUS KEY CODE	FUNCTION
X	X	X	00	No error detected. Action complete.
X			10	One or more destinatinos are disabled. Action completed.
X	X	X	20	One or more destinations unknown. Action completed for known destinations; no action taken for unknown destinations. Error key (data-name-4) indicates error or unknown.
X	X	X	30	Content of destination count (data-name-1) invalid. No action taken.
	X	X	40	Password invalid. No enabling or disabling action taken.
X			50	Text length (data-name-2) greater than length of (sending field). No action taken.
X			60	Partial segment with either zero text length or no sending area specified. No action taken.

After a SEND, ENABLE OUTPUT, or DISABLE OUTPUT statement is executed the error key data item is set to 0, unless the value of data-name-5 was not defined to the MCS. In that case, the error key code is set to 1 and the status key code (data-name-3) is set to 20.

The destinations referenced by data-name-5 must be previously defined to the MCS. The names must follow the rules for the formation of system-names.

If the CD has more than one destination specified, you may refer to data-name-4 and data-name-5 only by subscripting or indexing. If there is only one destination you may not use subscripting or indexing when referencing these fields.

Following is an example of an output:

```
CD CDOUT; FOR OUTPUT
   DESTINATION COUNT IS DEST-CT
   TEXT LENGTH IS TXT-LTH
   STATUS KEY IS ST-KEY
   DESTINATION TABLE OCCURS 3 TIMES
   ERROR-KEY IS ERR-KEY
   SYMBOLIC DESTINATION IS SYMDEST.
```

The name of this output CD is CDOUT. DEST-CT references the number of the table entries affected when the next statement associated with the CD is executed.

The number of characters in a message sent to the MCS from an identifier associated with a SEND statement is referenced by TEXT-LTH. The value of ST-KEY tells you the status of the previously executed SEND, ENABLE, OUTPUT, or DISABLE OUTPUT statement.

There are at most three destinations and associated error keys, represented by ERR-KEY and SYM-DEST, at any given time when a SEND, ENABLE OUTPUT, or DISABLE OUTPUT statement is executed. When you reference ERR-KEY and SYM-DEST they must be subscripted.

11.7. ACCEPT MESSAGE COUNT STATEMENT

You use ACCEPT MESSAGE COUNT statements to find how many complete messages are available in a particular input queue. The format is:

```
ACCEPT CD-name MESSAGE COUNT
```

The queue tested is referenced by data-name-1 through data-name-4 of the input CD named. When ACCEPT MESSAGE COUNT statements are executed, data-name-2 through data-name-4 of the CD may contain spaces, but data-name-1 must contain the name of a symbolic queue known to the MCS. Any symbolic subqueues (data-name-2 through data-name-4) you specify also must contain names known to the MCS.

After an ACCEPT MESSAGE COUNT statement is executed you find how many messages are available by testing data-name-11 of the CD. Testing data-name-10 of the CD tells you whether the ACCEPT MESSAGE COUNT statement executed successfully.

Given the following CD definition

```
CD    CDEXAM;  FOR  INPUT
      SYMBOLIC QUEUE IS SYM-Q:
      SYMBOLIC SUB-QUEUE-1 IS SUB-Q-1
      STATUS KEY IS STAT-KEY
      MESSAGE COUNT IS MSG-COUNT.
```

you code

```
ACCEPT CDEXAM MESSAGE COUNT
```

to find how many messages are in the queue SYM-Q, SUB-Q-1. That number is available in the field referenced by MSG-COUNT. The value of STAT-KEY tells you if the ACCEPT statement executed successfully.

11.8. DISABLE STATEMENT

DISABLE statements logically break the connection between the MCS and specified sources or destinations. The format is:

```
DISABLE {INPUT  [TERMINAL]} cd-name WITH KEY {identifier-1}
        {OUTPUT            }                 {literal-1   }
```

You use INPUT TERMINAL to disconnect the logical path between a specific source terminal and all queues and subqueues. You move the name of the terminal you want to logically deactivate to data-name-7 (SYMBOLIC SOURCE) of the area referenced by cd-name in the format. The contents of any other data fields in the same CD area are not meaningful when DISABLE INPUT TERMINAL statements are executed.

If you specify the INPUT phrase without the optional word TERMINAL, you disconnect the logical paths for all the sources associated with the queues and subqueues of the area referenced by cd-name.

The OUTPUT phrase similarly deactivates the logical paths between the output queues and the destinations specified by data-name-5 of the CD area specified.

Literal-1 or identifier-1 must match the system password (a 1-to-10-alphanumeric-character name known to the MCS). If it does not match, the DISABLE statement is not executed and the value of the STATUS KEY in the area referenced by cd-name is updated.

When the MCS and the specified sources or destinations are logically disconnected or are to be handled by some other means external to the program, the DISABLE statement is not required in the program.

The execution of DISABLE statements causes the logical disconnection only when the affected sources or destinations are inactive. It never causes the remaining portion of a message to be terminated during transmission to or from a terminal.

DISABLE statements do not affect the logical path for the transfer of data between your COBOL program and the MCS.

Here is an example of a DISABLE statement:

```
DISABLE INPUT TERMINAL CD-EXAM WITH KEY PASS-WORD
```

Execution of this statement logically inhibits the transfer of data between the terminal identified by data-name-7 of CD-EXAM, and the input queues. PASS-WORD is defined in the working-storage section as alphanumeric characters. Its value must be the same as that of the system password previously specified to the MCS.

11.9. ENABLE STATEMENT

ENABLE statements work like DISABLE statements except they establish, rather than break, the logical connection between the MCS and communication devices. The format is:

```
ENABLE {INPUT  [TERMINAL]}cd-name WITH KEY {identifier-1}
       {OUTPUT           }                 {literal-1   }
```

The use of the INPUT, TERMINAL, OUTPUT, and KEY phrases, and the rules of statement operation, are the same for ENABLE statements as for DISABLE statements.

Here is an example of an ENABLE statement:

```
ENABLE OUTPUT CDOUT WITH KEY PASS-WORD
```

Execution of this statement provides a logical connection between the MCS and the destinations referenced by data-name-5 of the output CD called CDOUT. You define PASS-WORD in the working storage section as alphanumeric characters. Its value must be the same as that of the system password previously specified to the MCS.

11.10. RECEIVE STATEMENT

RECEIVE statements transfer messages or portions of messages from input queues to your program. In addition, they cause the MCS to update the appropriate input CD area in your program's communication section. Here is the format:

```
RECEIVE cd-name {MESSAGE} INTO identifier-1 [; NO DATA imperative-statement]
                {SEGMENT}
```

The queue structure containing the message to be received is identified by data-name-1 through data-name-4 of the area referenced by cd-name. When RECEIVE statements are executed, the messages, message segments, or portions of messages or segments are transferred to the area referenced by identifier-1 and aligned to the left without space fill. Control then passes to the next executable statement.

If no message is available in the appropriate queue structure when the RECEIVE statement is executed, one of the following occurs:

1. If the NO DATA phrase is specified, the RECEIVE operation is considered complete, and the imperative statements in the NO DATA phrase are executed.

2. If the NO DATA phrase is not specified, execution of the object program is suspended until data becomes available.

If one of the queues or subqueues is unknown to the MCS, control passes to the next executable statement, whether or not the NO DATA phrase is specified.

When you specify RECEIVE SEGMENT, and an end-of-message indicator (EMI) or end-of-group indicator (EGI) is associated with the text being transferred, the existence of an end-of-segment indicator (ESI) is implied and the text is treated as a message segment.

When you specify RECEIVE MESSAGE, ESIs associated with the text are ignored.

If the length of the message or message segment is greater than that of the area referenced by identifier-1, you must execute multiple RECEVE statements to get the entire message or segment into your program. Once the execution of a RECEIVE statement returns a portion of a message, only subsequent execution of RECEIVE statements in that run unit can return the remaining portion of the message. Once a STOP RUN statement is executed, the disposition of the remaining portion of a message partially obtained in the run unit is defined by the implementor.

Here is an example of a RECEIVE statement:

```
RECEIVE CDEXAM MESSAGE INTO REC-MSG;
  NO DATA GO TO ABC.
```

The queue structure from which the message is retrieved is identified in data-name-1 through data-name-4 of CDEXAM. If no message is available when this statement is executed, control passes to the paragraph named ABC. REC-MSG is defined in the working-storage section.

11.11. SEND STATEMENT

SEND statements transfer your messages, or portions of messages, to output queues maintained by the MCS. They also may tell the MCS when a complete segment, message, or group of messages has been sent, and provide information about the vertical positioning of the messages on output devices.

There are two formats you can use for SEND statements:

Format 1:

```
SEND cd-name FROM identifier-1
```

Format 2:

```
SEND cd-name [FROM identifier-1] {WITH identifier-2}
                                 {WITH ESI       }
                                 {WITH EMI       }
                                 {WITH EGI       }
    [{BEFORE} ADVANCING {{{identifier-3} [LINE ]}}]
     {AFTER }           {{{integer     } [LINES]}}
                        {{{mnemonic-name}       }}
                        {{{PAGE         }       }}
```

You replace cd-name in the format with the name of the output CD associated with the destinations of your messages. Your message, before it is sent to the MCS, is contained in identifier-1. You move the length of the message, or portion of it that you wish to send to the MCS, to data-name-2 (TEXT LENGTH) before the SEND statement is executed. Once a complete message is received in an output queue, the MCS transmits it to the appropriate communication devices.

When you use Format 2, you can specify ESI, EMI, or EGI to tell the MCS that a segment, message, or group of message is complete. (See 11.4).

You may also specify the delimiting indicators via identifier-2. If you set identifier-2 to 1, it indicates a segment is complete; if you set it to 2, it means a message is complete; and if you set it to 3, it indicates a group of messages is complete.

If you specify identifier-2 and it equals any character other than 1, 2, or 3, no indicator is implied. In this event, you must specify identifier-1; otherwise, an error is indicated by the value of the status key field (data-name-3) of the associated CD and no data is transferred.

You use the ADVANCING phrase to control the vertical positioning of messages or message segments on a communication device. If vertical positioning is not applicable on the device, or if identifier-2 is not equal to 1, 2, or 3, (and you don't use ESI, EMI, or EGI), the MCS ignores the ADVANCING phrase.

You reposition messages or message segments on the communication device vertically downward according to the number of lines you specify, either as an integer number (replacing integer in the format), or as the integer value of identifier-3. If you use the BEFORE phrase, the message is written to the device before the repositioning; if you use the AFTER phrase, the message is written after the repositioning.

If you want to advance to a new page before or after a message is outputted, you use the PAGE phrase. If you specify the PAGE phrase and page has no meaning in relation to the communication device, the vertical positioning is the same as if you had coded (BEFORE or AFTER) ADVANCING 1 LINE.

If you do not use the ADVANCING phrase and vertical positioning is applicable for the device, the default value AFTER ADVANCING 1 LINE is assumed.

If you define a mnemonic-name in the SPECIAL-NAMES paragraph, and you use the ADVANCING mnemonic-name option, messages are positioned according to the rules specified by the implementor for that communication device.

When SEND statements are executed, the MCS reads the text length field (data-name-2) of cd-name to see how many characters you want to transfer from identifier-1 to the output queue. If the text length field is equal to zero, or if it is equal to a number greater than the size of identifier-1, no data is transferred. In the latter case, an error is indicated by the value of the status key field (data-name-3) of cd-name.

When the communication device that is to receive your message is oriented to a fixed line size (printer, display screen, card punch, etc.), the messages or message segments begin at the leftmost character position of the physical line, and all unused character positions to the right are space-filled. If the message is longer than the physical line size, the excess characters continue at the leftmost character position of the next line.

When the communication device that is to receive your message is oriented to handle variable length messages (paper tape punch, another computer, etc.), the message or message segments begin on the next available character position of the device.

Note that you can use Format 1 to send a portion of a message to the MCS without the indicators or vertical positioning information you use for Format 2. Also note, however, that no message is transferred by the MCS to a communication device unless the final portion of the message is sent by a Format 2 statement that specifies EGI, EMI, or identifier-2. Thus, the statement

```
SEND CDOUT FROM MSG-HOLDER
```

Standing alone, this statement does not send a message to an output device, because the MCS has no way of knowing whether it received a complete message. The MCS releases that message only after subsequent execution in the same run unit of a Format 2 SEND statement for CDOUT.

During the execution of the run unit, the disposition of a portion of a message not terminated by an EMI or EGI is undefined. Once a STOP RUN is executed, any portion of a message transferred from the run unit via a SEND statement (but not terminated by an EMI or EGI) is purged from the system.

The following sequence of statements illustrates the use of both formats of the SEND verb:

```
SEND CDOUT FROM MSG.
IF CONDITION-A,
  SEND CDOUT FROM MSG-1.
IF CONDITION-B,
  SEND CDOUT FROM MSG-2.
IF CONDITION-C,
  SEND CDOUT FROM MSG-3.
SEND CDOUT WITH EMI
  AFTER ADVANCING 2 LINES.
```

In this example, a message is constructed that may have only one part (the characters passed from MSG) or may have as many as four parts, depending on the value of CONDITION-A, CONDITION-B, and CONDITION-C. Since the size of the complete message is unknown until all conditions are tested, the end-of-message indicator and the vertical positioning information are sent to CDOUT separately, after all portions of the complete message are received by the MCS.

11.12. SAMPLE PROGRAM

Following is a sample program that uses the communication module. This program receives messages from a 1-level queue structure; writes these messages to an output file; and (for every message correctly received) sends back the message MESSAGE RECEIVED.

The 1-level queue structure, which is a symbolic queue with no subqueues, is defined in the COMMUNICATION SECTION of the program as MAIN-Q. The initialization of MAIN-Q is done in the MAIN SECTION by moving the symbolic queue name EXAM-Q into the MAIN-Q. Then MCS, using predefined criteria, determines the proper message to be transferred from the MAIN-Q to the area STORAGE-MESSAGE defined in working storage.

When a message is received properly, it is written from the STORAGE-MESSAGE area to the output file FILEA. After this, the message MESSAGE RECEIVED, which is stored in SEND-MSG, is transmitted via the SEND statement to the symbolic destination OUT-TERM. This symbolic destination is defined in the COMMUNICATION SECTION. The program continues to receive input messages until the contents indicate the end of input (in which case the program is terminated).

This program also tests the validity of input and output message transmission by testing the code contained in the STATUS KEY. The status keys, one for input messages and one for output, are indicated in the COMMUNICATION SECTION. If the STATUS KEY contains 00, the message is valid and no error processing is done. Otherwise; an error message like QUEUE NAME ERROR is displayed indicating the nature of the transmission difficulty. These validity tests are done after the execution of the ENABLE, RECEIVE, and SEND statements.

One item to note is that VALUE clauses are specified in the redefinition of the output CD description to intialize the DESTINATION COUNT, TEXT LENGTH, and SYMBOLIC DESTINATION fields.

Communication Module Program:

```
000100  IDENTIFICATION DIVISION.
000200  PROGRAM-ID, COMMEX.
000300  ENVIRONMENT DIVISION.
000400  CONFIGURATION SECTION.
000500  SOURCE-COMPUTER. UNIVAC-VS9.
000600
000700  OBJECT-COMPUTER. UNIVAC-VS9.
000800  INPUT-OUTPUT SECTION.
000900  FILE-CONTROL.
001000          SELECT FILEA ASSIGN TO DISK-DCD01-F.
001100  DATA DIVISION.
001200  FILE SECTION.
001300  FD    FILEA LABEL RECORDS STANDARD.
001400  01    RECA PIC X(50).
001500  WORKING-STORAGE SECTION.
001600  77    SEND-MSG PIC X(16)
001700           VALUE ''MESSAGE RECEIVED''.
002000  01    STORE-MESSAGE.
002100    02    MSG-TYPE PIC 9(4).
002200    02    DIV-NO PIC 9(3).
002300    02    PART-NO PIC X(10).
002400    02    WAREHOUSE-NO PIC X(7).
002500    02    FILLER PIC X(26).
002600  COMMUNICATION SECTION.
002700  CD     CD-EXAM-IN FOR INPUT
002800           SYMBOLIC QUEUE IS MAIN-Q
002900           STATUS KEY IS IN-STAT-KEY.
003000  CD     CD-EXAM-OUT FOR OUTPUT.
003100  01     REDEF-OUT.
003200    02     DEST-CT PIC 9(4) VALUE 1.
003300    02     OUT-TXT-LNGTH PIC 9(4) VALUE 16.
003400    02     OUT-STAT-KEY PIC XX VALUE SPACES.
003500    02     MSG-DEST.
003600        03     MSG-ERR-KEY PIC X VALUE SPACE.
003700        03     SYM-DEST PIC X(12) VALUE ''CUT-TERM''.
```

(continued)

```
003800  PROCEDURE DIVISION.
004800  MAIN SECTION.
004900  BEGIN-PROG.
005000        OPEN OUTPUT FILEA.
005100        MOVE ''EXAM-Q'' TO MAIN-Q.
005200        ENABLE INPUT CD-EXAM-IN WITH KEY ''EXAM-PASS''.
005300        IF IN-STAT-KEY NOT EQUAL ZEROS
005400            GO TO ERR-RTN-1.
005500        ENABLE OUTPUT CD-EXAM-OUT WITH KEY ''EXAM-OK''.
005600        IF OUT-STAT-KEY NOT EQUAL ZEROS
005700            GO TO ERR-RTN-2.
005800  REC-SEND.
005900        RECEIVE CD-EXAM-IN MESSAGE INTO STORE-MESSAGE.
005900        IF IN-STAT-KEY = ''00''
006100            WRITE RECA FROM STORE-MESSAGE
006200        ELSE
006300             GO TO ERR-RTN-3.
006400        SEND CD-EXAM-OUT FROM SEND-MSG WITH EMI.
006500        IF MSG-TYPE = 9999
006600            GO TO EOJ.
006700        IF OUT-STAT-KEY = ZEROS
006800            GO TO REC-SEND
006900        ELSE
007000            DISPLAY ''OUT-STAT-KEY ='' OUT-STAT-KEY
007100                ''  ERROR ON SEND STATEMENT''.
007200        GO TO EOJ.
007300  ERR-RTN-1.
007400        IF IN-STAT-KEY EQUAL 40
007500            DISPLAY ''INPUT CD PASSWORD ERROR''
007600        ELSE
007700            DISPLAY ''QUEUE NAME ERROR''.
007800        GO TO EOJ.
007900   ERR-RTN-2.
008000        IF OUT-STAT-KEY = 40
008100            DISPLAY ''OUTPUT CD PASSWORD ERROR''
008200        ELSE
008300            DISPLAY ''DESTINATION ERROR''.
008400        GO TO EOJ.
008500  ERR-RTN-3.
008600        DISPLAY ''QUEUE UNKOWN ON RECEIVE STATEMENT''.
008700  EOJ.
009300        CLOSE FILEA.
009400        STOP RUN.
```

12. Debug

The debug module helps you find errors during the execution of an object program. It allows you to code special routines executed each time control transfers to a specific procedure, each time the value of a data item changes, or under other conditions you designate. You include debug routines in sections immediately following the declaratives header in the procedure division.

12.1. COMPILE TIME SWITCH

A compile time switch lets you control whether your debugging code affects a particular compilation. You set it by including or excluding the WITH DEBUGGING MODE clause (12.5) in the SOURCE-COMPUTER paragraph. When the clause is present, all debugging sections and debugging lines (12.4) in the program are compiled. When it is not present, all debugging sections and debugging lines are compiled as if they were comment lines.

12.2. OBJECT TIME SWITCH

When you execute a program that includes the WITH DEBUGGING MODE clause, you use an object time switch to activate or deactivate the debugging code for that particular run. This switch is controlled outside the COBOL environment; it cannot be addressed in your program.

If you set the object time switch on at execution time, the debugging lines (12.4) and debugging sections (12.6) are executed. If you set it off, the debugging sections are not executed as if they were not present. The object time switch has no effect on execution of debugging lines.

The object time switch has no effect at run time if you do not specify the WITH DEBUGGING MODE clause in the source program at compile time.

12.3. DEBUG-ITEM

DEBUG-ITEM represents a special register associated with the execution of a debug routine. One DEBUG-ITEM is automatically allocated for programs that use the debug module. You reference DEBUG-ITEM only in debugging sections (12.6). Its implicit description is as follows:

```
01  DEBUG-ITEM
    02  DEBUG-LINE       PICTURE IS X(6).
    02  FILLER           PICTURE IS X VALUE SPACE.
    02  DEBUG-NAME       PICTURE IS X(30).
    02  FILLER           PICTURE IS X(VALUE SPACE.
    02  DEBUG-SUB-1      PICTURE IS S9999  SIGN IS LEADING SEPARATE CHARACTER.
    02  FILLER           PICTURE IS X  VALUE SPACE.
    02  DEBUG-SUB-2      PICTURE IS S9999  SIGN IS LEADING SEPARATE CHARACTER.
    02  FILLER           PICTURE IS X  VALUE SPACE.
    02  DEBUG-SUB-3      PICTURE IS S9999 SIGN IS LEADING SEPARATE CHARACTER.
    02  FILLER           PICTURE IS X VALUE SPACE.
    02  DEBUG-CONTENTS   PICTURE IS X(n).
```

The name DEBUG-ITEM and the names of its subordinate data items are reserved words. Before each execution of any of the program's debugging sections, spaces are moved to DEBUG-ITEM, then the subordinate data items are updated. The updating proceeds according to the rules for the MOVE statement, except that a move to DEBUG-CONTENTS is treated as an alphanumeric-to-alphanumeric elementary move with no conversion of data from one form of internal representation to another.

DEBUG-LINE identifies a particular source statement (the method of that identification is defined by the implementor); DEBUG-NAME gives you the first 30 characters of the name (procedure-name, file-name, etc.) that caused execution of the debugging section; and DEBUG-CONTENTS gives you additional information about why a particular debugging section was executed.

The relationship between these subordinate data items and the conditions that caused the execution of a debugging section is explained more fully in section 12.6.

All qualifiers of the name in DEBUG-NAME are separated by IN or OF. If the name is subscripted or indexed, the subscripts or indexes are not included in DEBUG-NAME. Instead, the occurrence number of each level is entered in DEBUG-SUB-1, DEBUG-SUB-2, and DEBUG-SUB-3, respectively, as needed.

12.4. DEBUGGING LINES

You identify source code lines as debugging lines by coding a D in the indicator area (column 7) of the line. You must place all debugging lines in the program after the OBJECT-COMPUTER paragraph. You should write the debugging lines so that a syntactically correct program is formed with or without the debugging lines being considered as comment lines.

12.5. WITH DEBUGGING MODE CLAUSE

When you debug your program, you need to include the WITH DEBUGGING MODE clause in your SOURCE-COMPUTER paragraph. As mentioned, the WITH DEBUGGING MODE clause controls the compile time switch that determines whether or not debugging lines and debugging sections are compiled as comment lines. The format is:

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

As you can see, the clause WITH DEBUGGING MODE, if needed, immediately follows the computer name. For example, if you code

```
SOURCE-COMPUTER. UNIVAC-OS3 WITH DEBUGGING MODE.
```

all debugging sections and debugging lines in the program are compiled; they are not treated as comments.

12.6. USE FOR DEBUGGING STATEMENT

The debugging sections are coded immediately following the declaratives header in the procedure division. In each debugging section, you must have a USE FOR DEBUGGING statement to define the conditions that cause the execution of that particular section. The format is:

```
section-name SECTION [segment-number].
  USE FOR DEBUGGING ON { cd-name-1                          }
                       { [ALL REFERENCE OF] identifier-1    }
                       { file-name-1                        }
                       { procedure-name-1                   }
                       { ALL PROCEDURES                     }
      [ , cd-name-2                                        ] ... .
      [ [ALL REFERENCES OF] identifier-2                   ]
      [ file-name-2                                        ]
      [ procedure-name-2                                   ]
      [ ALL PROCEDURES                                     ]
```

- cd-name-1

  If you specify cd-name-1, the associated debugging section is executed after any ENABLE, DISABLE, SEND, RECEIVE, or ACCEPT MESSAGE COUNT statement that references cd-name-1 is executed.

  Before the debugging section is executed, DEBUG-ITEM is updated so that (1) DEBUG-LINE identifies the source statement that references cd-name-1, (2) DEBUG-NAME contains the name of cd-name-1, and (3) DEBUG-CONTENTS contains the contents of the CD area associated with cd-name-1.

- identifier-1

  You use identifier-1 when you want the debugging section executed either every time identifier-1 is referenced (if you specify the ALL REFERENCES OF phrase), or every time identifier-1 is both referenced and changed in value (if you do not specify the ALL REFERENCES OF phrase).

  A debugging section that contains a USE FOR DEBUGGING clause that specifies identifier-1 also is executed:

  1. Immediately before the execution of a WRITE or REWRITE statement that explicitly references identifier-1, and after the execution of any implicit move resulting from the presence of an associated FROM phrase.

  2. Immediately after each initialization, modification, or evaluation of identifier-1 (when identifier-1 is referenced by a VARYING, AFTER, or UNTIL phrase associated with a PERFORM statement).

  3. Immediately before control is transferred by the execution of a GO TO statement with a DEPENDING ON phrase, and also prior to the execution of any debugging section associated with the procedure-name to which control is transferred by that GO TO statement.

  When the description of identifier-1 contains an OCCURS clause or is subordinate to an entry that contains an OCCURS clause, do not include the normally-required subscripting or indexing.

  When identifier-1 causes a debugging section to be executed, DEBUG-ITEM is updated so that DEBUG-LINE identifies the source statement that references identifier-1, DEBUG-NAME contains the name of identifier-1, and DEBUG-CONTENTS contains the value of identifier-1 when control passed to the debugging section.

- file-name-1

  You use file-name-1 if you want the debugging section executed after the execution of an OPEN, CLOSE, DELETE, or START statement that references file-name-1, or after the execution of any READ statement for file-name-1 (after any other specified USE procedure) that does not result in the execution of an associated AT END or INVALID KEY imperative statement.

  Before a debugging section associated with file-name-1 is executed: (1) DEBUG-ITEM is updated (so that DEBUG-LINE identifies the source statement that referenced file-name-1); (2) DEBUG-NAME contains the name of file-name-1; and (3) DEBUG-CONTENTS contains spaces (unless a READ statement was executed, so in that case it contains the entire record read).

- procedure-name-1

  When you use procedure-name-1, the debugging section is executed immediately before each execution of the procedure named, or immediately after the execution of an ALTER statement which references procedure-name-1. You must not define procedure-name-1 in a debugging section.

- ALL PROCEDURES

  If you want the debugging section executed before execution of every procedure in the program (except for those appearing within a debugging section), you can use the ALL PROCEDURES phrase. You may include the phrase only once in a program. When you do use it, you cannot specify procedure-name-1 in any USE FOR DEBUGGING statement.

  When a procedure-name causes the execution of a debugging section, the effect on DEBUG-ITEM depends on the procedure involved. The following rules apply:

  1. If the first execution of the first nondeclarative procedure causes execution of the debugging section: (a) DEBUG-LINE identifies the first statement of the procedure; (b) DEBUG-NAME contains the name of the procedure; and (c) DEBUG-CONTENTS contains the literal START PROGRAM.

  2. If a reference to procedure-name-1 in an ALTER statement causes execution of the debugging section: (a) DEBUG-LINE identifies the ALTER statement; (b) DEBUG-NAME contains procedure-name-1; and (c) DEBUG-CONTENTS contains the applicable procedure-name associated with the TO phrase of the ALTER statement.

  3. If the transfer of control associated with a GO TO statement causes execution of a debugging section: (a) DEBUG-LINE identifies the GO TO statement; (b) DEBUG-NAME contains procedure-name-1; and (c) DEBUG-CONTENTS contains spaces.

  4. If a reference to procedure-name-1 in the INPUT or OUTPUT phrase of a SORT or MERGE statement causes execution of a debugging section: (a) DEBUG-LINE identifies the SORT or MERGE statement that references procedure-name-1; (b) DEBUG-NAME contains procedure-name-1; and (c) DEBUG-CONTENTS contains; (i) the literal SORT INPUT if the reference is in the INPUT phrase of a SORT statement; (ii) SORT OUTPUT if it is in the OUTPUT phrase of a SORT statement; or (iii) MERGE OUTPUT if it is in the OUTPUT phrase of a MERGE statement.

  5. If the transfer of control associated with a PERFORM statement causes execution of the debugging section: (a) DEBUG-LINE identifies the PERFORM statement that references procedure-name-1; (b) DEBUG-NAME contains procedure-name-1; and (c) DEBUG-CONTENTS contains the literal PERFORM LOOP.

6. If procedure-name-1 is a USE procedure that is to be executed: (a) DEBUG-LINE identifies the statement that causes execution of the USE procedure; (b) DEBUG-NAME contains procedure-name; and (c) DEBUG-CONTENTS contains the literal USE PROCEDURE.

7. If an implicit transfer of control from the previous sequential paragraph to procedure-name-1 causes execution of a debugging setion: (a) DEBUG-LINE identifies the previous statement; (b) DEBUG-NAME contains procedure-name-1, and (c) DEBUG-CONTENTS contains the literal FALL THROUGH.

Everything that applies to cd-name-1, identifier-1, procedure-name-1, and file-name-1 in the format applies equally to cd-name-2, identifier-2, procedure-name-2, and file-name-2 in the format. You may use any given cd-name, identifier, procedure-name, or file-name in only one USE FOR DEBUGGING statement and only once in that statement. If you use any of the above as a qualifier, that does not count as a reference with respect to debugging.

All your debugging sections must appear together immediately after the declaratives header. Except in the USE FOR DEBUGGING statement itself, you cannot reference nondeclarative procedures within the debugging section. In addition, the statements that are not in debugging sections cannot reference procedure-names defined within debugging sections. If you want to include coding in one debugging section (other than in the USE FOR DEBUGGING statement itself) that references a procedure-name in another debugging section, you must do so using a PERFORM statement.

No debugging sections are executed more than once for a specific operand as a result of the execution of a single statement, regardless of the number of times that operand is explicitly specified. However, if you use a PERFORM statement with a phrase (such as VARYING or UNTIL) that causes successive executions of a referenced procedure, the associated debugging section is executed each time the procedure is executed. For purposes of debugging, each individual occurrence of an imperative verb within an imperative statement is considered a separate statement.

12.7. SAMPLE PROGRAM

Suppose you want to use debug to monitor the input communication description (CD) in the communication sample program (11.13). You code the program as shown in the following sample. Note that the programs are identical, except that the phrase WITH DEBUGGING MODE is added to the SOURCE-COMPUTER paragraph, and debugging lines (identified by a D in column 7) are added throughout the program.

The debugging section is executed after the execution of any ENABLE, DISABLE, SEND, RECEIVE, or ACCEPT MESSAGE COUNT statement that references CD-EXAM-IN (the cd-name specified in the USE FOR DEBUGGING statement).

In this program, CD-EXAM-IN is referenced by an ENABLE and RECEIVE statement and CD-EXAM-OUT is referenced by an ENABLE and SEND. The debugging section counts the number of times CD-EXAM-IN is referenced and the number of times CD-EXAM-OUT is referenced. At the end of the program, two counters are compared for equality.

Of course, you probably would not use the debug module to monitor such an elementary program, but you can apply the principles shown here to more complex problems.

Sample Debug Program:

```
000100  IDENTIFICATION DIVISION.
000200  PROGRAM-ID, COMMEX.
000300  ENVIRONMENT DIVISION.
000400  CONFIGURATION SECTION.
000500  SOURCE-COMPUTER. UNIVAC-VS9.
000600                          WITH DEBUGGING MODE.
000700  OBJECT-COMPUTER. UNIVAC-VS9.
000800  INPUT-OUTPUT SECTION.
000900  FILE-CONTROL.
001000          SELECT FILEA ASSIGN TO DISK-DCD01-F.
001100  DATA DIVISION.
001200  FILE SECTION.
001300  FD    FILEA LABEL RECORDS STANDARD.
001400  01    RECA PIC X(50).
001500  WORKING-STORAGE SECTION.
001600  77    SEND-MSG PIC X(16)
001700D 77       VALUE ''MESSAGE RECEIVED''.
001800D 77    CTR-1 PIC 9(3) VALUE ZEROS.
001900D 77    CTR-2 PIC 9(3) VALUE ZEROS.
002000  01    STORE-MESSAGE.
002100    02    MSG-TYPE PIC 9(4).
002200    02    DIV-NO PIC 9(3).
002300    02    PART-NO PIC X(10).
002400    02    WAREHOUSE-NO PIC X(7).
002500    02    FILLER PIC X(26).
002600  COMMUNICATION SECTION.
002700  CD     CD-EXAM-IN FOR INPUT
002800           SYMBOLIC QUEUE IS MAIN-Q
002900           STATUS KEY IS IN-STAT-KEY.
003000  CD     CD-EXAM-OUT FOR OUTPUT.
003100  01     REDEF-OUT.
003200    02     DEST-CT PIC 9(4) VALUE 1.
003300    02     OUT-TXT-LNGTH PIC 9(4) VALUE 16.
003400    02     OUT-STAT-KEY PIC XX VALUE SPACES.
003500    02     MSG-DEST.
003600        03     MSG-ERR-KEY PIC X VALUE SPACE.
003700        03     SYM-DEST PIC X(12) VALUE ''CUT-TERM''.
003800  PROCEDURE DIVISION.
003900  DECLARATIVES.
004000  THE-DEBUG SECTION.
004100          USE FOR DEBUGGING ON CD-EXAM-IN CD-EXAM-OUT.
```

(continued)

```
004200  DEB-1.
004300          IF DEBUG-NAME = ''CD-EXAM-IN''
004400              ADD 1 TO CTR-1
004500          ELSE
004600              ADD 1 TO CTR-2.
004700  END DECLARATIVES.
004800  MAIN SECTION.
004900  BEGIN-PROG.
005000        OPEN OUTPUT FILEA.
005100        MOVE ''EXAM-Q'' TO MAIN-Q.
005200        ENABLE INPUT CD-EXAM-IN WITH KEY ''EXAM-PASS''.
005300        IF IN-STAT-KEY NOT EQUAL ZEROS
005400            GO TO ERR-RTN-1.
005500        ENABLE OUTPUT CD-EXAM-OUT WITH KEY ''EXAM-OK''.
005600        IF OUT-STAT-KEY NOT EQUAL ZEROS
005700            GO TO ERR-RTN-2.
005800  REC-SEND.
005900        RECEIVE CD-EXAM-IN MESSAGE INTO STORE-MESSAGE.
005900        IF IN-STAT-KEY = ''00''
006100            WRITE RECA FROM STORE-MESSAGE
006200        ELSE
006300           GO TO ERR-RTN-3.
006400        SEND CD-EXAM-OUT FROM SEND-MSG WITH EMI.
006500        IF MSG-TYPE = 9999
006600            GO TO EOJ.
006700        IF OUT-STAT-KEY = ZEROS
006800            GO TO REC-SEND
006900        ELSE
007000            DISPLAY ''OUT-STAT-KEY ='' OUT-STAT-KEY
007100                ''  ERROR ON SEND STATEMENT''.
007200        GO TO EOJ.
007300  ERR-RTN-1.
007400        IF IN-STAT-KEY EQUAL 40
007500            DISPLAY ''INPUT CD PASSWORD ERROR''
007600        ELSE
007700            DISPLAY ''QUEUE NAME ERROR''.
007800        GO TO EOJ.
007900  ERR-RTN-2.
008000        IF OUT-STAT-KEY = 40
008100            DISPLAY ''OUTPUT CD PASSWORD ERROR''
008200        ELSE
008300            DISPLAY ''DESTINATION ERROR''.
008400        GO TO EOJ.
008500  ERR-RTN-3.
008600        DISPLAY ''QUEUE UNKOWN ON RECEIVE STATEMENT''.
008700  EOJ.
```

(continued)

```
008800D        IF CTR-1 = CTR-2
008900D            NEXT SENTENCE
009000D        ELSE
009100D            DISPLAY ''CTR-1 ='' CTR-1.
009200D            DISPLAY ''CTR-2 ='' CTR-2.
009300         CLOSE FILEA.
009400         STOP RUN.
```

13. Library

You often write source code lines in one COBOL program that are identical to source code lines you want to include in another program. If you use the library module, you can code frequently-used source text once, and store it in a library so that it is available to all COBOL programs in the operating environment. Then, when you write a program, you can represent selected library text with COPY statements (13.1) replaced at compile time by the desired code from the library. During this copying process, you can replace given literals, identifiers, words, or groups of words in the library text with alternate text.

The method of placing text in a COBOL library is defined by the implementor. Your library text must conform to the rules for COBOL reference format (2.5) and, when you supply alternate text, the replacement text words automatically are placed in your source program according to the rules for reference format.

13.1. COPY STATEMENT

COPY statements allow you to insert text from a COBOL library into your source program. You can use them anywhere in your program that a character string or a separator can be used, except within another COPY statement. Here is the format:

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

[ REPLACING ( [,] {==pseudo-text-1==} BY {==pseudo-text-2==} ) ... ]
                  {identifier-1      }    {identifier-2      }
                  {literal-1         }    {literal-2         }
                  {word-1            }    {word-2            }
```

If there is more than one COBOL library in your operating environment, you use the OF or IN library-name phrase to identify the particular library that contains the text (identified by text-name) you want inserted in your program when it is compiled. The text-name within a library must be unique.

You must precede COPY statements with a space and end them with a period. When COPY statements are executed, the entire statement, beginning with the reserved word COPY and ending with the period, inclusive, is replaced by the contents of text-name.

If you do not use the REPLACING phrase, the text from the COBOL library is copied into your program unchanged. If you do use the REPLACING phrase, you can alter the text so that certain changes are made as it is moved from the library to your program. Specifically, each matched occurrence of pseudo-text-1, identifier-1, literal-1, and word-1 (any single COBOL word) in the library text is replaced by the corresponding pseudo-text-2, identifier-2, literal-2, or word-2.

13.2. COMPILER SEARCH RULES

There are several general rules you need to know about how the compiler searches for these matched occurrences. In the matching process, an occurrence of any of the following in pseudo-text-1 or in the library text is treated as a single space:

- Each occurrence of a comma or semicolon (unless pseudo-text-1 consists solely of a comma or semicolon).
- Each sequence of one or more spaces.
- Any comment line.

The matching operation proceeds as follows:

- Any separator comma, semicolon, or spaces preceding the leftmost word in the library text you select is immediately copied into your source program.
- Starting with the leftmost library text word and the first pseudo-text-1, identifier-1, word-1, or literal-1 you coded in the REPLACING phrase, the entire REPLACING phrase operand preceding the word BY is compared to an equivalent number of contiguous library text words.
- If there is no match, the comparison is made with the next (if any) operand you coded to the left of BY. This continues with each pseudo-text-1, identifier-1, word-1, or literal-1 until there is no next successive REPLACING operand, or until there is a match.
- In the former case, the leftmost library text word is copied into the program, the next successive library text word is then considered leftmost, and the comparison cycle starts again.
- When there is a match, the appropriate pseudo-text-2, identifier-2, word-2, or literal-2 is copied into the source program; the library text word to the right of the last word included in the match becomes the leftmost library text-word; and the comparison cycle starts again with the first pseudo-text-1, identifier-1, word-1, or literal-1 that you coded.

- This comparison continues until the rightmost text word in your library text has either participated in a match or been considered as a leftmost library text word and participated in a complete comparison cycle.

You may not code pseudo-text-1 so that it consists solely of spaces or comment lines. Additionally, it may not be null. Pseudo-text-2, however, can be null.

13.3. DEBUGGING LINE REFERENCE

If you have debugging lines (12.4) within your library text, the D in the indicator area is ignored for purposes of matching. If a debugging line in your program includes a COPY statement, the library text you copy is, with one exception, compiled as though it were specified on debugging lines. The exception is that comment lines in the library text appear as comment lines in your program. You cannot use debugging lines within pseudo-text-1, but they are permitted within pseudo-text-2.

Note that the syntactic correctness of your library text cannot be independently determined, just as the syntactic correctness of your entire COBOL source program cannot be determined until all your COPY statements are processed.

13.4. CODING

The COPY statement often is used in the SOURCE-COMPUTER and OBJECT-COMPUTER paragraphs. If you code

```
SOURCE-COMPUTER.      COPY SRCCOM.
OBJECT-COMPUTER.      COPY OBJCOM.
```

in all your COBOL programs, the computer-names stored in text-names SRCCOM and OBJCOM are copied into the programs. If you want to run the programs on a different computer, you have to change the computer-names only once (in the COBOL library, not in all your programs).

13.5. STORAGE ROUTINE

You may want to use the library to store routines you use frequently in the procedure division. For instance, you might place the following in the library:

```
OPEN INPUT THE-FILE OUTPUT PRINT-FILE.
READ THE-FILE AT END GO TO END-JOB.
```

Then, in the appropriate location in your program, you code

```
COPY O-R-ROUTINE OF COB-LIB-1
        REPLACING THE-FILE BY CARD-FILE.
```

This statement is compiled as:

```
OPEN INPUT CARD-FILE OUTPUT PRINT-FILE.
READ CARD-FILE AT END GO TO END-JOB.
```

The O-R-ROUTINE phrase corresponds to text-name in the format; COB-LIB--1 corresponds to library-name in the format.

14. Segmentation

If main storage is a critical concern in your operating environment, you can use the segmentation facility to conserve main storage by controlling how much main storage is actually used at program execution time. This is accomplished by having only part of the entire program reside in main storage at execution time. And as the nonresident sections are needed they are brought into main storage overlaying certain resident sections.

To make this overlay mechanism work you have to indicate, in the source code, which parts of the program always reside in main storage and which of the parts can be overlaid. The COBOL segmentation facility only applies to the executable parts of a program and therefore you can segment only in the procedure division.

The final product of segmentation is object program segments that can or cannot be overlaid, which means that a program takes up less main storage space at execution time because the entire program is not resident during execution.

14.1. PROGRAM ORGANIZATION

When you segment your program, you must divide the entire procedure division into sections, and designate each section as belonging to a fixed permanent segment (14.2.1), a fixed overlayable segment (14.2.2), or an independent segment (14.3). You classify these sections based on logic requirements, frequency of use, and relationship to other sections.

14.2. FIXED PORTION

You classify sections (14.4) as part of the fixed portion of the object program if they must *logically* be in main storage during the entire execution of the program. You can have two types of segments in the fixed portion: (1) fixed permanent segments and (2) fixed overlayable segments. You control the number of fixed permanent and fixed overlayable segments in the program by using the SEGMENT-LIMIT clause (14.4.2) in the environment division.

14.2.1. Fixed Permanent Segments

Fixed permanent segments cannot be overlaid by any other part of the program and are always in main storage during the execution of the program.

14.2.2. Fixed Overlayable Segments

Fixed overlayable segments can be overlaid by other segments, but they are logically treated as if they were always in main storage. Such segments, when accessed, are always made available in their last-used state.

14.3. INDEPENDENT SEGMENTS

Independent segments can overlay (and can be overlaid by) fixed overlayable segments or other independent segments. The first time you transfer control (either implicitly or explicitly) to a given independent segment in your program, it is made available in its initial state. On subsequent transfers of control, it is in its initial state:

- When it receives control as the result of the implicit transfer of control between consecutive statements from a segment with a different segment-number. For example, this could happen if the sequence of program execution goes from a section in a fixed segment to the next consecutive section that is an independent segment.

- When it receives control as the result of the implicit transfer of control between a SORT or MERGE statement you coded in a segment with a different segment-number, and the associated input or output procedure in that independent segment. For example, a SORT statement in a fixed overlayable segment could reference its associated input procedure that is in an independent segment.

- When it receives control explicitly from a segment with a different segment-number, unless the EXIT PROGRAM statement causes the transfer. For example, a GO TO statement in a fixed segment could reference a procedure name in an independent segment. It is in its last used state.

- When it receives control as the result of the implicit transfer of control from a segment with a different segment-number (except the two implicit transfers of control noted in the previous paragraph). For example, a PERFORM exit that is in a fixed overlayable segment could return control to the statement following the PERFORM statement that is in an independent segment.

- When it receives control as the result of the explicit transfer of control because an EXIT PROGRAM statement was executed. For example, a called program could return control via the EXIT PROGRAM statement to the calling program's independent segment that contains the original CALL statement.

14.4. PROGRAM SEGMENT CLASSIFICATION

You classify program segments as fixed permanent, fixed overlayable, or independent by assigning segment numbers (14.4.1) to your sections in the procedure division, and by using the SEGMENT-LIMIT clause (14.4.2) in the OBJECT-COMPUTER paragraph of the environment division.

Generally, fixed permanent segments are sections you need for reference at all times, or sections you refer to frequently. You should classify less frequently used sections as fixed overlayable segments, and occasionally used sections as independent segments.

14.4.1. Segment Number

You classify sections in your program by including a segment-number in the section header as follows:

```
section-name SECTION [segment-number].
```

You assign a segment-number of 0 through 49 if you want the section to belong to the fixed portion of your object program. Sections in the declaratives must be in the fixed portion (segment-number less than 50). If you want the section to be part of an independent segment, you assign a segment-number of 50 through 99. You generally assign more frequently used sections lower segment-numbers than less frequently used sections.

Sections that have the same segment-number are considered part of the same segment; threrefore, you should assign the same segment-number to sections that frequently communicate with each other. Sections that have the same segment-number do not have to be grouped together in the source program.

If you do not include a segment-number in the section header, the segment-number is assumed to be 0.

14.4.2. SEGMENT LIMIT Clause

The SEGMENT-LIMIT clause allows you to classify some program segments as fixed overlayable. It appears in the OBJECT-COMPUTER paragraph and has the following format:

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

You replace segment-number with an integer from 1 through 49.

When you use this clause, segments having segment-numbers from 0 up to (but not including) the segment limit are considered fixed permanent segments, and segments with segment-numbers from the segment limit through 49 are considered fixed overlayable segments.

When you do not use the SEGMENT-LIMIT clause, all segments having segment-numbers from 1 through 49 are considered fixed permanent segments.

14.5. ALTER STATEMENT RESTRICTIONS

When you alter a GO TO statement in an independent segment you must code the ALTER statement in the same segment. All other uses of the ALTER statement are valid and performed even if the GO TO statement to which the ALTER statement refers is in a fixed overlayable segment.

14.6. PERFORM STATEMENT RESTRICTIONS

When you use PERFORM statements in segmented programs, the range of the statement is restricted.

PERFORM statements in sections that are not in independent segments may have, within their range, sections and paragraphs that are wholly contained in one or more nonindependent segments, or sections and paragraphs that are wholly contained in a single independent segment.

PERFORM statements that are in an independent segment may have, within their range, sections and paragraphs that are wholly contained in one or more nonindependent segments, or sections and paragraphs wholly contained in the same independent segment as the PERFORM statement.

14.7. SORT/MERGE STATEMENT RESTRICTIONS

When you use SORT or MERGE statements in sections that are not in independent segments, input procedures referenced by SORT statements and output procedures referenced by SORT or MERGE statements must appear totally within nonindependent segments or totally within a single independent segment.

When you use SORT or MERGE statements in independent segments, input procedures referenced by SORT statements and output procedures referenced by the SORT or MERGE statements must appear totally within nonindependent segments or totally within the same independent segment as the SORT or MERGE statement.

Index

SPERRY UNIVAC

USER COMMENT SHEET

Your comments concerning this document will be welcomed by Sperry Univac for use in improving subsequent editions.

Please note: This form is not intended to be used as an order blank.

_____

(Document Title)

_____ _____ _____

(Document No.) *(Revision No.)* *(Update No.)*

Comments:

Cut along line.

From:

_____

(Name of User)

_____

(Business Address)

Fold on dotted lines, and mail. (No postage stamp is necessary if mailed in the U.S.A.)
Thank you for your cooperation

FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 21 BLUE BELL, PA.

POSTAGE WILL BE PAID BY ADDRESSEE

SPERRY UNIVAC

ATTN.: SYSTEMS PUBLICATIONS

P.O. BOX 500
BLUE BELL, PENNSYLVANIA 19424

CUT

FOLD

SPERRY UNIVAC

USER COMMENT SHEET

Your comments concerning this document will be welcomed by Sperry Univac for use in improving subsequent editions.

Please note: This form is not intended to be used as an order blank.

_____

(Document Title)

_____ _____ _____

(Document No.) *(Revision No.)* *(Update No.)*

Comments:

Cut along line.

From:

_____

(Name of User)

_____

(Business Address)

Fold on dotted lines, and mail. (No postage stamp is necessary if mailed in the U.S.A.)
Thank you for your cooperation

FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 21 BLUE BELL, PA.

POSTAGE WILL BE PAID BY ADDRESSEE

SPERRY UNIVAC

ATTN.: SYSTEMS PUBLICATIONS

P.O. BOX 500
BLUE BELL, PENNSYLVANIA 19424

FOLD