cobol-
documentation

COBOL- and associated software are accompanied by the following documents:

1. COBOL- USER'S GUIDE
   describes all the procedures associated with running COBOL- , writing COBOL programs, and running the programs with your hardware.

2. COBOL- REFERENCE MANUAL
   provides extensive descriptions of COBOL- 's statements, syntax and organization.

3. UTILITY SOFTWARE MANUAL
   describes the use of the MACRO- Assembler, LINK- Linking Loader and LIB- Library Manager with the COBOL- compiler.

To report software bugs or errors in the documentation, please complete and return the Problem Report at the back of this manual.

Acknowledgment

"Any organization interested in reproducing the COBOL report and specifications in whole or in part, using ideas taken from this report as the basis for an instruction manual or for any other purpose is free to do so. However, all such organizations are requested to reproduce this section as part of the introduction to the document. Those using a short passage, as in a book review, are requested to mention, 'COBOL' in acknowledgment of the source, but need not quote this entire section.

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

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

"Procedures have been established for the maintenance of COBOL. Inquiries concerning the procedures for proposing changes should be directed to the Executive Committee of the Conference on Data Systems Languages.

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

FLOW-MATIC (Trademark of Sperry Rand Corporation),
Programming for the UNIVAC (R) I and II, Data Automation
Systems copyrighted 1958, 1959, by Sperry Rand Corporation;
IBM Commercial Translator, Form No. F28-8013, copyrighted
1959 by IBM; FACT, DSI 27A5260-2760, copyrighted 1960 by
Minneapolis-Honeywell

have specifically authorized the use of this material in whole or in part, in the COBOL specification in programming manuals or similar publications."

—from the ANSI COBOL STANDARD
(X3.23-1974)

FUNCTIONAL DIFFERENCES - Release 3 vs. Release 4

The purpose of this section is to alert people using Release 3 of COBOL- to changes that may affect your existing systems when you change over to using Release 4. The following areas are the only ones that should affect the source code for programs that now compile and execute under Release 3.

** Both the compiler and runtime library have increased in size due to the enhancements and corrections made. This may cause you to run out of memory during compilation or loading of your programs. However, you should be able to take advantage of the CHAIN and segmentation features to work around the memory constraints.

** Every CRT driver has 5 new entry points that are needed to support Release 4 ACCEPT and DISPLAY functions. If you have written your own CRT driver, refer to Appendix A of the COBOL- User's Guide to see how to upgrade your driver module.

** The VALUE OF FILE-ID name is now terminated by the first space character. If you have used any file names with embedded spaces, the names will have to be changed.

** File status 91 has been added, as mentioned above under "Enhancements." You may want to change your programs to check for this new status condition and react accordingly. If your programs have no I-O error checking at all, some programs that worked before could possibly terminate with a runtime error under Release 4, if they manipulate files and the files have been damaged.

** The size of some group items in the Data Division may be decreased due to the removal of alignment for COMP items. You may need to check whether your code depends on the size of group items that may have elementary COMP items within them.

Corrections to Version 3.02

These corrections are separated into compiler and runtime system corrections.

Compiler Corrections

- In some cases, the division operation of a COMPUTE statement did not retain enough digits of accuracy.

- VALUE OF FILE-ID IS literal and VALUE OF FILE-ID IS data-name in the same program did not compile correctly, causing an error at runtime.

- DISPLAY of an item with a non-literal subscript generated incorrect object code, causing an error at runtime.

- DISPLAY and GO TO...DEPENDING statements with too may operands could cause the compiler to hang.

- ACCEPT of a subscripted item WITH UPDATE caused the compiler to generate undefined global symbols (named #Gnnnn) that prevented loading of the program with LINK.

- Use of the LIN or COL special registers as subscripts was not functional.

- A VALUE clause given for a group item could cause the compiler to hang.

- The ON OVERFLOW clause of STRING or UNSTRING could cause subsequent IF statements to function incorrectly.

- A missing or misspelled DIVISION header could cause the compiler to hang or to fill the disk with the object file and stop with a Disk Full message.

- PERFORM...VARYING followed by a MOVE caused the compiler to generate incorrect object code in some circumstances.

- PERFORM...UNTIL...NUMERIC did not function correctly.

- LINAGE clauses that included the optional word "LINES" caused a spurious compiler diagnostic.

- DATA DIVISION error recovery was unacceptable in the case of an inadvertent period placed between clauses. Recovery now occurs at the next level number.

Version 3.02 Changes to Version 3.01

The primary addition to COBOL- Version 3.01 was that of full support for both CP/M versions 1 and 2 at runtime. This feature is mentioned in the Version 4.0 enhancements. In addition to that change, the following runtime corrections were made.

** Use of the COL special register in DISPLAY statements did not work correctly, causing the cursor to jump to an unpredictable location.

** If the position-spec of a DISPLAY or ACCEPT statement omitted either the line or column position, the cursor would jump to an unpredictable location. This was corrected so that either or both could be omitted and the effect is that the current line and/or column position is left unchanged. Examples of valid position-specs are (1, 10) (, 10) (1,) and (,).

Version 3.01 Changes to Version 3.0

The changes listed below are all corrections that were made to the runtime system for Version 3.01.

** A program that executed a very large number of DISPLAY statements, or a subprogram that executed any DISPLAY statement, terminated at runtime with an arbitrary error message.

** The STOP literal statement did not work.

** If a LINE SEQUENTIAL file was opened in EXTEND mode and written to, the appended data was in the wrong format and unreadable by later programs.

** The control characters coded into $CLIST of a user-written CRT driver were not recognized.

COBOL- Version 4.0

Update Notice

Version 4.0 of COBOL- has been enhanced and upgraded in a number of areas. This Update Notice specifies the modifications that have been made since the Version 3.0 release of COBOL- . The changes fall into two categories: enhancements, which are new features; and corrections, which pertain to bugs in Version 3.0. Also given here are the functional differences between Release 3 and Release 4 of COBOL-.

ENHANCEMENTS

The new features of Version 4.0 are listed below. They bring COBOL to the level specified in the Reference Manual Appendix VII.

- A SCREEN SECTION format has been added to the Data Division that allows a concise description of one or more screen forms, including numerous options for each field and associations between screen fields and Working-storage items. Each entire screen form can be manipulated by a single ACCEPT or DISPLAY statement. The syntax is compatible with that used by Data General Interactive COBOL. Refer to the sections on ACCEPT, DISPLAY, and SCREEN SECTION in the COBOL- Reference Manual.

- Segmentation has been implemented to Level 1 of the ANSI Standard. Refer to Chapter 10 in the COBOL- Reference Manual.

- A CHAIN facility has been added that allows a COBOL program to load and execute any program from disk and to pass parameters in a manner similar to that used by CALL. Refer to the section on CHAIN and the chapter on Interprogram Communication in the COBOL- Reference Manual.

- Format 3 of the ACCEPT statement has been enhanced to allow a large number of user-defined keys that terminate the ACCEPT. Then, using a format 1 ACCEPT statement, a COBOL program can determine exactly which key terminated the previous format 3 ACCEPT. Refer to the section on ACCEPT in the COBOL- Reference Manual.

- The compiler's error count given at the end of compilation now includes any low-level diagnostics that were encountered. A high level diagnostic is produced to provide a reference to the line number that generated the error.

- 01 level data items larger than 4095 bytes may now be defined. Such items are still prohibited as operands for Procedure Division verbs such as MOVE, but they can be used to contain tables larger than the old limit of 4K bytes.

- Runtime file handling has been improved. Several bugs have been corrected (see the list below), and some file integrity protection measures have been added. The runtime system will now automatically close all files that are open when execution of a program terminates due to a STOP RUN statement, a CHAIN statement, or a runtime error condition. A CLOSE statement given for a file that is not open is ignored and treated as if the close were successful; the "Redundant Close" runtime error is no longer generated. All OPEN INPUT and OPEN I-O statements for relative and indexed files check the condition of the file to see whether its structure is intact. (The file structure can be damaged, for instance, by a system crash that occurs while the file is open for output). If the file is not intact, the file status will be set to '91', which is a new status value reserved for this purpose. Under this condition, OPEN INPUT statements succeed in opening the file, but OPEN I-O statements do not. This allows a program to recover any remaining undamaged information from the file. See the section on File Status Reporting in the COBOL- Reference Manual.

- Alignment of COMPUTATIONAL items to even-byte ("word") boundaries has been removed, possibly decreasing memory requirements, and allowing better programmer control of record and table sizes.

- The CP/M version of the runtime system has been enhanced to support both versions 1 and 2 of CP/M. The primary difference between the versions with regard to COBOL- is that CP/M version 2 supports various disk types and files as large as 8M bytes. This enhancement was also available in version 3.02.

- Support for several additional types of CRTs has been added. Also, the source code for all the CRT drivers is now included with each copy of COBOL- . Refer to APPENDIX A of the COBOL- User's Guide.

CORRECTIONS

The corrections for bugs encountered in Version 3.0 are listed below. First are the corrections that are new since Version 3.02 was released. Then the differences between Versions 3.02 and 3.01 and between 3.01 and 3.0 are given.

COBOL- VERSION 4.01

UPDATE NOTICE

Version 4.01 of COBOL- contains several fixes to problems in version 4.0. It also contains modifications that allow interfacing with the M/SORT* sort utility.

COMPILER CORRECTIONS

1. The compiler error message "ERRONEOUS SELECT" no longer over-prints page headings.

2. The reserved word FCR is now acceptable in the SAME RECORD AREA clause.

3. An undeclared data-name is now diagnosed in the VALUE OF FILE-ID data-name clause.

4. A 01-level data declaration with an OCCURS clause is now diagnosed as an error.

5. The compiler no longer terminates or "hangs" abnormally when certain level 88 conditionals or STRING statement constucts are in the source program.

6. The compiler now diagroses any failure to group an overlay segment into contiguous SECTIONS.

7. Problems in using the figurative constant ZERO in MOVE statements have been resolved.

8. A terminal period is now required after the COPY statement in order to read source code from an alternate source file.
   eg: COPY file-name.

RUNTIME SYSTEM CORRECTIONS

1. The M/SORT sort utility can now be linked with a COBOL- program using LINK- , when the SORT verb is emptied.

2. All currently open files are closed whenever the program terminates due to a run-time error or STOP RUN statement.

3. A COBOL program now executes correctly when a GO TO statement is executed in an overlay and the destination is in the root segment. This error caused the message: "SEG 00 LOAD ERR" to be printed.

COBOL- LANGUAGE SPECIFICATION CORRECTIONS

1. The COPY statement now requires a terminating period (.) after the file-name that will be used as a secondary input file. The syntax is now:

   COPY text-file.

* M/SORT is a sort utility from Microsoft that interfaces with a COBOL- program. It is a product distinct from COBOL- and is purchased separately.

Runtime System Corrections

** A large INDEXED file to which several insertions or deletions were being applied could cause the program to hang in some cases, leaving the file in an unusable condition.

** Records larger than a logical sector (128 bytes for CP/M) inserted into a RELATIVE file caused an incorrect duplicate key error when the position of the COBOL record aligned with the beginning of a logical sector.

** The boundary violation error status for a RELATIVE or INDEXED file was set to '34' instead of '24'.

** An OPEN statement that was not successful did not clear the runtime system's internal "file open" indicator. This caused a subsequent OPEN statement (without having done a CLOSE) to abort with the "Redundant Open" runtime error.

** After a file open in DYNAMIC access mode had been read sequentially to end-of-file, successful READ, WRITE, and REWRITE operations did not enable subsequent READ NEXT statements to access records based on the new position in the file.

** A RELATIVE file created in SEQUENTIAL mode and later extended in DYNAMIC mode could cause an incorrect duplicate key error.

** A STRING statement INTO an identifier that was larger than 128 bytes would immediately execute the ON OVERFLOW condition.

** ACCEPT of a numeric item WITH UPDATE would cause the initial placement of the cursor in the trailing sign position, forcing the operator to use rubout or line-delete before entering data.

** If an ACCEPT statement was used with a data-item that had a JUSTIFIED clause, and if no data was entered from the terminal before the return key was typed, the program would hang.

** 3.02 bug ONLY

OPEN EXTEND of a file that was less than 128 bytes long would write the new record(s) over the existing one(s).

cobol-
user's
guide

COBOL- User's Guide

Table of Contents

Section 1

OVERVIEW

1.1 Introduction

The purpose of this CCBCL- User's Guide is to give you practical information about getting a COBOL- program up and running on your computing equipment. All the steps necessary to use CCBCL- successfully -- compiling, loading, executing, etc. -- are carefully described in the following pages.

In this guide, examples and file names are given which are based on a CP/M version of CCBCL- If you are using another operating system, the format of commands and filenames will be slightly different. See Appendix D for a description of how CCBCL is used with your operating system.

1.2 Your Distribution Disk

The disk you receive from Microsoft contains the following files:

The CCBCL Compiler
- COBOL.COM
- COBCL1.OVR
- COBCL2.OVR
- COBCL3.OVR
- COBCL4.OVR

The Runtime System
- COBLIB.REL — the runtime library
- CRT Drivers — file whose names begin with CD
  - Source - CD__.MAC
  - Object - CD__.REL, CRTDRV.REL

Utility Software
- L80.COM — the Linking Loader
- LIB.COM -- the Library Manager
- M80.COM — the Macro Assembler
- CREF80.COM -- the Assembly Cross-Reference Program

Miscellaneous Files
- SQUARO.CCB
- CRTEST.CCB
- SEQCVT.COM
- CCPCCB.SUB

1.2.1 The COBOL Compiler

The compiler consists of a main program and four overlays. These five parts correspond to the five "phases" of compilation. The main program is always memory-resident and controls the transition from each phase to the next. The overlay portion of the main program compiles the IDENTIFICATION and ENVIRONMENT DIVISIONs. Overlay 1 is brought in to compile the DATA DIVISION. The PROCEDURE DIVISION is compiled by overlay 2. These 3 parts constitute the first pass of compilation. Their function is to create an intermediate version of the program, which is stored on the current disk in a file named STEXT.INT. Overlay 3 reads the intermediate file and creates the object code. Finally, overlay 4 allocates the file control blocks and checks certain error conditions. The intermediate file is then deleted.

1.2.2 The Runtime System

e runtime library consists of a group of subroutines that interpret the object code of your program produced by the compiler. These subroutines will be included with your object program when you perform the loading step. (See section 3 of this guide). Not all programs will require all of the library routines. The loader will search the library and automatically include the portions you need and exclude the ones you don't. The CRT drivers are provided to enable you to configure your system for the type of CRT terminal you have. You will need to select the appropriate driver. (See Appendix A of this guide). Once you have done so, that driver will be automatically included with each program you load with the linking loader. The driver provides cursor positioning and other functions to support interactive ACCEPT and DISPLAY statements.

1.2.3 Utility Software

The linking loader is used to link COBOL object programs with the runtime system. (See section 3 of this guide). The other utilities are provided for your convenience. Each of these programs is documented in the Utility Software Manual.

1.2.4 Miscellaneous Files

SQUARO.COB is a COBOL source program that computes the square root of the number you provide. It is used to verify that you have a working version of the compiler and runtime system.

CRTEST.COB is a COBOL source program that tests the functions of the interactive CRT driver (see Appendix A).

SEQCVT.COM is a special utility program that converts COBOL files from LINE SEQUENTIAL format to SEQUENTIAL format. The COBOL- SEQUENTIAL file format was changed when version 3.0 was released. SEQUENTIAL organization files created by earlier versions are in the format that is now known as LINE SEQUENTIAL.

COPCOB.SUB is a command file that copies the files on your distribution disk to a second disk. It is provided as a convenience.

1.3 Getting Started

The first thing you should do when you receive your disk is to make a copy to use and save the original disk as a backup. This may be done by using the COPCOB command file supplied or with some other disk copying facility you may have.

Having done that, you should verify your copy of the compiler and runtime system by compiling, loading, and executing the test program SQUARO.COB. To do this, refer to the examples given below in section 1.4.

Finally, if you intend to use the interactive ACCEPT and DISPLAY facility in your CCBOL program, you must select a CRT driver and configure it into your runtime system. This procedure need be done only once; thereafter your selected driver will automatically be included with each of your object programs. See Appendix A of this guide for full instructions.

1.4 Program Development Steps

Preparation of a CCBOL program for execution consists of three basic steps:

1. Creating the source file with a text editor
2. Compiling with the CCBOL Compiler
3. Loading with the Linking Loader

The source program is a file which consists of lines of ASCII text terminated by carriage-return line-feed. You can create it with . EDIT- or any other editor that uses 7-bit ASCII character codes. Line numbers may be included in columns 1-6 of each line and these may be 8-bit ASCII codes. The compiler ignores characters other than TAB and carriage return until column 7 is reached. TAB stops assumed by the compiler are at columns 7, 17, 25, 33, 41, 49, 57, 65, and 73. All characters beyond column 73 are ignored until a CR is encountered. If you use EDIT· , you automatically begin typing in column 7 of each inserted line.

Having created the source program file, the next step is to compile it. This is done by typing a command line that will execute the CCBOL compiler and provide the name of your source file. Under CP/M, you must be logged-in on the disk that contains the CCBOL compiler, since the compiler overlays are always read from the current disk. The following example shows a command to compile the test program SQUARO which is included on your distribution disk, assuming drive A contains a copy of that disk.

```
A> CCBOL SQUARO.REL,TTY:=SQUARO.COB
```

This command will compile SQUARO.COB, placing the relocatable object code in a file named SQUARO.REL and printing the listing on your terminal. A shorter notation of this same command line takes advantage of default file-name extensions assumed by the compiler:

```
A>CCBOL SQUARO,TTY:=SQUARO
```

The shortest notation of all uses a compilation switch to force generation of an object file that defaults to the filename SQUARO.REL:

```
A>COBOL ,TTY:=SQUARO/R
```

These three example commands all produce exactly the same results. A full description of the command line syntax is given in section 2.

Once the source program is compiled, the final step before execution is to load the program with the Linking Loader L80. This step converts your relocatable object program into an absolute version and combines it with the COBOL runtime system. This absolute version is built in memory, where it may then be saved on disk, executed directly, or both. The following is a command to load SQUARO and execute it without saving the absolute version.

```
A>L80 SQUARO/G
```

( ) assumes the extension .REL for the file SQUARO that is to be loaded. Once SQUARO has completed execution, you could not execute it again without performing the load command, since the absolute version was not saved. To save the absolute version in a disk file without executing it directly, type:

```
A>L80 SQUARO/N,SQUARO/E
```

Then to execute the program, simply type:

```
A>SQUARO
```

Since the absolute version is saved, it may be executed at any time without performing the load step. To combine the 2 examples so that the absolute version is saved and then executed directly, type:

```
A>L80 SQUARO/N,SQUARO/G
```

Refer to section 3 of this guide and to the Utility Software Manual for a full description of L80 commands.

Section 2

COMPILING COBOL PROGRAMS

2.1 CCBCL- Command Line Syntax

The CCBCL- compiler reads your COBOL source program file as input and produces a listing and relocatable object version of your program. The command line invokes the CCBCL Compiler and tells it the names of the 3 files to use. The syntax of the line is to type CCBCL followed by a space, followed by a command string, as described below. CCBCL- is read from the disk and then examines the command string. If it is OK, compilation begins. If not, it types the message "?Command Error" followed by an asterisk prompt, then waits for another command string. When compilation is complete, CCBCL- always exits to the operating system.

The format of a CCBCL command string is:

objectfile,listfile=sourcefile

The separator characters are the comma and the equal sign. No spaces are allowed. The terms used in the format are:

objectfile
the name of the file to which the object program is to be written

listfile
the name of the file to which the program listing is to be written

sourcefile
the name of the CCBCL program source file

Each file can be the name of a disk file or the name of a system device. The full description of a file name depends on your operating system. For CP/M, a file description has the form:

device:filename.extension

Here the separators are the colon and period, and the terms mean:

device
: the name of the system device, which can be a disk drive, terminal, line printer, or other device supported by the operating system. If the device is a disk, the filename must also be given. If not, the device name itself is the full file description. COBOL- recognizes the following symbolic device names:

  TTY: for the console terminal
  LST: for the system printer
  RDR: for the high-speed reader

filename
: the name of the file on disk. If filename is specified without a device, the current disk is assumed as the device.

.extension
: the extension of the filename given. If not specified, the following defaults are assumed:

  .COB for the source program file
  .PRN for the listing file
  .REL for the object program file

In the command string, the objectfile, listfile, or both may be omitted. If neither a listing file nor an object file is requested, COBOL- will check for errors and print the total on the console. If nothing is typed to the left of the equals sign, an object file is written on the same cevice with the same filename as the source file, but with the default extension for object files.

Examples:

Command String	Effect
,=PAYROLL	Compiles the source from PAYROLL.COB and produces only an error count, which is displayed on the console.
=PAYROLL	Compiles PAYROLL.COB and places the object into PAYROLL.REL. No listing is generated.
,TTY:=PAYROLL	Compiles the source from PAYROLL.COB and places the program listing on the terminal. No object program is generated.
PAYOBJ,LST:=PAYROLL	Compiles the source from PAYROLL.COB, places the listing on the printer, and places the object into PAYOBJ.REL.
PAYOBJ=B:PAYROLL	Compiles PAYROLL.COB from disk B and places the object into PAYOBJ.REL. No listing is generated.
PAYROLL,PAYROLL=PAYROLL	Compiles PAYROLL.COB, places the listing into PAYROLL.PRN, and places the object into PAYROLL.REL.

2.2 Compiler Switches

The command string may be modified by appending one or more switches, which affect the compilation procedure as described below. To add a switch, type a slash followed by the one-character switch name.

Switch	Action
R	Force the compiler to generate an object file. This shorthand notation causes the compiler to write the object file on the same disk and with the same filename as the source file, but with the default extension for object files.
L	Force the compiler to generate a listing file. As with /R, this notation causes the compiler to write the listing file on the same disk and with the same filename as the source file, but with the default extension for listing files.
P	Each /P allocates an extra 100 bytes of stack space for the compiler's use. Use /P if stack overflow errors occur during compilation (see section 2.3 below). Otherwise /P is not needed.

Examples of command strings using switches:

Command String	Is Equivalent To
,=PAYROLL/R	PAYROLL=PAYROLL or =PAYROLL
,=B:PAYROLL/L	,B:PAYROLL=B:PAYROLL
,=B:PAYROLL/R/L	B:PAYROLL,B:PAYROLL=B:PAYROLL
=PAYROLL/L/P	PAYROLL,PAYROLL=PAYROLL/P

2.3 Output Listings and Error Messages

The listing file output by CCBOL· is a line-by-line account of the source file with page headings and error messages. The page heading line is printed 3 lines from the top of the page and is followed by 2 blank lines. Each source line listed is preceded by a consecutive 4-digit decimal number. This is used by the error messages at the end to refer back to lines in error, and also by the runtime system to indicate what statement has caused a runtime error after it occurs.

Two classes of diagnostic error messages may be produced during compilation.

Low level flags are displayed directly below source lines on the listing when simple syntax violations occur. Remedial action is assumed in each case, as documented below, and compilation continues. If a low-level error occurs, a high-level diagnostic will be generated at the end of the listing that refers to the line number attached to the low-level error, so the error count given at the end includes both classes of errors.

Flag	Reason for Flag	Remedial Action by Compiler
"QLIT"?	Faulty quoted literal 1. Zero length 2. Improper continuation 3. Premature end-of-file (before ending delimiter)	 Ignore and continue. Assume acceptable. Assume program end.
LENGTH?	Quoted literal length over 120 characters, or numeric literal over 18 digits, or 'word' (identifier or name) over 30 characters.	Excessive characters are ignored.
CHRCTR?	Illegal character	Ignore and continue.
PUNCT?	Improper punctuation (e.g. comma not followed by a space).	Assume acceptable.
BADWORD	Current word is malformed such as ending in hyphen, or multiple decimal points in a numeric literal.	Ignore and continue.
SEQ #	Improper sequence number (includes case of out-of-order sequence number).	Accept and continue.
NAME?	Name does not begin with a letter (A - Z).	Accept and continue.
PIC = X	An improper Picture.	PIC X is assumed.
COL.7?	An improper character appears in source line character 'column' 7, where only * - / D are permissible.	Assume a blank in column 7.
AREA A?	Area A, columns 8-12, is not blank in a continuation line.	Ignore contents of Area A (assume blank).

High level diagnostic messages consist of two or three parts:

1. The associated source line number — four digits, followed by a colon (:).

2. An English explanation of the error detected by the compiler. If this text begins with /W/, then it is only a warning; if not, it is an error sufficiently severe to inhibit linkage and execution of an object program.

3. (Optional) The program element cited at the point of error is listed. Design of the high level diagnostic message text is such that no list of 'messages and error codes' is necessary; the messages are self-explanatory.

Regardless of whether there is a list device, or what the list device may be physically, a message displaying the total number of errors or warnings is always disolayed on the console at the end of compilation. This allows you to make a simple change to a COBOL program, recompile it without a listing and still know whether the compiler encountered any questionable statements in the program.

Two error messages that occur infrequently and are also displayed on the console must be noted. One is "?Memory Full" which occurs when there is insufficient memory for all the symbols and other information the compiler obtains from your source program. It indicates that the program is too large and must be decreased in size or split into separately compiled modules. The symbol table of data-names and procedure-names is usually the largest user of space during compilation. All names require as many bytes as there are characters in the name, and there is an overhead requirement of about 10 bytes per data-name and 2 bytes per procedure-name. On the average, each line in the DATA DIVISION requires about 14 bytes of memory during compilation, and each line in the PROCEDURE DIVISION requires about 3 1/4 bytes.

The other error message, "?Compiler Error", occurs when the compiler becomes confused. It is usually caused by one of two problems: either the stack has overflowed, in which case using the /P switch will solve it; or the compiler or one of the overlay files on the disk has been damaged, in which case you should try your backup copy. If neither of these solutions works, you can sometimes determine the cause by compiling increasingly larger chunks of your program, starting with only a few lines, until the error recurs. These two error conditions cause immediate termination of compilation.

2.4 Files Used by CCBOL·

In addition to the source, listing and object files used by CCBOL· , the following files should be noted.

First, there is a file called STEXT.INT which the compiler always places on the current disk. It is used to hold intermediate symbolic text between pass one and pass two of the compiler. It is created, written, then closed, read, and then deleted before the compiler exits. Consequently, you should never run into it unless the compilation is aborted.

Another file of concern is the file to be copied due to a COPY verb in the CCBOL program. (See the discussion of COPY in the CCBOL- Reference Manual). Remember that copied files cannot have COPY statements within them and the rest of the line after a COPY statement is ignored.

Finally CCBOL programs that use segmentation cause the loader to create a file for each independent segment of the program. The filenames are formed as follows. The filename itself is the PROGRAM-ID defined in the IDENTIFICATION DIVISION. The extension is .Vnn where nn is a two-digit hexadecimal number that is the segment number minus 49.

Section 3

LOADING COBOL PROGRAMS

The Microsoft Linking Loader LINK- is used to convert the compiled relocatable object version of your program into an absolute version that is executable. It automatically combines the required portions of the COBOL runtime system with your object program. The loader is also used to link one or more subprograms together with a main program. These subprograms may be specified individually or extracted from a library, and may be written in COBOL, FORTRAN- or MACRO- assembly language.

3.1 LINK- Command Line Syntax

The complete syntax for LINK- commands is given in Chapter 4 of the Utility Software Manual. However, some functions described there are not useful when loading COBOL programs. This section summarizes use of the loader for COBOL programs.

You may invoke LINK- in one of two ways: either type L80 followed by a carriage return and enter a command string when the asterisk prompt is typed, or type L80 followed by a space, followed by the command string on the same line.

The command string is a list of filenames separated by commas. Each filename specified is brought into memory by the loader and placed at the next available memory address. Switches are used in the command string to specify functions the loader is to perform. The command string may be broken up into many small strings and entered on different lines. The loader will prompt with an asterisk and wait for more command strings until one with a G or E switch has been processed and the loader exits to the operating system. Filenames are specified in the same manner as for the compiler, except that the default extension is always .REL for files to be read by the loader. Such files are all expected to be in relocatable object format, so they must have been previously compiled (or assembled).

Switches most useful when loading COBOL programs are:

Switch	Effect
filename/N	Directs the loader to save the executable program on disk with name <filename> when the loading process is complete.
/E	Directs the loader to complete the loading process and exit to the operating system. The loader searches COBLIB.REL and CRTDRV.REL to resolve undefined global symbols. The final step is to save the executable program on disk, provided that the /N switch was specified.
/G	Directs the loader to complete the loading process and begin execution of the program. As with /E, the COBOL runtime library is searched, and the executable program is saved if /N was specified.

Switches used occasionally when loading COBOL programs are:

Switch	Effect
/R	Immediately resets the loader to its initial state. The effect is as if the loader was aborted and then reloaded from disk.
filename/S	Directs the loader to search <filename> to resolve undefined global symbols. This command is used to selectively load CALLed subroutines from a user-built library.
/M	Prints a map of all global symbols and their values. Undefined globals appear with an asterisk after the name.
/U	Prints a list of all undefined global symbols.

Examples:

Command String

MYPRCG,SVPRCG/N/E

Loads MYPRCG.REL, saves the absolute version in SVPRCG.COM and exits to the operating system.

MYPRCG/G

Loads MYPRCG.REL and begins execution without saving the absolute version.

MYPRCG,SUBPR1,B:SUBPR2,MYPROG/N/E

Loads MYPROG.REL,SUBPR1.REL, and B:SUBPR2.REL. Saves the absolute version in MYPROG.COM and exits to the operating system.

MYPRCG/N,MYPRCG,MYLIB/S/E

Loads MYPRCG.REL searches MYLIB.REL for subroutines referenced by "CALL" statements, saves the absolute version in MYPROG.COM and exits to the operating system.

3.2 Subprograms

If you have organized your program into a main module and one or more subprogram modules, the loader can combine them into one executable program. Before loading, compile (or assemble) all modules so that you have a relocatable object version of each. Then execute the loader and specify in the command string the name of each module you want to load. The modules may be specified in any order. For example, if you have a compiled main program file MAINPG.REL and 2 subprogram files SUBPR1.REL and SUBPR2.REL, you may load the executable program and save it with any of the following load commands:

1. L80 MAINPG/N,MAINPG,SUBPR1,SUBPR2/E

2. L80
   *MAINPG/N,MAINPG,SUBPR1,SUBPR2
   */E

3. L80 SUBPR1,SUBPR2
   *MAINPG/N
   *MAINPG/E

3.3 Function Libraries

The Library Manager LIB- (CP/M versions only) allows you to collect any number of subprograms into a single file (a library) that can be searched by the loader. For example, if you have six subprograms named SUBPR1.REL through SUBPR6.REL that are used by different main programs, you could make them into a library with the following command.

```
LIB
*USRLIB=SUBPR1,SUBPR2,SUBPR3,SUBPR4,SUBPR5,SUBPR6/E
```

This will create a library file named USRLIB.REL. (See Section 3 of the Utility Software Manual for a full description of LIB- . Then if you have a main program MAINPG that CALLs SUBPR2 and SUBPR6, the load command:

```
L80 MAINPG/N,MAINPG,USRLIB/S/E
```

will load MAINPG and search USRLIB for SUBPR2 and SUBPR6.

When making a library, you need to make sure that all subprogram ID's are unique. Since all COBOL runtime routines in CCBLIB have names that begin with dollar sign, you should avoid the dollar sign in naming your subprograms.

Section 4

EXECUTING COBOL PROGRAMS

You may execute a CCBCL program in any of three ways. The first is to use the G switch in the loader command string as described in section 3.1. The second is simply to type the name of an executable program file as saved by using the N switch in the loader command string. Finally, you may execute a program directly from within another CCBCL program by using the CALL or CHAIN statement. Refer to Chapter 5 of the CCBCL· Reference Manual for an explanation of program CALL and CHAIN.

4.1 The Runtime System

The relocatable object version of your program produced by the compiler is not 8080 or Z80 machine code. Instead, it is in the form of a special object language designed specifically for COBOL instructions. The CCBCL· runtime system executes your program by examining each object language instruction and performing the function required. This includes all processing needed to handle CRT, printer, and disk file input and output.

The runtime system consists of a number of machine language subroutines collected into a library named CCBLIB.REL and a CRT driver named CRTDRV.REL. When you load your COBOL program, CCBLIB is automatically searched by the loader to find and load routines that are required to perform the instructions in your source program. The number of routines needed depends on the number of COBOL language features you have used in your main program and subprograms. If DISPLAY or ACCEPT statements are included in the source program, the loader automatically searches the file CRTDRV.REL to include the terminal-dependent functions.

The amount of memory required by a COBOL program at runtime equals the amount required to store the data items defined in the DATA DIVISION, plus about 500 bytes per file, plus about 12 bytes per line of the PROCEDURE DIVISION, plus up to 24K bytes for the runtime system.

4.2 Printer File Handling

Printer files should be viewed simply as a stream of characters going to the printer. Records should be defined simply as the fields to appear on the printer. No extra characters are needed in the record for carriage control. Carriage return, line feed and form feed are sent to the printer as needed between lines. Note however, that blank characters (spaces) on the end of a print line are truncated to make printing faster.

No "VALUE OF" clause should be given for a PRINTER file in the FD, but "LABEL RECORD IS OMITTED" must be specified. The BLOCK clause must not be used for printer files.

4.3 Disk File Handling

Disk files must have "LABEL RECORD IS STANDARD" declared and have a "VALUE OF" clause that includes a File ID. File ID formats are described in the Utility Software Manual. Block clauses are checked for syntax but have no effect on any type file.

The format of regular SEQUENTIAL organization files is that of a two-byte count of the record length followed by the actual record, for as many records as exist in the file. The LINE SEQUENTIAL organization has the record followed by a carriage return/line feed delimiter, for as many records as exist in the file. Both organizations pad any remaining space of the last physical block with control-Z characters, indicating end-of-file. To make maximum use of disk space, records are packed together with no unnecessary bytes in between.

The format of RELATIVE files is always that of fixed length records of the size of the largest record defined for the file. No delimiter is needed, and therefore none is provided. Deleted records are filled with hex value '00'. Additionally, six bytes are reserved at the beginning of the file to contain system bookkeeping information.

The format of INDEXED files is too complicated to include in this document. It is a complex mixture of keys, data, linear pointers, deletion pointers, and scramble-function pointers. It is doubtful that the COBOL programmer would require access to such information.

4.4 CRT Handling

4.4.1 Terminal Output

All output to the terminal is done by the DISPLAY statement. Characters are sent one at at time by the DISPLAY runtime module or by the CRT driver. If no cursor positioning was specified for any of the displayed items, a carriage-return and line-feed are sent following the last displayed item. Otherwise, no assumptions about carriage control are made by the DISPLAY module.

4.4.2 Keyboard Input

All input from the keyboard is done by the ACCEPT statement. One of two methods of input are used, depending on the type of ACCEPT being performed.

For a format 2 ACCEPT, a full line of input is typed, using the operating system facilities for character echo and input editing, ending with a carriage return. For this type, the character codes defined in the CRT driver have no effect.

For a format 3 or 4 ACCEPT, each character typed is read directly by the runtime ACCEPT module by using a call to the operating system. The ACCEPT module performs all necessary character echo and input editing functions. The editing control characters, function keys, and terminator keys are defined in the CRT driver (see Appendix A).

4.5 Runtime Errors

Some programming errors cannot be detected by the compiler but will cause the program to terminate prematurely when it is being executed. Each of those errors produces a four-line synopsis, printed on the console.

```
** RUN-TIME ERR:
reason (see list below)
line number
program-id
```

The possible reasons for termination, with additional explanation, are listed below.

REDUNDANT OPEN	Attempt to open a file that is already open.
DATA UNAVAILABLE	Attempt to reference data in a record of a file that is not open or has reached the "AT END" condition.
SUBSCRIPT FAULT	A subscript has an illegal value (usually, less than 1). This applies to an index reference such as I - 2, the value of which must not be less than 1.
INPUT/OUTPUT	Unrecoverable I/O error, with no provision in the user's COBOL program for acting upon the situation by way of an AT END clause, INVALID KEY clause, FILE STATUS item, DECLARATIVE procedure, etc.
NON-NUMERIC DATA	Whenever the contents of a numeric item does not conform to the given PICTURE, this condition may arise. You should always check input data, if it is subject to error (because "input editing" has not yet been done) by use of the NUMERIC test.

PERFORM OVERLAP	An illegal sequence of PERFORMs as, for example, when paragraph A is performed, and prior to exiting from it another PERFORM A is initiated.
CALL PARAMETERS	There is a disparity between the number of parameters in a calling program and the called subprogram.
ILLEGAL READ	Attempt to READ a file that is not open in the input or I-O mode.
ILLEGAL WRITE	Attempt to WRITE to a file that is not open in the output mode for sequential access files, or in the output or I-O mode for random or dynamic access files.
ILLEGAL REWRITE	Attempt to REWRITE a record in a file not open in the I-O mode.
REWRITE; NO READ	Attempt to REWRITE a record of a sequential access file when the last operation was not a successful READ.
REDUNDANT CLOSE	Attempt to close file that is not open.
OBJ. CODE ERROR	An undefined object program instruction has been encountered. This should occur only if the absolute version of the program has been damaged in memory or on the disk file.
FEATURE UNIMPL.	An object program instruction that calls for an unimplemented feature has been encountered. This should occur only because of a damaged object program.
GO TO. (NOT SET)	Attempt to execute an uninitialized alterable paragraph containing only a null GO statement.
FILE LOCKED	Attempt to OPEN after earlier CLOSE WITH LOCK.
READ BEYOND EOF	Attempt to read (next) after already encountering end-of-file.
DELETE; NO READ	Attempt to DELETE a record of a sequential access file when the last operation was not a successful READ.
ILLEGAL DELETE	Relative file not opened for I-O.
ILLEGAL START	File not opened for input or I-O.
NO CRT DRIVER	An ACCEPT or DISPLAY statement using cursor positioning is being executed, but no CRT driver has been selected. (See Appendix A of this guide.)
SEG nn LOAD ERR	An unrecoverable read error has occurred while trying to load a segment of a segmented program. The two digits nn are the hexadecimal notation of the segment number minus 49 and match the name of the file extension (.Vnn) on the disk.

In the case of program CHAINing, error messages may be generated by the CHAIN processing module. These errors are of the form "**CHAIN: problem" and also cause termination of the program. See Appendix B for the list of CHAIN error messages.

Appendix A

CONFIGURING THE CRT

A.1 General Instructions

To enable the interactive ACCEPT and DISPLAY functions, CCBCL- requires a terminal driver module that provides primitive terminal-dependent functions. The system expects to find this module under the name CRTDRV.REL when programs are linked with L80.

A module named CRTDRV is provided with each Release 4 CCBCL disk. This is a default dummy driver that will enable programs to link successfully and will provide support for the ANSI standard ACCEPT and DISPLAY statements. Programs that use cursor positioning in ACCEPT or DISPLAY statements and link with the default driver will not run successfully; they will abort with the "NO CRT DRIVER" runtime error message.

The CRTDRV module should be replaced with the driver appropriate to the type of terminal being used before linking any CCBCL programs compiled with Release 4. To do this, simply copy the appropriate driver to CRTDRV.REL. Microsoft has provided drivers for a wide range of pocular terminals; these are listed below. If none of these drivers is suitable, one may be constructed; see section A.3: "Writing a CRT Driver".

The CRT driver modules supplied by are relocatable object files whose names begin with the letters CD (for CRT Driver). The MACRO- source code for each driver is also included. Any driver will support more than one type of terminal if the terminals have compatible control sequences. If your terminal is not listed below, check section A.2 to compare your terminal's function codes with those of the supplied terminal drivers. If your terminal matches the code for any supplied driver, use it. The terminals and associated drivers are:

1.	ANSI standard terminal	CDANSI
2.	Lear-Siegler ADM3-A	CDADM3
3.	Beehive 100, 150	CDBEE
4.	Microbee 2	CDBEE
5.	Cromemco 3101, 3102	CDBEE
6.	SOROC IQ	CDSROC
7.	Hazeltine 1500	CDHZ15
8.	Heath WH19	CDWH19
9.	DEC VT52	CDWH19
10.	ADDS Regent Terminals *	CDADDS
11.	Perkin-Elmer	CDPERK
12.	Zentec Zephr	CDZEPH
13.	Intertec Superbrain	CDISB
14.	IMSAI VIO	CDADM3

* Supports ADDS Regent 40, 60, 100, and 200 terminals. The highlight video codes are not available on the Regent 20 and 25, but the CDADDS driver can be used if that code is removed.

A.2 Terminal Charts

The following pages describe the characteristics of the terminals for which drivers are supplied on your distribution disk. There is one page for each terminal supported.

Section I of each page defines the keys that are recognized by CCBOL- to perform the functions of ACCEPT. The value listed under the heading "Escape Code" is the integer that is available using a format 1 ACCEPT...FROM ESCAPE KEY if the key caused termination of a format 3 or format 4 ACCEPT statement. The value listed under the heading "Input Code" is the hexadecimal code generated by the terminal when that key is typed. The entry under "Key Label" gives the name of the key as shown on the keyboard.

Section II of each page shows the sequences of codes that are sent to the terminal from CCBOL- to perform the functions of DISPLAY and ACCEPT. Spaces are shown to separate codes in the list, but they are not part of the sequence sent to the terminal. Each two-digit number represents an absolute hexadecimal value. All other codes describe standard ASCII character codes, except for some shorthand abbreviations, which have the following explanations:

R1 The binary row (line) number plus decimal 31.
R2 The row number converted to 2 ASCII digits, sent high digit first.

C1 The binary column number plus decimal 31.
C2 The column number converted to 2 ASCII digits, sent high digit first.
C3 If the column number is less than 32, a decimal 95 is added to the number. Otherwise, column number minus one is used.

N/A Function not available on this terminal.

E1 If the cursor is at the home position, a clear screen code (hexadecimal 1A) is used. Otherwise, enough spaces are sent to blank the remainder of the screen and the cursor is moved back to its original position.
E2 Enough spaces are sent to blank the remainder of the line and the cursor is moved back to its original position.

N1 Ten null (binary zero) characters.

CDADDS

ADDS Regent Terminals
24 Lines 80 Columns

I. Keyboard Input	Input Code	Key Label
A. Editing Keys		
1. Line delete/Field delete	15	CONTROL-U
2. Character delete	7F	DEL
3. Forward Space	06	CONTROL-F
4. Back Space	08	CONTROL-H
5. Plus Sign	2B	+
6. Minus Sign	2D	-

B. Terminator Keys	Escape Code	Input Code	Key Label
1. Backtab	99	02	CONTROL-B
2. Escape	01	1B	ESC
3. Field terminators	00		
a. Tab		09	CONTROL-I
b. Carriage Return		0D	NEW LINE
c. Line Feed		0A	LINE FEED

C. Function Keys	Escape Code	Input Code	Key Label
1.	02	01	CONTROL-A
2.	03	03	CONTROL-C
3.	04	18	CONTROL-X

II. Output Functions	Code Sequence
A. Set Cursor Position	ESC Y R1 C1
B. Backspace Cursor	08
C. Cursor On	N/A
D. Cursor Off	N/A
E. Erase to End of Screen	ESC k
F. Erase to End of Line	ESC K
G. Sound Bell	07
H. Set Highlight Mode	ESC 0 P
I. Reset Highlight Mode	ESC 0 @

CDADM3

Lear-Siegler ADM-3A
24 Lines 80 Columns

I. Keyboard Input	Input Code	Key Label
A. Editing Keys		
1. Line delete/Field delete	15	CONTROL-U
2. Character delete	7F	DEL
3. Forward Space	0C	CONTROL-L
4. Back Space	08	CONTROL-H
5. Plus Sign	2B	+
6. Minus Sign	2D	-

B. Terminator Keys	Escape Code	Input Code	Key Label
1. Backtab	99	02	CONTROL-B
2. Escape	01	1B	ESC
3. Field terminators	00		
a. Tab		09	CONTROL-I
b. Carriage Return		0D	RETURN
c. Line Feed		0A	LINE FEED

C. Function Keys	Escape Code	Input Code	Key Label
1.	02	01	CONTROL-A
2.	03	03	CONTROL-C
3.	04	18	CONTROL-X

II. Output Functions	Code Sequence
A. Set Cursor Position	ESC = R1 C1
B. Backspace Cursor	08
C. Cursor On	N/A
D. Cursor Off	N/A
E. Erase to End of Screen	E1
F. Erase to End of Line	E2
G. Sound Bell	07
H. Set Highlight Mode	N/A
I. Reset Highlight Mode	N/A

CCANSI

ANSI Standard Terminal
24 lines 80 Columns

I. Keyboard Input	Input Code	Key Label
A. Editing Keys		
1. Line delete/Field delete	15	CONTROL-U
2. Character delete	7F	DEL, RUB
3. Forward Space	06	CONTROL-F
4. Back Space	08	CONTROL-H
5. Plus Sign	2B	+
6. Minus Sign	2D	-

B. Terminator Keys	Escape Code	Input Code	Key Label
1. Backtab	99	02	CONTROL-B
2. Escape	01	1B	ESC
3. Field terminators	00		
a. Tab		09	TAB, CONTROL-I
b. Carriage Return		0D	RETURN, ENTER
c. Line Feed		0A	LINE FEED

C. Function Keys	Escape Code	Input Code	Key Label
1.	02	01	CONTROL-A
2.	03	03	CONTROL-C
3.	04	18	CONTROL-X

II. Output Functions	Code Sequence
A. Set Cursor Position	ESC [R2 ; C2 f
B. Backspace Cursor	08
C. Cursor On	ESC [> 5 l
D. Cursor Off	ESC [> 5 h
E. Erase to End of Screen	ESC [0 J
F. Erase to End of Line	ESC [0 K
G. Sound Bell	07
H. Set Highlight Mode	ESC [7 m
I. Reset Highlight Mode	ESC [0 m

CDBEE

Beehive Terminals
24 lines 80 Columns

I. Keyboard Input	Input Code	Key Label
A. Editing Keys		
1. Line delete/Field delete	15	CONTROL-U
2. Character delete	7F	DEL
3. Forward Space	06	CONTROL-F
4. Back Space	08	CONTROL-H
5. Plus Sign	2B	+
6. Minus Sign	2D	-

B. Terminator Keys	Escape Code	Input Code	Key Label
1. Backtab	99	02	CONTROL-B
2. Escape	01	1B	ESC
3. Field terminators	00		
a. Tab		09	TAB, CONTROL-I
b. Carriage Return		0D	RETURN
c. Line Feed		0A	LINE FEED

C. Function Keys	Escape Code	Input Code	Key Label
1.	02	01	CONTROL-A
2.	03	03	CONTROL-C
3.	04	18	CONTROL-X

II. Output Functions	Code Sequence
A. Set Cursor Position	ESC F R1 C1
B. Backspace Cursor	08
C. Cursor On	N/A
D. Cursor Off	N/A
E. Erase to End of Screen	ESC J N1
F. Erase to End of Line	ESC K
G. Sound Bell	07
H. Set Highlight Mode	ESC I
I. Reset Highlight Mode	ESC m

CDHZ15 Hazeltine 1500 Series Terminals
24 Lines 80 Columns

I. Keyboard Input	Input Code	Key Label
A. Editing Keys		
1. Line delete/field delete	15	CONTROL-U
2. Character delete	7F	DEL
3. Forward Space	5D	]
4. Back Space	08	BACK SPACE
5. Plus Sign	2B	+
6. Minus Sign	2D	-

B. Terminator Keys	Escape Code	Input Code	Key Label
1. Backtab	99	5C	\
2. Escape	01	1B	ESC
3. Field terminators	00		
a. Tab		09	TAB
b. Carriage Return		0D	RETURN
c. Line Feed		0A	LINE FEED

C. Function Keys	Escape Code	Input Code	Key Label
1.	02	01	CONTRCL-A
2.	03	03	CONTRCL-C
3.	04	18	CONTRCL-X

II. Output Functions	Code Sequence
A. Set Cursor Position	~ DC1 C3 R1
B. Backspace Cursor	08
C. Cursor On	N/A
D. Cursor Off	N/A
E. Erase to End of Screen	~ CAN
F. Erase to End of Line	~ SI
G. Sound Bell	07
H. Set Highlight Mode	~ US
I. Reset Highlight Mode	~ EM

CDI58

Intertec Superbrain
24 Lines 80 Columns

I. Keyboard Input	Input Code	Key Label
A. Editing Keys		
1. Line delete/Field delete	18	CONTROL-X
2. Character delete	7F	DEL
3. Forward Space	06	CONTROL-F
4. Back Space	08	BACK SPACE
5. Plus Sign	2B	+
6. Minus Sign	2D	-

B. Terminator Keys	Escape Code	Input Code	Key Label
1. Backtab	99	02	CONTROL-B
2. Escape	01	1B	ESC
3. Field terminators	00		
a. Tab		09	TAB
b. Carriage Return		0D	RETURN
c. Line Feed		0A	LINE FEED

C. Function Keys	Escape Code	Input Code	Key Label
1.	02	01	CONTROL-A
2.	03	03	CONTROL-C
3.	04	04	CONTROL-D

II. Output Functions	Code Sequence
A. Set Cursor Position	ESC Y R1 C1
B. Backspace Cursor	08
C. Cursor On	N/A
D. Cursor Off	N/A
E. Erase to End of Screen	ESC ~ k
F. Erase to End of Line	ESC ~ K
G. Sound Bell	07
H. Set Highlight Mode	N/A
I. Reset Highlight Mode	N/A

COPERK

Perkin - Elmer Terminals
24 Lines 80 Columns

I. Keyboard Input	Input Code	Key Label
A. Editing Keys		
1. Line delete/Field delete	15	CONTROL-U
2. Character delete	7F	DEL
3. Forward Space	06	CONTROL-F
4. Back Space	08	BACK SPACE
5. Plus Sign	2B	+
6. Minus Sign	2D	-

B. Terminator Keys	Escape Code	Input Code	Key Label
1. Backtab	99	02	CONTROL-B
2. Escape	01	1B	ESC
3. Field terminators	00		
a. Tab		09	TAB
b. Carriage Return		0D	RETURN
c. Line Feed		0A	LINE FEED

C. Function Keys	Escape Code	Input Code	Key Label
1.	02	01	CONTROL-A
2.	03	03	CONTROL-C
3.	04	18	CONTROL-X

II. Output Functions	Code Sequence
A. Set Cursor Position	ESC X R1 ESC Y C1
B. Backspace Cursor	08
C. Cursor On	N/A
D. Cursor Off	N/A
E. Erase to end of screen	ESC J
F. Erase to end of line	ESC I
G. Sound Bell	07
H. Set Highlight mode	N/A
I. Reset Highlight Mode	N/A

COSROC

SOROC IQ Terminals
24 Lines 80 Columns

I. Keyboard Input	Input Code	Key Label
A. Editing Keys		
1. Line delete/Field delete	15	CONTROL-U
2. Character delete	7F	DEL
3. Forward Space	0C	CONTROL-L
4. Back Space	08	CONTROL-H
5. Plus Sign	2B	+
6. Minus Sign	2D	-

B. Terminator Keys	Escape Code	Input Code	Key Label
1. Backtab	99	02	CONTROL-B
2. Escape	01	1B	ESC
3. Field terminators	00		
a. Tab		09	CONTROL-I
b. Carriage Return		0D	RETURN
c. Line Feed		0A	LINE FEED

C. Function Keys	Escape Code	Input Code	Key Label
1.	02	01	CONTROL-A
2.	03	03	CONTROL-C
3.	04	18	CONTROL-X

II. Output Functions	Code Sequence
A. Set Cursor Position	ESC = R1 C1
B. Backspace Cursor	08
C. Cursor On	N/A
D. Cursor Off	N/A
E. Erase to End of Screen	ESC Y
F. Erase to End of Line	ESC T
G. Sound Bell	07
H. Set Highlight Mode	N/A
I. Reset Highlight Mode	N/A

CCWH19

Heath WH19/DEC VT52
24 Lines 80 Columns

I. Keyboard Input	Input Code	Key Label
A. Editing Keys		
1. Line delete/Field delete	15	CONTROL-U
2. Character delete	7F	DELETE
3. Forward Space	06	CONTROL-F
4. Back Space	08	BACK SPACE
5. Plus Sign	2B	+
6. Minus Sign	2D	-

B. Terminator Keys	Escape Code	Input Code	Key Label
1. Backtab	99	02	CONTROL-B
2. Escape	01	1B	ESC
3. Field terminators	00		
a. Tab		09	TAB, CONTRCL-I
b. Carriage Return		0D	RETURN
c. Line Feed		0A	LINE FEED

C. Function Keys	Escape Code	Input Code	Key Label
1.	02	01	CONTRCL-A
2.	03	03	CONTRCL-C
3.	04	18	CONTRCL-X

II. Output Functions	Code Sequence
A. Set Cursor Position	ESC Y R1 C1
B. Backspace Cursor	08
C. Cursor On	ESC y 5
D. Cursor Off	ESC x 5
E. Erase to End of Screen	ESC J
F. Erase to End of Line	ESC K
G. Sound Bell	07
H. Set Highlight Mode	ESC p
I. Reset Highlight Mode	ESC q

CDZEPH

Zentec Zephr
24 Lines 80 Columns

I. Keyboard Input	Input Code	Key Label
A. Editing Keys		
1. Line delete/Field delete	15	CONTROL-U
2. Character delete	7F	DEL
3. Forward Space	06	CONTROL-F
4. Back Space	08	CONTROL-H
5. Plus Sign	2B	+
6. Minus Sign	2D	-

B. Terminator Keys	Escape Code	Input Code	Key Label
1. Backtab	99	02	CONTROL-B
2. Escape	01	1B	ESC
3. Field terminators	00		
a. Tab		09	TAB, CONTROL-I
b. Carriage Return		0D	RETURN
c. Line Feed		0A	LINE FEED

C. Function Keys	Escape Code	Input Code	Key Label
1.	02	01	CONTROL-A
2.	03	03	CONTROL-C
3.	04	18	CONTROL-X

II. Output Functions	Code Sequence
A. Set Cursor Position	ESC 0 R1 C1
B. Backspace Cursor	08
C. Cursor On	N/A
D. Cursor Off	N/A
E. Erase to End of Screen	ESC Y N1
F. Erase to End of Line	ESC T
G. Sound Bell	07
H. Set Highlight Mode	ESC G 4
I. Reset Highlight Mode	ESC G 0

4.3 Writing a CRT Driver

A CRT driver should be written in assembly language and assembled with the assemoler M80. It consists of 14 entry points that must be declared as global labels by using M80 ENTRY statements. The source code of all drivers is supplied on your distribution disk (files named CD__.MAC) to serve as examples and reference for the following explanation.

Once the CRT driver is written, you can test all the functions and key codes by using the program CRTEST that is supplied on your distribution disk. Compile it, link it with your CRT driver using L80, and execute it, following the instructions it provides.

Five of the entry points simply contain data that describe the terminal and keyboard. $CRLEN is a byte that contains the number of lines on the terminal and $CRWID contains the number of columns. $CLIST, $TLIST and $FLIST are sequences of bytes that define keyboard codes that invoke the functions of ACCEPT. Note that these codes are not sent to the terminal to perform the function; they merely declare the keys that should be recognized by the ACCEPT module. All of these codes should be unique.

$CLIST defines the editing keys, which must be specified in the following sequence:

1. Line delete (Field delete)
2. Character delete
3. Forward space (Cursor forward)
4. Backspace (Cursor back)
5. Plus sign
6. Minus sign

The list is terminated by a byte containing zero.

$FLIST defines function keys that terminate a format 3 or a format 4 ACCEPT statement. The order of placement of codes in $FLIST determines the ESCAPE KEY value available to the ACCEPT ... FROM ESCAPE KEY statement. The first key generates a value of 02, the second 03, and so on, up to a maximum value of 39. The list is terminated by a byte containing zero.

$TLIST defines several keys, all of which terminate format 3 type ACCEPT statements. First in the list must be the backtab key. If used in a format 4 ACCEPT, this key causes termination of the current field and the cursor to move to the previous input field, if one exists. If used in a format 3 ACCEPT, the backtab key terminates the ACCEPT and sets an escape code value of 99. Next in the list is the escape key. This key terminates either a format 3 or format 4 ACCEPT and sets an escape code value of 01. In addition, it causes the program to execute the ON ESCAPE clause of a format 4 ACCEPT. Finally, there is a list of normal field terminator keys, terminated by a zero byte. Any key in this list terminates the current input field and sets the escape code value to 00. Termination of the field ends a format 3 ACCEPT, and moves the cursor to the next field in a format 4 ACCEPT. If the cursor was in the last input field, the entire ACCEPT statement is terminated.

The remaining 9 entry points are subroutines that perform terminal functions by sending codes to the terminal. Each code is sent by calling the external routine $OUTCH with the value in the A register. $OUTCH preserves the values in registers HL and DE.

$SETCR moves the cursor to a specific position on the screen. Upon entry, register H contains the specified row (line) number and register L contains the column number. Note that COBOL considers the top line of the screen to be row 1 and the leftmost column to be column 1.

$CURBK moves the cursor to the left one position without disturbing the displayed character at that position. Upon entry, register HL contains the current cursor position in sequential format (i.e., a number between 1 and n where n is screen width times length). Most terminals honor the ASCII backspace code to perform this function.

$ALARM sounds the terminal's audible tone or bell. Most terminals honor the ASCII bell code to perform this function.

$CUROF and $CURON instruct the terminal to inhibit or enable display of the cursor. Many terminals do not provide this facility, however, and a simple RET instruction is appropriate for drivers of those terminals.

$ERASE clears that portion of the screen from the current cursor position to the end. The cursor must be left in its original position. Upon entry, register HL contains the current cursor position in sequential format. Some terminals, such as the ADM-3A, do not provide an escape sequence to perform this function. The example driver CDADM3 provides a routine that sends enough blanks to clear the screen and then returns the cursor to its original position. This routine may be used for any terminal that does not provide its own erase function.

$EOL clears that portion of the screen from the current cursor position to the end of the line, without moving the cursor. Upon entry, register H contains the current row number and L contains the current column.

$HILIT puts the terminal in reverse video mode (or some other highlight mode if reverse video is not available).

$LOLIT puts the terminal back in normal mode (cancels the effect of $HILIT).

Appendix B

INTERPROGRAM COMMUNICATION

This appendix describes the format of parameters passed between a main program and a subprogram via a CALL USING statement or between two main programs via a CHAIN USING statement. This parameter linkage is handled entirely by the CCBCL- runtime system if both programs are written in CCBCL. However, if the CALLed or CHAINed program is written in assembly language or FORTRAN, sections B.1 and B.2 are of interest.

B.1 Subprogram Calling Mechanism

It is possible for a CCBCL program to call CCBCL subprograms or to call FORTRAN or assembler subroutines. However, it is not possible, currently, for a FORTRAN or assembler program to call a CCBCL subroutine. Therefore, this section pertains to CCBCL programs which call FORTRAN or assembler subroutines. The calling sequence described below is identical to that of Microsoft's FORTRAN- as it calls FORTRAN or assembler subroutines.

The CCBCL runtime system transfers execution to a subroutine by means of a machine language CALL instruction. The subroutine should return via the normal assembler or FORTRAN return instruction.

Parameters are passed by reference, that is, by passing the address of the parameter. The method of passing these addresses depends on the number of parameters. If the number of parameters is less than or equal to 3, they are passed in the registers:

parameter 1 in HL
parameter 2 in DE
parameter 3 in BC

If the number of parameters is greater than 3, then 1 and 2 are still passed in HL and DE, but BC points to a contiguous data block in memory which holds the list of parameter addresses.

The subroutine can expect only as many parameters as are passed, and the calling program is responsible for passing the correct number of parameters. Neither the compiler nor the runtime system checks for the correct number of parameters. It is also entirely up to you to determine that the type and length of arguments passed the calling program are acceptable to the called subroutine. Note that alphanumeric data is the only type that is stored in the same format in CCBCL and FORTRAN. None of the numeric types of data are interchangeable.

The stack space used by a COBOL program is contained within the program boundaries, so assembler programs that use the stack must not overflow or underflow the stack. The most certain way to assure safety is to save the COBOL stack pointer upon entering the routine and to set the stack pointer to another stack area. The assembler routine must then restore the saved COBOL stack pointer before returning to the main program.

To call a subprogram, use the name of the subprogram in the COBOL CALL statement. If the subprogram is an assembler or FORTRAN program, the name is defined by an ENTRY, SUBROUTINE, or FUNCTION statement. The name of a COBOL subprogram is as given in the PROGRAM-ID paragraph. Then link the subprogram to the main program using LINK- , as described in section 3.2 of this guide.

8.2 CHAIN Parameters

The parameters passed between programs with a CHAIN USING statement are stored at the highest available memory address. The memory layout is as follows, starting at the highest available address and proceeding towards location zero. First, 32 bytes are reserved for stack space. Then the first parameter in the USING list follows, preceded by its length in bytes. The parameter length is stored in two bytes, high-order byte first. The parameter itself is stored as a string of bytes in the same order as they were stored in the DATA DIVISION, beginning at the address of the length minus the length itself. Each parameter in the USING list follows in order, each preceded by its length. The CHAINed program must expect the same number and format of parameters as were passed, as no checking can be done by the compiler or runtime system.

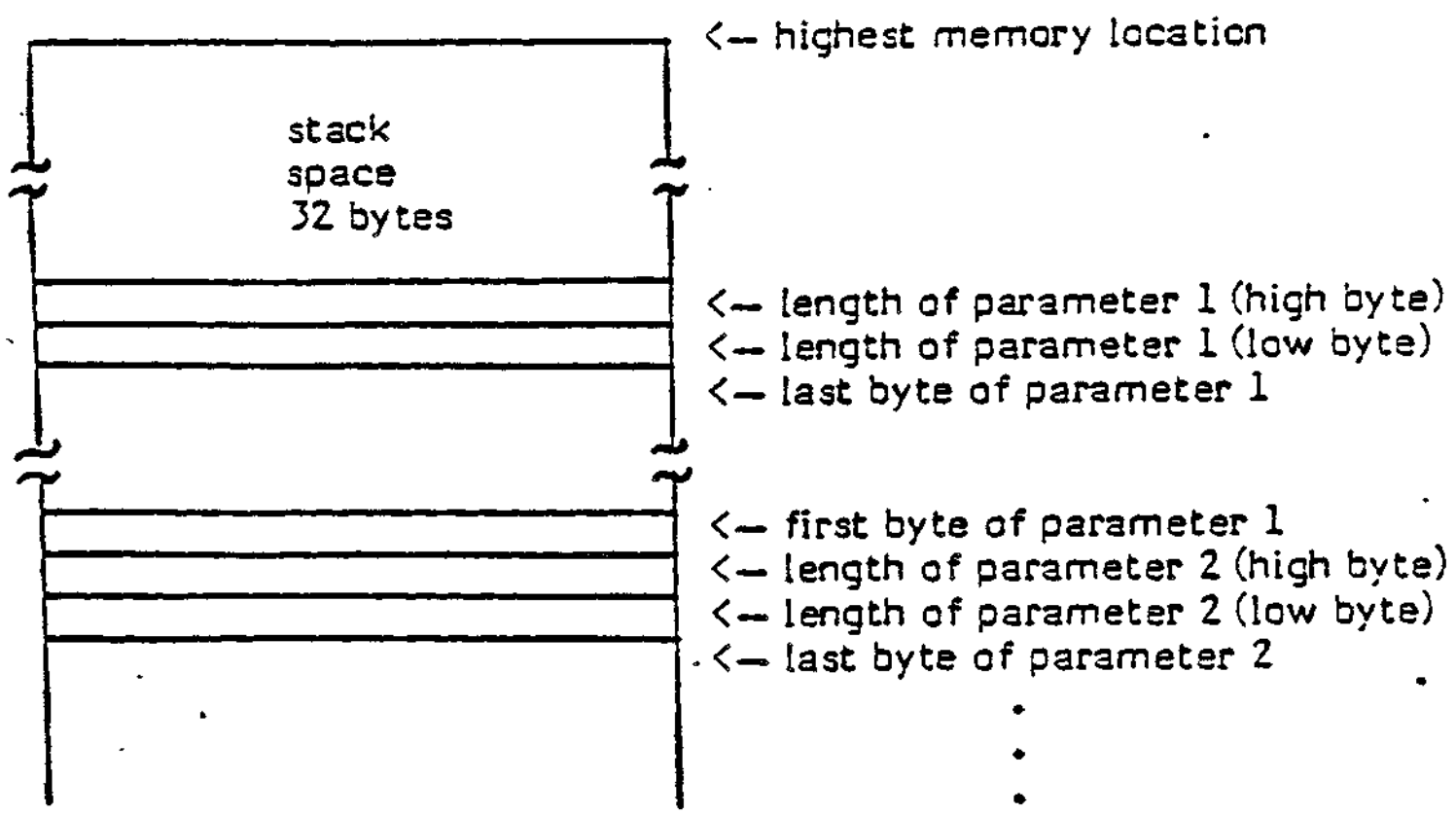


Figure 8-1

B.3 CHAIN Error Messages

During CHAIN processing, the normal mechanism for reporting runtime errors may have been overlayed by the new program. Therefore, the CHAIN processor generates its own error messages, which are of the form "**CHAIN: problem". The following is a list of possible "problems" and their causes.

Bad file name	The syntax of the file name that is to be loaded is not valid.
File not found	The specified file was not found on the disk.
Out of Memory	There was not sufficient memory available to load the new program. There must be enough memory for the larger of the CHAINing and CHAINed program, plus all CHAIN parameters, plus 256 bytes for the program loader.

APPENDIX C

CUSTOMIZATIONS

This appendix is intended for those of you who are handy with a debugger and/or assembly language and would like to change some of the built-in parameters of CCBCL-

C.1 Source Program Tab Stops

If tab characters (hex 09) are used in the CCBCL source program, the compiler converts them into enough spaces to reach the next tab stop as defined in its internal TAB table. As delivered, the table defines 9 stops at the following columns (counting from column 1):

7, 17, 25, 33, 41, 49, 57, 65, and 73

These may be changed by patching the table, whose address is 7 bytes from the start of CCBCL.CCM. There is one byte in the table for each tab stop. You may supply any values you like, proviced the numbers are in order and that there are still exactly 9 stops cefirec.

C.2 Compiler Listing Page Length

There is one byte in the compiler that defines the listing page length to be 55 (hex 37) lines. Its location is 6 bytes from the start of CCBCL.CCM and may be patched to any value between 1 and 255.

C.3 Runtime Day, Date, Time, Line Number

For all operating systems that do not provide date or time system calls, CCBCL uses the compiler release date for format 1 ACCEPT statements. For single-user systems, CCBCL always uses '00' for the line number. If you have a multi-user system or access to a system clock (or would like to use some other fixed date and time), you may replace the runtime module that performs this function. To do this, write an assembly language module according to the instructions given below, assemble it with MACRO- , and place it into CCBLIB.REL using the library manager. Assuming you name the module ACPDAT.MAC, a LIB- command to place it in the library is:

```
LIB
*NEWLIB=COBLIB<..ACPDAT-1>, ACPDAT
*COBLIB<ACPDAT+1..>/E
```

This will create NEWLIB.REL. You can then save COBLIB.REL and rename NEWLIB.REL to COBLIB.REL.

ACPDAT Module

Entry point: $ACPDT
Externals: $EVAL,$GETOP,$FLAGS,$ESKEY,$MOVE

This module handles the runtime support for the COBOL format 1 ACCEPT source statement:

ACCEPT identifier FROM	DAY DATE TIME ESCAPE KEY LINE NUMBER

It may be changed by modifying the ACLINE routine and by adding ACTIME, ACDAY, and ACDATE to the skeleton module given below. Each of these routines is entered with the address of the target storage area in the HL register. Each must exit by executing a JMP $GETOP, as indicated in the skeleton. The individual routines have the following requirements:

1. ACTIME - move an ASCII string representing the time (in form HHMMSSFF) to the target area.

2. ACDAY - move an ASCII string representing the Julian date (in form YYJJJ) to the target area.

3. ACDATE- move an ASCII string representing the date (in form YYMMDD) to the target area.

4. ACLINE - move 2 ASCII digits representing the line (CRT) number to the target area.

An external move routine is available to move a string of data from one address to another. It is used as follows:

```
EXT   $MOVE
      HL = address of source string
      DE = address of target area
      BC = length of the string in bytes
CALL  $MOVE
      HL = address of 1st byte beyond source
      DE = address of 1st byte beyond target
      BC = 0
```

Skeleton ACPDAT module:

```
          TITLE   ACPDAT - ACCEPT DAY/DATE/TIME/ESC KEY/LINE NUM
          ENTRY   $ACPDT
          EXT     $EVAL,$GETOP,$FLAGS,$ESKEY

$ACPDT:   POP     H
          INX     H
          MOV     A,M
          INX     H
          ANI     7
          STA     $FLAGS          ;SAVE ACCEPT OPTION
          CALL    $EVAL           ;GET TARGET ADDRESS
          LDA     $FLAGS
          CPI     2               ;WHICH OPTION?
          JM      ACDATE          ;DATE
          JZ      ACDAY           ;DAY
          CPI     4
          JC      ACTIME          ;TIME
          JZ      ACLINE          ;LINE NUMBER
ACESC:                            ;ESCAPE KEY CODE FROM ACCEPT
          XCHG
          LHLD    $ESKEY
          XCHG
ACESC1:   MOV     M,D
          INX     H
          MOV     M,E
          JMP     $GETOP
ACLINE:                           ;LINE (CRT) NUMBER - ALWAYS '00'
          LXI     D,3030H
          JMP     ACESC1
ACTIME:                           ;TIME: HHMMSSFF
          .
          .
          .
          JMP     $GETOP
ACDAY:                            ;DAY: YYJJJ
          .
          .
          .
          JMP     $GETOP
ACDATE:                           ;DATE: YYMMDD
          .
          .
          .
          JMP     $GETOP
          END
```

Appendix D

CCBCL- WITH NON-CP/M OPERATING SYSTEMS

Many of the examples and instructions given in the rest of this document refer to procedures and file names specific to the CP/M operating system. The syntax of command strings is the same for all operating sytems; however, the file specifications and switch separator character may differ. The A> shown in some examples is a prompt that is typed by CP/M and is not part of the command. If you have a different operating system, the following sections give descriptions of differences you should note.

D.1 TRSDOS Model II

D.1.1 Filename Descriptions

File specifications for CCBCL- and LINK- have the same form as described in the TRS-80 Owners Manual, namely:

filename/ext.password:d(diskette name)

The separator characters are the slash, period, and colon.

D.1.2 Your Distribution Disk

The names of the files on your distribution disk differ from the CP/M names and follow the TRSDOS file naming conventions. The disk contains the same files as a CP/M disk with the following exceptions:

*Only the CRT driver for the Model II terminal is included

*Some utility programs that are not available on the Model II are not included. They are:

LIB-
SEQCVT

D.1.3 Command Line Syntax

The command string for CCBOL- and LINK- have the same form under TRSDOS as under CP/M. However, the separator character for switches is a hyphen instead of a slash (since slash is used in TRSDOS file names) and the symbolic names of the console and printer devices are :TT and :LP respectively. Using the TRSDOS syntax, the following example shows how to compile, load, and execute the test program SQUARO/CCB.

```
TRSDOS READY
COBOL ,:TT=SQUARO-R
L80 SQUARO-N,SQUARO-E
SQUARO
```

The default file extensions assumed by CCBOL- are

/CCB for the source program file
/LST for the listing file
/REL for the object program file

D.1.4 DATE and TIME

CCBOL- uses the date and time supplied by TRSDOS to time-stamp the compiler listing page headings and to return the values requested by the ACCEPT TIME and DATE statements.

D.1.5 CRT Handling

Since the Model II has a built-in keyboard and display monitor, CCBOL is delivered configured for your hardware. You can ignore the description of configuring the CRT given in Appendix A. Figure D-1 shows how to use the keyboard for entering data for a format 3 or 4 ACCEPT statement and the supervisor calls used for the functions of DISPLAY. COBOL- uses supervisor call number 8 for all output to the screen except for the cursor position function, which uses supervisor call number 10.

TRS-80 Model II Terminal

I. Keyboard Input	Input Code	Key Label
A. Editing Keys		
1. Line delete/Field delete	15	CONTROL-U
2. Character delete	08	BACK SPACE
3. Forward Space	1D	→
4. Back Space	1C	←
5. Plus Sign	2B	+
6. Minus Sign	2D	-

B. Terminator Keys	Escape Code	Input Code	Key Label
1. Backtab	99	1E	↑
2. Escape	01	1B	ESC
3. Field terminators	00		
a. Tab		09	TAB
b. Carriage Return		0D	ENTER
c. Line Feed		1F	↓

C. Function Keys	Escape Code	Input Code	Key Label
1.	02	01	F1
2.	03	02	F2

II. Output Functions	Supervisor Call
A. Set Cursor Position	SVC 10 B=row-1 C=col-1 DE=00
B. Backspace Cursor	SVC 8 B=1C
C. Cursor On	SVC 8 B=01
D. Cursor Off	SVC 8 B=02
E. Erase to End of Screen	SVC 8 B=18
F. Erase to End of Line	SVC 8 B=17
G. Sound Bell	SVC 8 B=07
H. Set Highlight Mode	SVC 8 B=1A
I. Reset Highlight Mode	SVC 8 B=19

Figure D-1

cobol-
reference
manual

To report software bugs or errors in the documentation, please complete and return the Problem Report at the back of this manual.

Microsoft

COBOL Reference Manual

CONTENTS

Introduction

CCBOL is based upon American National Standard X3.23-1974. Elements of the CCBOL language are allocated to twelve different functional processing "modules."

Each module of the CCBOL Standard has two non-null "levels" — Level 1 represents a subset of the full set of capabilities and features contained in Level 2.

In order for a given system to be called CCBOL, it must provide at least Level 1 of the Nucleus, Table Handling and Sequential I-O Modules.

The following summary specifies the content of CCBOL with respect to the Standard.

Module — Features of CCBOL·

Nucleus — All of Level 1, plus these features of Level 2:

CONDITIONS:
- Level 88 conditions with value series or range
- Use of logical AND/OR/NOT in conditions
- Use of algebraic relational symbols (<,>,=)
- Implied subject, or both subject and relation, in relational conditions.
- Sign Test
- Nested IF statements; parentheses in conditions

VERBS:
- Extensions to the functions of ACCEPT and DISPLAY for formatted screen handling
- ACCEPTance of data from DATE/DAY/TIME
- STRING and UNSTRING statements
- COMPUTE with multiple receiving fields
- PERFORM VARYING . . . UNTIL

IDENTIFIERS:
- Mnemonic-names for ACCEPT or DISPLAY devices
- Procedure-names consisting of digits only
- Qualification of names (in Procedure Division statements only)

Module	Features
Sequential Relative, and Indexed I/O	All of Level 1 plus these features of Level 2: RESERVE clause Multiple operands in OPEN & CLOSE, with individual options per file VALUE OF FILE-ID IS data-name
Sequential I/O	EXTEND mode for OPEN WRITE ADVANCING data-name lines LINAGE phrase and AT END-OF-PAGE clause
Relative and Indexed I/O	DYNAMIC access mode (with READ NEXT) START (with key relationals EQUAL, GREATER, or NOT LESS)
Library	Level 1
Inter-Program Communication	Level 1
Table Handling	All of Level 1 Full Level 2 formats for SEARCH statement
Debugging	Special extensions to ANSI-74 Standard providing convenient trace style debugging. Conditional Compilation: lines with "D in column 7" are bypassed unlesss "WITH DEBUGGING MODE" is given in SOURCE-COMPUTER paragraph.
Segmentation	Level 1

CHAPTER 1

Fundamental Concepts of COBOL

1.1 Character Set

The COBOL source language character set consists of the following characters:

Letters A through Z
Blank or space
Digits 0 through 9
Special characters:
+ Plus sign
- Minus sign
* Asterisk
= Equal sign
> Relational sign (greater than)
< Relational sign (less than)
$ Dollar sign
, Comma
; Semicolon
. Period or decimal point
" Quotation mark
(Left parenthesis
) Right parenthesis
' Apostrophe (alternate quotation mark)
/ Slash

Of the previous set, the following characters are used for words:

0 through 9
A through Z
- (hyphen)

(Left parenthesis
) Right parenthesis
, Comma
. Period
; Semicolon

The following relation characters are used in simple conditions:

>
<
=

In the case of non-numeric (quoted) literals, comment entries, and comment lines, the COBOL character set is expanded to include the computer's entire character set.

1.2 Punctuation

The following general rules of punctuation apply in writing source programs:

1. As punctuation, a period, semicolon, or comma should not be preceded by a space, but must be followed by a space.

2. At least one space must appear between two successive words and/or literals. Two or more successive spaces are treated as a single space, except in non-numeric literals.

3. Relation characters and arithmetic operators should always be preceded by a space and followed by another space.

4. A comma may be used as a separator between successive operands of a statement, or between two subscripts.

5. A semicolon or comma may be used to separate a series of statements or clauses.

1.3 Word Formation

User-defined and reserved words are composed of a combination of not more than 30 characters, chosen from the following set of 37 characters:

0 through 9 (digits)
A through Z (letters)
- (hyphen)

All words must contain at least one letter or hyphen, except procedure-names which may consist entirely of digits. A word may not begin or end with a hyphen. A word is ended by a space or by proper punctuation. A word may contain more than one embedded hyphen; consecutive embedded hyphens are also permitted. All words are either reserved words, which have preassigned meanings, or programmer-supplied names. If a programmer-supplied name is not unique, there must be a unique method of reference to it by use of name qualifiers, e.g., TAX-RATE IN STATE-TABLE. Primarily, a non-reserved word identifies a data item or field and is called a data-name. Other cases of non-reserved words are file-names, condition-names, mnemonic-names, and procedure-names.

1.4 Format Notation

Throughout this publication, "general formats" are prescribed for various clauses and statements to guide the programmer in writing his own statements. They are presented in a uniform system of notation, explained in the following paragraphs.

1. All words printed entirely in capital letters are reserved words. These are words that have preassigned meanings. In all formats, words in capital letters represent actual occurrences of those words.

2. All underlined reserved words are required unless the portion of the format containing them is itself optional. These are key words. If any key word is missing or is incorrectly spelled, it is considered an error in the program. Reserved words not underlined may be included or omitted at the option of the programmer. These words are optional words; they are used solely for improving readability of the program.

3. The characters < > = (although not underlined) are required when present in statement formats.

4. All punctuation and other special characters represent actual occurrences of those characters. Punctuation is essential where it is shown. Additional punctuation can be inserted, according to the rules for punctuation specified in Section 1.2. As separators, all commas, semicolons and periods must be followed by a space (or blank).

5. Words printed in lower-case letters in formats represent generic terms (e.g., data-names) for which the user must insert a valid entry in the source program.

6. Any part of a statement or data description entry that is enclosed in square brackets ([ ]) is optional. Parts between matching braces ({ }) represent a choice of mutually exclusive options.

7. Certain entries in the formats consist of a capitalized word(s) followed by the word "Clause" or "Statement." These designate clauses or statements that are described in other formats, in appropriate sections of the text.

8. In order to facilitate reference to lower-case words in the explanatory text, some of them are followed by a hyphen and a digit or letter. This modification does not change the syntactical definition of the word.

9. Alternate options may be indicated by separating the mutually exclusive choices by a vertical stroke, e.g.:

AREA | AREAS is equivalent to $\left\{\begin{matrix}\text{AREA}\\\text{AREAS}\end{matrix}\right\}$

10. The ellipsis (...) indicates that the immediately preceding unit may occur once, or any number of times in succession. A unit means either a single lower-case word, or a group of lower-case words and one or more reserved words enclosed in brackets or braces. If a term is enclosed in brackets or braces, the entire unit of which it is part must be repeated when repetition is specified.

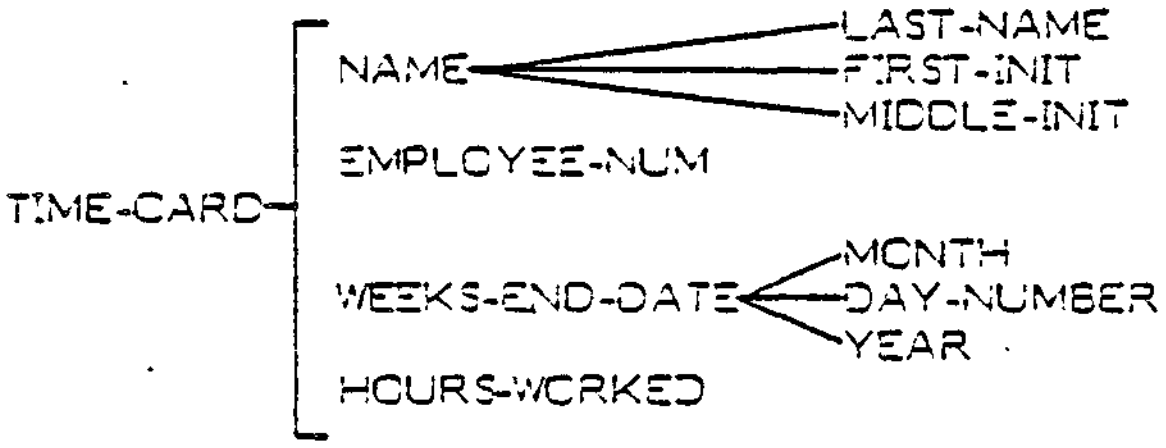
11. Optional elements may be indicated by parentheses instead of brackets, provided the lack of formality represents no substantial bar to clarity.

12. Comments, restrictions, and clarification on the use and meaning of every format are contained in the appropriate sections of this manual.

1.5 Level Numbers and Data-Names

For purposes of processing, the contents of a file are divided into logical records, with level number 01 initiating a logical record description. Subordinate data items that constitute a logical record are grouped in a hierarchy and identified with level numbers 02 to 49, not necessarily consecutive. Additionally, level number 77 identifies a "stand alone" item in Working Storage or Linkage Sections; that is, it does not have subordinate elementary items as does level 01. Level 88 is used to define condition-names and associated conditions. A level number less than 10 may be written as a single digit.

Levels allow specification of subdivisions of a record necessary for referring to data. Once a subdivision is specified, it may be further subdivided to permit more detailed data reference. This is illustrated by the following weekly timecard record, which is divided into four major items: name, employee-number, cate and hours, with more specific information appearing for name and date.



Subdivisions of a record that are not themselves further subdivided are called elementary items. Data items that contain subdivisions are known as group items. When a Procedure Division statement makes reference to a group item, the reference apolies to the area reserved for the entire group. All elementary items must be described with a PICTURE or USAGE IS INDEX clause. Consecutive logical records (01) subordinate to any given file represent implicit redefinitions of the same area whereas in the Working-Storage section, each record (01) is the definition of its own memory area.

Less inclusive groups are assigned numerically higher level numbers. Level numbers of items within groups need not be consecutive. A group whose level is k includes all groups and elementary items described under it until a level number less than or equal to k is encountered.

Separate entries are written in the source program for each level. To illustrate level numbers and group items, the weekly timecard record in the previous example may be described by Data Division entries having the following level numbers, data-names and PICTURE definitions.

```
01 TIME-CARD.
     02 NAME.
          03 LAST-NAME          PICTURE X(18).
          03 FIRST-INIT         PICTURE X.
          03 MIDDLE-INIT        PICTURE X.
     02 EMPLOYEE-NUM            PICTURE 99999.
     02 WEEKS-END-DATE.
               05 MONTH              PIC 99.
               05 DAY-NUMBER         PIC 99.
               05 YEAR               PIC 99.
     02 HOURS-WORKED            PICTURE 99V9.
```

A data-name is a word assigned by the user to identify a data item used in a program. A data-name always refers to a region of data, not to a particular value. The item referred to often assumes a number of different values during the course of a program.

A data-name must begin with an alphabetic character. A data-name or the key word FILLER must be the first word following the level number in each Record Description entry, as shown in the following general format:

level number $\left\{\begin{matrix}\text{data-name}\\ \underline{\text{FILLER}}\end{matrix}\right\}$

This data-name is the defining name of the entry and is used to refer to the associated data area (containing the value of a data item).

If some of the characters in a record are not used in the processing steps of a program, then the data description of these characters need not include a data-name. In this case, FILLER is written in lieu of a data-name after the level number.

1.6 File Names

A file is a collection of data records, such as a printed listing or a region of floppy disk, containing individual records of a similar class or application. A file-name is defined by an FD entry in the Data Division's File Section. FD is a reserved word which must be followed by a unique programmer-supplied word called the file-name. Rules for composition of the file-name word are identical to those for data-names (see Section 1.3). References to a file-name appear in the Procedure Division statements OPEN, CLOSE and READ, as well as in the Environment Division. CAUTION: File names are not to be confused with file ID's as described in Section 3.14.2.

1.7 Condition-Names

A condition-name is defined in level 88 entries within the Data Division. It is a name assigned to a specific value, set or range of values, within the complete set of values that a data item may assume. Rules for formation of name words are specified in Section 1.3. Explanations of condition-name declarations and procedural statements employing them are given in the chapters devoted to the Data and Procedure Divisions.

1.8 Mnemonic-Names

A mnemonic-name is assigned in the Environment Division for reference in ACCEPT or DISPLAY statements. It assigns a user-defined word to an implementor-chosen name, such as PRINTER. A mnemonic-name is composed according to the rules in Section 1.3.

1.9 Literals

A literal is a constant that is not identified by a data-name in a program. A literal is either non-numeric or numeric.

Non-Numeric Literals

A non-numeric literal must be bounded by matching quotation marks (or apostrophes) and may consist of any combination of characters in the ASCII set, except quotation marks (or apostrophe). All spaces enclosed by the quotation marks are included as part of the literal. A non-numeric literal must not exceed 120 characters in length.

The following are examples of non-numeric literals:

```
"ILLEGAL CONTROL CARD"

'CHARACTER-STRING'

"DO's & DON'T'S"
```

Each character of a non-numeric literal (following the introductory delimiter) may be any character other than the delimiter. That is, if the literal is bounded by apostrophes, then quotation (") marks may be included within the literal, and vice versa. Length of a non-numeric literal excludes the delimiters; minimum length is one.

A succession of two "delimiters" within a literal is interpreted as a single representation of the delimiter within the literal.

Non-numeric literals may be "continued" from one line to the next. When a non-numeric literal is of a length such that it cannot be contained on one line of a coding sheet, the following rules apply to the next line of coding (continuation line):

1. A hyphen is placed in column 7 of the continuation line.

2. A delimiter is placed in Area B preceding the continuation of the literal.

3. All spaces at the end of the previous line and any spaces following the delimiter in the continuation line and preceding the final delimiter of the literal are considered to be part of the literal.

4. On any continuation line, Area A should be blank.

Numeric Literals

A numeric literal must contain at least one and not more than 18 digits. A numeric literal may consist of the characters 0 through 9 (optionally preceded by a sign) and the decimal point. It may contain only one sign character and only one decimal point. The sign, if present, must appear as the leftmost character in the numeric literal. If a numeric literal is unsigned, it is assumed to be positive.

A decimal point may appear anywhere within the numeric literal, except as the rightmost character. If a numeric literal does not contain a decimal point, it is considered to be an integer.

The following are examples of numeric literals:

72 +1011 3.14159 -6 -.333 0.5

By use of the Environment Division specification DECIMAL-POINT IS COMMA, the functions of the characters period and comma are interchanged, putting the "European" notation into effect. In this case, the value of "pi" would be 3,1416 when written as a numeric literal.

1.10 Figurative Constants

A figurative constant is a special type of literal. It represents a value to which a standard name has been assigned. A figurative constant is not bounded by quotation marks.

ZERO may be used in many places in a program as a numeric literal. Other figurative constants are available to provide non-numeric data; the reserved words representing various characters are as follows:

SPACE	the blank character represented by "octal" 40
LOW-VALUE	the character whose "octal" representation is 00
HIGH-VALUE	the character whose "octal" representation is 177
QUOTE	the quotation mark, whose "octal" representation is 42
ALL literal	one or more instances of the literal, which must be a one-character non-numeric literal or a figurative constant, in which case ALL is redundant but serves for readability.

The plural forms of these figurative constants are acceptable to the compiler but are equivalent in effect. A figurative constant represents as many instances of the associated character as are required in the context of the statement.

A figurative constant may be used anywhere a literal is called for in a "general format" except that whenever the literal is restricted to being numeric, the only figurative constant permitted is ZERO.

1.11 Structure of a Program

Every CCBOL source program is divided into four divisions. Each division must be placed in its proper sequence, and each must begin with a division header.

The four divisions, listed in sequence, and their functions are:

IDENTIFICATION DIVISION, which names the program.

ENVIRONMENT DIVISION, which indicates the computer equipment and features to be used in the program.

DATA DIVISION, which defines the names and characteristics of data to be processed.

PROCEDURE DIVISION, which consists of statements that direct the processing of data at execution time.

It is very difficult for COBOL to compile source code if the Division headers are omitted, misspelled, or are accidentally commented out. In this case, unpredictable events may occur.

The following skeletal coding defines program component structure and order.

```
IDENTIFICATION DIVISION.

PROGRAM-ID. program-name.

[AUTHOR. comment-entry ...]

[INSTALLATION. comment-entry ...]

[DATE-WRITTEN. comment-entry ...]

[DATE-COMPILED. comment-entry ...]

[SECURITY. comment-entry ...]

ENVIRONMENT DIVISION.

[CONFIGURATION SECTION.

[SOURCE-COMPUTER. entry]

[OBJECT-COMPUTER. entry]

[SPECIAL-NAMES. entry] ]

[INPUT-OUTPUT SECTION.

FILE-CONTROL. entry ...

[I-O-CONTROL. entry ...] ]

DATA DIVISION.

[FILE SECTION.

[file description entry

record description entry ...]...]

[WORKING-STORAGE SECTION.

[data item description entry ...]...]

[LINKAGE SECTION.

[data item description entry ...]...]

[SCREEN SECTION.

[screen-description-entry ...] ...]

PROCEDURE DIVISION [USING [identifier-1] ...].

[DECLARATIVES.

[section-name SECTION. USE Sentence.

[paragraph-name. [sentence]...]...]...

END DECLARATIVES.]

[[section-name SECTION. [segment number]]

[paragraph-name. [sentence]...]...
```

1.12 Coding Rules

Since CCBOL is a subset of American National Standards Institute (ANSI) COBOL, programs may be written on standard COBOL coding sheets, and the following rules are applicable.

1. Each line of code should have a six-digit sequence number in columns 1-6, such that the punched cards are in ascending order. Blanks are also permitted in columns 1-6.

2. Reserved words for division, section, and paragraph headers must begin in Area A (columns 8-11). Procedure-names must also appear in Area A (at the point where they are defined). Level numbers may appear in Area A. Level numbers 01, 77 and level indicator "FD" must begin in Area A.

3. All other program elements should be confined to columns 12-72, governed by the other rules of statement punctuation.

4. Columns 73-80 are ignored by the compiler. Frequently, these columns are used to contain the deck identification.

5. Explanatory comments may be inserted on any line within a source program by placing an asterisk in column 7 of the line. The line will be produced on the source listing but serves no other purpose. If a slash (/) appears in column 7, the associated card is treated as comments and will be printed at the top of a new page when the compiler lists the program.

6. Any program element may be "continued" on the following line of a source program. The rules for continuation of a non-numeric ("quoted") literal are explained in Section 1.9. Any other word or literal or other program element is continued by placing a hyphen in the column 7 position of the continuation line. The effect is concatenation of successive word parts, exclusive of all trailing spaces of the last predecessor word and all leading spaces of the first successor word on the continuation line. On a continuation line, Area A must be blank.

7. Any tab characters in a line are expanded as if there were tab stops at every eighth column past column 1, except that the first tab stop is in column 7, just past the six sequence-number columns. Subsequent tab stops are columns 17, 25, 33, etc. as determined by the general rule.

1.13 Qualification of Names

When a data-name, condition-name or paragraph name is not unique, reference thereto may be accomplished uniquely by use of qualifier names. For example, if there were two or more items named YEAR, the qualified reference

```
YEAR OF HIRE-DATE
```

might differentiate between YEAR fields in HIRE-DATE and TERMINATION-DATE.

Qualifiers are preceded by the word OF or IN; successive data-name or condition-name qualifiers must designate lesser-level-numbered groups that contain all preceding names in the composite reference, i.e., HIRE-DATE must be a group item (or file-name) containing an item called YEAR. Paragraph-names may be qualified by a section-name. The maximum number of qualifiers is five. File-names and mnemonic-names must be unique.

A qualified name may only be written in the Screen Section or Procedure Division. A reference to a multiply-defined paragraph-name need not be qualified when referred to from within the same section.

1.14 COPY Statement

The COPY statement is used to logically embed the text of a disk file (other than the source file) in the source code input to the CCBCL-80 compiler. The format of the COPY statement is:

```
COPY text-name
```

where text-name is a disk file name in the format required by the operating system in use. For example, suppose BDEF.CCB is a text file containing the following source code:

```
05 B.
   10 B1 PIC X.
   10 B2 PIC X.
```

Then a source file containing

```
05 A.
   10 A1 PIC 9.
COPY BDEF.CCB
05 C.
   10 C1 PIC Z.
```

will compile exactly as if the following had been coded:

```
05 A.
   10 A1 PIC 9.
05 B.
   10 B1 PIC X.
   10 B2 PIC X.
05 C.
   10 C1 PIC Z.
```

The portion of a source line containing a COPY statement must contain only spaces from the end of text-name to the end of the line.

CHAPTER 2

Identification and Environment Divisions

2.1 Identification Division

Every CCBOL program begins with the header: IDENTIFICATION DIVISION. This division is divided into paragraphs having preassigned names:

PROGRAM-ID.	program-name.
AUTHOR.	comments.
INSTALLATION.	comments.
DATE-WRITTEN.	comments.
DATE-COMPILED.	comments.
SECURITY.	comments.

Only the PROGRAM-ID paragraph is required, and it must be the first paragraph. Program-name is any alphanumeric string of characters, the first of which must be alphabetic. Only the first 6 characters of program-name are retained by the compiler. The program-name identifies the object program and is contained in headings on compilation listings.

The contents of any other paragraphs are of no consequence, serving only as documentary remarks.

2.2 Environment Division

The Environment Division specifies a standard method of expressing those aspects of a CCBOL program that are dependent upon physical characteristics of a specific computer. It is required in every program.

The general format of the Environment Division is:

```
ENVIRONMENT DIVISION.

CONFIGURATION SECTION.

SOURCE-COMPUTER.  Computer-name  [WITH DEBUGGING MODE].

OBJECT-COMPUTER.  Computer-name
     [MEMORY SIZE integer WORDS | CHARACTERS | MODULES]
     [PROGRAM COLLATING SEQUENCE IS ASCII].

SPECIAL-NAMES.  [PRINTER IS mnemonic-name]  ASCII IS  { STANDARD-1 | NATIVE }

     [CURRENCY SIGN IS literal]

     [DECIMAL-POINT IS COMMA].

INPUT-OUTPUT SECTION.

FILE-CONTROL.  {file-control-entry}...

I-O-CONTROL.

     [SAME  [ RECORD | SORT | SORT-MERGE ]  AREA FOR file-name...]...
```

2.2.1 CONFIGURATION SECTION

The CONFIGURATION SECTION, which has three possible paragraphs, is optional. The three paragraphs are SOURCE-COMPUTER, OBJECT-COMPUTER, and SPECIAL- NAMES. The contents of the first two paragraphs are treated as commentary, except for the clause WITH DEBUGGING MODE, if present (see Section 4.22). The third paragraph, SPECIAL-NAMES, relates implementor names to user-defined names and changes default editing characters. The PRINTER IS phrase allows definition of a name to be used in the DISPLAY statement with UPON.

If the currency symbol is not desired to be the dollar sign, the user may specify a single character non-numeric literal in the CURRENCY SIGN clause. However, the designated character may not be a quote mark, nor any of the characters defined for PICTURE representations, nor digits (0-9).

The "European" convention of separating integer and fraction positions of numbers with the comma character is specified by employment of the clause DECIMAL-POINT IS COMMA.

Note that the reserved word IS is required in entries for currency sign definition and decimal-point convention specification.

The entry ASCII IS NATIVE/STANDARD-1 specifies that data representation adheres to the American Standard code for Information Interchange. However, this convention is assumed even if the ASCII-entry is not specifically present. In this compiler, NATIVE and STANDARD-1 are identical, and refer to the character set representation specified in Appendix IV.

2.2.2 INPUT-OUTPUT SECTION

The second section of the Environment Division is mandatory unless the program has no data files; it begins with the header:

INPUT-OUTPUT SECTION.

This section has two paragraphs: FILE-CONTROL and I-O-CONTROL. In this section, the programmer defines the file assignment parameters, including specification of buffering.

2.2.2.1 FILE-CONTROL ENTRY (SELECT ENTRY)

For each file having records described in the Data Division's File Section, a Select Sentence-Entry (beginning with the reserved word SELECT) is required in the FILE-CONTROL paragraph. The format of a Select Sentence-Entry for a sequential file is:

SELECT file-name ASSIGN TO DISK | PRINTER

[RESERVE integer AREAS | AREA]

[FILE STATUS IS data-name-1]

[ACCESS MODE IS SEQUENTIAL] [ORGANIZATION IS [LINE] SEQUENTIAL].

The SELECT entry must begin to the right of Area A of the source line. All phrases after "SELECT filename" can be in any order. Both the ACCESS and ORGANIZATION clauses are optional for regular sequential input-output processing. For Indexed or Relative files, alternate formats are available for this section, and are explained in the chapters on Indexed and Relative files.

Two formats are available for sequential disk files. One is the regular form which is requested by ORGANIZATION IS SEQUENTIAL, and the other is requested by ORGANIZATION IS LINE SEQUENTIAL. Both forms assume the records in the file are variable-length. The regular sequential organization is that of a two-byte count of the record length followed by the actual record, for as many records as exist in the file. The line sequential organization has the record followed by a carriage return/line feed delimiter, for as many records as exist in the file. No COMP or COMP-3 information should be written into a Line Sequential file because these data items may contain the same binary codes used for carriage return and line feed which therefore would cause a problem when subsequently reading the file. Both organizations pad any remaining space of the last physical block with Control-Z characters, indicating end-of-file. All records are placed in the file with no gaps; they span physical block boundaries.

The RESERVE clause is not functional in CCBOL- but is scanned for correct syntax. One physical block buffer is always allocated to the logical record area assigned to it. This allows logical records to be spanned over physical block boundaries. For files assigned to PRINTER, the logical record area is used as the physical buffer as well.

In the FILE STATUS entry, data-name-1 must refer to a two-character Working-Storage or Linkage Section item of category alphanumeric into which the run-time data management facility places status information after an I-O statement. The left-hand character of data-name-1 assumes the values:

'0' for successful completion
'1' for end-of-file
'2' for invalid key (only for indexed and relative files)
'3' for a non-recoverable I-O error

The right-hand character of data-name-1 is set to '0' if no further status information exists for the previous I-O operation. The following combinations of values are possible:

File Status Left	File Status Right	Meaning
'0'	'0'	O.K.
'1'	'0'	EOF
'3'	'0'	Permanent error
'3'	'4'	Disk space full
'9'	'1'	File damaged

In an OPEN INPUT or OPEN I-O statement, a File Status of '30' means 'File Not Found.'

For values of status-right when status-left has a value of '2', see the chapters on Indexed or Relative files.

2.2.2.2 I-O-CONTROL PARAGRAPH

The SAME AREA clause is optional. Only the SAME RECORD AREA form is functional in CCBOL The other forms are checked for correct syntax but do not cause any sharing of physical buffer space.

The SAME RECORD AREA form causes all the named files to share the same logical record area in order to conserve memory space.

The format of the SAME AREA entry is:

SAME [RECORD / SORT / SORT-MERGE] AREA FOR filename...

All files named in a given SAME AREA clause need not have the same organization or access. However, no file may be listed in more than one SAME AREA clause.

The SORT and SORT-MERGE options are allowed only in those versions of CCBOL supporting the SORT facility.

CHAPTER 3

Data Division

The Data Division, which is one of the required divisions in a program, is subdivided into four sections: File Section, Working-Storage Section, Linkage Section, and Screen Section. Each is discussed in Sections 3.13-3.16, but first, aspects of data specification that apply in all sections will be described.

3.1 Data Items

Several types of data items can be described in COBOL programs. These data items are described in the following paragraphs.

3.1.1 Group Items

A group item is defined as one having further subdivisions, so that it contains one or more elementary items. In addition, a group item may contain other groups. An item is a group item if, and only if, its level number is less than the level number of the immediately succeeding item. If an item is not a group item, then it is an elementary item. Ordinarily, the maximum size of any data item is 4095 bytes. In order to allow tables to exceed this limit, however, level 01 group items are not checked for length. Such an item longer than 4095 bytes will be disallowed by the compiler as an operand of a Procedure Division statement such as MOVE, INSPECT, etc.

3.1.2 Elementary Items

An elementary item is a data item containing no subordinate items.

Alphanumeric Item: An alphanumeric item consists of any combination of characters, making a "character string" data field. If the associated picture contains "editing" characters, it is an alphanumeric edited item.

Report (Edited) Item: A report item is an edited "numeric" item containing only digits and/or special editing characters. It must not exceed 30 characters in length. A report item can be used only as a receiving field for numeric data. It is designed to receive a numeric item but cannot be used as a numeric item itself.

3.1.3 Numeric Items

Numeric items are elementary items intended to contain numeric data only.

External Decimal Item: An external decimal data item is an item in which one character (byte) is employed to represent one digit. A maximum number of 18 digits is permitted; the exact number of digit positions is defined by writing a specific number of 9-characters in the PICTURE description. For example, PICTURE 999 defines a 3-digit item. That is, the maximum decimal value of the item is nine hundred ninety-nine.

If the PICTURE begins with the letter S, then the item also has the capability of containing an "operational sign." An operational sign does not occupy a separate character (byte), unless the "SEPARATE" form of SIGN clause is included in the item's description. Regardless of the form of representation of an operational sign, its purpose is to provide a sign that functions in the normal algebraic manner.

The USAGE of an external decimal item is DISPLAY (see USAGE clause, Section 3.4).

Internal Decimal Item: An internal decimal item is stored in packed decimal format. It is attained by inclusion of the COMPUTATIONAL-3 USAGE clause.

A packed decimal item defined by n 9's in its PICTURE occupies 1/2 of (n + 2) (rounded down) bytes in memory. All bytes except the rightmost contain a pair of digits, and each digit is represented by the binary equivalent of a valid digit value from 0 to 9. The item's low order digit and the operational sign are found in the rightmost byte of a packed item. For this reason, the compiler considers a packed item to have an arithmetic sign, even if the original PICTURE lacked an S-character.

Binary Item: A binary item uses the base 2 system to represent an integer in the range -32768 to 32767. It occupies one 16-bit word. The leftmost bit of the reserved area is the operational sign. A binary item is specified by USAGE IS COMPUTATIONAL.

Index-Data-Item: An index-data-item has no PICTURE; it is defined by the USAGE IS INDEX clause. (Refer to Chapter 6, "Table Handling by the Indexing Method.")

3.2 DATA DESCRIPTION ENTRY

A Data Description entry specifies the characteristics of each field (item) in a data record. Each item must be described in a separate entry in the same order in which the items appear in the record. Each Data Description entry consists of a level number, a data-name, and a series of independent clauses followed by a period. The general format of a Data Description entry is:

level-number { data-name / FILLER } (REDEFINES-clause) (JUSTIFIED-clause)

(PICTURE-clause) (USAGE-clause) (SYNCHRONIZED-clause)

(OCCURS-clause) (BLANK-clause) (VALUE-clause) (SIGN-clause).

When this format is applied to specific items of data, it is limited by the nature of the data being described. The format allowed for the description of each data type appears below. Clauses that are not shown in a format are specifically forbidden in that format. Clauses that are mandatory in the description of certain data items are shown without parentheses. The clauses may appear in any order except that a REDEFINES-clause, if used, should come first.

Group Item Format

level-number { data-name / FILLER } (REDEFINES-clause) (USAGE-clause)

(OCCURS-clause) (VALUE clause) (SIGN-clause).

Example:

```
01 GROUP-NAME.
       02 FIELD-B PICTURE X.
       02 FIELD-C PICTURE X.
```

> NOTE
>
> The USAGE clause may be written at a group level to avoid repetitious writing of it at the subordinate element level.

3.3 FORMATS FOR ELEMENTARY ITEMS

ALPHANUMERIC ITEM (also called a character-string item)

level-number { data-name / FILLER } (REDEFINES-clause) (OCCURS-clause)

PICTURE IS an-form (USAGE IS DISPLAY) (JUSTIFIED-clause)

(VALUE IS non-numeric-literal) (SYNCHRONIZED-clause).

Examples:

```
02 MISC-1 PIC X(53).
02 MISC-2 PICTURE BXXXBXXB.
```

REPORT ITEM (also called a numeric-edited item)

level-number { data-name / FILLER } (REDEFINES-clause) (OCCURS-clause)

PICTURE IS report-form (BLANK WHEN ZERO) (USAGE IS DISPLAY)

(VALUE IS non-numeric-literal) (SYNCHRONIZED-clause).

Example:

```
02 XTOTAL PICTURE $999,999.99-.
```

DECIMAL ITEM

level-number { data-name / FILLER } (REDEFINES-clause) (OCCURS-clause)

PICTURE IS numeric-form (SIGN-clause)

(USAGE-clause) (VALUE IS numeric-literal) (SYNCHRONIZED-clause).

Examples:

```
02 HOURS-WORKED PICTURE 99V9, USAGE IS DISPLAY.
02 HOURS-SCHEDULED PIC S99V9, SIGN IS TRAILING.
- - - - - - - - - - - - - - - - - - - - - -
11 TAX-RATE PIC S99V999 VALUE 1.375, COMPUTATIONAL-3.
```

BINARY ITEM

level-number { data-name / FILLER } (REDEFINES-clause) (OCCURS-clause)

PICTURE IS numeric-form

USAGE IS COMPUTATIONAL | COMP | INDEX

(VALUE IS numeric-literal) (SYNCHRONIZED-clause).

> NOTE
>
> A PICTURE or VALUE must not be given for an INDEX Data Item.

Examples:

```
02 SUBSCRIPT PICTURE 999 COMP, VALUE ZERO.
02 YEAR-TO-DATE PIC S9(5) COMPUTATIONAL.
```

3.4 USAGE CLAUSE

The USAGE clause specifies the form in which data is represented.

The USAGE clause may be written at any level. If USAGE is not specified, the item is assumed to be in "DISPLAY" mode. The general format of the USAGE clause is:

USAGE IS { COMPUTATIONAL / INDEX / DISPLAY / COMPUTATIONAL-3 }

INDEX is explained in Chapter 6, Table Handling. COMPUTATIONAL, which may be abbreviated COMP, usage defines an integer binary field. COMPUTATIONAL-3,. which may be abbreviated COMP-3, defines a packed (internal decimal) field.

If a USAGE clause is given at a group level, it applies to each elementary item in the group. The USAGE clause for an elementary item must not contradict the USAGE clause of a group to which the item belongs.

3.5 PICTURE CLAUSE

The PICTURE clause specifies a detailed description of an elementary level data item and may include specification of special report editing. The reserved word PICTURE may be abbreviated PIC.

The general format of the PICTURE clause is:

PICTURE IS { an-form / numeric-form / report-form }

There are three possible types of pictures: An-form, Numeric-form and Report-form.

An-Form Option: This option applies to alphanumeric (character string) items. The PICTURE of an alphanumeric item is a combination of data description characters X, A or 9 and editing characters B, 0 and /. An X indicates that the character position may contain any character from the computer's ASCII character set. A PICTURE that contains at least one of the combinations:

(a) A and 9, or
(b) X and 9, or
(c) X and A

in any order is considered as if every 9, A or X character were X. The characters B, 0 and / may be used to insert blanks or zeros or slashes in the item. This is then called an alphanumeric-edited item.

If the string has only A's and B's, it is considered alphabetic; if it has only 9's, it is numeric (see below).

Numeric-Form Option: The PICTURE of a numeric item may contain a valid combination of the following characters:

9 The character 9 indicates that a digit position which must contain a numeric character. The maximum number of 9's in a PICTURE is 18.

V The optional character V indicates the position of an assumed decimal point. Since a numeric item cannot contain an actual decimal point, an assumed decimal point is used to provide the compiler with information concerning the scaling alignment of items involved in computations. Storage is never reserved for the character V. Only one V is permitted in any single PICTURE.

S The optional character S indicates that the item has an operational sign. It must be the first character of the PICTURE. See also, SIGN clause, Section 3.12.

P The character P indicates an assumed decimal scaling position. It is used to specify the location of an assumed decimal point when the point is not within the number that appears in the data item. The scaling position character P is not counted in the size of the data item; that is, memory is not reserved for these positions. However, scaling position characters are counted in determining the maximum number of digit positions (18) in numeric edited items or in items that appear as operands in arithmetic statements. The scaling position character P may appear only to the left or right of the other characters in the string as a continuous string of P's within a PICTURE description. The sign character S and the assumed decimal point V are the only characters which may appear to the left of a leftmost string of P's. Since the scaling position character P implies an assumed decimal point (to the left of the P's if the P's are leftmost PICTURE characters and to the right of the P's if the P's are rightmost PICTURE characters), the assumed decimal point symbol V is redundant as either the leftmost or rightmost character within such a PICTURE description.

Report-Form Option: This option describes a data item suitable as an "edited" receiving field for presentation of a numeric value. The editing characters that may be combined to describe a report item are as follows:

9 V . Z CR DB , $ + * B 0 - P /

The characters 9, P and V have the same meaning as for a numeric item. The meanings of the other allowable editing characters are described as follows:

. The decimal point character specifies that an actual decimal point is to be inserted in the indicated position and the source item is to be aligned accordingly. Numeric character positions to the right of an actual decimal point in a PICTURE must consist of characters of one type. The decimal point character must not be the last character in the PICTURE character string. PICTURE character 'P' may not be used if '.' is used.

Z The characters Z and * are called * replacement characters. Each one represents a digit position. During execution, leading zeros to be placed in positions defined by Z or * are suppressed, becoming blank or *. Zero suppression terminates upon encountering the decimal point (. or V) or a non-zero digit. All digit positions to be modified must be the same (either Z or *), and contiguous starting from the left. Z or * may appear to the right of an actual decimal point only if <u>all</u> digit positions are the same.

CR DB CR and DB are called credit and debit symbols and may appear only at the right end of a PICTURE. These symbols occupy two character positions and indicate that the specified symbol is to appear in the indicated positions if the value of a source item is negative. If the value is positive or zero, spaces will appear instead. The PICTURE, CR, DB, +, and - symbols are mutually exclusive.

, The comma specifies insertion of a comma between digits. Each insertion character is counted in the size of the data item, but does not represent a digit position. The comma may also appear in conjunction with a floating string, as described below. It must not be the last character in the PICTURE character string.

A floating string is defined as a leading, continuous series of one of either $ or - or +, optionally interrupted by one or more insertion commas and/or decimal points. For example:

```
$$,$$$,$$$
++++
-,---,--
+(8).++
$$,$$$.$$
```

A floating string containing N + 1 occurrences of $ or + or - defines N digit positions. When moving a numeric value into a report item, the appropriate character floats from left to right, so that the developed report item has exactly one actual $ or + or - immediately to the left of the most significant nonzero digit, in one of the positions indicated by $ or + or in the PICTURE. Blanks are placed in all character positions to the left of the single developed $ or + or -. If the most significant digit appears in a position to the right of positions defined by the floating string, then the developed item contains $ or + or in the rightmost position of the floating string, and non-significant zeros may follow. The presence of an actual or implied decimal point in a floating string is treated as if all digit positions to the right of the point were indicated by the PICTURE character 9. In the following examples, b represents a blank in the developed items.

PICTURE	Numeric Value	Developed Item
$$$999	14	bb$014
$$$$$$	14	bbb$14
--,---,999	-456	bbbbbb-456

A floating string need not constitute the entire PICTURE of a report item, as shown in the preceding examples. Restrictions on characters that may follow a floating string are given later in the description.

When a comma appears to the right of a floating string, the string character floats through the comma in order to be as close to the leading digit as possible.

+ - The character + or - may appear in a PICTURE either singly or in a floating string. As a fixed sign control character, the + or - must appear as the last symbol in the PICTURE. The plus sign indicates that the sign of the item is indicated by either a plus or minus sign placed in the character position, depending on the algebraic sign of the numeric value placed in the report field. The minus sign indicates that blank or minus is placed in the character position, depending on whether the algebraic sign of the numeric value placed in the report field is positive or negative, respectively.

B Each appearance of B in a PICTURE represents a blank in the final edited value.

/ Each slash in a PICTURE represents a slash in the final edited value.

0 Each appearance of 0 in a PICTURE represents a position in the final edited value where the digit zero will appear.

Other rules for a report (edited) item PICTURE are:

1. The appearance of one type of floating string precludes any other floating string.

2. There must be at least one digit position character.

3. The appearance of a floating sign string or fixed plus or minus insertion character precludes the appearance of any other of the sign control insertion characters, namely +, -, CR, DB.

4. The characters to the right of a decimal point up to the end of a PICTURE, excluding the fixed insertion characters +, -, CR, DB (if present), are subject to the following restrictions:

   a. Only one type of digit position character may appear. That is, Z, *, 9, and floating-string digit position characters $, +, -, are all mutually exclusive.

   b. If one of the numeric character positions to the right of a decimal point is represented by + or - or $ or Z, then all the numeric character positions in the PICTURE must be represented by the same character.

5. The PICTURE character 9 can never appear to the left of a floating string, or replacement character.

Additional notes on the PICTURE Clause:

1. A PICTURE clause must only be used at the elementary level.

2. An integer enclosed in parentheses and following X 9 $ Z P * B - or + indicates the number of consecutive occurrences of the PICTURE character.

3. Characters V and P are not counted in the space allocation of a data item. CR and DB occupy two character positions each.

4. A maximum of 30 character positions is allowed in a PICTURE character string. For example, PICTURE X(89) consists of five PICTURE characters.

5. A PICTURE must contain at least one of the characters A, Z, *, X, or 9, or at least two consecutive appearances of the + or - or $ characters.

6. The characters ., S, V, CR, and DB can appear only once in a PICTURE.

7. When DECIMAL-POINT IS COMMA is specified, the explanations for period and comma are understood to apply to comma and period, respectively.

The examples below illustrate the use of PICTURE to edit data. In each example, a movement of data is implied, as indicated by the column headings. (Data value shows contents in storage; scale factor of this source data area is given by the PICTURE.)

Source Area		Receiving Area	
PICTURE	Data Value	PICTURE	Edited Data
9(5)	12345	$$$,$$9.99	$12,345.00
9(5)	00123	$$$,$$9.99	$123.00
9(5)	00000	$$$,$$9.99	$0.00
9(4)V9	12345	$$$,$$9.99	$1,234.50
V9(5)	12345	$$$,$$9.99	$0.12
S9(5)	00123	-------.99	123.00
S9(5)	-00001	-------.99	-1.00
S9(5)	00123	+++++++.99	+123.00
S9(5)	00001	-------.99	1.00
9(5)	00123	+++++++.99	+123.00
9(5)	00123	-------.99	123.00
S9(5)	12345	+++++++.99CR	++12345.00
S999V99	02345	ZZZVZZ	2345
S999V99	00004	ZZZVZZ	04

3.6 VALUE CLAUSE

The VALUE clause specifies the initial value of working-storage items. The format of this clause is:

VALUE IS literal

The VALUE clause must not be written in a Data Description entry that also has an OCCURS or REDEFINES clause, or in an entry that is subordinate to an entry containing an OCCURS or REDEFINES clause. Furthermore, it cannot be used in the File or Linkage Sections, except in level 88 condition descriptions.

The size of a literal given in a VALUE clause must be less than or equal to the size of the item as given in the PICTURE clause. The positioning of the literal within a data area is the same as would result from specifying a MOVE of the literal to the data area, except that editing characters in the PICTURE have no effect on the initialization, nor do BLANK WHEN ZERO or JUSTIFIED clauses. The type of literal written in a VALUE clause depends on the type of data item, as specified in the data item formats earlier in this text. For edited items, values must be specified as non-numeric literals, and must be presented in edited form. A figurative constant may be given as the literal.

When an initial value is not specified, no assumption should be made regarding the initial contents of an item in Working-Storage.

The VALUE clause may be specified at the group level, in the form of a correctly sized non-numeric literal, or a figurative constant. In these cases the VALUE clause cannot be stated at the subordinate levels with the group. However, the value clause should not be written for a group containing items with descriptions including JUSTIFIED, SYNCHRONIZED and USAGE (other than USAGE IS DISPLAY). (A form used in level 88 items is explained in Section 3.16)

3.7 REDEFINES CLAUSE

The REDEFINES clause specifies that the same area is to contain different data items, or provides an alternative grouping or description of the same data. The format of the REDEFINES clause is:

REDEFINES data-name-2

When written, the REDEFINES clause should be the first clause following the data-name that defines the entry. The data description entry for data-name-2 should not contain a REDEFINES clause, nor an OCCURS clause.

When an area is redefined, all descriptions of the area remain in effect. Thus, if B and C are two separate items that share the same storage area due to redefinition, the procedure statements MOVE X TO B or MOVE Y TO C could be executed at any point in the program. In the first case, B would assume the value of X and take the form specified by the description of B. In the second case, the same physical area would receive Y according to the description of C.

For purposes of discussion of redefinition, data-name-1 is termed the subject, and data-name-2 is called the object. The levels of the subject and object are denoted by s and t, respectively. The following rules must be obeyed in order to establish a proper redefinition.

1. s must equal t, but must not equal 88.

2. The object must be contained in the same record (01 group level item), unless s=t=01.

3. Prior to definition of the subject and subsequent to definition of the object there can be no level numbers that are numerically less than s.

The length of data-name-1, multiplied by the number of occurrences of data-name-1, may not exceed the length of data-name-2, unless the level of data-name-1 is 01 (permitted only outside the File Section). Data-name-1 and entries subordinate to data-name-1 must not contain any value clauses, except in level 88. In the File Section, multiple level 01 entries subordinate to any given FD represent implicit redefinitions of the same area.

3.8 OCCURS CLAUSE

The OCCURS clause is used in defining related sets of repeated data, such as tables, lists and arrays. It specifies the number of times, up to a maximum of 1023, that a data item with the same format is repeated. Data Description clauses associated with an item whose description includes an OCCURS clause apply to each repetition of the item being described. When the OCCURS clause is used, the data name that is the defining name of the entry must be subscripted or indexed whenever it appears in the Procedure Division. If this data-name is the name of a group item, then all data-names belonging to the group must be subscripted or indexed whenever they are used.

The OCCURS clause must not be used in any Data Description entry having a level number 01 or 77. The OCCURS clause has the following format:

OCCURS integer TIMES [INDEXED BY index-name...]

Since the OCCURS clause can only be used at subordinate levels within a data record, the maximum size of a table is limited by the rules for the size of a group item. See Section 3.1.1 on "Group Items".

Subscripting: Subscripting provides the facility for referring to data items in a table or list that have not been assigned individual data-names. Subscripting is determined by the appearance of an OCCURS clause in a data description. If an item has an OCCURS clause or belongs to a group having an OCCURS clause, it must be subscripted or indexed whenever it is used. See the chapter on Table Handling for explanations on Indexing and Index Usage. (Exception: the table-name in a SEARCH statement must be referenced without subscripts.)

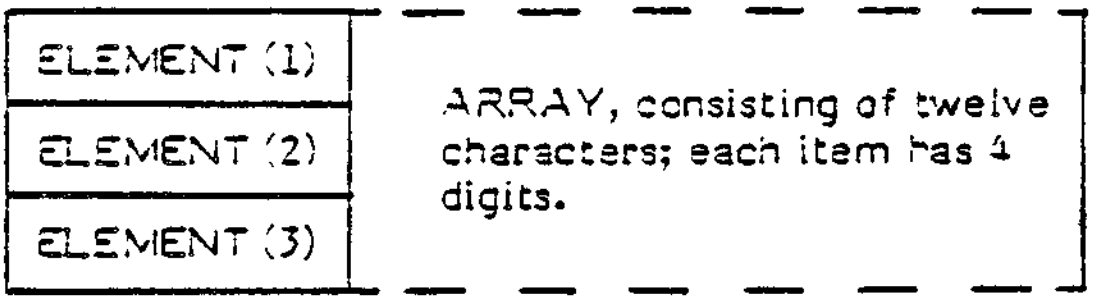
A subscript is a positive nonzero integer whose value determines an element to which a reference is being made within a table or list. The subscript may be represented either by a literal or a data-name that has an integer value. Whether the subscript is represented by a literal or a data-name, the subscript is enclosed in parentheses and appears after the terminal space of the name of the element. A subscript must be a decimal or binary item. (The latter is strongly recommended, for the sake of efficiency.)

At most three OCCURS clauses may govern any data item. Consequently, one, two or three subscripts may be required. When more than one subscript is required, they are written in the order of successively less inclusive dimensions of the data organization. Multiple subscripts are separated by commas, viz. ITEM (I, J).

Example:

```
01 ARRAY.
     03 ELEMENT, OCCURS 3, PICTURE 9(4).
```

The above example would be allocated storage as shown below.



A data-name may not be subscripted if it is being used for:

1. a subscript
2. the defining name of a data description entry
3. data-name-2 in a REDEFINES clause
4. a qualifier

3.9 SYNCHRONIZED CLAUSE

The SYNCHRONIZED clause was designed in order to allocate space for data in an efficient manner, with respect to the computer central "memory." However, in this compiler, the SYNCHRONIZED specification is treated as commentary only.

The format of this clause is:

SYNC | SYNCHRONIZED [LEFT | RIGHT]

3.10 BLANK WHEN ZERO CLAUSE

The BLANK WHEN ZERO clause specifies that a report (edited) field is to contain nothing except blanks if the numeric value moved to it has a value of zero. When this clause is used with a numeric picture, the field is consicered a report field.

3.11 JUSTIFIED CLAUSE

The JUSTIFIED RIGHT clause is only applicable to unedited alphanumeric (character string) items. It signifies that values are stored in a right-to-left fashion, resulting in space fill on the left when a short field is moved to a longer JUSTIFIED field, or in truncation on the left when a long field is moved to a shorter JUSTIFIED field. The JUSTIFIED clause is effective only when the associated field is employed as the "receiving" field in a MOVE statement.

The word JUST is a permissible abbreviation of JUSTIFIED.

3.12 SIGN CLAUSE

For an external decimal item, there are four possible manners of representing an operational sign; the choice is controlled by inclusion of a particular form of the SIGN clause, whose general form is:

[SIGN IS] { TRAILING / LEADING } [SEPARATE CHARACTER]

The following chart summarizes the effect of four possible forms of this clause.

SIGN Clause	Sign Representation
TRAILING	Embedded in rightmost byte
LEADING	Embedded in leftmost byte
TRAILING SEPARATE	Stored in separate rightmost byte
LEADING SEPARATE	Stored in separate leftmost byte

When the above forms are written, the PICTURE must begin with S. If no S appears, the item is not signed (and is capable of storing only absolute values), and the SIGN clause is prohibited. When S appears at the front of a PICTURE but no SIGN clause is included in an item's description, the "default" case SIGN IS TRAILING is assumed.

The SIGN clause may be written at a group level; in this case the clause specifies the sign's format on any signed subordinate external decimal item. The SEPARATE CHARACTER phrase increases the size of the data item by 1 character. The entries to which the SIGN clause apply must be implicitly or explicitly described as USAGE IS DISPLAY.

(Note: When the CODE-SET clause is specified for a file, all signed numeric data for that file must be described with the SIGN IS SEPARATE clause.)

3.13 LEVEL 88 CONDITION-NAMES

The level 88 condition-name entry specifies a value, list of values, or a range of values that an elementary item may assume, in which case the named condition is true, otherwise false. The format of a level 88 item's value clause is

VALUE IS literal-1 [literal-2...]

VALUES ARE literal-1 THRU literal-2

A level 88 entry must be preceded either by another level 88 entry (in the case of several consecutive condition-names pertaining to an elementary item) or by an elementary item (which may be FILLER). Index data items should not be followed by level 88 items.

Every condition-name pertains to an elementary item in such a way that the condition-name may be qualified by the name of the elementary item and the elementary item's qualifiers. A condition-name is used in the Procedure Division in place of a simple relational condition. A condition-name may pertain to an elementary item (a conditional variable) requiring subscripts. In this case, the condition-name, when written in the Procedure Division, must be subscripted according to the same requirements as the associated elementary item. The type of literal in a condition-name entry must be consistent with the data type of the conditional variable. In the following example, PAYROLL-PERIOD is the conditional variable. The picture associated with it limits the value of the 88 condition-name to one digit.

```
02 PAYROLL-PERIOD PICTURE IS 9.
     88 WEEKLY VALUE IS 1.
     88 SEMI-MONTHLY VALUE IS 2.
     88 MONTHLY VALUE IS 3.
```

Using the above description, the following procedural condition-name test may be written:

```
IF MONTHLY GO TO DO-MONTHLY
```

An equivalent statement is:

```
IF PAYROLL-PERIOD = 3 GO TO DO-MONTHLY.
```

For an edited elementary item, values in a condition-name entry must be expressed in the form of non-numeric literals.

A VALUE clause may not contain both a series of literals and a range of literals.

3.14 FILE SECTION, FD ENTRIES (SEQUENTIAL I-O ONLY)

In the FILE SECTION of the Data Division, an FD entry (file description) must appear for every SELECTed file. This entry precedes the descriptions of the file's record structure(s).

The general format of an FD entry is:

```
FD file name LABEL-clause [VALUE-OF-clause]
   [DATA-RECORD(S)-clause] [BLOCK-clause] [RECORD-clause]
      [CODE-SET-clause] [LINAGE clause].
```

After "FD filename," the order of the clauses is immaterial.

3.14.1 LABEL CLAUSE

The format of this required FD entry clause is:

```
LABEL  { RECORD  }  [ IS  ]  { OMITTED  }
       { RECORDS }  [ ARE ]  { STANDARD }
```

The OMITTED option specifies that no labels exist for the file; this must be specified for files assigned to PRINTER.

The STANDARD option specifies that labels exist for the file and that the labels conform to system specifications; this must be specified for files assigned to DISK.

3.14.2 VALUE OF CLAUSE

The VALUE OF clause appears in any FD entry for a DISK-assigned file, and contains a filename expressed as a non-numeric literal of at most 16 characters or as a data-name. The filename is specified according to the rules for filenames of the operating system being used. It must not contain any embedded space characters. If a data-name is specified, the filename it contains may be as many characters as desired, but it must be terminated by a space character. The general form is:

VALUE OF FILE-ID IS { data-name / literal }

Examples:

```
VALUE OF FILE-ID  "A:MASTER.ASM"   (CP/M)
VALUE OF FILE-ID  "EMPLOY/DAT:2"   (TRSDOS Model II)
VALUE OF FILE-ID  ":F1:INVNT.LST"  (ISIS-II)
```

A reminder: if a file is ASSIGNed to PRINTER, it is unlabeled and the VALUE clause must not be included in the associated FD. If a file is ASSIGNed to DISK, it is necessary to include both LABEL RECORDS STANDARD and VALUE clauses in the associated FD. See the CCBOL-80 User's Guide for filename formats for specific operating systems.

3.14.3 DATA RECORD(S) CLAUSE

The optional DATA RECORDS clause identifies the records in the file by name. This clause is documentary only, in this and all COBOL systems. Its general format is:

DATA { RECORD IS / RECORDS ARE } data-name-1 [data-name-2...]

The presence of more than one data-name indicates that the file contains more than one type of data record. That is, two or more record descriptions may apply to the same storage area. The order in which the data-names are listed is not significant.

Data-name-1, data-name-2, etc., are the names of data records, and each must be preceded in its record description entry by the level number 01, following the appropriate file description (FD) in the File Section.

3.14.6 CODE-SET CLAUSE

The format of this clause is:

CODE-SET IS ASCII

The CODE-SET clause, which should be specified only for non-mass-storage files, serves only the purposes of documentation in this compiler, reflecting the fact that both internal and external data are represented in ASCII code. However, any signed numeric data description entries in the file's records should include the SIGN IS SEPARATE clause and all data in the file should have USAGE DISPLAY.

3.14.7 LINAGE CLAUSE

For a file assigned to PRINTER, the LINAGE clause provides a means of specifying the size of the printable portion of a page, called the "page body." The number of lines in the page body is specified along with, optionally, the size of the top and bottom margins and the line number within the page body at which a footing area begins. The general format is:

LINAGE IS { data-name-1 / integer-1 } LINES, [WITH FOOTING AT { data-name-2 / integer-2 }]

[LINES AT TOP { data-name-3 / integer-3 }] [LINES AT BOTTOM { data-name-4 / integer-4 }]

All data-names must refer to unsigned numeric integer data items. Integer-1 must be greater than zero, and integer-2 must not be greater than integer-1.

The total page size is the sum of the values in each phrase except for FOOTING. If TOP or BOTTOM margins are not specified, their size is assumed zero. The footing area comprises that part of the page body between the line indicated by the FOOTING value, and the last line of the page body, inclusive.

The values in each phrase at the time the file is opened (by the execution of an OPEN OUTPUT statement) specify the number of lines that comprise each of the sections of the first logical page. Whenever a WRITE statement with the ADVANCING PAGE phrase is executed or a "page overflow" condition occurs (see the WRITE statement), the values in each phrase, at that time, will be used to specify the number of lines in each section of the next logical page.

A LINAGE-COUNTER is created by the presence of a LINAGE clause. The value in the LINAGE-COUNTER at any given time represents the line number at which the printer is positioned within the current page body. LINAGE-COUNTER may be referenced but may not be modified by Procedure Division statements. It is automatically modified during execution of a WRITE statement, according to the following rules:

1. When the "ADVANCING PAGE" phrase of the WRITE statement is specified or a "page overflow" condition occurs (see the WRITE statement), the LINAGE COUNTER is reset to one.

2. When the "ADVANCING identifier or integer" phrase is specified, LINAGE-COUNTER is incremented by the ADVANCING value.

3. When the ADVANCING phrase is not specified, LINAGE-COUNTER is incremented by one.

See the description of the WRITE statement for additional information about the effects of LINAGE specifications.

3.15 WORKING-STORAGE SECTION

The second section of the DATA DIVISION begins with the following header:

WORKING-STORAGE SECTION.

This section describes records and other data which are not part of external data files but which are developed and processed internally.

Data description entries in this section may employ level numbers 01-49, as in the File Section, as well as 77. Value clauses, prohibited in the File Section (except for level 88), are permitted throughout the Working-Storage Section.

3.16 LINKAGE SECTION

The third section of the Data Division is defined by the header:

LINKAGE SECTION.

In this section, the user describes data by name and attribute, but storage space is not allocated. Instead, these "dummy" descriptions are applied (through the mechanism of the USING list on the Procedure Division header) to data whose addresses are passed into a subprogram by a call upon it from a separately compiled program. Consequently, VALUE clauses are prohibited in the Linkage Section, except in level 88 condition-name entries. Refer to Chapter 5, Inter-Program Communication, for further information.

3.17 SCREEN SECTION

The fourth section of the Data Division is used to define CRT screen formats and is composed of screen data description entries. As in the File and Working-Storage sections, descriptions may be grouped through the assignment of appropriate level numbers. Thus there are two types of screen items. Elementary screen items define the individual display and/or data entry fields within the screen layout. Group screen items are used to name any group of elementary screen items so that they may be ACCEPTed or DISPLAYed with a single Procedure Division statement. The format of a group screen description entry is:

level-number screen-name [AUTO] [SECURE].

level number must be an integer in the range 01 through 49. screen-name must conform to the rules for the formation of names given in section 1.3. The group screen description entry must be followed by one or more subordinate screen items as indicated by increasing level-numbers. If AUTO or SECURE is coded for a group screen item, the effect is as if AUTO or SECURE had been coded for every elementary screen item subordinate to that group screen item.

The format of an elementary screen item is:

```
level-number [screen-name]
      [BLANK SCREEN]
      [LINE NUMBER is [PLUS] integer-1]
      [COLUMN NUMBER IS [PLUS] integer-2]
      [BLANK LINE]
      [BELL]
      [ { HIGH-LIGHT } ]
      [ { BLINK     } ]

      [[VALUE] IS literal-1]

      [ { PICTURE } IS picture-string  { [FROM { literal-2   } ]      [TO identifier-2] } ]
      [ { PIC     }                    {      { identifier-1 }        }                 }
                                       {  [USING identifier-3]                          }

            [BLANK WHEN ZERO]
            { JUSTIFIED } RIGHT]
            { JUST      }
            [AUTO]
            [SECURE]
```

level-number and screen-name are subject to the same rules as in the group screen data description. The order of clauses in the elementary screen data description entry is not significant, except that screen-name, if present, must immediately follow level-number. If PICTURE is coded, then either USING or at least one of FROM and TO must be present. A screen item may have both a FROM and TO clause. AUTO, SECURE, BLANK WHEN ZERO, and JUSTIFIED may be given only if PICTURE is specified. The maximum length of an elementary screen item is 80 characters.

The clauses specified with each elementary screen data description can affect data input and data display operations when ACCEPT and DISPLAY statements are executed at runtime. The effects of each specification are as follows:

1. BLANK SCREEN causes the entire screen to be erased and the cursor to be placed at the home position (line 1, column 1).

2. LINE and COLUMN affect the screen location associated with an elementary screen item. As the SCREEN SECTION is processed at compile time, a current cursor position is maintained so that each elementary screen item can be identified with a particular region of the screen. When a level 01 screen item is encountered, the current screen position is reset to line 1, column 1. Then, as each elementary screen data description is processed, the current position is adjusted for the size of each definition. Therefore, by default, successively defined fields appear end to end in successive areas of the CRT screen. The position current at the start of any elementary screen data description may be changed by means of the LINE and COLUMN specifications. If neither LINE nor COLUMN is coded, the current screen position is not changed. If COLUMN is coded without LINE, the current screen line is not adjusted. If LINE is coded without COLUMN, COLUMN 1 is assumed. The LINE integer or COLUMN integer clause without PLUS causes the specified integer to be taken as the line or column at which the current screen item should start. The LINE PLUS integer or COLUMN PLUS integer clause causes the specified integer to be added to the current screen line or column, and the result to be used as the line or column at which the current screen item should start. If LINE (COLUMN) is given without integer-1 (integer-2), LINE PLUS 1 (COLUMN PLUS 1) is assumed.

3. BLANK LINE causes erasure of the screen from the current cursor position to the end of the current line and leaves the cursor position unchanged.

> NOTE
>
> The following functions are always executed in the order shown below, regardless of the order in which they are specified.

1. BLANK SCREEN
2. LINE/COLUMN positioning
3. BLANK LINE
4. Display or accept operation

3.13 DATA DIVISION LIMITATIONS

There is a limitation on the number of items in the Working-Storage, Linkage, and File sections of the Data Division. In those implementations of COBCL which have the Communications Level I facility, the number of CDs is relevant also. The sum:

$$\frac{W+4095}{4096} + F + L + C$$

must be less than or equal to 14, where W is the size of Working-Storage in bytes, F is the number of files described in the File Section, L is the number of level 01 or 77 entries in the Linkage Section, and C is the number of CD's in the Communications Section. Furthermore, the maximum number of files which may be open in the same run unit (main program linked together with an arbitrary number of subprograms) is 14.

CHAPTER 4

Procedure Division

In this chapter, the basic concepts of the Procedure Division are explained. Advanced topics (such as indexing of tables, indexed file accessing, interprogram communication and declaratives) are discussed in subsequent chapters.

4.1 STATEMENTS, SENTENCES, PROCEDURE-NAMES

The Procedure portion of a source program specifies those procedures needed to peform a particular data processing function. These steps (computations, logical decisions, etc.) are expressed in statements similar to English, which employ the concept of verbs to denote actions, and statements and sentences to describe procedures. The Procedure portion must begin with the following header:

PROCEDURE DIVISION.

A statement consists of a verb followed by appropriate operands (data-names or literals) and other words that are necessary for the completion of the statement. The two types of statements are imperative and conditional.

Imperative Statements

An imperative statement specifies an unconditional action to be taken by the object program. An imperative statement consists of a verb and its operands, excluding the IF and SEARCH conditional statements and any statement which contains an INVALID KEY, AT END, SIZE ERROR, OVERFLOW, or ON ESCAPE clause.

Conditional Statements

A conditional statement stipulates a condition that is tested to determine whether an alternate path of program flow is to be taken. The IF and SEARCH statements provide this capability. Any I/O statement having an INVALID KEY or AT END clause is also considered to be conditional. When an arithmetic statement posesses a SIZE ERROR suffix, the statement is considered to be conditional rather than imperative. STRING or UNSTRING statements having an OVERFLOW clause and ACCEPT with the ON ESCAPE clause are also conditional.

Sentences

A sentence is a single statement or a series of statements terminated by a period and followed by a space. If desired, a semi-colon or comma may be used between statements in a sentence.

Paragraphs

A paragraph is a logical entity consisting of zero, one or more sentences. Each paragraph must begin with a paragraph-name.

Sections

A section is composed of one or more successive paragraphs, and must begin with a section-heacer. A section header consists of a section-name conforming to the rules for procedure-name formation, followed by the word SECTION, an optional segment number, and a period. A section header must appear on a line by itself. Each section-name must be unique.

4.2 ORGANIZATION OF THE PROCEDURE DIVISION

The procedure part of a program may be subdivided in three possible ways:

1. The Procedure Division consists only of paragraphs.

2 The Procedure Division consists of zero or more paragraphs followed by a number of sections (each section subdivided into one or more paragraphs).

3. The Procedure Division consists of a DECLARATIVES portion and a series of sections (each section subdivided into one or more paragraphs).

The DECLARATIVES portion of the Procedure Division is optional; it provides a means of designating a procedure to be invoked in the event of an I/O error. If Declaratives are utilized, only possibility 3 may be used. Refer to Chapter 9 for a complete discussion.

4.3 MOVE STATEMENT

The MOVE statement is used to move data from one area of main storage to another and to perform conversions and/or editing on the data that is moved. The MOVE statement has the following format:

$$\underline{\text{MOVE}} \left\{\begin{matrix}\text{data-name-1}\\ \text{literal}\end{matrix}\right\} \quad \underline{\text{TO}} \text{ data-name-2} \quad [\text{data-name-3...}]$$

The data represented by data-name-1 or the specified literal is moved to the area designated by data-name-2. Additional receiving fields may be specified (data-name-3 etc.). When a group item is a receiving field, characters are moved without regard to the level structure of the group involved and without editing.

Subscripting or indexing associated with data-name-2 is evaluated immediately before data is moved to the receiving field. The same is true for other receiving fields (data-name-3, etc., if any). But for the source field, subscripting or indexing (associated with data-name-1) is evaluated only once, before any data is moved.

To illustrate, consider the statement

```
MOVE  A (B)  TO  B,  C (B),
```

which is equivalent to

```
MOVE A (B) TO temp
MOVE temp TO B
MOVE temp TO C (B)
```

where temp is an intermediate result field assigned automatically by the compiler.

The following considerations pertain to moving items:

1. Numeric (external or internal decimal, binary, numeric literal, or ZERO) or alphanumeric to numeric or report:

   a. The items are aligned by decimal points, with generation of zeros or truncation on either end, as required. If source is alphanumeric, it is treated as an unsigned integer and should not be longer than 31 characters.

   b. When the types of the source field and receiving field differ, conversion to the type of the receiving field takes place. Alphanumeric source items are treated as unsigned integers with Usage Display.

c. The items may have special editing performed on them with suppression of zeros, insertion of a dollar sign, etc., and decimal point alignment, as specified by the receiving area.

d. One should not move an item whose PICTURE declares it to be alphabetic or alphanumeric edited to a numeric or report item, nor is it possible to move a numeric item of any sort to an alphabetic item though numeric integers and numeric report items can be moved to alphanumeric items with or without editing, but operational signs are not moved in this case even if "SIGN IS SEPARATE" has been specified.

2. Non-numeric sourceand destinations:

a. The characters are placed in the receiving area from left to right, unless JUSTIFIED RIGHT applies.

b. If the receiving field is not completely filled by the data being moved, the remaining positions are filled with spaces.

c. If the source field is longer than the receiving field, the move is terminated as soon as the receiving field is filled.

3. When overlapping fields are involved, results are not predictable.

4. Appendix II shows, in tabular form, all permissible combinations of source and receiving field types.

5. An item having USAGE IS INDEX cannot appear as an operand of a MOVE statement. See SET in Chapter 6, Table Handling.

Examples of Data Movement (b represents blank):

Source Field		Receiving Field		
PICTURE	Value	PICTURE	Value before MOVE	Value after MOVE
99V99	1234	S99V99	9876-	1234+
99V99	1234	99V9	987	123
S9V9	12-	99V999	98765	01200+
XXX	A2B	XXXXX	Y9X8W	A2Bbb
9V99	123	99.99	87.65	01.23

4.4 INSPECT STATEMENT

The INSPECT statement enables the programmer to examine a character-string item. Options permit various combinations of the following actions:

1. counting appearances of a specified character
2. replacing a specified character with another
3. limiting the above actions by requiring the appearance of other specific characters

The format of the INSPECT statement is:

INPECT data-name-1 [TALLYING-clause] [REPLACING-clause]

where TALLYING-clause has the format

TALLYING data-name-2 FOR { CHARACTERS | ALL / LEADING operand-3 }

[BEFORE / AFTER INITIAL operand-4]

and REPLACING-clause has the format

REPLACING { CHARACTERS | ALL / LEADING / FIRST operand-5 } BY operand-6

[BEFORE / AFTER INITIAL operand-7]

Because data-name-1 is to be treated as a string of characters by INSPECT, it must not be described by USAGE IS INDEX, COMP, or COMP-3. Data-name-2 must be a numeric data item.

In the above formats, operand-n may be a quoted literal of length one, a figurative constant signifying a single character, or a data-name of an item whose length is one.

TALLYING-clause and REPLACING-clause may not both be omitted; if both are present, TALLYING-clause must be first.

TALLYING-clause causes character-by-character comparison, from left to right, of data-name-1, incrementing data-name-2 by one each time a match is found. When an AFTER INITIAL operand-4 subclause is present, the counting process begins only after detection of a character in data-name-1 matching operand-4. If BEFORE INITIAL operand-4 is specified, the counting process terminates upon encountering a character in data-name-1 which matches operand-4. Also going from left to right, REPLACING-clause causes replacement of characters under conditions specified by the REPLACING-clause. If BEFORE INITIAL operand-7 is present, replacement does not continue after detection of a character in data-name-1 matching operand-7. If AFTER INITIAL operand-7 is present, replacement does not commence until detection of a character in data-name-1 matching operand-7.

With bounds on data-name-1 thus determined, TALLYING and REPLACING is done on characters as specified by the following:

1. "CHARACTERS" implies that every character in the bounded data-name-1 is to be TALLYed or REPLACEd.

2. "All operand" means that all characters in the bounded data-name-1 which match the "operand" character are to participate in TALLYing/REPLACing.

3. "LEADING operand" specifies that only characters matching "operand" from the leftmost portion of the bounded data-name-1 which are contiguous (such as leading zeros) are to participate in TALLYing or REPLACing.

4. "FIRST operand" specifies that only the first-encountered character matching "operand" is to participate in REPLACing. (This option is unavailable in TALLYing.)

When both TALLYING and REPLACING clauses are present, the two clauses behave as if two INSPECT statements were written, the first containing only a TALLYING-clause and the second containing only a REPLACING-clause.

In developing a TALLYING value, the final result in data-name-2 is equal to the tallied count plus the initial value of data-name-2. In the first example below, the item COUNTX is assumed to have been set to zero initially elsewhere in the program.

INSPECT ITEM TALLYING COUNTX FOR ALL "L" REPLACING LEADING "A" BY "E" AFTER INITIAL "L"

Original (ITEM):	SALAMI	ALABAMA
Result (ITEM):	SALEMI	ALEBAMA
Final (COUNTX):	1	1

INSPECT WORK-AREA REPLACING ALL DELIMITER BY TRANSFORMATION

Original (WORK-AREA):	NEW YORK N Y (length 16)
Original (DELIMITER):	(space)
Original (TRANSFORMATION):	. (period)
Result (WORK-AREA):	NEW.YORK..N.Y...

> NOTE
>
> If any data-name-1 or operand-n is described as signed numeric, it is treated as if it were unsigned.

4.5 ARITHMETIC STATEMENTS

There are five arithmetic statements: ADD, SUBTRACT, MULTIPLY, DIVIDE and COMPUTE. Any arithmetic statement may be either imperative or conditional. When an arithmetic statement includes an ON SIZE ERROR specification, the entire statement is termed conditional, because the size error condition is data-depencent.

An example of a conditional arithmetic statement is:

```
ADD 1 TO RECORD-COUNT
    ON SIZE ERROR MOVE ZERO TO RECORD-COUNT
    DISPLAY "LIMIT 99 EXCEEDED".
```

If a size error occurs (in this case, it is apoarent that RECORD-COUNT has PICTURE 99, and cannot hold a value of 100), both the MOVE and DISPLAY statements are executed.

The three statement components that may appear in arithmetic statements (GIVING option, ROUNDED option, and SIZE ERROR option) are discussed in detail later in this section.

Basic Rules for Arithmetic Statements

1. All data-names used in arithmetic statements must be elementary numeric data items that are defined in the Data Division of the program, except that operands of the GIVING option may be report (numeric edited) items. Index-names and index data items are not permissible in these arithmetic statements (see Chapter 6).

2. Decimal point alignment is supplied automatically throughout the computations.

3. Intermediate result fields generated for the evaluation of arithmetic expressions assure the accuracy of the result field, except where high-order truncation is necessary.

5.1 SIZE ERROR OPTION

If, after decimal-point alignment and any low-order rounding, the absolute value of a calculated result exceeds the largest value which the receiving field is capable of holding, a space size error condition exists.

The optional SIZE ERROR clause is written immediately after any arithmetic statement, as an extension of the statement. The format of the SIZE ERROR option is:

ON SIZE ERROR imperative statement ...

If the SIZE ERROR option is present, and a size error condition arises, the value of the resultant data-name is unaltered and the series of imperative statements specified for the condition is executed.

If the SIZE ERROR option has not been specified and a size error condition arises, no assumption should be made about the final result.

An arithmetic statement, if written with the 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.

4.5.2 ROUNDED OPTION

If, after decimal-point alignment, the number of places in the fraction of the result is greater than the number of places in the fractional part of the data item that is to be set equal to the calculated result, truncation occurs unless the ROUNDED option has been specified.

When the ROUNDED option is specified, the least significant digit of the resultant data-name has its value increased 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.

The following chart illustrates the relationship between a calculated result and the value stored in an item that is to receive the calculated result, with and without rounding.

	Item to Receive Calculated Result		
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	S99V	66	65
.0055	SV999	.006	.005

Illustration of Rounding

When the low order integer positions in a resultant-identifier are represented by the character P in its PICTURE, rounding or truncation occurs relative to the rightmost integer position for which storage is allowed.

4.5.3 GIVING OPTION

If the GIVING option is written, the value of the data-name that follows the word GIVING is made equal to the calculated result of the arithmetic operation. The data-name that follows GIVING is not used in the computation and may be a report (numeric-edited) item.

4.5.4 ADD STATEMENT

The ADD statement adds two or more numeric values and stores the resulting sum. The ADD statement general format is:

```
ADD  {numeric-literal}
     {data-name-1    }  ...

     {TO    }
     {GIVING}data-name-n   [ROUNDED] [SIZE-ERROR-clause]
```

When the TO option is used, the values of all the data-names (including data-name-n) and literals in the statements are added, and the resulting sum replaces the value of data-name-n. When the GIVING option is used, at least two data-names and/or numeric literals must be coded between ADD and GIVING. The sum of the values of these data-names and literals (not including data-name-n) replaces the value of data-name-n.

The following are examples of proper ADD statements:

```
ADD INTEREST, DEPOSIT TO BALANCE ROUNDED
ADD REGULAR-TIME OVERTIME GIVING GROSS-PAY.
```

The first statement would result in the sum of INTEREST, DEPOSIT, and BALANCE being placed at BALANCE, while the second would result in the sum of REGULAR-TIME and OVERTIME earnings being placed in item GROSS-PAY.

4.5.5 SUBTRACT STATEMENT

The SUBTRACT statement subtracts one or more numeric data items from a specified item and stores the difference.

The SUBTRACT statement general format is:

```
SUBTRACT {data-name-1    } ... FROM
         {numeric-literal-1}

         {data-name-m [GIVING data-name-n]    }
         {numeric literal-m GIVING data-name-n}

    [ROUNDED] [SIZE-ERROR-clause]
```

The effect of the SUBTRACT statement is to sum the values of all the operands that precede FROM and subtract that sum from the value of the item following FROM.

The result (difference) is stored in data-name-n, if there is a GIVING option. Otherwise, the result is stored in data-name-m.

4.5.6 MULTIPLY STATEMENT

The MULTIPLY statement multiplies two numeric data items and stores the product.

The general format of the MULTIPLY statement is:

```
MULTIPLY {data-name-1        }
         {numeric-literal-1  }

    BY {data-name-2 [GIVING data-name-3]   }
       {numeric-literal-2 GIVING data-name-3}

    [ROUNDED] [SIZE-ERROR-clause]
```

When the GIVING option is omitted, the second operand must be a data-name; the product replaces the value of data-name-2. For example, a new BALANCE value is computed by the statement MULTIPLY 1.03 BY BALANCE. (Since this order may seem somewhat unnatural, it is recommended that GIVING always be written, e.g. MULTIPLY 1.03 BY BALANCE GIVING BALANCE.)

4.5.7 DIVIDE STATEMENT

The DIVIDE statement divides two numeric values and stores the quotient. The general format of the DIVIDE statement is:

```
DIVIDE {data-name-1        } {BY  } {data-name-2        }
       {numeric-literal-1  } {INTO} {numeric-literal-2  }

    [GIVING data-name-3] [ROUNDED] [SIZE-ERROR-clause]
```

The BY-form signifies that the first operand (data-name-1 or numeric-literal-1) is the dividend (numerator), and the second operand (data-name-2 or numeric-literal-2) is the divisor (denominator). If GIVING is not written in this case, then the first operand must be a data-name, in which the quotient is stored.

The INTO-form signifies that the first operand is the divisor and the second operand is the dividend. If GIVING is not written in this case, then the second operand must be a data-name, in which the quotient is stored.

Division by zero always causes a size-error condition.

4.5.8 COMPUTE STATEMENT

The COMPUTE statement evaluates an arithmetic expression and then stores the result in a designated numeric or report (numeric edited) item.

The general format of the COMPUTE statement is:

COMPUTE data-name-1 [ROUNDED]...=

{data-name-2 / numeric-literal / arithmetic-expression} [SIZE-ERROR-clause]

An example of such a statement is:

```
COMPUTE GROSS-PAY ROUNDED = BASE-SALARY *
    (1 + 1.5 * (HOURS - 40) / 40).
```

An arithmetic expression is a proper combination of numeric literals, data-names, arithmetic operators and parentheses. In general, the data-names in an arithmetic expression must designate numeric data. Consecutive data-names (or literals) must be separated by an arithmetic operator, and there must be one or more blanks on either side of the operator. The operators are:

- `+` for addition
- `-` for subtraction
- `*` for multiplication
- `/` for division
- `**` for exponentiation to an integral power.

When more than one operation is to be executed using a given variable or term, the order of precedence is:

1. Unary (involving one variable) plus and minus
2. Exponentiation
3. Multiplication and Division
4. Addition and Subtraction

Parentheses may be used when the normal order of operations is not desired. Expressions within parentheses are evaluated first; parentheses may be nested to any level. Consider the following expression.

```
A + B / (C - D * E)
```

Evaluation of the above expression is performed in the following ordered sequence:

1. Compute the product D times E, considered as intermediate result R1.
2. Compute intermediate result R2 as the difference C - R1.
3. Divide B by R2, providing intermediate result R3.
4. The final result is computed by addition of A to R3.

Without parentheses, the expression

```
A + B / C - D * E
```

is evaluated as:

```
R1 = B / C
R2 = A + R1
R3 = D * E
final result = R2 - R3
```

When parentheses are employed, the following punctuation rules should be used:

1. A left parenthesis is preceded by one or more spaces.
2. A right parenthesis is followed by one or more spaces.

The expression A - B - C is evaluated as (A - B) - C. Unary operators are permitted, e.g.:

```
COMPUTE A = +C + -4.6
COMPUTE X = -Y
COMPUTE A, B(I) = -C - D(3)
```

4.6 GO TO STATEMENT

The GO TO statement transfers control from one portion of a program to another. It has the following general format:

GO TO [procedure-name-1 [[procedure-name-2] ...DEPENDING ON data-name]]

The simple form GO TO procedure-name-1 changes the path of control to a designated paragraph or section. If the GO statement is without a procedure-name, then that GO statement must be the only one in a paragraph, and must be ALTERed (see 4.12) prior to its execution.

The more general form designates N procedure-names as a choice of N paths to transfer to, if the value of data-name is 1 to N, respectively. Otherwise, there is no transfer of control and execution proceeds in the normal sequence. Data-name must be a numeric elementary item and have no positions to the right of the decimal point.

If a GO (non-DEPENDING) statement appears in a sequence of imperative statements, it must be the last statement in that sequence.

4.7 STOP STATEMENT

The STOP statement is used to terminate or delay execution of the object program.

The format of this statement is:

```
STOP    { RUN     }
        { literal }
```

STOP RUN terminates execution of a program, closing all files and returning control to the operating system. If used in a sequence of imperative statements, it must be the last statement in that sequence.

The form STOP literal displays the specified literal on the console and suspends execution. Execution of the program is resumed only after operator intervention. Presumably, the operator performs a function suggested by the content of the literal, prior to resuming program execution by pressing the carriage return key.

4.8 ACCEPT STATEMENT

The ACCEPT statement is used by a processing program to obtain low-volume input at runtime. Four formats are available:

Format 1:

```
ACCEPT identifier-1 FROM { DATE        }
                         { DAY         }
                         { TIME        }
                         { LINE NUMBER }
                         { ESCAPE KEY  }
```

Format 2:

```
ACCEPT identifier-2
```

Format 3:

ACCEPT position-spec identifier-3 [WITH { SPACE-FILL / ZERO-FILL / LEFT-JUSTIFY / RIGHT-JUSTIFY / TRAILING-SIGN / PROMPT / UPDATE / LENGTH-CHECK / AUTO-SKIP / BEEP } ...]

Format 4:

ACCEPT screen-name [ON ESCAPE imperative-statement]

The function of each form of the ACCEPT statement is to acquire data from a source external to the program and place it in a specified receiving field or set of receiving fields. The forms differ primarily in the data source with which they are designed to interface. The format 1 ACCEPT obtains information from system-defined data items. The other formats of the ACCEPT statement receive data keyed in by an operator at the system console device. For format 2, this device is assumed to be a teletype, a glass teletype, or a CRT terminal in scrolling mode. For format 3, it is assumed that the input device is a video terminal and that scrolling is not desired. The format 4 ACCEPT receives an entire data entry form (as defined in the SCREEN SECTION) when it has been completed by the terminal operator. Note that an ordinary CRT terminal is suitable as an input device for a format 2, 3, or 4 ACCEPT, although the possible effects on the appearance of the screen will differ as indicated in the discussion below. The effects of the various WITH phrase options of the format 3 ACCEPT statement are summarized in Section 4.8.3.3.

4.8.1 FORMAT 1 ACCEPT STATEMENT

Any of several system-defined data items may be obtained at execution time by use of the format 1 ACCEPT statement.

The formats of the system-defined data items are:

DATE -- a six digit value of the form YYMMDD (year, month, day). Example: July 4, 1976 is 760704

DAY — a five digit "Julian date" of the form YYNNN where YY is the two low order digits of year and NNN is the day-in-year number between 1 and 366.

TIME -- an eight digit value of the form HHMMSSFF where HH is from 00 to 23, MM is from 00 to 59, SS is from 00 to 59, and FF is from 00 to 99; HH is the hour, MM is the minutes, SS is the seconds, and FF represents hundredths of a second.

LINE NUMBER -- a two digit value that represents the line (terminal) on which the program is currently running. In the COBOL-80 system, the value of LINE NUMBER is always 00.

ESCAPE KEY -- a two digit code generated by the key that terminated the most recently executed format 3 or format 4 ACCEPT statement. Identifier-1 can be interrogated to determine exactly which key was typed. Input may be terminated by any of the following keys, and cause the ESCAPE KEY value to be set as shown:

Backtab (terminates only format 3 ACCEPTs)	99
Escape	01
Field-terminator (of the last field if format 4 ACCEPT is used)	00
Function key	02-nn

All key codes are defined in the CRT driver for the terminal being used (refer to Appendix A of the User's Guide). On most terminals, backtab may be entered as CONTROL-B or ^; escape is the ESCAPE or ALT key; field-terminator may be entered as CARRIAGE RETURN, LINE FEED, TAB, ENTER, NEW LINE or CONTROL-I; and the function keys are usually CONTROL-A, CONTROL-C, and CONTROL-X, generating ESCAPE KEY values of 02, 03, and 04 respectively. If input is terminated as a result of using the AUTO-SKIP option (i.e., no terminator key is struck), the ESCAPE KEY value is set to 00.

identifier-1 should be an unsigned numeric integer whose length agrees with the content of the system-defined data item. If not, the standard rules for a MOVE govern storage of the source value in the receiving item (identifier-1).

4.8.2 FORMAT 2 ACCEPT STATEMENT

Format 2 of the ACCEPT statement is used to accept a string of input characters from a scrolling device such as a teletype or a CRT in scrolling mode. When the ACCEPT statement is executed, input characters are read from the console device until a carriage return is encountered, then a carriage return/line feed pair is sent back to the console. The input data string is considered to consist of all characters keyed prior to (but not including) the carriage return.

For a Format 2 ACCEPT with an alphanumeric receiving field, the input data string is transferred to the receiving field exactly as if it were being MOVEd from an alphanumeric field of length equal to the number of characters in the string. (That is, left justification, space filling, and right truncation occur by default, and right justification and left truncation occur if the receiving field is described as JUSTIFIED RIGHT.) If the receiving field is alphanumeric-edited, it is treated as an alphanumeric field of equal length (as if each character in its PICTURE were "X"), so that no insertion editing will occur.

For a Format 2 ACCEPT with a numeric or numeric-edited receiving field, the input data string is subjected to a validity test which depends on the PICTURE of the receiving field. (If the receiving field is described as COMP, its PICTURE is treated as "S9(5)" for purposes of this discussion.) The digits 0 through 9 are considered valid anywhere in the input data string.

The decimal point character (period or comma, depending on the DECIMAL POINT IS clause of the CONFIGURATION SECTION) is considered valid if:

1. it occurs only once in the input data string, and

2. if the PICTURE of the receiving field contains a fractional digit position, that is, a "9", "Z", "*", or floating insertion character which appears to the right of either an assumed decimal point ("V") or an actual decimal point (".").

The operational sign characters "+" and "-" are considered valid only as the first or last character of the input string and only if the PICTURE of the receiving field contains one of the sign indicators "S", "+", "-", "CR", or "DB".

All other characters are considered invalid. If the input data string is invalid, the message "INVALID NUMERIC INPUT - PLEASE RETYPE" is sent to the console, and another input data string is read.

When a valid input data string has been obtained, data is transferred to the receiving field exactly as if the instruction being executed were a MOVE to the receiving field from a hypothetical source field with the following characteristics:

1. a PICTURE of the form S9...9V9...9

2. USAGE DISPLAY

3. a total length equal to the number of digits in the input data string

4. as many digit positions to the right of the assumed decimal point as there are digits to the right of the explicit decimal point in the input data string (zero if there is no decimal point in the input data string)

5. current contents equal to the string of digits embedded in the input data string

6. a separate sign with a current negative status if the input data string contains the character "-", and a current positive status otherwise.

4.8.3 FORMAT 3 ACCEPT STATEMENT

Format 3 of the ACCEPT statement is used to accept data into a field from a non-scrolling video terminal. The following syntax rules must be observed when the format 3 ACCEPT is used:

1. identifier-3 must reference a data item whose length is less than or equal to 1920 characters

2. the options SPACE-FILL and ZERO-FILL may not both be specified in the same ACCEPT statement

3. the options LEFT-JUSTIFY and RIGHT-JUSTIFY may not both be specified within the same ACCEPT statement

4. if identifier-3 is described as a numeric-edited item, the UPDATE option must not be specified

5. the TRAILING-SIGN option may be specified only if identifier-3 is described as an elementary numeric data item. If identifier-3 is described as unsigned, the TRAILING-SIGN option is ignored

6. for alphanumeric or alphanumeric-edited identifier-3, the SPACE-FILL option is assumed if the ZERO-FILL option is not specified, and the LEFT-JUSTIFY option is assumed if the RIGHT-JUSTIFY option is not specified

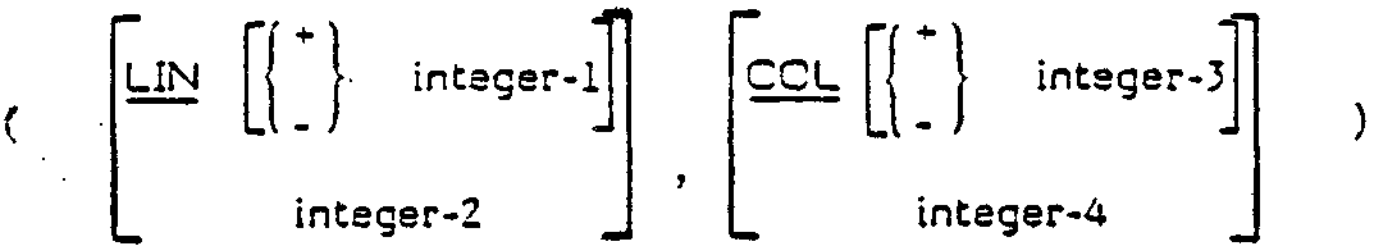
7. for numeric or numeric-edited identifier-3, the ZERO-FILL option is assumed if the SPACE-FILL option is not specified.

4.8.3.1 Data Input Field

The position-spec and receiving field (identifier-3) specifications of the format 3 ACCEPT statement are used to define the location and characteristics of a data input field on the screen of the console video terminal.

Location of the Data Input Field

The position-spec is of the form



The opening and closing parentheses and the comma and space separating the two major bracketed groups are required. The position-spec specifies the position on the console CRT screen at which the data input field will begin. LIN and COL are COBOL special registers. Each behaves like a numeric data item with USAGE COMP, but they may be referenced by every COBOL program without being declared in the DATA DIVISION.

If LIN is specified, the data input field will begin on the screen row whose number is equal to the value of the LIN special register, incremented (or decremented) by integer-1 if "+ integer-1" (or "- integer-1") is specified. If integer-2 is specified, the data input field will begin on the row whose number is integer-2. If neither LIN nor integer-2 is specified, the data input field will begin on the screen row containing the current cursor position.

If COL is specified, the data input field will begin in the screen column whose number is equal to the value of the COL special register, incremented (or decremented) by integer-3 if "+ integer-3 (or "- integer-3") is specified. If integer-4 is specified, the data input field will begin in the screen column whose number is integer-4. If neither COL nor integer-4 is specified, the data input field will begin in the screen column containing the current cursor position.

Characteristics of the Data Input Field

The characteristics (other than position) of the data input field on the CRT screen are determined by the receiving field's PICTURE specification (which is treated as S9(5) in the case of an item whose USAGE is COMPUTATIONAL). For alphanumeric or alphanumeric-edited identifier-3, the data input field is simply a string of data input character positions starting at the screen location specified by position-spec. The length of the data input field in character positions is equal to the length of the receiving field in memory.

For numeric or numeric-edited identifier-3, the data input field may contain any or all of the following: integer digit positions, fractional digit positions, sign position, decimal point position. There will be one digit position for each "9", "Z", "*", "P", or non-initial floating insertion symbol (a floating insertion symbol is a "+", "-", or "$" which is not the last symbol in a PICTURE character string) in the PICTURE of identifier-3. Each digit position in the data input field is a fractional digit position if the corresponding PICTURE character is to the right of an assumed decimal point ("V") or actual decimal point (".") in the PICTURE of identifier-3. Otherwise it is an integer digit position. There will be one sign position if identifier-3 is described as signed, and no sign position otherwise. There will be one decimal point position if there is at least one fractional digit position, and no decimal point position otherwise.

The data input positions which are defined will occupy successive character positions on the CRT screen beginning with the position specified by position-spec. If TRAILING-SIGN is specified in the ACCEPT statement, the data input positions will be in the following sequence: integer digit positions (if any), decimal point position (if any), fractional digit positions (if any), sign position (if any). If TRAILING-SIGN is not specified, the data input positions will be in the following sequence: sign position (if any), integer digit positions (if any), decimal point position (if any), fractional digit positions (if any).

4.8.3.2 Data Input and Data Transfer

A character entered into the data input field by the terminal operator may be treated either as an editing character, a terminator key or a data character. When a terminator key is typed, the ACCEPT is terminated and the ESCAPE KEY value is set as described in section 4.8.1. This value can be interrogated by using a format 1 ACCEPT statement FROM ESCAPE KEY.

The editing characters are line-delete, forward-space, backspace, and rubout. On most terminals, these characters may be entered as control-U, control-F, control-H, and DEL (or RUB) respectively. The action of the editing characters is described later in this section; for now, only data characters will be considered.

See the COBOL-80 User's Guide for further information on the definition of editing and terminator characters.

Alphanumeric Receiving Field

Consider first the execution of the format 3 ACCEPT statement with an alphanumeric or alphanumeric-edited receiving field. An alphanumeric-edited receiving field is treated as an alphanumeric field of the same length (as if every character in its PICTURE were "X"). Specifically, no insertion editing will occur.

The initial appearance of the data input field depends on the specifications in the WITH phrase of the ACCEPT statement. If UPDATE is specified, the current contents of identifier-3 are displayed in the input field. In this case all data input positions will be treated as if they were keyed by the terminal operator. If UPDATE is not specified, but PROMPT is specified, a period (".") is displayed in each input data position. If neither UPDATE nor PROMPT is specified, the data input field is not changed. The cursor is placed in the first data input position, and characters are accepted as they are keyed by the operator until a terminator character (normally carriage return) is encountered. If AUTO-SKIP is specified in the ACCEPT statement, the ACCEPT will also be terminated if the operator keys a character into the last (rightmost) data input position.

As each input character is received, it is echoed to the CRT screen, except that non-displayable characters are echoed as "?". If all positions of the data input field are filled, additional input is ignored until a terminator character or editing character (listed above) is encountered. If RIGHT-JUSTIFY was specified in the ACCEPT statement, the operator-keyed characters are shifted to the rightmost positions of the data input field when the ACCEPT is terminated. All unkeyed character positions are filled on termination; the fill character is either space (if SPACE-FILL is in effect) or zero (if ZERO-FILL was specified).

The contents of the receiving field will be the same set of characters as appear in the input field; however, the justification of operator-keyed characters will be controlled by the JUSTIFIED specification in the receiving field's data description, not by the RIGHT- or LEFT-JUSTIFY option of the ACCEPT. Excess positions of the receiving field will be filled with spaces or zeroes based on the SPACE- or ZERO-FILL specification in the ACCEPT statement.

Numeric Receiving Field

Next, consider the execution of a format 3 ACCEPT statement with a numeric or numeric-edited receiving field. As described above, the data input field on the console CRT screen may contain integer digit positions, fractional digit positions, or both. First assume that both are present; the other cases will be treated as variations.

As with the alphanumeric ACCEPT, the data input field may be initialized in a way determined by the WITH options specified in the ACCEPT statement. If UPDATE is specified (not permitted for a numeric-edited receiving field), the integer and fractional parts of the data inout field will be set to the integer and fractional parts of the decimal representation of the initial value of the receiving field, with leading and trailing zeroes included, if necessary, to fill all digit positions. Except for leading zeroes, these initialization characters are treated as operator-keyed data. If UPDATE is not specified, but PROMPT is specified, a zero will be displayed in each input digit position. In either of these cases (UPDATE or PROMPT) a decimal point will be displayed at the decimal point position.

If neither UPDATE nor PROMPT is specified, the input field on the screen will not be initialized, except for the sign position. The sign position is always initialized positive except when UPDATE is specified, in which case it is initialized according to the sign of the current contents of the receiving field. On most systems, a positive sign position is shown as a space, and a negative sign position is shown as a minus sign.

The cursor is initially placed in the rightmost integer digit position, and characters are accepted one at a time as they are keyed by the operator. A received character may be treated in one of several ways. If the incoming character is a digit, previously keyed digits are shifted one position to the left in the input field and the new digit is displayed in the rightmost integer digit position. If all integer digit positions have not been filled, the cursor remains on the rightmost digit position and another character is accepted. If the entire integer part of the input field has been filled and AUTO-SKIP was specified, the integer part is terminated and the cursor is moved to the leftmost fractional digit position. If the integer part has been filled and AUTO-SKIP was not specified, the cursor is moved to the decimal point position, and any further digits keyed are ignored until the integer part is terminated with a decimal point.

If the character entered is one of the sign characters "+" or "-", the sign position is changed to a positive or negative status respectively. Cursor position is not affected.

If the character entered is a decimal point character, the integer part is terminated and the cursor is moved to the leftmost fractional digit position.

If the character entered is a field terminator (normally carriage-return), the ACCEPT is terminated and the cursor is turned off. Any other character is ignored.

When the integer part is terminated, the cursor is placed in the leftmost fractional digit position, and operator-keyed characters are again accepted. Digits are simply echoed to the terminal. The sign characters "+" and "-" are treated exactly as they were while integer part digits were being entered. The field terminator character terminates the ACCEPT. (If AUTO-SKIP is in effect, filling the entire fractional part also terminates the ACCEPT.) Other characters are ignored. After all digit positions of the fractional part have been filled, further digits are also ignored.

If no fractional digit positions are present, the decimal point is ignored as an input character, and entry of integer part digits may be terminated only by terminating the entire ACCEPT. If no integer digit positions are present, the cursor is initially placed in the leftmost fractional digit position and entry of the fractional part digits proceeds as described above.

On termination of the format 3 ACCEPT of a numeric or numeric-edited item, data is transferred to the receiving field. The exact form of the data in the receiving field after execution of the ACCEPT is as described in the last paragraph of the discussion of the format 2 ACCEPT, where the role of the "input data string" mentioned in that paragraph is taken by the string of characters displayed in the data input field. After termination, if SPACE-FILL is in effect, leading zeroes in the integer part of the data input field (not in the receiving field) will be replaced by spaces, and the leading operational sign, if present, will be moved to the rightmost space thus created.

Editing Characters

The editing characters (line-delete, forward-space, backspace, and rubout) may be used to change data which has already been keyed (or supplied by the CCBCL runtime system as a result of a WITH UPDATE specification). Entering the line-delete character will cause the ACCEPT to be restarted and all data keyed by the operator or initially present in the receiving field to be lost. The data input field on the console screen will be re-initialized if PRCMPT is in effect. Otherwise, the data input field will be filled with spaces or zeroes according to the SPACE-FILL or ZERO-FILL specification.

Typing the forward-space or backspace characters will move the cursor forward or back one data input position in the case of an alphanumeric or alphanumeric-edited receiving field, or one digit position in the case of a numeric or numeric-edited receiving field. In no case, however, will the forward-space or backspace characters move the cursor outside the range of positions including (1) the positions already keyed by the operator (or filled by CCBCL runtime support when WITH UPDATE is specified), and (2) the rightmost data input position which the cursor has occupied during the execution of this ACCEPT. If the cursor is moved to a position of this range other than the rightmost, and a legal data character is entered, it is displayed at the current cursor position and the cursor is moved forward one data position (alphanumeric or alphanumeric-edited) or digit position (numeric or numeric-edited).

Typing the rubout character effectively cancels the last data character entered. The cursor is moved back one data position (digit position if the receiving field is numeric or numeric-edited) and a fill character (space or zero) is displayed under the cursor (except when the cursor is to the left of the decimal point for a numeric ACCEPT. Then no fill character is displayed and the cursor is not moved. but the digit at the cursor position is deleted and all digits to the left of it are shifted one position to the right.) The rubout character has no effect unless the cursor is in position to accept a new data character; in other words, it has no effect if backspace character(s) have been used to move the cursor back over already keyed positions.

4.8.3.3 WITH Phrase Summary

The following list summarizes the effects of the WITH phrase specifications for a format 3 ACCEPT with an alpnanumeric or alphanumeric-edited receiving field:

1. SPACE-FILL causes unkeyed character positions of the data input field and the receiving field to be space-filled when the ACCEPT is terminated.

2. ZERO-FILL causes unkeyed character positions of the data input field and the receiving field to be set to ASCII zeroes when the ACCEPT is terminated.

3. LEFT-JUSTIFY is treated by this compiler as commentary.

4. RIGHT-JUSTIFY causes operator-keyed characters to occupy the rightmost positions of the data input field (on the screen) after the ACCEPT is terminated. Note that the justification of transferred data in the receiving field is controlled by the JUSTIFIED declaration or default of the receiving field's data description, not by the WITH RIGHT-JUSTIFY phrase.

5. PROMPT causes the data input field on the screen to be set to all periods (".") before input characters are accepted.

6. UPDATE causes the data input field to be initialized with the initial contents of the receiving field and the initial data to be treated as operator-keyed data.

7. LENGTH-CHECK causes a field terminator character to be ignored unless every data input position has been filled.

8. AUTO-SKIP forces the ACCEPT to be terminated when all data input positions have been filled. A terminator character explicitly keyed has its usual effect.

9. BEEP causes an audible alarm to sound when the ACCEPT is initialized and the system is ready to accept operator input.

The following list summarizes the effects of the WITH phrase specifications for the format 3 ACCEPT with a numeric or numeric-edited receiving field:

1. SPACE-FILL causes unkeyed digit positions of the data input field (not of the receiving field) to the left of the (possibly implied) decimal point to be space-filled when the ACCEPT is terminated and any leading operational sign to be displayed in the rightmost space thus created.

2. ZERO-FILL causes all unkeyed digit positions of the data input field to be set to zero when the ACCEPT is terminated.

3. LEFT-JUSTIFY and RIGHT-JUSTIFY have no effect for a numeric or numeric-edited receiving field.

4. TRAILING-SIGN causes the operational sign to appear as the rightmost position of the data input field. Ordinarily the sign is the leftmost position of the field.

5. PROMPT causes the data input field positions to be initialized as follows before input characters are accepted: digit positions to zero, decimal point position (if any) to the decimal point character, and sign position (if any) to space.

6. UPDATE causes the data input field to be initialized to the current contents of the receiving field and this initial data to be treated like operator-keyed data.

7. LENGTH-CHECK causes a received decimal point character to be ignored unless all integer digit positions have been keyed and a field terminator character to be ignored unless all digit positions have been keyed.

8. AUTO-SKIP causes the integer part of the ACCEPT to be terminated when all integer digit positions have been keyed and the entire ACCEPT to be terminated when all digit positions have been keyed.

9. BEEP causes an audible alarm to sound when the ACCEPT is initialized and the system is ready to accept operator input.

4.8.4 Examples Using the Format 3 ACCEPT Statement

Example 1:

Receiving Field: 01 RS-DISCOUNT PIC X(8). Initial Contents: ABCDEFGH ACCEPT Statement: ACCEPT (1, 1) RS-DISCOUNT WITH PROMPT	Set-up prior to executing
At Start of ACCEPT: :..... Operator Enters N: N:..... Operator Enters ONE: NONE:... Operator Enters Carriage Return: NONEØØØØ	Executing the ACCEPT
Final Contents of Receiving Field: NONEØØØØ	Result

Example 2:

Receiving Field: 10 VEND-NAME PIC X(12). Initial Contents: ACME␣WIDGETS ACCEPT Statement: ACCEPT (1, 1) VEND-NAME WITH PROMPT UPDATE.	Set-up prior to executing
At Start of ACCEPT: ACME␣WIDGETS (If operator enters carriage return here, the receiving field will not be changed.) Operator Enters Line-delete: :..... Operator Enters XYZ: XYZ:..... Operator Enters Carriage Return: XYZ␣␣␣␣␣␣␣␣␣	Executing the ACCEPT
Final Contents of Receiving Field: XYZ␣␣␣␣␣␣␣␣␣	Result

Example 3:

Receiving Field: 05 CREDIT PIC S9(4)V99. Initial Contents: + 111111 ACCEPT Statement: ACCEPT (LIN + 4, COL - 3) CREDIT WITH PROMPT TRAILING-SIGN.	Set-up prior to executing
At Start of ACCEPT: 0000.00% Operator Enters 8: 0008.00% Operator Enters 7: 0087.00% Operator Enters -: 0087.00- Operator Enters 6: 0876.00- Operator Enters N: 0876.00- Operator Enters .: 0876.00- Operator Enters 5: 0876.50- Operator Enters Carriage Return: 0876.50-	Executing the ACCEPT
Final Contents of Receiving Field: 0876 ^ 50	Result

4.8.5 FORMAT 4 ACCEPT STATEMENT

Format 4 of the ACCEPT statement causes a transfer of information from the operator's console to all TO and/or USING fields specified in the SCREEN SECTION definition of screen-name (or screen items subordinate to screen-name.) Screen items having only a VALUE literal or a FROM clause have no effect on the operation of the ACCEPT statement. Each transfer consists of an implicit format 3 ACCEPT of a field defined by the appropriate screen item's PICTURE followed by an implicit MOVE to the associated TO or USING field. When the ACCEPT is terminated, the ESCAPE KEY value is set as described below and in section 4.8.1. This value can be interrogated by using a format 1 ACCEPT statement FROM ESCAPE KEY. Fields are ACCEPTed in the order in which they are defined under screen-name in the SCREEN SECTION. This order can be changed by use of the backtab key, as described below, but the position of the field on the screen does not affect the order.

If an escape key is typed during data input, the entire ACCEPT is terminated without moving the current field to the associated TO or USING item, the ESCAPE KEY value is set to 01, and the ON ESCAPE statement is executed. If a function key is typed, the appropriate ESCAPE KEY value is set and the entire ACCEPT is terminated. If a field-terminator key (carriage return, tab, etc.) is typed, the ESCAPE KEY value is set to 00 and the cursor moves to the next input field defined under screen-name, if one exists. If the current field is the last field, the entire ACCEPT is terminated. If the backtab key is typed, the current field is terminated and the cursor moves to the previous input field defined under screen-name. If the current field is the first field, the cursor does not move from that field. When a field is terminated by a function key, field-terminator key, or backtab key, the contents of the current field are moved to the associated TO or USING item, except in the case where no data characters and no editing characters have been entered in that field. This allows the operator to tab forward or backward through the input fields without affecting the contents of the receiving items.

All the editing and validation features described in section 4.8.3.2 for the format 3 ACCEPT apply to the format 4 ACCEPT as well. Several SCREEN SECTION specifications listed in section 3.17 correspond to the format 3 ACCEPT options: AUTO corresponds to AUTO-SKIP; BELL corresponds to BEEP; and JUSTIFIED corresponds to RIGHT-JUSTIFY. Furthermore, if an input field specifies the USING clause or both a FROM and TO clause, the ACCEPT will be executed with the UPDATE option. Format 4 ACCEPT statements always use the PROMPT and TRAILING-SIGN options when executing the individual format 3 ACCEPTs.

If the screen item's PICTURE specifies a numeric-edited or alphanumeric-edited input field, the ACCEPT is executed as if the field were numeric or alphanumeric, respectively. When the field is terminated the data is edited according to the PICTURE and redisplayed in the specified screen position. In this case, the JUSTIFIED clause has no effect.

Moves from screen fields to receiving items follow the standard COBOL- rules for MOVE statements, except that moves from numeric-edited fields are allowed. In this case, the data is input as if the field were numeric and the move uses only the sign, decimal point and digit characters.

The format 4 ACCEPT does not cause the display of any text or prompting label information. See the discussion of DISPLAY in section 4.9.

4.9 DISPLAY STATEMENT

The DISPLAY statement provides the capability of outputting low-volume data at runtime without the overhead of file definition. The format of the DISPLAY statement is:

```
DISPLAY { { [position-spec]  { identifier }                    }
          {                  { literal    }  ... [UPON mnemonic-name] }
          {                  { ERASE      }                    }
          {                                                    }
          {                  [screen-name]                     }
```

The DISPLAY statement must be coded in accordance with the following rules:

1. identifier must reference a data item whose length is less than or equal to 1920 characters.

2. mnemonic-name must be defined in the PRINTER IS clause of the SPECIAL-NAMES paragraph of the CONFIGURATION SECTION

3. screen-name must be defined in the SCREEN SECTION of the DATA DIVISION.

The DISPLAY statement will cause output to be sent to the system console device unless UPON mnemonic-name is specified, in which case output will be sent to the printer. Each display-item (that is, each occurrence of identifier, literal, ERASE, or screen-name) will be processed in turn as described in the paragraphs below; then, if neither position-spec nor screen-name is coded in the entire DISPLAY statement, a carriage return/line-feed pair will be sent to the receiving device.

4.9.1 Position-spec

For each display-item, if position-spec is specified, the cursor is positioned prior to the transfer of data for the item. position-spec is of the form:

```
(  [ LIN [ [ + ]          integer-1 ] ]   [ COL [ [ + ]          integer-3 ] ]   )
   [     [ - ]                    ]   ,   [     [ - ]                    ]
   [                      integer-2 ]   [                      integer-4 ]
```

The opening and closing parentheses and the comma and space separating the two major bracketed groups are required. Position-spec specifies the position on the console CRT screen at which the cursor will be placed. LIN and COL are COBOL special registers. Each behaves like a numeric data item with USAGE COMP, but they may be referenced by every COBOL program without being declared in the DATA DIVISION.

If LIN is specified, the cursor will be placed on the screen row whose number is equal to the value of the LIN special register, incremented (or decremented) by integer-1 if "+ integer-1" (or "- integer-1") is specified. If integer-2 is specified, the cursor will be placed on the row whose number is integer-2. If neither LIN nor integer-2 is specified, the cursor will be placed on the screen row containing the current cursor position.

If COL is specified, the cursor will be placed in the screen column whose number is equal to the value of the COL special register, incremented (or decremented) by integer-3 if "+ integer-3" (or "- integer-3") is specified. If integer-4 is specified, the cursor will be placed in the screen column whose number is integer-4. If neither COL nor integer-4 is specified, the cursor will be placed in the screen column containing the current cursor position.

4.9.2 Identifier, Literal, and ERASE

If identifier or literal is specified for a given display-item, the contents of identifier or the value of literal are sent to the receiving device. Since the data transfer occurs without conversion or reformatting, it is recommended that numeric data be moved to numeric-edited fields for purposes of DISPLAY.

If ERASE is specified and if position-spec is coded for this or a previous display-item, the console screen will be cleared from the current cursor position to the end of the screen. The initial cursor position for the next display-item will be that specified by the position-spec coded in the ERASE display-item, if present, or the position in which the cursor was left by the previous display-item. If ERASE is specified and no position-spec has been encountered up to this point in the DISPLAY statement, no action will be taken.

4.9.3 Screen-name

The DISPLAY screen-name statement causes a transfer of information from screen-name (or each elementary screen item subordinate to screen-name) to the console CRT screen. For each such screen item having a VALUE, FROM, or USING specification, the specified literal or field is the source of the displayed data. For a field having only a TO clause, the effect is as if FROM ALL "." (period) had been specified. The source data is MOVEd implicitly to a temporary item defined by the appropriate screen item's PICTURE (or by the length of the data in the case of a VALUE literal). Then an implied identifier-type DISPLAY of the constructed temporary is executed as modified by the positioning and control clauses coded in the definition of the appropriate screen item. See section 3.17 (SCREEN SECTION).

4.10 PERFORM STATEMENT

The PERFORM statement permits the execution of a separate body of program steps. Two formats of the PERFORM statement are available:

Option 1

PERFORM range [{integer / data-name} TIMES]

Option 2

PERFORM range [VARYING {index-name / data-name} FROM
amount-1 BY amount-2] UNTIL condition.

(A more extensive version of option 2 is available for varying 2 or 3 items concurrently, as explained in Appendix VI.)

In the above syntactical presentation, the following definitions are assumed:

1. Range is a paragraph-name, a section-name, or the construct procedure-name-1 THRU procedure-name-2. (THROUGH is synonymous with THRU.) If only a paragraph-name is specified, the return is after the paragraph's last statement. If only a section-name is specified, the return is after the last statement of the last paragraph of the section. If a range is specified, control is returned after the appropriate last sentence of a paragraph or section. These return points are valid only when a PERFORM has been executed to set them up; in other cases, control will pass right through.

2. The generic operands amount-1 and amount-2 may be a numeric literal, index-name, or data-name. In practice, these amount specifications are frequently integers, or data-names that contain integers, and the specified data-name is used as a subscript within the range.

In Option 1, the designated range is performed a fixed number of times, as determined by an integer or by the value of an integer data-item. If no "TIMES" phrase is given, the range is performed once. When any PERFORM has finished, execution proceeds to the next statement following the PERFORM statement.

In Option 2, the range is performed a variable number of times, in a step-wise progression, varying from an initial value of data-name = amount-1, with increments of amount-2, until a specified condition is met, at which time execution proceeds to the next statement after the PERFORM.

The condition in an Option 2 PERFORM is evaluated prior to each attempted execution of the range. Consequently, it is possible to not PERFORM the range, if the condition is not met at the outset. Similarly, in Option 1, if data-name ≤ 0, the range is not performed at all.

At run-time, it is illegal to have concurrently active PERFORM ranges whose terminus points are the same.

4.11 EXIT STATEMENT

The EXIT statement is used where it is necessary to provice an endpoint for a procedure.

The format for the EXIT statement is:

EXIT

EXIT must appear in the source program as a one-word paragraph preceded by a paragraph-name and followed by a period. An exit paragraph provides an end-point to which preceding statements may transfer control if it is decided to bypass some part of a section.

4.12 ALTER STATEMENT

The ALTER statement is used to modify a simple (non-depending) GO TO statement elsewhere in the Procedure Division, thus changing the sequence of execution of program statements.

The ALTER statement general format is:

ALTER paragraph TO [PROCEED TO] procedure-name

Paragraph (the first operand) must be a COBOL paragraph that consists of only a simple GO TO statement; the ALTER statement in effect replaces the former operand of that GO TO by procedure-name. Consider the ALTER statement in the context of the following program segment.

```
GATE.      GO TO MF-OPEN.
MF-OPEN.   OPEN INPUT MASTER-FILE.
           ALTER GATE TO PROCEED TO NORMAL.
NORMAL.    READ MASTER-FILE, AT END GO TO EOF-MASTER.
```

Examination of the above code reveals the technique of "shutting a gate," providing a one-time initializing program step.

4.13 IF STATEMENT

The IF statement permits the programmer to specify a series of procedural statements to be executed in the event a stated condition is true. Optionally, an alternative series of statements may be specified for execution if the condition is false. The general format of the IF statement is:

IF condition { statement(s)-1 / NEXT SENTENCE } [ELSE { statement(s)-2 / NEXT SENTENCE }]

The IF statement must be followed immediately by a period.

Examples of IF statements:

1. IF BALANCE = 0 GO TO NOT-FOUND.
2. IF T LESS THAN 5 NEXT SENTENCE ELSE GO TO T-1-4.
3. IF ACCOUNT-FIELD = SPACES OR NAME = SPACES ADD 1 TO SKIP-COUNT ELSE GO TO BYPASS.

The first series of statements is executed only if the designated condition is true. The second series of statements (ELSE part) is executed only if the designated condition is false. Refer to Appendix III for discussion of nested IF statements.

Regardless of whether the condition is true or false, the next sentence is executed after execution of the appropriate series of statements, unless a GO TO is contained in the imperatives that are executed, or unless the nominal flow of program steps is superseded because of an active PERFORM statement.

4.13.1 Conditions

A condition is either a simple condition or a compound condition. The four simple conditions are the relational, class, condition-name, and sign condition tests. A simple relational condition has the following structure:

operand-1 relation operand-2

where "operand" is a data-name, literal, or figurative-constant.

A compound condition may be formed by connecting two conditions, of any sort, by the logical operator AND or OR, e.g., A < B OR C = D. Refer to Appendix I for further permissible forms involving parenthesization, NOT, or "abbreviation."

The simplest "simple relations" have three basic forms, expressed by the relational symbols equal to, less than, or greater than (i.e., = or < or >).

Another form of simple relation that may be used involves the reserved word NOT, preceding any of the three relational symbols. In summary, the six simple relations in conditions are:

Relation	Meaning
=	equal to
<	less than
>	greater than
NOT =	not equal to
NOT <	greater than or equal to
NOT >	less than or equal to

It is worthwhile to briefly discuss how relation conditions can be compounded. The reserved words AND or OR permit the specification of a series of relational tests, as follows:

1. Individual relations connected by AND specify a compound condition that is met (true) only if all the individual relationships are met.

2. Individual relations connected by OR specify a compound condition that is met (true) if any one of the individual relationships is met.

The following is an example of a compound relation condition containing both AND and OR connectors. Refer to Appendix I for formal specification of evaluation rules.

```
IF X = Y AND FLAG = 'Z' OR SWITCH = 0 GO TO PROCESSING.
```

In the above example, execution will be as follows, depending on various data values.

Data Value				Does Execution Go to PROCESSING?
X	Y	FLAG	SWITCH	
10	10	'Z'	1	Yes
10	11	'Z'	1	No
10	11	'Z'	0	Yes
10	10	'P'	1	No
6	3	'P'	0	Yes
6	6	'P'	1	No

Usages of reserved word phrasings EQUAL TO, LESS THAN, and GREATER THAN are accepted equivalents of = < > respectively. Any form of the relation may be preceded by the word IS, optionally.

Before discussing class-test, sign-test, and condition-name conditions, methods of performing comparisons will be discussed.

Numeric Comparisons: The data operands are compared after alignment of their decimal positions. The results are as defined mathematically, with any negative values being less than zero, which in turn is less than any positive value. An index-name or index data item (see Chapter 6) may appear in a comparison. Comparison of any two numeric operands is permitted regardless of the formats specified in their respective USAGE clauses, and regardless of length.

Character Comparisons: Non-equal-length comparisons are permitted, with spaces being assumed to extend the length of the shorter item, if necessary. Relationships are defined in the ASCII code; in particular, the letters A-Z are in an ascending sequence, and digits are less than letters. Group items are treated simply as character strings when compared. Refer to Appendix IV for all ASCII character representations.

Returning to our discussion of simple conditions, there are three additional forms of a simple condition, in addition to the relational form, namely: class test, condition-name test, and sign test.

A class test condition has the following syntactical format:

data-name IS [NOT] { NUMERIC | ALPHABETIC }

This condition specifies an examination of the data item content to determine whether all characters are proper digit representations regardless of any operational sign (when the test is for NUMERIC), or only alphabetic or blank space characters (when the test is for ALPHABETIC). The NUMERIC test is valid only for a group, decimal, or character item (not having an alphabetic PICTURE). The ALPHABETIC test is valid only for a group or character item (PICTURE an-form).

A sign test has the following syntactical format:

data-name IS [NOT] NEGATIVE | ZERO | POSITIVE

This test is equivalent to comparing data-name to zero in order to determine the truth of the stated condition.

In a condition-name test, a conditional variable is tested to determine whether its value is equal to one of the values associated with the condition-name. A condition-name test is expressed by the following syntactical format:

condition-name

where condition-name is defined by a level 88 Data Division entry.

4.14 OPEN STATEMENT (Sequential I-O)

The OPEN statement must be executed prior to commencing file processing. The general format of an OPEN statement is:

For a sequential INPUT file, opening initiates reading the file's first records into memory, so that subsequent READ statements may be executed without waiting.

For an OUTPUT file, opening makes available a record area for development of one record, which will be transmitted to the assigned output device upon the execution of a WRITE statement. An existing file which has the same name will be superceded by the file created with OPEN OUTPUT.

An OPEN I-O statement is valid only for a DISK file; it permits use of the REWRITE statement to modify records which have been accessed by a READ statement. The WRITE statement may not be used in I-O mode for files with sequential organization. The file must exist on disk at OPEN time; it cannot be created by OPEN I-O.

When the EXTEND phrase is specified, the OPEN statement positions the file immediately following the last logical record of that file. Subsequent WRITE statements referencing the file will add records to the end of the file. Thus, processing proceeds as though the file had been opened with the OUTPUT phrase and positioned at its end. EXTEND can be used only for sequential or line sequential files.

Failure to precede (in terms of time sequence) file reading or writing by the execution of an OPEN statement is an execution-time error which will cause abnormal termination of a program run. See the COBOL- User's Guide. Furthermore, a file cannot be opened if it has been CLOSEd "WITH LOCK."

Sequential files opened for INPUT or I-O access must have been written in the appropriate format described in the User's Guide for such files.

4.15 READ STATEMENT (Secuential I-O)

The READ statement makes available the next logical data record of the designated file from the assigned device, and updates the value of the FILE STATUS data item, if one was specified. The general format of a READ statement is:

READ file-name RECORD [INTO data-name]
[AT END imperative statement].

Since at some time the end-of-file will be encountered, the user should include the AT END clause. The reserved word END is followed by any number of imperative statements, all of which are executed only if the end-of-file situation arises. The last statement in the AT END series must be followed by a period to indicate the end of the sentence. If end-of-file occurs but there is no AT END clause on the READ statement, an applicable Declarative procedure is performed. If neither AT END nor Declarative exists and no FILE STATUS item is specified for the file, the program is aborted with a run-time error.

When a data record to be read exists, successful execution of the READ statement is immediately followed by execution of the next sentence.

When more than one level-01 item is subordinate to a file description, these records share the same storage area. Therefore, the user must be able to distinguish between the types of records that are possible, in order to determine exactly which type is currently available. This is accomplished with a data comparison, using an IF statement to test a field which has a unique value for each type of record.

The INTO option permits the user to specify that a copy of the data record is to be placed into a designated data field in addition to the file's record area. The data-name must not be defined in the File Section.

Also, the INTO phrase should not be used when the file has records of various sizes as indicated by their record descriptions. Any subscripting or indexing of data-name is evaluated after the data has been read but before it is moved to data-name. Afterward, the data is available in both the file record and data-name.

In the case of a blocked input file (such as disk files), not every READ statement performs a physical transmission of data from an external storage device; instead, READ may simply obtain the next logical record from an input buffer.

If the actual record is shorter than the file record area, the file record area is padded on the right with spaces.

4.16 WRITE STATEMENT (Sequential I-O)

The general format of a WRITE statement is:

```
WRITE record-name [FROM data-name-1]

[ {AFTER  } ADVANCING {operand LINE(S)} ]
[ {BEFORE}            {     PAGE      } ]

    [AT {END-OF-PAGE} imperative-statement]
        {EOP        }
```

Ignoring the ADVANCING option for the moment, we proceed to explain the main functions of the WRITE statement.

In COBOL, file output is achieved by execution of the WRITE statement. Depending on the device assigned, "written" output may take the form of printed matter or magnetic recording on a floppy disk storage medium. The associated file must be open in the OUTPUT or I-O mode at time of execution of a WRITE statement.

Record-name must be one of the level 01 records defined for an output file, and may be qualified by the filename. The execution of the WRITE statement releases the logical record to the file and updates its FILE STATUS item, if one is defined.

If the data to be output has been developed in Working-Storage or in another area (for example, in an input file's record area), the FROM suffix permits the user to stipulate that the designated data (data-name-1) is to be copied into the record-name area and then output from there. Record-name and data-name-1 must refer to separate storage areas.

When an attempt is made to write beyond the externally defined boundaries of a sequential file, a Declarative procedure will be executed (if available) and the FILE STATUS (if available) will indicate a boundary violation. If neither is available, a fatal runtime error occurs.

The ADVANCING option is restricted to line printer output files, and permits the programmer to control the line spacing on the paper in the printer. Operand is either an unsigned integer literal or data-name; values from 0 to 120 are permitted:

Integer	Carriage Control Action
0	No spacing
1	Normal single spacing
2	Double spacing
3	Triple spacing
.	.
.	.
.	.

Single spacing (i.e., "after advancing 1 line") is assumed if there is no BEFORE or AFTER option in the WRITE statement.

Use of the key word AFTER implies that the carriage control action precedes printing a line, whereas use of BEFORE implies that writing precedes the carriage control action. If PAGE is specified, the data is printed BEFORE or AFTER the printer is repositioned to the next physical page. However, if a LINAGE clause is associated with the file, the repositioning is to the first line that can be written on the next logical page as specified in the LINAGE clause.

If the END-OF-PAGE phrase is specified, the LINAGE clause must be specified in the file description entry for the associated file. EOP is equivalent to END-OF-PAGE.

An end-of-page condition is reached whenever a WRITE statement with the END-OF-PAGE phrase causes printing or spacing within the footing area of a page body. This occurs when such a WRITE statement causes the LINAGE-COUNTER to equal or exceed the value specified by the FOOTING value, if specified. In this case, after the WRITE statement is executed, the imperative statement in the END-OF-PAGE phrase is executed.

A "page overflow" condition is reached whenever a WRITE statement cannot be fully accommodated within the current page body. This occurs when a WRITE statement would cause the LINAGE-COUNTER to exceed the value specified as the size of the page body in the LINAGE clause. In this case, the record is printed before or after (depending on the phrase used) the printer is repositioned to the first line of the next logical page. The imperative statement in the END-OF-PAGE clause, if specified, is executed after the record is written and the printer has been repositioned.

Clearly, if no FOOTING value is specified in the LINAGE clause, or if the end-of-page and overflow conditions occur simultaneously, then only the overflow condition is effective.

4.17 CLOSE STATEMENT (Sequential I-O)

Upon completion of the processing of a file, a CLOSE statement must be executed, causing the system to make the proper disposition of the file. Whenever a file is closed, or has never been opened, READ, REWRITE, or WRITE statements cannot be executed properly; a runtime error would occur, aborting the run.

The general format of the CLOSE statement is:

CLOSE {file-name [WITH LOCK]} ...

If the LOCK phrase is used, the runtime system will cause subsequent OPENs of the file to fail during the current job. If LOCK is not specified immediately after a file-name, then that file may be re-OPENed later in the program, if the program logic dictates the necessity.

An attempt to execute a CLOSE statement for a file that is not currently open is a runtime error, and causes execution to be discontinued.

Examples of CLOSE statements:

```
CLOSE MASTER-FILE-IN WITH LOCK, WORK-FILE;
CLOSE PRINT-FILE, TAX-RATE-FILE, JOB-PARAMETERS WITH LOCK
```

4.18 REWRITE STATEMENT (Sequential I-O)

The REWRITE statement replaces a logical record on a sequential disk file. The general format is:

REWRITE record-name [FROM data-name]

Record-name is the name of a logical record in the File Section of the Data Division and may be qualified. Record-name and data-name must refer to separate storage areas.

At the time of execution of this statement, the file to which record-name belongs must be open in the I-O mode (see OPEN, Section 4.14).

If a FROM phrase is included in this statement, the effect is as if MOVE data-name TO record-name were executed just prior to the REWRITE.

Execution of REWRITE replaces the record that was accessed by the most recent READ statement; said prior READ must have been completed successfully. If the record which is rewriting the record in the file is longer than the file's record, only as many bytes as will fit are actually rewritten. On the other hand, if the record which is rewriting the record in the file is shorter than the file's record, unpredictable information will be written after the record, until the beginning of the next record in the file.

4.19 GENERAL NOTE ON I/O ERROR HANDLING

If an I/O error occurs, the file's FILE STATUS item, if one exists, is set to the appropriate two-character code, otherwise it assumes the value "00".

If an I/O error occurs and is of the type that is pertinent to an AT END or INVALID KEY clause, then the imperative statements in such a clause, if present on the statement that gave rise to the error, are executed. But, if there is not an appropriate clause (such clauses may not appear on Open or Close, for example, and are optional for other I/O statements), then the logic of program flow is as follows:

1. If there is an associated Declaratives ERROR procedure (see Section 9), it is performed automatically; user-written logic must determine what action is taken because of the existence of the error. Upon return from the ERROR procedure, normal program flow to the next sentence (following the I/O statement) is allowed.

2. If no Declaratives ERROR procedure is applicable but there is an associated FILE STATUS item, it is presumed that the user may base actions upon testing the STATUS item, so normal flow to the next sentence isallowed.

Only if none of the above (INVALID KEY/AT END clause, Declaratives ERROR procedure, or testable FILE STATUS item) exists, then the run-time error handler receives control; the location of the error (source program line number) is displayed on the console, and the run is terminated "abnormally."

These remarks apply to processing of any file, whether organization is sequential, line sequential, indexed or relative.

4.20 STRING STATEMENT

The STRING statement allows concatenation of multiple sending data item values into a single receiving item. The general format of this statement is:

```
STRING { operand-1...   DELIMITED BY { operand-2 } } ...
                                     { SIZE      }
INTO identifier-1 [WITH POINTER identifier-2]
    [ON OVERFLOW imperative-statement]
```

In this format, the term operand means a non-numeric literal, one-character figurative constant, or data-name. Identifier-1 is the receiving data-item name, which must be alphanumeric without editing symbols or the JUSTIFIED clause. Identifier-2 is a counter and must be an elementary numeric integer data item of sufficient size (plus 1) to point to character positions within identifier-1.

If no POINTER phrase exists, the default value of the logical pointer is one. The logical pointer value designates the beginning position of the receiving field into which data placement begins. During movement to the receiving field, the criteria for termination of an individual source are controlled by the "DELIMITED BY" phrase:

DELIMITED BY SIZE: the entire source field is moved (unless the receiving field becomes full)

DELIMITED BY operand-2: the character string specified by operand-2 is a search pattern which, if found to match a contiguous sequence of sending characters, terminates the function for the current sending operand (and causes automatic switching to the next sending operand, if any). The matching characters in the sending fields are not moved to identifier-1.

If at any point the logical pointer (which is automatically incremented by one for each character stored into identifier-1) is less than one or greater than the size of identifier-1, no further data movement occurs, and the imperative statement given in the OVERFLOW phrase (if any) is executed. If there is no OVERFLOW phrase, control is transferred to the next executable statement.

There is no automatic space fill into any position of identifier-1. That is, unaccessed positions are unchanged upon completion of the STRING statement.

Upon completion of the STRING statement, if there was a POINTER phrase, the resultant value of identifier-2 equals its original value plus the number of characters moved during execution of the STRING statement.

4.21 UNSTRING STATEMENT

The UNSTRING statement causes data in a single sending field to be separated into subfields that are placed into multiple receiving fields. The general format of the statement is:

```
UNSTRING identifier-1

[DELIMITED BY [ALL] operand-1 [OR [ALL] operand-2] ...]

INTO {identifier-2  [DELIMITER IN identifier-3]
                    [COUNT IN identifier-4]} ...

[WITH POINTER identifier-5]
[TALLYING IN identifier-6]
[ON OVERFLOW imperative-statement]
```

Criteria for separation of subfields may be given in the "DELIMITED BY" phrase. Each time a succession of characters matches one of the non-numeric literals, one-character figurative constants, or data-item values named by operand-i, the current collection of sending characters is terminated and moved to the next receiving field specified by the INTO-clause. When the ALL phrase is specified, more than one contiguous occurrence of operand-i in identifier-1 is treated as one occurrence. The delimiting string is not moved into the current receiving field.

When two or more delimiters exist, an 'OR' condition exists. Each delimiter is compared to the sending field in the order specified in the UNSTRING statement.

Identifier-1 must be a group or character string (alphanumeric) item. When a data-item is employed as any operand-i, that operand must also be a group or character string item.

Receiving fields (identifier-2) may be any of the following types of items:

1. an unedited alphabetic item
2. a character-string (alphanumeric) item
3. a group item
4. an external decimal item (numeric, usage DISPLAY) whose PICTURE does not contain any P character.

When any examination encounters two contiguous delimiters, the current receiving area is either space or zero filled depending on its type. If there is a "DELIMITED BY" phrase in the UNSTRING statement, then there may be "DELIMITER IN" phrases following any receiving item (identifier-2) mentioned in the INTO clause. In this case, the character(s) that delimit the data moved into identifier-2 are themselves stored in identifier-3, which should be an alphanumeric item. Furthermore, if a "COUNT IN" phrase is present, the number of characters that were moved into identifier-2 is moved to identifier-4, which must be an elementary numeric integer item.

If there is a "POINTER" phrase, then identifier-5 must be an integer numeric item, and its initial value becomes the initial logical pointer value (otherwise, a logical pointer value of one is assumed). The examination of source characters begins at the position in identifier-1 specified by the logical pointer; upon completion of the UNSTRING statement, the final logical pointer value will be copied back into identifier-5.

If at any time the value of the logical pointer is less than one or exceeds the size of identifier-1, then overflow is said to occur and control passes over to the imperative statements given in the "ON OVERFLOW" clause, if any.

Overflow also occurs when all receiving fields have been filled prior to exhausting the source field.

During the course of source field scanning (looking for matching delimiter sequences), a variable length character string is developed which, when completed by recognition of a delimiter or by acquiring as many characters as the size of the current receiving field can hold, is then moved to the current receiving field in the standard MOVE fashion.

If there is a "TALLYING IN" phrase, identifier-6 must be an integer numeric item. The number of receiving fields acted upon, plus the initial value of identifier-6, will be produced in identifier-6 upon completion of the UNSTRING statement.

Any subscripting or indexing associated with identifer-1, 5, or 6 is evaluated only once at the beginning of the UNSTRING statement. Any subscripting associated wth operands-i or identifier-2, 3, 4 is evaluated immediately before access to the data item.

4.22 DYNAMIC DEBUGGING STATEMENTS

The execution TRACE mode may be set or reset dynamically. When set, procedure-names are printed on the user's console in the order in which they are executed.

Execution of the READY TRACE statements sets the trace mode to cause printing of every section and paragraph name each time it is entered. The RESET TRACE statement inhibits such printing. A printed list of procedure-names in the order of their execution is invaluable in detection of a program malfunction; it aids in determination of the point at which actual program flow departed from the expected program flow.

Another debugging feature may be required in order to reveal critical data values at specifically designated points in the procedure. The EXHIBIT statement provides this facility.

The statement form

EXHIBIT NAMED { [position-spec] { identifer / literal / ERASE } } ...[UPON mnemonic-name]

produces a printout of values of the indicated literal, or data items in the format data-name = value. position-spec and the UPON phrase have the same effect as in the DISPLAY statement.

Statements EXHIBIT. READY TRACE and RESET TRACE are extensions to ANS-74 standard COBOL designed to provide a convenient aid to program debugging.

Programming Note: It is often desirable to include such statements on source lines that contain D in column 7, so that they are ignored by the compiler unless WITH DEBUGGING MODE is included in the SOURCE-COMPUTER paragraph.

CHAPTER 5

Inter-Program Communication

Separately compiled COBOL program modules may be combined into a single executable program. Inter-program communication is made possible through the use of the Linkage Section of the Data Division (which follows the Working-Storage Section) and by the CALL statement and the USING list appendage to the Procedure Division header of a subprogram module. The Linkage Section describes data made available in memory from another program module. Record description entries in the LINKAGE section provide data-names by which data areas reserved in memory by other programs may be referenced. Entries in the LINKAGE section do not reserve memory areas because the data is assumed to be present elsewhere in memory, in a CALLing program.

Any record description clause may be used to describe items in the Linkage Section as long as the VALUE clause is not specified for other than level 88 items.

The program CHAINing facility allows a COBOL program to transfer control to any other executable program and, optionally, to pass data items as parameters to the CHAINed program.

5.1 CALL STATEMENT

The CALL statement format is

CALL literal [USING data-name ...]

Literal is a subprogram name defined as the PROGRAM-ID of a separately compiled program, and is non-numeric. Data names in the USING list are made available to the called subprogram by passing addresses to the subprogram; these addresses are assigned to the Linkage Section items declared in the USING list of that subprogram. Therefore the number of data-names specified in matching CALL and Procedure Division USING lists must be identical. Information passing conventions at the machine language level are described in the COBOL User's Guide.

NOTE

Correspondence between caller and callee lists is by position, not by identical spelling of names.

5.2 EXIT PROGRAM STATEMENT

The EXIT PROGRAM statement, appearing in a called subprogram, causes control to be returned to the next executable statement after CALL in the calling program. This statement must be a paragraph by itself.

5.3 CHAIN STATEMENT

The CHAIN statement is coded according to the following format:

$$\text{CHAIN} \left\{ \begin{array}{l} \text{literal} \\ \\ \text{identifier-1} \end{array} \right\} \text{[USING identifier-2...]}$$

Literal and identifier-1 must be alphanumeric, and identifier-1 must contain a terminating space. Each occurrence of identifier-2 must be defined in the WORKING-STORAGE or LINKAGE SECTION or in the record area of a file open at the time the CHAIN statement is executed.

When the CHAIN statement is executed, the value of literal or identifier-1, up to but not including the first space encountered (or the end of the literal), is interpreted as the name of an executable program file in the format of the appropriate operating system. The named program is loaded into memory and executed. All program and data structures of the CHAINing program are permanently destroyed except that the USING clause may be used to transfer parameters to the CHAINed program. See section 5.4 (PROCEDURE DIVISION Header with CALL and CHAIN).

The CHAINed program need not be a COBOL program. If it is, it must be a main program.

5.4 PROCEDURE DIVISION HEADER WITH CALL AND CHAIN

The PROCEDURE DIVISION header of a main program is written as:

PROCEDURE DIVISION [CHAINING data-name-1...].

The PROCEDURE DIVISION header of a subprogram is written as:

PROCEDURE DIVISION USING [data-name-2...].

The various forms of the PROCEDURE DIVISION header describe the linkage and parameter initialization requirements of a program. A main program must be linked by itself or with any number of subprograms. It may then be run independently or invoked by the execution of a CHAIN statement in another program. A subprogram must be linked with exactly one main program and, optionally, any number of other subprograms. It may only be executed by the action of a CALL statement. For a description of the linking process, see the COBOL User's Guide.

A CHAINed or CALLed program should have a CHAINing list or non-empty USING list if and only if the invoking CHAIN or CALL statement has a USING list. Furthermore, the numbers of entries in the lists should be equal, and positionally corresponding entries in the two lists should reference data items of the same size and USAGE. Failure to conform to these rules will not be diagnosed and will cause unpredictable results at runtime.

The values of the data items named in the PROCEDURE DIVISION header are established at program initialization time by using the contents of positionally corresponding data items named in the invoking CALL or CHAIN statement. In the case of CALL, the identification is made by passing pointers. Therefore, if the value of a data item named in a PROCEDURE DIVISION USING clause is changed during subprogram execution, the corresponding data item in the CALLing program will reflect the change after control is returned from the subprogram.

For a description of the formats in which parameters are passed by the CALL and CHAIN statements, see the COBOL User's Guide.

CHAPTER 6

Table Handling by the Indexing Method

In addition to the capabilities of subscripting described in Chapter 3, COBOL provides the indexing method of table handling.

6.1 INDEX NAMES AND INDEX ITEMS

An index name is declared not by the usual method of level number, name, and data description clauses, but implicitly by appearance in the "INDEXED BY index-name" appendage to an OCCURS clause. An index-name must be unique.

An index data item is an item defined by the USAGE IS INDEX phrase. An index data item must not have a PICTURE. An index name or index data item may only be referred to by a SET or SEARCH statement, a CALL statement's USING list or a Procedure header USING list; or used in a relation condition or as the variation item in a PERFORM VARYING statement, or in place of a subscript. In all cases the process is equivalent to dealing with a binary word integer subscript. Index-name must be initialized to some value before use via SET, SEARCH or PERFORM.

6.2 SET STATEMENT

The SET statement permits the manipulation of index-names, index items, or binary subscripts for table-handling purposes. There are two formats.

Format 1:

```
SET   { index-name-1 }  ... TO  { index-name-2 }
      { index-item-1 }          { index-item-2 }
      { data-name-1  }          { data-name-2  }
                                { integer-2    }
```

Format 2:

```
SET   index-name-3  ...  { UP BY   }  { data-name-4 }
                         { DOWN BY }  { integer-4   }
```

Format 1 is equivalent to moving the "TO" value (e.g., integer-2) to multiple receiving fields written immediately after the verb SET.

Format 2 is equivalent to reduction (DOWN) or increase (UP) applied to each of the quantities written immediately after the verb SET: the amount of the reduction or increase is specified by a name or value immediately following the word BY.

In any SET statement, data-names are restricted to integer items.

6.3 RELATIVE INDEXING

A user reference to an item in a table controlled by an OCCURS clause is expressed with a proper number of subscripts (or indices), separated by commas. The whole is enclosed in matching parentheses, for example:

```
TAX-RATE (BRACKET, DEPENDENTS)
XCODE (I, 2)
```

where subscripts are ordinary integer decimal data-names, or integer constants, or binary integer (COMPUTATIONAL or INDEX) items, or index-names. Subscripts may be qualified, but not, themselves, subscripted. A subscript may be signed, but if so, it must be positive. The lowest acceptable value is 1, pointing to the first element of a table. The highest permissible value is the maximum number of occurrences of the item as specified in its OCCURS clause.

A further capability exists, called relative indexing. In this case, a "subscript" is expressed as

name ± integer constant

where a space must be on either side of the plus or minus, and "name" may be any proper index-name.

Example:

```
XCODE (I + 3, J - 1).
```

6.4 SEARCH STATEMENT -- Format 1

A linear search of a table may be done using the SEARCH statement. The general format is:

SEARCH table [VARYING identifier | index-name]

[AT END imperative-statement-1]

{WHEN Condition-1 {NEXT SENTENCE / imperative-statement-2}} ...

Table is the name of a data-item having an OCCURS clause that includes an INDEXED BY list; table must be written without subscripts or indexes because the nature of the SEARCH statement causes automatic variation of an index-name associated with a particular table.

There are four possible VARYING cases:

1. NO VARYING phrase -- the first-listed index-name for the table is varied.

2. VARYING index-name-in-a-different-table -- the first-listed index-name in the table's definition is varied, implicitly, and the index-name listed in the VARYING phrase is varied in like manner, simultaneously.

3. VARYING index-name-defined-for-table -- this specific index-name is the only one varied.

4. VARYING integer-data-item-name -- both this data-item and the first-listed index-name for table are varied, simultaneously.

The term variation has the following interpretation:

1. The initial value is assumed to have been established by an earlier statement such as SET.

2. If the initial value exceeds the maximum declared in the applicable OCCURS clause, the SEARCH operation terminates at once; and if an AT END phrase exists, the associated imperative statement-1 is executed.

3. If the value of the index is within the range of valid indexes (1,2,... up to and including the maximum number of occurrences), then each WHEN-condition is evaluated until one is true or all are found to be false. If one is true, its associated imperative statement is executed and the SEARCH operation terminates. If none is true, the index is incremented by one and step (3) is repeated. Note that incrementation of index applies to whatever item and/or index is selected according to rules 1-4.

If the table is subordinate to another table, an index-name must be associated with each dimension of the entire table via INDEXED BY phrases in all the OCCURS clauses. Only the index-name of the SEARCH table is varied (along with another VARYING index-name or data-item). To search an entire two- or three-dimensional table, a SEARCH must be executed several times with the other index-names set appropriately each time, probably with a PERFORM, VARYING statement.

The logic of a Format 1 SEARCH is depicted on page 84.

6.5 SEARCH STATEMENT -- Format 2

Format 2 SEARCH statements deal with tables of ordered data. The general format of such a SEARCH ALL statement is:

SEARCH ALL table [AT END imperative-statement-1...]

WHEN condition { imperative-statement-2... | NEXT SENTENCE }

Only one WHEN clause is permitted, and the following rules apply to the condition:

1. Only simple relational conditions or condition-names may be employed, and the subject must be properly indexed by the first index-name associated with table (along with sufficient other indexes if multiple OCCURS clauses apply). Furthermore, each subject data-name (or the data-name associated with condition-name) in the condition must be mentioned in the KEY clause of the table. The KEY clause is an appendage to the OCCURS clause having the following format:

   ASCENDING | DESCENDING KEY IS data-name ...

   where data-name is the name defined in this Data Description entry (following level number) or one of the subordinate data-names. If more than one data-name is given, then all of them must be the names of entries subordinate to this group item. The KEY phrase indicates that the repeated data is arranged in ascending or descending order according to the data-names which are listed (in any given KEY phrase) in decreasing order of significance. More than one KEY phrase may be specified.

2. In a simple relational condition, only the equality test (using relation = or IS EQUAL TO) is permitted.

3. Any condition-name variable (Level 88 items) must be defined as having only a single value.

4. The condition may be compounded by use of the Logical connector AND, but not OR.

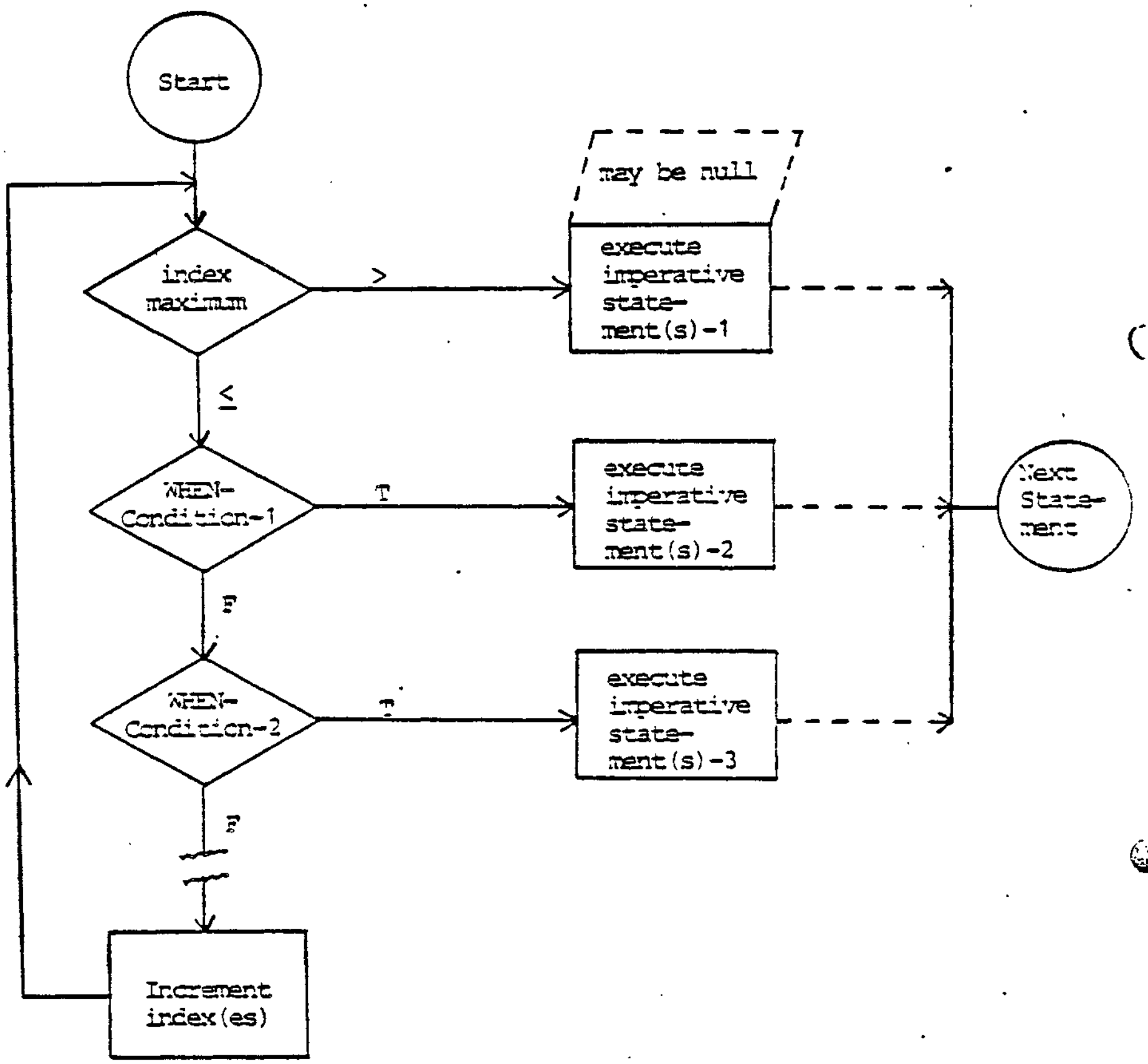
5. In a simple relational condition, the object (to the right of the equal sign) may be a literal or an identifier; the identifier must NOT be referenced in the KEY clause of the table or be indexed by the first index-name associated with the table. (The term identifier means data-name, including any qualifiers and/or subscripts or indexes.)

Failure to conform to these restrictions may yield unpredictable results. Unpredictable results also occur if the table data is not ordered in conformance to the declared KEY clauses, or if the keys referenced in the WHEN-condition are not sufficient to identify a unique table element.

In a Format 2 SEARCH, a nonserial type of search operation may take place, relying upon the declared ordering of data. The initial setting of the index-name for table is ignored and its setting is varied automatically during the searching, always within the bounds of the maximum number of occurrences. If the condition (WHEN) cannot be satisfied for any valid index value, control is passed to imperative-statement-1, if the AT END clause is present, or to the next executable sentence in the case of no AT END clause.

If all the simple conditions in the single WHEN-condition are satisfied, the resultant index value indicates an occurrence that allows those conditions to be satisfied, and control passes to imperative-statement-2. Otherwise the final setting is not predictable.

Logic Diagram for Format 1 SEARCH



CHAPTER 7

Indexed Files

7.1 DEFINITION OF INDEXED FILE ORGANIZATION

An indexed-file organization provides for recording and accessing records of a data file by keeping a directory (called the control index) of pointers that enable direct location of records having particular unique key values. An indexed file must be assigned to DISK in its defining SELECT sentence.

A file whose organization is indexed can be accessed either sequentially, dynamically or randomly.

Sequential access provides access to data records in ascending order of RECORD KEY values.

In the random access mode, the order of access to records is controlled by the programmer. Each record desired is accessed by placing the value of its key in a key data item prior to an access statement.

In the dynamic access mode, the programmer's logic may change from sequential access to random access, and vice versa, at will.

7.2 SYNTAX CONSIDERATIONS

In the Environment Division, the SELECT entry must specify ORGANIZATION IS INDEXED. and the ACCESS clause format is

ACCESS MODE IS SEQUENTIAL | RANDOM | DYNAMIC.

Assign, Reserve, and File Status clause formats are identical to those specified in Section 2.2.1 of this manual.

In the FD entry for an INDEXED file, both LABEL RECORDS STANDARD and a VALUE OF FILE-ID clause must appear. The formats of Section 3.13 apply, except that only the DISK-related forms are applicable.

7.2.1 RECORD KEY CLAUSE

The general format of this clause, which is required, is:

RECORD KEY IS data-name-1

where data-name-1 is an item defined within the record descriptions of the associated file description, and is a group item or an elementary alphanumeric item. The maximum key length is 60 bytes and the key should never be made to contain all nulls.

If random access mode is specified, the value of data-name-1 designates the record to be accessed by the next DELETE, READ, REWRITE or WRITE statement. Each record must have a unique record key value.

7.2.2 FILE STATUS REPORTING

If a FILE STATUS clause appears in the Environment Division for an Indexed organization file, the designated two-character data item is set after every I-O statement. The following table summarizes the possible settings.

Status Data Item LEFT Character	Status Data Item RIGHT Character				
	No Further Description (0)	Structure Error (1)	Duplicate Key (2)	No Record Found (3)	Disk Space Full (4)
Successful Completion (0)	X				
At End (1)	X				
Invalid Key (2)		X	X	X	X
Permanent Error(3)	X				X
Special Cases (9)		X			

File Status '21' arises if ACCESS MODE is SEQUENTIAL when WRITEs do not occur in ascending sequence for an indexed file, or the key is altered prior to REWRITE. In an OPEN INPUT or OPEN I-O statement, a File Status of '30' means 'File Not Found.' File Status '91' occurs on an OPEN INPUT or OPEN I-O statement for a relative or indexed file whose structure has been destroyed (for example, by a system crash during output to the file). When this status is returned on an OPEN INPUT, the file is considered to be open, and READs may be executed. On an OPEN I-O, however, the file is not considered to be open, and all I/O operations fail. The other settings are self-explanatory.

Note that "Disk Space Full" occurs with Invalid Key (2) for indexed and relative file handling, whereas it occurred with "Permanent Error" (3) for sequential files.

If an error occurs at execution time and no AT END or INVALID KEY statements are given and no apropriate Declarative ERROR section is supplied and no FILE STATUS is specified, the error will be displayed on the console and the program will terminate. See Section 4.19.

7.3 PROCEDURE DIVISION STATEMENTS FOR INDEXED FILES

The syntax of the sequential file OPEN statement (Section 4.14) also applies to indexed organization files, except that EXTEND is not applicable.

The following table summarizes the available statement types and their permissibility in terms of ACCESS mode and OPEN option in effect. Where X appears, the statement is permissible, otherwise it is not valid under the associated ACCESS mode and OPEN option.

ACCESS MODE IS	Procedure Statement	OPEN Option in Effect		
		Input	Output	I-O
SEQUENTIAL	READ	X		X
	WRITE		X	
	REWRITE			X
	START	X		X
	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

In addition to the above statements, CLOSE is permissible under all conditions; the same format shown in Section 4.17 is used.

7.4 READ STATEMENT

Format 1 (Sequential Access):

READ file-name [NEXT] RECORD [INTO data-name-1]

[AT END imperative-statement ...]

Format 2 (Random or Dynamic Access):

READ file-name RECORD [INTO data-name-1] [KEY IS data-name-2]

[INVALID KEY imperative-statement...]

Format 1 without NEXT must be used for all files having SEQUENTIAL ACCESS mode. Format 1 with the NEXT option is used for sequential READs of a DYNAMIC access mode file. The AT END clause is executed when the logical end-of-file condition arises. If this clause is not written in the source statement, an appropriately assigned Declaratives ERROR section is given control at end-of-file time, if available.

Format 2 is used for files in random-access mode or for files in dynamic-access mode when records are to be retrieved randomly.

In format 2, the INVALID KEY clause specifies action to be taken if the access key value does not refer to an existing key in the file. If the clause is not given, the appropriate Declaratives ERROR section, if supplied, is given control.

The optional KEY IS clause must designate the record key item declared in the file's SELECT entry. This clause serves as documentation only. The user must ensure that a valid key value is in the designated key field prior to execution of a random-access READ.

The rules for sequential files regarding the INTO phrase apply here as well.

7.5 WRITE STATEMENT

The WRITE statement releases a logical record for an output or input-output file; its general format is:

```
WRITE record-name [FROM data-name-1]
    [INVALID KEY imperative-statement...]
```

Just prior to executing the WRITE statement, a valid (unique) value must be in that portion of the record-name (or data-name-1 if FROM appears in the statement) which serves as RECORD KEY.

In the event of an improper key value, the imperative statements are executed if the INVALID KEY clause appears in the statement; otherwise an appropriate Declaratives ERROR section is invoked, if applicable. The INVALID KEY condition arises if:

1. for sequential access, key values are not ascending from one WRITE to the next WRITE;
2. the key value is not unique;
3. the allocated disk space is exceeded.

7.6 REWRITE STATEMENT

The REWRITE statement logically replaces an existing record; the format of the statement is:

```
REWRITE record-name [FROM data-name]
    [INVALID KEY imperative-statement...]
```

For a file in sequential-access mode, the last READ statement must have been successful in order for a REWRITE statement to be valid. If the value of the record key in record-name (or corresponding part of data-name, if FROM appears in the statement) does not equal the key value of the immediately previous READ, then the invalid key condition exists and the imperative statements are executed, if present; otherwise an applicable Declaratives ERROR section is executed, if available.

For a file in a random or dynamic access mode, the record to be replaced is specified by the record key; no previous READ is necessary. The INVALID KEY condition exists when the record key's value does not equal that of any record stored in the file.

7.7 DELETE STATEMENT

The DELETE statement logically removes a record from an indexed file. The general format of the statement is:

DELETE file-name RECORD [INVALID KEY imperative-statement...]

For a file in the sequential access mode, the last input-output statement executed for file-name must have been a successful READ statement. The record that was read is deleted. Consequently, no INVALID KEY phrase should be specified for sequential-access mode files.

For a file having random or dynamic access mode, the record deleted is the one associated with the record key; if there is no such matching record, the invalid key condition exists, and control passes to the imperative statements in the INVALID KEY clause, or to an applicable Declarative ERROR section if no INVALID KEY clause exists.

7.8 START STATEMENT

The START statement enables an indexed organization file to be positioned for reading at a specified key value. This is permitted for files open in either sequential or dynamic access modes. The format of this statement is:

START file-name [KEY IS {GREATER THAN / NOT LESS THAN / EQUAL TO} data-name]

[INVALID KEY imperative statement...]

Data-name must be the declared record key and the value to be matched by a record in the file must be pre-stored in the data-name. When executing this statement, the file must be open in the input or I-O mode.

If the KEY phrase is not present, equality between a record in the file and the record key value is sought. If key relation GREATER or NOT LESS is specified, the file is positioned for next access at the first record greater than, or greater than or equal to, the indicated key value.

If no matching record is found, the imperative statements in the INVALID KEY clause are executed, or an appropriate Declaratives ERROR section is executed.

CHAPTER 8

Relative Files

8.1 DEFINITION OF RELATIVE FILE ORGANIZATION

Relative organization is restricted to disk files. Records are differentiated on the basis of a relative record number which ranges from 1 to 32,767, or to a lesser maximum for a smaller file. Unlike the case of an indexed file, where the identifying key field occupies a part of the data record, relative record numbers are conceptual and are not embedded in the data records.

A relative organization file may be accessed either sequentially, dynamically or randomly. In sequential access mode, records are accessed in the order of ascending record numbers.

In random access mode, the secuence of record access is controlled by the program, by placing a number in a relative key item. In dynamic access mode, the program may inter-mix random and sequential access at will.

8.2 SYNTAX CONSIDERATIONS

In the Environment Division, the SELECT entry must specify ORGANIZATION IS RELATIVE, and the ACCESS clause format is

ACCESS MODE IS SEQUENTIAL | RANDOM | DYNAMIC.

Assign, Reserve, and File Status clause formats are identical to those used for sequential or indexed organization files. The values of STATUS Key 2 when STATUS Key 1 equals '2' are:

'2' for attempt to WRITE a duplicate key

'3' for nonexistent record

'4' for disk space full

In the associated FD entry, STANDARD labels must be declared and a VALUE OF FILE-ID clause must be included.

Thie first byte of the record area associated with a relative file should not be described as part of a COMP or COMP-3 item by any record description for the file.

8.2.1 RELATIVE KEY CLAUSE

In addition to the usual clauses in the SELECT entry, a clause of the form

RELATIVE KEY IS data-name-1

is required for random or dynamic access mode. It is also required for sequential-access mode, if a START statement exists for such a file.

Data-name-1 must be described as an unsigned binary integer item not contained within any record description of the file itself. Its value must be positive and nonzero.

8.3 PROCEDURE DIVISION STATEMENTS FOR RELATIVE FILES

Within the Procedure Division, the verbs OPEN, CLOSE, READ, WRITE, REWRITE, DELETE and START are available, just as for files whose organization is indexed. (Therefore the charts in Sections 7.2.2 and 7.3 also apply to RELATIVE files.) The statement formats for sequential file OPEN and CLOSE (see Sections 4.14 and 4.17) are applicable to relative files, except for the "EXTEND" phrase.

8.4 READ STATEMENT

Format 1:

READ file-name [NEXT] RECORD [INTO data-name]

[AT END imperative statement...]

Format 2:

READ file-name RECORD [INTO data-name]

[INVALID KEY imperative statement...]

Format 1 must be used for all files in sequential access mode. The NEXT phrase must be present to achieve sequential access if the file's declared mode of access is Dynamic. The AT END clause, if given, is executed when the logical end-of-file condition exists, or, if not given, the appropriate Declaratives ERROR section is given control, if available.

Format 2 is used to achieve random access with declared mode of access either Random or Dynamic.

If a Relative Key is defined (in the file's SELECT entry), successful execution of a format 1 READ statement updates the contents of the RELATIVE KEY item ("data-name-1") so as to contain the record number of the record retrieved.

For a format 2 READ, the record that is retrieved is the one whose relative record number is pre-stored in the RELATIVE KEY item. If no such record exists, however, the INVALID KEY condition arises, and is handled by (a) the imperative statements given in the INVALID KEY portion of the READ, or (b) an associated Declaratives section.

The rules for sequential files regarding the INTO phrase apply here as well.

8.5 WRITE STATEMENT

The format of the WRITE statement is the same for a relative file as for an indexed file:

```
WRITE record-name [FROM data-name]

    [INVALID imperative statement...]
```

If access mode is sequential, then completion of a WRITE statement causes the relative record number of the record just output to be placed in the RELATIVE KEY item.

If access mode is random or dynamic, then the user must pre-set the value of the RELATIVE KEY item in order to assign the record an ordinal (relative) number. The INVALID KEY condition arises if there already exists a record having the specified ordinal number, or if the disk space is exceeded.

8.6 REWRITE STATEMENT

The format of the REWRITE statement is the same for a relative file as for an indexed file:

```
REWRITE record-name [FROM data-name]

    [INVALID KEY imperative statement ...]
```

For a file in sequential access mode, the immediately previous action must have been a successful READ; the record thus previously made available is replaced in the file by executing REWRITE. If the previous READ was unsuccessful, a run-time error will terminate execution. Therefore, no INVALID KEY clause is allowed for sequential access.

For a file with dynamic or random access mode declared, the record that is replaced by executing REWRITE is the one whose ordinal number is pre-set in the RELATIVE KEY item. If no such item exists, the INVALID KEY condition arises.

8.7 DELETE STATEMENT

The format of the DELETE statement is the same for a relative file as for an indexed file:

DELETE file-name RECORD

[INVALID KEY imperative statement...]

For a file in a sequential access mode, the immediately previous action must have been a successful READ statement; the record thus previously made available is logically removed from the file. If the previous READ was unsuccessful, a run-time error will terminate execution. Therefore, an INVALID KEY phrase may not be specified for sequential-access mode files.

For a file with dynamic or random access mode declared, the removal action pertains to whatever record is designated by the value in the RELATIVE KEY item. If no such numbered record exists, the INVALID KEY condition arises.

8.3 START STATEMENT

The format of the START statement is the same for a relative file as for an indexed file:

START file-name [KEY IS { GREATER THAN / NOT LESS THAN / EQUAL TO } data-name-1]

[INVALID KEY imperative statement...]

Execution of this statement specifies the beginning position for reading operations; it is permissible only for a file whose access mode is defined as sequential or dynamic.

Data-name may only be that of the previously declared RELATIVE KEY item, and the number of the relative record must be stored in it before START is executed. When executing this statement, the associated file must be currently open in INPUT or I-O mode.

If the KEY phrase is not present, equality between a record in the file and the record key value is sought. If key relation GREATER or NOT LESS is specified, the file is positioned for next access at the first record greater than, or greater than or equal to, the indicated key value.

If no such relative record is found, the imperative statements in the INVALID KEY clause are executed, or an appropriate Declaratives ERROR section is executed.

CHAPTER 9

Declaratives and the Use Sentence

The Declaratives region provides a method of including procedures that are executed not as part of the sequential coding written by the programmer, but rather when a condition that cannot normally be tested by the programmer occurs.

Although the system automatically handles checking and creation of standard labels and executes error recovery routines in the case of input/output errors, additional procedures may be specified by the COBOL programmer.

Since these procedures are executed only at the time an error in reading or writing occurs, they cannot appear in the regular sequence of procedural statements. They must be written at the beginning of the Procedure Division in a subdivision called DECLARATIVES. Related procedures are preceded by a USE sentence that specifies their function. A declarative section ends with the occurrence of another section-name with a USE sentence or with the key words END DECLARATIVES.

The key words DECLARATIVES and END DECLARATIVES must each begin in Area A and be followed by a period.

```
PROCEDURE DIVISION.

DECLARATIVES.

{section-name SECTION. USE sentence.

{paragraph-name. {sentence}...} ...} ...

END DECLARATIVES.
```

The USE sentence defines the applicability of the associated section of coding.

A USE sentence, when present, must immediately follow a section header in the Declarative portion of the Procedure Division and must be followed by a period followed by a space. The remainder of the section must consist of zero, one or more procedural paragraphs that define the procedures to be used. The USE sentence itself is never executed; rather, it defines the conditions for the execution of the USE procedure. The general format of the USE sentence is

USE AFTER STANDARD EXCEPTION | ERROR PROCEDURE

ON {file-name... | INPUT | OUTPUT | I-O | EXTEND}.

The words EXCEPTION and ERROR may be used interchangeably. The associated declarative section is executed (by the PERFORM mechanism) after the standard I-O recovery procedures for the files designated, or after the INVALID KEY or AT END condition arises on a statement lacking the INVALID KEY or AT END clause. A given file-name may not be associated with more than one declarative section.

Within a declarative section there must be no reference to any nondeclarative procedure. Conversely, in the nondeclarative portion there must be no reference to procedure-names that appear in the declaratives section, except that PERFORM statements may refer to a USE statement and its procedures; but in a range specification (see PERFORM, Section 4.10) if one procedure-name is in a Declarative Section, then the other must be in the same Declarative Section.

An exit from a Declarative Section is inserted by the compiler following the last statement in the section. All logical program paths within the section must lead to the exit point.

CHAPTER 10

Segmentation

The program segmentation facility is provided to enable the execution of COBOL programs which are larger than physical memory. When segmentation is used (that is, when any section header in the program contains a segment number) the entire PROCEDURE DIVISION must be written in sections. Each section is assigned a segment number by a section header of the form:

section-name SECTION [segment-number].

segment-number must be an integer with a value in the range from 0 through 99. If segment-number is omitted, it is assumed to be 0. Declaratives sections must have segment-numbers less than 50. All sections which have the same segment number constitute a single program segment and must occur together in the source program. Furthermore, all segments with numbers less than 50 must occur together at the beginning of the PROCEDURE DIVISION.

Segments with numbers 0 through 49 are called fixed segments and are aways resident in memory during execution. Segments with numbers greater than 49 are called independent segments. Each independent segment is treated as a program overlay. An independent segment is in its initial state when control is passed to it for the first time during the execution of a program, and also when control is passed to that section (implicitly or explicitly) from another segment with a different segment number. Specifically, an independent segment is in its initial state when it is reached by "falling through" the end of a fixed or different independent segment.

Segmentation causes the following restrictions on the use of the ALTER and PERFORM statements:

1. A GO TO statement in an independent segment must not be referred to by an ALTER statement in any other segment.

2. A PERFORM statement in a fixed segment may have within its range only

   a. sections and/or paragraphs wholly contained within fixed segments, or

   b. sections and/or paragraphs wholly contained in a single independent segment.

3. A PERFORM statement in an indepencent segment may have within its range only

   a. sections and/or paragraphs wholly contained within fixed segments, or

   b. sections and/or paragraphs wholly contained within the same independent segment as the PERFORM statement.

APPENDIX I

Advanced Forms of Conditions

Evaluation Rules for Compound Conditions

1. Evaluation of individual simple conditions (relation, class, condition-name, and sign test) is done first.

2. AND-connected simple conditions are evaluated next as a single result.

3. OR and its adjacent conditions (or previously evaluated results) are then evaluated.

EXAMPLES:

1. A < B OR C = D OR E NOT > F

   The evaluation is equivalent to (A<B) OR (C=D) OR (E≤F) and is true if any of the three individual parenthesized simple conditions is true.

2. WEEKLY AND HOURS NOT = 0

   The evaluation is equivalent, after expanding level 88 condition-name WEEKLY, to

   (PAY-CODE = 'W') AND (HOURS ≠ 0)

   and is true only if both the simple conditions are true.

3. A = 1 AND B = Z AND G > -3

   OR P NOT EQUAL TO "SPAIN"

   is evaluated as

   [(A = 1) AND (B = 2) AND (G >-3)]

   OR (P ≠"SPAIN")

   If P = "SPAIN", the compound condition can only be true if all three of the following are true:

   (c.1) A = 1
   (c.2) B = 2
   (c.3) G > -3

   However, if P is not equal to "SPAIN", the compound condition is true regardless of the values of A, B and G.

Parenthesized Conditions

Parentheses may be written within a compound condition or parts thereof in order to specify precedence in the evaluation order.

Example:

```
IF A = B AND (A = 5 OR A = 1)
    PERFORM PROCEDURE-44.
```

In this case, PROCEDURE-44 is executed if A = 5 OR A = 1 while at the same time A = B. In this manner, compound conditions may be formed containing other compound conditions, not just simple conditions, via the use of parentheses.

Abbreviated Conditions

For the sake of brevity, the user may omit the "subject" when it is common to several successive relational tests. For example, the condition A = 5 OR A = 1 may be written A = 5 OR = 1. This may also be written A = 5 OR 1, where both subject and relation being implied are the same.

Another example:

```
IF A = B OR < C OR Y
```

is a shortened form of

```
IF A = B OR A < C OR A < Y
```

The interpretation applied to the use of the word 'NOT' in an abbreviated condition is:

1. If the item immediately following 'NOT' is a relational operator, then the 'NOT' participates as part of the relational operator;

2. otherwise, the beginning of a new, completely separate condition must follow 'NOT', not to be considered part of the abbreviated condition.

Caution: Abbreviations in which the subject and relation are implied are permissible only in relation tests; the subject of a sign test or class test cannot be omitted.

NOT, the Logical Negation Operator

In addition to its use as a part of a relation (e.g., IF A IS NOT = B), "NOT" may precede a condition. For example, the condition NOT (A = B OR C) is true when (A = B OR A = C) is false. The word NOT may precede a level 88 condition name, also.

APPENDIX II

Table of Permissible MOVE Operands

Source Operand	Receiving Operand in MOVE Statement					
	Numeric Integer	Numeric Non-integer	Numeric Edited	Alphanumeric Edited	Alphanumeric	Grou
Numeric Integer	OK	OK	OK	OK (A)	OK (A)	OK
Numeric Non-integer	OK	OK	OK			OK
Numeric Edited				OK	OK	OK
Alphanumeric Edited				OK	OK	OK
Alpnanumeric	OK (C)	OK (C)	OK (C)	OK	OK	OK
Group	OK (B)	OK (B)	OK (B)	OK(B)	OK (B)	OK

KEY: (A) Source sign, if any, is ignored

(B) If the source operand or the receiving operand is a Group Item, the move is considered to be a Group Move. See Section 4.3 for a discussion of the effect of a Group Move.

(C) Source is treated as an unsigned integer; source length may not exceed 31.

NOTE: No distinction is made in the compiler between alphabetic and alphanumeric; one should not move numeric items to alpnabetic items and vice versa.

APPENDIX III

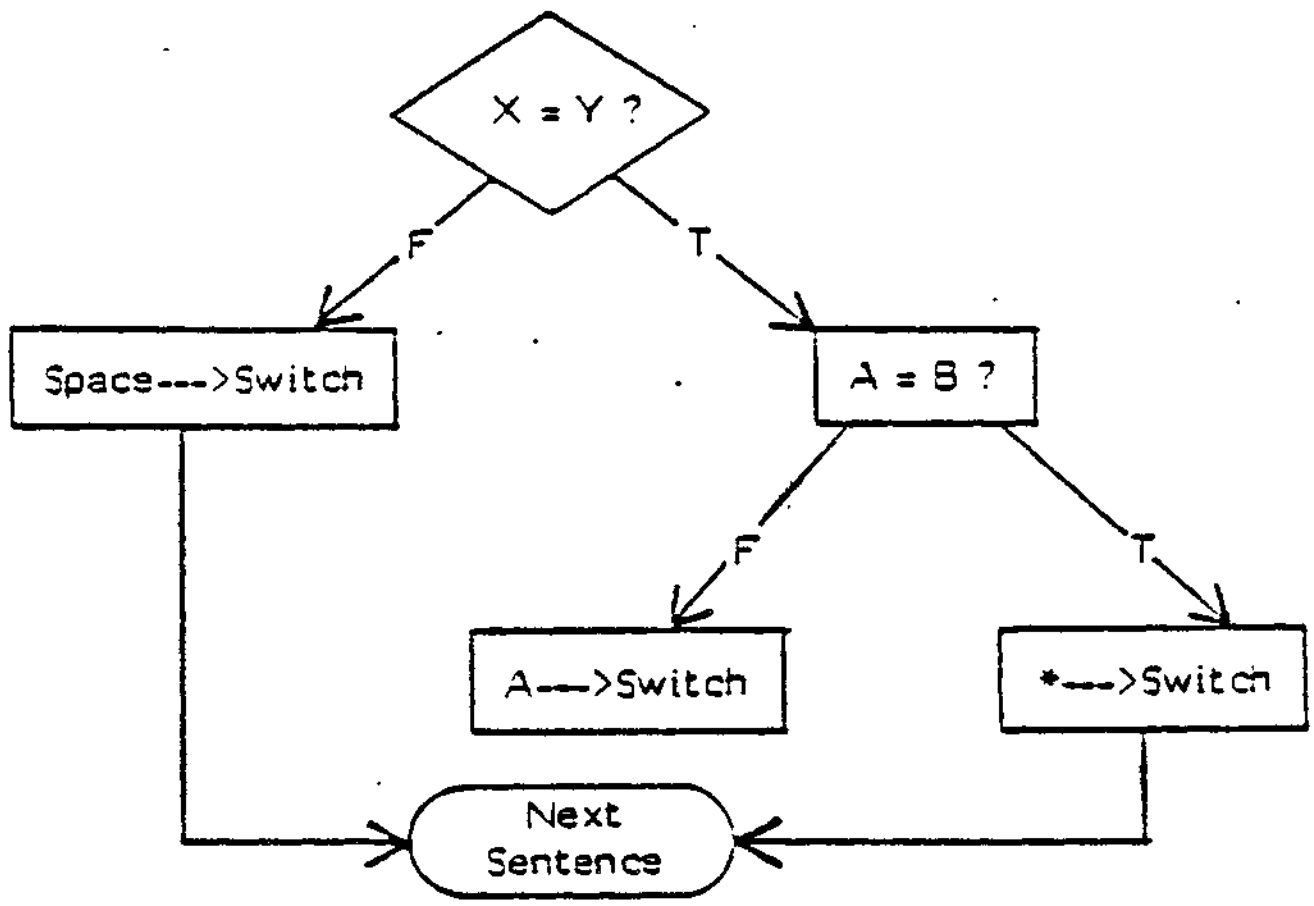
Nesting of IF Statements

A "nested IF" exists when the verb IF appears more than once in a single sentence.

Example:

```
IF X = Y
    IF A = B
        MOVE "*" TO SWITCH
    ELSE
        MOVE "A" TO SWITCH
ELSE
        MOVE SPACE TO SWITCH
```

The flow of the above sentence may be represented by a tree structure:



Another useful way of viewing nested IF structures is based on numbering IF and ELSE verbs to show their priority.

```
                    IF1     X = Y
                    |
                    |   IF2     A = B
true                |           true-action : MOVE "*" TO SWITCH
action1:            |   ELSE2 false-action2 : MOVE "A" TO SWITCH
                    |
                    ELSE1
                            false-action1 : MOVE SPACE TO SWITCH.
```

The above illustration shows clearly the fact that IF2 is wholly nested within the true-action side of IF1.

The number of ELSEs in a sentence need not be the same as the number of IFs; there may be fewer ELSE branches.

Examples:

```
IF M = 1
    IF K = 0
        GO TO M1-K0
    ELSE
        GO TO M1-KNOT0.

IF AMOUNT IS NUMERIC
    IF AMOUNT IS ZERO
        GO TO CLOSE-OUT.
```

In the latter case, IF2 could equally well have been written as AND.

APPENDIX IV

ASCII Character Set For ANS-74 COBOL

Character	Octal Value	Character	Octal Value
A	101	0	60
B	102	1	61
C	103	2	62
D	104	3	63
E	105	4	64
F	106	5	65
G	107	6	66
H	110	7	67
I	111	8	70
J	112	9	71
K	113	(SPACE)	40
L	114	"	42
M	115	$	44
N	116	' (non-ANSI)	47
O	117	(	50
P	120	)	51
Q	121	*	52
R	122	+	53
S	123	,	54
T	124	-	55
U	125	.	56
V	126	/	57
W	127	;	73
X	130	<	74
Y	131	=	75
Z	132	>	76

Plus-zero (zero with emoedded positive sign); 173
Minus-zero (zero with embedded negative sign); 175

APPENDIX V

Reserved Words

+ indicates additional words required by COBOL- for interactive screens, Debug extensions, and packed decimal format

ACCEPT	ACCESS	ADD
ADVANCING	AFTER	ALL
ALPHABETIC	ALSO	ALTER
ALTERNATE	AND	ARE
AREA(S)	ASCENDING	ASCII+
ASSIGN	AT	AUTHOR
AUTO	AUTO-SKIP+	BEEP+
BEFORE	BELL	BLANK
BLANK	BLINK	BLOCK
BOTTOM	BY	CALL
CANCEL	CD	CF
CH	CHAIN	CHAINING
CHARACTER(S)	CLOCK-UNITS	CLOSE
COBOL	CODE	CODE-SET
COL+	COLLATING	COLUMN
COLUMN	COMMA	COMMUNICATION
COMP	COMP-3+	COMPUTATIONAL
COMPUTATIONAL-3+	COMPUTE	CONFIGURATION
CONTAINS	CONTROL(S)	COPY
CORR(ESPONDING)	COUNT	CURRENCY
DATA	DATE	DATE-COMPILED
DATE-WRITTEN	DAY	DE(TAIL)
DEBUG-CONTENTS	DEBUG-ITEM	DEBUG-LINE
DEBUG-NAME	DEBUG-SUB-1	DEBUG-SUB-2
DEBUG-SUB-3	DEBUGGING	DECIMAL-POINT
DECLARATIVES	DELETE	DELIMITED
DELIMITER	DEPENDING	DESCENDING
DESTINATION	DISABLE	DISK+
DISPLAY	DIVIDE	DIVISION
DOWN	DUPLICATES	DYNAMIC
EGI	ELSE	EMI
ENABLE	END	END-OF-PAGE
ENTER	ENVIRONMENT	EOP
EQUAL	ERASE+	ERROR
ESCAPE	ESI	EVERY
EXCEPTION	EXHIBIT+	EXIT
EXTEND	FD	FILE
FILE CONTROL	FILE-ID+	FILLER
FINAL	FIRST	FOOTING
FOR	FROM	FROM
GENERATE	GIVING	GO
GREATER	GROUP	HEADING
HIGH-VALUE(S)	HIGHLIGHT	I-O
I-O-CONTROL	IDENTIFICATION	IF
IN	INDEX	INDEXED
INITIAL	INITIATE	INPUT
INPUT-OUTPUT	INSPECT	INSTALLATION
INTO	INVALID	IS

JUST(IFIED)	KEY	LABEL
LAST	LEADING	LEFT
LEFT-JUSTIFY+	LENGTH	LENGTH-CHECK+
LESS	LIMIT(S)	LIN+
LINAGE	LINAGE-COUNTER	LINE
LINE(S)	LINE-COUNTER	LINKAGE
LOCK	LOW-VALUE(S)	MEMORY
MERGE	MESSAGE	MODE
MODULES	MOVE	MULTIPLE
MULTIPLY	NAMES+	NATIVE
NEGATIVE	NEXT	NOT
NUMBER	NUMBER	NUMERIC
OBJECT-COMPUTER	OCCURS	OF
OFF	OMITTED	ON
ON	OPTIONAL	OR
ORGANIZATION	OUTPUT	OVERFLOW
PAGE	PAGE-COUNTER	PEN
PERFORM	PF	PH
PIC(TURE)	PLUS	PLUS
POINTER	POSITION	POSITIVE
PRINTER+	PRINTING	PROCEDURE(S)
PROCEED	PROGRAM	PROGRAM-ID
PROMPT+	QUEUE	QUOTE(S)
RANDOM	RD	READ
READY+	RECEIVE	RECORD(S)
REDEFINES	REEL	REFERENCES
RELATIVE	RELEASE	REMAINDER
REMOVAL	RENAMES	REPLACING
REPORT(S)	REPORTING	RERUN
RESERVE	RESET	RETURN
REVERSED	REWIND	REWRITE
RF	RH	RIGHT
RIGHT-JUSTIFY+	ROUND	RUN
SAME	SCREEN	SD
SEARCH	SECTION	SECURE
SECURITY	SEGMENT	SEGMENT-LIMIT
SELECT	SEND	SENTENCE
SEPARATE	SEQUENCE	SEQUENTIAL
SET	SIGN	SIZE
SORT	SORT-MERGE	SOURCE
SOURCE-COMPUTER	SPACE(S)	SPACE-FILL+
SPECIAL-NAMES	STANDARD	STANDARD-1
START	STATUS	STOP
STRING	SUB-QUEUE-1,2,3	SUBTRACT
SUM	SUPPRESS	SYMBOLIC
SYNC(HRONIZED)	TABLE	TALLYING
TAPE	TERMINAL	TERMINATE
TEXT	THAN	THROUGH
THRU	TIME	TIMES
TO	TOP	TRACE+
TRAILING	TRAILING-SIGN+	TYPE
UNIT	UNSTRING	UNTIL
UP	UPDATE+	UPON
USAGE	USE	USING
VALUE(S)	VARYING	WHEN
WITH	WORDS	WORKING-STORAGE
WRITE	ZERO((E)S)	ZERO-FILL+
*	**	+
-	/	<
=	>	

APPENDIX VI

PERFORM with VARYING and AFTER Clauses

```
PERFORM range

    VARYING identifier-1 FROM amount-1 BY amount-2
        UNTIL condition-1

  [ AFTER identifier-2 FROM amount-3 BY amount-4
        UNTIL condition-2

    [AFTER identifier-3 FROM amount-5 BY amount-6]
        UNTIL condition-3 ]
```

Identifier here means a data-name or index-name. Amount-1, -3, and -5 may be a data-name, index-name, or literal. Amount-2, -4, and -6 may be a data-name or literal only.

The operation of this complex PERFORM statement is equivalent to the following COBOL statements (example varying three items):

```
START-PERFORM.
    MOVE amount-1 TO identifier-1
    MOVE amount-3 TO identifier-2
    MOVE amount-5 TO identifier-3.

TEST-CONDITION-1.
    IF condition-1 GO TO END-PERFORM.

TEST-CONDITION-2.
    IF condition-2
        MOVE amount-3 TO identifier-2
        ADD amount-2 TO identifier-1
        GO TO TEST-CONDITION-1.

TEST-CONDITION-3.
    IF condition-3
        MOVE amount-5 TO identifier-3
        ADD amount-4 TO identifier-2
        GO TO TEST-CONDITION-2.

    PERFORM range
    ADD amount-6 TO identifier-3
    GO TO TEST-CONDITION-3.

END-PERFORM. Next statement.
```

NOTE

If any identifier above were an index-name, the associated MOVE would instead be a SET (TO form), and the associated ADD would be a SET (UP form).

APPENDIX VII

COBOL-
With Respect to the ANSI Standard

To understand how COBOL- is a 1974 ANSI COBOL, one must know the structure of that standard. The COBOL ANSI standard is divided into 12 "modules":

1. Nucleus
2. Table handling
3. Sequential I/O
4. Relative I/O
5. Indexed I/O
6. Interprogram communication
7. Library
8. Communication
9. Debug
10. Report-Writer
11. Segmentation
12. Sort/Merge

Each module has two defined levels of implementation, namely Level I and Level II (which is a superset of Level I). According to the standard, the first three modules in the list above should be implemented at least to Level I, but the other nine modules may or may not be implemented.

Referring to the Nucleus and Table Handling modules, Microsoft COBOL- includes all Level II features except:

I. GENERAL

1. Figurative constant ALL "lit" for literals greater than one character

2. Qualification of names is not allowed in the Environment Division.

3. Switch testing facility (actually a Level I feature)

4. Alphabet-name must be "ASCII" and cannot be defined with a literal phrase

II. DATA DIVISION

1. Occurs depending on ...

2. Level 88 having list of items intermixed with range (either list or range may be used but not both at one time)

3. COMP data items always require 2 bytes:

   -- PICTURE 9(5) only allows a range of -32768 to 32767
   -- PICTUREs 9,99,999,9999 are equivalent to PIC 9(5) for COMP items
   -- Diagnostic is given when more than 5 digits are specified

4. Unsigned COMP data items

   -- PIC 9 is equivalent to PIC S9

5. Renames phrase

III. PROCEDURE DIVISION

1. MOVE, ADD, SUBTRACT CORRESPONDING

2. Multiple destinations for results of arithmetic statements

3. Division remainders

4. Inspect Level II

5. Arithmetic expressions in conditions

6. ALTER series of procedure names

Regarding the file handling modules, CCBOL- includes all Level II facilities except Multiple Index Keys and special language for TAPE handling, that is:

1. optional tape file existence by specifying "SELECT OPTIONAL filename"

2. buffering of input/output by allowing a fully functional "RESERVE Integer AREA(S)" clause

3. multi-file tapes by specifying the "MULTIPLE FILE TAPE CONTAINS" clause

4. control over blocking of fixed and variable-length records by allowing fully functional "BLOCK CONTAINS" and "RECORD CONTAINS" clauses in the FD of tape files

5. multi-reel files, tape reversal, and tape positioning by means of fully implemented CLOSE and OPEN statements

However, the file handling modules do not include the Level I Rerun facility, because most microcomputer operating systems have no support for it.

The Interprogram Communication and Library modules are implemented to Level I.

The Debug and Report-Writer modules are not implemented at all, and Microsoft has no plans for them because they are not very widely used. However, CCBOL- does include the IBM COBOL Debug facility extensions to the ANSI standard.

Another extension Microsoft has incorporated in CCBOL- is in interactive screen control by allowing special options to the ACCEPT and DISPLAY statements. Still another extension is the COMP-3 data format which allows numeric data to be packed two digits to the byte so that mass storage requirements are reduced.

INDEX