codex

COBOL Program Reference Manual

Document Number 72792a

---

*COBOL, Edition 1974, produced by joint efforts of the CODASYL COBOL Committee and the European Computer Manufacturers Association (ECMA).

Table of Contents Page

List of Figures Page

List of Tables Page

CHAPTER 1 - INTRODUCTION

An acronym for COmmon Business Oriented Language, COBOL is an industry standard high-level language widely used for (but not limited to) business processing applications. Preferred by many for its ease of use and high level of self-documentation, COBOL has gained further acceptance due to the ease with which programs can be transferred between different manufacturers' systems.

Designed for use with Codex Intelligent Terminal Systems, CDX-CBL COBOL is a microcomputer implementation of a subset of the American National Standards Institute (ANSI) 1974 COBOL Level One standard. In addition to the many ANSI high-level features supported, the following modules are also included: nucleus, table handling, sequential file access, and library operations.

The File Management System included in COBOL permits sequential and ISAM disk file access, multiple files, and duplicate keys, as well as allowing records to be added, deleted, or updated. Also included is the UCALL statement, which allows programs written in other languages to be called and executed by Codex COBOL programs.

1.1 HARDWARE SUPPORT REQUIRED

The minimum hardware configuration required to support COBOL on the CODOS based system consists of:

-- CDX-68 Basic Display Terminal with the appropriate firmware options

-- 32K bytes of user memory (RAM)

-- .5 Mb or 1 Mb Diskette Storage (CDX-FS Series) or 10 Mb Disk Storage (CDX-FS/DR)

-- Microcomputer Module D (CDX-SBC/D)

-- System Self-Test firmware package (CDX-SST/D)

-- CObug firmware package (CDX-CBG/D)

1.2 OPTIONAL HARDWARE SUPPORTED

CODOS also supports a variety of printers, including matrix and character printers (the Codex SP Series) including the text quality (55 CPS) printer. These optional printers are linked to the Basic Display Terminal through either the Microcomputer Module D or the Printer Interface Module

(CDX-PI).

1.3 SOFTWARE SUPPORT REQUIRED

The CODOS Software package (71709) is required.

1.4 SOFTWARE INSTALLATION

Each software disk shipped contains a file called "INSTALL.SA." This file contains all necessary instructions to install any CODEX software system.

CHAPTER 2 - INTRODUCTION TO COBOL

As its name implies, COBOL (COmmon Business Oriented Language) is a high-level computer language that is suited for business applications since it is quite efficient in manipulating large data files. COBOL'S vocabulary, syntax, and punctuation comes from the English language. This feature makes COBOL easier to learn than machine-oriented languages since there is no need to know machine-language code and computer hardware architecture. The programmer can use English words and arithmetic symbols to direct computer operations. A few typical COBOL sentences appear below.

```
ADD NEW-PURCHASES TO TOTAL-CHARGES.
MULTIPLY QTY BY UNIT-PRICE GIVING VALUE.
PERFORM FEDERAL-TAX-CALCULATION.
IF ITEM-CODE IS NUMERIC, GO TO CHECK-ACCOUNT-NUMBER.
```

2.1 THE COBOL SYSTEM

COBOL sentences are easily understandable to the programmer, but must be translated into machine-language code before they can be performed by the computer. The COBOL compiler checks the source program statements for compliance with COBOL language rules, then translates them into computer-understandable object code, and assigns addresses to the program's instructions and data. The compiler does not, however, check the program's logic.

In order to compile a program, the COBOL compiler is first loaded into the computer from the diskette or disk. Next, the source program is entered into the machine, where the compiler reads and analyzes it before converting it into object code. The compiled program in object code form is used at runtime to execute the program with user-supplied data running under a COBOL interpreter.

Compiler output may be sent to the printer or CRT screen for ease of checking erroneous items. Such errors must be corrected before the program can be completely compiled and subsequently executed.

The logic of the program is checked by executing it with test data and comparing the results with the desired output. Until it works as expected, the source program is modified, recompiled, and run again. Once created, the object program may be used repeatedly to process data.

2.2 COBOL PROGRAM STRUCTURE

A COBOL program is divided into four major functional parts that are called divisions. The four division headers are:

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
DATA DIVISION.
PROCEDURE DIVISION.
```

in that format and order. Each division header must appear on a line by itself.

The Identification Division identifies the program by stating its name. It may also list the programmer, installation, and dates. Remarks may be included that explain the program's purpose. A sample Identification Division appears below.

```
IDENTIFICATION DIVISION.
PROGRAM-ID.      SCREEN-TO-DISKETTE.
AUTHOR.          COMPUTER-PERSON.
INSTALLATION.    CODEX-PHOENIX.
DATE-WRITTEN.    APRIL 2, 1980.
DATE-COMPILED.   APRIL 2, 1980.
```

The Environment Division specifies the computers that compile the source program and execute the object program. This division also links the input and output files with their respective computer peripherals. A sample Environment Division appears below.

```
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.  CDX-68.
OBJECT-COMPUTER.  CDX-68.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT DISK-FILE ASSIGN TO DISK DATADISK:DATAFILE.
    SELECT PRINT-FILE ASSIGN TO PRINTER.
```

The Data Division has a maximum of three sections. The File Section describes the record structure of each file named in the Environment Division. The Working-Storage Section describes data items stored in internal work areas. The Linkage Section contains internal areas that may be accessed by both COBOL programs and called assembly language programs and allows the programmer to pass data between programs. A sample Data Division appears in the program in Appendix E.

The Procedure Division contains the commands that direct the computer to process data and generate output in a specific order. Alternate logic paths may be listed for performance under certain conditions. A sample Procedure Division appears in the program in Appendix E.

2.3 COBOL REFERENCE FORMAT

The reference format, which provides a method for describing COBOL source programs, is described in terms of character positions or columns on a CRT line. The line may be up to 80 characters in length. Rules for spacing given in the discussion of the reference format take precedence over all other rules for spacing. Division of a source program is ordered as follows: the IDENTIFICATION DIVISION, then the ENVIRONMENT DIVISION, then the DATA DIVISION, then the PROCEDURE DIVISION. Each division must be written according to the rules for the reference format.

The standard COBOL line format is as follows:

Columns 1-6	six-digit sequence number
Column 7	continuation area
Columns 8-11	area A
Columns 12-72	area B
Columns 73-80	identification area

Since the COBOL programs are maintained by the EDITOR, a slightly more compact format is used:

Columns 1-4	four-digit line number
Column 6	continuation area
Columns 6-7	area A
Columns 8-80	area B

The sample program shown in Appendix E is an example of the compressed format. If line format compatability with the COBOL standard is desired, the following format should be used:

Columns 1-4	four-digit line number
Column 6	continuation area
Columns 7-10	area A
Columns 11-71	area B

The line numbers may then be easily expanded to six digits with the EDITOR prior to writing the source program to external media.

2.3.1 REFERENCE FORMAT REPRESENTATION

Margin L	designates the line number area consisting of four digits followed by a space.
Margin C	represents the continuation column-column 6. An * (asterisk) in margin C causes the compiler to treat the entire line as a comment line. A / (slash) in Margin C will cause the compiler to start printing the source program on the top of a new page. The remainder of the line is treated as a comment. A - (hyphen) in margin C is used to continue a non-numeric literal from one line to the next.
Margin A	represents the first column in the coding area. Normally, this will be the same column as margin C (column 6). However, column 7 may be used if desired.
Margin B	represents the second area in the coding portion of the line. Normally, column 8 is used. However, column 11 may be used if compatibility with the standard COBOL line format is desired.

2.3.2 CONTINUATION OF NON-NUMERIC LITERALS

When a non-numeric literal is continued from one line to another, a hyphen is placed in Margin C of the continuation line and a quotation mark is placed in Area B following the hyphen. All spaces at the end of the continued line and any spaces following the quotation mark of the continuation line and preceding the final quotation mark of the literal are considered part of the literal. Note that each line in this system is terminated by a carriage return. If it is desired that additional spaces are to be included at the end of the continued line, they must actually be typed in.

2.3.3 DIVISION HEADER

The division header must be the first line of a division reference format. The division header starts in margin A with the division-name followed by a space, the word DIVISION, and a period. No other text may appear on the same line as the division header.

2.3.4 SECTION HEADER

The section header begins on any line except the first line of a division reference format. The section header starts in Area A with the section-name followed by a space, the word SECTION, and a period followed by a space. No other text may appear on the same line as the section header.

A section consists of paragraphs in the ENVIRONMENT and PROCEDURE DIVISIONs and Data Description entries in the DATA DIVISION. Paragraph-names but not section-names are permitted in the IDENTIFICATION DIVISION.

2.3.5 PARAGRAPH-NAME AND PARAGRAPH

The name of a paragraph starts in Area A of any line following the first line of a division reference format (or section header if sections are used) and ends with a period followed by a space.

A paragraph consists of one or more successive sentences. The first sentence in a paragraph begins in Area B of either the same line as the paragraph-name or the line immediately following. Successive sentences begin either in Area B of the same line as the preceding sentence or in Area B of the next line.

A sentence consists of one or more statements followed by a period and a space. When the sentences of a paragraph require more than one line, they may be continued on successive lines.

2.4 COBOL PROGRAM FORMAT

Within each division, COBOL words are arranged into statements using the formats that are described in this manual. One or more statements form a sentence, which is terminated by a period.

One or more sentences, in turn, constitute a paragraph, which should be given a name so that program control can branch to a paragraph simply by referencing its name. Paragraph names do not contain the word paragraph, but are terminated by a period. A paragraph name may be followed on the same line by one of its sentences.

Similarly, several paragraphs make up a section that can also have a name that is followed by the word SECTION and a period, and must appear on a line by itself.

The Identification Division contains no section names. The Environment and Data Divisions are composed of fixed-name sections, while section names are optional in the Procedure Division and are created by the programmer.

2.5 COBOL LANGUAGE ELEMENTS

COBOL words are formed from the set of alphabetic, numeric, and special characters that are listed below.

Character	Meaning
0,1,...9	Digits
A,B,...Z	Letters
	Space
+	Plus Sign
-	Minus Sign
*	Asterisk
/	Slash
=	Equal sign
$	Dollar sign, hexadecimal sign
,	Comma
;	Semicolon
.	Period, decimal point
" or '	Quotation mark
(	Left parenthesis
)	Right parenthesis
>	Greater than symbol
<	Less than symbol
DB	Debit
CR	Credit

NOTE: A space is a COBOL character and takes up a byte of program memory. Single or double quotation marks must be used with the Codex COBOL compiler.

These characters form words that have specific meanings to the COBOL compiler. The basic elements of the COBOL language are:

Programmer-supplied names
Reserved words
Symbols
Literals
Special registers
Level numbers
Pictures

Reserved words are fixed and cannot be modified by the programmer. The other elements, however, are created by the programmer according to certain rules. All of these elements are discussed in this section with the exception of level numbers and pictures, which are discussed in Chapter 4, The Data Division.

2.5.1 PROGRAMMER-SUPPLIED NAMES

In COBOL, programmer-supplied names are symbols for the data, subroutines, internal word areas, etc. of a program. These names are grouped into two categories: data-names and procedure-names.

Data-names are assigned in the Data Division to the data items (also called fields) of a record that are processed by the program. In the following example, two data items are added, and the sum placed in a third field.

```
ADD FIRST-FIELD, SECOND-FIELD GIVING TOTAL-FIELD.
     ↑              ↑                     ↑
     └──────────────┴──────────────── data-names
```

Procedure-names are assigned to Procedure Division paragraphs for reference purposes. A procedure-name may be composed solely of numeric characters, but two references to a numeric procedure name must have the same value and the same number of digits to be considered equivalent by the compiler. For example, 0023 is not equivalent to 23 when both are procedure names.

In the example shown below, the program branches to either the CREDIT-ACCT or BILL-ACCT paragraphs, depending on whether or not the value of data item ACCT-BALANCE is negative.

```
          IF ACCT-BALANCE IS NEGATIVE GO TO CREDIT-ACCT,
          ELSE GO TO BILL-ACCT.
          .
          .
          .
     CREDIT-ACCT.
          .
          .
          .
     BILL-ACCT.
```

The following rules must be observed when creating a programmer-supplied name:

1. The name cannot be more than 30 characters long.

2. Only characters A through Z, 0 through 9, and hyphens are allowed.

3. A hyphen cannot begin or end the name.

4. Spaces are not allowed in a name.

5. Reserved words must not form the whole name.

6. Data-names must have at least one letter; procedure-names may be formed entirely of digits.

Some valid programmer-supplied names are:

NAME	SEARCH-TABLE
STREET-ADDRESS	REFUND
NET-INCOME	UPDATE-MASTER-FILE
UNIT-PRICE	COMPUTER-TOTAL

Some invalid programmer-supplied names are:

LAST NAME	(embedded space)
DATE	(reserved word)
#1	(# is a special character)
4	(A valid procedure-name, but an invalid data-name)

2.5.2 RESERVED WORDS

Reserved words are a fixed set of words that have special meanings for the COBOL compiler, and always appear in capital letters in a COBOL statement format. A reserved word cannot be used as a programmer-supplied name, but if two or more reserved words are connected by hyphens, they can be used as a data-name or a procedure-name as shown below.

Reserved Words	Valid Data-Names
FILE	OPEN-FILE
OPEN	
PERFORM	PERFORM-VALUE
INITIAL	INITIAL-VALUE
VALUE	COMPUTE-VALUE
COMPUTE	

The three types of reserved words are key words, optional words, and connectives. Each COBOL statement must contain at least one key word to impart the basic functional meaning of the statement to the compiler. Examples of key words are: ADD, MULTIPLY, CLOSE, PERFORM, and PICTURE.

All key words of a COBOL statement format are underlined in the following examples to indicate that they must be included.

Key words are: verbs such as SUBTRACT and OPEN; required syntactical words such as TO and GIVING; or words with a specific functional meaning such as NUMERIC.

Optional words are not underlined in the following examples of a statement format. These words have no effect on the generated object code; if used, however, they must be spelled correctly and cannot be replaced by another word. For example, the format:

```
IF data-name-1 IS EQUAL TO data-name-2
--                -----
```

was used in writing the following statement.

```
     A IS EQUAL TO B
     A IS EQUAL B
     A EQUAL TO B
     A EQUAL B
```

All of the above statements are valid and equivalent since IS and TO are optional words.

Optional words in one statement format may be keywords in another. For example, in the MOVE statement format:

```
MOVE data-name-1 TO data-name-2
----             --
```

TO is a key word. However, in the conditional statement format:

```
IF data-name-1 IS EQUAL TO data-name-2, ...
--                -----
```

TO is an optional word.

Connectives make a statement easier to read and understand. The logical connectives AND, OR, AND NOT, and OR NOT are used in compound conditional statements. The qualifier connective OF and IN connect a programmer-supplied name with its qualifier. The comma can be used as a series connective for two or more consecutive data-names or literals. For example:

```
ADD A, B TO C
```

means that the values of A and B are added to C.

2.5.3 SYMBOLS

COBOL symbols impart specific meanings to the compiler and can be divided into three groups: punctuation, arithmetic, and conditional. The punctuation symbols are:

Punctuation Symbol	Meaning
	Space
,	Comma
;	Semicolon
.	Period
" or '	Quotation mark
(	Left parenthesis
)	Right parenthesis

The arithmetic symbols are:

Arithmetic Symbol	Meaning
+	Addition, plus sign
-	Subtraction, minus sign
*	Multiplication
/	Division

The conditional symbols are:

Conditional Symbol	Meaning
=	Equal to
>	Greater than
<	Less than

There are no $\geq$ (greater than or equal to), $\leq$ (less than or equal to), or $\neq$ (not equal to) symbols in COBOL. Reserved words that describe these relational symbols are their equivalents.

The following rules for symbols must be observed in writing a COBOL program.

1. One space separates words, arithmetic and conditional symbols, parenthetical expressions, and literals. Consecutive spaces are treated as a single space, except when contained in a literal.

2. A semicolon or comma may be followed, but not preceded, by a space. A decimal point, however, is not followed by a space.

3. A period ends a sentence.

4. A comma may separate successive statement operands as well as a series of clauses.

5. A semicolon may separate a series of statements.

6. A left parenthesis is followed by a space; a right parenthesis is preceded by a space.

7. Plus and minus symbols that indicate positive and negative values do not have a space following them.

2.5.4 LITERALS

A literal is a fixed value that may be used in an instruction. There are three kinds of COBOL literals: figurative constants, numeric literals, and non-numeric literals. A fourth type, hexadecimal constants, are also used by the Codex COBOL compiler as screen control codes.

The figurative constants and their values are:

Figurative Constant	Value
ZERO ZEROS ZEROES	A string of one or more zeroes.
SPACE SPACES	A string of one or more spaces.
HIGH-VALUE HIGH-VALUES	A string of one or more characters that have the highest value in the ASCII collating sequence.
LOW-VALUE LOW-VALUES	A string of one or more characters that have the lowest value in the ASCII collating sequence.
QUOTE QUOTES	A string of one or more quotation marks that cannot bound a non-numeric literal.
ALL	Generates a string of characters specified by the non-numeric literal following it. May not be used in DISPLAY or STOP statements.

Singular and plural forms of a figurative constant are equivalent and may be used interchangeably. Only ZERO can be used for any data item. The rest must be used with either alphabetic or alphanumeric data-names.

The length of the figurative constant string depends on the following rules:

1. When it is moved to or compared with the contents of a data-item, the length of the string is equal to the data item's length.

2. When it is not associated with a data item, such as in a DISPLAY or STOP statement, the string is one character long.

The rules for writing numeric literals are:

1. It may consist of digits 0 through 9 with an optional decimal point and/or positive or negative sign.

2. Embedded spaces are not allowed.

3. A positive or negative sign, if used, must be the left-most character.

4. A decimal point must not be the right-most character of the numeric literal.

5. Unsigned numeric literals are positive.

6. More than one decimal point is not allowed.

Some valid numeric literals are:

+547	-1.23
.01	7431
74.1	+8.46

Some invalid numeric literals are:

2,568	(comma)
+ 8.5	(embedded space)
$56	(dollar sign)
'38'	(quotation marks)
248K	(alphabetic character)
128-	(minus sign must be left-most)

Non-numeric literals are alphanumeric and are useful in creating report headers and screen displays. Rules for creating non-numeric literals are:

1. Quotation marks always bound the non-numeric literal.

2. With the exception of the quotation mark, any COBOL character or reserved word may form the non-numeric literal.

3. Its maximum length is 255 characters.

Some non-numeric literals are:

```
'NON-NUMERIC LITERAL'
'759'
'LAST NAME      FIRST NAME     MIDDLE INITIAL'
```

When a numeric literal is bounded by quotation marks, it is treated as a non-numeric literal and classified as alphanumeric.

Hexadecimal constants are used by the Codex COBOL compiler as display control codes; these are listed in Appendix B. A hexadecimal constant is a string of digits preceded by a dollar sign, where each digit pair is the contents of one byte. All hexadecimal constants are considered to be non-numeric literals, and therefore alphanumeric.

Some hexadecimal constants are:

```
     $FF
     $30313233
     $8386
```

2.5.5 SPECIAL REGISTERS

COBOL has the following special registers: DATE, BREAK-KEY, TIME, and LINAGE-COUNTER. These registers can be read but not altered, and can be used with IF and MOVE statements.

DATE is a six-character date in the form MMDDYY that can be set at the CODOS ready prompt (=). For example:

```
77  TODAY PIC 99/99/99.
MOVE DATE TO TODAY.
```

TIME is a four-character time in the form HHMM that can be set at the CODOS ready prompt (=). For example:

```
77  CURRENT-TIME PIC 99.99.
MOVE TIME TO CURRENT-TIME.
```

BREAK-KEY is a one-character code that indicates whether or not the break key specified by the program has been pressed. Y is returned if a break has occurred; N means that a break has not occurred.

LINAGE-COUNTER yields the current printer line number whose value does not include the top margin lines. Example:

```
IF LINAGE-COUNTER EQUALS 60 THEN PERFORM TOP-PAGE.
```

2.6 COBOL FORMAT NOTATION

COBOL elements may not be arbitrarily strung together when writing a program, but must be used in fixed statement formats that are explained in this manual. Each format shows the correct syntax order, required and optional reserved words, and indicates where information is to be supplied by the programmer.

The following format notation rules are used in this manual.

1. Underlined words with all capital letters are keywords and are required in the statement unless they appear in square brackets.

2. Words that have all capital letters but are not underlined are reserved words that may be included at the programmer's discretion. If included, these words must be spelled correctly.

3. Elements within braces ({ }) indicate that the programmer must choose one of them to use in the statement.

4. Elements in square brackets ([ ]) are optional.

5. Lowercase words must be replaced as indicated with a data-name, procedure-name, or literal.

6. A horizontal ellipsis (...) means that the previous item may be repeated.

7. A vertical ellipsis means that commands have been omitted in the program sample shown as an illustration.

8. Punctuation and symbols are required when they appear in a format. Additional punctuation may be used if it conforms to the punctuation rules that were described earlier in this chapter.

CHAPTER 3 - THE IDENTIFICATION & ENVIRONMENT DIVISIONS

3.1 THE IDENTIFICATION DIVISION

The Identification Division must contain the name of the program. Optional entries are the name of the author, the dates that the program was written and compiled, and security information. The format of the Identification Division is:

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

The IDENTIFICATION DIVISION header and each paragraph name must appear in the order shown above and must start at Margin A. All comment-sentences are regarded as commentary by the compiler, which only checks to see that each one ends with a period.

The PROGRAM-ID paragraph is the only required paragraph in the Identification Division. It states the name of the COBOL source program at hand.

The AUTHOR, INSTALLATION, DATE WRITTEN, and DATE-COMPILED paragraphs are self-explanatory.

The SECURITY paragraph specifies which personnel should be allowed to use the program. Provisions for entering and checking a password, however, are not made in this paragraph.

The Comment Lines should always be included in a COBOL program to explain the basic purpose of the program.

A sample IDENTIFICATION DIVISION is shown below.

```
IDENTIFICATION DIVISION.
PROGRAM-ID.       INVENTORY-RPT.
AUTHOR.           COMPUTER PERSON.
INSTALLATION.     CODEX-PHOENIX.
DATE-WRITTEN.     APRIL 7, 1980.
DATE-COMPILED.    APRIL 7, 1980.
SECURITY.         THIS PROGRAM IS ONLY TO BE USED
                  BY SHIPPING AND RECEIVING
                  PERSONNEL.
```

3.2 THE ENVIRONMENT DIVISION

The Environment Division is hardware-oriented since it defines the physical characteristics of the computer system that compiles and/or executes the COBOL program. It also links the files to be processed with their respective computer peripherals, such as disk or printer. The format of the Environment Division is shown below.

```
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.  source-computer-entry.
OBJECT-COMPUTER.  object-computer-entry
                 [MEMORY SIZE integer CHARACTERS].
INPUT-OUTPUT SECTION.
FILE-CONTROL.
   SELECT clause
   [ASSIGN clause]
   [ORGANIZATION clause]
   [ACCESS clause]
   [RECORD KEY clause]
[I-O-CONTROL.  SAME AREA clause].
```

The division header and all paragraph names begin at Margin A. The clauses of a paragraph begin either on the same line as the paragraph name, or at Margin B.

Each paragraph can contain a COPY statement that refers to a file containing the entire paragraph. See Chapter 5, for an explanation of the COPY statement.

3.2.1 THE CONFIGURATION SECTION

The CONFIGURATION SECTION states the names of the computers that are to compile the source program and execute the object program.

The SOURCE-COMPUTER paragraph specifies the computer that is to compile the source program, while the OBJECT-COMPUTER paragraph describes the computer on which the compiled program is to process data. The contents of both paragraphs, however, are treated as commentary by the compiler, which just checks to see that a period ends each one.

When compiling or executing a program on a CDX-68 system, this section should be written as shown below:

```
     CONFIGURATION SECTION.
     SOURCE-COMPUTER.  CDX-68.
     OBJECT-COMPUTER.  CDX-68.
```

3.2.2 THE INPUT-OUTPUT SECTION

The INPUT-OUTPUT SECTION provides the information that is needed to control the transmission of data between external storage media and the object program. The FILE-CONTROL paragraph names the files involved in the program and links them with their respective computer peripherals. The optional I-O-CONTROL paragraph describes file overlap information.

The SELECT clause is required as the first entry in the FILE-CONTROL paragraph and names an input or output data file that is used in the COBOL program. The format of this clause is:

```
     SELECT file-name-1
```

File-name is a programmer-supplied name that is used to reference the file for the rest of the program. This name must not be used for any other item in the program.

The ASSIGN clause links a file with the external device that either stores or receives the file. This clause must immediately follow the SELECT clause. The format of this clause is:

```
     ASSIGN TO implementor-1
```

Acceptable implementor-names are:

```
     PRINTER
     DISK diskid:file-name[:suffix]
```

where diskid	= the 8-character disk identification.
file-name	= 8 alphanumeric characters that form the name of the file as recorded in the directory on the disk or diskette. The first character must be alphabetic. All file-names have DF suffixes, with the exception of those whose organization is CODOS-SA.
suffix	= 2 character suffix (e.g. SA) for CODOS-SA type files. If omitted, the default suffix is "DF."

The ORGANIZATION clause states whether the file has a sequential or indexed organization. The format of this clause is:

$$\text{ORGANIZATION IS} \left\{\begin{array}{l}\text{SEQUENTIAL}\\ \text{INDEXED}\\ \text{CODOS-SA}\end{array}\right\}$$

When INDEXED organization is specified, the RECORD KEY clause must also be included. When this clause is omitted, sequential organization is assumed. When CODOS-SA is specified, the file is designated as a CODOS source sequential type file. The number of CODOS-SA type files is limited to eight for a given program.

The ACCESS clause tells how records are obtained from or placed into the file. The format of this clause is:

$$\text{ACCESS MODE IS} \left\{\begin{array}{l}\text{SEQUENTIAL}\\ \text{RANDOM}\\ \text{DYNAMIC}\end{array}\right\}$$

With SEQUENTIAL access, records are obtained or placed in the file one after the other. RANDOM and DYNAMIC modes require a RECORD KEY clause and the assignment of the file to a direct-access device such as disk or diskette; the specified record is located or placed according to its record key. In DYNAMIC mode, the file may be accessed either sequentially or randomly, depending on the I/O statement. For example, a record can be found by its record key in a random read; the records following it can be read sequentially; then, another random read can switch back to random access.

The RECORD KEY clause is required when INDEXED ORGANIZATION, ACCESS IS RANDOM, or ACCESS IS DYNAMIC clauses have been specified. The format of this clause is:

```
RECORD KEY IS dataname [WITH DUPLICATES]
```

Dataname is used by READ and WRITE statements to locate a record in the file. The data item that dataname represents must be part of the record and cannot be more than 60 characters long. The WITH DUPLICATES option permits records that have duplicate record keys to exist in the file.

3.2.3 THE I-O-CONTROL PARAGRAPH

This optional paragraph defines file overlay information. The format of this paragraph is:

```
I-O-CONTROL.
    SAME AREA FOR filename-1 [, filename-2]...
```

The SAME AREA clause causes data areas to overlap for all files mentioned in the clause; therefore, only one of the listed files may be open at a time. A COBOL program may have more than one SAME AREA clause, but a specific filename may appear in no more than one such clause.

CHAPTER 4 - THE DATA DIVISION

The Data Division describes the structure, organization, and characteristics of all data that is manipulated and processed by the program. A maximum of three sections may be contained in this division: the File, Working-Storage, and Linkage Sections. The FILE Section describes the records in each file mentioned in the Environment Division. The Working-Storage Section describes internally-generated data and internal work areas that are used by the program. The Linkage Section describes memory areas used for program-to-program communication.

The organization of the Data Division is:

```
DATA DIVISION.
FILE SECTION.
FD entry for file-name-1
Record description entries for file-name-1.
  .
  .
  .
WORKING-STORAGE SECTION.
Internal independent data item descriptions
Internal grouped item descriptions
 LINKAGE SECTION.
Linkage item descriptions
```

4.1 THE FILE SECTION

The File Section contains two types of entries: FD (file description) entries and record description entries.

4.1.1 FD (FILE DESCRIPTION) ENTRY

The FD entry describes the physical characteristics of one file. For disk files, its format is:

```
     FD file-name-1
        [RECORD CONTAINS clause]
        [LABEL RECORDS clause]
        [DATA RECORD clause].
```

For printer files, the format of the FD entry is:

```
     FD file-name-2
        [LINAGE clause]
        [TOP clause]
        [BOTTOM clause].
```

FD must begin at Margin A, and any additional lines of this entry begin at Margin B. The file-name is identical to the file-name listed in the Environment Division following the word SELECT.

The RECORD CONTAINS clause states the file's record length. The Codex COBOL compiler accepts records that have a maximum length of 502 characters. Since the compiler calculates the record's length from its record description entry, this clause is not required, but is useful as a documentation aid. The format of this clause is:

```
RECORD CONTAINS integer-1 CHARACTERS
```

The LABEL RECORDS clause is not required by Codex's COBOL compiler, which treats it as a comment entry. The format of this clause is:

$$\text{LABEL} \begin{Bmatrix} \text{RECORD IS} \\ \text{RECORDS ARE} \end{Bmatrix} \begin{Bmatrix} \text{STANDARD} \\ \text{OMITTED} \end{Bmatrix}$$

All file labels are internal to the CDX-68 file management system.

The DATA RECORD clause states the programmer-supplied name for each record in the file. The format of this clause is:

$$\text{DATA} \begin{Bmatrix} \text{RECORD IS} \\ \text{RECORDS ARE} \end{Bmatrix} \text{data-name-1 [data-name-2]...}$$

Since a record-name is presented in each record description entry, this clause is not required, but is useful as a documentation aid. The records in a given file may have different lengths and formats, but their order in a DATA RECORD clause is not important. Only one record from a file may be processed at a time.

For example, if one record is read from a file, and another record is read from the same file, the second record replaces the first in computer memory.

```
DATA RECORD IS PRINT-RECORD
DATA RECORDS ARE MASTER-RECORD, DETAIL-1, DETAIL-2
```

The LINAGE clause states the maximum number of lines that are printed per page. The format of this clause is:

```
LINAGE IS integer
```

The maximum number is 60 lines. When this clause is omitted, the compiler assumes that each page contains 60 printed lines.

The TOP clause specifies the size of the top margin in line units. The format of this clause is:

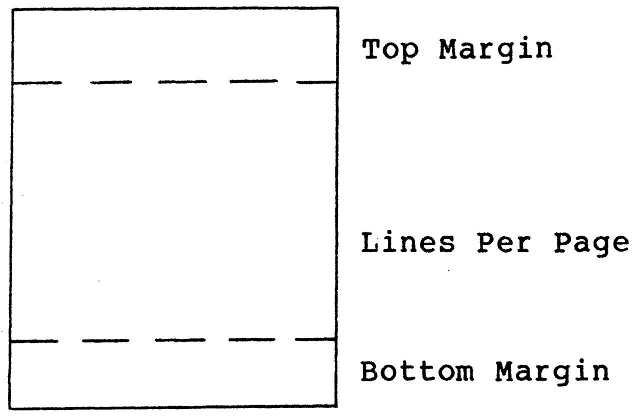
```
TOP IS integer
```

When this clause is omitted, the top margin is assumed to be 6 lines.

The BOTTOM clause reveals the size of the page's bottom margin in line units. The format of this clause is:

```
BOTTOM IS integer
```

This clause is unnecessary if the LINAGE clause has been included. When both LINAGE and BOTTOM clauses are omitted, the default bottom margin is 0 lines. For example:



4.1.2 RECORD DESCRIPTION ENTRIES

The record description delineates the structure, organization, and characteristics of the file's records. Each record within a file must have its own record description, as shown in the following example.

```
DATA DIVISION.
FILE SECTION.
FD DISK-FILE, RECORD CONTAINS 80 CHARACTERS,
   LABEL RECORDS ARE STANDARD, DATA RECORDS ARE
   MASTER-RECORD, DETAIL-1, DETAIL-2.
(Record description entries for MASTER-RECORD)
(Record description entries for DETAIL-1)
(Record description entries for DETAIL-2)
```

Before presenting the format for a record description entry, data organization within a record should be reviewed. Records are usually divided into data items or fields that contain categories of information to be processed. A COBOL field is either a group or elementary item. Group items are divided into smaller fields; elementary items are not subdivided. Consider the structure of the record shown below.

RECORD-1

ID-NUM	NAME			ADDRESS				PHONE
	LAST	FIRST	M-I	STREET	CITY	STATE	ZIP	

RECORD-1 is directly divided into four data items: two elementary and two group. NAME and ADDRESS are group items since they are further subdivided: NAME into LAST, FIRST, and M-I; ADDRESS into STREET, CITY, STATE, and ZIP. ID-NUM and PHONE are not subdivided, therefore they are elementary items.

A COBOL record description specifies the name, order from left to right, and size of the record's data items, and tells how they are related to each other. For example, RECORD-1's data items could be listed in the following manner:

```
RECORD-1                  record's name
     ID-NUM               elementary item
     NAME                 group item
          LAST
          FIRST           elementary items
          M-I
     ADDRESS              group item
          STREET
          CITY            elementary items
          ZIP
     PHONE                elementary item
```

To make this list understandable to the computer, level numbers are used to show the data item hierarchy. Level numbers 01 through 15 may be used in a record description entry for the Codex COBOL compiler. Level 01 is always used for the record's name, and starts at Margin A.

A description of RECORD-1 that shows the level numbers is:

```
01 RECORD-1.
02 ID-NUM       (elementary item description).
02 NAME.
   03 FIRST     (elementary item description).
   03 LAST      (elementary item description).
   03 M-I       (elementary item description).
02 ADDRESS.
   03 STREET    (elementary item description).
   03 CITY      (elementary item description).
   03 STATE     (elementary item description).
   03 ZIP       (elementary item description).
02 PHONE        (elementary item description).
```

The period after the words RECORD-1, NAME, and ADDRESS means that they are group items and therefore divided into smaller fields. Level numbers 02 and 03 start at either Margins A or B; indenting them might make the program easier to read.

Note that the direct immediate subdivisions of RECORD-1 have level number 02, and that the subdivisions of each group item have level number 03. Other level numbers could have been used provided that the level number of a group item is smaller than that of its elementary items.

The compiler regards all data items that follow a group item as belonging to that group item until the compiler encounters a level number equal to or greater than the level number of the group item.

The format and order of a record description entry is:

```
level-number  { data-name }  [REDEFINES clause]
              { FILLER    }  [COPY statement]
                             [PICTURE clause]
                             [USAGE clause]
                             [BLANK WHEN ZERO clause]
                             [JUSTIFIED RIGHT clause]
                             [LINE clause]
                             [COLUMN clause]
                             [OCCURS clause]
```

FILLER indicates that the data item either contains no information, or that the information is not referenced directly in the program.

The REDEFINES clause is used with either group or elementary items in the file section that DO NOT HAVE a level number of 01. A level 01 REDEFINES clause is permitted in the Working Storage Section. This clause allocates the same computer memory space for different items at different times during program execution. It may also provide an alternate grouping or description of the same data. A REDEFINES clause in the file section at level 01 can cause PROGRAM OVERFLOW errors or erroneous data or runtime errors. A record description entry's format with this clause is:

```
     level-number data-name-1 REDEFINES data-name-2
```

The level numbers of data-name-1 and data-name-2 must be identical. These data-names may be a group or an elementary item.

The following are some guidelines for the use of the REDEFINES clause. These guidelines reference the example:

```
     nn A REDEFINES B . . .
```

1. B must not be a redefinition itself.

2. "nn" must not be level 01 within the file section (level 01 redefinition is permitted within the working storage section).

3. The size of A must equal the size of B.

4. The level numbers of A and B must be equal.

5. "Value" clauses are not allowed within group or elementary items which are a redefinition of another group.

6. The program may contain multiple 01 record definitions for a file following the "FD" entry in the file section. The use of multiple 01 levels implicitly defines several records which share the same memory area (a redefinition by another name).

The description selected for an item that is redefined depends on the reference made to it. For example, if B redefines A, the statement MOVE X TO A moves X to the area described as A; MOVE Y TO B moves Y to the same area in computer memory, but which has been redefined as B. The description of this memory area depends on the order in which such statements are executed. For example:

```
05   PRICE-NOT-NUMERIC REDEFINES PRICE-IN
05   NON-ALPHABETIC-DATA REDEFINES DATA-IN
```

The COPY statement is described in the Compiler-Directing Statements section of Chapter 5. Basically, this statement enables prewritten statements to be included in several programs.

The PICTURE clause describes the length and character type of an elementary item. It may also specify an operational sign, an assumed decimal point, and editing characters. The format of this clause is:

$$\left\{\begin{matrix}\text{PICTURE}\\ \text{PIC}\end{matrix}\right\}\ \text{[IS] character-string}$$

The character string must be no more than 30 characters long.

COBOL classifies all data as being alphabetic, alphanumeric, or numeric. Alphabetic data contains only the letters of the English alphabet and spaces. Alphanumeric data is formed from any printable characters. Numeric data consists of digits 0 through 9 and a positive or negative sign.

Numeric data may have an assumed decimal point and/or operational sign in its PICTURE clause. A numeric field may be edited by moving it to a field whose PICTURE clause causes the insertion of a decimal point, comma, or dollar sign, as well as suppressing the printing of leading zeroes. The resultant edited item is always classified as alphanumeric.

The three categories of PICTURE characters that describe data are:

A	(alphabetic data)	Data character symbols
X	(alphanumeric data)	
9	(numeric data)	
S	(signed numeric field)	Operational symbols
V	(assumed decimal point)	
B	(space)	Editing symbols
0	(zero)	
+	(plus)	
-	(minus)	
CR	(credit)	
DB	(debit)	
Z	(zero suppression)	
*	(check protection)	
$	(dollar sign)	
,	(comma)	
.	(period or decimal point)	

Data character symbols say that the item is either alphabetic, alphanumeric, or numeric. Alphanumeric fields cannot be used in arithmetic operations. The number of data characters in the PICTURE character string must equal the length of the data item as shown in the following examples:

Field Length	Data Type	Character String
5 characters	Alphabetic	AAAAA or A(5)
4 characters	Alphanumeric	XXXX or X(4)
6 characters	Numeric	999999 or 9(6)

As seen by these examples, parentheses are a shortcut to showing the number of times that the data character is repeated in the character string.

Operational symbols show the sign and assumed decimal point position in a numeric item. Only one S and/or V may appear in a PICTURE clause. When used, S must be the leftmost character of the string and indicates the presence of positive or negative data. Unsigned numeric fields are assumed to be positive. V shows an assumed decimal point position in a numeric item, and cannot be the rightmost character. Both S and V do not actually take up space in the data item. For example, S99V99 is a four-character numeric item. This sign will take up one-half byte in packed numeric and will cause an additional byte if the data size is an even number.

Before data is printed, it is often desirable to edit it to make it more readable. Editing symbols specify the type of editing to be done before an elementary item is printed. The editing process inserts certain characters and/or suppresses the printing of others. An edit is performed by moving the data from a source field to a receiving field whose PICTURE clause contains the editing characters.

Parentheses are not allowed in a PICTURE clause character string that contains editing characters. Commas, decimal points, slashes (/), spaces (B), or zeroes should be written in the string where they appear in the output item. An example of an edited move is shown below.

Field	PICTURE	Data
Source	999V99	54321
Receiving	999.99	543.21

The receiving field in the above example has 6 characters to include the printed decimal point; the source had only five.

A decimal point can appear only once in a PICTURE character string and cannot be the right-most character.

Z's replace leading zeroes with spaces when printing numeric items. A Z is written in each numeric position where a leading zero is to be suppressed by a space. Examples of zero suppression are shown below:

Source PICTURE	Source Data	Receiving PICTURE	Received Data
9999V99	0123456	ZZ,ZZZ.99	1,234.56
999V99	00123	ZZZ.99	1.23
99V99	0000	ZZ.99	.00

Asterisks (*) replace leading zeroes with asterisks. This process is often used when printing the dollar amount on a check. Examples of these are shown below:

Source PICTURE	Source Data	Receiving PICTURE	Received Data
99999V99	0195430	**,***.99	*1,954.30
999V99	00575	***.99	**5.75
999V99	00850	***.99	**8.50

Z's and asterisks are often used with the fixed dollar sign ($) when printing dollar amounts. To place a fixed dollar sign in an item, write one dollar sign ($) as the string's left-most character. Examples of such items are shown below:

Source PICTURE	Source Data	Receiving PICTURE	Received Data
99999V99	0154025	$ZZ,ZZZ.99	$ 1,540.25
999V99	00575	$***.99	$**5.75
99V99	0005	$ZZ.99	$ .05

The dollar sign ($) can also specify an item that has a floating dollar sign. The difference between a fixed and a floating dollar sign is shown below.

Fixed Dollar Sign PICTURE $ZZ,ZZZ.99	Floating Dollar Sign PICTURE $$$,$$$.99
$78,359.25	$78,359.25
$ 482.89	$482.89
$ 1.50	$1.50
$ .02	$.02

Note that there must be one more floating dollar sign than the number of character positions through which it is to float.

A fixed-position plus (+) or minus (-) sign may be written as either the first or last character of a PICTURE string, but cannot be used with the dollar sign. A minus editing symbol causes a minus sign to be printed in that position for negative data, and a space in that position for positive data. A plus symbol prints a plus sign in that position for positive data, and a minus sign for negative data. Unsigned data is always considered to be positive. A few examples are shown below.

Source PICTURE	Source Data	Receiving PICTURE	Received Data
S999	123	-999	-123
S99V99	4995	-ZZ.99	49.95
S99	76	Z9	76
S99999	01852	+ZZ,ZZ9	- 1,852

Plus and minus symbols can also be used as floating editing symbols. The rules for writing these symbols are the same as for writing floating dollar signs, except that plus and minus symbols are used, as shown in the following examples.

Source PICTURE	Source Data	Receiving PICTURE	Received Data
S9999V99	000542	-----.99	-5.42
S99V999	06985	---.999	-6.985
S99V99	0005	+++.99	+.05
S99	48	++9	+48

Credit (CR) and debit (DB) symbols occupy two positions each and may appear only as the rightmost characters of a PICTURE string. For negative data, the edited result contains CR or DB as indicated. For positive data, CR or DB is replaced by spaces. A few examples are shown below.

Source PICTURE	Source Data	Receiving PICTURE	Received Data
S99999	12345	ZZ,ZZ9CR	12,345CR
S99V99	1580	ZZ.99CR	15.80CR
S9V99	349	Z.99DB	3.49DB
S99	23	Z9DB	23DB

The USAGE clause specifies the form in which data is stored in the computer and can be written at any level. When this clause is used with a group item, it applies to all of the group's elementary items. Additionally, the USAGE clause of an elementary item cannot contradict the USAGE clause of the group item to which it belongs.

The format of the USAGE clause is:

$$\text{USAGE IS} \quad \left\{ \begin{matrix} \text{DISPLAY} \\ \left[\begin{matrix} \text{COMPUTATIONAL} \\ \text{COMP} \end{matrix} \right] \\ \text{INDEX} \end{matrix} \right\}$$

DISPLAY means that the item is stored in ASCII format where one character is stored in each byte; the leftmost byte of a numeric item can contain an operational sign in addition to a digit. COMPUTATIONAL defines a packed decimal data item whose length is specified by its PICTURE clause. It can only be used for unedited numeric items. COMPUTATIONAL should be used only to save memory or file space.

INDEX defines an item that is called an index data item and will contain a value that corresponds to an occurrence number of a table element. Index data items must be elementary data items. Since USAGE IS INDEX totally defines the internal representation of the data, a PICTURE clause is not used with an index data item.

The BLANK WHEN ZERO clause may be used only in conjunction with an edited numeric PICTURE clause. When the source item has a value of zero and the BLANK WHEN ZERO clause is used for the receiving numeric edit field, the edited item contains all spaces. The format of this clause is:

BLANK WHEN ZERO

The JUSTIFIED RIGHT clause is used only with alphabetic or alphanumeric elementary items. The format of this clause is:

$$\left\{ \begin{matrix} \text{JUSTIFIED} \\ \text{JUST} \end{matrix} \right\} \quad \text{RIGHT}$$

This clause is applicable only to alphabetic or alphanumeric items. Normally, when data is moved into an alphabetic or alphanumeric field, the source data is aligned at the leftmost character position of the receiving data item and moved with space fill or truncation on the right.

When the receiving data item is described with the JUSTIFIED clause and the sending data item is larger than the receiving data item, the leftmost characters are truncated.

When the receiving data item is described with the JUSTIFIED clause and is larger than the sending data item, the data are aligned at the rightmost character position in the data item with other characters space-filled.

The OCCURS clause eliminates the need for separate entries of repeated data items and supplies information necessary for the use of subscripts as described in Chapter 5, Table-Handling Statement section. The format of this clause is:

```
OCCURS integer [TIMES]
    [INDEXED BY index-name-1 [, index-name-2]...]
```

Examples:

```
     OCCURS 10
     OCCURS 10 TIMES
     OCCURS 5 TIMES INDEXED BY RATE-INDEX
```

4.2 THE WORKING STORAGE SECTION

The Working-Storage Section describes data items that store either intermediate results developed during program execution or literals that are needed by the program. Such items may represent either independent or grouped work areas. The basic organization of the Working-Storage Section is:

```
     WORKING-STORAGE SECTION.
     Independent item description entries.
     Grouped item description entries.
```

4.2.1 INDEPENDENT AREAS IN WORKING-STORAGE

An independent area is a single data item that stands alone and has the level number of 77. It is not subdivided, and it is also not a subdivision of another item. The format of an independent Working-Storage area is:

```
     77 data-name [USAGE clause]
                  [PICTURE clause]
                  [JUSTIFIED clause]
                  [VALUE clause].
```

All of these clauses have been described previously in the File Section of the Data Division, excepting the VALUE clause.

The VALUE clause defines the initial or constant value of the item and has the format:

```
     VALUE IS literal
```

The literal must belong to the same data category as the character string in the item's PICTURE clause: a non-numeric literal for an alphabetic or alphanumeric PICTURE; a numeric literal for a numeric PICTURE. A numeric literal's value must lie within the range specified by the PICTURE or USAGE clauses. For example:

```
     VALUE IS '      NAME '
     VALUE IS +0.85
     VALUE ZERO
```

4.2.2 GROUPED AREAS IN WORKING-STORAGE

A grouped area is formed from several data items that are grouped together in the same way that a record's items are related to each other, and is often used for storing tables, report headers, and screen display formats. The rules for writing grouped area description entries are the same as for writing record description entries.

The format of a grouped area in working-storage is:

```
level-number data-name [REDEFINES clause]
                       [PICTURE clause]
                       [USAGE clause]
                       [LINE clause]
                       [COLUMN clause]
                       [BLANK WHEN ZERO clause]
                       [JUSTIFIED RIGHT clause]
                       [VALUE clause]
                       [OCCURS clause].
```

All of these clauses have been defined in this chapter with the exception of the LINE and COLUMN clauses which are used in describing a screen format.

A 01 level group area becomes a CRT format description when a LINE or COLUMN clause appears before the first VALUE clause in the group area description. Subsequent entries do not have to include LINE or COLUMN clauses unless they are needed to position the data item. Refer to Chapter 6 for more detail on CRT control.

Before each screen data description entry, the compiler generates the column FILLER entry that contains CRT control codes. Group area sizes are automatically increased by the compiler to include these internally generated codes.

The basic description format for a data item that is to be displayed on the screen is:

```
level-number FILLER PICTURE clause
                    [LINE clause]
                    [COLUMN clause]
                    VALUE clause.
```

The LINE clause indicates the line number on which the associated data item is to be displayed. The format of this clause is:

$$\text{LINE IS} \begin{Bmatrix} \text{integer} \\ \text{NEXT PAGE} \end{Bmatrix}$$

The integer specifies the line on which the data item will be displayed. NEXT PAGE clears the screen prior to writing the new data items at the top of the display; this option is used only with a 01 level data item.

If the LINE clause is omitted, the data is displayed on the same line as the previous data item if the new column number is larger than the column number of the previous data item. Otherwise, the new data item is displayed on the following line. For example:

```
LINE IS 5
LINE NEXT PAGE
```

The COLUMN clause states the number of the screen column in which the leftmost character of the data item will be displayed. The format of this clause is:

```
COLUMN IS integer
```

The integer must not be 1, since this column is used for CRT control codes. For example:

```
COLUMN IS 38
COLUMN 65
```

When specifying a column position for a data item, leave at least a one-column separation between items for the storage of internally generated screen control codes. If the COLUMN clause is omitted, the data is displayed starting in column two of the line. If both the LINE and COLUMN clauses are omitted, the new data is separated by one column from the end of the previous item.

Video attributes are automatically generated for screen data descriptions based on the type of the literal contained in the VALUE clause. When a numeric or non-numeric literal is specified, the data item is a protected field, meaning that its content cannot be altered. A figurative constant such as ZEROES or SPACES results in an underlined and unprotected data item.

When a hexadecimal constant is specified in the VALUE clause, the data item is composed entirely of screen control codes; these codes are listed in Appendix B. The default line and column positions are not changed.

4.2.3 STORING TABLES IN WORKING-STORAGE

Tables that are used by COBOL programs may have one, two, or three levels. These tables may be stored in the Working-Storage area by using the REDEFINES, OCCURS, and sometimes VALUE clauses as described in this section.

Table codes and values can be either stored in the data description through the VALUE clause, or can be loaded during program execution by Procedure Division statements. In this discussion, all table values will be contained in VALUE clauses.

A single-level table has one variable or code that has a corresponding value as shown in the following example.

Sale Code	Bonus
1	100
2	250
3	450
4	700

These values are stored in one of two ways as follows:

Method 1:
```
          01  BONUS-TABLE-VALUES.
              05  FILLER  PIC 999 VALUE 100.
              05  FILLER  PIC 999 VALUE 250.
              05  FILLER  PIC 999 VALUE 450.
              05  FILLER  PIC 999 VALUE 700.
```

Method 2:
```
          01  BONUS-TABLE-VALUES.
              05  FILLER  X(12) VALUE '100250450700'.
```

Method 1 is easier to read and thus is a better documentation aid than Method 2. Also, Method 2 cannot be used if COMP usage is required.

Since the VALUE clause may not be used in conjunction with the REDEFINES and OCCURS clauses, the next and last step is to write the following.

```
01  BONUS-TABLE REDEFINES BONUS-TABLE-VALUES.
    05  BONUS OCCURS 4 TIMES PIC 999.
```

The above table will be searched by using subscripts in Procedure Division statements (see Chapter 5, Table-Handling Statements).

This method works when the variable's code goes from 1 to the last number with no gaps. In many tables however, the code does not do this, as shown below.

Part Number	Retail Price
15	2.50
17	0.45
31	6.30
85	1.00

Notice that only the code needs to be in ascending order; the price does not. The following method stores such a table that is searched with subscripts.

```
01  PART-TABLE-VALUES.
     05  FILLER  PIC X(5) VALUE '15250'.
     05  FILLER  PIC X(5) VALUE '17045'.
     05  FILLER  PIC X(5) VALUE '31630'.
     05  FILLER  PIC X(5) VALUE '81100'.
01  PART-TABLE REDEFINES PART-TABLE-VALUES.
     05  PART-TABLE-ENTRY OCCURS 4 TIMES.
         10  PART-NUMBER  PIC 99.
         10  RETAIL-PRICE PIC 9V99.
```

To convert this table to an indexed one, add an INDEXED BY clause to the PART-TABLE-ENTRY OCCURS 4 TIMES statement. In this example, INDEXED BY PART-INDEX could be used. Refer to the Table-Handling Statements section in Chapter 5 for a further explanation of an indexed table.

In two- and three-level tables, the value chosen depends on two and three variables, respectively. Storing such tables requires OCCURS clauses within OCCURS clauses. For example, consider the following two-level table.

		Product Code			
	01-03	02	12	22	40
Quantity	04-06	05	14	27	46
Sold	07-09	08	16	33	52
	10-12	11	20	37	58
	13-15	14	24	42	64

The following description stores this table.

```
01  COMMISSION-TABLE-VALUES.
    05   FILLER  PIC X(12) VALUE '010302122240'.
    05   FILLER  PIC X(12) VALUE '040605142746'.
    05   FILLER  PIC X(12) VALUE '070908163352'.
    05   FILLER  PIC X(12) VALUE '101211203758'.
    05   FILLER  PIC X(12) VALUE '131514244264'.
01  COMMISSION-TABLE REDEFINES COMMISSION-TABLE-VALUES.
    05   QUANTITY-SOLD OCCURS 4 TIMES
                        INDEXED BY QUANTITY-INDEX.
         10  LO-QUANTITY  PIC 99.
         10  HI-QUANTITY  PIC 99.
         10  PRODUCT-CODE OCCURS 5 TIMES
                          INDEXED BY PRODUCT-INDEX.
```

To describe this table for subscripting, simply omit the INDEXED BY clauses. An indexed table may still be accessed by the use of subscripts. A three-level table is described in the same way as a two-level table, except that a third variable is added whose level number would be 15 (using the sequence in the above table) and which also has an OCCURS clause to show how many values it has in the table.

4.3 THE LINKAGE SECTION

The Linkage Section provides the user with a maximum of 256 internal memory bytes for program-to-program communication. Information in this common area may be accessed by the COBOL program being executed or by a program or subroutine that has been loaded by a CALL or UCALL statement in the Procedure division.

Information stored in the Linkage Section can be referenced based on physical location. The following example shows how the Linkage Section can be used to send parameters to an assembly routine.

COBOL Program:

```
     IDENTIFICATION DIVISION.
              .
              .
              .
     LINKAGE SECTION.
     01 EXAMPLE.
        02 EX1 PIC X.
        02 EX2 PIC XX.
               .
               .
               .
     PROCEDURE DIVISION.
               .
               .
               .
          MOVE A-PARAM TO EX1.
          MOVE X-PARAM TO EX2.
          UCALL USERPROG.
               .
               .
               .
             (end)
```

Assembly Routine: Routine is entered with X register containing address of first parameter.

```
           TTL   USEPROG
           OPT   REL
     START LDAA  0,X       GET A-PARAM
           LDX   1,X       GET X-PARAM
            .
            .
            .
          (end)
```

CHAPTER 5 - THE PROCEDURE DIVISION

The Procedure Division of a COBOL source program specifies a sequence of actions that must be taken to solve a given problem. The following units may be used to form the Procedure Division: statements, sentences, paragraphs, and sections.

A statement consists of a COBOL verb followed by reserved words and data-names or literals. The three basic types of statements are: imperative, conditional, and compiler-directing.

An imperative statement specifies an action that the computer will unconditionally perform, such as ADD, GO TO, and OPEN.

A conditional statement describes a condition that is tested to determine which of several logic paths should be taken, such as in the IF statement.

A compiler-directing statement tells the compiler to take certain actions at compilation time, such as occurs when a COPY statement is included.

A sentence is a series of one or more statements, and is terminated by a period. A semicolon may be used to separate the statements within a sentence.

A paragraph consists of one or more sentences, and is identified by a procedure-name that begins in Margin A and ends with a period.

A section is formed from one or more paragraphs, and has a section name which consists of a procedure-name that is followed by the word SECTION and a period. A section name begins at Margin A and appears on a line by itself.

The Procedure Division is rather loosely structured since it consists of a series of paragraphs that may be also grouped into sections. Statements are executed in the order in which they appear, but the execution sequence can be altered by sequence control statements.

Paragraph and section names are not required in a COBOL program, but are used when sequence control statements refer to them in order to alter the sequence of program execution.

The functional categories of Procedure Division statements are: input/output, data manipulation, arithmetic, conditional, sequence control, table handling, and compiler directing.

5.1 INPUT/OUTPUT STATEMENTS

Input/output statements are directed to computer peripherals, such as disks that contain data files. An input data file is opened, a record is read and processed, and then the output is sent to an output file. Data sent to the printer is considered to be an output print file in COBOL. When all processing has been completed for a file, it is closed.

The input/output statements are: OPEN, START, READ, WRITE, REWRITE, DELETE, DISPLAY, ACCEPT, and CLOSE.

5.1.1 THE OPEN STATEMENT

An OPEN statement must be the first statement that is performed on any file. This statement however, does not actually obtain or dispatch data: READ and WRITE statements do that. The format of the OPEN statement is:

```
OPEN  [INPUT file-name-1 [, file-name-2]...]
      [OUTPUT file-name-1 [, file-name-2]...]
      [I-O file-name-1 [, file-name-2]...]
      [EXTEND file-name-1 [, file-name-2]...]
```

Specify one of the above options for each file in the program. The OUTPUT option is equivalent to the CLOSE filename WITH DELETE statement since neither one saves data. To save the output data, select the I-O option for an indexed file. To add new records at the end of an existing sequential file, select the EXTEND option.

A file must first be closed before a second OPEN statement can be executed on that file. For example:

```
OPEN INPUT TRANSACTION-FILE OUTPUT PRINT-FILE
OPEN EXTEND PARTS-FILE
OPEN I-O MASTER-FILE
.
.
.
CLOSE PARTS-FILE  TRANSACTION-FILE
OPEN INPUT PARTS-FILE
.
.
.
```

FILE ACCESS MODE	OPEN MODE / STATEMENT	SEQUENTIAL I	SEQUENTIAL O	SEQUENTIAL I-O	SEQUENTIAL E	INDEXED I	INDEXED O	INDEXED I-O	INDEXED E	CODOS-SA I	CODOS-SA O	CODOS-SA I-O	CODOS-SA E
SEQUENTIAL	READ	X		X		X		X		X			
	WRITE		X		X		X	X			X		X
	REWRITE			X				X					
	START												
	DELETE							X					
RANDOM	READ					X		X					
	WRITE						X	X					
	REWRITE							X					
	START												
	DELETE							X					
DYNAMIC	READ					X		X					
	WRITE						X	X					
	REWRITE							X					
	START					X		X					
	DELETE							X					

I = INPURT
O = OUTPUT
I-O = INPUT-OUTPUT
E = EXTEND

Table 5-1. I/O COMMANDS

5.1.2 THE START STATEMENT

The START statement provides the means for positioning the current record pointer within an indexed file for record retrieval. This statement requires that its associated file be open in INPUT or I-O mode. The format of the START statement is:

```
                                ⎧ EQUAL TO              ⎫
                                ⎪ =                     ⎪
START file-name [KEY IS         ⎨ GREATER THAN          ⎬data-name]
                                ⎪ >                     ⎪
                                ⎩ NOT ⎧LESS  THAN⎫      ⎭
                                      ⎩  <       ⎭
                  [INVALID KEY imperative-statement]
```

The file must have an indexed organization and dynamic access mode and must have been opened in INPUT or I-O mode.

The data-name is either the file's RECORD KEY or an alphanumeric data item that is subordinate to the RECORD KEY and starts in the same character position as does the RECORD KEY.

The EQUAL option compares the data-name's current value with the corresponding field in the file's records. When a match is found, the current record pointer is positioned at that record.

The other KEY options operate in a similar manner to the EQUAL TO option; the current record pointer is placed at the first record that satisfies the comparison. If no record does so, an INVALID KEY condition exists, and the position of the current record pointer is undefined.

When the KEY clause is omitted, the compiler acts as if the statement:

```
     START file-name KEY IS EQUAL TO data-name
```

had been written, where the data-name is equal to the file's RECORD KEY. For example:

```
START CUSTOMER-FILE
START PERSONNEL-FILE KEY = EMPLOYEE-NUMBER
          INVALID KEY PERFORM NOT-IN FILE
```

5.1.3 THE READ STATEMENT

The READ statement makes a record available for processing whose file has been opened in INPUT or I-O mode. When a READ statement is executed, the selected logical record is accessible in the logical record area in computer memory as defined by its record description entry. The record is available there until a subsequent READ or CLOSE statement is executed for that file.

When multiple record descriptions follow a File Description (FD) entry, the programmer is responsible for recognizing which record is present in the logical record area at any time. The READ statement has two formats:

Format 1:

```
     READ file-name [NEXT] [INTO identifier]

               [AT END imperative-statement]
```

Format 2:

```
     READ file-name [INTO identifier]

          INVALID KEY imperative-statement
```

Before a Format 1 statement can be executed, the current record pointer must be positioned by the prior, successful execution of an OPEN, START, or READ statement. Format 1 provides for the sequential processing of a sequential or indexed file by making available one of the following: the next logical record from a sequential file; the next record with the next higher RECORD KEY value in the collating sequence for a sequentially-accessed indexed file. For example:

```
READ DISK-FILE-1
READ DISK-FILE-1 NEXT
```

For indexed files whose access mode is dynamic, the NEXT option must be specified for the sequential retrieval of the next logical record in the file. A series of imperative statements in the AT END option are executed when an end-of-file status is detected. For example:

```
READ DISK-FILE-1 NEXT INTO WORK-RECORD
READ DISK-FILE-1 INTO WORK-RECORD
```

A sequential file whose AT END condition has been recognized cannot be READ again until successful CLOSE and OPEN statements have been executed for it. An indexed file whose AT END condition is recognized also cannot be READ again until one of the following actions has been successfully executed for it: a CLOSE statement followed by an OPEN statement; a Format 2 READ statement for a file in dynamic access mode; or a START statement. For example:

```
          READ DISK-FILE-1 AT END GO TO END-JOB
```

Format 2 is used for random processing of indexed files in random or dynamic access mode. This statement compares the file's RECORD KEY value to the content of the corresponding key field in the file's records. When a match is found, the current record pointer is positioned at this record which makes it available for processing. If no match is found, an INVALID KEY condition exists, and the specified imperative statement is executed. For example:

```
READ MASTER-FILE INVALID KEY PERFORM
          NO-RECORDS
READ CUSTOMER-FILE INTO WORK-RECORD
          INVALID KEY PERFORM NOT-IN-FILE
```

The INTO option of both formats is equivalent to a READ followed by a MOVE since the record becomes available in both the file's record area and in the specified identifier area. When the file has records with varying lengths, the longest length is assumed for the moved record.

5.1.4 THE WRITE STATEMENT

The WRITE statement releases a logical record to an output file. The released logical record may or may not still exist in the logical record area for the file. An OPEN OUTPUT, OPEN EXTEND, or OPEN I-O statement must have been performed for the file before a WRITE statement can be executed for it. The two formats of this statement are:

```
Format 1:
WRITE record-name  [FROM identifier-1]
      [ {BEFORE}  ADVANCING  {identifier-2 lines} ]
      [ {AFTER }             {integer LINES     } ]
      [                      {PAGE              } ]
Format 2:
WRITE record-name  [FROM identifier-1]

      INVALID KEY imperative-statement
```

A WRITE statement with the FROM option is equivalent to a MOVE identifier-1 TO record-name statement that is followed by a WRITE record-name statement.

Format 1 is used for files opened for sequential access. The ADVANCING option applies to files containing output to be printed, where the line is printed BEFORE or AFTER advancing a nonnegative number of lines or to the top of the next page.

For example:

```
WRITE PRINT-RECORD
WRITE PRINT-RECORD FROM WORK-RECORD
WRITE PRINT-RECORD AFTER ADVANCING 2 LINES
```

Format 2 is used for all other files. The imperative statements in the INVALID KEY clause are executed when the file either has no space for the record or has been opened for OUTPUT or I-O and a record corresponding to the contents of the RECORD KEY already exists in the file. For example:

```
WRITE PROCESSED-RECORD INVALID KEY GO TO
        NO-RECORD
WRITE PROCESSED-RECORD FROM WORK-RECORD INVALID
        KEY GO TO NOT-IN-FILE
```

NOTE: When writing records to a CODOS-SA file, the record area must be one byte larger than the actual length of the data. The system uses this position to store a "CR" as the record termination character.

5.1.5 THE REWRITE STATEMENT

The REWRITE statement rewrites a previously-read record to the output file. A file must have been opened in I-O mode before a REWRITE can be executed for it. The format of this statement is:

```
REWRITE record-name [FROM identifier];

     INVALID KEY imperative-statement
```

The INVALID KEY clause is performed when the contents of the associated RECORD KEY data item are found invalid.

For sequential files, the REWRITE statement rewrites the last record read.

For indexed sequential files in sequential access mode or with duplicate records, the REWRITE statement rewrites the last record read when the record's key field value is equal to the RECORD KEY value.

If the indexed sequential file has no duplicate records, the REWRITE statement rewrites the record whose key field value is the same as the RECORD KEY value.

Once the REWRITE is performed, the logical record rewritten may or may not still exist in the logical record area for the file. For example:

```
REWRITE MASTER-RECORD INVALID KEY PERFORM NOT-HERE
REWRITE MASTER-RECORD FROM WORK-RECORD INVALID
          KEY PERFORM NOT-IN-FILE
```

5.1.6 THE DELETE STATEMENT

The DELETE statement deletes a record from the output file. A file must have been opened in I-O mode before a DELETE can be performed on it. The format of this statement is:

```
DELETE file-name; INVALID KEY imperative-statement
```

The INVALID KEY clause is executed when the record corresponding to the RECORD KEY value is not found in the file.

For sequential files, the DELETE statement deletes the last record read.

For indexed sequential files in sequential access mode or with duplicate records, the DELETE statement deletes the last record read if its key field value is equal to the RECORD KEY value.

If the indexed sequential file has no duplicate records, the DELETE statement deletes the record whose key field value is the same as the RECORD KEY value. For example:

```
DELETE MASTER-FILE; INVALID KEY PERFORM NO-RECORD
DELETE UPDATE01; INVALID KEY PERFORM NOT-IN-FILE
```

5.1.7 THE DISPLAY STATEMENT

The DISPLAY statement writes data to the screen. The format of this statement is:

$$\text{DISPLAY} \left\{\begin{array}{l}\text{@(row,column)}\\ \text{identifier-1}\\ \text{literal-1}\end{array}\right\} \quad , \left[\left\{\begin{array}{l}\text{@(row,column)}\\ \text{identifier-2}\\ \text{literal-2}\end{array}\right\}\right] \ldots$$

The @(row,column) clause positions the CRT cursor, where row and column are numeric literals or items. If this clause is omitted, a carriage return/line feed pair is sent to the display station.

When a DISPLAY statement contains more than one operand, the specified items and/or literals are displayed consecutively with no spaces between them unless specified. The line's remaining positions at the end of the data transfer are left unchanged.

A literal in a DISPLAY statement may be numeric, nonnumeric, or a hexadecimal constant as shown in the following example.

```
     DISPLAY @(3,25), $8085, DATA-1, $84F3.
```

Refer to Chapter 6 for additional information on CRT control.

5.1.8 THE ACCEPT STATEMENT

The ACCEPT statement sends unprotected data from the screen into computer memory. The format of this statement is:

```
     ACCEPT identifier-1 [, identifier-2]...
```

The identifier must be an unedited or group item. Refer to Chapter 6 for additional information on CRT control. For example:

```
ACCEPT SCREEN-1
ACCEPT IDENTIFICATION-D, TRANSACTIONS-D, TOTAL-D
```

5.1.9 THE CLOSE STATEMENT

The CLOSE statement ends the processing of a file, and performs the standard closing procedures on it. The format of this statement is:

```
     CLOSE file-name [WITH DELETE]...
```

A file must be opened before it can be closed. Once closed, a file may not be referenced again until another OPEN statement is performed for it. When the WITH DELETE option is specified, all records in the file are deleted. For example:

```
     CLOSE DISK-FILE
     CLOSE DISK-FILE WITH DELETE
     CLOSE DISK-FILE, PRINT-FILE
```

5.2 DATA MANIPULATION STATEMENTS

There are two data manipulation statements: MOVE and INSPECT.

5.2.1 THE MOVE STATEMENT

The MOVE statement transfers data from one area of computer storage to another. When the PICTURE clause of the receiving item contains editing characters, it also edits data by inserting, deleting, or replacing characters as specified.

The word MOVE is rather misleading since this process actually copies the data of the sending field and places this copy in one or more receiving field. Thus, the data in the sending field is not altered or destroyed by the MOVE statement. The format of this statement is:

```
MOVE  { identifier-1 }  TO identifier-2 [, identifier-3]...
      {  literal     }
                        [ON SIZE ERROR imperative-statement]
```

Both source and receiving items can be group or elementary. The literal may be numeric, non-numeric, or a figurative constant.

When the ON SIZE ERROR clause is included, its imperative statement is executed whenever significant characters are truncated during a MOVE. This feature facilitates checking input data for a valid range as shown in the following example.

Source Field	PICTURE of Receiving Field	Received Data
1 0 3 4 0 5	999V99	0 3 4 0 5

The manner in which the MOVE is performed depends on the PICTURE clauses of the source and receiving items. See Table 5-1 for a listing of legal and illegal MOVE's. There are three basic permissible moves: alphanumeric, numeric, and edited.

Note: A group item, even if it is composed of all numeric subfields, is treated as an alphanumeric item by the MOVE statement. This feature is quite useful for printing headers for numeric reports.

5.2.2 ALPHANUMERIC MOVES

In an alphanumeric move, the receiving field is an alphabetic, alphanumeric, or group item that stores the data left-justified unless otherwise specified. When the receiving field is longer than the sending field, the system fills the extra positions with spaces. When the receiving field is shorter than the sending field, the data characters are placed in the receiving field one at a time from left to right until the field is filled; the remaining characters are truncated.

When the figurative constant is ALL any-literal, the literal following the word ALL must be enclosed in quotation marks and may be only one character in length.

Source Field	PICTURE of Receiving Field	Received Data
A B C D	X(6) or A(6)	A B C D
A B C D	X(3) or A(3)	A B C
A B C D	X(4) or A(4)	A B C D
ALL 'X'	X(5)	X X X X X

5.2.3 NUMERIC MOVE

A numeric move occurs when a numeric field is moved to another numeric field, or when an alphanumeric field that contains only integers is moved to a numeric field. The assumed decimal points of both fields are aligned, and any extra positions at either end of the receiving item are filled with zeroes. When an item has no decimal point, one is assumed to be on the right end. When the receiving field is shorter than the source field, the source numeric data is stored in the receiving field beginning at the assumed decimal point and proceeding outward; when the field is full, unstored data is truncated.

Source Field	PICTURE of Receiving Field	Received Data
1 2 3	99V99	1 2 3 0
1 2 3 4	99V9	1 2 3
1 2 3 4	9(4)	1 2 3 4
1 2 3 4	9V99	2 3 4

NOTE: In the last example shown, a size error has occurred since the most significant digit, 1, was dropped during the move.

TABLE 5-1. Rules for Constructing Arithmetic-Expressions

First Symbol	Second Symbol					
	Variable	*or/	- or +	unary -	(	)or End of Expression
Variable	-	P	P	-	2	P
* or /	P	-	1	P	P	-
- or +	P	-	1	P	P	-
unary -	P	-	-	-	P	-
(or Begin-ning of Expres-	P	-	P	P	P	-
sion)	-	P	P	-	-	P

1 This is permitted when "-" indicates the sign of a numeric literal.

2 Parentheses immediately following a data-name indicate the presence of a subscript. The subscript is considered part of the variable.

P A specified pair of symbols is permitted.

- A specified pair of symbols is not permitted.

5.2.4 DE-EDIT MOVE

Whenever a MOVE occurs from an alphanumeric field to a numeric field, the following actions (called a de-edit) are performed. Leading and trailing blanks are deleted from the source data. Any leading positive or negative sign is removed and placed at the right of the destination field. Both fields are aligned by their assumed decimal points, and the data is moved.

5.2.5 EDITED MOVE

When editing is specified by the PICTURE clause of the receiving item, the source data is edited during the move after decimal point alignment. Data is moved to a receiving edited field in accordance with the rules for alphanumeric moves. Edited numeric data cannot be moved into a regular numeric field. For an explanation of editing symbols in a field, see the description of the PICTURE clause in Chapter 3.

5.2.6 THE INSPECT STATEMENT

The INSPECT statement replaces selected characters in a data item with other specified characters. The format of this statement is:

$$\text{INSPECT identifier-1 REPLACING} \left\{\begin{array}{l}\text{ALL}\\ \text{LEADING}\\ \text{FIRST}\end{array}\right\} \left\{\begin{array}{l}\text{identifer-2}\\ \text{literal-1}\end{array}\right\}$$
$$\text{BY} \left\{\begin{array}{l}\text{identifier-3}\\ \text{literal-2}\end{array}\right\}$$

Identifier-1 must be a group or elementary item whose USAGE IS DISPLAY. Remember that if a data item's description has no USAGE clause, the compiler assumes that USAGE IS DISPLAY. Identifiers -2 and -3 reference one-byte alphabetic, alphanumeric, or numeric elementary items whose usage is also display. Literals are non-numeric or any figurative constant except ALL.

The inspection of the item proceeds from left to right. The data item's contents are treated as follows:

1. An alphanumeric item is treated as a character string.

2. An unsigned numeric item is treated as if it had been been redefined as alphanumeric.

3. A signed numeric item is treated as if it had been moved to an unsigned numeric data item of the same length, and is subject to rule 2 shown above.

The rules for replacing data are as follows:

1. When literal-1 is a figurative constant, each character in identifier-1's field that is equal to it is replaced by the single character referenced by literal-2 or identifier-3.

2. When literal-2 is a figurative constant, each character in identifier-2's field that is equal to literal-1/identifier-2 is replaced by the literal-2 character.

The words ALL, LEADING, and FIRST qualify the replacement process as follows:

1. ALL replaces every literal-1/identifier-2 in identifier-1.

2. If LEADING is used, literal-1/identifier-2 must be equal to the field's leftmost character. Inspection/replacement then proceeds and stops only when a a character is encountered that is not the same as literal-1/identifier-2.

3. FIRST replaces the leftmost character in identifier-1 that is equal to literal-1/identifier-2. Inspection/replacement then stops.

An example of this process appears below.

```
Working-Storage:
     77 SS-NUMBER PIC 9(9) VALUE 123456789.
     77 EDITED-SS-NUMBER PIC 999/99/9999.

Procedure-Division:
         MOVE SS-NUMBER TO EDITED-SS-NUMBER.
         INSPECT EDITED-SS-NUMBER REPLACING ALL '/' BY '-'.
```

The new value of EDITED-SS-NUMBER is 123-45-6789.

5.3 ARITHMETIC STATEMENTS

Arithmetic operations in COBOL are specified by the key words ADD, SUBTRACT, MULTIPLY, DIVIDE, and COMPUTE. Only the COMPUTE statement uses arithmetic symbols (+, -, /, *, and =); the other statements use words to convey the arithmetic meanings.

The following general rules apply to all arithmetic statements:

1. All literals are numeric.

2. All identifiers are numeric elementary items whose maximum length is 15 decimal digits. These items may have different lengths, since decimal points are automatically aligned during the arithmetic process.

3. No item that is used in an arithmetic process may contain editing symbols. An item that simply stores arithmetic results, however, may contain editing symbols.

4. When the GIVING option is selected, the arithmetic result is stored in the identifier following GIVING.

5. The ROUNDED option is used when the arithmetic result has more digits to the right of its decimal point than does the receiving field. This overflow is called excess digits. When the ROUNDED option is omitted, excess digits are truncated. When this option is included and the most significant excess digit is greater than or equal to 5, then the least significant digit of the receiving field is increased by 1.

6. A SIZE ERROR occurs when the number of integer places in an arithmetic result exceeds the number of integer places in the receiving field. When this option is included and a size error occurs, the value of the resultant item is unchanged, and the specified imperative statement is executed. An arithmetic statement with a SIZE ERROR option is considered to be a conditional statement and is prohibited in contexts that allow only imperative statements. When the SIZE ERROR option is omitted and a size error occurs, program flow is not interrrupted, but the final result may or may not be correct.

5.3.1 GIVING OPTION

If the GIVING option is written, the value the identifier that follows the word GIVING is made equal to the calculated result of the arithmetic operation.

If the GIVING option is not written, each operand following the words TO, FROM, BY, and INTO in the ADD, SUBTRACT, MULTIPLY, AND DIVIDE statements, respectively, must be an identifier (not a literal). Each identifier is used in the computation, and also receives the result.

5.3.2 ROUNDED OPTION

If the ROUNDED option is not specified, truncation occurs when the number of places calculated (after decimal-point alignment) for the result is greater than the number of places in the data item that is to be set equal to the calculated result. When the ROUNDED option is specified, the least significant digit of the resultant data-name increases in value by 1 whenever the most significant digit of the excess is greater than or equal to 5.

Rounding of a computed negative result is performed by rounding the absolute value of the computed result and then making the final result negative.

Table 5-2 illustrates the relationship between a calculated result and the value stored in an item that is to receive the calculated result.

TABLE 5-3. Rounding or Truncation of Calculations

Calculated Result	Item to Receive Calculated Result		
	PICTURE	Value After Rounding	Value After Truncating
-12.36	S99V9	-12.4	-12.3
8.432	9V9	8.4	8.4
35.6	99V9	35.6	35.6
65.6	99V	66	65
0.0055	V999	0.006	0.005

5.3.3 SIZE ERROR OPTION

An arithmetic statement, if written with a SIZE ERROR option, is not an imperative-statement. Rather, it is a conditional statement and is prohibited in contexts where only imperative statements are allowed.

Whenever the number of integer places in the calculated result exceeds the number of integer places specified for the resultant item, a size error condition arises. If the SIZE ERROR option is specified and a size error condition arises, the value of the resultant item is not altered, and the series of imperative-statements specified for the condition is executed.

If the SIZE ERROR option is not specified and a size error condition arises, no assumption should be made about the correctness of the final result even though the program flow is not interrupted.

5.3.4 THE ADD STATEMENT

The ADD statement adds the values of two or more numeric items and/or literals and stores the result. The ADD statement must be written according to the general rules listed at the beginning of this section and must also conform to the guidelines described in the following paragraphs. The formats of this statement are:

Format 1:

```
ADD  { identifier-1 }    [ , identifier-2 ..., ]
     { literal-1    }    [ , literal-2         ]

     identifer-n  [ROUNDED]

     [; ON SIZE ERROR imperative-statement]
```

Format 2:

```
ADD  { identifier-1 }    ,[ identifier-2 ] ... TO
     { literal-1    }    ,[ literal-2    ]

identifier-m [ROUNDED]  [, identifier-n  [ROUNDED]]

[; ON SIZE ERROR imperative-statement]
```

Format 3:

```
        ADD  {identifier-1}  ,   {identifier-2}
             {literal-1   }      {literal-2   }

        , identifier-3 ... GIVING identifier-m [ROUNDED]
        , literal-3

        [;  ON SIZE ERROR imperative-statement]
```

Format 1 adds the values of all identifiers/literals and stores the result in identifier-n. A sample Format 1 statement is: ADD A, B, C. The values of A, B, and C before and after execution are:

	A	B	C
Before	5	6	8
After	5	6	19

Note that the values of A and B do not change as a result of the addition.

Format 2 adds the values of the identifiers/literals that precede the word TO. This intermediate result is then added to each data item specified by identifier-m, identifier-n, etc. to form a final result that is stored in identifier-m, identifier-n, etc. A sample Format 2 statement is: ADD W, X, Y, TO Z. The W, X, Y, and Z values before and after execution are:

	W	X	Y	Z
Before	2	7	8	12
After	2	7	8	29

Note that the values of all operands are added to form the sum stored in Z.

Format 3 adds the values of the operands preceding the word GIVING and places this immediate result in the identifiers that follow the word GIVING. A sample Format 3 statement is: ADD A, B, C GIVING D. The values of A, B, C, and D before and after execution are:

	A	B	C	D
Before	1	2	3	5
After	1	2	3	6

Note that the intermediate result replaces the value of D and is not added to D.

5.3.5 THE SUBTRACT STATEMENT

The SUBTRACT statement subtracts the value of one or more numeric items/literals from another numeric item/literal and stores the result. The formats of this statement are:

Format 1:

```
          SUBTRACT  { identifier-1 }   [ ,  identifier-2 ] ...
                    { literal-1    }   [ , literal-2     ]

                   FROM identifier-m [ROUNDED]

                   [; ON SIZE ERROR imperative-statement]
```

Format 2:

```
          SUBTRACT  { identifier-1 }   [ ,  identifier-2 ] ...
                    { literal-1    }   [ , literal-2     ]

             FROM  { identifier-m }  GIVING identifier-n
                   { literal-m    }

           [ROUNDED]  [; ON SIZE ERROR imperative-statement]
```

Format 1 subtracts the operands preceding the word FROM from identifier-m and places the result in identifier-m.

Format 2 subtracts the operands preceding the word FROM identifier-m/literal-m and places the result in the item following the word GIVING. The contents of identifier-m are not changed by a Format 2 SUBTRACT statement. A sample Format 2 statement is: SUBTRACT A FROM B GIVING C. The values of the operands before and after execution are:

	A	B	C
Before	10	80	90
After	10	80	70

5.3.6 THE MULTIPLY STATEMENT

The MULTIPLY statement multiplies two numeric items/literals together and stores the resultant product. The formats of this statement are:

Format 1:

```
          MULTIPLY  { identifier-1 }  BY  identifier-2 [ROUNDED]
                    {              }
                    { literal-1    }

                   [; ON SIZE ERROR imperative-statement]
```

Format 2:

```
        MULTIPLY  {identifier-1}  BY {identifier-2}
                  {literal-1   }     {literal-2   }

                  GIVING identifier-3 [ROUNDED]

                  [; ON SIZE ERROR imperative-statement]
```

Format 1 multiplies identifier-1/literal-1 by identifier-2 and stores the product in identifier-2. A sample Format 1 MULTIPLY statement is: MULTIPLY A BY B. The values of the operands before and after execution are:

	A	B
Before	10	20
After	10	200

Note that the values of operand B change to reflect the multiplication.

Format 2 multiplies identifier-1/literal-1 by identifier-2/literal-2 and stores the product in identifier-3. The contents of identifier-1 and identifier-2 are unaffected by the execution of a Format 2 MULTIPLY statement. A sample Format 2 statement is: MULTIPLY A BY B GIVING C. The values of the operands before and after execution are:

	A	B	C
Before	5	10	20
After	5	10	50

Note that the values of operands A and B remain the same, while the value of C changes.

5.3.7 THE DIVIDE STATEMENT

The DIVIDE statement divides the value of one numeric item/literal either into or by the value of another numeric item/literal and stores the result. The formats of this statement are:

Format 1:

```
        DIVIDE  {identifier-1}  INTO identifier-2 [ROUNDED]
                {literal-1   }

                [; ON SIZE ERROR imperative-statement]
```

Format 2:

```
        DIVIDE  {identifier-1}  INTO {identifier-2}
                {literal-1   }       {literal-2   }

                GIVING identifier-3 [ROUNDED]

                [; ON SIZE ERROR imperative-statement]
```

Format 3:

```
        DIVIDE  {identifier-1}  BY {identifier-2}
                {literal-1   }     {literal-2   }

                GIVING identifier-3 [ROUNDED]

                [; ON SIZE ERROR imperative-statement]
```

Format 1 divides identifier-1/literal-1 into identifier-2 and stores the resultant quotient in identifier-2. A sample Format 1 DIVIDE statement is: DIVIDE A INTO B. The values of the operands before and after execution are:

	A	B
Before	5	10
After	5	2

Note that the value of B has changed as a result of the division since the quotient is stored in it.

Format 2 divides identifier-1/literal-1 into identifier-2 and stores the resultant quotient in identifier-3. The contents of identifier-1 and identifier-2 are unchanged by a Format 2 DIVIDE statement. A sample Format 2 statement is: DIVIDE A INTO B GIVING C. The values of the operands before and after execution are:

	A	B	C
Before	5	10	15
After	5	10	2

Only the value of C changes as a result of a Format 2 DIVIDE.

Format 3 divides identifier-1/literal-1 by identifier-2 and stores the result in identifier-3. The values of identifier-1 and identifier-2 are unchanged by Format 3. A sample Format 3 statement is: DIVIDE A BY B GIVING C. The values of the operands before and after execution are:

	A	B	C
Before	10	5	30

```
     After    10     5     2
```

Only the value of C changes in a Format 3 DIVIDE.

5.3.8 THE COMPUTE STATEMENT

The COMPUTE statement combines the processing of ADD, SUBTRACT, MULTIPLY, and DIVIDE statements by using an arithmetic expression and storing the result in one numeric item. The format of this statement is:

```
                                  ( identifier-2          )
COMPUTE identifier-1 [ROUNDED] = <  literal-1              >
                                  ( arithmetic-expression )

           [; ON SIZE ERROR imperative-statement]
```

This statement is equivalent to a MOVE statement when the literal or identifier-2 is used.

An arithmetic expression is a series of numeric literals and/or data-names separated by arithmetic operators, and is reduced to a single numeric value and stored in a numeric item. The arithmetic operators are:

+	Addition
-	Subtraction
*	Multiplication
/	Division

A space is left on either side of an arithmetic operator. The only exception to this rule occurs when a minus sign indicates a logical negation (called a unary operator): a space does not have to follow it.

A few examples of arithmetic expressions are:

```
RATE  *  TIME
GROSS  -  DEDUCTIONS
OVERTIME  *  1.5  +  REGULAR-TIME
```

When parentheses are not used to specify the order of computation, the COBOL compiler evaluates an arithmetic expression by the following rules:

1. Logical negations (unary -) are performed first.
2. Next, multiplication and division are performed.
3. Finally, addition and subtraction are performed.

In each step, computation proceeds from left to right. Thus, A * B / C is computed as (A * B) / C, and A / B * C is computed as (A / B) * C.

When parentheses are used, computation begins with the innermost set and proceeds to the outermost. Items within parentheses are evaluated according to the rules above, and the results are then treated as if the parentheses were removed. For example:

```
COMPUTE RESULT-1 ROUNDED = (A + B) / (C + D + E)
COMPUTE TOTAL-RESULT = -((A - B) / 3 * (1.14 + C))
```

5.4 CONDITIONAL STATEMENTS

A conditional statement describes one or more conditions that are tested to determine the selection of alternate action paths. The basic COBOL conditional statement is the IF statement. The format of this statement is:

$$\text{IF condition [THEN]} \left\{\begin{matrix}\text{statement-1}\\ \text{NEXT SENTENCE}\end{matrix}\right\} \left[\text{ELSE} \left\{\begin{matrix}\text{statement-2}\\ \text{NEXT SENTENCE}\end{matrix}\right\}\right]$$

When the IF condition is found to be true, the actions specified in statement-1 are performed. The ELSE option specifies that statement-2 is performed when the condition is false. When the ELSE option is omitted, and the IF condition is false, program execution proceeds to the next sentence.

The condition may be compound or simple. Compound conditions are formed from a series of simple conditions that are separated by the logical operators AND, OR, and NOT. The test specified in a simple condition belongs to one of three categories: relational, class, and sign.

5.4.1 RELATIONAL TEST CONDITION

For numeric or non-numeric items, a relational condition compares two items and determines whether or not one is greater than, equal to, or less than another. The format of this condition is:

$$\left\{\begin{matrix}\text{identifier-1}\\ \\ \text{literal-1}\\ \\ \text{arithmetic-}\\ \text{expression-1}\end{matrix}\right\} \text{ IS [NOT] } \left\{\begin{matrix}\text{GREATER THAN}\\ >\\ \text{LESS THAN}\\ <\\ \text{EQUAL TO}\\ =\end{matrix}\right\} \left\{\begin{matrix}\text{identifier-2}\\ \\ \text{literal-2}\\ \\ \text{arithmetic-}\\ \text{expression-2}\end{matrix}\right\}$$

Numeric items are compared algebraically after decimal point alignment.

Non-numeric items are compared with respect to the binary collating sequence of ASCII characters. When such items have the same length, the comparison proceeds from high-order to the low-order until it encounters either a difference between the two items or the end.

When the non-numeric items have different lengths, the comparison proceeds as described in the previous paragraph. If the end of the shorter item is encountered in the comparison, the shorter item is considered to be less than the longer item unless only spaces remain in the longer item. For example:

```
AMOUNT-DUE IS GREATER THAN AMOUNT-OWED
ENTRY-COUNTER IS EQUAL TO 10
DIVISOR-1 NOT EQUAL TO ZERO
```

5.4.2 CLASS TEST CONDITION

In a class test, the data-name has a USAGE DISPLAY. If the USAGE clause had not been specified, the compiler assumes that USAGE IS DISPLAY. The format of this test is:

$$\text{data-name IS [NOT]} \left\{\begin{array}{l}\text{NUMERIC}\\ \text{ALPHABETIC}\end{array}\right\}$$

Examples:

```
     NAME-IN IS NOT ALPHABETIC
     NUMBER-ORDERED IS NOT NUMERIC
     ID-NUMBER NOT NUMERIC
```

5.4.3 SIGN TEST CONDITION

The sign test determines whether or not the value of a numeric item is positive, negative, or zero. The format of this test is:

$$\left\{\begin{array}{l}\text{data-name}\\ \text{arithmetic-}\\ \text{expression}\end{array}\right\} \text{ IS [NOT] } \left\{\begin{array}{l}\text{POSITIVE}\\ \text{NEGATIVE}\\ \text{ZERO}\end{array}\right\}$$

Examples:

```
     AMOUNT-DUE IS POSITIVE
     A - C / D IS NOT NEGATIVE
     NET-PAY ZERO
```

This sign test is the functional equivalent of a relational test that determines whether an item is greater than, less than, or equal to zero. For example, statements GROSS IS NEGATIVE and GROSS IS LESS THAN 0 are equivalent;

GROSS IS POSITIVE and GROSS IS GREATER THAN 0 are similarly equivalent.

5.4.4 LOGICAL OPERATORS IN COMPOUND CONDITIONS

The three logical operators are: AND, OR, and NOT. AND and OR are used to create a compound condition when two or more tests are specified in the same expression. NOT specifies the negation of a condition.

In the following AND example:

```
IF CODE IS ZERO OR AGE GREATER THAN 21 ADD A TO B.
```

OR means that if either or both conditions are fulfilled, the ADD C TO D statement is executed.

The logical operator NOT can be placed either within or immediately preceding a simple relational condition. Consider the following NOT examples:

AGE NOT GREATER THAN 21   and   NOT AGE GREATER THAN 21

These two statements mean exactly the same thing to the COBOL compiler. However, if NOT precedes a simple relational condition that also contains a NOT, a double negative results and causes an error. For example, the statement:

```
     NOT AGE NOT GREATER THAN 21
```

is an invalid condition since it contains a double negative.

5.4.5 EVALUATION OF THE CONDITION

The condition is evaluated before any action is taken. When the condition is true, statement-1 is executed; control is transferred to the following sentence; and the ELSE clause is ignored.

When the condition is false, either statement-2 or NEXT SENTENCE is executed. NEXT SENTENCE transfers control to the succeeding sentence.

Statement-1 or statement-2 may be a series of statements that is terminated by a period or an ELSE clause.

5.4.6 NESTED CONDITIONAL STATEMENTS

Statements-1 and -2 can be imperative statements that may also contain a conditional statement. When a conditional statement is contained in either statement-1, statement-2, or both, the conditional statement is said to be nested.

Nested conditional statements may in turn contain conditional statements. These nested conditional statements are analogous to the use of parentheses of combining subordinate arithmetic expressions so that the expressions become part of a larger arithmetic unit. For example:

```
IF A IS EQUAL TO ZERO IF B IS EQUAL TO ZERO
   PERFORM ZERO-VALUES ELSE NEXT SENTENCE
   ELSE PERFORM NON-ZERO-VALUES
```

Nested IF statements (IF's within IF's) must be considered as paired IF and ELSE combinations, proceeding from left to right. Therefore, any encountered ELSE applies to the immediately preceding IF that has not already been paired with an ELSE.

In essence, the number of occurrences of ELSE in any conditional statement must be equal to the number of occurrences of IF, regardless of the complexity caused by nesting.

The only exception to this rule occurs when ELSE NEXT SENTENCE directly precedes the period of the sentence. In this case, the entire phrase may be omitted and the period specified at the end of the previous phrase. This rule is extended to resulting sentences, etc. The statement in each ELSE clause is executed only when the conditional expression in the corresponding IF is false. If there are more IF's than ELSE's in a statement, it is assumed that the ELSE NEXT end of the sentence have been omitted. For example:

```
IF TOTAL GREATER THAN LIMIT IF ACTION-CODE EQUAL
    TO EXCEPTION-CODE PERFORM EXCEPTION-ROUTINE
    ELSE PERFORM EXCEEDS LIMIT.
```

In another example, the following sentence contains two independent nests of conditional statements. The first nest ends after the statement PERFORM procedure-name-2. The second nest consists of the remainder of the sentence and has an implied ELSE NEXT SENTENCE before the period. Each upper-case letter corresponds to a conditional expression.

```
IF A IF B PERFORM procedure-name-1 ELSE NEXT SENTENCE
ELSE IF C NEXT SENTENCE ELSE PERFORM procedure-name-2
IF D PERFORM procedure-name-3 IF E PERFORM
procedure-name-4 IF F PERFORM procedure-name-5 ELSE
PERFORM procedure-name-6 ELSE STOP RUN.
```

5.4.7 CONDITIONAL STATEMENTS WITH EXCEPTION BRANCHES

The READ, WRITE, REWRITE, DELETE, ADD, SUBTRACT, MULTIPLY, and DIVIDE verbs specify a required or optional exception branch as part of the statement. When the exception branch is present, it makes a conditional statement out of an otherwise imperative one. The format of an exception branch is:

$$\left\{\begin{array}{l}\text{AT END}\\ \text{INVALID KEY}\\ \text{ON SIZE ERROR}\end{array}\right\}\ \text{imperative-statements}$$

5.5 SEQUENCE CONTROL STATEMENTS

The starting point for program execution is at the first statement of the Procedure Division. When a program contains no sequence control statements, control proceeds to successive statements until the end of the paragraph or section is reached. Control then passes to the first statement in the next paragraph or section and proceeds in this manner to the end of the program.

A sequence control statement, however, alters the normal sequence of control. These statements are:

GO TO	-	Permanently releases control to the first statement of the paragraph or section to which it refers.
CALL	-	Transfers control from one runtime COBOL program to another.
UCALL	-	Transfers control to a loadable program written in a language other than COBOL. Loadable program can return to the next statement by using a RTS assembly instruction.
PERFORM	-	Executes the statements of specified paragraphs or sections, then returns control to the statement that follows the PERFORM statement.
IF	-	Causes control to branch into one of two paths, depending on the outcome of a conditional test. These paths rejoin at the beginning of the next sentence unless a GO TO branch is used in one or both paths. The IF statement is discussed in the Conditional Branch section.
EXIT	-	Declares the end of a paragraph.

STOP RUN - Terminates the program in an orderly manner.

STOP "LIT" - Can be used to display a character string on the display screen.

5.5.1 THE GO TO STATEMENT

The GO TO statement conditionally or unconditionally transfers control permanently to another point in a program. The formats of this statement are:

Format 1:

```
        GO TO procedure-name-1
```

Format 2:

```
        GO TO procedure-name-1 [, procedure-name-2]...,

              procedure-name-n DEPENDING ON identifier-1
```

Format 1 is the unconditional GO TO statement that permanently transfers control to another paragraph or section that is specified by procedure-name-1. Examples of unconditional GO TO statements are shown below.

```
GO TO TEST- ROUTINE

IF A EQUALS B GO TO SINCE-ROUTINE ELSE ADD A TO B
GO TO START-ROUTINE
```

Format 2 is the conditional GO TO which is a multiple branch point that references a maximum of 16 paragraphs or sections. Since the execution of the branch depends on the value of the identifier, this item is tested when the statement is executed to determine which branch to take. An example of a conditional GO TO statement is:

```
GO TO FEDERAL-TAX, STATE-TAX, LOCAL-TAX DEPENDING
ON GROSS-SALARY-CODE.
```

When the conditional GO TO statement is executed, control is transferred to procedure-names -1, -2, or -n, depending on whether the identifier's value is 1, 2, or n. The identifier must be an elementary item of numeric integers that can be subscripted if necessary. If identifier's value is not within the specified range, no transfer occurs, and control passes to the next statement.

5.5.2 THE CALL STATEMENT

The CALL statement transfers control from one runtime COBOL program to another. The Linkage Section can be used with the CALL since this section is a common area for data that the called program can also access. The format of this statement is:

$$\text{CALL} \left\{\begin{array}{l}\text{file-name}\\ \text{FKEYnn}\\ \text{SKEYnn}\end{array}\right\} \quad \text{[USING DATA-NAME1...DATA-NAME20]}$$

File-name represents a COBOL compiler-generated loadable file. FKEYnn and SKEYnn are loadable program files whose name assigns them to a function or shift-function key, respectively: nn is 01 to 16 for function keys, or 03 to 16 for shift function keys.

The program in which the CALL statement appears is the calling program; the program whose name is specified in the CALL statement is the called program. The execution of a CALL statement transfers control to the called program. Control is only returned to the calling program by a subsequent call, and execution begins at the top of the recalled program.

5.5.3 THE UCALL STATEMENT

The UCALL statement allows the user to include at runtime subroutines written in a language other than COBOL. The format of this statement is:

```
UCALL filename [(n) USING data-name-1, data-name-2,...
                          data-name-20].
```

File-name represents a loadable program-file with a .LO suffix. Optionally, n represents the entry point parameter (0-9) to be passed in the A-register. Data-name-1 thru data-name-20 represents the data items to be referenced by the called program. A pointer to this list of addresses will be passed in the X-register.

To create a loadable program file, first create a relocatable object program by assembling the user program, using the REL (relocatable program) option discussed in the Assembly Language Manual, as shown in the following example.

```
          TTL USERPROG
          OPT REL
START     LDX #$6600
            .
            .
          RTS
          END START

CMAP    USERPROG;0=USERPROG.RO
```

Next, compile the COBOL program containing the UCALL statement. At the end of the COBOL compilation, the compiler prints/displays all necessary load information as shown in the following example.

```
       PROCEDURE DIVISION
               .
               .
               .
               UCALL USERPROG USING DATA1,DATA2.
               .
               .
               .
               STOP RUN.

SIZE:  PROG 0100           DATA 0050
LOAD UCALL OBJ. ABOVE HEX. LOC. 6850
MAX. UCALL OBJ. IS HEX. 77B0
```

Finally, an RLOAD activity can be used to put the user program in a loadable format. The user program can be loaded anywhere above location $6850 in the example. The following RLOAD activity shows how this can be accomplished.

```
     = RLOAD
     ? IF=XX.IF
     ? BASE=$6851
     ? LOAD=USERPROG.RO
     ? OBJA=USERPROG.LO
     ? MO=#LP
     ? MAPF
     ? EXIT
```

After the RLOAD activity has been successfully completed, the COBOL program can be run and the subsequent UCALL to the user program will be performed.

During runtime, the execution of a UCALL statement transfers control to the called user program. An RTS instruction returns control to the statement following the UCALL statement. The called user program may not exceed available memory locations.

5.5.4 THE PERFORM STATEMENT

The PERFORM statement executes all statements in one or more specified paragraphs or sections for a specified number of times or until a condition is satisfied. When a PERFORM statement has finished execution, program control returns to the statement that follows the PERFORM statement. The formats of this statement are:

Format 1:

```
     PERFORM procedure-name-1 [THRU procedure-name-2]
```

Format 2:

```
     PERFORM procedure-name-1 [THRU procedure-name-2]

         {identifier}   TIMES
         {integer   }
```

Format 3:

```
     PERFORM procedure-name-1 [THRU procedure-name-2]

         UNTIL condition
```

Format 4:

```
     PERFORM procedure-name-1 [THRU procedure-name-2]

         VARYING {index-name-1}  FROM {index-name-2}
                 {identifier-1}       {identifier-2}
                                      {literal-1   }

         BY {identifier-3}  UNTIL condition-1
            {literal-2   }
         [
         [AFTER {index-name-4}  FROM {index-name-5}
         [      {literal-3   }       {identifier-5}
         [                           {literal-4   }

         BY {identifier-6}  UNTIL condition-2
            {literal-5   }

         [AFTER {index-name-7}  FROM {index-name-8}
         [      {identifier-7}       {identifier-8}
                                     {literal-6   }

         BY {identifier-9}  UNTIL condition-3]]
            {literal-7   }
```

Normal program sequence resumes after the last statement of the last specified paragraph or section has been executed. The last statement performed, however, must not be an unconditional GO TO statement.

When procedure-name-2 is included in any PERFORM format, it must follow procedure-name-1 in the normal logical sequence of the program. GO TO statements and other PERFORM statements are permitted between procedure-name-1 and the last statement of procedure-name-2 provided that the sequence ultimately returns to the final statement of procedure-name-2. For example:

```
PERFORM BONUS-CALCULATION
PERFORM BONUS-CALCULATION THRU PRINT-LINE
```

If a conditional branch is required before the final sentence, procedure-name-2 must be the name of a paragraph consisting solely of the EXIT statement; all paths must eventually lead to this point. For example:

```
PERFORM ERROR-IN-DATA THRU ERROR-IN-DATA-EXIT.
PERFORM DATA-UPDATE THRU DATA-UPDATE-EXIT.
```

When a sequence of statements executed by a PERFORM statement includes another PERFORM statement, the sequence of procedures associated with them including the PERFORM must be either totally included in, or totally excluded from, the logical sequence referred to by the first PERFORM. Thus, an active PERFORM statement whose execution point begins within the range of another PERFORM must not contain within its range the exit point of the other active PERFORM statement.

The TIMES option in Format 2 executes the procedures a specified number of times. A counter is set up with a value equal to the value of the identifier or integer. Before each execution of the specified procedure, this counter is tested for a negative or zero value. If the counter's value is positive, the procedure is executed, and the value of the counter is decreased by one. When the value is negative or zero, the procedure has been executed the correct number of times, and control transfers to the statement following the PERFORM statement. For example:

```
PERFORM COUNTER-ROUTINE 100 TIMES
```

The UNTIL option in Format 3 states that the number of times the procedures are executed depends on the fulfillment of a simple or compound condition that is evaluated before the procedures are executed. When the condition is found to be false, the procedure is executed, and the expression is evaluated again. This process is repeated until the expression is found to be true; control then transfers to the statement following the PERFORM statement. If the conditional expression is found to be true when the PERFORM statement is first encountered, the specified procedure is not executed. For example:

```
PERFORM COUNTER-ROUTINE UNTIL ITEM-COUNTER
    GREATER THAN 101
```

The VARYING option in Format 4 executes the PERFORM repeatedly while increasing or descreasing the value of one to three identifiers or index-names once for each execution until a conditional is satisfied. For example:

```
PERFORM TABLE-SEARCH VARYING SUB-1 FROM 1 BY 1
    UNTIL TRANS-CODE = ITEM-CODE (SUB-1)
```

The flow charts in Figure 5-1 illustrate the logic of the PERFORM statement when one, two, or three identifiers are varied.

For example, assume that a billing procedure uses a three-dimensional rate table. This table has rates for five regions, each of which includes ten states, each of which contains rates for twelve cities. The basic updating procedure for this table could be:

```
RATE-UPDATING.
    MULTIPLY RATE (REGION, STATE, CITY) BY
    ADJUST-FACTOR GIVING RATE (REGION, STATE, CITY).
```

To execute this procedure once for each city of each state in each region, the following PERFORM statement could be used:

```
PERFORM RATE-UPDATING VARYING REGION FROM 1 BY 1
    UNTIL REGION IS GREATER THAN 5 AFTER STATE FROM 1
    BY 1 UNTIL STATE EQUALS 11 AFTER CITY FROM 1 BY 1
    UNTIL CITY IS GREATER THAN 12.
```

When this PERFORM is executed at object time, the RATE-UPDATING procedure is executed for the first city of the first state in the first region, then for the next city, etc. The PERFORM is complete when the procedure is executed for the twelfth city of the tenth state of the fifth region, by which time the procedure has been executed 600 times.

5.5.5 THE EXIT STATEMENT

The EXIT statement ends a procedure that is executed by a PERFORM statement. The format of this statement is:

```
paragraph-name.  EXIT
```

EXIT must be the paragraph's only statement, and is equivalent to a paragraph with no sentences. This statement generates no code. For example:

```
TABLE-LOOK-UP-EXIT.   EXIT.
NON-NUMERIC-DATA-EXIT.   EXIT.
```

5.5.6 THE STOP STATEMENT

The format of the STOP statement is:

$$\text{STOP} \begin{Bmatrix} \text{literal} \\ \text{RUN} \end{Bmatrix}$$

STOP RUN terminates the program; STOP "literal" displays the literal and then continues with program execution.

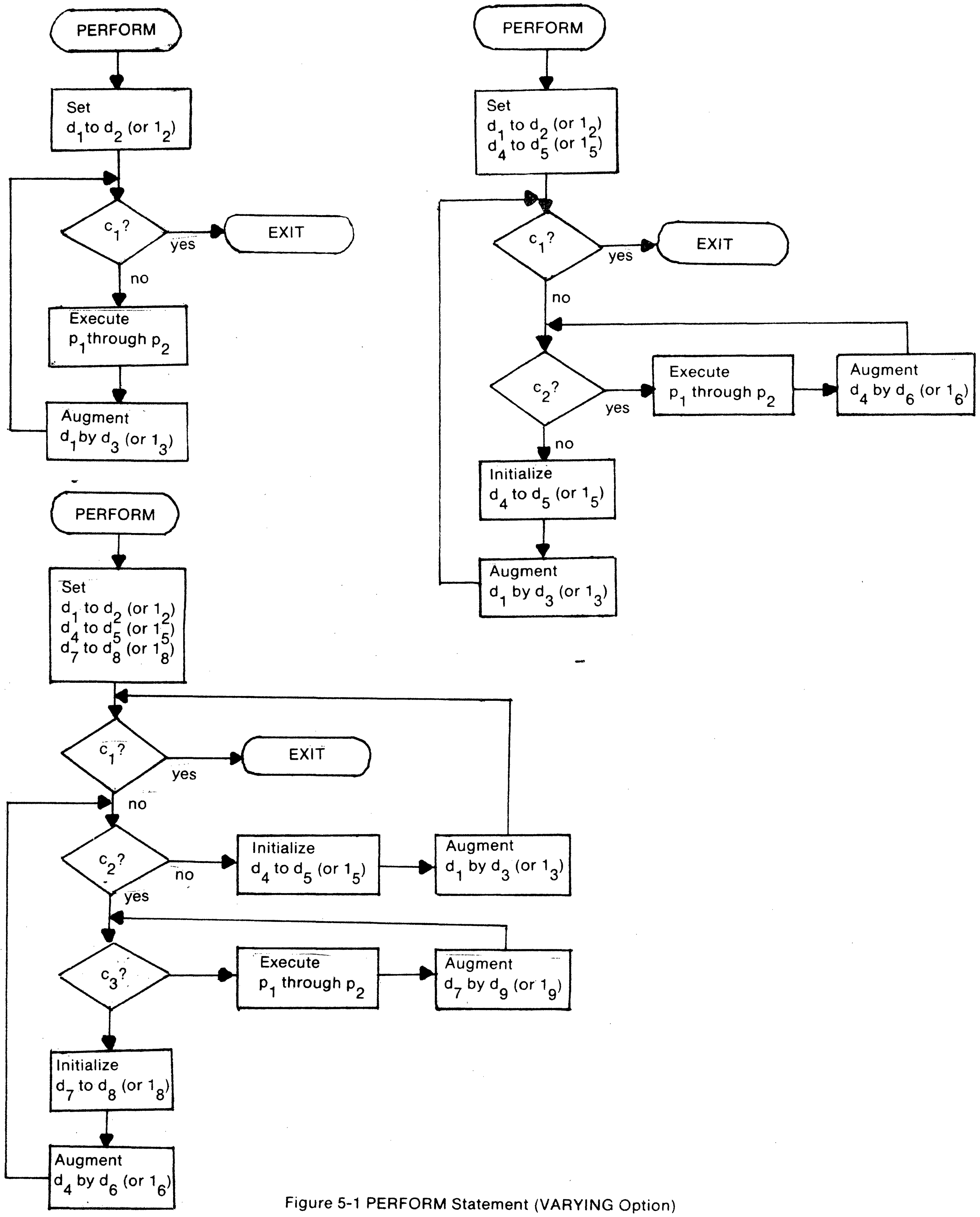

PERFORM
Set d_1 to d_2 (or 1_2)
c_1?
yes
EXIT
no
Execute p_1 through p_2
Augment d_1 by d_3 (or 1_3)
PERFORM
Set d_1 to d_2 (or 1_2) d_4 to d_5 (or 1_5)
c_1?
yes
EXIT
no
c_2?
yes
Execute p_1 through p_2
Augment d_4 by d_6 (or 1_6)
no
Initialize d_4 to d_5 (or 1_5)
Augment d_1 by d_3 (or 1_3)
PERFORM
Set d_1 to d_2 (or 1_2) d_4 to d_5 (or 1_5) d_7 to d_8 (or 1_8)
c_1?
yes
EXIT
no
c_2?
no
Initialize d_4 to d_5 (or 1_5)
Augment d_1 by d_3 (or 1_3)
yes
c_3?
Execute p_1 through p_2
Augment d_7 by d_9 (or 1_9)
Initialize d_7 to d_8 (or 1_8)
Augment d_4 by d_6 (or 1_6)

Figure 5-1 PERFORM Statement (VARYING Option)

5.6 TABLE-HANDLING STATEMENTS

A table is defined in the Working-Storage Section of the Data Division through the use of the OCCURS and REDEFINES clauses. The values of a table may be loaded through the VALUE clause, or they may be loaded through a program subroutine.

The entries of a table may be accessed through either subscripting or indexing. Indexing is a technique similar to subscripting, but has the advantage that it contains a direct pointer to an individual element in a table rather than an occurrence number. There are two statements used in indexing a table: SET and SEARCH.

5.6.1 SUBSCRIPTING A TABLE

A subscript must be assigned a dataname that represents a numeric elementary item in the Working-Storage Section. When used in a statement to refer to an entry in the table, the format is:

```
     data-name (subscript)
```

where data-name is the name of the redefined table, and the subscript has a numeric value that represents a specific table entry. A subscript represents an occurrence value.

For example, a sample table is defined by the Data Division statements:

```
01   PARTS-TABLE VALUES.
     05  FILLER   PIC 9(5) VALUE 53925.
     05  FILLER   PIC 9(5) VALUE 29383.
     05  FILLER   PIC 9(5) VALUE 49582.
     05  FILLER   PIC 9(5) VALUE 39049.
01   PARTS-TABLE REDEFINES PARTS-TABLE-VALUES.
     05   PART-NUMBER  PIC 9(5) OCCURS 4 TIMES.
```

The subscript that will be used to locate table entries is defined as:

```
     77 SUB-1 PIC 9 VALUE ZERO.
```

To find a particular table entry for a sample transaction code, the following statements could be used.

```
MOVE TRANS-CODE TO SUB-1 ON SIZE ERROR PERFORM
    ERRONEOUS-CODE.
MOVE PART-NUMBER (SUB-1) TO PART-NUMBER-OUT.
```

For tables whose codes do not run consecutively from 1, 2, 3, etc., the following statements would locate entries.

```
    PERFORM TABLE-SEARCH VARYING SUB-1 FROM 1 BY 1
        UNTIL TRANS-CODE IS EQUAL TO TABLE-ENTRY-CODE
        OR SUB-1 IS GREATER THAN TABLE-LENGTH.
                .
                .
                .
TABLE-SEARCH.
    IF TRANS-CODE EQUAL TO TABLE-ENTRY-CODE MOVE
        TABLE-ENTRY-VALUE (SUB-1) TO VALUE-OUT.
```

5.6.2 INDEXING A TABLE

The difference between subscripting a table and indexing a table is that a subscript represents an occurrence number while an index stands for a displacement value from the start of the table. Generally, indexed produce a more efficient object code than do subscripts, since they don't have to be converted into displacement values to refer to table entries. The general format that is used to refer to an indexed table entry is:

```
data-name (index-name)
```

where the index-name must conform to the rules for it that are mentioned in the SET and SEARCH statements.

5.6.3 THE SET STATEMENT

The SET statement establishes table-handling reference points by setting index-names associated with table elements. An index-name must always be SET to an appropriate value before the first execution of a SEARCH statement. The formats of this statement are:

Format 1:

$$\text{SET} \begin{Bmatrix}\text{index-name-1}\\ \text{data-name-1}\end{Bmatrix} \text{ TO } \begin{Bmatrix}\text{index-name-2}\\ \text{data-name-2}\\ \text{literal}\end{Bmatrix}$$

Format 2:

$$\text{SET index-name} \begin{Bmatrix}\text{UP BY}\\ \text{DOWN BY}\end{Bmatrix} \begin{Bmatrix}\text{data-name}\\ \text{literal}\end{Bmatrix}$$

An index name is defined in a table's INDEXED BY clause. Format 1 data-names are either data items or numeric elementary items that contain only integers. The data-name Format 2 must not be an index data item. Literals must be positive integers.

In Format 1, the following actions occur:

1. When index-name-1 is chosen, it is set to a value that corresponds to the same occurrence number to which either index-name-2, data-name-2, or literal corresponds. If data-name-2 is an index data item, or if index-name-2 is related to the same table as index-name-1, no conversion takes place.

2. When data-name-1 is chosen, and it is an index data item, it may be set equal to either the contents of index-name-2 or data-name-2 where the latter is also an index data item; literal cannot be used.

3. If data-name-1 is chosen, and it is not an index data item, it may be set only to an occurrence number that corresponds to the value of index-name-2; neither data-name-2 nor literal can be used.

In Format 2, the value of index-name is incremented (UP BY) or decremented (DOWN BY) by a value that corresponds to the number of occurrences specified by the value of literal or identifier. For example:

```
SET ITEM-INDEX TO 1
SET RATE-INDEX UP BY 1
SET TABLE-INDEX DOWN BY 1
```

5.6.4 THE SEARCH STATEMENT

The SEARCH statement searches a table for a table element that satisfies the specified condition and adjusts the associated index-name to indicate that table element. The format of this statement is:

```
SEARCH identifier-1 [VARYING {index-name-1}]
                             {identifier-2}

        [; AT END imperative-statement-1]

         ; WHEN condition-1   {imperative-statement-2}
                              {NEXT SENTENCE         }

       [[; WHEN condition-2   {imperative-statement-3}]
                              {NEXT SENTENCE         }]
```

Data-name-1 may not be subscripted or indexed, and its data description must contain OCCURS and INDEXED BY clauses.

Data-name-2, when specified, must have a USAGE IS INDEX clause in its data description, or must represent a numeric elementary item that contains only integers. It is incremented by the same amount and at the same time as the occurrence number represented by the index-name of data-name-1.

When the SEARCH statement is executed, a serial search occurs that starts with the current index setting and follows one of the procedures described below.

When the index-name contains a value that is less than the highest permissible occurrence number for data-name-1, the specified conditions are evaluated sequentially as written, making use of any specified index settings to determine the occurrence of items to be tested. When no condition is satisfied, the index-name is incremented to reference the next occurrence. This process is repeated.

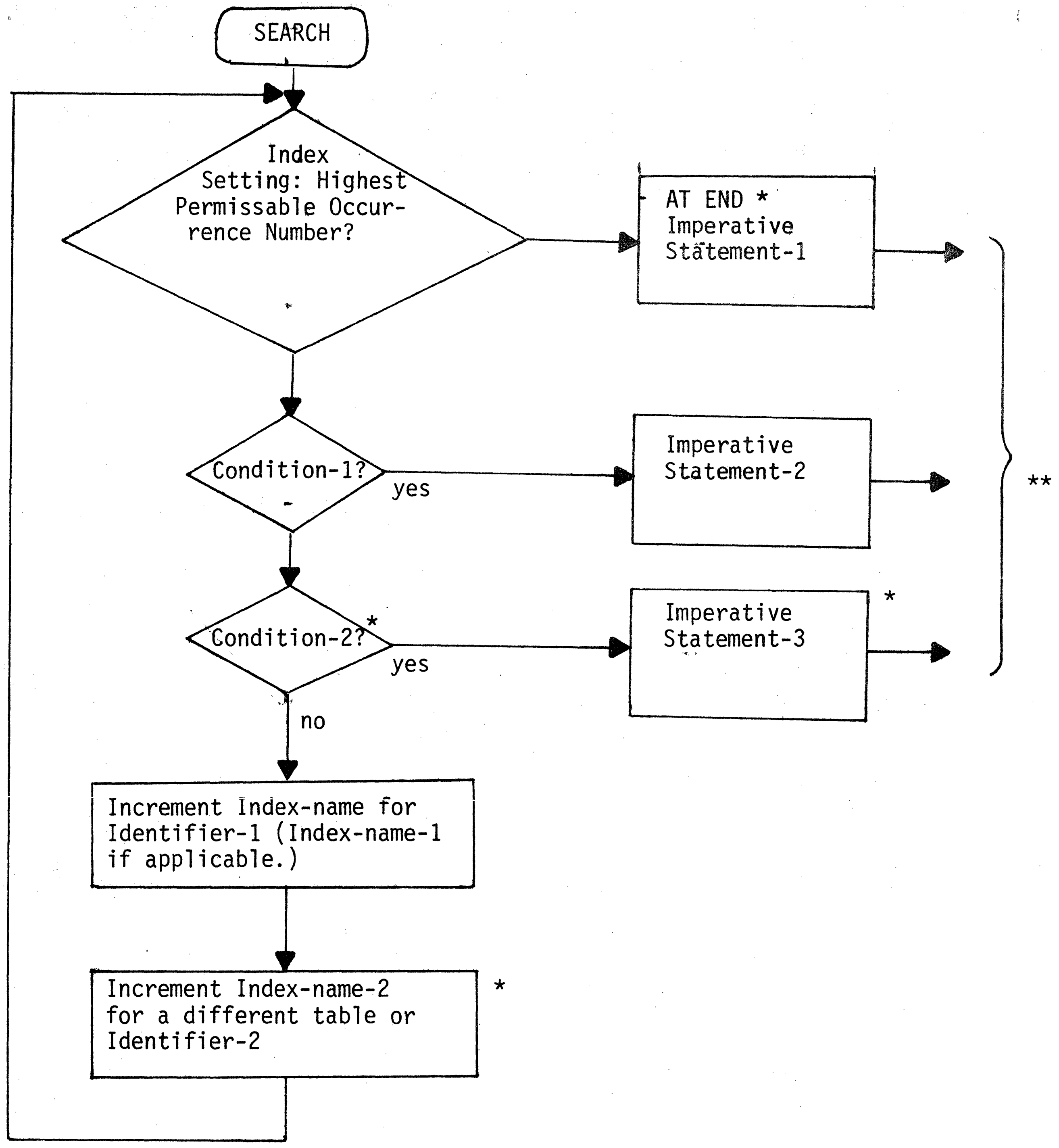
When the index-name contains a value that is greater than the highest permissible occurrence number for data-name-1, the SEARCH is immediately terminated. When the AT END clause is included, imperative-statement-1 is executed; otherwise, control passes to the next sentence.

When one of the conditions is satisfied, the search immediately terminates, and the specified imperative statement is executed. The index-name remains set at the occurrence that caused the condition to be satisfied.

When a specified imperative statement is executed that does not include a GO TO statement, program control passes to the next sentence.

In the VARYING option, the index-name is only used if it appears in the table's INDEXED BY clause; otherwise, the first index-name in the INDEXED BY clause is used. If the index-name appears in the INDEXED BY clause of another table, the occurrence number represented by the index-name is incremented by the same amount and at the same time as the occurrence number represented by the index-name associated with data-name-1.

If data-name-1 is in a hierarchy of groups, each group's description must have an OCCURS clause and an index-name. The settings of these index-names are used throughout the execution of the SEARCH statement to refer to data-name-1 or items therein. These index settings are not modified by the execution of the SEARCH statement (unless stated as the index-name); only the index-name or data-name-2 associated with data-name-1 is incremented by the SEARCH. A diagram of a SEARCH operation containing two WHEN phrases is shown in Figure 5-2.


SEARCH
Index Setting: Highest Permissable Occurrence Number?
AT END *
Imperative Statement-1
Condition-1?
yes
Imperative Statement-2
Condition-2? *
yes
Imperative Statement-3
*
no
**
Increment Index-name for Identifier-1 (Index-name-1 if applicable.)
Increment Index-name-2 for a different table or Identifier-2
*

* These operations are only included if called for in the statement.

** Each of these operations transfers control to the next sentence unless the statement ends with a GO TO statement.

Figure 5-2

SEARCH Statement Operation

5.7 COPY STATEMENTS

The COPY statement incorporates source text obtained from files external to the COBOL source program.

The COPY may be written in any of the following places in a COBOL program.

1. In the Environment Division
   SOURCE-COMPUTER. Copy-statement.
   OBJECT-COMPUTER. Copy-statement.
   FILE-CONTROL. Copy-statement.
   I-O CONTROL. Copy-statement.

2. In the Data Division
   FILE SECTION.
   FD file-name copy-statement.
   01 data-name copy-statement.
   WORKING-STORAGE SECTION.
   01 data-name copy-statement.

3. In the Procedure Division

$$\left\{\begin{array}{l}\text{paragraph-name}\\ \text{section-name SECTION}\end{array}\right\}\quad \text{copy statement.}$$

Source text, composed of either one paragraph or one section, is is stored as a file on some storage medium, such as disk or diskette. The specified file is copied during compilation, and the result is the same as if the routine were actually a part of the source program. The COPY statement will copy source text into any part of a program after the Environment Division; the text can be anywhere from one line to an entire paragraph. The format of this COPY statement is:

```
COPY file-name.
```

When the source text is composed of one paragraph, it is copied into the source program in place of the COPY statement, with the procedure-name of the routine.

When the source text is composed of one section, it is copied into the source program in place of the COPY section, with the section-name of the COPY section automatically replacing the section-name of the section being copied.

Examples of the COPY statement are shown below.

```
FD MASTER-FILE COPY FILEA.
```

FILEA is the file-name of the file that contains a complete File Description entry to be copied into the source program as the description of the file named MASTER-FILE.

```
01  SUM-DATA COPY SUMMARY-A.
```

SUMMARY-A is the name of a file whose contents is a record description entry of the form:

```
     02  COUNT        PIC 9(3).
     02  G-TOTAL      PIC 9(5)V99.
     02  0-TOTAL      PIC 9(6)V99.
     02  G-DEVIATION  PIC 9(4)V99.
     02  O-DEVIATION  PIC 9(4)V99.
```

The data description copied into the source program in place of the line bearing the COPY clause is:

```
     01  SUM-DATA.
     02  COUNT          PIC 9(3).
     02  G-TOTAL        PIC 9(5)V99.
     02  O-TOTAL        PIC 9(6)V99.
     02  G-DEVIATION    PIC 9(4)V99.
     02  O-DEVIATION    PIC 9(4)V99.
```

CHAPTER 6 - GUIDE TO WRITING COBOL CRT TRANSACTIONS

This chapter describes the COBOL CRT screen I/O support. The FORMS Design Facility (FORMS), however, provides a more simplified and more powerful logical screen access method.

FORMS allows the programmer to interactively create and modify screen formats for data entry and validation applications. FORMS allows the operator to view on the screen both prompting menus and the format being created.

Facilities are available to generate COBOL data definition structures from the format. These structures are suitable for copying into a COBOL applications program that wants to reference that format.

The FORMS Access Facility (CFORMS) allows the COBOL user to handle screen input and output operations at field or entire format levels. CFORMS maintains storage areas for the actual formats, decreasing the COBOL program storage requirements for format storage and field edit coding.

6.1 WRITING A CRT FORMAT

A screen format is set up in the Working-Storage Section as described in Chapter 4. A sample screen description appears below.

```
01 SCREEN-1 LINE IS NEXT PAGE.
   02 FILLER PIC X(5) LINE 3 COLUMN 3 VALUE 'NAME'.
   02 FILLER PIC X(15) COLUMN 9 VALUE SPACE.
   02 FILLER PIC X(7) LINE 5, COLUMN 3; VALUE 'ADDRESS'
   02 FILLER PIC X(20) VALUE SPACE.
```

LINE IS NEXT PAGE clears the screen and protects all positions.

LINE 3 COLUMN 3 VALUE 'NAME' displays the word NAME on line 3 at column 3. No data should ever be placed in column 1, which contains screen control codes.

COLUMN 9 moves the cursor's position to column 9, but stays on line 3 since a new line had not been stated. At least one column should separate screen items for the storage of internally-generated screen control codes. When the VALUE clause contains a figurative constant, such as SPACE or ZERO, a field is created that is the size of the PICTURE. This field is unprotected, underscored, and accepts data when displayed on the screen.

LINE 5 COLUMN 3 displays the word ADDRESS where specified.

The X(20) VALUE SPACE entry creates an unprotected, underlined field that is 20 characters long. Since there is no column clause in this entry, the compiler generates one space automatically between the word ADDRESS and the beginning of this field.

6.2 SENDING DATA TO THE SCREEN

The Procedure Division statement, DISPLAY SCREEN-1, results in the following display.

```
NAME ________________________________

ADDRESS _____________________________
```

6.3 RECEIVING DATA FROM THE SCREEN

A screen-response record is written in the Working-Storage Section to accommodate the data entered in unprotected fields on the screen. A response record for the SCREEN-1 display would be:

```
01 RESPONSE.
   02 RNAME PIC X(15).
   02 RADDRESS PIC X(20).
```

When the SEND PAGE key is pressed at transaction time, the Procedure Division statement ACCEPT RESPONSE moves data from the NAME and ADDRESS fields to RNAME and RADDRESS.

6.4 WRITING ERROR INDICATIONS

The following technique causes a field to blink and reverse video when erroneous data has been detected during editing tests described later in this chapter.

```
(Working-Storage)
     77  X PIC 9.
     01  ERR-ROW.
         02  FILLER PIC 99 VALUE 3.
         02  FILLER PIC 99 VALUE 5.
     01  ROW REDEFINES ERR-ROW PIC 99 OCCURS 2 TIMES.
     01  ERR-COL.
         02  FILLER PIC 99 VALUE 15.
         02  FILLER PIC 99 VALUE 10.
     01  COL REDEFINES ERR-COL PIC 99 OCCURS 2 TIMES.

(Procedure Division)
         MOVE 2 TO X.
         PERFORM ERR-BLANK THRU ERR-BLINK-EDIT.
             .
             .
             .
     ERR-BLINK.
         DISPLAY @(ROW(X), COL(X)) $E0E2.
     ERR-BLINK-EDIT.  EXIT.
```

When the program is checking the data entered in RADDRESS and an error is discovered, the Procedure Division instructions shown above would be performed. The Working-Storage tables would be used for determining the field location.

The statement MOVE 2 TO X means that the error is in the second data field. The @(ROW(X),COL(X)) phrase establishes the line and column location on the screen, using subscripts and the tables.

A single dollar sign ($) precedes any of the screen control codes. E0 sets the blink condition, and E2 sets the reverse video condition. A complete list of screen control codes is provided in Appendix B.

The following Procedure Division statements clear the blink and reverse video condition.

```
    MOVE 2 TO X.
    PERFORM ERR-OFF THRU ERR-OFF-EXIT.
        .
        .
        .
ERR-OFF.
DISPLAY @(ROW(X),COL(X)) $E1E3.
ERR-OFF-EXIT.  EXIT.
```

E1 clears the blink condition, and E3 turns off the reverse video.

The following is an example of a situation in which some data are always the same, such as titles, and other data are variable upon input from another source.

```
WORKING-STORAGE SECTION.
01  DATA-RECORD.
    02  PART-NUMBER  PIC X(4).
    02  DESC         PIC X(16).
    02  LOCATION     PIC 999.
    02  QTY-ON-HAND  PIC 9(4).
    02  COST         PIC 9999V99.
01  SCREEN-2 LINE IS NEXT PAGE.
    02  FILLER       PIC X(11)   LINE 4 COLUMN 2;
                                 VALUE 'DESCRIPTION'.
    02  DESC         PIC X(16)   VALUE ' '.
    02  FILLER       PIC X(8)    LINE 6 COLUMN 2;
                                 VALUE 'LOCATION'.
    02  SLOC         PIC ZZ9     COLUMN 26; VALUE ' '.
    02  FILLER       PIC X(8)    LINE 8, COLUMN 2;
                                 VALUE 'QUANTITY'.
    02  SQTY         PIC ZZ9     VALUE ' '.
    02  FILLER       PIC X(4)    COLUMN 50;
                                 VALUE 'COST'.
    02  SCOST        PIC ZZZ.99  COLUMN 46 VALUE 'X'.
```

The VALUE ' ' for DESC, SLOC, SQTY, and SCOST makes them protected fields, so that their data cannot be changed on the screen.

The following instructions can be used with DATA-RECORD and SCREEN-2 to read disk file data and move it to the screen.

```
    OPEN DATAFILE.
    READ DATAFILE INTO DATA-RECORD.
    MOVE DESC TO SDESC.
    MOVE LOCATION TO SLOC.
    MOVE QTY-ON-HAND TO SQTY.
    MOVE COST TO SCOST.
    DISPLAY SCREEN-2.
```

When these statements are executed, data is read from DATAFILE into DATA-RECORD. Each data item in DATA-RECORD is moved separately to its corresponding data item in SCREEN-2, which is then displayed on the CRT. All fields in SCREEN-2 are protected, which means that none of them can be changed through the keyboard.

6.5 SCREEN MESSAGES

The following statement sends a message to the screen.

```
DISPLAY @(24,3) $E0EAC4 'INVALID ENTRY' $E1.
```

This message begins on line 24, column 3. Screen control code E0 sets the blink code, while EA protects the field.

NOTE: When a screen control code is used on an unprotected field, EA should follow the code to protect the field, unless it is meant to be unprotected.

The C4 code moves the cursor to the next position. If this code had been omitted, the I of INVALID ENTRY would overlay the E0 and EA codes. The final E1 code clears the blinking condition so that the rest of the line remains protected and does not blink.

CAUTION: Never start a message on column 1 since this position contains the protected line code. Placing data in column 1 overlays this code, and the entire line would be unprotected and returned as data when an ACCEPT command was executed.

6.6 EDITING DATA FIELDS

Unprotected fields on the screen accept any input, therefore the program must edit the data that is read into memory to ensure that the correct type of data has been entered. Tests such as ON SIZE ERROR, IF NUMERIC, IF ALPHABETIC, IF NEGATIVE, etc., can be performed as explained in the Conditional Statement section in Chapter 5 and illustrated in the following program extract.

```
WORKING-STORAGE SECTION.
77  E-NAME       PIC A(10).
77  E-QTY        PIC S9(4).
77  E-PRICE      PIC S999V99.
77  ERROR-FLAG   PIC 9 VALUE ZERO.
01  SCREEN-3 LINE IS NEXT PAGE.
    02  FILLER PIC X(4) LINE 2, COLUMN 2 VALUE 'NAME'.
    02  S-NAME PIC X(10) VALUE SPACE.
    02  FILLER PIC X(8) LINE 4 COLUMN 2.
                        VALUE 'QUANTITY'.
    02  S-QTY  PIC X(4) COLUMN 12 VALUE SPACES.
    02  FILLER PIC X(5) COLUMN 50 VALUE 'PRICE'.
    02  S-PRICE PIC X(5) VALUE SPACE.
01  SCREEN-IN.
    02  I-NAME  PIC X(10).
    02  I-QTY   PIC X(4).
    02  I-PRICE PIC X(5).
01  CLEAR-BLINK.
    02  FILLER  PIC X LINE 2 COLUMN 6 VALUE $E1.
    02  FILLER  PIC X LINE 4 COLUMN 11 VALUE $E1.
PROCEDURE DIVISION.
START.
    DISPLAY SCREEN-3.
GET-SCREEN.
    ACCEPT SCREEN-IN.
    PERFORM BLINK-OFF.
    MOVE I-NAME TO E-NAME.
    IF I-NAME EQUAL SPACES PERFORM NAME-ERR
        GO TO EDIT-QTY.
    IF E-NAME NOT ALPHABETIC PERFORM NAME-ERR.
EDIT-QTY.
    MOVE I-QTY TO E-QTY ON SIZE ERROR PERFORM QTY-ERR
        GO TO EDIT-PRICE.
    IF E-QTY IS NOT NUMERIC PERFORM QTY-ERR
        GO TO EDIT-PRICE.
    IF E-QTY IS NEGATIVE PERFORM QTY-ERR
        GO TO EDIT-PRICE.
    IF E-QTY IS GREATER THAN 10 PERFORM QTY-ERR.
EDIT-PRICE.
    MOVE I-PRICE TO E-PRICE ON SIZE ERROR
        PERFORM PRICE-ERR GO TO CHECK-MORE.
    IF E-PRICE IS NOT NUMERIC PERFORM PRICE-ERR
        GO TO CHECK-MORE.
    IF E-PRICE IS NEGATIVE PERFORM PRICE-ERR.
CHECK-MORE.
    IF ERROR-FLAG = 1
        DISPLAY @(2,24) $E2EAC4 'INVALID ENTRY' $EA
        GO TO GET-SCREEN.
NAME-ERR.
    DISPLAY @(2,6) $E0.
    MOVE 1 TO ERROR-FLAG.
NAME-ERR-EXIT.  EXIT.
QTY-ERR.
    DISPLAY @(4,10) $E0.
    MOVE 1 TO ERROR-FLAG.
QTY-ERR-EXIT.  EXIT.
PRICE-ERR.
    DISPLAY @(4,55) $E0.
    MOVE 1 TO ERROR-FLAG.
PRICE-ERR-EXIT.  EXIT.
BLINK-OFF.
    DISPLAY CLEAR-BLINK.
    MOVE ZERO TO ERROR-FLAG.
BLINK-OFF-EXIT.  EXIT.
```

In this program, the NAME field must not be blank and must have only alphabetic characters. The QUANTITY field must have only positive numeric integers that are not greater than 10.

The PRICE field must have only positive numeric characters that have no more than 3 positions to the left of the decimal point and no more than 2 decimal positions to the right of it.

The ON SIZE ERROR is usually included with a move in which the receiving field is defined as decimal or decimal with an implied decimal point.

NOTE: The blink clear code in CLEAR-BLINK was placed one column before the start of the field since screen control codes are always stored there.

The following technique can be used to set up a date with slash separators.

```
(Data Division)
    01  DATE-PRINT PIC ZZ/ZZ/ZZ.
(Procedure Division)
        MOVE DATE TO DATE-PRINT.
```

A display of DATE-PRINT after the MOVE would print the system date. If the system date contained zeroes, DATE-PRINT would contain only blanks.

6.7 WRITING TO THE PRINTER

The following program extract shows how data is printed.

```
(Environment Division)
     FILE-CONTROL.
         SELECT PRTFILE ASSIGN TO PRINTER.

(Data Division)
     FILE SECTION.
     FD  PRTFILE.  LINAGE IS 60 TOP IS 6 BOTTOM IS 0
         DATA RECORD IS PRINT-REC.
     01  PRINT-REC PIC X(132).
     WORKING-STORAGE SECTION.
     01  HEAD-LINE PIC X(80).
     01  DETAIL-1  PIC X(80).
     01  DETAIL-2  PIC X(80).

(Procedure Division)
         OPEN OUTPUT PRTFILE.
     PRINT-LOOP.
         IF LINAGE-COUNTER EQUAL ZERO
             WRITE PRINT-REC FROM HEAD-LINE
         WRITE-PRINT-REC FROM DETAIL-1
             AFTER ADVANCING 1 LINE.
         WRITE PRINT-REC FROM DETAIL-2
             AFTER ADVANCING 2 LINES.
         IF BREAK-KEY = 'Y' GO TO END-JOB.
         GO TO PRINT-LOOP.
```

The automatic top of page function is performed in the example shown above when LINAGE-COUNTER reaches 60 since LINAGE was defined as 60.

Another method of handling top-of-page is using the WRITE AFTER ADVANCING PAGE statement before the LINAGE-COUNTER reaches its maximum. The use of AFTER ADVANCING and BEFORE ADVANCING in the same program is not recommended.

The BREAK-KEY command determines whether or not the BREAK-KEY has been activated. If it has, appropriate actions are taken such as doing a wrap-up and terminating the program.

NOTE: If while trying to print data from the screen the printer deselects, check to see that you have entered the proper data.

CHAPTER 7 - FILE MANAGEMENT

The Codex File Management System (CFMS) supports sequential, indexed sequential files, and CODOS-SA sequential files. CFMS commands create and list these files; CODOS utility programs initialize and format data diskettes as well as provide backup and recovery capabilities. CODOS-SA files may be created directly by a COBOL program without using the CFMS utility program.

A maximum of 160 files are allowed per CFMS data diskette. Each file is designated by an 8-character name that is followed by a .DF (for data file) suffix or .SA (for CODOS-SA file). Space is allocated for a file on a data diskette one cluster (512 bytes) at a time as needed. CFMS files may also be stored on any system diskette, and a user may have files open on several diskettes simultaneously.

7.1 FILE ORGANIZATION

There are three types of file organization: sequential, indexed, and CODOS-SA.

7.1.1 SEQUENTIAL FILE ORGANIZATION

A sequential file is one whose records are organized in a consecutive manner. There is no identifying key associated with each record; therefore, records can be accessed only sequentially. Sequential files may be assigned to any type of input/output device. Sequential file organization is indicated in a COBOL program when the ORGANIZATION IS SEQUENTIAL clause is written in the Environment Division, or when the ORGANIZATION clause is omitted.

7.1.2 INDEXED FILE ORGANIZATION

Indexed files are those in which each record is associated with an identifying key. These indexed files may be accessed either directly or sequentially, but they must be assigned to input/output devices that are capable of direct access such as disk or diskette. Indexed file organization is indicated in the COBOL program by the ORGANIZATION IS INDEXED clause.

7.1.3 CODOS-SA FILE ORGANIZATION

A CODOS-SA file is a sequential file having the same format as source files produced by the EDITOR. CODOS-SA files are compatible with all utilities which are able to access files created with the EDITOR.

7.2 FILE ACCESS

The three methods of accessing files are sequential, random, and dynamic.

7.2.1 SEQUENTIAL ACCESS

Sequential access is the technique of referencing records serially within a file. The order in which records are read or written is determined implicitly by relative physical position within the file. This access method is specified by the ACCESS MODE IS SEQUENTIAL clause or is implied by the omission of that clause.

7.2.2 RANDOM ACCESS

Random access is the technique of reading and writing records of a file in an order that is specified by the programmer. It may only be used with files that have an ORGANIZATION IS INDEXED clause in their description. The record to be referenced is indicated by the value of a key at the time that the input/output command is issued. This access method is specified by the ACCESS MODE IS RANDOM clause. The RECORD KEY clause specifies the key.

7.2.3 DYNAMIC ACCESS

Dynamic access mode allows the file to be accessed either sequentially or randomly depending upon the I/O statement. It may only be used with files having an ORGANIZATION IS INDEXED clause. This access mode is specified by the ACCESS IS DYNAMIC clause. The RECORD KEY clause is also required.

7.3 FILE-HANDLING METHODS

A file-handling method is the effect of the combination of file organization, the manner in which the file is opened, and the access technique.

7.3.1 WITH SEQUENTIAL ACCESS

An OPEN OUTPUT statement for a file with sequential access creates a sequential file. New records replace any previous file contents.

An OPEN EXTEND statement adds new records to the end of an existing sequential file.

An OPEN INPUT statement for a file with sequential organization obtains records during a READ statement serially in the order in which they were originally written. For a file with indexed organization, the READ statement would obtain records serially in key value order, which is not necessarily in the order in which they were written.

7.3.2 WITH RANDOM ACCESS

An OPEN OUTPUT statement creates a new indexed file by deleting all records in an existing file. A RECORD KEY clause must be specified whose contents are consulted upon each WRITE statement.

An OPEN INPUT statement can only be used with an indexed file when random access has been specified. A RECORD KEY must also be specified whose contents are consulted upon each READ statement to locate the desired record within the file.

An OPEN I-O statement is only used with an indexed file whose RECORD KEY has been specified. This is the only statement that allows a file to be updated in one process instead of merely being referenced during a READ statement. Thus, the file can be both read and written to with the OPEN I-O statement.

7.4 CFMS FILE STRUCTURE

All files with the exception of CODOS-SA files, must be created using the CFMS file utility prior to attempting to access them with COBOL programs. Files created with the CFMS utility may have a sequential or indexed organization.

7.4.1 SEQUENTIAL FILES

Each sequential file record consists of a 1-byte record control character that is followed by the data portion of the record.

The maximum length of a sequential file record is 502 bytes.

7.4.2 INDEXED SEQUENTIAL FILES

The records of an indexed sequential file consist of a 1-byte record control byte, an n-byte record key, and the data portion of the record.

The maximum size of the combined record key and data portions of an indexed sequential record is 502 bytes.

7.5 DISKETTE INITIALIZATION

Before any files can be created on a diskette, the diskette must be properly formatted and a CODOS file catalog created. This is accomplished on a CODOS system using the appropriate utility program as explained in the CODOS User's Guide.

7.6 DATA FILE CREATION

Perform the following steps to create a data file with a .DF suffix. Refer to Table 7-1 for a listing of any error messages that may appear.

1. After the CODOS equal sign (=) prompt, key CFMS, then press RETURN. The following messages appear:

   ```
   CDX-68 FILE MANAGEMENT SYSTEM -- REV. X.XX
   SPECIFY DISK DRIVE NUMBER OF YOUR NEW DATA DISK.
   ```

2. Key a 0, 1, 2, or 3 as appropriate. The CFMS question mark (?) prompt appears to indicate that CFMS commands may now be entered.

3. Key: C or CREATE file-name,record-size,key-size,D

where: file-name =	8 alphabetic and/or numeric characters; the first character must be alphabetic. Special characters, such as the space and hyphen, are not permitted.
record-size =	502 bytes maximum for only the data and record key portions of a record.
key-size =	60 bytes maximum for the record key length of an indexed sequential file. This option is omitted when specifying a sequential file record.
D =	an optional entry that indicates that records with duplicate record keys are allowed in this indexed sequential file.

Examples:

```
CREATE PRSNNL01,500,9
CREATE NAMES,80,15,DUP
CREATE XACTIONS,125
```

4. Press RETURN. View the following messages that are displayed. The messages for a sequential file are:

```
SHOULD SEQUENTIAL FILE file-name. DF BE ADDED TO
disk-name DISKETTE WITH record-size BYTE RECORDS
?
```

   The messages for an indexed sequential file are:

```
SHOULD INDEXED SEQUENTIAL FILE file-name.DF BE ADDED TO
diskname DISKETTE WITH record-size BYTE RECORDS AND A
key-size BYTE KEY FIELD AND WITH NO DUPLICATIONS ALLOWED
?
```

5. Press Y, then RETURN to add the new file to the diskette. The question mark (?) prompt character appears on a new line.

6. To return to the CODOS equal sign (=) level, key END (or E) then press RETURN. The CODOS equal sign prompt appears.

NOTE: The CREATE command does not place data in the file. A COBOL program must do that.

Table 7-1. CFMS ERROR MESSAGES

Number	Message
1	INVALID FILE NAME
2	INCONSISTENT RECORD SIZE
3	INCONSISTENT KEY SIZE
4	ILLEGAL FILE TYPE
5	ILLEGAL PROCESSING MODE
6	FILE CURRENTLY BUSY
8	END-OF-FILE ENCOUNTERED
9	RECORD NOT FOUND
10	DUPLICATE KEY VALUE DETECTED
12	NEXT RECORD NOT DEFINED
13	ILLEGAL COMPARE INDICATOR
16	FILE CURRENTLY OPEN BY THIS USER
18	INVALID COMMAND BYTE
20	FILE NOT OPEN & COMMAND NOT OPEN
21	OPERATING SYSTEM ERROR
22	DISC SPECIFIED IN OPEN NOT MOUNTED

NOTE: CODOS disk error messages 0 through 25 can also be returned during CFMS processing. See Appendix D for a listing of these messages.

7.7 DATA FILE LISTS

To list all CFMS data files on a diskette, perform the following steps.

1. After the CFMS question mark (?) prompt, key LIST (or L), then press RETURN.
2. The following information is displayed for each file that has a .DF suffix and the CFMS file format.

```
DISK ID:________DATE:________USER:______________________
NAME         REC. SIZE     TYPE   KEY SIZE         DUPL.
_______.DF     ____        ____     ____           ____
_______.DF     ____        ____     ____           ____
_______.DF     ____        ____     ____           ____
_______.DF     ____        ____     ____           ____
```

3. Following the LIST process, key an END (or E) command as explained below.

7.8 THE END COMMAND

To end CFMS program execution, perform the following step.

1. Key END (or E), then press RETURN. The CODOS equal sign (=) appears.

7.8.1 CODOS-SA TYPE FILES

The CODOS-SA organization specifies a CODOS source image file which can be used as input/output. The CODOS-SA type data file allows the user to build, retrieve, and update in a sequential method. The ONLY input/output statements supported by the CODOS-SA type files are: OPEN, READ, WRITE, and CLOSE. A COBOL program can define a maximum of eight CODOS-SA type files.

CHAPTER 8 - USING A COBOL PROGRAM WITH THE CDX-68

COBOL source programs are created and maintained by using the EDITOR program. A source program is then compiled by the COBOL compiler to produce a loadable program module. The COBOL compiler and the EDITOR program operate under the direction of the Codex disk operating system (CODOS).

Before writing or compiling a COBOL program with CODOS, the programmer should have some general knowledge of the uses of the CODOS operating system and its utility programs that are available as an aid to software development. Information on this operating system may be found in the CODOS User's Manual.

8.1 READYING THE CDX-68 SYSTEM

The heart of the CDX-68 system is the CRT terminal in which the microcomputer, memory, interfaces, and other elements are stored. The three DIP switch groups on the back of the terminal must be set as follows for proper system operation.

GROUP ONE		GROUP TWO	
ENABLE	ON	DUPLEX	FULL
DISPLAY	OFF	PARITY	NO
			EVEN
TRANS MODE	OFF		
VIDEO INV	OFF	XMIT WORDS	8-BIT
A	ON	STOP BIT	1
B	OFF	CONNECTION	DIRECT
C	OFF	MODEM TYPE	103
SPEC CHAR	OFF	TURN AROUND	S-CHAN
LINE FREQ	OFF	CODE SEL	EOT

GROUP THREE

BAUD RATE-SELECT ONLY ONE.
(9600 is the most desirable)

8.2 ENTERING AND EDITING A NEW PROGRAM

Perform the following steps to enter a new COBOL program.

1. Ensure that the CODOS equal sign (=) prompt appears on the screen, then press the ALL CAPS key. This key's indicator illuminates to show that keyed letters will appear in uppercase.

2. Key: EDIT file-name.SA:drive-number:C

   where file-name = a maximum of 8 alphabetic and/or numeric characters; the first character must be alphabetic. Special characters, such as the space and hyphen, are not permitted.

   drive-number = the number of the diskette drive that is to store the source program, such as 0, 1, 2, etc.

   C = COBOL tabs set at columns 6, 9, and 12. The space bar is used as the tab key. Margin A is column 6; Margin B is column 9. Column X is Margin C, the continuation column.

Examples:

```
EDIT UPDATE01.SA:1;C
EDIT ORDERS05.SA:0;C
```

NOTE: To have COBOL tabs that conform to the ANS standard, refer to the EDITOR User's Guide.

3. Press RETURN. The screen clears, then displays:

   ```
   0010
   ```

4. Key the COBOL source program, pressing RETURN at the end of each line.

5. After the program has been keyed, press the F1 key. The cursor moves to the bottom of the screen. Key QUIT, then press RETURN. The CODOS equal sign (=) prompt appears.

6. To edit the COBOL program, refer to the EDITOR User's Guide for editing instructions. These instructions are too numerous to discuss in this manual.

8.2.1 DATETIME COMMAND

The CODOS command, DATETIME, requests the current date and time from the operator. This information is then stored for use in COBOL programs. Perform the following steps to use this command.

1. Ensure that the CODOS equal sign (=) prompt appears on the screen, then key:

   ```
   DATETIME
   ```

   and press RETURN. The following message appears:

   ```
   DATE (MMDDYY)-
   ```

2. Key the date, then press RETURN. The following message appears:

   ```
   TIME (HHMM)-
   ```

3. Key the time, then press RETURN. The system then returns to the CODOS (=) prompt.

8.3 COMPILING A SOURCE COBOL PROGRAM

To compile a source COBOL program, first ensure that the CODOS equal sign (=) prompt appears on the screen, then key:

```
     COBOL file-name;options
```

and press RETURN. File-name is the file name of a COBOL source program that is stored on disk or diskette. The default file suffix is .SA. The default disk number is zero. For example:

```
     COBOL PRTALL
     COBOL PRTALL.SA
     COBOL PRTALL.SA:0
```

The options are:

none	List only source program errors and object program size on the screen.
P	List only source program errors on the printer.
S	List the source program on the screen as it is compiled.
L	List the source program and the compiled object code to the screen.
SP	List the source program to the line printer as it is compiled.
LP	List the source program and generated object code to the line printer as it is compiled.
O=file-name	Saves the compiled code in the stated file-name for later execution. The default file suffix is .LO, and the default disk drive is 0.
D	Debug mode, which causes the compiled program to print each paragraph name on the printer during program execution.
P.n	Sets the line count on a printed page during compilation to n. The default page line count during compilation is 58.

NOTE: In general, the options may be written in any order. However, the O=file-name option must appear last.

Examples:

COBOL Command	Result after Compilation
COBOL PAY	Displays source program errors on the screen.
COBOL PAY;SPO=PAY	Prints the source program and any of the source program's errors. The compiled code is saved in the file PAY.LO:0.
COBOL PAY;PDO=FKEY05	Source program errors are printed. The compiled code is saved in the file FKEY05.LO:0. Trace coding is generated which prints each paragraph name on the printer during execution.

If desired, the COBOL compilation process may be halted by pressing the BREAK key.

8.4 EXECUTING THE COBOL PROGRAM

This section describes the two methods to execute compiled COBOL programs.

8.4.1 INTERACTIVE EXECUTION

Perform the following steps to execute a compiled COBOL program.

1. Ensure that the CODOS equal sign (=) prompt appears on the screen, then key:

   ```
   RUN
   ```

   and press RETURN. The following messages are displayed:

   ```
   CDX68 COBOL
   RUN TIME VERSION X.XX
   COPYRIGHT 1980 BY CODEX CORP.
   READY-
   ```

2. Key the name of a compiled COBOL program. If this name is in the form FKEYnn or SKEYnn, press the function or shift key that corresponds with its name.

NOTE: Files with names FKEY01 through FKEY16, or SKEY03 through SKEY16 are automatically assigned to the respective function or shift function keys.

For example, a program that had the name FKEY03 could be called after the READY prompt simply by pressing F3.

3. Observe any message that may appear. System messages are of two types: operator messages and programmer error messages.

   Operator messages request some form of operator interaction such as readying the printer. Some operator messages require a keyboard response: Y for yes and N for no. These messages are listed in Appendix B.

   Programmer error messages normally indicate a problem that is too serious for the operator to handle, such as a bug in the COBOL program or system. These messages have the following format:

   ```
   ** ERROR (xx) **
   a brief descriptive message
   AT (pc)
   ```

   where xx is the programmer message number, and pc is the COBOL program counter when the error occurred. The program counter that was running when the error occurred is aborted. Programmer error messages are listed in Appendix D.

4. The system can be returned to the CODOS "=" prompt level from the COBOL RUN mode simply by keying:

   ```
   STOP
   ```

   and pressing RETURN, or by keying:

   ```
   SHIFT F1.
   ```

   When the program executes a STOP RUN, the screen is not cleared, but the READY-prompt is written on screen line 23. This process allows the program to leave information on the screen for the operator.

8.4.2 BATCH EXECUTION

Perform the following steps to execute compiled COBOL programs.

1. Ensure that the CODOS equal sign (=) prompt appears on the screen, then key:

   ```
   RUN;B
   ```

   and press RETURN. The following messages are displayed:

   ```
   CDX68 COBOL
   RUN TIME VERSION X.XX
   COPYRIGHT 1980 BY CODEX CORP.
   READY-
   ```

2. Key the name of a compiled COBOL program.

   NOTE: Program files may not be assigned to function keys.

   Using the COBOL runtime in batch mode allows the user to execute COBOL programs from a CHAIN file. COBOL programs may also be executed interactively by using the ";B" option. In either case, the user may optionally specify disk and file substitution of data-files referenced by the COBOL program. This allows the user to assign disk and file names in the COBOL program, and then use this program to access disks and files that do not necessarily match those specified in the COBOL program.

   Please note that the "ACCEPT" verb will not obtain data from a CHAIN file as keyboard input cannot be entered when using a CHAIN file.

   Each program name should be prefaced with an equal sign (=); the disk drive number ":n" may (optionally) follow the program name. The default disk drive number is zero. The syntax "P=" object filename or "PROGRAM=" object filename may be used to make CHAIN files more readable.

   ```
   =EXAMPLE:1
   ```

   NOTE: To stop Runtime, key "=STOP," or if using a CHAIN file, have "=STOP" in the file.

   Up to four different VOLID:FILNAM subsitutions may be defined at Runtime (per program). Each substitution block defines the Runtime substitution of a previously specified VOLID:FILNAM with a (new) VOLID:FILNAM.

Multiple substitution blocks are separated by a comma. If multiple lines are used, each line must be terminated with a comma; a substitution may not be split between lines. If used, VOLID and FILNAMs must not exceed 8 characters.

EXAMPLE: A program called UPDATE identifies an employee master file as "MASTER:EMPLY" that normally resides on volume MASTER. A volume backup is kept with the volume name, "BACKUP." At program execution time, volume substitution would allow the program to execute using the EMPLY file on volume BACKUP.

```
=UPDATE;MASTER:EMPLY=BACKUP:EMPLY
```

If a file copy is only maintained on the MASTER volume under the name, "EMPLYOLD," the file name substitution would allow it to access the backup file copy on the same volume.

```
=UPDATE;MASTER:EMPLY=MASTER:EMPLYOLD
```

3. The syntax errors that will appear if there is any deviation from the expected form are:

```
0  :  Error in old VOLID specified for substitution
1  :  Error in old FILNAM specified for substitution
2  :  Error in new VOLID specified for substitution
3  :  Error in new FILNAM specified for substitution
4  :  = not found
5  :  Name exceeds 8 characters
6  :  Too many substitutions defined.
```

4. Observe any message that may appear. System messages are of two types: operator messages and programmer error messages.

Operator messages request some form of operator interaction such as readying the printer. Some operator messages require a keyboard response: Y for yes and N for no. These messages are listed in Appendix B.

Programmer error messages normally indicate a problem that is too serious for the operator to handle, such as a bug in the COBOL program or system. These messages have the followig format:

```
** ERROR (xx) **
a brief descriptive message
AT (pc)
```

where xx is the programmer message number, and pc is the COBOL program counter when the error occurred. The program counter that was running when the error occurred is aborted.

8.4.3 CHAIN FILE OPERATION EXAMPLE

A chain file mode of operation can be utilized to take full advantage of the Batch COBOL facilities to create a COBOL system job stream. This requires no operator interaction other than the initiation of the chain file. The following example illustrates this use. For further information on CHAIN files, see the CODOS User's Guide.

```
RUN;B                                     *(Begin Runtime)
PGM=INPUT                                 *(Input payroll data)
PGM=EDIT                                  *(Edit input data)
PGM=STOP                                  *(Terminate Runtime)
SORTM INPUT.SA,OUTPUT.SA,K=PARAMS.SA      *(SORT input file)
RUN;B                                     *(Restart Runtime)
PGM=UPDATE                                *(Update program)
PGM=PRINT;MASTER:EMPLY=MASTER:EMPLYNEW,   *(Print payroll
          MASTER:CHECK=MASTER:CHECKNEW      check file)
PGM=STOP                                  *(End Job stream)
```

8.5 DEBUGGING

The Codex COBOL system supports two debugging features: a paragraph name trace and the EXHIBIT statement. The paragraph name trace has already been listed as compilation option D (for debug mode). The execution of the EXHIBIT statement causes specified data or literals to be displayed on the printer. The format of the EXHIBIT statement is:

```
EXHIBIT [NAMED]    identifier-1     identifier-2
                   literal-1        literal-2     ...
```

The items specified in the EXHIBIT statement are separated by blanks when printed. When the NAMED option is included, the data for an identifier is preceded by the name of the identifier. For example:

```
EXHIBIT VALUE-A, VALUE-B.
EXHIBIT NAMED VALUE-A, VALUE-B.
EXHIBIT 'VALUES ARE' VALUE-A, VALUE-B.
EXHIBIT NAMED 'VALUES ARE' VALUE-A, VALUE-B.
```

Assuming that VALUE-A contains 123 and VALUE-B contains 'XYZ' then the data printed by the above statements will be:

```
123 XYZ
VALUE-A 123 VALUE-B XYZ
VALUES ARE 123 XYZ
VALUES ARE VALUE-A 123 VALUE-B XYZ
```

APPENDIX A - COBOL RESERVED WORDS

ACCEPT
ACCESS
ACTUAL
ADD
ADDRESS
ADVANCING
AFTER
ALL
ALPHABETIC
ALTER
ALTERNATE
AND
ARE
AREA
AREAS
ASCENDING
ASSIGN
AT
AUTHOR

BEFORE
BEGINNING
BLANK
BLOCK
BOTTOM
BREAK-KEY
BY

CALL
CANCEL
CF
CH
CHARACTERS
CLOCK-UNITS
CLOSE
COBOL
CODE
COLUMN
COMMA
COMP
COMP-1
COMP-2
COMP-3
COMPUTATIONAL
COMPUTATIONAL-1
COMPUTATIONAL-2
COMPUTATIONAL-3
COMPUTE
CONFIGURATION
CONTAINS
CONTROL
CONTROLS
COPY
CORR
CORRESPONDING
COUNT
CURRENCY

DATA
DATE
DATE-COMPILED
DATE-WRITTEN
DE
DEBUG-CONTENTS
DEBUG-ITEM
DEBUG-LINE
DEBUG-NAME
DEBUG-SUB1
DEBUG-SUB3
DEBUGGING
DECIMAL-POINT
DECLARATIVES
DELIMITED
DELIMITER
DEPENDING
DESCENDING
DETAIL
DISPLAY
DIVIDE
DIVISION
DOWN
DUPLICATES
DYNAMIC

ELSE
END
ENDING
ENTER
ENVIRONMENT
EQUAL
EQUALS
ERROR
EVERY
EXHIBIT
EXIT
EXTEND
EXIT
EXTEND

FD
FILE
FILE-CONTROL
FILE-LIMIT
FILE-LIMITS
FILLER
FINAL
FIRST
FOOTING
FOR
FROM

GENERATE
GIVING
GO
GREATER
GROUP

HEADING
HIGH-VALUE
HIGH-VALUES

I-O
I-O CONTROL
IDENTIFICATION
IF
IN
INDEX
INDEXED
INDICATE
INITIAL
INITIATE
INPUT
INPUT-OUTPUT
INSPECT
INSTALLATION
INTO
INVALID
IS
JUST
JUSTIFIED

KEY
KEYS

LABEL
LAST
LEADING
LEFT
LESS
LIBRARY
LIMIT
LIMITS

LINAGE
LINAGE-COUNTER
LINE
LINE-COUNTER
LINES
LINKAGE
LOCK
LOW-VALUE
LOW-VALUES

MEMORY
MODE
MODULES
MOVE
MULTIPLE
MULTIPLY

NAMED
NEGATIVE
NEXT
NO
NOT
NUMBER
NUMERIC

OBJECT-COMPUTER
OCCURS
OF
OFF
OH
OMITTED
ON
OPEN
OPTIONAL
OR
OUTPUT
OV
OVERFLOW

PAGE
PAGE-COUNTER
PERFORM
PF
PH
PIC
PICTURE
PLUS
POINTER
POSITION
POSITIVE
PROCEDURE
PROCEDURES
PROCEED
PROCESSING
PROGRAM
PROGRAM-ID

QUOTE
QUOTES

RANDOM
RD
READ
RECORD
RECORDS
REDEFINES
REEL
REFERENCES
RELEASE
REMAINDER
REMARKS
RENAMES
REPLACING
REPORT
REPORTING
REPORTS
RERUN
RESERVE
RESET
RETURN
REVERSED
REWIND
REWRITE
RF
RH
RIGHT
ROUNDED
RUN

SAME
SD
SEARCH
SECTION
SECURITY
SEEK
SEGMENT-LIMIT
SELECT
SELECTED
SENTENCE
SEQUENTIAL
SET
SIGN
SIZE
SORT
SOURCE
SOURCE-COMPUTER
SPACE
SPACES
SPECIAL-NAMES
STANDARD
STATUS
STOP
STRING
SUBTRACT
SUM
SYNC
SYNCHRONIZED

TALLY
TALLYING
TAPE
TERMINATE
THAN
THEN
THROUGH
THRU
TIMES
TO
TOP
TYPE

UCALL
UNIT
UNSTRING
UNTIL
UP
UPON
USAGE
USE
USING

VALUE
VALUES
VARYING

WHEN
WITH
WORDS
WORKING-STORAGE
WRITE

ZERO
ZEROES
ZEROS

APPENDIX B - DISPLAY CONTROL CODES

CODE	COMMAND
C0	CURSOR TO HOME POSITION
C1	CURSOR UP ONE LINE
C2	CURSOR DOWN ONE LINE
C3	CURSOR LEFT ONE COLUMN
C4	CURSOR RIGHT ONE COLUMN
C5	LOAD CURSOR POSITION
C6	READ CURSOR POSITION
C7	SET PAGE MODE
C8	SET SCROLL MODE
C9	SET TOP DISPLAY LINE
CA	SET LAST DISPLAY LINE
CB	SET LEFT DISPLAY COLUMN
CC	SET RIGHT DISPLAY COLUMN
CD	SET PROTECT MODE
CE	WRITE ABSOLUTE
CF	READ ABSOLUTE
D0	CHARACTER INSERT
D1	CHARACTER DELETE
D2	ENABLE KEYBOARD
D3	DISABLE KEYBOARD
D4	PAGE ERASE
D5	LINE ERASE
D6	LINE INSERT
D7	LINE DELETE
D8	CLEAR/HOME
D9	SEND PAGE
DA	TAB
DB	BACK TAB
DC	SET TABS
DD	START DATA
DE	END DATA
DF	SEND LINE
E0	SET BLINK
E1	RESET BLINK
E2	SET (FIELD) VIDEO INVERT
E3	RESET (FIELD) VIDEO INVERT
E4	SET HALF BRIGHT
E5	RESET HALF BRIGHT
E6	SET UNDERLINE
E7	RESET UNDERLINE
E8	SET NON-DISPLAY
E9	RESET NON-DISPLAY
EA	SET FIELD PROTECT
EB	RESET FIELD PROTECT
EC	SET TRANSPARENT MODE
ED	RESET TRANSPARENT MODE

CODE	COMMAND
EE	SET VIDEO INVERT-FULL SCREEN
EF	RESET VIDEO INVERT-FULL SCREEN
F1	TERMINAL RESET
F2	ALLOW STATUS INDICATORS
F3	DISALLOW STATUS INDICATORS
FA	ENABLE LOAD FUNCTION
FB	DISABLE LOAD FUNCTION
FC	SET DISPLAY SPECIAL CHARACTERS
FD	RESET DISPLAY SPECIAL CHARACTERS
07	BELL
08	BACKSPACE
09	HORIZONTAL TAB
0A	LINE FEED
0B	VERTICAL FEED
0C	FORM FEED
0D	CARRIAGE RETURN
18	CANCEL
98	STATUS--PARITY ERROR
9A	STATUS--RECEIVED OVERRUN
9B	STATUS--FRAMING ERROR

APPENDIX C - COMPILER ERROR MESSAGES

NUMBER	EXPLANATION
0	Illegal character has been used.
1	Continuation expected here.
3	Parse stack overflow (compiler error).
4	Variable contains more than 30 characters or a number contains more than 15 digits.
5	Syntax error.
7	The compiler has generated an out-of-range branch address.
8	COPY statements may not be nested.
9	File-name used in a COPY statement cannot be found. The file-name must end in SA.
10	This statement label has already been used.
11	This name is already defined.
12	Number of OCCURS is too large.
13	OCCURS is not allowed with FILLER.
14	This program is too big to execute.
15	The generated code and symbol table are overlapping in memory.
16	An invalid level number is being used.
17	REDEFINES statement is illegal.
18	PICTURE clause is too large.
19	The item previous to this statement is not a group. Therefore, this statement is out of order.
1A	Level 01 is missing.
1B	VALUE is not allowed on REDEFINES.
1C	VALUE clause not allowed in LINKAGE SECTION.
1D	Subroutine name exceeds the maximum allowed.
20	Too many statement labels are being used in a GOTO.
21	External name already used.
22	Dummy argument name already used.
23	Too many clauses in a PERFORM statement.
24	Already resolved.
30	Too many operands in an expression.
31	The number of subscripts are not equal to the number of dimensions.
32	Subscript out of range.
33	An undefined name has been used.
34	Operand must be an integer.
35	Name is not fully qualified.
36	More than three (3) subscripts are invalid.
37	A literal used with "ALL" must be only byte.
38	A literal used with "ALL" must not be numeric.
39	No OCCURS clause for the table.
40	An unknown edit code has been used.
41	There is a conflict between numeric and alphanumeric.
42	A replacement code must be the leading character.
43	Only one edit sign may be used.

NUMBER	EXPLANATION
44	Only one V or period is allowed in a picture.
45	Only one S is allowed in an edit picture.
46	The characters CR and DB must be the last two characters of the edit field.
47	There is a conflict between USAGE and PICTURE.
48	There is a conflict between VALUE and PICTURE.
49	Justified RIGHT/PICTURE conflict.
4A	BLANK when ZERO/PICTURE conflict.
50	Too many nested IF statements.
51	An incomplete logical expression has been found.
60	The RECORD KEY clause is missing.
61	The referenced file is not RANDOM.
62	An invalid device type is being used, or a disk file name is invalid.
63	A line or column number is too large. Line maximum is 24, column maximum is 80.
64	A line or column overlaps last field.
65	This field overflows the screen.
66	A field cannot start in column one.
67	CRT cursor controls must be the first of any elementary item.
68	A level 77 cannot be used for cursor controls.
69	FD missing.
6A	File description missing.
70	Operand must be numeric.
71	Arithmetic operation not recognized.
72	Invalid logical expression.
73	Operand cannot be a literal. Also can mean that the program has exceeded available memory.
74	Compute statement requires too many intermediate results to be saved.

APPENDIX D - EXECUTION ERROR MESSAGES

Operator Error Messages

FILE ******** DOES NOT EXIST
(An undefined COBOL program or function key was requested.

PRINTER NOT READY--RETRY (Y/N)?
(The printer is not ready. Ready it and press Y to retry; press N to abort the request and the program.)

Programmer Error Messages

The following messages are preceded by "ERROR (XX)" and followed by "AT (PC)" where XX is the error number, and PC is the COBOL program counter in hexadecimal code. This code corresponds with the codes printed on the programs's compilation listing.

ERROR NUMBER	ERROR MESSAGE
01	INVALID FILE NAME
02	INCONSISTENT RECORD SIZE
03	INCONSISTENT KEY SIZE
04	ILLEGAL FILE TYPE
05	ILLEGAL PROCESSING MODE
06	FILE CURRENTLY BUSY
12	NEXT RECORD NOT DEFINED
16	FILE CURRENTLY OPEN BY THIS USER
18	INVALID COMMAND BYTE
20	FILE NOT OPEN & CMD NE OPEN
21	OS ERROR
22	DISK SPECIFIED IN OPEN NOT MOUNTED
23	LOAD ERROR
24	ILLEGAL SUBSCRIPT
30	ILLEGAL I-O CALL
31	VERSIONS DON'T MATCH

APPENDIX E - SAMPLE COBOL PROGRAM

```
PAGE 001  ADD   .SA:0  APPENDIX E - SAMPLE COBOL PROGRAM.

0010 IDENTIFICATION DIVISION.
0020 *
0030 PROGRAM-ID. ADD
0040 AUTHOR.     CODEX CORP.
0050 DATE-WRITTEN.    03/18/80.
0060 DATE-COMPILED.   03/20/80.
0070 *
0080 *
0090 ENVIRONMENT DIVISION.
0100 *
0110 CONFIGURATION SECTION.
0120 SOURCE-COMPUTER.   CDX-68.
0130 OBJECT-COMPUTER.   CDX-68.
0140 INPUT-OUTPUT SECTION.
0150 FILE-CONTROL.
0160   SELECT DATAFILE
0170     ASSIGN TO DISK INV:INVFILE
0180     ORGANIZATION IS INDEXED
0190     ACCESS IS RANDOM
0200     RECORD KEY IS PART-NUMBER.
0210 *
0220 DATA DIVISION.
0230 *
0240 FILE SECTION.
0250 FD  DATAFILE
0260     LABEL RECORDS ARE OMITTED
0270     DATA RECORD IS MASTER-REC.
0280 COPY MASREC.
0290 *
0300 WORKING-STORAGE SECTION.
0310 77  ERRCK   PIC 9  VALUE ZERO.
0320 77  E       PIC 99 VALUE ZERO.
0330 77  X       PIC 99 VALUE ZERO.
0340 77  Y       PIC 99.
0350 77  X1      PIC 99.
0360 77  X2      PIC 99.
0370 77  TAB     PIC X   VALUE $09.
0380 77  WORK-5  PIC S9(5).
0390 01  FUNMSG LINE IS NEXT PAGE.
0400   02 FILLER  PIC X(3) COLUMN 37; VALUE 'ADD'.
0410 *
0420 *   DISPLAY ITEM NUMBER REQUEST ON LINE 2 OF SCREEN
0430 01 ITEM-NUM-LINE.
0440   02 FILLER  PIC X(11)  LINE 2; VALUE 'ITEM NUMBER'.
0450   02 FILLER  PIC X(8)   COLUMN 15; VALUE SPACES.
0460 01 ANS.
0470   02 ANS-ITEM PIC X(8).
0480 *  ERRROW & ERRCOL ARE TABLE POS OF ROW & COL FOR
0490 *  CURSOR FOR ERRORS
```

```
0500 01  ERRROW.
0510   02 FILLER PIC 99 VALUE 06.
0520   02 FILLER PIC 99 VALUE 08.
0530   02 FILLER PIC 99 VALUE 08.
0540   02 FILLER PIC 99 VALUE 08.
0550   02 FILLER PIC 99 VALUE 10.
0560   02 FILLER PIC 99 VALUE 10.
0570   02 FILLER PIC 99 VALUE 10.
0580   02 FILLER PIC 99 VALUE 12.
0590   02 FILLER PIC 99 VALUE 12.
0600   02 FILLER PIC 99 VALUE 12.
0610   02 FILLER PIC 99 VALUE 14.
0620   02 FILLER PIC 99 VALUE 14.
0630 01 ROW REDEFINES ERRROW PIC 99 OCCURS 12 TIMES.
0640 01 ERRCOL.
0650   02 FILLER PIC 99 VALUE 73.
0660   02 FILLER PIC 99 VALUE 16.
0670   02 FILLER PIC 99 VALUE 41.
0680   02 FILLER PIC 99 VALUE 71.
0690   02 FILLER PIC 99 VALUE 18.
0700   02 FILLER PIC 99 VALUE 43.
0710   02 FILLER PIC 99 VALUE 72.
0720   02 FILLER PIC 99 VALUE 18.
0730   02 FILLER PIC 99 VALUE 43.
0740   02 FILLER PIC 99 VALUE 73.
0750   02 FILLER PIC 99 VALUE 20.
0760   02 FILLER PIC 99 VALUE 43.
0770 01 COL REDEFINES ERRCOL PIC 99 OCCURS 12 TIMES.
0780 01  ERR-TABLE.
0790   02 FILLER PIC X(20) VALUE 'INVALID ENTRY       '.
0800   02 FILLER PIC X(20) VALUE 'ITEM ALREADY EXISTS'.
0810   02 FILLER PIC X(20) VALUE '    SUCCESSFUL      '.
0820   02 FILLER PIC X(20) VALUE '   DISK I/O ERROR'.
0830 01  ERR-MSG REDEFINES ERR-TABLE PIC X(20) OCCURS 4 TIMES.
0840 01 MASTER-AT-SCREEN.
0850   02 FILLER  PIC X     LINE 2,COLUMN 14; VALUE $EA.
0860   02 FILLER  PIC X(11) LINE 6;           VALUE 'DESCRIPTION'.
0870   02 FILLER  PIC X(16)        COLUMN 18; VALUE SPACES.
0880   02 FILLER  PIC X(16)        COLUMN 57; VALUE 'LOCATION/BIN'.
0890   02 FILLER  PIC X(5)                    VALUE SPACES.
0900   02 FILLER  PIC X(14) LINE 8;           VALUE 'COST'.
0910   02 FILLER  PIC X(7)                    VALUE SPACES.
0920   02 FILLER  PIC X(16)                   VALUE '  LIST PRICE'.
0930   02 FILLER  PIC X(7)                    VALUE SPACES.
0940   02 FILLER  PIC X(21)                   VALUE '  TRADE PRICE'.
0950   02 FILLER  PIC X(7)                    VALUE SPACES.
0960   02 FILLER  PIC X(16) LINE 10;          VALUE 'QTY ON HAND'.
0970   02 FILLER  PIC X(5)                    VALUE SPACES.
0980   02 FILLER  PIC X(18)                   VALUE 'QTY ON ORDER'.
0990   02 FILLER  PIC X(5)                    VALUE SPACES.
```

```
PAGE 003  ADD   .SA:0  APPENDIX E - SAMPLE COBOL PROGRAM.

1000   02 FILLER  PIC X(14)                  VALUE '  DATE ORDERED'.
1010   02 FILLER  PIC X(8)                   VALUE '(MMDDYY)'.
1020   02 FILLER  PIC X(6)                   VALUE SPACES.
1030   02 FILLER  PIC X(16) LINE 12;         VALUE 'REORDER POINT'.
1040   02 FILLER  PIC X(5)                   VALUE SPACES.
1050   02 FILLER  PIC X(18)                  VALUE '  STOCKING QTY'.
1060   02 FILLER  PIC X(5)                   VALUE SPACES.
1070   02 FILLER  PIC X(16)                  VALUE '  QTY/PKG FOR'.
1080   02 FILLER  PIC X(7)                   VALUE 'REORDER'.
1090   02 FILLER  PIC XXX                    VALUE SPACES.
1100   02 FILLER  PIC X(18) LINE 14;         VALUE 'VENDOR/TYPE CODE'.
1110   02 FILLER  PIC XXX                    VALUE SPACES.
1120   02 FILLER  PIC X(18)                  VALUE '  LEAD TIME'.
1130   02 FILLER  PIC XXX                    VALUE SPACES.
1140   02 FILLER  PIC X(23)       COLUMN 52; VALUE 'BACK ORDER FLAG'.
1150   02 FILLER  PIC X                      VALUE SPACE.
1160   02 FILLER  PIC X(13) LINE 17;         VALUE 'COMMENT'.
1170   02 FILLER  PIC X(8)                   VALUE SPACES.
1180 *      DATA FROM SCREEN
1190 01 DATAIN.
1200   02 SDESC  PIC X(16).
1210   02 SLOC   PIC X(5).
1220   02 SCOST  PIC X(7).
1230   02 SLIST  PIC X(7).
1240   02 STRADE PIC X(7).
1250   02 SQTY-HAND PIC X(5).
1260   02 SQTY-ORDER PIC X(5).
1270   02 SDATE.
1280    03 SDATE-MO  PIC XX.
1290    03 SDATE-DAY PIC XX.
1300    03 SDATE-YR  PIC XX.
1310   02 SREORDER   PIC X(5).
1320   02 SSTOCKING  PIC X(5).
1330   02 SQTY-PER-PK PIC XXX.
1340   02 SVEND      PIC XXX.
1350   02 SLEAD      PIC XXX.
1360   02 SBACK-O-FLAG PIC X.
1370   02 SCOMMENT PIC X(8).
1380 01   BLINK-OFF.
1390 *    LOC
1400   02 FILLER PIC X     LINE 6,COLUMN 73; VALUE $E3.
1410 *    COST
1420   02 FILLER PIC X     LINE 8,COLUMN 16; VALUE $E3.
1430 *    LIST PRICE
1440   02 FILLER PIC X            COLUMN 41; VALUE $E3.
1450 *    TRADE PRICE
1460   02 FILLER PIC X            COLUMN 71; VALUE $E3.
1470 *     QTY ON HAND
1480   02 FILLER PIC X     LINE 10,COLUMN 18; VALUE $E3.
1490 *     QTY ON ORDER
```

```
  02 FILLER PIC X              COLUMN 43; VALUE $E3.
*    DATE
  02 FILLER PIC X              COLUMN 72; VALUE $E3.
*    QTY FOR RE-ORDER
  02 FILLER PIC X      LINE 12,COLUMN 18; VALUE $E3.
*    STOCKING QTY
  02 FILLER PIC X              COLUMN 43; VALUE $E3.
*    QTY PER PACK
  02 FILLER PIC X              COLUMN 73; VALUE $E3.
  02 FILLER PIC X      LINE 14,COLUMN 20; VALUE $E3.
  02 FILLER PIC X              COLUMN 43; VALUE $E3.
*    ERASE LINE 24
  02 FILLER PIC X      LINE 24,COLUMN 3; VALUE $D5.
***********************************************************
PROCEDURE DIVISION.
AA-DRIVER-SECTION.
*
*  THIS SECTION WILL DO INITIALIZATION/OPEN/CLOSE
*  PERFORM PROCESSING ROUTINES & WRAP UP
*
   OPEN I-O DATAFILE
*    FUNCTION TYPE MSG
   DISPLAY FUNMSG.
*    GET PART NUM
GET-ITEM.
   MOVE ZERO TO ERRCK Y.
   DISPLAY ITEM-NUM-LINE.
   ACCEPT ANS.
*
   PERFORM BA-READ-RECORD THRU BA-READ-RECORD-EXIT.
   IF ERRCK EQUAL 1 GO TO GET-ITEM.
*    DISPLAY BUILD REC SCREEN
   DISPLAY MASTER-AT-SCREEN.
   PERFORM BB-DATA-EDIT THRU BB-DATA-EDIT-EXIT.
*
   PERFORM BC-DATA-UPDATE THRU BC-DATA-UPDATE-EXIT.
*
   CLOSE DATAFILE.
   STOP RUN.
AA-DRIVER-EXIT.   EXIT.
*
BA-READ-RECORD.
*
*   THIS ROUTINE WILL VALIDATE THAT THE PART IS IN THE FILE
*
   DISPLAY @(24,2) $D5.
   MOVE ANS-ITEM TO PART-NUMBER.
   READ DATAFILE
        INVALID KEY GO TO BA-READ-RECORD-EXIT.
*  RECORD ALREADY PRESENT
```

```
    MOVE 2 TO E.
    MOVE 1 TO ERRCK.
    PERFORM MSG-PRT THRU MSG-PRT-EXIT.
BA-READ-RECORD-EXIT.   EXIT.
*
*
BB-DATA-EDIT.
*
* THIS ROUTINE WILL READ SCREEN & VALIDATE DATA FIELDS
*
    ACCEPT DATAIN.
    DISPLAY BLINK-OFF.
    MOVE ZERO TO ERRCK Y.
*   EDIT LOCATION
    MOVE 1 TO X.
    MOVE SLOC TO WORK-5 ON SIZE ERROR
      PERFORM ERR-BLINK THRU ERR-BLINK-EXIT.
    PERFORM EDIT-CK THRU EDIT-CK-EXIT.
*     EDIT COST
CK-COST.
    MOVE SCOST TO COST
         ON SIZE ERROR GO TO COST-ERR.
    IF COST IS NUMERIC GO TO CK-LIST.
COST-ERR.
    MOVE 2 TO X.
    PERFORM ERR-BLINK THRU ERR-BLINK-EXIT.
CK-LIST.
*    VALIDATE LIST PRICE
    MOVE SLIST TO LIST-PRICE
         ON SIZE ERROR GO TO LIST-ERR.
    IF LIST-PRICE IS NUMERIC GO TO CK-TRADE.
LIST-ERR.
    MOVE 3 TO X.
    PERFORM ERR-BLINK THRU ERR-BLINK-EXIT.
CK-TRADE.
*     VALIDATE TRADE PRICE
    MOVE STRADE TO TRADE-PRICE
         ON SIZE ERROR GO TO TRADE-ERR.
    IF TRADE-PRICE IS NUMERIC GO TO CK-QTY-HAND.
TRADE-ERR.
    MOVE 4 TO X.
    PERFORM ERR-BLINK THRU ERR-BLINK-EXIT.
CK-QTY-HAND.
*     VALIDATE QTY ON HAND
    MOVE 5 TO X.
    MOVE SQTY-HAND TO WORK-5 ON SIZE ERROR
         PERFORM ERR-BLINK THRU ERR-BLINK-EXIT.
    PERFORM EDIT-CK THRU EDIT-CK-EXIT.
*  VALIDATE QTY-ON-ORDER
CK-ORD.
```

```
    MOVE 6 TO X.
    MOVE SQTY-ORDER TO WORK-5 ON SIZE ERROR
       PERFORM ERR-BLINK THRU ERR-BLINK-EXIT.
    PERFORM EDIT-CK THRU EDIT-CK-EXIT.
*   DATE EDIT
CK-DATE.
    IF SDATE EQUAL SPACES
        MOVE ZERO TO ORDER-MONTH ORDER-DAY ORDER-YEAR
        GO TO CK-REORD.
    IF SDATE IS NOT NUMERIC GO TO DATE-ERR.
    MOVE SDATE TO ORDER-DATE.
    IF ORDER-MONTH GREATER THAN 12 OR LESS THAN ZERO
          GO TO DATE-ERR.
    IF ORDER-DAY IS GREATER THAN 31 GO TO DATE-ERR.
    IF ORDER-YEAR IS GREATER THAN 76 GO TO CK-REORD.
DATE-ERR.
    MOVE 7 TO X.
    PERFORM ERR-BLINK THRU ERR-BLINK-EXIT.
CK-REORD.
*   EDIT REORDER POINT
    MOVE 8 TO X.
    MOVE SREORDER TO WORK-5 ON SIZE ERROR
      PERFORM ERR-BLINK THRU ERR-BLINK-EXIT.
    PERFORM EDIT-CK THRU EDIT-CK-EXIT.
*   EDIT STOCKING QTY
CK-STOCK.
    MOVE 9 TO X.
    MOVE SSTOCKING TO WORK-5 ON SIZE ERROR
       PERFORM ERR-BLINK THRU ERR-BLINK-EXIT.
    PERFORM EDIT-CK THRU EDIT-CK-EXIT.
*   QTY PER PACKAGE
CK-PACK.
    MOVE 10 TO X.
    MOVE SQTY-PER-PK TO WORK-5 ON SIZE ERROR
       PERFORM ERR-BLINK THRU ERR-BLINK-EXIT.
    PERFORM EDIT-CK THRU EDIT-CK-EXIT.
* EDIT VENDOR CODE
CK-VEND.
    MOVE 11 TO X.
    MOVE SVEND TO WORD-5 ON SIZE ERROR
       PERFORM ERR-BLINK THRU ERR-BLINK-EXIT.
    PERFORM EDIT-CK THRU EDIT-CK-EXIT.
*   EDIT LEAD TIME
CK-LEAD.
    MOVE 12 TO X.
    MOVE SLEAD TO WORK-5 ON SIZE ERROR.
       PERFORM ERR-BLINK THRU ERR-BLINK-EXIT.
    PERFORM EDIT-CK THRU EDIT-CK-EXIT.
*
CK-ANY-ERR.
```

```
   IF ERRCK EQUAL 1
     MOVE 1 TO E
     PERFORM MSG-PRT THRU MSG-PRT-EXIT
     PERFORM TAB-IT THRU TAB-IT-EXIT Y TIMES
   ELSE
      MOVE SLOC TO LOCATION-BIN
      MOVE SQTY-HAND TO QTY-ON-HAND
      MOVE SQTY-ORDER TO QTY-ON-ORDER
      MOVE SREORDER TO REORDER-POINT
      MOVE SSTOCKING TO STOCKING-QTY
      MOVE SQTY-PER-PK TO QTY-PER-PACK
      MOVE SVEND TO VENDOR-CODE
      MOVE SLEAD TO LEAD-TIME
      MOVE SDESC TO DESCRIPTION
      MOVE SBACK-O-FLAG TO BACK-ORDER-IND
      MOVE ZERO TO ISS-MONTH-1 ISS-MONTH-2 ISS-MONTH-3
      MOVE ZERO TO ISS-QUARTER-1 ISS-QUARTER-2 ISS-QUARTER-3
      MOVE SCOMMENT TO COMMENT.
BB-DATA-EDIT-EXIT.   EXIT.
*
EDIT-CK.
   IF WORK-5 EQUAL ZERO GO TO EDIT-CK-EXIT.
   IF WORK-5 NOT NUMERIC GO TO EDIT-ERR.
   IF WORK-5 POSITIVE GO TO EDIT-CK-EXIT.
EDIT-ERR.
   PERFORM ERR-BLINK THRU ERR-BLINK-EXIT.
EDIT-CK-EXIT.   EXIT.
*
ERR-BLINK.
*  ROUTINE TO BLINK FIELD IN ERROR
*
   DISPLAY @(ROW(X),COL(X)) $E2.
   IF Y EQUAL ZERO MOVE X TO Y.
   MOVE 1 TO ERRCK.
ERR-BLINK-EXIT.   EXIT.
*
*
BC-DATA-UPDATE.
*
*   THIS ROUTINE WILL WRITE THE MASTER TO DISK
*
   WRITE MASTER-REC INVALID KEY
                    MOVE 4 TO E
                    PERFORM MSG-PRT THRU MSG-PRT-EXIT
                    GO TO BC-DATA-UPDATE-EXIT.
*   SUCCESSFUL MSG
   MOVE 3 TO E.
   PERFORM MSG-PRT THRU MSG-PRT-EXIT.
BC-DATA-UPDATE-EXIT.   EXIT.
MSG-PRT.
```

```
3500 *
3510 *  ROUTINE TO PRINT MESSAGES ON SCREEN
3520 *
3530    DISPLAY @(24,4) $EAE0C4 ERR-MSG(E).
3540 MSG-PRT-EXIT.    EXIT.
3550 *
3560 TAB-IT.
3570    DISPLAY TAB.
3580 TAB-IT-EXIT.   EXIT.
```

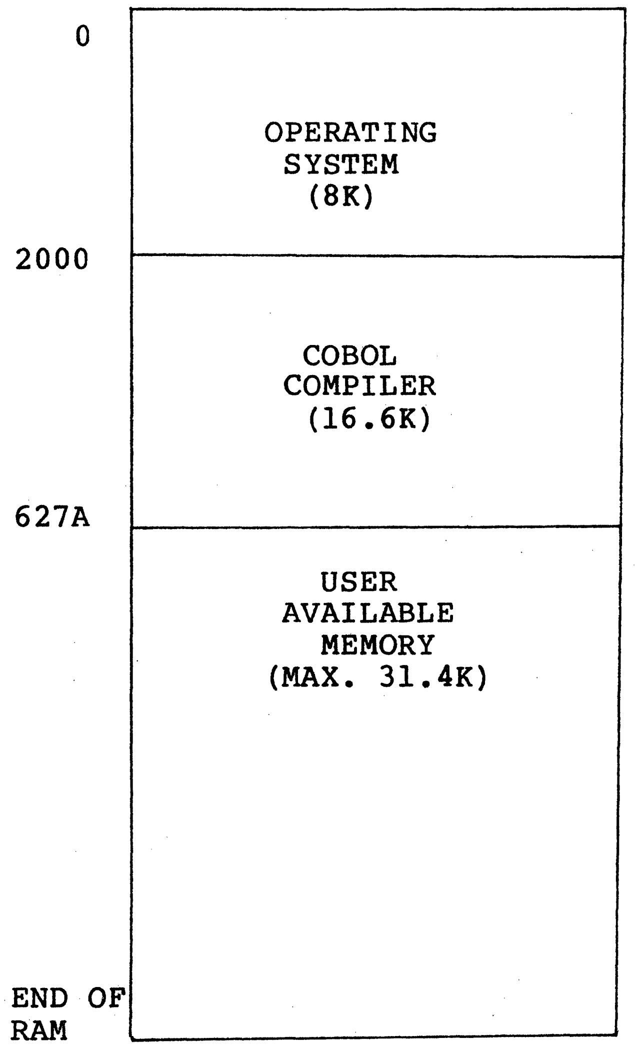
```
PAGE 001  MASREC  .SA:0  APPENDIX E - SAMPLE COBOL PROGRAM.

01  MASTER-REC.
  02 PART-NUMBER   PIC X(8).
  02 DESCRIPTION   PIC X(16).
  02 LOCATION-BIN  PIC 9(5).
  02 COST          PIC 9(4)V99.
  02 LIST-PRICE    PIC 9(4)V99.
  02 TRADE-PRICE   PIC 9(4)V99.
  02 QTY-ON-HAND   PIC 9(5).
  02 QTY-ON-ORDER  PIC 9(5).
  02 QTY-PER-PACK  PIC 999.
  02 REORDER-POINT PIC 9(5).
  02 STOCKING-QTY  PIC 9(5).
  02 VENDOR-CODE   PIC 999.
  02 LEAD-TIME     PIC 999.
  02 ORDER-DATE.
    03 ORDER-MONTH PIC 99.
    03 ORDER-DAY   PIC 99.
    03 ORDER-YEAR  PIC 99.
  02 ISSUE-COUNT.
    03 ISS-MONTH-1 PIC 9(5).
    03 ISS-MONTH-2 PIC 9(4).
    03 ISS-MONTH-3 PIC 9(4).
    03 ISS-QUARTER-1 PIC 9(4).
    03 ISS-QUARTER-2 PIC 9(4).
    03 ISS-QUARTER-3 PIC 9(4).
  02 BACK-ORDER-IND PIC X.
  02 COMMENT         PIC X(8).
```

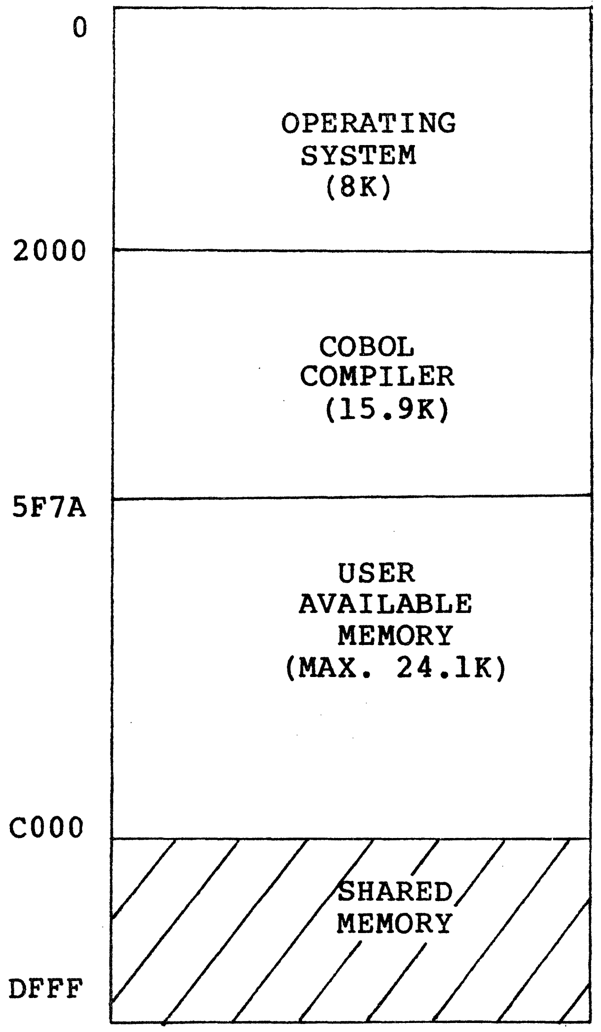
APPENDIX F - PERFORMANCE DATA

COMPILER MEMORY MAP

F5.1 Compiler Memory Map
(Floppy Disk)

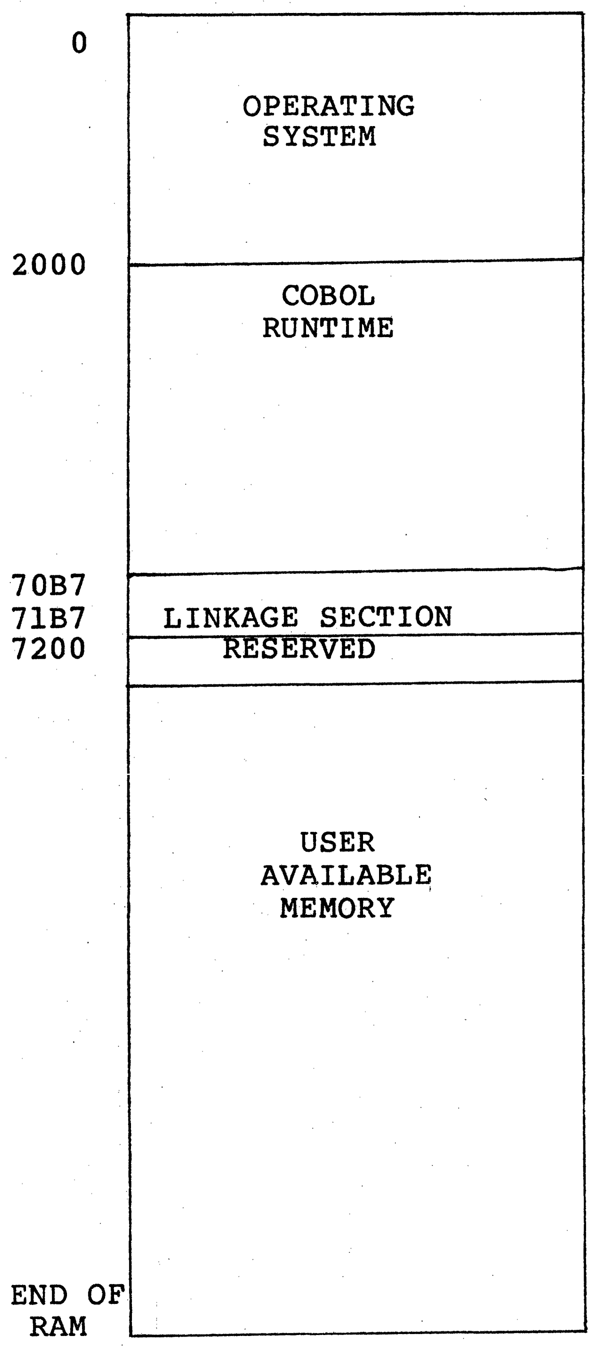


D5.1 Compiler Memory Map
(Rigid Disk)

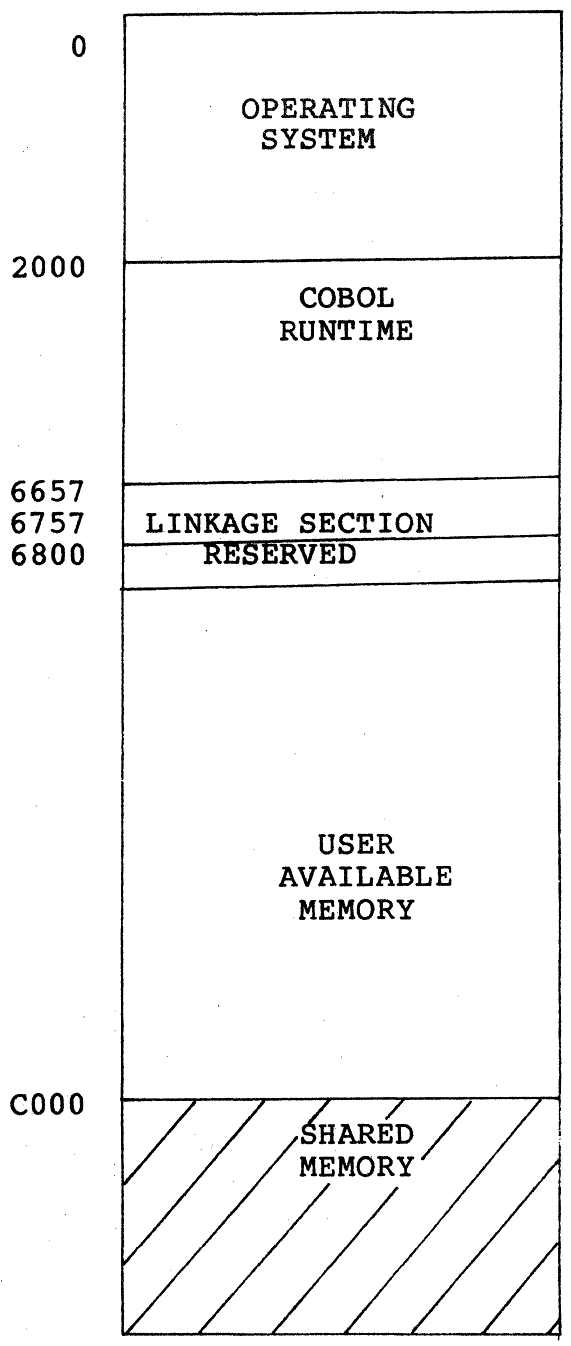


RUNTIME MEMORY MAP

F5.1 Runtime Memory Map
(Floppy Disk)



D5.1 Runtime Memory Map
(Rigid Disk)



STORAGE SIZE

RUNTIME INTERPRETER:

Average Number of Bytes per/Op Code:

OPEN/CLOSE	-	4 bytes
MOVE	-	5 bytes
ACCEPT	-	3 bytes
DISPLAY	-	4 bytes
READ	-	5 bytes
WRITE	-	5 bytes
PERFORM	-	5 bytes
SUBTRACT, ADD, MULTIPLY, DIVIDE	-	7 bytes
IF __THEN GOTO	-	11 bytes

READERS
COMMENTS

We welcome comments on the usefulness and readability of this manual. Your comments will help us improve the quality of future publications.

The COBOL REFERENCE MANUAL

	YES	NO
o Does this publication meet your needs?	☐	☐
o Did you find the material:		
- Easy to read and understand?	☐	☐
- Complete?	☐	☐
- Written for your level?	☐	☐

o Please indicate any comments in the space provided, if you have any.

FOLD THE FORM ON THE DOTTED LINES, STAPLE, ADD POSTAGE STAMP, AND MAIL.

fold fold

CODEX CORP.
INTELLIGENT TERMINAL SYSTEMS GROUP
3013 S. 52nd ST.
TEMPE, AZ 85252
ATTN: TECHNICAL DOCUMENTATION DEPT.

---

fold fold

codex

A Subsidiary of MOTOROLA INC.

CODEX CORPORATION
20 Cabot Boulevard
Mansfield, Massachusetts 02048

CODEX PHOENIX
INTELLIGENT TERMINAL SYSTEMS
2002 West 10th Place
Tempe, Arizona 85281
(602) 994-6580

Member of
IDCMA

Printed in U.S.A.