microsoft cobol-80 documentation

Microsoft COBOL-80 and associated software are accompanied by the following documents:

1. COBOL-80 USER'S GUIDE
   describes all the procedures associated with running COBOL-80, writing COBOL programs, and running the programs with your hardware.

2. COBOL-80 REFERENCE MANUAL
   provides extensive descriptions of COBOL-80's statements, syntax and organization.

3. MICROSOFT UTILITY SOFTWARE MANUAL
   describes the use of the MACRO-80 Assembler, LINK-80 Linking Loader and LIB-80 Library Manager with the COBOL-80 compiler.

Information in this document is subject to change without notice and does not represent a commitment on the part of Microsoft. The software described in this document is furnished under a license agreement or non-disclosure agreement. The software may be used or copied only in accordance with the terms of the agreement.

(C) Microsoft, 1978

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

CP/M is a registered trade mark of Digital Research

USER'S GUIDE UPDATE NOTICE

The text file, README.DOC, is included on one of your distribution disks. README.DOC contains the most current information concerning the enhancements and extensions to Microsoft COBOL since the last release.

Microsoft Corporation

UPDATE NOTICE
for Microsoft COBOL Compiler
Release 4.66

This update (June 10, 1983) describes the differences between the Microsoft COBOL Compiler, Release 4.66, for CP/M-80 operating systems (including SoftCard) and the previous Release 4.65.

Release 4.66 contains enhancements and corrections which are described in the following sections.

Important

If you are using the MS-COBOL-Hosted Version of the Microsoft SORT Sorting and Merging Facility with MS-COBOL Compiler, Release 4.66, an update is also required for your MS-SORT package. For information on how to receive this update, contact:

Product Update Administrator
Microsoft Corporation
10700 Northup Way
Bellevue, Washington 98004
(206) 828-8080

Enhancements

1. Uppercase and lowercase characters are treated the same unless the characters are inside a non-numeric (quoted) literal.

2. End-of-source-file processing has been improved. Blank lines at the end of the file will no longer generate error messages. Files with last records that have not been terminated with a carriage

return will now have the last record processed.

3. Error handling for non-numeric (quoted) literals has been improved so the compiler recovers more quickly from such an error.

   However, non-numeric literals that are broken across more than one line, but do not use a continuation character, now generate an error message.

Corrections

1. In Release 4.65, when a COMPUTE statement with a ROUNDED option was the last or only statement in a paragraph, and multiplication was performed on the statement, the result may have been ten times too great.

   This has been corrected in this Release 4.66.

   No other statements (for example, MULTIPLY), share this problem.

2. Target paragraphs of the GO TO statement with the DEPENDING ON clause may now be in independent segments (overlays).

3. The MS-SORT library has been changed to correct problems found in sorting large (around one megabyte in size) files.

Microsoft COBOL Compiler

Release 4.65 Update Notice

October 1982

Microsoft COBOL Compiler Release 4.65 contains several enhancements and corrections to the previous release. These changes are listed below and, where necessary, are documented in the Microsoft COBOL User's Guide.

Note that if you are using the MS-SORT utility with MS-COBOL Version 4.65, you will need to contact Microsoft to receive the MS-SORT update. Contact the Product Update Administrator, Microsoft Corporation, 10700 Northup Way, Bellevue, Washington 98004 (Phone: (206) 828-8080).

Enhancements

1. Sequential, Line Sequential, and Relative files now use a sophisticated buffer management scheme in which no buffers are allocated within the compiled program. This new method of buffer management will decrease the size of compiled programs and improve runtime performance.

2. Relative files will no longer be non-contiguous: all space between records is set to zeroes, with deleted records indicated by a null byte as the first byte of the record. This new format allows users to copy relative files with the CP/M PIP utility.

   Note that if current files have large gaps between record numbers, the format will not be compatible with this version of COBOL. To correct this incompatibility, you will need to create an MS-COBOL program, under a previous release of MS-COBOL, that writes records into the unallocated gaps of your relative files. Then write these new records to the file with a null byte as the first character, or DELETE them after they have been written.

3. The Debug facility has been modified to make use of a new buffering mechanism for accessing the Debug Information File. Two new commands have been added:

Address [<dataname>] - displays the absolute address (hexadecimal) of a data item in memory.

Dump [<addr1>[,<addr2>]] - displays memory addresses (hexadecimal and ASCII equivalents) from address 1 through address 2.

These commands are also documented in Section 4.2 of the Microsoft COBOL User's Guide.

4. An Indexed File Recovery Utility, called REBUILD, has been added. It reads through the data portion of an indexed file, skipping over deleted records, and creates new key and data files. This utility can be used whenever a damaged indexed file must be recovered--for example, when a "Disk full" condition occurs.

5. Using the REWRITE statement with an indexed file formerly wrote records to the file with a deleted flag set on (reference count=0). This fault has been corrected in Release 4.65. Note that this did not interfere with normal MS-COBOL processing of the file. However, the Indexed File Recovery Utility (REBUILD) will skip over records whose reference count is 0. Therefore, if you wish to use REBUILD with such files, these records should be recorded correctly in the data file. The skeleton program RECOVR.COB has been provided on your second distribution disk for this purpose. Once modified and compiled, it will read an old indexed file and write a new one.

6. If you run out of disk space after a file is written to or closed, MS-COBOL will now move "24" (for a sequential or line-sequential file) or "34" (for an indexed or relative file) into the file status item, indicating that data have been lost. It is therefore a good idea to check the file status after a WRITE or CLOSE.

Corrections

**In previous versions, if files were open at the time of a CHAIN, arguments may have overwritten disk buffers. This problem has been corrected.

**Various intermittent problems, including incorrect evaluation of subscripted variables, were formerly caused when the compiler did not allocate sufficient space to a temporary memory area when compiling a segmented program. This has been corrected.

**Correction has been made to the exponentiation algorithm where the scale of the result was not computed correctly for a negative exponent.

**Corrections have been made to the processing of all file types when an out-of-space condition occurs during a WRITE. Now, when a CLOSE is executed following an unsuccessful WRITE attempt, all buffers are written to disk and a call is made to the operating system to close the file and record its full length. Note that there is still a possibility that the last record written will only partly exist in the file. Because of the complicated block manipulation required for indexed files, the out-of-space condition requires use of the Indexed File Recovery Utility (see REBUILD, above).

**In previous versions, indexed files that contained _variable_ length records did not perform REWRITEs correctly. This problem has been corrected.

**In previous versions, problems occurred when SORT was used with input or output procedures in independent segments. These problems have been corrected.

**Two changes have been made to fields involved in a screen ACCEPT: (1) Highlight and blink will now remain active during an ACCEPT. (2) When the USING or WITH UPDATE attributes are active, individual numeric fields must either be accepted as displayed or completely replaced. The first digit typed while the cursor is in the field will clear the original number.

**Default tab settings have been changed to columns

8, 12, 20, 28, 36, 44, 52, 60, and 68

See Appendix C of the _Microsoft COBOL User's Guide_ for information on how to change these settings.

Microsoft COBOL user's guide

for 8080 and Z80 microprocessors

Information in this document is subject to change without notice and does not represent a commitment on the part of Microsoft Corporation. The software described in this document is furnished under a license agreement or nondisclosure agreement. The software may be used or copied only in accordance with the terms of the agreement. It is against the law to copy Microsoft COBOL on cassette tape, disk, or any other medium for any purpose other than the purchaser's personal use.

Copyright (C) Microsoft Corporation, 1980

LIMITED WARRANTY

MICROSOFT CORPORATION shall have no liability or responsibility to purchaser or any other person or entity with respect to any liability, loss or damage caused or alleged to be caused directly or indirectly by this product, including but not limited to any interruption of service, loss of business or anticipatory profits or consequential damages resulting from the use or operation of this product. This product will be exchanged within twelve months from date of purchase if defective in manufacture, labeling or packaging, but except for such replacement the sale or subsequent use of this program is without warranty or liability.

THE ABOVE IS A LIMITED WARRANTY AND THE ONLY WARRANTY MADE BY MICROSOFT CORPORATION. ANY AND ALL WARRANTIES FOR MERCHANTABILITY AND/OR FITNESS FOR A PARTICULAR PURPOSE ARE EXPRESSLY EXCLUDED.

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

Microsoft COBOL Compiler is a trademark of Microsoft Corporation.
CP/M is a registered trademark of Digital Research, Inc.
ISIS-II is a trademark of Intel Corporation.

Document No. 8302-465-04

Contents

CHAPTER 1 OVERVIEW

1.1 Introduction
1.2 Your Distribution Disks
1.2.1 The Microsoft COBOL Compiler
1.2.2 The Runtime System
1.2.3 The CRT Drivers
1.2.4 Utility Software
1.2.5 Miscellaneous Files
1.3 Getting Started
1.4 Program Development Steps

CHAPTER 2 COMPILING MICROSOFT COBOL PROGRAMS

2.1 Microsoft COBOL Command Line Syntax
2.2 Compiler Switches
2.3 Output Listings and Error Messages
2.4 Files Used by Microsoft COBOL

CHAPTER 3 LINKING MICROSOFT COBOL PROGRAMS

3.1 MS-LINK Command Line Syntax
3.2 Subprograms
3.3 Function Libraries

CHAPTER 4 EXECUTING MICROSOFT COBOL PROGRAMS

4.1 The Runtime System
4.2 The Interactive Debug Facility
4.3 Printer File Handling
4.4 Disk File Handling
4.5 CRT Handling
4.5.1 Terminal Output
4.5.2 Keyboard Input
4.6 Runtime Errors

APPENDIX A CONFIGURING THE CRT

A.1 General Instructions
A.2 Terminal Charts
A.3 Writing a CRT Driver

APPENDIX B INTERPROGRAM COMMUNICATION

B.1 Subprogram Calling Mechanism
B.2 CHAIN Parameters
B.3 CHAIN Error Messages

APPENDIX C CUSTOMIZATIONS

C.1 Source Program Tab Stops
C.2 Compiler Listing Page Length
C.3 Runtime DAY, DATE, TIME, LINE NUMBER

APPENDIX D REBUILD: Indexed File Recovery Utility

D.1 Overview
D.2 Running REBUILD
D.3 Sample REBUILD Session

APPENDIX E EXTENSIONS FOR FILE HANDLING UNDER CP/M-80

CHAPTER 1

OVERVIEW

1.1 INTRODUCTION

The purpose of this *Microsoft COBOL User's Guide* is to give you practical information about getting a Microsoft COBOL (MS-COBOL) program up and running on your 8080 or Z80 computer. All the steps necessary to use Microsoft COBOL successfully -- compiling, linking, executing, etc. -- are carefully described in the following pages.

1.2 YOUR DISTRIBUTION DISKS

You will receive from one to four disks from Microsoft, depending on your implementation of MS-COBOL. The following files are on disk (though not necessarily in this order):

The MS-COBOL Compiler
- COBOL.COM
- COBOL1.OVR
- COBOL2.OVR
- COBOL3.OVR
- COBOL4.OVR

The Runtime System
- RUNCOB.COM -- the runtime executor
- COBLBX.REL -- the runtime executor's library
- CRTDRV.REL -- the dummy CRT driver
- DEBUG.REL -- the interactive debug facility

Utility Software
- L80.COM -- the Microsoft Linking Loader
- LD80.COM -- the Microsoft Link-to-Disk Linking Loader
- LIB.COM -- the Microsoft Library Manager
- M80.COM -- the Microsoft Macro Assembler
- CREF80.COM -- the Microsoft Cross-Reference Facility

CRT Drivers -- files whose names begin with CD
- CD____.MAC -- source code
- CD____.REL -- relocatable object code

Conversion Utilities
- SEQCVT.COM -- converts old format sequential files to line sequential
- CVISAM.COM -- converts old format indexed files to new format
- REBUILD.COM -- "recovers" a damaged indexed file
- RECOVR.COB -- a skeleton MS-COBOL program which converts indexed files created previous to Version 4.65 to format compatible with REBUILD

Demonstration Programs
- SQUARO.COB
- CRTEST.COB

Miscellaneous
- COBLIB.REL -- a relocatable version of RUNCOB.COM
- COBLOC -- gives the load address of MS-COBOL programs
- COPCOB.SUB -- a SUBMIT file to copy all files from this diskette

1.2.1 The Microsoft 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

The runtime system consists of an executable program that interprets the object code of your program produced by the compiler. It resides in a file called RUNCOB.COM. In addition, some optional routines are contained in COBLBX.REL. These optional routines will be linked with your compiled program by the linker. At runtime, the compiled program relocates itself to a high memory address (the address contained in the first line of COBLOC), then reads in the executor from RUNCOB.COM, and jumps into the interpretation code of the runtime system.

Certain extensions for file handling under CP/M-80 are included in COBLBX.REL. See Appendix E for the description of these extensions.

1.2.3 The CRT Drivers

The CRT drivers enable you to configure your system for the type of CRT terminal you are using. You will need to select the appropriate driver (see Appendix A of this Guide). Once you have done so, the driver will be automatically included with each program you link with the linking loader. The driver provides cursor positioning and other functions to support interactive ACCEPT and DISPLAY statements.

1.2.4 Utility Software

The Microsoft linking loader is used to link MS-COBOL object programs with the runtime system (see Chapter 3 of this Guide). The other utilities are provided for your convenience. Each of these programs is documented in the Microsoft Utility Software Package Manual.

1.2.5 Miscellaneous Files

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

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

SEQCVT.COM is a special utility program that converts MS-COBOL files from SEQUENTIAL format to LINE SEQUENTIAL format. The Microsoft 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. To use this program, enter:

```
SEQCVT <newfile>=<oldfile>
```

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

COBLOC provides two pieces of information to the linking loader, and to the runtime system. On the first line of COBLOC is the address at which a compiled program will be linked. The linker will use this value as the argument for an implied /P switch. The second line of COBLOC tells the compiled program which drive contains the runtime system. You may change the information on this line to suit your configuration. An A: on this line indicates that RUNCOB.COM will reside on drive A at runtime. A colon only (:) on this line indicates that the executor will reside on the currently logged drive. COBLOC must not contain editor line numbers.

DEBUG.REL is the Microsoft COBOL interactive debug facility. It can be linked with an MS-COBOL object program as described in Section 4.2.

COBLIB.REL is an alternative version of the runtime system. Unlike the executor (RUNCOB.COM), this file is linked with your compiled program to form one entire executable unit.

This runtime system can only be used when your program is compiled with the /X switch (see Section 2.2). Use of COBLIB may create smaller runtime images, because the linking loader will not link various optional modules into the executable program unless they are referenced.

CVISAM.COM is a utility program that converts indexed files created before Version 4.6 to new format. Before using this program, be sure to make a backup of the old format file. To use CVISAM, enter:

```
CVISAM
```

The program will prompt you for the information it needs to perform the conversion.

REBUILD.COM is a utility program used for recovering damaged indexed files. REBUILD.COM reads the data portion of the file, skipping over deleted records, and creates new data and key files (see Section 4.4 for detailed discussion of indexed files). Use REBUILD.COM just like CVISAM.COM: enter

```
REBUILD
```

and the program will prompt you for the information it needs. See Appendix D for a more detailed discussion of REBUILD.

Note: If you have indexed files created prior to Version 4.65, they may contain records with a delete flag incorrectly set on. In this case, use the RECOVR.COB program (described below) to rewrite the file.

RECOVR.COB is a skeleton MS-COBOL program that reads an indexed file and creates a new file, ensuring that delete flags within rewritten records are set off. This program should be used to re-create indexed files written before Version 4.65, which may contain incorrectly set delete flags. The RECOVR.COB program may be edited and altered for particular situations.

1.3 GETTING STARTED

The first thing to do when you receive your disk is make a copy for program development, saving the original disk as a backup. This may be done by using the COPCOB command file supplied with MS-COBOL or with some other disk copying facility you may have.

Having done that, verify your copy of the compiler and runtime system by compiling, linking, 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 MS-COBOL 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 an MS-COBOL program for execution consists of three basic steps:

1. Creating the source file with a text editor

2. Compiling with the MS-COBOL Compiler

3. Linking 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 a source file with Microsoft EDIT-80 or any other editor that uses 7-bit ASCII character codes. The file should be prepared according to the coding rules given in the Microsoft COBOL Reference Manual. 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 8, 12, 20, 28, 36, 44, 52, 60, and 68. Columns past 72 should not be used. If you use EDIT-80, 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 MS-COBOL compiler and provide the name of your source file. Under CP/M-80, you must be logged-in on the disk that contains the MS-COBOL 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 file:

```
A>COBOL 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 filename extensions assumed by the compiler:

```
A>COBOL 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 Chapter 2.

Once the source program is compiled, the final step before execution is to link the program with the Linking Loader, MS-LINK. This step converts your relocatable object program into an absolute version and combines it with the Microsoft 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 link SQUARO and execute it without saving the absolute version:

```
A>LD80 SQUARO/G
```

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

```
A>LD80 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 link step. To combine the two examples so that the absolute version is saved and then executed directly, type:

```
A>LD80 SQUARO/N,SQUARO/G
```

Refer to Chapter 3 of this Guide and to the Microsoft Utility Software Package Manual for a full description of MS-LINK command line syntax.

CHAPTER 2

COMPILING MICROSOFT COBOL PROGRAMS

2.1 MICROSOFT COBOL COMMAND LINE SYNTAX

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

The syntax of an MS-COBOL command line is:

```
objectfile,listfile=sourcefile
```

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

<u>objectfile</u>
: the name of the file to which the object program is to be written

<u>listfile</u>
: the name of the file to which the program listing is to be written

<u>sourcefile</u>
: the name of the MS-COBOL program source file

Each file can be the name of a disk file or the name of a system device. 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. MS-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 file

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

Note that in each example given below, a debug information file (see Section 4.2) is created and placed on the same disk as the object file.

Examples:

Command String	Effect
,=PAYROLL	Compiles the source from PAYROLL.COB and produces only an error count, which is displayed on the console.

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

Switch	Action
/R	Force the compiler to generate an object file. This switch 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. Much like /R, the /L switch 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 only if stack overflow errors occur during compilation (see Section 2.3 below).
/D	Omit generation of a debug information file. Also, do not output object code information about source file line numbers. This switch saves 2 bytes of object code for each PROCEDURE DIVISION line.
/X	Prepare an object file that can only be linked with the runtime system contained in COBLIB.REL. The executor program file (RUNCOB.COM) will not be loaded by the compiled program. This option allows you to link MS-COBOL programs in the manner used by versions prior to 4.6.
/Fn	Generate FIPS flagging messages on the listing file. The compiler can detect use of statements that are Microsoft extensions to the ANSI 1974 COBOL standard. The compiler can also detect the level of the standard that a particular statement implements. The numeric value that follows the switch identifies which level of features will be excluded from the list of features. The admissable values for <u>n</u> are:

value	meaning
0	List all FIPS flagging features and Microsoft extensions
1	Exclude LOW-INTERMEDIATE features
2	Exclude LOW-INTERMEDIATE and HIGH-INTERMEDIATE features
3	Exclude all except extensions
4	List no FIPS flagging features

Examples of command lines using switches:

Command String	Is Equivalant 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 Microsoft COBOL is a line-by-line account of the source file with page headings and error messages. 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 which statement has caused a runtime error.

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 generated at the end of the listing will refer to the line number attached to the low level error. Therefore, 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.
NAME?	Illegal COPY file name	Ignore 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 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. The program element cited at the point of error is listed.

An error message preceded by a line number in the range one through fourteen relates to a diagnostic about file use. The line number actually represents the internal reference number that the compiler assigns to the file. This number is assigned in the same order that FD statements are encountered in the FILE SECTION of the source program. These diagnostics detect semantic incompatibilities between the declared attributes of a file and its use within the PROCEDURE DIVISION. These error messages should be heeded, but they will not impede execution of the program.

Regardless of whether there is a list device, or what the list device may be, a message displaying the total number of errors or warnings is always displayed on the console at the end of compilation. This allows you to make a simple change to an MS-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. This message 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 in Phase n at address xxxx
```

occurs when the compiler becomes confused. It is usually caused by one of two problems: either the source program has been damaged, such as having been overwritten by binary data; 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 appears to be the cause of the problem, 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 MICROSOFT COBOL

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

First, the compiler always places a file called STEXT.INT 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 deleted before the compiler exits. Consequently, you should never run into it unless the compilation is aborted.

Second, note the placement of any of the files to be included because of COPY verbs in the MS-COBOL program. (See the discussion of COPY in the _Microsoft COBOL Reference Manual._) Remember that copied files cannot have COPY statements within them.

Finally, MS-COBOL programs that use segmentation cause the linker to create a file for each independent segment of the program. 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 decimal segment number minus 49 decimal.

CHAPTER 3

LINKING MICROSOFT COBOL PROGRAMS

The Microsoft Linking Loader (MS-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 MS-COBOL runtime library with your object program. The linker is also used to link one or more subprograms with a main program. These subprograms may be specified individually or extracted from a library, and may be written in MS-COBOL, MS-FORTRAN, or MS-Macro Assembly language.

When you link MS-COBOL programs, the file COBLOC must be on the currently logged drive. The first line of this text file contains an address that should never be altered. The second line contains a disk drive specification which will indicate to the runtime system the location of the MS-COBOL executor, RUNCOB.COM. The contents of this line can be altered to suit your particular needs. (See Section 4.1 for further discussion.)

3.1 MS-LINK COMMAND LINE SYNTAX

The complete syntax for MS-LINK commands is given in the Microsoft Utility Software Package Manual. However, some functions described there are not useful when linking MS-COBOL programs. This chapter summarizes use of the linker for MS-COBOL programs.

You may invoke MS-LINK in one of two ways: either type LD80 followed by a carriage return and enter a command line when the asterisk prompt is typed, or type LD80 followed by a space, followed by the command line on the same line. (To use the in-memory version of the linking loader, enter the same commands described in this chapter for LD80, except type L80 instead of LD80.)

The command line is a list of filenames separated by commas. Each filename specified is brought into memory by the linker and placed at the next available memory address. Switches are used in the command line to specify functions the linker is to perform. The command line may be broken up into many small lines and entered on different lines. The linker will prompt with an asterisk and wait for more command lines until one with a /G or /E switch has been processed and the linker exits to the operating system.

All relocatable object files are assumed by the linker to have a filename extension of .REL. The generated executable file will have a default extension of .COM. These defaults may be overridden by explicitly entering an extension with the filename.

Switches most useful when linking MS-COBOL programs are:

Switch	Effect
filename/N	Directs the linker to save the executable program on disk with name <filename> when the linking process is complete.
/E	Directs the linker to complete the linking process and exit to the operating system. The linker searches the appropriate MS-COBOL runtime library (COBLIB.REL or COBLBX.REL) and the interactive CRT driver (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 linker to complete the linking process and begin execution of the program. As with /E, the MS-COBOL runtime library is searched, and the executable program is saved if /N was specified.

Switches used occasionally when linking MS-COBOL programs are:

Switch	Effect
/R	Immediately resets the linker to its initial state. The effect is as if the linker was aborted and then reloaded from disk.
filename/S	Directs the linker 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 Line

MYPROG,SVPROG/N/E

Links MYPROG.REL, saves the absolute version in SVPROG.COM and exits to the operating system.

MYPROG/G

Links MYPROG.REL and begins execution without saving the absolute version.

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

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

MYPROG/N,MYPROG,MYLIB/S/E

Links MYPROG.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 linking loader must be used to combine them into one executable program. Before linking, compile (or assemble) all modules so that you have a relocatable object version of each. Then execute the linker and specify in the command line the name of each module you want to link. The first module loaded must be an MS-COBOL program.

Note that if a subprogram module does not have a USING clause in its PROCEDURE DIVISION header, the compiler will not recognize it as a subprogram, but rather assumes it is a main program. Because the linker sets the starting address for execution at the last starting address given in the command line, a command such as

```
LD80 MAINPG/N,MAINPG,SUBPG1
```

would result in the subprogram being executed first. To guard against this possibility, issue the command line with the main program listed last.

For example, if you have a compiled main program file MAINPG.REL and two subprogram files SUBPR1.REL and SUBPR2.REL that do not contain a USING clause in the PROCEDURE DIVISION header, you may load the executable program and save it with the following link command:

```
LD80 MAINPG/N,SUBPR1,SUBPR2,MAINPG/E
```

3.3 FUNCTION LIBRARIES

The Microsoft Library Manager MS-LIB (CP/M-80 versions only), allows you to collect any number of subprograms into a single file (a library) that can be searched by the linker. For example, if you have six subprograms named SUBPR1.REL through SUBPR6.REL that are used by different main programs, you could incorporate 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 the _Microsoft Utility Software Package Manual_ for a full description of MS-LIB.) Then, if you have a main program MAINPG that CALLs SUBPR2 and SUBPR6, the link command:

```
LD80 MAINPG/N,MAINPG,USRLIB/S/E
```

will link MAINPG and search USRLIB for SUBPR2 and SUBPR6.

You may also use MS-LIB to append modules to the MS-COBOL runtime library.

The MS-COBOL Compiler gives a name to each program or subprogram compiled. The name is taken from the PROGRAM-ID paragraph. When making a library, you need to make sure that all subprogram IDs are unique. Since all MS-COBOL runtime routines in the MS-COBOL runtime library have names that begin with dollar sign, avoid the dollar sign in naming your subprograms.

CHAPTER 4

EXECUTING MICROSOFT COBOL PROGRAMS

You may execute an MS-COBOL program in any of three ways. The first is to use the /G switch in the MS-LINK command line as described in Section 3.1. Since this approach does not automatically save the program as an executable file on disk, it is not widely used and will not be discussed further in this chapter. The second method is simply to type the name of an executable program file as saved by using the /N switch in the linker command line. Finally, you may execute a program directly from another MS-COBOL program by using the CHAIN statement. Refer to Chapter 5 of the Microsoft COBOL Reference Manual for an explanation of program CHAINing.

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 MS-COBOL instructions. The Microsoft COBOL 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 can be present in two forms. Normally, it resides in the file RUNCOB.COM. When a program is invoked by typing its name, RUNCOB.COM must be on the drive specified in the second line of the COBLOC file. Once loaded into memory, RUNCOB.COM will not be reloaded when chaining from one MS-COBOL program to another.

The COBLOC file is a text file referred to only at link time. The first line of this file contains an address that should never be altered. The second line contains a disk drive specification which indicates to the runtime system the location of RUNCOB.COM at execution time. The contents of this line can be altered to suit your particular runtime configuration. This second line should contain only one

character which is a disk drive letter. If RUNCOB.COM will always be on the currently logged drive at runtime, a colon (:) only may be entered on this line.

The runtime system can also reside within the compiled MS-COBOL program. This option is selected by using the /X switch during compilation. This switch produces instructions (within the relocatable object file) to the linking loader to combine COBLIB.REL with the compiled program. COBLIB.REL contains an alternative version of the runtime system. Use of this version precludes use of the runtime system contained in RUNCOB.COM.

Whether the runtime system is to reside in RUNCOB.COM or in COBLIB.REL, the linking loader performs three additional functions. First, it resolves relative address references in the compiled program to absolute values. Second, it automatically binds optional modules from the MS-COBOL library to the program's object code. If the /X switch is given to the compiler, the optional modules reside in the COBLIB.REL runtime library. Without the /X switch, the optional modules reside alone in the COBLBX.REL library. If subprograms are called from the main MS-COBOL program, they too will be bound into the executable file by the linking loader. Third, the linker automatically links the file CRTDRV.REL to include terminal dependent functions.

The amount of memory required by an MS-COBOL program at runtime equals the amount required to store the data items defined in the DATA DIVISION, plus about 100 bytes per file, plus about 12 bytes per line of the PROCEDURE DIVISION, plus up to 19K bytes for the optional runtime library modules, plus 29K for the runtime system RUNCOB.COM.

All I/O is done by standard operating system calls.

4.2 THE INTERACTIVE DEBUG FACILITY

The debug facility will be invoked when you execute a compiled program that has been linked with the file DEBUG.REL. You will have to specify this file explicitly in the linker command line for the file to be included; unlike the runtime library and CRT driver, the debug file is <u>not</u> automatically linked.

The debug facility uses two types of information: datanames and line numbers. Both items are placed in a "debug information file" automatically created by the compiler. (The /D switch suppresses generation of this file.) The debug information file will have the same filename as the name contained in the PROGRAM-ID paragraph of the MS-COBOL program. The extension will always be .DBG. For example, if a program has the paragraph:

```
PROGRAM-ID.  FOOBAR.
```

the filename of the debug information file will be:

```
FOOBAR.DBG
```

To use the debug facility, first remember to include the file DEBUG.REL in the linker command line. Then execute your program. You will see the message:

```
MS-COBOL Interactive Debug Facility v. xxx
Program:  <PROGRAM-ID>
  Type help for list of commands
```

on your console screen. The debug asterisk prompt (*) follows to indicate that the debug facility is ready to accept any of the debug commands listed below. The debug information file should be on the current disk. If is it not, the message

```
**No debug information file found
```

will follow the messages already displayed.

Without a debug information file, limited debugging is possible. You can execute any of the debug commands except Change, Exhibit, and Goto <line-number>. However, without the debug information file, the debug facility cannot verify that line numbers specified in the Breakpoint command are valid PROCEDURE DIVISION line numbers that contain statements or section or paragraph names.

Debug commands may be typed in full or abbreviated to the first letter (underlined in the chart below) of the command name. Upper and lower cases are equivalent. Arguments to the commands (line numbers, datanames, ALL, OFF) must be given in full. Though spaces are shown below, arguments can be separated from commands by any non-alphabetic character. Further, when a numeric argument is expected, the debug facility will scan until the first digit on the line is found. For example, the following commands are equivalent (set a breakpoint at line 100):

```
Breakpoint 100
BREAK @ 100
b100
break for me at line 100, if you would please
```

The following functions are available with the debug facility:

Function	Description
Address [<dataname>]	Display absolute address (hexadecimal) of <dataname>* in memory.
Breakpoints	List all breakpoints.
Breakpoint <line-num>	Set a breakpoint at <line-num>. You may have up to 8 breakpoints set at any given time. Debug verifies that <line-num> is a PROCEDURE DIVISION line that contains a statement or a section or paragraph name.
Change <dataname>	Display the contents of <dataname>* and allow a new value to be entered.
Dump [<addr1>[,<addr2>]]	Display memory addresses (hexadecimal and ASCII equivalents) from <addr1> through <addr2>.
Exhibit <dataname>	Display the contents of <dataname>*.
Go	Resume execution from the last breakpoint or current program position until a breakpoint or end of program is encountered.
Goto <line-num>	Begin execution at <line-num> until a breakpoint or end of program is encountered.
Help	Display the list of debug commands.
Kill <line-num>	Remove the breakpoint at <line-num>.
Kill ALL	Remove all breakpoints from the breakpoint list.
Line	Display the <line-num> of the current line.
Quit	Terminate the program (closing all open files).
Step	Execute one statement.

Trace	Set trace mode. When trace mode is set, the line number of each line will be displayed as the line is executed.
Trace OFF	Turn off trace mode. This command sets trace mode off. (See the description of Trace.)

*Note that subscripted variables cannot be used as datanames.

4.3 PRINTER FILE HANDLING

When printer files (files transmitted directly to the printer) are generated, no extra characters are needed in the record for carriage control. The compiler sends carriage returns, line feeds, and form feeds 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.

4.4 DISK FILE HANDLING

All of the following formats are subject to change without notice.

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 Microsoft Utility Software Package Manual. Block clauses are checked for syntax but have no effect on any file type.

The format of 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. The end-of-file marker for a SEQUENTIAL file is one Control-Z character following the last record of the file. The remainder of the disk sector will be filled with nulls. For a LINE SEQUENTIAL file, two Control-Z characters follow the last record of the file. Again, the remainder of the last disk sector following the end-of-file

marker will be filled with nulls. To make maximum use of disk space, records are packed together with no unnecessary bytes between them.

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. The runtime marks records as deleted by placing a null byte in the first byte of the deleted record. In this case, programs will not be able to access the record again. It is therefore extremely important that records written to RELATIVE files _not_ contain a null byte in the first position.

Six bytes are reserved at the beginning of a relative file for system bookkeeping information. The first byte of this six byte header indicates whether the file is damaged. It is set to 254 (decimal) when the file is opened in I-O mode. If the file is not closed properly, this byte will not be cleared, indicating a possible damaged status. When the file is closed properly, this byte is set to null.

Each INDEXED file declared in an MS-COBOL program will generate two disk files. The file specification in the VALUE OF FILE-ID clause specifies a file containing data only. The filename included in the file specification is concatenated with an extension "KEY" to form the file specification of the key file. The key file contains keys, pointers to keys, and pointers to data. The format of this file is very complicated, but follows the guidelines for a prefix B+ tree.* The data file consists of data records. Each data record is preceded by a two-byte length field and a one-byte "reference count" that indicates if a record has been deleted. A reference count of 0 indicates a deleted record; a count of greater than 0 indicates the record is referenced by the key file. The data file is terminated by a control record that has a length field containing 2, followed by two bytes of high-values.

The key file is divided into 256 byte units, called granules. There are five possible granule types. A type indicator is located in the first byte of each granule. The granule type indicators have the following values:

Value	Type Indicator
1	Data Set Control Block
2	Key Set Control Block
3	Node
4	Leaf
5	Deleted granule

---

* See Comer, Douglas. "The Ubiquitous B-Tree." _Computing Surveys of the ACM._ Vol 11, no. 2 (June 1979), pp. 121-137.

The key file will possess only one Data Set Control Block in the first granule, one Key Set Control Block for the primary file key, and additional Key Set Control Blocks for alternate keys.

Damaged flags exist in the fourth byte of the data Set Control Block and in the fourth byte of each Key Set Control Block. They are set to nonzero values when the file is opened for updating and restored to zero when the file is closed.

4.5 CRT HANDLING

4.5.1 Terminal Output

All output to the terminal is done by the DISPLAY or EXHIBIT statements. Characters are sent one at a 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.

The runtime performs no conversion of data prior to its display. Therefore, unless specifically intended, do not display fields containing binary information.

4.5.2 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).

4.6 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, in the format:

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

The possible reasons for termination, with additional explanations, are listed on the next pages.

Reason	Explanation
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 MS-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 content 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 using 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; or more than 22 levels of PERFORM nesting.
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.

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 a file 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 decimal segment number minus 49 decimal. They 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 a list of CHAIN error messages.

APPENDIX A

CONFIGURING THE CRT

A.1 GENERAL INSTRUCTIONS

To enable the interactive ACCEPT and DISPLAY functions, MS-COBOL 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 MS-LINK.

A module named CRTDRV is provided with the Release 4 MS-COBOL disks. 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 MS-COBOL programs. To do this, simply copy the appropriate driver to CRTDRV.REL. Microsoft has provided drivers for a wide range of popular 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 Microsoft are relocatable object files whose names begin with the letters CD (for CRT Driver). The MS-Macro Assembler 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 is not charted separately but matches the code for any of the supplied drivers, use that chart. 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

---

*Does not support the cursor on/off functions or the highlight/blink function. When using the VT52, replace the appropriate routines with RETURN opcodes.

** 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 MS-COBOL 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 MS-COBOL 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 two ASCII digits, sent high digit first.

C1 The binary column number plus decimal 31.
C2 The column number converted to two 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 Rl Cl
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		0B	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

CDANSI

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		0B	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 Rl Cl
B. Backspace Cursor	08
C. Cursor On	N/A
D. Cursor Off	N/A
E. Erase to end of Screen	ESC J Nl
F. Erase to End of Line	ESC K
G. Sound Bell	07
H. Set Highlight Mode	ESC l
I. Reset Highlight Mode	ESC m

CDHZ15 — Hazeltine 1500 Series Terminals
24 Lines 80 Columns

I. Keyboard Input

A. Editing Keys	Input Code	Key Label
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	CONTROL-A
2.	03	03	CONTROL-C
3.	04	18	CONTROL-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

CDISB — 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	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
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

NOTE

Highlight can be implemented in nonstandard ways.

CDPERK — Perkin-Elmer Terminals
24 Lines 80 Columns

I. Keyboard Input

A. Editing Keys	Input Code	Key Label
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 Rl ESC Y Cl
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

CDSROC

SOROC IQ Terminals
24 Lines 80 Columns

I. Keyboard Input

A. Editing Keys	Input Code	Key Label
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

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

CDWH19

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, 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 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

A. Editing Keys

	Input Code	Key Label
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 = Rl Cl
B. Backspace Cursor	08
C. Cursor On	N/A
D. Cursor Off	N/A
E. Erase to end of Screen	ESC Y Nl
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

A.3 WRITING A CRT DRIVER

A CRT driver should be written in assembly language and assembled with the Microsoft Macro Assembler. A driver consists of 16 entry points that must be declared as global labels by using MS-Macro Assembler ENTRY statements. The source codes for a number of drivers are supplied on your distribution disk (files named CD____.MAC) to serve as examples and reference for the following explanation. It may be convenient to modify one of these drivers instead of starting from scratch.

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. To use CRTEST, compile it, link it with your CRT driver using MS-LINK, and execute it, following the instructions provided by the program itself.

Five of the driver entry points 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 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 decimal. 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 moves the cursor 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 11 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. Be sure to preserve values in all registers except the accumulator and condition flags.

$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 MS-COBOL considers the top line of the screen to be row 1 and the leftmost column to be column 1. If necessary, you can convert the row and column number to a sequential screen position by calling $SQPOS. This routine expects the row and column to be in HL as they are passed to $SETCR, and returns the result in HL. $SQPOS is permanently resident in the runtime.

$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. The sequential format may be converted to row-column screen format by calling $ROWCL. It expects HL to contain a sequential screen position and will return with a converted row and column number in HL. $ROWCL is permanently resident in the runtime.

$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 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).

$INCRT sends a character to the MS-COBOL runtime system in the A register. Use $INKEY to get a character from the operating system, which in turn receives the character from the keyboard. $INKEY returns with the character in the A register.

$OUCRT sends the character in the A register to the operating system, which in turn sends the character to the CRT screen.

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 Microsoft COBOL runtime system if both programs are written in MS-COBOL. However, if the CALLed or CHAINed program is written in assembly language or MS-FORTRAN, Sections B.1 and B.2 will apply.

B.1 SUBPROGRAM CALLING MECHANISM

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

The MS-COBOL runtime system transfers execution to a subroutine by means of a machine language CALL instruction. The subroutine should return via the normal assembler or MS-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 MS-COBOL and MS-FORTRAN. None of the numeric types of data are interchangeable.

The stack space used by an MS-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 MS-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 MS-COBOL stack pointer before returning to the main program.

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

B.2 CHAIN PARAMETERS

The parameters passed between programs with a CHAIN USING statement are stored at the highest available memory address. The memory layout of the CHAINed program 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.

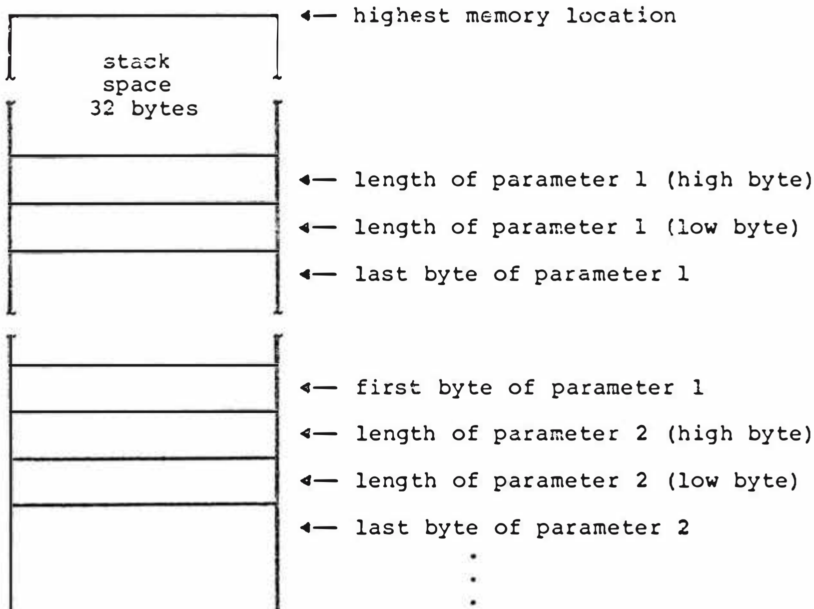


Figure B-1. Memory Layout of CHAIN parameters.

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 programs, plus all CHAIN parameters, plus 256 bytes for the program loader.

APPENDIX C

CUSTOMIZATIONS

This appendix is intended for those of you who are experienced assembly language programmers and would like to change some of the built-in parameters of MS-COBOL.

C.1 SOURCE PROGRAM TAB STOPS

If tab characters (hex 09) are used in the MS-COBOL 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):

8, 12, 20, 28, 36, 44, 52, 60, and 68

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

C.2 COMPILER LISTING PAGE LENGTH

One byte in the compiler defines the listing page length to be 55 (hex 37) lines. Its location is 6 bytes from the start of COBOL.COM 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, MS-COBOL uses the compiler release date for format 1 ACCEPT statements. For single-user systems, MS-COBOL always uses '00' for the line number. If you have a multi-user system of 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 MS-Macro Assembler, and place it in the appropriate runtime library, either COBLBX.REL or COBLIB.REL, using the library manager. Assuming you name the module ACPDAT.MAC, an MS-LIB command to place it in the library is:

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

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

ACPDAT Module

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

This module handles the runtime support for the MS-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
On entry:
      HL = address of source string
      DE = address of target area
      BC = length of the string in bytes
CALL  $MOVE
After call:
      HL = address of first byte beyond source
      DE = address of first 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        ;DAT
          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

REBUILD: INDEXED FILE RECOVERY UTILITY

The Indexed File Recovery Utility (REBUILD) can be used to recover or restore information contained within indexed files. The indexed files that are compatible with this utility are those that have been created by a program compiled under MS-COBOL Version 4.64 or later.

D.1 OVERVIEW

REBUILD works by reading the data file portion of an indexed file and generating new key and data files for that indexed file. The new indexed file has the same structure as the old one. The utility will skip over all deleted records and any other control records within the data file.

Use of REBUILD is recommended in the following situations:

1. When space is exhausted during a WRITE operation to the disk on which the indexed file resides.

2. When electrical power to the computer system is interrupted or the operating system is rebooted while an indexed file is open in I-O or OUTPUT mode.

3. When the data file portion of the indexed file contains large areas of unused space, usually as a result of numerous record DELETE and REWRITE operations, and especially when records within the file have varying lengths.

Situation 1 (above) occurs when WRITE produces a boundary error (file status "24"), indicating that the the disk is full. When this happens, you should perform a CLOSE in order to write as much information as possible to disk. It is likely, however, that the CLOSE will also return with a boundary error. As in the case of a system failure during the addition of records, the last 256 bytes of information will not be present within the data file, and are therefore not recoverable by REBUILD.

Recovery from situation 2 (above) may also be limited, because without a transaction file to rebuild the indexed file, recovery from some types of system failure is problematic. Because of the high degree of disk file buffering in memory, a system failure may leave the data file with partially-written data records. This may cause REBUILD to fail to completely recover an indexed file for two reasons:

a. Because a good deal of information is kept in memory, if the system failure occurred during a file update job, the file may contain records with both original and new information. The recovery utility cannot determine which part of the data was written during the aborted job, and therefore cannot exclude the new, incomplete data from the rebuilt file. Adding a current date field to data records may help discriminate between original and new data.

b. If the system failure occurred while records were being added to the indexed file, the last 256 bytes of data will not be written to disk. The recovery utility will detect that information is missing from the end of the file but cannot add it to the recovered file.

D.2 RUNNING REBUILD

Invoke the recovery utility by entering:

```
REBUILD
```

in response to the operating system prompt.

The utility will respond with the following header information:

```
REBUILD by Microsoft Corporation
Indexed File Recovery Utility
V. xxx

Use this utility to recover indexed files when they are
damaged, or to reorganize indexed files by removing unused
space.
Compatible indexed files are those generated by MS-COBOL (C)
for versions 4.64 and later.
```

The recovery utility will then ask a series of questions. Your answers will provide the information necessary for rebuilding a new indexed file from the original data file. The flow of control within the recovery utility, as it relates to the operator, is diagrammed in Figure D-1. Following the diagram are detailed descriptions of the individual recovery steps and a sample REBUILD session.

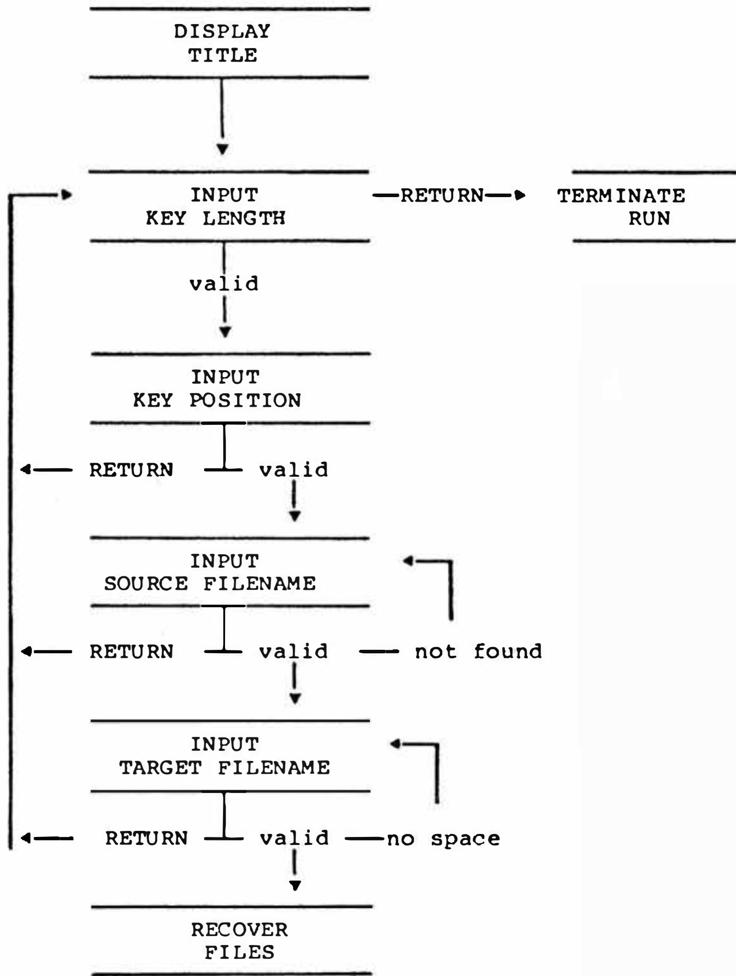


Figure D.1. Control flow within REBUILD

1) Input Key Length

Enter the key length in reply to the prompt:

```
Input the key length (in bytes)
or <RETURN> to terminate program ——▶
```

Enter a key length or press the <RETURN> key* to immediately terminate the program. If you enter a key length, the program will proceed to the next prompt.

The key length should be a positive integer that represents the number of bytes contained in the item specified by the RECORD KEY clause of an MS-COBOL program. Failure to enter the correct key length may not hamper the execution of REBUILD, but programs will not be able to access the generated indexed file.

2) Input Key Position

Enter the key position in reply to the prompt:

```
Input the byte position of the key field,
starting at 1,
or <RETURN> to return to the Key Length prompt ——▶
```

Enter the position of the key data item within the record; or press the <RETURN> key to move back to the Input Key Length prompt in order to correct information or terminate the program. If you enter a key position, the program will proceed to the next prompt.

The key position should be a positive integer that represents the position within the record of the data item specified by the RECORD KEY clause of an MS-COBOL program. As with the key length, REBUILD does not check whether an incorrect response has been entered; but the result of an incorrect response will be that programs will not be able to access the generated indexed file.

3) Input Source Filename

Enter the filename of the source file in reply to the prompt:

```
Input the filename of the source data file
(should not have extension of .KEY)
or <RETURN> to return to the Key Length prompt ——▶
```

---

*On some terminals, the <RETURN> key is labelled as NEWLINE or ENTER.

Enter a filename; or press the <RETURN> key to move back to the Input Key Length prompt so that you can correct and re-enter previous information or terminate the program.

The source filename should be the name that is used in the VALUE OF FILE-ID clause in MS-COBOL programs that refer to the indexed file. The filename used here should be the name of the data file. The key file, which has the same name but an extension of .KEY, will not be used in the recovery operation and should not be entered in response to this prompt.

The source filename may contain a drive specifier.

After the source filename is entered, REBUILD will check for the presence of the file. If it is not present, the following message will be displayed:

```
***Source file not found
```

and the Input Source Filename prompt will be redisplayed.

4) Input Target Filename

Enter the filename of the indexed file to be generated in reply to the prompt:

```
Input the filename of the target data file
(should not have extension of .KEY)
or <RETURN> to return to the Key Length prompt ——>
```

Enter a filename or press the <RETURN> key. As usual, <RETURN> moves you back to the Input Key Length prompt so that you can re-enter information or terminate the program.

As with the source file, this name is the name of the data file. Do not enter the key file, which has the same name but the .KEY extension.

The target filename should be unique within a directory. Therefore, if you wish to use a name identical to the source filename, you should send the target file to a different disk by including a drive specifier in the filename. The target file can be generated on the same disk as the source file, but you will have to use a different name. Once the recovery operation is complete, you can then rename the target filename to the source filename.

If the recovery utility cannot successfully create a new indexed file, either because the disk directory is full or because of insufficient space on the disk, the program will display the message:

```
*** No space for target file
```

and will redisplay the Input Target Filename prompt.

5) Recover File

After you have answered all questions, the recovery utility will display:

    Now reading <source-file>
    and creating <target-file>

The program will begin building the new indexed file from the old data file. When this process is finished, the following message will be displayed:

    Conversion successfully completed.
    Source records read: xxx,xxx
    Target records read: xxx,xxx

The record counts should match. If they do not, some type of input-output error occurred during the recovery operation.

Regardless of whether the record counts match, REBUILD will then display another Input Key Length prompt. You can begin another file recovery operation (or redo the one that had an input-output error) or terminate the program.

D.3 SAMPLE REBUILD SESSION

The following program fragment accesses the indexed file IXFILE.DAT:

```
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL
       SELECT IX-FILE
              ASSIGN TO DISK
              ORGANIZATION INDEXED
              ACCESS DYNAMIC
              RECORD KEY IX-KEY
              FILE STATUS IX-STAT.

DATA DIVISION.
FILE SECTION.
FD IX-FILE
   LABEL RECORD STANDARD
   VALUE OF FILE-ID "IXFILE.DAT"
   RECORD CONTAINS 75 CHARACTERS
   DATA RECORD IX-REC.
01 IX-REC.
   05 IX-DATE      PIC X(6).
   05 IX-TIME      PIC X(6).
   05 IX-KEY.
      10 IX-STATE     PIC XX.
      10 IX-CITY      PIC X(20).
      10 IX-STREET    PIC X(30).
   05 IX-ZIP       PIC X(5).
   05 IX-ZONE      PIC X(6).
```

For this program fragment, the responses to the REBUILD utility would be:

```
   Input Key Length:   52

   Input Key Position:   13

   Input Source Filename:   IXFILE.DAT

   Input Target Filename:   NEWIX.DAT
```

The result of the recovery operation would be to generate a new indexed file with the key filename NEWIX.KEY and the data filename NEWIX.DAT.

APPENDIX E

EXTENSIONS FOR FILE HANDLING UNDER CP/M-80

Various routines have been included to enhance file handling capabilities from within an MS-COBOL program. These routines are contained within the MS-COBOL runtime library, and will not be linked into the executable program if none of them are referenced from your MS-COBOL program. They are invoked by issuing subprogram CALLs with optional parameters (see Chapter 5 of the *Microsoft COBOL Reference Manual*).

The names and functions of these routines are as follows:

DSKRES   Reset the disk drives so that more than one diskette may be utilized per drive. WARNING: Do not call this routine if files are currently open.

FILENQ   Interrogate directory to see if a specified file exists.

KILL   Delete a specified file from the disk.

RENAM   Rename a specified file to another name.

The calling conventions of these routines are the following:

```
CALL 'DSKRES' USING status

CALL 'FILENQ' USING status,filename

CALL 'KILL USING status,filename

CALL 'RENAM' USING status,old-filename,new-filename
```

The variables status and filename in these calling sequences should have the following characteristics:

status — defined in the Working-Storage section with PICTURE XX. Returned status codes are:

'00'	Command executed with no errors
'30'	Error: file not found
'40'	Error: syntax error in filename
'50'	Error: duplicate file found

filename — defined in the Working-Storage section. It should be an alphanumeric field containing the filenames followed by a space. This format is similar to the data item referenced by the VALUE OF FILE-ID clause in the FD entry of the FILE SECTION.

INDEX

"QLIT"? 2-7

$ALARM A-15
$CLIST A-14
$CRLEN A-14
$CRWID A-14
$CURBK A-15
$CUROF A-16
$CURON A-16
$EOL A-16
$ERASE A-16
$FLIST A-15
$HILIT A-16
$INCRT A-16
$INKEY A-16
$LOLIT A-16
$OUCRT A-16
$OUTCH A-15
$ROWCL A-15
$SETCR A-15
$SQPOS A-15
$TLIST A-15

** RUN-TIME ERR: 4-8
**CHAIN: problem 4-10, B-3

.COB 2-2
.PRN 2-2
.REL 2-2
.Vnn 2-10, 4-10

/D 2-4, 4-2
/E 3-2
/Fn 2-4
/G 3-2, 4-1
/L 2-4
/M 3-3
/N 3-2
/P 1-4, 2-4
/R 2-4, 3-3
/S 3-3
/U 3-3
/X 1-5, 2-4, 4-2

?Command Error 2-1
?Compiler Error 2-9
?Memory Full 2-8

ACCEPT 1-3, 1-6, 4-7, 4-10, A-1, A-3, A-14, C-3

ACPDAT module . . . . C-3 to C-4
Address (debug command) 4-4
ADDS Regent Terminals A-4
ADM-3A A-16
ANSI Standard Terminal A-6
AREA A? 2-7
AT END condition. 1-9

B+ tree 4-6
Bad file name B-3
BADWORD 2-7
Beehive Terminals A-7
BLOCK clause 4-5
Breakpoint (debug command) . . 4-4
Breakpoints (debug command) . . 4-4

CALL 3-3, 4-9, B-1
CALL PARAMETERS 4-9
CALL USING B-1
CDADDS A-4
CDADM3 . . . . A-5, A-16
CDANSI A-6
CDBEE A-7
CDHZ15 . . . . A-8
CDISB A-9
CDPERK A-10
CDSROC A-11
CDWH19 . . . . A-12
CDZEPH A-13
CD____.MAC 1-2, A-14
CD____.REL 1-2
CHAIN 4-1, 4-10, B-1
CHAIN error messages B-3
CHAIN parameters B-2
CHAIN USING . . . . B-1 to B-2
CHAIN: problem 4-10, B-3
Change (debug command) 4-4
CHRCTR? 2-7
COBLBX.REL 1-1, 1-3
COBLIB.REL 1-2, 1-4, 2-4, 3-2, 4-2, C-2
COBLOC . . . . 1-2 to 1-4, 3-1, 4-1
COBOL.COM 1-1
COBOL1.OVR 1-1
COBOL2.OVR 1-1
COBOL3.OVR 1-1
COBOL4.OVR 1-1
COL.7? 2-7
Comer, Douglas 4-6
Command line syntax 2-1
Compiler switches 2-4
COPCOB 1-5
COPCOB.SUB 1-2, 1-4
COPY 2-7, 2-10
CREF80.COM 1-2

CRT Drivers 1-3
CRT handling 4-7
CRTDRV A-1
CRTDRV.REL 1-1, 3-2, 4-2, A-1
CRTEST A-14
CRTEST.COB 1-2, 1-4
CVISAM.COM 1-2, 1-5

DATA DIVISION 1-3
DATA UNAVAILABLE 4-9
Date C-2
Day C-2
Debug commands
  Address 4-4
  Breakpoint 4-4
  Breakpoints 4-4
  Change 4-4
  Dump 4-4
  Exhibit 4-4
  Go 4-4
  Goto 4-4
  Help 4-4
  Kill 4-4
  Kill ALL 4-4
  Line 4-4
  Quit 4-4
  Step 4-4
  Trace 4-5
  Trace OFF 4-5
Debug file 2-4
DEBUG.REL 1-1, 4-2
DECLARATIVE procedure 4-9
DELETE; NO READ 4-10
Disk file handling 4-5
DISPLAY 1-3, 1-6, 4-7, 4-10, A-1, A-3
Distribution disks 1-1
Dump (debug command) 4-4

Entry points A-14
ENVIRONMENT DIVISION 1-3
Error messages 2-6
  "QLIT"? 2-7
  **CHAIN: problem 4-10, B-3
  ?Compiler Error 2-9
  ?Memory Full 2-8
  AREA A? 2-7
  Bad file name B-3
  BADWORD 2-7
  CALL PARAMETERS 4-9
  CHAIN: problem B-3
  CHRCTR? 2-7
  COL.7? 2-7
  COPY 2-7
  DATA UNAVAILABLE 4-9

DELETE; NO READ 4-10
FEATURE UNIMPL. 4-10
FILE LOCKED 4-10
File not found B-3
GO TO. (NOT SET) 4-10
ILLEGAL DELETE 4-10
ILLEGAL READ 4-9
ILLEGAL REWRITE 4-9
ILLEGAL START 4-10
ILLEGAL WRITE 4-9
INPUT/OUTPUT 4-9
LENGTH? 2-7
NAME? 2-7
NO CRT DRIVER 4-10
NON-NUMERIC DATA 4-9
OBJ. CODE ERROR 4-10
Out of memory B-3
PERFORM OVERLAP 4-9
PIC = X 2-7
PICTURE 2-7
PUNCT? 2-7
READ BEYOND EOF 4-10
REDUNDANT OPEN 4-9
REWRITE; NO READ 4-9
SEG nn LOAD ERR 4-10
SUBSCRIPT FAULT 4-9
Escape Code A-3
Escape Key A-15
Executor (runtime system) . . . 1-3
EXHIBIT 4-7
Exhibit (debug command) 4-4

FD statements 2-8
FEATURE UNIMPL. 4-10
File ID 4-5
FILE LOCKED 4-10
File not found B-3
FILE SECTION 2-8
FILE STATUS item 4-9
filename/N 3-2
filename/S 3-3
FIPS flagging 2-4
Function libraries 3-4

Getting started 1-5
Global labels A-14
Go (debug command) 4-4
GO TO. (NOT SET) 4-10
Goto (debug command) 4-4
Granule type indicators 4-6
Granules 4-6

Hazeltine 1500 Series Terminals A-8
Heath WH19/DEC VT52 . . . . . . A-12
Help (debug command) 4-4

High level diagnostic 2-6
High-level diagnostic 2-8

IDENTIFICATION DIVISION 1-3, 2-10
ILLEGAL DELETE 4-10
ILLEGAL READ 4-9
ILLEGAL REWRITE 4-9
ILLEGAL START 4-10
ILLEGAL WRITE 4-9
INDEXED 4-6
Indexed File Recovery Utility . D-1
Input Code A-3
INPUT/OUTPUT 4-9
Interactive Debug Facility . . . 4-2
Intertec Superbrain A-9
INVALID KEY clause 4-9
ISAM 4-6

KEY 4-6
Key Label A-3
Keyboard input 4-7
Kill (debug command) 4-4
Kill ALL (debug command) . . . 4-4

L80 3-1
L80.COM 1-2
LABEL RECORD IS OMITTED 4-5
LABEL RECORD IS STANDARD 4-5
LD80 3-1
LD80.COM 1-2
Lear-Siegler ADM-3A A-5
LENGTH? 2-7
LIB.COM 1-2
Libraries 3-4
Line (debug command) 4-4
Line number C-2
LINE SEQUENTIAL 1-4, 4-5
Linking 3-1
Listing page length C-2
Low level error 2-6
LST: 2-2

M80.COM 1-2
Miscellaneous Files 1-4
MS-COBOL Interactive Debug Facility 4-2
MS-LIB 3-4
MS-LINK 1-7, 3-1
MS-Macro Assembler A-14

NAME? 2-7
NO CRT DRIVER 4-10, A-1
NON-NUMERIC DATA 4-9
NUMERIC test 4-9

OBJ. CODE ERROR 4-10

ON ESCAPE clause A-15
Out of memory B-3
Output listings 2-6

Page length C-2
Parameters B-1
PERFORM OVERLAP 4-9
Perkin-Elmer Terminals A-10
PIC = X 2-7
PICTURE 2-7, 4-9
Printer file handling 4-5
PROCEDURE DIVISION 1-3, 2-4, 2-8, 4-3
PROCEDURE DIVISION Header . . . 3-4
PROGRAM-ID 2-10
PUNCT? 2-7

Quit (debug command) 4-4

RCISAM.COM 1-5
RDR: 2-2
READ BEYOND EOF 4-10
REBUILD D-1
REBUILD.COM 1-2
RECOVR.COB 1-2, 1-5
REDUNDANT OPEN 4-9
RELATIVE 4-6, 4-10
REWRITE; NO READ 4-9
RUN-TIME ERR: 4-8
RUNCOB.COM 1-1, 1-3, 2-4, 4-1
Runtime date C-2
Runtime day C-2
Runtime errors 4-8
Runtime line number C-2
Runtime System . . . . 1-3 to 1-4, 4-1
Runtime time C-2

SEG nn LOAD ERR 4-10
SEQCVT.COM 1-2, 1-4
SEQUENTIAL 1-4, 4-5
SOROC IQ Terminals A-11
SQUARO 1-6
SQUARO.COB 1-2, 1-4, 1-6
SQUARO.REL 1-6
Stack space . . . . 2-4, B-2 to B-3
Step (debug command) 4-4
STEXT.INT 1-3, 2-10
Subprogram calling B-1
Subprograms 3-4
SUBSCRIPT FAULT 4-9
Switches
  Compiler
    /D 2-4, 4-2
    /Fn 2-4
    /L 2-4
    /P 2-4

/R 2-4
/X 1-5, 2-4, 4-2
MS-LINK
/E 3-2
/G 3-2, 4-1
/M 3-3
/N 3-2
/P 1-4
/R 3-3
/S 3-3
/U 3-3
filename/N 3-2
filename/S 3-3
Switches - compiler 2-4
Switches - MS-LINK 3-2
Syntax, command line 2-1

Tab stops C-1
Terminal charts A-3
Terminal output 4-7
Time C-2
Trace (debug command) 4-5
Trace OFF (debug command) . . . 4-5
TTY: 2-2

VALUE OF clause 4-5
VALUE OF FILE-ID clause 4-6

Writing a CRT driver A-14

Zentec Zephr A-13

Microsoft COBOL reference manual

Information in this document is subject to change without notice and does not represent a commitment on the part of Microsoft, Inc. The software described in this document is furnished under a license agreement or non-disclosure agreement. The software may be used or copied only in accordance with the terms of the agreement. It is against the law to copy Microsoft COBOL on cassette tape, disk, or any other medium for any purpose other than the purchaser's personal use.

Copyright © Microsoft, Inc., 1980, 1981, 1982

LIMITED WARRANTY

MICROSOFT, Inc. shall have no liability or responsibility to purchaser or any other person or entity with respect to any liability, loss or damage caused or alleged to be caused directly or indirectly by this product, including but not limited to any interruption of service, loss of business or anticipatory profits or consequential damages resulting from the use or operation of this product. This product will be exchanged within twelve months from date of purchase if defective in manufacture, labeling, or packaging, but except for such replacement the sale or subsequent use of this program is without warranty or liability.

THE ABOVE IS A LIMITED WARRANTY AND THE ONLY WARRANTY MADE BY MICROSOFT, INC. ANY AND ALL WARRANTIES FOR MERCHANTABILITY AND/OR FITNESS FOR A PARTICULAR PURPOSE ARE EXPRESSLY EXCLUDED.

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

Microsoft COBOL and COBOL-80 are trademarks of Microsoft, Inc.

Document No. 8301-460-03
Part No. 00F11RM

Microsoft
COBOL Reference Manual

CONTENTS

Introduction

CHAPTER 1 Fundamental Concepts of COBOL

1.1 Character Set
1.2 Punctuation
1.3 Word Formation
1.4 Format Notation
1.5 Level Numbers and Data-Names
1.6 File-Names
1.7 Condition-Names
1.8 Mnemonic-Names
1.9 Literals
1.10 Figurative Constants
1.11 Structure of a Program
1.12 Coding Rules
1.13 Qualification of Names
1.14 COPY Statement

CHAPTER 2 Identification and Environment Divisions

2.1 Identification Division
2.2 Environment Division
2.2.1 Configuration Section
2.2.2 Input-Output Section
2.2.2.1 File-Control Entry
2.2.2.2 I-O Control Paragraph

CHAPTER 3 Data Division

3.1 Data Items
3.1.1 Group Items
3.1.2 Elementary Items
3.1.3 Numeric Items
3.2 Data Description Entry
3.3 Formats for Elementary Items
3.4 USAGE Clause
3.5 PICTURE Clause
3.6 VALUE Clause
3.7 REDEFINES Clause
3.8 OCCURS Clause
3.9 SYNCHRONIZED Clause
3.10 BLANK WHEN ZERO Clause
3.11 JUSTIFIED Clause
3.12 SIGN Clause
3.13 Level 88 Condition-Names
3.14 File Section, FD Entries

(Sequential I-O Only)
3.14.1 LABEL Clause
3.14.2 VALUE OF Clause
3.14.3 DATA RECORDS Clause
3.14.4 BLOCK Clause
3.14.5 RECORD Clause
3.14.6 CODE-SET Clause
3.14.7 LINAGE Clause
3.15 Working-Storage Section
3.16 Linkage Section
3.17 Screen Section
3.18 Data Division Limitations

CHAPTER 4 Procedure Division

4.1 Statements, Sentences, Procedures-Names
4.2 Organization of the Procedure Division
4.3 MOVE Statement
4.4 INSPECT Statement
4.5 Arithmetic Statements
4.5.1 SIZE ERROR Option
4.5.2 ROUNDED Option
4.5.3 GIVING Option
4.5.4 ADD Statement
4.5.5 SUBTRACT Statement
4.5.6 MULTIPLY Statement
4.5.7 DIVIDE Statement
4.5.8 COMPUTE Statement
4.6 GO TO Statement
4.7 STOP Statement
4.8 ACCEPT Statement
4.8.1 Format 1 ACCEPT Statement
4.8.2 Format 2 ACCEPT Statement
4.8.3 Format 3 ACCEPT Statement
4.8.3.1 Data Input Field
4.8.3.2 Data Inpuut and Data Transfer
4.8.3.3 WITH Phrase Summary
4.8.4 Examples Using the ACCEPT Statement
4.9 DISPLAY Statement
4.9.1 Position-spec
4.9.2 Identifier, Literal, and ERASE
4.10 PERFORM Statement
4.11 EXIT Statement
4.12 ALTER Statement
4.13 IF Statement
4.13.1 Conditions
4.14 OPEN Statement (Sequential I-O)
4.15 READ Statement (Sequential I-O)
4.16 WRITE Statement (Sequential I-O)
4.17 CLOSE Statement (Sequential I-O)
4.18 REWRITE Statement (Sequential I-O)
4.19 General Note on I/O Error Handling
4.20 STRING Statement
4.21 UNSTRING Statement

4.22 Dynamic Debugging Statements

CHAPTER 5 Inter-Program Communication

5.1 CALL Statement
5.2 EXIT PROGRAM Statement
5.3 CHAIN Statement
5.4 Procedure Division Header with CALL and CHAIN

CHAPTER 6 Table Handling by the Indexing Method

6.1 Index-Names and Index Items
6.2 SET Statement
6.3 Relative Indexing
6.4 SEARCH Statement - Format 1
6.5 SEARCH Statement - Format 2

CHAPTER 7 Indexed Files

7.1 Definition of Indexed File Organization
7.2 Syntax Considerations
7.2.1 RECORD KEY Clause
7.2.2 File Status Reporting
7.3 Procedure Division Statements for Indexed Files
7.4 READ Statement
7.5 WRITE Statement
7.6 REWRITE Statement
7.7 DELETE Statement
7.8 START Statement

CHAPTER 8 Relative Files

8.1 Definition of Relative File Organization
8.2 Syntax Considerations
8.2.1 RELATIVE KEY Clause
8.3 Procedure Division Statements for Relative Files
8.4 READ Statement
8.5 WRITE Statement
8.6 REWRITE Statement
8.7 DELETE Statement
8.8 START Statement

CHAPTER 9 DECLARATIVES and the USE Sentence

CHAPTER 10 Segmentation

Appendix A	Advanced Forms of Conditions
Appendix B	Table of Permissible MOVE Operands
Appendix C	Nesting of IF Statements
Appendix D	ASCII Character Set
Appendix E	Reserved Word List
Appendix F	PERFORM with VARYING and AFTER Clauses
Appendix G	Microsoft COBOL With Respect to the ANSI Standard

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)

Introduction

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

Each module of the COBOL 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 COBOL, it must provide at least Level 1 of the Nucleus, Table Handling, and Sequential I-O Modules.

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

Module	Features of Microsoft COBOL
<u>Nucleus</u>	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 for equality or inequalities (,>,=)
- 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)

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

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

0 through 9
A through Z
- (hyphen)

The following characters are used for punctuation:

(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 single space, except in non-numeric literals.

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

4. When the period, comma, plus, or minus characters are used in the PICTURE clause, they are governed solely by rules for report items.

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

6. 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 such formats are used.

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. In general, terminal periods are shown in formats in the manual because they are required; semicolons and commas are not usually shown because they are optional. To be 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 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 explained by separating the mutually exclusive choices by a vertical stroke, e.g.:

   AREA | AREAS is equivalent to { AREA / AREAS }

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.

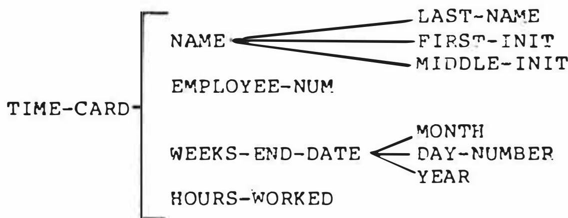
11. Optional elements may be indicated by parentheses instead of brackets, provided the lack of formality represents no substantial bar to clarity.

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 heirarchy 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, date 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 statement makes reference to a group item, the reference applies 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 (in part) 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    { data-name }
                { FILLER    }
```

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 Procedure 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 3.13.2.

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 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, but is completely defined by its own identity. 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, respectively. 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 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 specification DECIMAL-POINT IS COMMA, the functions of 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 data-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 (7-8 in punched cards)
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 COBOL 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 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 Microsoft COBOL 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 one for a paragraph-name, five for a data-name or condition-name. 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 Microsoft COBOL 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.COB 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.COB
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 COBOL 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 COBOL 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.

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

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.

In case the currency symbol is not supposed 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 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 Microsoft COBOL, 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 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 condition
- '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

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 Microsoft COBOL. 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:

```
       ┌ RECORD     ┐
SAME   │ SORT       │ AREA FOR filename...
       └ SORT-MERGE ┘
```

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 Microsoft COBOL 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. The maximum size of a group item is 4095 characters.

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 data item is an item in which one computer 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) 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; USAGE

IS INDEX. (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:

```
             { data-name }
level-number { 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.

<u>Group Item Format</u>

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

   (OCCURS-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 ITEMS (also called a character-string item)

```
              { data-name }
level-number  { 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)

```
             data-name
level-number 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

```
              { data-name }
level-number  { 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

```
                  { data-name }
level-number      { 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 COMP, VALUE ZERO.
     02  YEAR-TO-DATE COMPUTATIONAL.
```

3.4 USAGE CLAUSE

The USAGE clause specifies the form in which numeric 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:

```
                 { COMPUTATIONAL   }
        USAGE IS { 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 appreviated 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, optionally, 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 the actual or conceptual digit position contains 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, and is redundant if it is the rightmost character.

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 O - 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. CR and DB and + and - 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 -, or a string composed of one such character 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
$$$999	14	bb$014
--,---,999	-456	bbbbbb-456
$$$$$$	14	bbb$14

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 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 character, 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 6, 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.

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 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...]

The maximum size of a table is 4095 bytes.

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.

ELEMENT (1)	ARRAY, consisting of twelve characters; each item has 4 digits.
ELEMENT (2)	
ELEMENT (3)	

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 considered 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 definition) 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 | RECORDS IS | ARE OMITTED | 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 file ID expressed as a data-name or COBOL-type "quoted" literal of at most 16 characters. The general form is:

```
                         { data-name }
VALUE OF FILE-ID IS      {           }
                         { "literal" }
```

Examples:

```
VALUE OF FILE-ID "A:MASTER.ASM"      (CP/M DOS)
VALUE OF FILE-ID IS "D0:X201A.L"     (DTC)
VALUE OF FILE-ID "F0:INVNT.LST"      (Altair )
```

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 Utility Software Manual for file ID 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:

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

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, in the appropriate file declaration (FD) in the File Section.

3.14.4 BLOCK CLAUSE

The BLOCK CONTAINS clause is used to specify characteristics of physical records in relation to the concept of logical records. The general format is:

```
                                  { CHARACTERS }
BLOCK CONTAINS integer-2          {            }
                                  { RECORDS    }
```

Files assigned to PRINTER must not have a BLOCK clause in the associated FD entry. Furthermore, the BLOCK clause has no effect on disk files in this COBOL system, but it is examined for correct syntax. It is normally applicable to tape files, which are not supported by this COBOL.

When used, the size of a physical block is usually stated in RECORDS, except when the records are variable in size or exceed the size of a physical block; in these cases the size should be expressed in CHARACTERS.

When the BLOCK CONTAINS clause is omitted, it is assumed that records are not blocked. When neither the CHARACTERS nor the RECORDS option is specified, the CHARACTERS option is assumed. When the RECORDS option is used, the compiler assumes that the block size provides for integer-2 records of maximum size and then provides additional space for any required control characters.

3.14.5 RECORD CLAUSE

Since the size of each data record is defined fully by the set of data description entries constituting the record (level 01) declaration, this clause is always optional and documentary. The format of this clause is:

```
RECORD CONTAINS [integer-1 TO] integer-2 CHARACTERS
```

Integer-2 should be the size of the biggest record in the file declaration. If the records are variable in size, Integer-1 must be specified and equal the size of the smallest record. The sizes are given as character positions required to store the logical records.

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 record should include the SIGN IS SEPARATE clause and all data in the file should have DISPLAY USAGE.

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:

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

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

3.15 WORKING-STORAGE SECTION

The second section of the DATA DIVISION begins with the 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 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 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 has been coded for evey 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]
      [{[HIGHLIGHT]}]
      [{[BLINK]    }]


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


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

level-number and screen-name are subject to the same rules as in the group screen 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. AUTO and SECURE may only be given if PICTURE is specified.

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 ecountered, 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 encountered. 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.

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 of data

4. BELL will sound the terminal's audio alarm, if the terminal is so equiped.

5. HIGHLIGHT and BLINK are synonymous. They cause a screen item to appear on the CRT highlighted by flashing, high intensity, inverted video, or some other method provided by the particular type of terminal hardware in use.

6. BLANK WHEN ZERO causes a screen item to be displayed as spaces if its value is zero.

7. JUSTIFIED and JUST specify that operator-keyed data or data from a FROM field, USING field, or literal will be aligned with the right boundary of the screen item when it is displayed on the screen.

8. VALUE IS literal explicitly specifies the character string which should be displayed on the screen when the screen item being defined is referenced by a DISPLAY statement. A screen item for which VALUE is specified is ignored by all ACCEPT statements.

9. PICTURE specifies the format in which data is to be presented on the screen. It is coded according to the rules for WORKING-STORAGE PICTURE clauses described in section 3.2. During a DISPLAY statement, the contents of a FROM or USING field are MOVEd to an implicit temporary item with the specified PICTURE before being displayed on the screen. During an ACCEPT statement, the displayed contents of the field being entered are punctuated so as to conform with the given PICTURE format.

10. FROM, TO, and USING describe relationsips between a screen item and literals and/or fileds in the FILE, WORKING-STORAGE, and/or LINKAGE sections. On DISPLAY of a screen item, a MOVE occurs from any FROM or USING literal or field to a temporary item defined by the screen item's PICTURE. The resulting contents of the temporary item are then exhibited on the screen. On an ACCEPT of the screen item, the runtime system implicitly MOVEs the ACCEPTed data to any TO or USING field specified for the item.

11. AUTO specified that when a field has been filled by operator input, the cursor automatically skips to the next input field, rather than waiting for a terminator character to be typed. If there are no more input fields remaining, the ACCEPT is terminated.

12. SECURE suppresses the echoing of input characters. Instead, an asterisk is displayed for each data character ACCEPTed.

3.18 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 Microsoft COBOL 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 solve a given EDP problem. 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 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, or OVERFLOW 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 possesses a SIZE ERROR suffix, the statement is considered to be conditional rather than imperative. STRING or UNSTRING statements having an OVERFLOW 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.

Paragraph-names and section-names are procedure- names. Procedure-names follow the rules for name-formation. In addition, a procedure-name may consist only of digits. An all-digit procedure-name may not consist of more than 18 digits; if it has leading zeros, they are all significant.

Sections

A section is composed of one or more successive paragraphs, and must begin with a section-header. 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 a number of 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:

```
MOVE {data-name-1}  TO data-name-2   [data-name-3...]
     {literal    }
```

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 source and 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

```
                             { CHARACTERS                }
TALLYING data-name-2 FOR     { ALL | LEADING operand-3   }
[BEFORE | AFTER INITIAL operand-4]
```

and REPLACING-clause has the format

```
           { CHARACTERS                        }
REPLACING  { 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 <u>plus</u> 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-dependent.

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".
```

Note that if a size error occurs (in this case, it is apparent 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-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.

4.5.1 SIZE ERROR OPTION

If, after decimal-point alignment and any low-order rounding, the value of a calculated result exceeds the largest value which the receiving field is capable of holding, a 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 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:

```
       { numeric-literal }
ADD    { 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:

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

 {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 might seem somewhat unnatural, it is recommended that GIVING always be written.)

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       }  [SIZE-ERROR-clause]
{ arithmetic-expression }
```

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 [...DEPENDING ON data-name]

The simple form GO TO procedure-name changes the path of flow 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, 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. Three formats are available:

Format 1:

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

Format 2:

```
ACCEPT identifier-2
```

Format 3:

```
                                         SPACE-FILL
                                         ZERO-FILL
                                       { LEFT-JUSTIFY   }
                                       { RIGHT-JUSTIFY  }
ACCEPT position-spec identifier-3 [WITH{ 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 date or time information from the operating system clock. The next two formats of ehe 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 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 standard values may be obtained at execution time by use of the format 1 ACCEPT statement.

The formats of the standard values 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 0 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 - The ACCEPT...FROM LINE NUMBER statement is provided for compatibility, but in the Microsoft COBOL system, the value of LINE NUMBER is always zero.

ESCAPE KEY - a two digit code generated by the key that terminated the most recently executed format 3 or format 4 ACCEPT statement.
Indentifier-1 can be interrogated to deterine exactly which key was typed.
Input may be terinated by any of the following keys,

and cause the ESACPE KEY value to be set as shown:

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

All key codes are defined in the CRT driver for the terminal being used (refer to Appendix A of the Microsoft COBOL 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 functions 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 AUTP-SKIP option (i.e., no terminator key is struck), the ESCAPE KEY value is set to 00.

Identifier-1 should be an unsigned numberic interger whose length agrees with the content of the system-defined data item. If not, the standard rules for a MOVE govern storgage 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

```
  ┌                            ┐   ┌                            ┐
  │ LIN [ { + } integer-1 ]    │   │ COL [ { + } integer-3 ]    │
( │       { - }                │ , │       { - }                │ )
  │                            │   │                            │
  │      integer-2             │   │      integer-4             │
  └                            ┘   └                            ┘
```

The opening and closing parentheses and the comma 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 COMP). 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 character, or as 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.

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 input 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 COBOL 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 PROMPT 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 COBOL 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 alphanumeric 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 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 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: 05 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: NONEbbbb	Executing the ACCEPT
Final Contents of Receiving Field: NONEbbbb	Result

Example 2:

<table>
<tr><td>Receiving Field:
10 VEND-NAME PIC X(12).

Initial Contents:
ACME WIDGETS

ACCEPT Statement:
ACCEPT (1, 1) VEND-NAME
WITH PROMPT UPDATE.</td><td>Set-up
prior to executing</td></tr>
<tr><td>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:
XYZbbbbbbbbb</td><td>Executing
the ACCEPT</td></tr>
<tr><td>Final Contents
of Receiving Field:
XYZbbbbbbbbb</td><td>Result</td></tr>
</table>

Example 3:

Receiving Field: 05 CREDIT PIC S9(4)V99 Initial Contents: + 111111 ACCEPT Statement: ACCEPT (LIN + 4, CL - 3) CREDIT WITH PROMPT TRAILING-SIGN.	Set-up prior to executing
At Start of ACCEPT: 0000.00b Operator Enters 8: 0008.00b Operator Enters 7: 0087.00b 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 any screen item subordinate to screen-name. Screen items having only VALUE literals of FROM fields or literals have no effect on the operation of the ACCEPT statement. Each such 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. If an escape key is typed during data input, the entire ACCEPT is terminated and the ON ESCAPE statement is executed.

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 ot 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 define 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 terinated 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 Microsoft 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 discussing 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 complexities of file definition. The format of the DISPLAY statement is:

```
DISPLAY { { [position-spec] { identifier } } ...[UPON mnemonic-name] }
        {                   { literal    }                           }
        {                   { 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, or ERASE) will be processed in turn as described in the paragraphs below; then, if no position-spec 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 this 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 separating the two major bracketed groups are required. The 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 leteral 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 teh positioning and control clause 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

```
                  [ { integer   }           ]
PERFORM range     [ { data-name }     TIMES ]
```

Option 2

```
                            { index-name }
PERFORM range [VARYING      { 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.

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 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 provide 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. 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 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  { NEXT SENTENCE    } [ ELSE { statement(s)-2 } ]
              { statement(s)-1   } [      { NEXT SENTENCE  } ]
```

The "ELSE NEXT SENTENCE" phrase may be omitted if it immediately precedes the terminal period of the sentence.

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. The second series (ELSE part) is terminated by a sentence-ending period unless it is "ELSE NEXT SENTENCE", in which case more statements may be written before the period. If there is no ELSE part to an IF statement, then the first series of statements must be terminated by a sentence-ending period. 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 <u>any one</u> 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 GOTO 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 <u>EQUAL</u> TO, <u>LESS</u> THAN, and <u>GREATER</u> 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-test conditions, methods of performing comparisons will be discussed.

<u>Numeric Comparisons:</u> 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 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.

<u>Character Comparisons:</u> 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 characters when compared. Refer to Appendix IV for all ASCII character representations. If one operand is numeric and the other is not, it must be an integer and have an implicit or explicit USAGE IS DISPLAY.

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 (88), and sign test.

A class test condition has the following syntactical format:

```
                         { NUMERIC    }
    data-name IS [NOT]   { 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:

```
         { { INPUT  }                 }
   OPEN  { { I-O    } file-name...    } ...
         { { OUTPUT }                 }
         { { EXTEND }                 }
```

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 existent 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 Microsoft 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 (SEQUENTIAL 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, a run-time I/O error is processed.

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 definition, 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 user is reminded also that you READ file-name, but you WRITE record-name. The associated file must be open in the OUTPUT 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 specified.

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 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 suffix is used, the file is not re-openable 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 part 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 is allowed.

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 noted, 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 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 "Key" which, if found to match a like-numbered succession of sending characters, terminates the function for the current sending operand (and causes automatic switching to the next sending operand, if any).

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.

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 anv 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 identifier-1, 5, or 6 is evaluated only once at the beginning of the UNSTRING statement. Any subscripting associated with 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 detection 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

```
                  {                    { identifer }  }
EXHIBIT NAMED     { [position-spec]    { literal   }  }   ...[UPON mnemonic-name]
                  {                    { ERASE     }  }
```

produces a printout of values of the indicated literal, or data items in the format data-name = value. For more details concerning the syntax, see 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.

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 Microsoft 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:

```
        { literal      }
CHAIN   {              }   [USING identifier-2...]
        { identifier-1 }
```

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 coded as:

```
PROCEDURE DIVISION [CHAINING data-name...].
```

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 Microsoft 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 Microsoft 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 defiend by the USAGE IS INDEX phrase. And 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:

```
      { index-name-1 }         { index-name-2 }
  SET { index-item-1 } ...  TO { 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 indexes), 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.

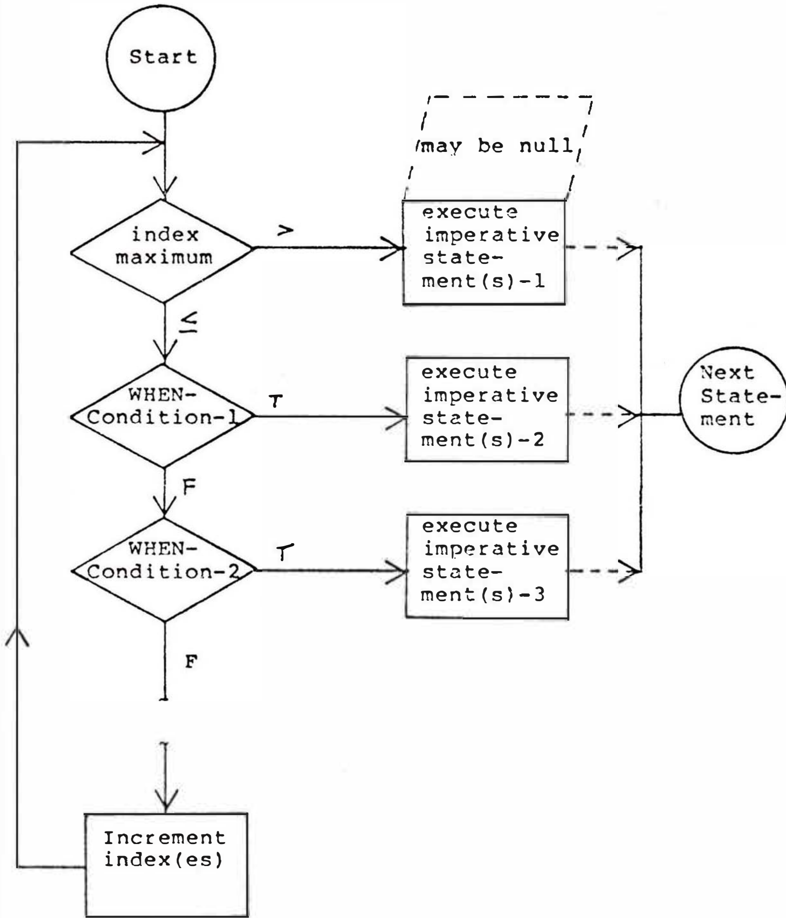
5. In a simple relational condition, the object (to the right of the equal sign) may be a literal or an identifier; the identifier must NOT be referenced in the KEY clause of the table or be indexed by the first index-name associated with the table. (The term identifier means data-name, including any 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 base" 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)	Sequence 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				
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 appropriate 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 organized files, except EXTEND is inapplicable.

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 existent 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 the 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 would 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:

```
                 ┌        ⎧ GREATER THAN  ⎫           ┐
START file-name  │ KEY IS ⎨ NOT LESS THAN ⎬ data-name │
                 └        ⎩ EQUAL TO      ⎭           ┘

     [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 sequence 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 sequentially- or indexed-organized 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.

The 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 STATEMENT 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 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 would 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 would 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.

8.8 START STATEMENT

The format of the START statement is the same for a Relative file as for an Indexed file:

```
                  [          { GREATER THAN  }              ]
START file-name   [ KEY IS   { NOT LESS THAN }  data-name-1 ]
                  [          { EQUAL TO      }              ]

    [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

<u>USE</u> <u>AFTER</u> STANDARD <u>EXCEPTION</u> | <u>ERROR</u> <u>PROCEDURE</u>

ON {file-name... | <u>INPUT</u> | <u>OUTPUT</u> | <u>I-O</u> | <u>EXTEND</u>}.

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 segmenation facility is provided to enable the exection of Microsoft COBOL programs which are larger than physical memory. When segmentation is used (that is, when any section header is the program contains a segment number) the entire PROCEDURE DIVISION must be written in section. Each section is assigned a segment number by a section header of the form:

section-name <u>SECTION</u> [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 number 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. Wach independent segment is treated as a program overlay. An independent segment is in its intitial 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 contined in a single independent segment.

3. A PERFORM statement in an independent 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 degment as the PERFORM statement.

APPENDIX A

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 take 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 B

Table of Permissible MOVE Operands

Source Operand	Receiving Operand in MOVE Statement					
	Numeric Integer	Numeric Non-integer	Numeric Edited	Alpha-numeric Edited	Alpha-numeric	Group
Numeric Integer	OK	OK	OK	OK (A)	OK (A)	OK (B)
Numeric Non-integer	OK	OK	OK			OK (B)
Numeric Edited				OK	OK	OK (B)
Alphanumeric Edited				OK	OK	OK (B)
Alphanumeric	OK (C)	OK (C)	OK (C)	OK	OK	OK (B)
Group	OK (B)	OK (B)	OK (B)	OK (B)	OK (B)	OK (B)

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 alphabetic items and vice versa.

APPENDIX C

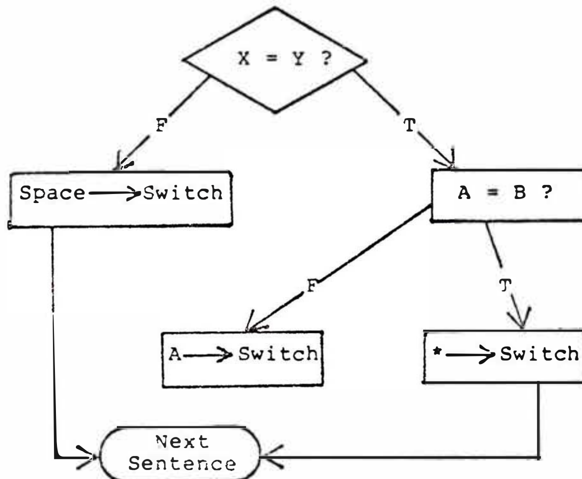
Nesting of IF Statements

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-KNOTO

    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 D

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 embedded positive sign); 173
Minus-zero (zero with embedded negative sign); 175

APPENDIX E

Reserved Words

+ indicates additional words required by Microsoft 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-SKIP

+BEEP
BEFORE
BLANK
BLOCK
BOTTOM
BY

CALL
CANCEL
CD
CF
CH
CHARACTER(S)
CLOCK-UNITS
CLOSE
COBOL
CODE
CODE-SET
+COL
COLLATING
COLUMN
COMMA
COMMUNICATION
COMP
COMPUTATIONAL
+COMPUTATIONAL-3
+COMP-3
COMPUTE
CONFIGURATION
CONTAINS
CONTROL(S)
COPY
CORR(ESPONDING)
COUNT
CURRENCY

DATA
DATE
DATE-COMPILED
DATE-WRITTEN
DAY
DEBUGGING
DEBUG-CONTENTS
DEBUG-ITEM
DEBUG-LINE
DEBUG-NAME
DEBUG-SUB-1
DEBUG-SUB-2
DEBUG-SUB-3
DECIMAL-POINT
DECLARATIVES
DELETE
DELIMITED
DELIMITER
DEPENDING
DESCENDING

DESTINATION
DE(TAIL)
DISABLE
+DISK
DISPLAY
DIVIDE
DIVISION
DOWN
DUPLICATES
DYNAMIC

EGI
ELSE
EMI
ENABLE
END
END-OF-PAGE
ENTER
ENVIRONMENT
EOP
EQUAL
+ERASE
ERROR
ESI
EVERY
EXCEPTION
+EXHIBIT
EXIT
EXTEND

FD
FILE
FILE CONTROL
+FILE-ID
FILLER
FINAL
FIRST
FOOTING
FOR
FROM

GENERATE
GIVING
GO
GREATER
GROUP

HEADING
HIGH-VALUE(S)

IDENTIFICATION
IF
IN
INDEX
INDEXED
INITIAL
INITIATE
INPUT
INPUT-OUTPUT
INSPECT
INSTALLATION
INTO
INVALID
IS
I-O
I-O-CONTROL

JUST(IFIED)
KEY

LABEL
LAST
LEADING
LEFT
+LEFT-JUSTIFY
LENGTH
+LENGTH-CHECK
LESS
LIMIT(S)
+LIN
LINAGE
LINAGE-COUNTER
LINE(S)
LINE-COUNTER
LINKAGE
LOCK
LOW-VALUE(S)

MEMORY
MERGE
MESSAGE
MODE
MODULES
MOVE
MULTIPLE
MULTIPLY

+NAMES
NATIVE
NEGATIVE
NEXT
NO
NOT
NUMBER

NUMERIC

OBJECT-COMPUTER
OCCURS
OF
OFF
OMITTED
ON
OPEN
OPTIONAL
OR
ORGANIZATION
OUTPUT
OVERFLOW
PAGE
PAGE-COUNTER
PERFORM
PF
PH
PIC(TURE)
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
SD
SEARCH
SECTION
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 F

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 G

Microsoft COBOL With Respect to the ANSI Standard

To understand how Microsoft 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, Microsoft COBOL 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, Microsoft COBOL does include the IBM COBOL Debug facility extensions to the ANSI standard.

Another extension Microsoft has incorporated in Microsoft COBOL 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

Accept . . . . 3-21 to 3-22, 3-24, 4-29
ACCEPT statement 1-7, 4-14
ACCESS clause 2-3, 7-1, 8-1
ADD statement 4-9
ADVANCING option 4-40
ALL phrase 4-44
Alphanumeric item 3-2, 3-4, 3-6
Alphanumeric-edited item 3-6
Alter 10-2
ALTER statement 4-34
ANSI level 1 . . . . 8, G-1
ANSI level 2 . . . . 8, G-1
Arithmetic expression 4-11
Arithmetic statements 4-7
ASCII-entry 2-3
AT END clause . . . . 4-1, 4-38, 7-3 to 7-4, 8-3
AUTHOR 2-1
Auto 3-24
Auto secure 3-22

Bell 3-22, 3-24
Binary item 3-2, 3-5
Blank line 3-22, 3-24
Blank screen 3-22
Blank when zero 3-22, 3-24
BLANK WHEN ZERO clause 3-15
Blink 3-22, 3-24
BLOCK clause 3-19

Call 5-2
CALL statement 5-1
Chain 5-2
Character comparisons 4-36
Character set 1-1
Class test condition 4-37
Close 7-3
CLOSE statement 4-41
CODE-SET clause 3-20
Column 3-24
Comments 1-12
Compound condition 4-35
COMPUTATIONAL 3-2, 3-5
COMPUTATIONAL-3 . . . . 3-2, 3-5
COMPUTE statement 4-11
Condition-name 1-3, 1-7, 3-16
Condition-name test 4-37
Conditional statements 4-1, 4-7
Conditions 1-1, 4-35
CONFIGURATION SECTION 2-2

Continuation line 1-8, 1-13
Control index 7-1
COUNT IN phrase 4-45
Crt screen formats 3-21
CURRENCY SIGN 2-2

Data description entry 3-3, 3-21
Data Division 1-6, 1-10
Data item 1-5, 3-1
DATA RECORDS clause 3-18
Data-name . . . . 1-3, 1-5 to 1-6, 3-3
DATE-COMPILED 2-1
DATE-WRITTEN 2-1
Debugging 9, 2-2, 4-46
Decimal item 3-4, 3-15
Decimal point 3-7
DECIMAL-POINT IS COMMA 1-8, 2-2
DECLARATIVES 4-2, 9-1
Declaratives 10-1
DELETE statement 7-6, 8-4
DELIMITED BY phrase 4-43
Display . . . . 3-21 to 3-22, 3-24
DISPLAY statement 1-7, 4-30
DIVIDE statement 4-11

Editing 4-29
Elementary item . . . . 1-5, 3-2, 3-4 to 3-5
Elementary screen items 3-21
Ellipsis 1-4
Environment Division 1-7, 1-10, 2-1
Escape 4-14, 4-29
Escape key 4-14
EXHIBIT statement 4-46
EXIT PROGRAM statement 5-2
EXIT statement 4-33
EXTEND phrase 4-38
External decimal item 3-2

FD entry 1-6, 1-12, 3-17
Figurative constants 1-8
File 1-5
File name 1-6
File Section 1-6, 3-17
FILE STATUS clause 2-4, 7-2
FILE STATUS data item 4-38
FILE-CONTROL 2-3
File-name 1-3
FILLER 3-3
Fixed segments 10-1
Floating string 3-8
Format notation 1-3
From 3-22, 3-24

General Formats 1-3
GIVING option 4-9
GO TO statement 4-13
Group 3-5
Group item 1-5, 3-1, 3-3, 3-13, 4-3

HIGH-VALUE 1-9
Highlight 3-22, 3-24

I-O 4-38
I-O error handling 4-42
I-O-CONTROL paragraph . . . . 2-3 to 2-4
Identification Division . . . 1-10, 2-1
IF statement 4-34
Imperative statements 4-1, 4-7
Independen segments 10-1
Index data-item 3-3, 3-5, 6-1
Index-name 6-1
Indexed I-O 9
Indexed-file organization . . 7-1
INPUT file 4-38
INPUT-OUTPUT SECTION 2-3
INSPECT statement 4-5
INSTALLATION 2-1
Inter-Program Communication . 9
Internal decimal item 3-2
INTO option 4-39
INVALID KEY clause . . . . . . 4-1, 7-3 to 7-6, 8-3 to 8-4

Just 3-22, 3-24
Justified 3-22
Justified 3-24
JUSTIFIED RIGHT clause 3-15

KEY clause 6-4
KEY IS clause 7-4

LABEL clause 3-17
Level 88 3-16
Level number 1-5, 1-12, 3-1, 3-3, 3-21
Level-number 3-22
Library 9
LINAGE clause 3-20
Line 3-22, 3-24
Line number 4-14
Linkage section 3-21, 5-3
Literals 1-7
LOCK suffix 4-41
LOW-VALUE 1-9

Main program 5-3
Memory 10-1
Memory requirements 10-1
Mnemonic-name 1-3, 1-7
Modules 8
MOVE statement 4-3
MULTIPLY statement 4-10

Nested IF C-1
Non-numeric literals 1-7
Nucleus 8
Numeric comparisons 4-36
Numeric item 3-2, 3-6

Numeric literals 1-8

OBJECT-COMPUTER 2-2
OCCURS clause 3-13
OMITTED 3-17
ON OVERFLOW clause 4-45
Open 7-3
OPEN statement 4-37
ORGANIZATION clause 2-3
OUTPUT file 4-38
OVERFLOW 4-1
Overlays 10-1

Packed decimal 3-2
Paragraph-name 4-2
Paragraphs 4-2
Parentheses 1-4
Perform 10-2
PERFORM statement 4-32
Pic 3-22
PICTURE 3-2
Picture 3-22, 3-24
PICTURE clause 3-6
Plus 3-22
POINTER phrase 4-43
PRINTER 1-7, 3-17, 3-19
Procedure Division 1-10, 4-1
Procedure division header . . 5-2
Procedure-name 1-3, 1-12, 4-2
PROGRAM-ID 2-1
Punctuation . . . . 1-1 to 1-3

Qualification 1-13
QUOTE 1-9

Range (PERFORM) 4-33
READ statement 4-38, 7-4, 8-3
READY TRACE statement 4-46
RECORD CONTAINS clause 3-19
RECORD KEY clause 7-2
Records 1-5
REDEFINES clause 3-12
Relative I-O 9
Relative indexing 6-2
RELATIVE KEY clause 8-2
RELATIVE KEY item 8-3
Relative organization 8-1
REPLACING clause 4-5
Report item 3-2, 3-4, 3-7
RESERVE clause 2-4
Reserved words . . . . 1-2 to 1-3, 1-12
RESET TRACE statement 4-46
REWRITE statement 4-42, 7-5, 8-3
ROUNDED option 4-8

Same 2-4
Screen data description entries 3-21
Screen section 4-14, 4-29

Screen-name 4-14, 4-29
SEARCH ALL statement 6-4
SEARCH statement 6-3
Section 10-1
Section header 10-1
Section-name 4-2
Sections 4-2
Secure 3-24
SECURITY 2-1
Segment 10-1
Segment number 10-1
Segmentation 10-1
SELECT entry . . . . 2-3, 7-1, 8-1 to 8-2
Sentences . . . . 4-1 to 4-2
Separator 1-2
Sequence number 1-12
Sequential I-O 8
SET statement 6-1
SIGN clause 3-2, 3-15
Sign test 4-37
Simple condition 4-35
SIZE ERROR option 4-1, 4-8
SOURCE-COMPUTER 2-2
SPACE 1-9
SPECIAL-NAMES 2-2
STANDARD 3-17
START statement 7-6, 8-4
Statements 4-1
STOP statement 4-13
STRING statement 4-43
Subprogram 5-3
Subscripts 3-13, 3-16
SUBTRACT statement 4-10
SYNCHRONIZED clause 3-14

Table Handling 8
TALLYING clause 4-5
To 3-22, 3-24, 4-29
TRACE mode 4-46

UNSTRING statement 4-44
USAGE clause 3-5
USE sentence 9-1
Using 3-22, 3-24, 4-29
USING list 3-21

Validation 4-29
Value 3-22, 3-24
VALUE IS clause 3-11, 3-21
VALUE OF clause 3-18
VARYING 6-3
Verbs 4-1

WHEN clause 6-4
Word 1-1, 1-3
Working-storage section 3-21
WRITE statement 4-39, 7-4, 8-3

utility software package reference manual

for 8080 microprocessors

Microsoft, Inc.
Microsoft Building
10700 Northup Way
Bellevue, WA 98004

Information in this document is subject to change without notice and does not represent a commitment on the part of Microsoft, Inc. The software described in this document is furnished under a license agreement or non-disclosure agreement. The software may be used or copied only in accordance with the terms of the agreement. It is against the law to copy The Utility Software Package on cassette tape, disk, or any other medium for any purpose other than purchaser's personal use.

Copyright © Microsoft, Inc., 1981

LIMITED WARRANTY

MICROSOFT, Inc. shall have no liability or responsibility to purchaser or to any other person or entity with respect to any liability, loss or damage caused or alleged to be caused directly or indirectly by this product, including but not limited to any interruption of service, loss of business or anticipatory profits or consequential damages resulting from the use or operation of this product. This product will be exchanged within twelve months from date of purchase if defective in manufacture, labeling, or packaging, but except for such replacement the sale or subsequent use of this program is without warranty or liability.

THE ABOVE IS A LIMITED WARRANTY AND THE ONLY WARRANTY MADE BY MICROSOFT, INC. ANY AND ALL WARRANTIES FOR MERCHANTABILITY AND/OR FITNESS FOR A PARTICULAR PURPOSE ARE EXPRESSLY EXCLUDED.

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

CP/M is a registered trademark of Digital Research.
The Utility Software Package, MACRO-80, LINK-80, CREF-80, and LIB-80 are trademarks of Microsoft, Inc.

Document No. 8401-343-04
Catalog No.
Part No.

Contents

Chapter 1 Introduction

1.1 Contents of the Utility Software Package 1-1
1.2 System Requirements 1-2
1.3 Whom Is the Utility Software Package for? 1-2
1.4 A Word about This Manual 1-3
1.5 Overview 1-4

Chapter 2 Features of the Utility Software Package

2.1 Two Assembly Languages 2-2
2.2 Relocatability 2-2
2.3 Macro Facility 2-2
2.4 Conditional Assembly 2-3
2.5 Utility Programs 2-3

Chapter 3 Programming with the Utility Software Package

3.1 Source File Organization 3-1
3.2 Symbols 3-3
3.3 Opcodes and Pseudo-ops 3-9
3.4 Arguments: Expressions 3-10
3.4.1 Operands 3-10
3.4.2 Operators 3-14

Chapter 4 Assembler Features

4.1 Single-Function Pseudo-ops 4-1
4.2 Macro Facility 4-36
4.3 Conditional Assembly Facility 4-48

Chapter 5 Running MACRO-80

5.1 Invoking MACRO-80 5-2
5.2 MACRO-80 Command Line 5-2
5.3 MACRO-80 Listing File Formats 5-13
5.4 Error Codes and Messages 5-15

Chapter 6 LINK-80 Linking Loader

6.1 Invoking LINK-80 6-1
6.2 LINK-80 Commands 6-2
6.2.1 Filenames 6-3
6.2.2 Switches 6-4
6.3 Error Messages 6-19

Chapter 7 CREF-80 Cross Reference Facility

7.1 Creating a CREF Listing 7-1
7.2 CREF Listing Control Pseudo-ops 7-3

Chapter 8 LIB-80 Library Manager

8.1 Sample LIB-80 Session 8-2
8.2 LIB-80 Commands 8-3

Appendix A Compatibility with Other Assemblers

Appendix B The Utility Software Package with TEKDOS

B.1 TEKDOS Command Files B-1
B.2 MACRO-80 B-1
B.3 CREF-80 B-2
B.4 LINK-80 B-2

Appendix C ASCII Character Codes

Appendix D Format of LINK Compatible Object Files

Appendix E Table of MACRO-80 Pseudo-ops

Appendix F Table of Opcodes

F.1 Z80 Opcodes F-1
F.2 8080 Opcodes F-3

Index

Contents

Chapter 1 Introduction

1.1 Contents of the Utility Software Package 1-1
1.2 System Requirements 1-2
1.3 Whom Is the Utility Software Package for? 1-2
 Books on Assembly Language Programming 1-2
1.4 A Word about This Manual 1-3
 Organization 1-3
 Syntax Notation 1-3
1.5 Overview 1-4

CHAPTER 1

INTRODUCTION

Welcome to the world of Utility Software Package programming. During the course of this manual, we will learn what the Utility Software Package is, why you use it, and how to use it.

1.1 CONTENTS OF THE UTILITY SOFTWARE PACKAGE

One diskette with the following files:

M80.COM - MACRO-80 Macro Assembler program
L80.COM - LINK-80 Linking Loader program
CREF80.COM - Cross-Reference Facility
LIB.COM - Library Manager program
(CP/M versions only)

One Manual

The Utility Software Package Reference Manual

IMPORTANT

Always make backup copies of your diskettes before using them.

1.2 SYSTEM REQUIREMENTS

MACRO-80 requires about 19K of memory, plus about 4K for buffers. LINK-80 requires about 14K of memory. CREF-80 requires about 6K of memory. LIB-80 requires about 5K of memory. The operating system usually requires about 6K bytes of memory. So a minimum system requirement for the Utility Software Package is 29K bytes of memory. While it is possible to run Utility Software Package programs with only one disk drive, we recommend strongly that you have two disk drives available.

1.3 WHOM IS THE UTILITY SOFTWARE PACKAGE FOR?

The Utility Software Package is a complete assembly language development system with powerful features that support advanced assembly language programming skills. This manual describes the Utility Software Package thoroughly, but the descriptions assume that the reader understands assembly language programming and has experience with an assembler.

If you have never programmed in assembly language, we suggest that you gain some experience on a simpler assembler.

Books on Assembly Language Programming

We can also recommend the following books for basic instruction in assembly language programming:

Leventhal, Lance A. *8080A/8085 Assembly Language Programming.* Berkeley: Osborne/McGraw-Hill, 1978.

Leventhal, Lance A. *Z80 Assembly Language Programming.* Berkeley: Osborne/McGraw-Hill, 1979.

Zaks, Rodnay. *Programming the Z80.* Second edition. Berkeley: Sybex, 1980.

1.4 A WORD ABOUT THIS MANUAL

Organization

In front of each chapter is a contents page that expands the entries on the contents page at the beginning of the manual. Chapter 1 gives introductory, background, and overview information about the Utility Software Package. Chapters 2-8 describe the use and operation of the Utility Software Package programs. The manual concludes with several appendices which contain some helpful reference information.

Syntax Notation

The following notation is used throughout this manual in descriptions of command and statement syntax:

- [ ] Square brackets indicate that the enclosed entry is optional.
- < > Angle brackets indicate user entered data. When the angle brackets enclose lower case text, the user must type in an entry defined by the text; for example, <filename>. When the angle brackets enclose upper case text, the user must press the key named by the text; for example, <RETURN>.
- { } Braces indicate that the user has a choice between two or more entries. At least one of the entries enclosed in braces must be chosen unless the entries are also enclosed in square brackets.
- ... Ellipses indicate that an entry may be repeated as many times as needed or desired.
- CAPS Capital letters indicate portions of statements or commands that must be entered, exactly as shown.

All other punctuation, such as commas, colons, slash marks, and equal signs, must be entered exactly as shown.

1.5 OVERVIEW

The Utility Software Package is an assembly language programming system that parallels the design and programming power of assemblers and related software on big computers. Consequently, the design and use of the Utility Software Package involves traits and methods that may be new to you. As explained earlier, we assume that you have some experience in assembly language programming. Your knowledge of when and why to use particular operation codes and pseudo-operations is the base on which you can build your knowledge of the Utility Software Package.

One word of caution: some terms used in this manual may be familiar to you from other sources. Be sure to notice especially how familiar terms are used in the Utility Software Package so that you are not confused or misled.

The Utility Software Package programming relies on two important software programs -- an assembler and a linking loader. To develop an assembly language program that runs on your computer, you must use both the assembler and the linking loader. The whole process is diagrammed on the facing page. The numbers on the diagram correspond to the numbers in the explanations below.

1. You create an assembly language source program using some editor.

2. You assemble your source program using the MACRO-80 macro assembler. The result is a file that contains intermediate object code. This intermediate code is closer to machine code than your source code, but cannot be executed.

3. You link and load separately assembled file(s) into a single program file using the LINK-80 linking loader. LINK-80 converts the file(s) of intermediate code into a single file of true machine code which can be executed from the operating system.

These are only the basics of the whole process. This two step process of converting a source file to an executable program allows you to manipulate your programs to save you time and to extend your programs usefulness in the following ways:

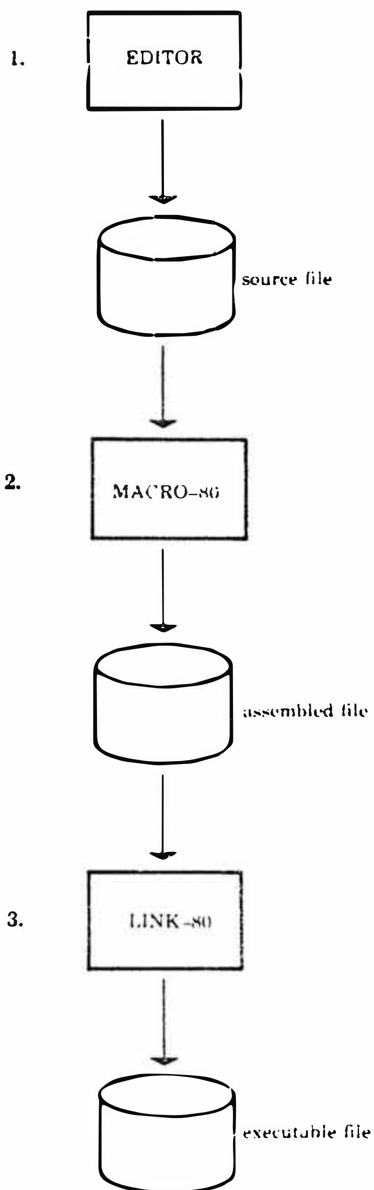


Figure 1.1: Developing Assembly Language Programs

First, you can break your program in convenient parts called modules. You can manipulate these modules at will. You can assemble the modules individually. You fix only those that do not work right and reassemble them. This saves you time.

Second, you can manipulate the placement of modules in memory, subject to certain restrictions; or allow LINK-80 to place modules for you. (This trait is described below under the fourth trait.)

Third, you can use assembled modules in other programs or in variations of the original program because there is no permanent connection among the modules. This saves you recoding time if a part of a program performs some useful, often-repeated task.

Whenever you want to combine assembled modules into an executable program, you use the LINK-80 linking loader. If you simply tell LINK-80 the modules you want combined, it loads them end-to-end in memory. But you have an additional choice. You can set up a direct connection between a statement in one module and a statement inside another module. This direct connection (or "link") means that a value (usually a program address) in one module can be used in another module exactly at the point required.

LINK-80 creates the links between modules. You give LINK-80 the signals it needs to create these links. The signals are called symbols, specifically EXTERNAL symbols and PUBLIC symbols. An EXTERNAL symbol signals LINK-80 that you want it to link a value from another module into this point in the program. The value to be linked-in is defined by a PUBLIC symbol, which is a signal that directs LINK-80 to the correct module and statement line. LINK-80 then links the PUBLIC symbol's value to the EXTERNAL symbol, then continues loading the module with the EXTERNAL symbol. The diagram below suggests this process.

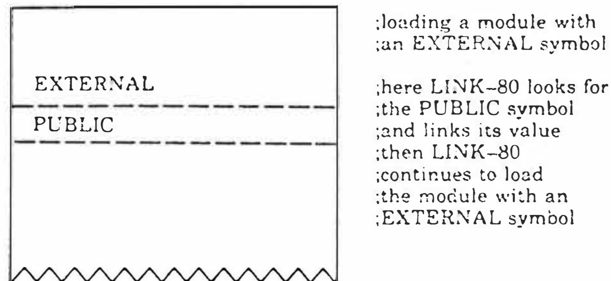


Figure 1.2: PUBLIC symbol linked into module at EXTERNAL

Fourth, modules can be assembled into different modes, even within a single module. The four modes are Absolute, Data-relative, Code-relative, and COMMON-relative. The absolute mode is similar the code produced by most small system assemblers. The code is assembled at fixed addresses in memory. The other three modes are very different and are the reason you can place modules anywhere in memory. Each of the three relative modes assembles to a separate segment. The addresses within each segment are relative addresses. This means the first instruction byte of a segment is given a relative address of 0, the second byte is given relative address 1, and so on. When LINK-80 loads the module, it changes the relative addresses in the segments to fixed addresses in memory. The relative addresses are offsets from some fixed address that LINK-80 uses. For the first module loaded, this address is 103H under the CP/M operating system. Thus, relative addresses in the first module are offsets from 103H. The second module is loaded at the end of the first, and the relative addresses are offsets from the last address in the first module. Subsequent modules are loaded (and offset) similarly. You can change the default start address for the first module at link time. Then, the relative addresses become offsets from the fixed address you specify.

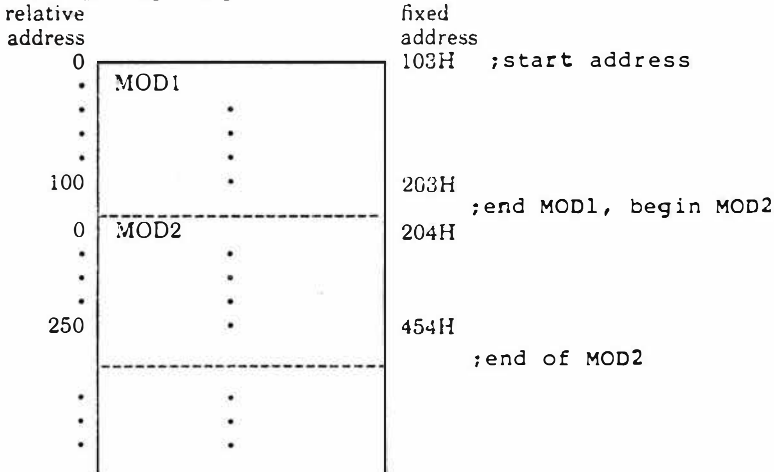
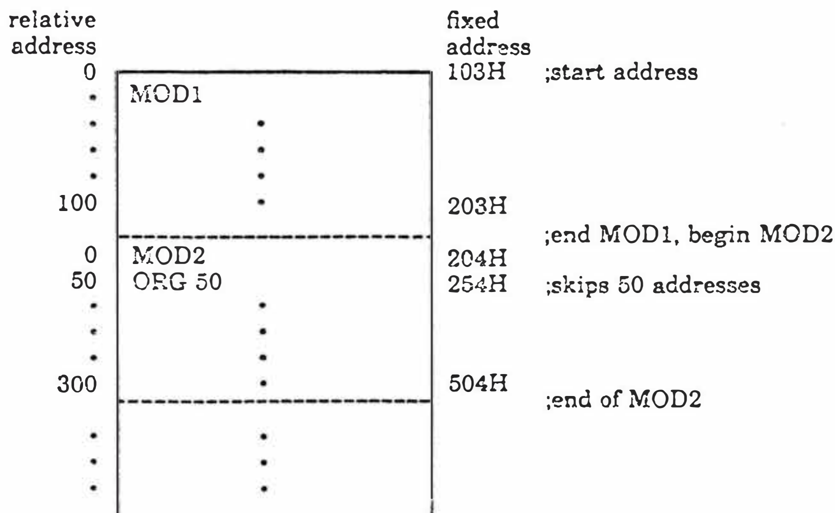


Figure 1.3: Loading Changes Relative Addresses to Fixed

One effect of this relative addressing method is that ORG statements become very different creatures. For the relative segments, the ORG statement specifies an offset rather than a fixed address (as most assemblers do -- ORG specifies a fixed address in the absolute segment). Thus, a relative segment with an ORG statement would skip over a specified number of addresses before beginning to load the rest of the code in that segment.



You should read carefully the description of ORG found in Chapter 4.

The ability to manipulate the placement of modules in memory, with some restrictions (see Chapter 6), derives from the assembler giving relative addresses instead of absolute addresses. This ability to manipulate module placement in memory is called relocatability; the modules are relocatable: the intermediate code produced by the assembler for the three relative segments is called relocatable code. That is why assembled modules are given the filename extension .REL, and these assembled files are called REL files.

Each mode serves a different purpose. The absolute mode contains code you want placed in specific memory addresses. Each relative mode is loaded into memory as a separate segment. The data-relative segment contains data items and any code that may change often and should only be placed in RAM. The code-relative segment contains code that will not change and therefore is suitable for ROM and PROM. The COMMON-relative segment contains data items that can be shared by more than one module.

Source statements in these modes take on the traits of their mode. The symbols and expressions in statements are evaluated by the assembler according to the mode in which they are found and the type of data and other entries that define the symbol or make up the parts of an expression. The mode traits attributed to a symbol or expression are called, appropriately, its Mode; that is, a symbol or expression is absolute, data-relative, code-relative, or COMMON-relative. This concept of mode is important because it is the source of both flexibility and complexity. If all

source statements are assembled in absolute mode, symbols and expressions always have absolute values, and using absolute symbols and expressions is not complex. The problem with absolute mode is that relocatability is possible only through the most complex and time consuming of techniques. Absolute mode effectively reduces your ability to reuse code in a new program.

The relative modes (data, code, and COMMON) are the basis of relocatability and, therefore, of the flexibility to manipulate modules. The complexity is that relative symbols and relative expressions are much more difficult to evaluate. In fact, the assembler must pass through the source statements twice to assemble a module. During the first pass, the assembler evaluates the statements and expands macro call statemnts, calculates the amount of code it will generate, and builds a symbol table where all symbols and macros are assigned values. During the second pass, the assembler fills in the symbol and expression values from the symbol table, expands macro call statements, and emits the intermediate code into a REL file.

When the REL files are given to LINK-80, the segments are linked together and loaded into fixed memory addresses. The relative addresses are converted to absolute addresses. The fixed addresses are assigned to the relative segments in the order: COMMON-relative and data-relative, then code-relative. The relative segments are loaded relative to default address 103H under CP/M. (The addresses 100H-102H are used for a jump to the start address of the first program instruction, which is normally the first address following the COMMON and data area.)

When LINK-80 is finished linking modules together and assigning addresses, the result can be saved in a file that is executable from the operating system. Executing the program is then as simple as entering an operating system command, so these linked and loaded files are called command files.

This short overview should give you a general idea of the workings and processes of the Utility Software Package. Short descriptions of all the Utility Software Package programs are given in the next chapter. Detailed descriptions are given in the rest of this manual. Therefore, the information contained in this overview will be repeated in fuller detail elsewhere in this manual.

As an aid to the description in the next chapter and the rest of this manual, the next page contains an expanded version of the diagram at the beginning of this overview. The expanded diagram shows the relationships among all the programs in the Utility Software Package.

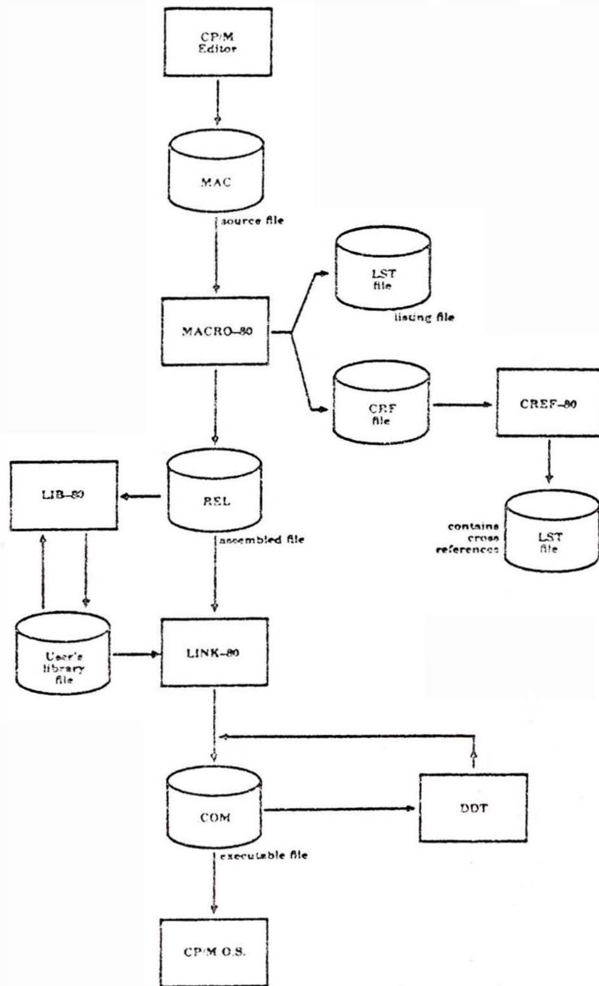


Figure 1.5: Relationships among programs

Contents

Chapter 2 Features of the Utility Software Package

2.1 Two Assembly Languages 2-2
2.2 Relocatability 2-2
2.3 Macro Facility 2-2
2.4 Conditional Assembly 2-3
2.5 Utility Programs 2-3
LINK-80 Linking Loader 2-3
CREF-80 Cross Reference Facility 2-4
LIB-80 Library Manager 2-4

CHAPTER 2

FEATURES OF THE UTILITY SOFTWARE PACKAGE

The Utility Software Package is an Assembly Language Development System that assembles relocatable code from two assembly languages, supports a macro facility and conditional assembly, and provides several utility programs that enhance program development.

WHAT IS AN UTILITY SOFTWARE PACKAGE?

An Utility software package is more than an assembler. An Utility Software Package is a series of related utility programming tools:

- for assembling an assembly language source file,
- for linking several assembled modules into one program,
- for creating library files of subroutines (also assembled modules),
- for creating cross-reference listings of program symbols,
- for testing and debugging binary (machine executable) program files,

Microsoft's Utility Software Package provides versions of these tools that make the Utility Software Package extremely powerful and useful as a program development system. Each tool in the Utility Software Package is described in detail in its own chapter.

2.1 TWO ASSEMBLY LANGUAGES

The assembler in your Utility Software Package supports two assembly languages. Microsoft's MACRO-80 macro assembler supports both 8080 and Z80 mnemonics.

2.2 RELOCATABILITY

MACRO-80 can produce modules of relocatable code. Also, like many assemblers, the MACRO-80 assembler can produce absolute code. The key advantage of relocatability is that programs can be assembled in modules. Then, within certain restrictions described in Chapter 6, the modules can then be located almost anywhere in memory.

Relocatable modules also offer the advantages of easier coding and faster testing, debugging, and modifying. In addition, it is possible to specify segments of assembled code that will later be loaded into RAM or into ROM/PROM.

Relocatability will be discussed further under Section 3.2, Symbols.

2.3 MACRO FACILITY

The MACRO-80 assembler supports a complete, Intel standard macro facility. The macro facility allows a programmer to write blocks of code for a set of instructions used frequently. The need for recoding these instructions is eliminated.

The programmer gives this block of code a name, called a macro. The instructions are the macro definition. Each time the set of instructions is needed, instead of recoding the set of instructions, the programmer simply "calls" the macro. MACRO-80 expands the macro call by assembling the block of instructions into the program automatically. The macro call also passes parameters to the assembler for use during macro expansion. The use of macros reduces the size of a source module because the macro definitions are stored in disk files and come into the module only when needed during assembly.

Macros can be nested, that is, a macro can be called from inside another macro. Nesting of macros is limited only by memory.

2.4 CONDITIONAL ASSEMBLY

MACRO-80 also supports conditional assembly. The programmer can determine a condition under which portions of the program are either assembled or not assembled. Conditional assembly capability is enhanced by a complete set of conditional pseudo operations that include testing of assembly pass, symbol definition, and parameters to macros. Conditionals may be nested up to 255 levels.

2.5 UTILITY PROGRAMS

Three utility programs provide the additional support needed to develop powerful and useful assembly language programs: LINK-80 Linking Loader, LIB-80 Library Manager, and CREF-80 Cross Reference Facility.

LINK-80 Linking Loader

The Microsoft LINK-80 Linking Loader is used to convert the assembled module (.REL file) into an executable module (.COM file). The .REL file is not an executable file.

LINK-80 can also be used to:

- load, link, and run one or more modules
- load relocatable programs at user-specified locations
- load program areas and data areas into separate memory locations

While performing these tasks, LINK-80 resolves external references between modules (that is, any program that calls an external value, something defined in a different program or module, will have the outside references filled at link time by LINK-80), and saves the executable object (.COM) file on disk, so it can be run from the operating system.

These load capabilities mean that the assembled program may be linked with the user's library to add routines to one of the high-level langauge runtime libraries. Assembled programs can be linked to high-level language programs -- COBOL-80 and FORTRAN-80, for example -- as well as to MACRO-80 programs.

CREF-80 Cross Reference Facility

The CREF-80 Cross Reference Facility processes a cross reference file generated by MACRO-80. The result is a cross reference listing that can aid in the debugging of your program.

LIB-80 Library Manager (CP/M versions only)

LIB-80 is designed as a runtime library manager for CP/M versions of the Utility Software Package. LIB-80 may also be used to create your own library of assembly language subroutines.

LIB-80 creates runtime libraries from assembly language programs that are subroutines to COBOL, FORTRAN, and other assembly language programs. The programs collected by LIB-80 may be special modules created by the programmer or modules from an existing library. With LIB-80, you can create specialized runtime libraries for whatever execution requirements you design.

Contents

Chapter 3 Programming with the Utility Software Package

3.1 Source File Organization 3-1
- File Organization 3-1
- Statement Line Format 3-1
- Comments 3-2

3.2 Symbols 3-3
- LABEL: 3-4
- PUBLIC 3-5
- EXTERNAL 3-6
- Modes 3-7

3.3 Opcodes and Pseudo-ops 3-9

3.4 Arguments: Expressions 3-10

3.4.1 Operands 3-10
- Numbers 3-10
- ASCII Strings 3-11
- Character Constants 3-11
- Symbols in Expressions 3-12
- Current Program Counter Symbol 3-13
- 8080 Opcodes as Operands 3-13

3.4.2 Operators 3-14

CHAPTER 3

PROGRAMMING WITH THE UTILITY SOFTWARE PACKAGE

This chapter describes what the user needs to know to create MACRO-80 macro assembler source files. Source files are created using a text editor, such as CP/M ED. The Utility Software Package does not include a text editor program.

Source files are assembled using the procedures described in Chapter 4.

3.1 SOURCE FILE ORGANIZATION

File Organization

A MACRO-80 macro assembler source file is a series of lines written in assembly language. The last line of the file must be an END statement. Matching statements (such as IF...ENDIF) must be entered in the proper sequence. Otherwise, lines may appear in any order the programmer designs.

Statement Line Format

Source files input to the MACRO-80 macro assembler consist of statement lines divided into parts or "fields."

```
      BUF:        DS          1000H       ;create a buffer
       ^           ^            ^           ^
       |           |            |           |
     SYMBOL    OPERATION    ARGUMENT     COMMENT
```

SYMBOL field contains one of the three types of symbol (LABEL, PUBLIC, and EXTERNAL), followed by a colon unless it is part of a SET, EQU, or MACRO statement.

OPERATION field contains an OPCODE, a PSEUDO-OP, a MACRO name, or an expression.

ARGUMENT field contains expressions (specific values, variables, register names, operands and operators).

;COMMENT field contains comment text always preceded by a semicolon.

All fields are optional. You may enter a completely blank line.

Statement lines may begin in any column. Multiple blanks or tabs may be inserted between fields to improve readability, but at least one space or tab is required between each field.

Comments

A MACRO-80 macro assembler source line is basically an Operation and its Argument. Therefore, the MACRO-80 macro assembler requires that a COMMENT always begin with a semicolon. A COMMENT ends with a carriage return.

For long comments, you may want to use the .COMMENT pseudo-op to avoid entering a semicolon for every line. See the File Related Pseudo-ops section of Chapter 4 for the description of .COMMENT.

3.2 SYMBOLS

Symbols are simply names for particular functions or values. Symbol names are created and defined by the programmer.

Symbols in the Utility Software Package belong to one of three types, according to their function. The three types are LABEL, PUBLIC, and EXTERNAL. All three types of symbols have a MODE attribute that corresponds to the segment of memory the symbol represents. Refer to the section on modes following the description of symbol types.

All three types of symbols have the following characteristics:

1. Symbols may be any length, but the number of significant characters passed to the linker varies with the type of symbol:

   a. for LABELs, only the first sixteen characters are significant.

   b. for PUBLIC and EXTERNAL symbols, only the first six characters are passed to the linker.

   Additional characters are truncated internally.

2. A legal symbol name may contain the characters:

   A-Z 0-9 $. ? @ _

3. A symbol may not start with a digit or an underline

4. When a symbol is read, lower case is translated into upper case, so you may enter the name using either case or both.

LABEL:

A LABEL: is a reference point for statements inside the program module where the label appears. A LABEL: sets the value of the symbol LABEL to the address of the data that follows. For example, in the statement:

```
        BUF:     DS     1000H
```

BUF: equals the first address of the 1000H byte reserved space.

Once a label is defined, the label can be used as an entry in the ARGUMENT field. A statement with a label in its argument loops to the statement line with that label in its SYMBOL field, which is where the label is defined. The label's definition replaces the label used in an ARGUMENT field. For example,

```
        STA     BUF
```

sends the value in the accumulator to the area in memory represented by the label BUF.

A LABEL may be any legal symbol name, up to 16 characters long.

If you want to define a LABEL, it must be the first item in the statement line. 8080 and Z80 labels must be followed immediately by a single colon (no space), unless the LABEL is part of a SET or EQU statement. (If two colons are entered, the "label" becomes a PUBLIC symbol. See PUBLIC Symbols below.)

PUBLIC

A PUBLIC symbol is defined much like a LABEL. The difference is that a PUBLIC symbol is available as a reference point for statements in other program modules, too.

A symbol is declared PUBLIC by:

two colons (::) following the name. For example,

```
FOO::       RET
```

one of the pseduo-ops PUBLIC, ENTRY, or GLOBAL. For example,

```
PUBLIC    FOO
```

See the Data Definition and Symbol Definition Pseudo-ops section in Chapter 4 for descriptions of how to use these pseudo-ops.

The result of both methods of declaration is the same. Therefore,

```
FOO::       RET
```

is equivalent to

```
PUBLIC     FOO
FOO:       RET
```

EXTERNAL

An EXTERNAL symbol is defined outside the program module where it appears. An EXTERNAL symbol is defined as a PUBLIC symbol in another, separate program module. At link time (when the LINK-80 Linking Loader is used), the EXTERNAL symbol is given the value of the PUBLIC symbol in the other program module. For example:

MOD1

```
FOO::     DB     7       ;PUBLIC FOO = 7
```

MOD2

```
          BYTE EXT     FOO       ;EXTERNAL FOO
```

At link time, LINK-80 goes to the address of PUBLIC FOO and uses the value there (7) for EXTERNAL FOO.

A symbol is declared EXTERNAL by:

1. two pound signs (##) following a reference to a symbol name. For example:

   ```
   CALL     FOO##
   ```

   declares FOO as a two-byte symbol defined in another program module.

2. one of the pseudo-ops EXT, EXTRN, or EXTERNAL for two-byte values. For example:

   ```
   EXT     FOO
   ```

   declares FOO as a two-byte value defined in another program module.

3. one of the pseudo-ops BYTE EXT, BYTE EXTERN, or BYTE EXTERNAL for one-byte values. For example:

   ```
   BYTE EXT     FOO
   ```

   declares FOO as a one-byte value defined in another program module.

   See the Symbol Definition Pseudo-ops section in Chapter 4 for descriptions of how to use these pseudo-ops.

As for PUBLIC symbols, the result of both methods of declaration is the same. Therefore,

```
        CALL    FOO##
```

is equivalent to

```
        EXT     FOO
        CALL    FOO
```

MODES

A symbol is referenced by entering its name in the ARGUMENT field of a statement line. When a symbol is referenced, the value of the symbol (derived from the instruction which defines the symbol) is substituted for the symbol name and used in the operation.

The value of a symbol is evaluated according to its program counter (PC) mode. The PC mode determines whether a section of a program will be loaded into memory at addresses predetermined by the programmer (absolute mode), or at relative addresses that change depending on the size and number of programs (code relative mode) and amount of data (data relative mode), or at addresses shared with another program module (COMMON mode). The default mode is Code Relative.

Absolute Mode: Absolute mode assembles non-relocatable code. A programmer selects Absolute mode when a block of program code is to be loaded each time into specific addresses, regardless of what else is loaded concurrently.

Data Relative Mode: Data Relative mode assembles code for a section of a program that may change and therefore must be loaded into RAM. This applies to program data areas especially. Symbols in Data Relative Mode are relocatable.

Code Relative Mode: Code (program) Relative mode assembles code for sections of programs that will not be changed and therefore can be loaded into ROM/PROM. Symbols in Code Relative Mode are relocatable.

COMMON Mode: COMMON mode assembles code that is loaded into a defined common data area. This allows program modules to share a block of memory and common values.

To change mode, use a PC mode pseudo-op in a statement line. The PC mode pseudo-ops are:

```
        ASEG    Absolute mode
        DSEG    Data Relative mode
        CSEG    Code Relative mode--default mode
        COMMON  COMMON mode
```

These pseudo-ops are described in detail in the PC Mode Pseudo-ops section of Chapter 4.

This PC mode capability in the MACRO-80 macro assembler allows a programmer to develop assembly language programs that can be relocated. Many assembly language programmers may have learned always to set an Origin statement at the beginning of every module, subroutine, or main assembly language program. Under MACRO-80 this mode of addressing is called Absolute mode because hard (or actual addresses) are specified beginning, especially, with the Origin statement.

MACRO-80 has two other, "relative" modes of addressing available, called Code (Program) relative and Data relative. Segments of code written in these two modes are relocatable. Relocatable means the program module can be loaded starting at any address in available memory, using the /P and /D switches (special commands) in LINK-80.

3.3 OPCODES AND PSEUDO-OPS

Opcodes are the mnemonic names for the machine instructions. Pseudo-ops are directions to the assembler, not the microprocessor.

MACRO-80 supports two instruction sets: 8080 and Z80. A list of the opcodes with brief summaries of their functions is included as Appendix F. To program with the opcodes of the different languages, the user must first enter the pseudo-op which tells the assembler which language is being coded. Refer to the Language Set Selection Pseudo-ops section of Chapter 4 for details.

MACRO-80 also supports a large variety of pseudo-ops that direct the assembler to perform many different functions. The pseudo-ops are described extensively in Chapter 4 and are summarized in Appendix E.

Opcodes and pseudo-ops are (usually) entered in the OPERATION field of a statement line. (A program statement line usually has an entry in the operation field, unless the line is a Comment line only. The Operation field will be the first field filled if no label is entered.) An Operation may be any 8080 or Z80 mnemonic; or a MACRO-80 macro assembler pseudo-op, macro call, or expression.

The OPERATION field entries are evaluated in the following order:

1. Macro call

2. Opcode/Pseudo-op

3. Expressions

MACRO-80 compares the entry in the OPERATION filed to an internal list of macro names. If the entry is found, the macro is expanded. If the entry is not a macro, MACRO-80 tries to evaluate the entry as an opcode. If the entry is not an opcode, MACRO-80 tries to evaluate the entry as a pseudo-op. If the entry is not a pseudo-op, MACRO-80 evaluates the entry as an expression. If an expression is entered as a statement line without an opcode, pseudo-op, or macro name in front of it, the MACRO-80 macro assembler does not return an error. Rather, the assembler assumes that a define byte pseudo-op belongs in front of the expression and assembles the line.

Because of the order of evaluation, a macro name that is the same as an opcode prevents you from using the opcode again, except as a macro call. For example, if you give a block of macro code the name ADD in your program, you cannot use ADD as an opcode in that program.

3.4 ARGUMENTS: EXPRESSIONS

Arguments for the opcodes and pseudo-ops are usually called expressions because they resemble mathematical expressions, such as 5+4*3. The parts of an expression are called operands (5, 4, and 3 in the mathematical expression) and operators (the + and * are examples). Expressions may contain one operand or more than one. One operand expressions are probably the form most commonly used as arguments. If the expression contains more than one operand, the operands are related to each other by an operator. For example:

```
5+4    6-3    7*2    8/7    9>8
```

and so on. In MACRO-80, operands are numeric values represented by numbers, characters, symbols, or 8080 opcodes. Operators may be arithmetic or logical.

You are probably familiar with the various forms of expressions that can be used as arguments, but you may want to review the details given below for characteristics unique to MACRO-80.

The following sections define the forms of operands and operators MACRO-80 supports.

3.4.1 Operands

Operands may be numbers, characters, symbols, or 8080 opcodes.

Numbers

The default base for numbers is decimal. The base may be changed by the .RADIX pseudo-op. Any base from 2 (binary) to 16 (hexadecimal) may be selected. When the radix is greater than 10, A-F are used for the digits following 9. If the first digit of a number is not numeric, the number must be preceded by a zero.

A number is always evaluated in the current radix unless one of the following special notations is used:

```
nnnnB      Binary
nnnnD      Decimal
nnnnO      Octal
nnnnH      Hexadecimal
X'nnnn'    Hexadecimal
```

Numbers are 16-bit unsigned binary quantities. Overflow of a number beyond two bytes (16 bits -- that is, 65535 decimal) is ignored, and the result is the low order 16 bits.

ASCII Strings

A string is composed of zero or more characters delimited by quotation marks. Either single (') or double (") quotation marks may be used as string delimiters. When a quoted string is entered as an argument, the values of the characters are stored in memory one after the other. For example:

```
        DB      "ABC"
```

stores the ASCII value of A at the first address, B at the second address, and C at the third.

The delimiter quotes may be used as characters if they appear twice for every character occurrence desired. For example, the statement

```
        "I am ""great"" today"
```

stores the string

```
        I am "great" today
```

If no characters are placed between the quotation marks, the string is evaluated as a null string.

Character Constants

Like strings, character constants are composed of zero, one, or two ASCII characters, delimited by quotation marks. Either single or double quotation marks may be used as delimiters. The delimiter quotes may be used as characters if they appear twice for every character occurrence desired.

The differences are:

1. A character constant is only zero, one, or two characters.

2. Quoted characters are a character constant only if the expression has more than one operand. If the characters are entered as the only operand, they are evaluated and stored as a string. For example:

   'A'+1 is a character constant, but

   'A' is a string.

3. The value of a character constant is calculated, and the result is stored with the low-byte in the first address and the high-byte in the second. For example:

3. The value of a character constant is calculated, and the result is stored with the low-byte in the first address and the high-byte in the second. For example:

```
DW      'AB'+0
```

evaluates to 4142H and stores 42 in the first address and 41 in the second.

A character constant comprised of one character has as its value the ASCII value of that character. That is, the high order byte of the value is zero, and the low order byte is the ASCII value of the character. For example, the value of the constant 'A' is 41H.

A character constant comprised of two characters has as its value the ASCII value of the first character in the high order byte and the ASCII value of the second character in the low order byte. For example, the value of the character constant 'AB'+0 is 41H*256+42H+0.

The ASCII decimal and hexadecimal values for characters are listed in Appendix C.

Symbols in Expressions

A symbol may be used as an operand in an expression. The symbol is evaluated, and the value is substituted for the symbol. The Operation is performed using the symbol's value.

The benefit of using symbols as operands is that the programmer need not remember the exact value each time it is needed; rather, the symbol name can be used. The name is usually easier to remember, especially if the symbol name is made mnemonic. The use of symbols as operands becomes more attractive, of course, as the number of symbols in a program increases.

Rules Governing the Use of EXTERNALS in expressions:

1. EXTERNAL symbols may be used in expressions with the following operators only:

   + - * / MOD HIGH LOW

2. If an EXTERNAL symbol is used in an expression, the result of the expression is always external.

MODE Rules affecting SYMBOLS in expressions:

1. In any operation, except AND, OR, or XOR, the operands may be any mode.

2. For AND, OR, XOR, SHL, and SHR, both operands must be absolute and internal.

3. When an expression contains an Absolute operand and an operand in another mode, the result of the expression will be in the other (not Absolute) mode.

4. When subtracting two operands in different modes, the result will be in Absolute mode. Otherwise, the result will be in the mode of the operands.

5. When adding a data relative symbol and a code relative symbol, the result will be unknown, and MACRO-80 passes the expression to LINK-80 as an unknown, which LINK-80 resolves.

Current Program Counter Symbol

One additional symbol for the Argument field only must be noted: the current program counter symbol. The current program counter is the address of the next instruction to be assembled. The current program counter is often a convenient reference point for calculating new addresses. Instead of remembering or calculating the current program address, the programmer uses a symbol that tells the assembler to use the value of the current program address.

The current program counter symbol is $.

8080 Opcodes as Operands

8080 opcodes are valid one-byte operands _in 8080 mode only._ During assembly, the opcode is evaluated to its hexadecimal value.

To use 8080 opcodes as operands, first set the .8080 pseudo-op. See the Language Set Selection Pseudo-ops section of Chapter 4 for a description of how to use the .8080 pseudo-op.

Only the first byte is a valid operand. Use parentheses to direct the assembler to generate one byte for opcodes that normally generate more than one. For example:

```
MVI     A,(JMP)
ADI     (CPI)
MVI     B,(RNZ)
CPI     (INX H)
ACI     (LXI B)
MVI     C,MOV A,B
```

The assembler returns an error if more than one byte is included in the operand (inside the parentheses) -- such as (CPI 5), (LXI B,LABEL1), or (JMP LABEL2).

Opcodes that generate one byte normally may be used as operands without being enclosed in parentheses.

3.4.2 Operators

MACRO-80 allows both arithmetic and logical operators. Operators which return true or false conditions return true if the result is any non-zero value and false if the result is zero.

The following arithmetic and logical operators are allowed in expressions.

Operator	Definition
NUL	Returns true if the argument (a parameter) is null. The remainder of the line after NUL is taken as the argument to NUL. The conditional IF NUL <argument> is false if the first character of the argument is anything other than a semicolon or carriage return. Note that IFB and IFNB perform the same functions but are simpler to use. (Refer to the Conditional Assembly Facility section in Chapter 4.)
TYPE	The TYPE operator returns a byte that describes two characteristics of its argument: 1) the mode, and 2) whether it is External or not. The argument to TYPE may be any expression (string, numeric, logical). If the expression is invalid, TYPE returns zero. The byte that is returned is configured as

follows:

The lower two bits are the mode. If the lower two bits are:

0 the mode is Absolute
1 the mode is Program Relative
2 the mode is Data Relative
3 the mode is Common Relative

The high bit (80H) is the External bit. If the high bit is on, the expression contains an External. If the high bit is off, the expression is local (not External).

The Defined bit is 20H. This bit is on if the expression is locally defined, and it is off if the expression is undefined or external. If neither bit is on, the expression is invalid.

TYPE is usually used inside macros, where an argument type may need to be tested to make a decision regarding program flow; for example, when conditional assembly is involved.

EXAMPLE:

```
FOO     MACRO    X
        LOCAL    Z
Z       SET TYPE X
IF      Z...
```

TYPE tests the mode and type of X. Depending on the evaluation of X, the block of code beginning with IF Z... may be assembled or omitted.

LOW Isolates the low order 8 bits of an absolute 16-bit value.

HIGH Isolates the high order 8 bits of an absolute 16-bit value.

* Multiply

/ Divide

MOD Modulo. Divide the left operand by the right operand and return the value of the remainder (modulo).

SHR	Shift Right. SHR is followed by an integer which specifies the number of bit positions the value is to be right shifted.
SHL	Shift Left. SHL is followed by an integer which specifies the number of bit positions the value is to be left shifted.
- (Unary Minus)	Indicates that following value is negative, as in a negative integer.
+	Add
-	Subtract the right operand from the left operand.
EQ	Equal. Returns true if the operands equal each other.
NE	Not Equal. Returns true if the operands are not equal to each other.
LT	Less Than. Returns true if the left operand is less than the right operand.
LE	Less than or Equal. Returns true if the left operand is less than or equal to the right operand.
GT	Greater Than. Returns true if the left operand is greater than the right operand.
GE	Greater than or Equal. Returns true if the left operand is greater than or equal to the right operand.
NOT	Logical NOT. Returns true if left operand is true and right is false or if right is true and left is false. Returns false if both are true or both are false.
AND	Logical AND. Returns true if both operators are true. Returns false if either operator is false or if both are false. Both operands must be absolute values.
OR	Logical OR. Returns true if either operator is true or if both are true. Returns false if both operators are false. Both operands must be absolute values.

XOR Exclusive OR. Returns true if either operator is true and the other is false. Returns false if both operators are true or if both operators are false. Both operands must be absolute values.

The order of precedence for the operators is:

NUL, TYPE

LOW, HIGH

*, /, MOD, SHR, SHL

Unary Minus

+, -

EQ, NE, LT, LE, GT, GE

NOT

AND

OR, XOR

Subexpressions involving operators of higher precedence than an expression are computed first. The order of precedence may be altered by using parentheses around portions of an expression you wish to give higher precedence.

All operators except +, -, *, and / must be separated from their operands by at least one space.

The byte isolation operators (HIGH and LOW) isolate the high- or low-order 8 bits of a 16-bit value.

Contents

CHAPTER 4 Assembler Features

4.1 Single-Function Pseudo-ops 4-1
- Instruction Set Selection 4-2
- Data Definition and Symbol Definition 4-4
- PC Mode 4-13
- File Related 4-20
- Listing 4-27
  - Format Control 4-28
  - General Listing Control 4-31
  - Conditional Listing Control 4-33
  - Macro Expansion Listing Control 4-34
  - CREF Listing Control 4-35

4.2 Macro Facility 4-36
- Macro Definition 4-37
  - Calling a Macro 4-38
- Repeat Pseudo-ops 4-40
- Termination 4-44
- Macro Symbol 4-45
- Special Macro Operators 4-46

4.3 Conditional Assembly Facility 4-48

CHAPTER 4

ASSEMBLER FEATURES

The MACRO-80 macro assembler features three general facilities: single-function pseudo-ops, a macro facility, and a conditional assembly facility.

4.1 SINGLE-FUNCTION PSEUDO-OPS

Single-function pseudo-ops involve only their own statement line and direct the assembler to perform only one function. (Macros and conditionals involve more than one line of code, so they may be thought of as block pseudo-ops.)

The Single-Function Pseudo-ops are divided into five types: Instruction Set Selection, Data Definition and Symbol Definition, PC Mode, File Related, and Listing Control.

INSTRUCTION SET SELECTION

The default instruction set mode is 8080. If the correct instruction set selection pseudo-op is not given, the assembler will return fatal errors for opcodes that are not valid for the current instruction set selection mode. That is, .Z80 assembles Z80 opcodes only; .8080 assembles 8080 opcodes only. Therefore, if you have written any assembly language programs for Z80, you need to insert the .Z80 instruction set pseudo-op at the beginning of the program file.

Note that all the pseudo-ops listed in this chapter will assemble in both instruction set modes.

.Z80

.Z80 takes no arguments. .Z80 directs MACRO-80 to assemble Z80 opcodes.

.8080

.8080 takes no arguments. .8080 directs MACRO-80 to assemble 8080 opcodes. (default)

All opcodes entered following an Instruction Set Selection pseudo-op will be assembled as that type of code until a different Instruction Set Selection pseudo-op is encountered.

If you enter an opcode not belonging to the selected instruction set, MACRO-80 will return an Objectionable Syntax error (letter O).

DATA DEFINITION AND SYMBOL DEFINITION

All of the data definition and symbol definition pseudo-ops are supported in both instruction set modes. (The one notable exception is SET, which is illegal in .Z80 mode. For your information, The following notation has been placed before the pseudo-op syntax to indicate which microprocessor the pseudo-op is usually associated with:

* indicates a Z80 pseudo-op

No asterisk indicates an Intel 8080 pseudo-op

Define Byte

```
   DB <exp>[,<exp>...]
*  DEFB <exp>[,<exp>...]
   DB <string>[<string>...]
*  DEFM <string>[,<string>...]
```

The arguments to DB are either expressions or strings. The arguments to DEFB are expressions. The arguments to DEFM are strings. Strings must be enclosed in quotes, either single or double.

NOTE: DB is used throughout the following explanation to represent all the Define Byte pseudo-ops.

DB is used to store a value (string or numeric) in a memory location, beginning with the current location counter.

Expressions must evaluate to one byte. (If the high byte of the result is 0 or 255, no error is given; otherwise, an A error results.)

Strings of three or more characters may not be used in expressions (i.e., they must be immediately followed by a comma or the end of the line). The characters in a 8080 or Z80 string are stored in the order of appearance, each as a one-byte value with the high order bit set to zero.

EXAMPLE:

```
                         DB      'AB'
                         DB      'AB' AND 0FFH
                         DB      'ABC'
```

assembles as:

```
0000'  41 42             DB      'AB'
0002'  42                DB      'AB' AND 0FFH
0003'  41 42 43          DB      'ABC'
```

Define Character

DC <string>

DC stores the characters in <string> in successive memory locations beginning with the current location counter. As with DB, characters are stored in order of appearance, each as a one-byte value with the high order bit set to zero. However, DC stores the last character of the string with the high order bit set to one. An error will result if the argument to DC is a null string.

EXAMPLE:

```
                FOO:    DC      "ABC"
```

assembles to:

```
0000'  41 42 C3 FOO:    DC      "ABC"
```

Define Space

```
   DS  <exp>[,<val>]
 * DEFS <exp>[,<val>]
```

The define space pseudo-ops reserve an area of memory. The value of <exp> gives the number of bytes to be reserved.

To initialize the reserved space, set <val> to the value desired. If <val> is nul (that is, omitted), the reserved space is left as is (uninitialized); the reserved block of memory is not automatically initialized to zeros. As an alternative to setting <val> to zero, when you want the define space block initialized to zeros, you may use the /M switch at assembly time. See the Switches section in Chapter 5, Running MACRO-80, for a description of the /M switch.

All names used in <exp> must be previously defined (i.e., all names known at that point on pass 1). Otherwise, a V error is generated during pass 1, and a U error may be generated during pass 2. If a U error is not generated during pass 2, a phase error will probably be generated because the define space pseudo-op generated no code on pass 1.

EXAMPLE:

```
    DS      100H
```

reserves 100H bytes of memory, uninitialized (whatever values were in those bytes before the program was loaded will still be there). Use the /M switch at assembly time to initialized the 100H bytes to zero, if you want. Or, use the following statement to initialize a reserved space to zero or any other value:

```
    DS      100H,2
```

reserves 100H bytes, each initialized to a value of 2.

Define Word

```
    DW <exp>[,<exp>...]
 *  DEFW <exp>[,<exp>...]
```

The define word pseudo-ops store the values of the expressions in successive memory locations beginning with the current location counter. Expressions are evaluated as 2-byte (word) values. Values are stored low-order byte first, then high-order byte.

Contrast with DDB.

EXAMPLE:

```
                         FOO:    DW          1234H
```

assembles as:

```
0000'  1234      FOO:      DW     1234H
```

Note: The bytes are shown on the listing in the order entered, not the order stored.

Equate

<name> EQU <exp>

EQU assigns the value of <exp> to <name>. The <name> may be a label, a symbol, or a variable, and may be used subsequently in expressions. <name> may not be followed by colon(s).

If <exp> is External, an error is generated. If <name> already has a value other than <exp>, an M error is generated.

If you will want to redefine <name> later in the program, use the SET or ASET pseudo-op to define <name> instead of EQU.

Contrast with SET.

EXAMPLE:

```
    BUF   EQU   0F3H
```

External Symbol

```
    EXT <name>[,<name>...]
    EXTRN <name>[,<name>...]
*   EXTERNAL <name>[,<name>...]
    BYTE EXT <symbol>
    BYTE EXTRN <symbol>
    BYTE EXTERNAL <symbol>
```

The External symbol pseudo-ops **declare** that the name(s) in the list are External (i.e., defined in a different module). If any item in the list refers to a name that is defined in the current program, an M error results. A reference to a name where the name is followed immediately be two pound signs (e.g., NAME##) also declares the name as External.

Externals may evaluate to either one or two bytes. For all External symbol names, only the first 6 characters are passed to the linker. Additional characters are truncated internally.

EXAMPLE:

```
    EXTRN      ITRAN      ;tranf init rtn
```

MACRO-80 will generate no code for this statement when this module is assembled. When ITRAN is used as an argument to a CALL statement, the CALL ITRAN statement generates the code for CALL but a zero value (0000*) for ITRAN. At link time, LINK-80 will search all modules loaded for a PUBLIC ITRAN statement and use the definition of ITRAN found in that module to define ITRAN in the CALL ITRAN statement.

Public Symbol

ENTRY <name>[,<name>...]
GLOBAL <name>[,<name>...]
PUBLIC <name>[,<name>...]

The Public symbol pseudo-ops declare each name in the list as internal and therefore available for use by this program and other programs to be loaded concurrently and linked with LINK-80. All of the names in the list must be defined in the current program, or a U error results. An M error is generated if the name is an External name or common block name.

Only the first 6 characters of a Public symbol name are passed to the linker. Additional characters are truncated internally.

EXAMPLE:

```
            PUBLIC  ITRAN  ;tranf init rtn
              .
              .
              .
    ITRAN:  LD      HL,PASSA  ;store addr of
                              ;reg pass area
```

MACRO-80 assembles the LD statement as usuall but generates no code for the PUBLIC ITRAN statement. When LINK-80 sees EXTRN ITRAN in another module, it knows to search until it sees this PUBLIC ITRAN statement. Then, LINK-80 links the value of ITRAN: LD HL,PASSA statement to the CALL ITRAN statement in the other module(s).

Set

```
   <name> SET <exp>      (Not in .Z80 mode)
 * <name> DEFL <exp>
   <name> ASET <exp>
```

The Set pseudo-ops assign the value of <exp> to <name>. The <name> may be a label, a symbol, or a variable, and may be used subsequently in expressions. <name> may not be followed by colon(s). If <exp> is External, an error is generated.

The SET pseudo-op may not be used in .Z80 mode because SET is a Z80 opcode. Both ASET and DEFL may be used in both instruction set modes.

Use one of the SET pseudo-ops instead of EQU to define and redefine <name>s you may want to redefine later. <name> may be redefined with any of the Set pseudo-ops, regardless of which pseudo-op was used to define <name> originally (the prohibition against SET in .Z80 mode still applies, however).

Contrast with EQU.

EXAMPLE:

```
    F00 ASET BAZ+1000H
```

Whenever FOO is used as an expression (operand), the ALDS assembler will evaluate BAZ+1000H and substitute the value for FOO. Later, if you want FOO to represent a different value, simply reenter the FOO ASET statement with a different expression.

```
    F00 ASET BAZ+1000H
           .
           .
           .
    F00 ASET 3000H
           .
           .
           .
    F00 DEFL 6CDEH
```

PC MODE

Many of the pseudo-ops operate on or from the current location counter, also known as the program counter or PC. The current PC is the address of the next byte to be generated.

In MACRO-80, the PC has a mode, which gives symbols and expressions their modes. (Refer again to the Overview in Chapter 1 and the Symbols section in Chapter 3, if necessary.) Each mode is given a segment of memory by LINK-80 for the instructions assembled to each mode.

The four modes are Absolute, Data Relative, Code Relative, and COMMON Relative.

If the PC mode is absolute, the PC is an absolute address. If the PC mode is relative, the PC is a relative address and may be considered an offset from the absolute address where the beginning of that relative segment will be loaded by LINK-80.

The PC mode pseudo-ops are used to specify in which PC mode a segment of a program will be assembled.

Absolute Segment

ASEG

ASEG never has operands. ASEG generates non-relocatable code.

ASEG sets the location counter to an absolute segment (actual address) of memory. The ASEG will default to 0, which could cause the module to write over part of the operating system. We recommend that each ASEG be followed with an ORG statement set at 103H or higher.

Code Segment

CSEG

CSEG never has an operand. Code assembled in Code Relative mode can be loaded into ROM/PROM.

CSEG resets the location counter to the code relative segment of memory. The location will be that of the last CSEG (default is 0), unless an ORG is done after the CSEG to change the location.

Note, however, that the ORG statement does not set a hard (absolute) address under CSEG mode. An ORG statement under CSEG causes the assembler to add the number of bytes specified by the <exp> argument in the ORG statement to the last CSEG address loaded. If, for example, ORG 50 is given, MACRO-80 will add 50 bytes to the current CSEG location then begin loading the CSEG. The clearing effect of the ORG statement following CSEG (and DSEG as well) can be used to give the module an offset. The rationale for not allowing ORG to set an absolute address for CSEG is to keep the CSEG relocatable.

To set an absolute address for the CSEG, use the /P switch in LINK-80.

CSEG is the default mode of the assembler. Assembly begins with a CSEG automatically executed, and the location counter in the Code Relative mode, pointing to location 0 in the Code Relative segment of memory. All subsequent instructions will be assembled into the Code Relative segment of memory until an ASEG, DSEG, or COMMON pseudo-op is executed. CSEG is then entered to return the assembler to Code Relative mode, at which point the location counter returns to the next free location in the Code Relative segment.

Data Segment

DSEG

The DSEG pseudo-op never has operands. DSEG specifies segments of assembled relocatable code that will later be loaded into RAM only.

DSEG sets the location counter to the Data Relative segment of memory. The location of the data relative counter will be that of the last DSEG (default is 0), unless an ORG is done after the DSEG to change the location.

Note, however, that the ORG statement does not set a hard (absolute) address under DSEG mode. An ORG statement under DSEG causes the assembler to add the number of bytes specified by the <exp> argument in the ORG statement to the last DSEG address loaded. If, for example, ORG 50 is given, MACRO-80 will add 50 bytes to the last DSEG address loaded then begin loading the DSEG. The clearing effect of the ORG statement following DSEG (and CSEG as well) can be used to give the module an offset. The rationale for not allowing ORG to set an absolute address for DSEG is to keep the DSEG relocatable.

To set an absolute address for the DSEG, use the /D switch in LINK-80.

Common Block

COMMON /<block name>/

The argument to COMMON is the common block name. COMMON creates a common data area for every COMMON block that is named in the program. If <block name> is omitted or consists of spaces, the block is considered to be blank common.

COMMON statements are non-executable, storage allocating statements. .COMMON assigns variables, arrays, and data to a storage area called COMMON storage. This allows various program modules to share the same storage area. Statements entered following the .COMMON statement are assembled to the COMMON area under the <block name>. The length of a COMMON area is the number of bytes required to contain the variables, arrays, and data declared in the COMMON block, which ends when another PC mode pseudo-op is encountered. COMMON blocks of the same name may be different lengths. If the lengths differ, then the program module with the longest COMMON block must be loaded first (that is, must be the first module name given in the LINK-80 command line; see Chapter 6 for the description of LINK-80).

COMMON sets the location counter to the selected common block in memory. The location is always the beginning of the area so that compatibility with the FORTRAN COMMON statement is maintained.

EXAMPLE:

```
        COMMON /DATABIN/
ANVIL   EQU     100H
        DB      0FFH
        DW      1234H
        DCI     'FORGE'
        CSEG
          .
          .
          .
```

Set Origin

ORG <exp>

At any time, the value of a location counter may be changed by use of ORG. Under the ASEG PC mode, the location counter is set to the value of <exp>, and the assembler assigns generated code starting with that value. Under the CSEG, DSEG, and COMMON PC modes, the location counter for the segment is incremented by the value of <exp>, and the assembler assigns generated code starting with the value of that last segment address loaded plus the value of <exp>. All names used in <exp> must be known on pass 1, and the value must either be Absolute or in the same area as the location counter.

EXAMPLE:

```
DSEG
ORG     50
```

sets the Data Relative location counter to 50, relative to the start of the Data Relative segment of memory. This means that the first 50H addresses will be filled with 0. This method provides relocatability. The ORG <exp> statement does not specify a fixed address in CSEG or DSEG mode; rather, LINK-80 loads the segment at a flexible address appropriate to the modules being loaded together.

On the other hand, a program that begins with the statements

```
ASEG
ORG     800H
```

and is assembled entirely in Absolute mode will always load beginning at 800H, unless the ORG statement is changed in the source file. That is, ORG <exp> following ASEG originates the segment at a fixed (i.e., absolute) address specified by <exp>. However, the same program, assembled in Code Relative mode with no ORG statement, may be loaded at any specified address by appending the /P:<address> switch to the LINK-80 command string. (For details, see Section 6.3, Switches.)

Relocate

```
.PHASE <exp>
   .
   .
   .
.DEPHASE
```

.PHASE allows code to be located in one area, but executed only at a different area with a start address specified by <exp>. The <exp> must be an absolute value. .DEPHASE is used to indicate the end of the relocated block of code.

The PC mode within a .PHASE block is absolute, the same as the mode of the <exp> in the .PHASE statement. The code, however, is loaded in the area in effect when the .PHASE statement is encountered. The code within the block is later moved to the address specified by <exp> for execution.

EXAMPLE:

```
                .PHASE      100H
        FOO:    CALL        BAZ
                JMP         ZOO
        BAZ:    RET
                .DEPHASE
        ZOO:    JMP         5
```

assembles as:

```
                        .PHASE      100H
0100    CD 0106 FOO:    CALL        BAZ
0103    C3 0007'        JMP         ZOO
0106    C9      BAZ:    RET
                        .DEPHASE
0007'   C3 0005 ZOO:    JMP         5
                        END
```

.PHASE....DEPHASE blocks are a way to execute a block of code at a specific absolute address.

FILE RELATED

The file related pseudo-ops insert long comments in the program, give the module a name, end the module, or move other files into the current program.

<u>Comment</u>

.COMMENT <delim><text><delim>

The first non-blank character encountered after .COMMENT is taken as the delimiter. The <text> following the delimiter becomes a comment block which continues until the next occurrence of <delimiter>.

Use the .COMMENT pseudo-op to make long comments. It is not necessary to enter the semicolon to indicate a COMMENT. Indeed, the main reason for using .COMMENT is to override the need to begin each comment line with a semicolon. During assembly, .COMMENT blocks are ignored and not assembled.

EXAMPLE:

```
.COMMENT * any amount of text
entered here
          .
          .
          .* ;return to normal assembly
```

End of Program

END [<exp>]

The END statement specifies the end of the module. If the END statement is not included, a %No END statement warning error message results.

The <exp> may be a label, symbol, number, or any other legal argument that LINK-80 can load as the starting point into the first address to be loaded. If <exp> is present, LINK-80 will place an 8080 JMP instruction at 0100H to the address of <exp>. If <exp> is not present, then no start address is passed to LINK-80 for that program, and execution begins at the first module loaded. (Also, if <exp> is not specified, the LINK-80 /G switch will not work for the module.)

The <exp> tells LINK-80 that the program is a main program. Without <exp>, LINK-80 takes assembly language programs as subroutines. If you link only assembly language programs and none contains an END statement with <exp>, LINK-80 will ask for a main program. If you link two or more programs with END <exp> statements, LINK-80 cannot distinguish which should be the main program.

If you want to link two or more main programs, use the /G:Name or /E:Name switches in LINK-80 (see Section 6.2.2, Switches). The "Name" will be the <exp> of the END statement for the program you want to serve as the main program.

If any high-level language program is loaded with assembly language modules, LINK-80 takes the high-level language program as the main program automatically. Therefore, if you want an assembly language module executed before the high-level language program, use the /G:Name or /E:Name switch in LINK-80 to set the assembly language module as the beginning of the program.

As an alternative, we recommend that you place a CALL or INCLUDE statement at the beginning of the high-level language program, and call in the assembly language program for execution prior to execution of the high-level language program.

Include

INCLUDE <filename>
$INCLUDE <filename>
MACLIB <filename>

All three pseudo-ops are synonomous.

These Include pseudo-ops insert source code from an alternate assembly language source file into the current source file during assembly. Use of an Include pseudo-op eliminates the need to repeat an often-used sequence of statements in the current source file.

The <filename> is any valid file specification for the operating system. If the filename extension and/or device designation are other than the default, source filename specifications must include them. The default filename extension for source files is .MAC. The default device designation is the currently logged drive or device.

The included file is opened and assembled into the current source file immediately following the Include pseudo-op statement. When end-of-file is reached, assembly resumes with the next statement following Include pseudo-op.

Nested Includes are not allowed. If encountered, they will result in an objectionable syntax error, O.

The file specified in the operand field must exist. If the file is not found, the error V (value error) is returned, and the Include is ignored. The V error is also returned if the Include filename extension is not .MAC.

On a MACRO-80 listing, the letter C is printed between the assembled code and the source line on each line assembled from an included file. See the Listing Control Pseudo-op section below for a description of listing file formats.

Name Module

NAME ('modname')

Name defines a name for the module. The parentheses and quotation marks around modname are required. Only the first six characters are significant in a module name.

A module name may also be defined with the TITLE pseudo-op. In the absence of both the NAME and TITLE pseudo-ops, the module name is created from the source filename.

Radix

.RADIX <exp>

The <exp> in a .RADIX statement is always a decimal numeric constant, regardless of the current radix.

The default input radix (or base) for all constants is decimal. The .RADIX pseudo-op allows you to change the input radix to any base in the range 2 to 16.

.RADIX does not change the radix of the listing; rather, it allows you to input numeric values in the radix you choose without special notation. (Values in other radixes still require the special notations described in Section 3.4.1.) Values in the generated code remain in hexadecimal radix.

EXAMPLE:

```
DEC:    DB      20
        .RADIX  2
BIN:    DB      00011110
        .RADIX  16
HEX:    DB      0CF
        .RADIX  8
OCT:    DB      73
        .RADIX  10
DECI:   DB      16
HEXA:   DB      0CH
```

assembles as:

```
0000'  14     DEC:    DB      20
0002                  .RADIX  2
0001'  1E     BIN:    DB      00011110
0010                  .RADIX  16
0002'  CF     HEX:    DB      0CF
0008                  .RADIX  8
0003'  3B     OCT:    DB      73
000A                  .RADIX  10
0004'  10     DECI:   DB      16
0005'  0C     HEXA:   DB      0CH
```

Request

.REQUEST <filename>[,<filename>...]

When you run LINK-80, .REQUEST sends a request to the LINK-80 linking loader to search the filenames in the list for undefined external symbols. If LINK-80 finds any undefined external symbols (external symbols for which a corresponding PUBLIC symbol is not currently loaded), you will know that you need to load one or more additional modules to complete linking.

The filenames in the list should be in the form of legal symbols. <filename> should not include a filename extension or device designation. LINK-80 assumes the default extension (.REL) and the currently logged disk drive.

EXAMPLE:

```
        .
        .
        .
    .REQUEST   SUBR1
        .
        .
        .
```

LINK-80 will search SUBR1 for external symbols which do not have coresonding PUBLIC symbol definitions declared among the currently loaded modules.

LISTING

Listing pseudo-ops perform two general functions: format control and listing control. Format control pseudo-ops allow the programmer to insert page breaks and direct page headings. Listing control pseudo-ops turn on and off the listing of all or part of the assembled file.

Format Control

These pseudo-ops allow you to direct page breaks, titles, and subtitles on your program listings.

Form Feed

```
*   *EJECT [<exp>]
    PAGE <exp>
    $EJECT
```

The form feed pseudo-ops cause the assembler to start a new output page. The assembler puts a form feed character in the listing file at the end of the page.

The value of <exp>, if included, becomes the new page size (measured in lines per page) and must be in the range 10 to 255. The default page size is 50 lines per page.

EXAMPLE:

```
        .
        .
        .
    *EJECT 58
        .
        .
        .
```

The assembler causes the printer to start a new page every time 58 lines of program have been printed.

Title

TITLE <text>

TITLE specifies a title to be listed on the first line of each page. If more than one TITLE is given, a Q error results. The first six characters of the title are used as the module name, unless a NAME pseudo-op is used. (If neither a TITLE nor a NAME pseudo-op is used, the module name is created from the source filename.)

EXAMPLE:

```
    TITLE PROG1
        .
        .
        .
```

The module name is now PROG1. The module may be called by this name, which will be printed at the top of every listing page.

Subtitle

SUBTTL <text>
$TITLE ('<text>')

SUBTTL specifies a subtitle to be listed in each page heading on the line after the title. The <text> is truncated after 60 characters.

Any number of SUBTTLs may be given in a program. Each time the assembler encounters SUBTTL, it replaces the <text> from the previous SUBTTL with the <text> from the most recently encountered SUBTTL. To turn off SUBTTL for part of the output, enter a SUBTTL with a null string for <text>.

EXAMPLE:

```
    SUBTTL SPECIAL I/O ROUTINE
        .
        .
        .
    SUBTTL
        .
        .
        .
```

The first SUBTTL causes the subtitle SPECIAL I/O ROUTINE to be printed at the top of every page. The second SUBTTL turns off subtitle (the subtitle line on the listing is left blank).

General Listing Control

.LIST - List all lines with their code
.XLIST - Suppress all listing

.LIST is the default condition. If you specify a listing file in the command line, the file will be listed.

When .XLIST is encountered in the source file, source and object code will not be listed. .XLIST remains in effect until a .LIST is encountered.

.XLIST overrides all other listing control pseudo-ops. So, nothing will be listed, even if another listing pseudo-op (other than .LIST) is encountered.

EXAMPLE:

```
    .
    .
    .
.XLIST    ;listing suspended here
    .
    .
    .
.LIST    ;listing resumes here
```

Print At Terminal

.PRINTX <delim><text><delim>

The first non-blank character encountered after .PRINTX is the delimiter. The following text is listed on the terminal during assembly until another occurrence of the delimiter is encountered. .PRINTX is useful for displaying progress through a long assembly or for displaying the value of conditional assembly switches.

.PRINTX will output on both passes. If only one printout is desired, use the IF1 or IF2 pseudo-op, depending on which pass you want displayed. See the Conditional pseudo-ops for IF1 and IF2.

EXAMPLE:

```
    .PRINTX *Assembly half done*
```

The assembler will send this message to the terminal screen when encountered.

```
IF1
.PRINTX *Pass 1 done*    ;pass 1 message only
ENDIF

IF2
.PRINTX *Pass 2 done*    ;pass 2 message only
ENDIF
```

Conditional Listing Control

The three conditional listing control pseudo-ops are used to specify whether or not you wish statements contained within a false conditional block to appear on the listing. See also the description of the /X switch in Chapter 5.

Suppress False Conditionals

.SFCOND

.SFCOND suppresses the portion of the listing that contains conditional expressions that evaluate as false.

List False Conditionals

.LFCOND

.LFCOND assures the listing of conditional expressions that evaluate false.

Toggle False Listing Conditional

.TFCOND

.TFCOND toggles the current setting. .TFCOND operates independently from .LFCOND and .SFCOND. .TFCOND toggles the default setting, which is set by the presence or absence of the /X switch in the assembler command line. When /X is present, .TFCOND will cause false conditionals to list. When /X is not given, .TFCOND will suppress false conditionals.

Macro Expansion Listing Control

Expansion listing pseudo-ops control the listing of lines inside macro and repeat pseudo-op (REPT, IRP, IRPC) blocks, and may be used only inside a macro or repeat block.

Exclude Non-code Macro Lines

.XALL

.XALL is the default.

.XALL lists source code and object code produced by a macro, but source lines which do not generate code are not listed.

List Macro Text

.LALL

.LALL lists the complete macro text for all expansions, including lines that do not generate code.

Suppress Macro Listing

.SALL

.SALL suppresses listing of all text and object code produced by macros.

CREF Listing Control Pseudo-ops

You may want the option of generating a cross reference listing for part of a program but not all of it. To control the listing or suppressing of cross references, use the cross reference listing control pseudo-ops, .CREF and .XCREF, in the source file for MACRO-80. These two pseudo-ops may be entered at any point in the program in the OPERATOR field. Like the other listing control pseudo-ops, .CREF and .XCREF support no ARGUMENTs.

Suppress Cross References

.XCREF

.XCREF turns of the .CREF (default) pseudo-op. .XCREF remains in effect until MACRO-80 encounters .CREF. Use .XCREF to suppress the creation of cross references in selected portions of the file. Because neither .CREF nor .XCREF takes effect until the /C switch is set in the MACRO-80 command line, there is no need to use .XCREF if you want the usual List file (one without cross references); simply omit /C from the ALDS assembler command line.

List Cross References

.CREF

.CREF is the default condition. Use .CREF to restart the creation of a cross reference file after using the .XCREF pseudo-op. .CREF remains in effect until MACRO-80 encounters .XCREF. Note, however, that .CREF has no effect until the /C switch is set in the MACRO-80 command line.

4.2 MACRO FACILITY

The macro facility allows you to write blocks of code which can be repeated without recoding. The blocks of code begin with either the macro definition pseudo-op or one of the repetition pseudo-ops and end with the ENDM pseudo-op. All of the macro pseudo-ops may be used inside a macro block. In fact, nesting of macros is limited only by memory.

The macro facility of the MACRO-80 macro assembler includes pseudo-ops for:

- macro definition:
  - MACRO
- repetitions:
  - REPT (repeat)
  - IRP (indefinite repeat)
  - IRPC (indefinite repeat character)
- termination:
  - ENDM
  - EXITM
- unique symbols within macro blocks:
  - LOCAL

The macro facility also supports some special macro operators:

- &
- ;;
- !
- %

Macro Definition

```
<name> MACRO <dummy>[,<dummy>...]
          .
          .
          .
        ENDM
```

The block of statements from the MACRO statement line to the ENDM statement line comprises the body of the macro, or the macro's definition.

<name> is like a LABEL and conforms to the rules for forming symbols. Note that <name> may be any length, but only the first 16 characters are passed to the linker. After the macro has been defined, <name> is used to invoke the macro.

A <dummy> is a place holder that is replaced by a parameter in a one-for-one text substitution when the MACRO block is used. Each <dummy> may be up to 32 characters long. The number of dummys is limited only by the length of a line. If you specify more than one dummy, they must be separated by commas. MACRO-80 interprets all characters between commas as a single dummy.

NOTE

A dummy is always recognized exclusively as a dummy. Even if a register name (such as A or B) is used as a dummy, it will be replaced by a parameter during expansion.

A macro block is not assembled when it is encountered. Rather, when you call a macro, the assembler "expands" the macro call statement by bringing in and assembling the appropriate macro block.

If you want to use the TITLE, SUBTTL, or NAME pseudo-ops for the portion of your program where a macro block appears, you should be careful about the form of the statement. If, for example, you enter SUBTTL MACRO DEFINITIONS, MACRO-80 will assemble the statement as a macro definition with SUBTTL as the macro name and DEFINITIONS as the dummy. To avoid this problem, alter the word MACRO in some way; e.g., - MACRO, MACROS, and so on.

Calling a Macro

To use a macro, enter a macro call statement:

```
<name> <parameter>[,<parameter>...]
```

<name> is the <name> of the MACRO block. A <parameter> replaces a <dummy> on a one-for-one basis. The number of parameters is limited only by the length of a line. If you enter more than one parameter, they must be separated by commas. If you place angle brackets around parameters separated by commas, the assembler will pass all the items inside the angle brackets as a single parameter. For example:

```
FOO 1,2,3,4,5
```

passes five parameters to the macro, but:

```
FOO <1,2,3,4,5>
```

passes only one.

The number of parameters in the macro call statement need not be the same as the number of dummys in the MACRO definition. If there are more parameters than dummys, the extras are ignored. If there are fewer, the extra dummys will be made null. The assembled code will include the macro block after each macro call statement.

EXAMPLE:

```
EXCHNG  MACRO   X,Y
        PUSH    X
        PUSH    Y
        POP     X
        POP     Y
        ENDM
```

If you then enter as part of a program some code and a macro call statement:

```
LDA     2FH
MOV     HL,A
LDA     3FH
MOV     DE,A
EXCHNG  HL,DE
```

assembly generates the code:

```
0000'   3A 002F                LDA      2FH
0003'   67                     MOV      HL,A
0004'   3A 003F                LDA      3FH
0007'   57                     MOV      DE,A
                               EXCHNG   HL,DE
0008'   E5          +          PUSH     HL
0009'   D5          +          PUSH     DE
000A'   E1          +          POP      HL
000B'   D1          +          POP      DE
```

Repeat Pseudo-ops

The pseudo-ops in this group allow the operations in a block of code to be repeated for the number of times you specify. The major differences between the Repeat pseudo-ops and MACRO pseudo-op are:

1. MACRO gives the block a name by which to call in the code wherever and whenever needed; the macro block can be used in many different programs by simply entering a macro call statement.

2. MACRO allows parameters to be passed to the MACRO block when a MACRO is called; hence, parameters can be changed.

Repeat pseudo-op parameters must be assigned as a part of the code block. If the parameters are known in advance and will not change, and if the repetition is to be performed for every program execution, then Repeat pseudo-ops are convenient. With the MACRO pseudo-op, you must call in the MACRO each time it is needed.

Note that each Repeat pseudo-op must be matched with the ENDM pseudo-op to terminate the repeat block.

Repeat

```
REPT <exp>
   .
   .
   .
ENDM
```

Repeat block of statements between REPT and ENDM <exp> times. <exp> is evaluated as a 16-bit unsigned number. If <exp> contains an External symbol or undefined operands, an error is generated.

EXAMPLE:

```
          X    SET   0
               REPT  10     ;generates DB 1 - DB 10
          X    SET   X+1
               DB    X
               ENDM
```

assembles as:

```
0000           X    SET   0
                    REPT  10     ;generates DB 1 - DB 10
               X    SET   X+1
                    DB    X
                    ENDM
0000'   01     +    DB    X
0001'   02     +    DB    X
0002'   03     +    DB    X
0003'   04     +    DB    X
0004'   05     +    DB    X
0005'   06     +    DB    X
0006'   07     +    DB    X
0007'   08     +    DB    X
0008'   09     +    DB    X
0009'   0A     +    DB    X
                          END
```

Indefinite Repeat

```
IRP <dummy>,<parameters inside angle brackets>
 .
 .
 .
ENDM
```

Parameters must be enclosed in angle brackets. Parameters may be any legal symbol, string, numeric, or character constant. The block of statements is repeated for each parameter. Each repetition substitutes the next parameter for every occurrence of <dummy> in the block. If a parameter is null (i.e., <>), the block is processed once with a null parameter.

EXAMPLE:

```
    IRP     X,<1,2,3,4,5,6,7,8,9,10>
    DB      X
    ENDM
```

This example generates the same bytes (DB 1 - DB 10) as the REPT example.

When IRP is used inside a MACRO definition block, angle brackets around parameters in the macro call statement are removed before the parameters are passed to the macro block. An example, which generates the same code as above, illustrates the removal of one level of brackets from the parameters:

```
    FOO     MACRO     X
            IRP       Y,<X>
            DB        Y
            ENDM
            ENDM
```

When the macro call statement

```
    FOO <1,2,3,4,5,6,7,8,9,10>
```

is assembled, the macro expansion becomes:

```
    IRP     Y,<1,2,3,4,5,6,7,8,9,10>
    DB      Y
    ENDM
```

The angle brackets around the parameters are removed, and all items are passed as a single parameter.

Indefinite Repeat Character

```
IRPC <dummy>,<string>
 .
 .
 .
ENDM
```

The statements in the block are repeated once for each character in the string. Each repetition substitutes the next character in the string for every occurrence of <dummy> in the block.

EXAMPLE:

```
    IRPC    X,0123456789
    DB      X+1
    ENDM
```

This example generates the same code (DB 1 - DB 10) as the two previous examples.

Termination

End Macro

ENDM

ENDM tells the assembler that the MACRO or Repeat block is ended.

Every MACRO, REPT, IRP, and IRPC must be terminated with the ENDM pseudo-op. Otherwise, the 'Unterminated REPT/IRP/IRPC/MACRO' message is generated at the end of each pass. An unmatched ENDM causes an O error.

If you wish to be able to exit from a MACRO or repeat block before expansion is completed, use EXITM.

Exit Macro

EXITM

The EXITM pseudo-op is used inside a MACRO or Repeat block to terminate an expansion when some condition makes the remaining expansion unnecessary or undesirable. Usually EXITM is used in conjunction with a conditional pseudo-op.

When an EXITM is assembled, the expansion is exited immediately. Any remaining expansion or repetition is not generated. If the block containing the EXITM is nested within another block, the outer level continues to be expanded.

EXAMPLE:

```
FOO     MACRO   X
Y       SET     0
        REPT    X
Y       SET     Y+1
        IFE     Y-0FFH  ;test Y
        EXITM   ;if true, exit REPT
        ENDIF
        DB      Y
        ENDM
        ENDM
```

Macro Symbol

LOCAL <dummy>[,<dummy>...]

The LOCAL pseudo-op is allowed only inside a MACRO definition block. When LOCAL is executed, the assembler creates a unique symbol for each <dummy> and substitutes that symbol for each occurrence of the <dummy> in the expansion. These unique symbols are usually used to define a label within a macro, thus eliminating multiple-defined labels on successive expansions of the macro. The symbols created by the assembler range from ..0001 to ..FFFF. Users should avoid the form ..nnnn for their own symbols. A LOCAL statement must precede all other types of statements in the macro definition.

EXAMPLE:

```
FOO     MACRO   NUM,Y
        LOCAL   A,B,C,D,E
A:      DB      7
B:      DB      8
C:      DB      Y
D:      DB      Y+1
E:      DW      NUM+1
        JMP     A
        ENDM
        FOO     0C00H,0BEH
        END
```

generates the following code (notice that MACRO-80 has substituted LABEL names in the form ..nnnn for the instances of the dummy symbols):

```
                        FOO        MACRO   NUM,Y
                                   LOCAL   A,B,C,D,E
                        A:         DB      7
                        B:         DB      8
                        C:         DB      Y
                        D:         DB      Y+1
                        E:         DW      NUM+1
                                   JMP     A
                                   ENDM
                                   FOO     0C00H,0BEH
0000'   07              +..0000:   DB      7
0001'   08              +..0001:   DB      8
0002'   BE              +..0002:   DB      0BEH
0003'   BF              +..0003:   DB      0BEH+1
0004'   0C01            +..0004:   DW      0C00H+1
0006'   C3 0000'        +          JMP     ..0000
                                   END
```

Special Macro Operators

Several special operators can be used in a macro block to select additional assembly functions.

& Ampersand concatenates text or symbols. (The & may not be used in a macro call statement.) A dummy parameter in a quoted string will not be substituted in expansion unless preceded immediately by &. To form a symbol from text and a dummy, put & between them.

For example:

```
ERRGEN      MACRO    X
ERROR&X:    PUSH     B
            MVI      B,'&X'
            JMP      ERROR
            ENDM
```

The call ERRGEN A will then generate:

```
ERRORA:     PUSH    B
            MVI     B,'A'
            JMP     ERROR
```

;; In a block operation, a comment preceded by two semicolons is not saved as a part of the expansion (i.e., it will not appear on the listing even under .LALL). A comment preceded by only one semicolon, however, will be preserved and appear in the expansion.

! An exclamation point may be entered in an argument to indicate that the next character is to be taken literally. Therefore, !; is equivalent to <;>.

% The percent sign is used only in a macro argument to convert the expression that follows it (usually a symbol) to a number in the current radix (set by the .RADIX pseudo-op). During macro expansion, the number derived from converting the expression is substituted for the dummy. Using the % special operator allows a macro call by value. (Usually, a macro call is a call by reference with the text of the macro argument substituting exactly for the dummy.)

The expression following the % must conform to the same rules as expressions for the DS (Define Space) pseudo-op. That is, a valid expression that evaluates to an absolute (non-relocatable) constant is required.

EXAMPLE:

```
PRINTE     MACRO      MSG,N
            .PRINTX   * MSG,N *
           ENDM
SYM1       EQU        100
SYM2       EQU        200
           PRINTE <SYM1 + SYM2 = >,%(SYM1 + SYM2)
```

Normally, the macro call statement would cause the string (SYM1 + SYM2) to be substituted for the dummy N. The result would be:

```
    .PRINTX    * SYM1 + SYM2 = (SYM1 + SYM2)
```

When the % is placed in front of the parameter, the assembler generates:

```
    .PRINTX    * SYM1 + SYM2 = 300 *
```

4.3 CONDITIONAL ASSEMBLY FACILITY

Conditional pseudo-ops allow users to design blocks of code which test for specific conditions then proceed accordingly.

All conditionals follow the format:

```
IFxxxx [argument]     COND [argument]
   .                     .
   .                     .
   .                     .
[ELSE                 [ELSE
   .                     .
   .                     .
   .]                    .]
ENDIF                 ENDC
```

Each IFxxxx must have a matching ENDIF to terminate the conditional. Each COND must have a matching ENDC to terminate the conditional. Otherwise, an 'Unterminated conditional' message is generated at the end of each pass. An ENDIF without a matching IF or an ENDC without a matching COND causes a C error.

The assembler evaluates the conditional statement to TRUE (which equals FFFFH, or -1, or any non-zero value), or to FALSE (which equals 0000H). The code in the conditional block is assembled if the evaluation matches the condition defined in the conditional statement. If the evaluation does not match, the assembler either ignores the conditional block completely or, if the conditional block contains the optional ELSE statement, assembles only the ELSE portion.

Conditionals may be nested up to 255 levels. Any argument to a conditional must be known on pass 1 to avoid V errors and incorrect evaluation. For IF/IFT/COND and IFF/IFE the expression must involve values which were previously defined, and the expression must be Absolute. If the name is defined after an IFDEF or IFNDEF, pass 1 considers the name to be undefined, but it will be defined on pass 2.

Each conditional block may include the optional ELSE pseudo-op, which allows alternate code to be generated when the opposite condition exists. Only one ELSE is permitted for a given IFxxxx/COND. An ELSE is always bound to the most recent, open IF. A conditional with more than one ELSE or an ELSE without a conditional will cause a C error.

Conditional Pseudo-ops

IF <exp>
IFT <exp>
* COND <exp>

If <exp> evaluates to not-0, the statements within the conditional block are assembled.

IFE <exp>
IFF <exp>

If <exp> evaluates to 0, the statements in the conditional block are assembled.

IF1 Pass 1 Conditional

If the assembler is in pass 1, the statements in the conditional block are assembled.

IF2 Pass 2 Conditional

If the assembler is in pass 2, the statements in the conditional block are assembled.

IFDEF <symbol>

If the <symbol> is defined or has been declared External, the statements in the conditional block are assembled.

IFNDEF <symbol>

If the <symbol> is not defined or not declared External, the statements in the conditional block are assembled.

IFB <arg>

The angle brackets around <arg> are required.

If the <arg> is blank (none given) or null (two angle brackets with nothing in between, <>), the statements in the conditional block are assembled.

IFNB <arg>

The angle brackets around <arg> are required.

If <arg> is not blank, the statements in the conditional block are assembled. Used for testing for dummy parameters.

IFIDN <arg1>,<arg2>

The angle brackets around <arg1> and <arg2> are required.

If the string <arg1> is identical to the string <arg2>, the statements in the conditional block are assembled.

IFDIF <arg1>,<arg2>

The angle brackets around <arg1> and <arg2> are required.

If the string <arg1> is different from the string <arg2>, the statements in the conditional block are assembled.

ELSE

The ELSE pseudo-op allows you to generate alternate code when the opposite condition exists. May be used with any of the conditional pseudo-ops.

ENDIF
* ENDC

These pseudo-ops terminate conditional blocks. A terminate pseudo-op must be given for every conditional pseudo-op used. ENDIF must be matched with an IFxxxx pseudo-op. ENDC must be matched with the COND pseudo-op.

Contents

Chapter 5 Running MACRO-80

5.1 Invoking MACRO-80 5-2
5.2 MACRO-80 Command Line 5-2
- Source 5-3
- Object 5-4
- List 5-5
- Switches 5-6
- Additional Command Line Entries 5-9
  - Filename Extensions 5-10
  - Device Designations 5-11
  - Device Designations as Filenames 5-12

5.3 MACRO-80 Listing File Formats 5-13
- File Format 5-13
- Symbol Table Format 5-14

5.4 Error Codes and Messages 5-15

CHAPTER 5

RUNNING MACRO-80

When you have completed creating the assembly language source file, you are ready to assemble it. MACRO-80 assembles the source file statements, including expanding macros and repeat pseudo-ops. The result of assembly will be relocatable object code which is ready to link and load with LINK-80. The relocatable object code can be saved in a disk file, which the assembler gives the filename extension .REL. The assembled (REL) file is not an executable file. The file will be executable only after it is processed through LINK-80.

MACRO-80 resides in approximately 19K of memory and has an assembly rate of over 1000 lines per minute. MACRO-80 runs under the CP/M operating system.

MACRO-80 assembles your source file in two passes. During pass 1, MACRO-80 evaluates the program statements, calculates how much code it will generate, builds a symbol table where all symbols are assigned values, and expands macro call statements. During pass 2, MACRO-80 fills in the symbol and expression values from the symbol table, again expands macro call statements, and emits the relocatable code. MACRO-80 checks the values of symbols, expressions, and macros during both passes. If a value during pass 2 is different from the value during pass 1, MACRO-80 returns a phase error code.

Before MACRO-80 can be run, the diskette which contains MACRO-80 must be inserted in the appropriate disk drive. The diskette on which you created the source file must also be in a disk drive.

5.1 INVOKING MACRO-80

To invoke MACRO-80, enter:

```
M80
```

The program file M80.COM will be loaded. MACRO-80 will display an asterisk (*) to indicate that the assembler is ready to accept a command line.

5.2 MACRO-80 COMMAND LINE

The command line for MACRO-80 consists of four fields, labeled:

```
Object,List=Source/Switch
```

The command line may be entered on its own line, or it may be entered at the same time as the M80 command. (If M80 and the command line are entered on one line, MACRO-80 will not return the asterisk prompt.) Entering the command line on its own line allows single drive configurations to use MACRO-80. In addition, by entering M80 and the command line separately, you are able to perform another assembly without reinvoking MACRO-80. When assembly is finished, MACRO-80 will return the asterisk (*) prompt and wait for another command line. To exit MACRO-80 when you have entered M80 and the command line separately, type <CTRL-C>.

If you are performing only one assembly, entering the command line on the same line as M80 is convenient; it requires less typing and allows the assembly operation to be part of a SUBMIT command. When you enter M80 and the command line together, MACRO-80 exits automatically to the operating system.

NOTE

> If you enter M80 and the command line separately, you must enter the command line in upper case only. If you do not, MACRO-80 will return a ?Command Error message. If you enter M80 and the command line on one line, the entries may be in either upper or lower case (or mixed) because CP/M converts all entries to upper case before passing the entries.

Source (=filename)

To assemble your source program, you must enter at least an equal sign (=) and the source filename.

The =filename indicates which source file you want to assemble. If the source file disk is not in the currently logged drive, you must include the drive designation as part of the filename. If the source filename is entered without an extension, MACRO-80 assumes that the extension is .MAC. If the extension is not .MAC, you must include the extension as part of the filename. For other possibilities for drive/device designations and filename extensions, see the Additional Command Line Entries section, below.)

The Source entry is the only entry required besides M80.

The simplest command is:

```
M80 =Source
```

This command directs MACRO-80 to assemble the source file and save the result in a relocatable object file (called a REL file) with the same name as the source file. If the source file is NEIL.MAC, the command line:

```
M80 =NEIL
```

generates an assembled file named NEIL.REL.

An additional option is to enter only a comma (,) to the left of the equal sign. When MACRO-80 sees a comma as the first entry after the M80 entry, it suppresses all output files (Object and List). The command line

```
M80 ,=NEIL
```

causes MACRO-80 to assemble the file NEIL.MAC, but no output files are created. Programmers use this command line to check syntax in the source program before saving the assembled program. Because no files are generated, the assembly is completed faster and errors are known sooner.

Object (filename)

An Object entry is always optional. However, certain circumstances will compel you to make some entry for the Object.

The Object file saves the assembled program in a disk file. LINK-80 uses the Object file to create an executable program. If both Object and List entries are omitted from a command line (as in =Source), MACRO-80 will generate an Object file with the same filename as the Source, but with the default extension .REL.

If you want your Object file to have a name different from the source file, you must enter a filename in the Object field. MACRO-80 will still append the filename extension .REL, unless you also enter an extension.

Also, if you want both a List file and a REL file generated, you must enter a filename for the Object, even if you want the REL file named after the source file. If you enter a filename for the List but omit the Object, no REL file will be generated. Programmers do use this feature for checking the program for errors before final assembly. The program listing aids debugging.

The name for the Object file may be the same as the source filename or any other legal filename you choose. Since it is practical to have all files which relate to a program carry some mutual indication of their relationship, most often you will want to give your object file the same name as your source file.

List (,filename)

A List entry is always optional. The comma is required in front of all List entries. If you want a List file, enter a ,filename for the List. (There is an alternative to this rule. See the Switches section below for discussion of the /L switch.)

MACRO-80 appends the default extension .PRN to the List file unless you specify a different extension in the List entry.

The command line:

```
M80 ,NEIL=NEIL
```

assembles the file NEIL.MAC (source file) and creates the List file NEIL.PRN. An Object (REL) file is not created.

The name may be the same as the source filename or any other legal filename you choose. Since it is practical to have all files which relate to a program carry some mutual indication of their relationship, most often you will want to give your listing file the same name as your source file.

Avoid entering only a comma for the List after entering a filename for the Object. For example:

```
M80 NEIL,=NEIL
```

MACRO-80 will probably ignore the comma and assemble the source file into a REL file. It is possible that MACRO-80 might return a COMMAND ERROR message.

If you enter only a comma for the List and nothing for the Object, MACRO-80 will assemble the source file, but will generate no output files. This command

```
M80 ,=Source
```

allows you to check the source program for syntax errors before saving the assembled program in a disk file. While MACRO-80 always checks for errors, this command form provides much faster assembly because the output files do not have to be created.

At the end of assembly, MACRO-80 will print the message:

```
[xx][No] Fatal errors [,xx warnings]
```

This message reports the number of fatal errors and warning errors encountered in the program. The message is listed at the end of every assembly on the terminal screen and in the listing file. When the message appears, the assembler has finished. When the message No Fatal Errors appears, the assembly is complete and successful.

Switches (/Switch)

You can command MACRO-80 to perform some additional functions besides assembling and creating object and listing files. These additional commands are given to MACRO-80 as entries at the end of the command line. A Switch entry directs MACRO-80 to "switch on" some additional or alternate function; hence, these entries are called switches. Switches are letters preceded by slash marks (/). Any number of switches may be entered, but each switch must be preceded by a slash. For example:

```
M80 ,=NEIL/L/R
```

The available switches for MACRO-80 are:

Switch	Action
/O	Octal listing. MACRO-80 generates List file addresses in octal radix.
/H	Hexadecimal listing. MACRO-80 generates List file addresses in hexadecimal. This is the default.
/R	Force generation of an Object file with the same name as the source file. May be used instead of giving a filename in the Object field of the command line.

This switch is convenient when you want a REL file but forgot to enter a filename in the Object field and entered a comma and filename or a comma only in the List field. (Remember: if no filenames or comma is entered before the equal sign, a REL file is generated.) Thus, if you enter

```
    M80 ,NEIL=NEIL
 or M80 ,=NEIL
```

then decide, before pressing <ENTER>, that you want a REL file, simply add /R. The command line would then be:

```
    M80 ,NEIL=NEIL/R
 or M80 ,=NEIL/R
```

/L Force generation of a listing file with the same name as the source file. May be used instead of giving a filename in the List field of the command line.

This switch is convenient when you want a List file but forgot to enter a filename in the List field. If you enter the command line:

```
    M80 =NEIL
 or M80 ,=NEIL
 or M80 NEIL=NEIL
```

then decide, before pressing <ENTER>, that you do want a List file, simply add /L. The command would then be:

```
    M80 =NEIL/L
 or M80 ,=NEIL/L
 or M80 NEIL=NEIL/L
```

/C Causes MACRO-80 to generate a special List file (with the same name as the Source file) for use with CREF-80 Cross Reference Facility. If you want to use CREF-80, you must assemble your file with this switch set. See Chapter 8, CREF-80 Cross Reference Facility, for further details.

/Z Directs MACRO-80 to assemble Z80 opcodes. If your source file contains Z80 opcodes and if you do not include the .Z80 pseudo-op in your source file, then you must use the /Z switch at assembly time so that MACRO-80 will assemble the Z80 opcodes.

/I Directs MACRO-80 to assemble 8080 opcodes. If your source file contains 8080 opcodes and if you do not include the .8080 pseudo-op in your source file, then you must use the /I switch at assembly time so that MACRO-80 will assemble the 8080 opcodes. (Default)

/P Each /P allocates an extra 256 bytes of stack space for use during assembly. Use /P if stack overflow errors occur during assembly. Otherwise, /P is not needed.

/M The /M switch initializes Block data areas. If you want the area that is defined by the DS (Define Space) pseudo-op initialized to zeros, then you should use the /M switch in the command line. Otherwise, the space is not guaranteed to contain zeros. That is, DS does not automatically initialize the space to zeros, in which case you may not know what is stored in the DS space or how the program will be affected.

/X The /X switch sets the default and current setting to suppress the listing of false conditionals. Absence of /X in the command line sets the default and current setting to list conditional blocks which evaluate false. /X is often used in conjunction with the conditional listing pseudo-op .TFCOND. Refer to the Listing Pseudo-ops section in Chapter 4 for details.

Additional Command Line Entries

Each command line field supports two additional types of entries--filename extensions and device designations. These two types of entries are actually part of a "file specification." A file specification includes the device where a file is located, the name of the file, and the filename extension.

Usually, filename extensions and device designations are handled by defaults--the MACRO-80 program "inserts" these entries if their positions are left blank in a command line. The default assignments in no way prevent you from entering either filename extensions or device designations, including entries that match the default entries. The programmer may enter or omit these additional entries in any combination.

The format for a file specification under MACRO-80 is:

dev:filename.ext

where: dev: is a 1-3 letter device designation followed by a (required) colon.

filename is a 1-8 letter filename.

.ext is a 1-3 character filename extension preceded by a (required) period.

Filename Extensions (.ext)

To distinguish between Source file, Object file, and List file, MACRO-80 appends an extension to each filename. Filename extensions are three-letter mnemonics appended to the filename with a period (.) between the filename and the extension. The extension which MACRO-80 appends reflects the type of file. Since the extensions are supplied by MACRO-80, they are called default extensions. The default extensions which MACRO-80 supplies are:

.REL	Relocatable object file
.PRN	Listing file
.COM	Absolute (executable object) file

Also, MACRO-80 assumes that, if no filename extension is entered, a source file carries the filename extension .MAC.

You may supply your own extensions, if you find this necessary or desirable. The disadvantage is that whenever you call the file, you must always remember to include your extension. Also, you must remember what type of file it is--relocatable, source, absolute, etc. The advantage of allowing MACRO-80 to assign default extensions is that you always have a mnemonic indication of the type of file, and you can call the filename without appending the extension, in most cases.

Device Designations (dev:)

Each of the fields in a command line (except Invocation) also may include a device designation.

When a device designation is specified in the Source field, the designation tells MACRO-80 where to find the source file. When a device designation is specified in the Object or List fields, the designation tells MACRO-80 where to output the object or list file. If the device designation is omitted from any of these fields, MACRO-80 assumes (defaults to) the currently logged drive. Thus, any time the device designation is the currently logged drive or device, the device designation need not be specified.

It is important to include device designations if several devices or drives will be used during an assembly. For example, if your ALDS diskette is in drive A and your program diskette is in drive B, and you want your REL file output to drive B, you need to give the command line:

```
M80 =B:NEIL
```

When the REL file is output, the currently logged drive is drive B. (However, when MACRO-80 is finished, drive A will be the currently logged drive again.) In contrast, if you saved your source program on the MACRO-80 diskette in drive A and want the REL file output to a diskette in drive B, then you need to enter the command line:

```
M80 B:=A:NEIL
```

As a rule of thumb, if you are not sure if you need to include the device designation (especially the drive designation), enter a designation; it is the one sure way to get the right files in the right places.

The available device designations for MACRO-80 are:

A:, B:, C:,...	Disk drives
LST:	Line Printer
TTY:	Terminal Screen or Keyboard
HSR:	High Speed Reader

Device Designations as Filenames

As an option, you may enter a device designation only in the command line fields in place of a filename. The use of this option gives various results depending on which device is specified and in which field the device is specified. For example:

```
M80 ,TTY:=TTY:
```

allows you to assemble a line immediately on input to check for syntax or other errors. A modification of this command (that is, M80 ,LST:=TTY:), provides the same result but printed on a line printer instead of the terminal screen.

If you use either of these commands (,TTY:=TTY: or ,LST:=TTY:), your first entry should be an END statement. You need to put the assembler into pass 2 before it will emit the code. If you simply start entering statement lines without first entering END, you will receive no response until an END statement is entered. Then you will have to reenter all your statements before you see any code generated.

The following table illustrates the results of the various choices. The table is meant to indicate the possibilities rather than provide an exhaustive list of the combinations.

dev:	Object	,List	=Source
A:, B:, C:, D:	write file to drive specified	write file to drive specified	N/A (a filename must be specified)
HSR:	N/A (input only)	N/A (input only)	reads source program from high-speed reader
LST:	N/A (unreadable file format)	writes listing to line printer	N/A (output only)
TTY:	N/A (unreadable file format)	"writes" listing to screen	"reads" source program from keyboard

Figure 5.1: Effects of Device Designations without Filenames

5.3 MACRO-80 LISTING FILE FORMATS

A listing of a MACRO-80 file displays the two parts of the file in two different formats. One format displays the file lines. The second format displays symbol table listings.

File Format

Each page of a MACRO-80 listing prints header data in the first two lines. If no header data were commanded in the source file (neither the TITLE nor SUBTTL pseudo-op was given), those portions of the header lines are left blank.

The format is:

```
[TITLE text]     M80 z.zz        PAGE x
[SUBTTL text]
```

where: TITLE text is the text supplied with the .TITLE pseudo-op, if .TITLE was included in the source file. If no .TITLE pseudo-op was given in the source file, this space is left blank.

z.zz is the version number of your MACRO-80 program.

x is the page number, which is shown and incremented only when a .PAGE pseudo-op is encountered in the source file, or whenever the current page size has been filled.

SUBTTL text is the text supplied with the .SUBTTL pseudo-op, if .SUBTTL was included in the source file. If no .SUBTTL was given in the source file, this space is left blank.

A blank line follows the header data. The text of the listing file begins on the next line.

The format of a listing line is:

```
[error] ####m xx xxxxm[w]    text
```

where: error represents a one-letter error code. An error code is printed only if the line contains an error. Otherwise, the space is left blank.

represents the location counter. The number is a 4-digit hexadecimal number or a 6-digit octal number. The radix of the location counter number is determined by the use of the /O or /H switch in the MACRO-80 command line Switch field. If no radix switch was given, the default radix is hexadecimal (4-digit).

m represents the PC mode indicator character. The possible symbols are:

'	Code Relative
"	Data Relative
!	COMMON Relative
<space>	Absolute
*	External

xx and xxxx represent the assembled code. xx represents a one-byte value. One-byte values are always followed immediately by a space. xxxx represents a two-byte value, with the high-order byte printed first (this is the opposite of the order in which they are stored). Two-byte values are followed by one of the mode indicators discussed above (indicated by the second m).

[w] represents a line in the MACRO-80 file that came from another file through an INCLUDE pseudo-op; or a line that is part of an expansion (MACRO, REPT, IRP, IRPC). For lines from an INCLUDE statement, a C is printed following the assembled code; for lines in an expansion, a plus sign (+) is printed following the assembled code. Otherwise, this space is blank.

text represents the rest of the line, including labels, operations, arguments, and comments.

Symbol Table Format

The symbol table listing page follows the same header data format as the file line pages. However, instead of a page number, the symbol table page shows PAGE S.

Then, in a symbol table listing, all macro names in a program are listed alphabetically. Next, all symbols are listed, also alphabetically. A tab follows each symbol, then the value of the symbol is printed. Each symbol value is followed by one of the following characters:

I	PUBLIC symbol
U	Undefined symbol
C	COMMON block name. The value shown for the COMMON block name is its length in bytes in hexadecimal or octal radix.
*	External symbol
<space>	Absolute value

' Program relative value

" Data relative value

! COMMON relative value

5.4 ERROR CODES AND MESSAGES

Errors encountered during assembly cause MACRO-80 to return either an error code or an error message. Error codes are one-character flags printed in column one of the listing file. If a listing file is not being printed on the terminal screen, the lines containing errors will nevertheless be printed on the terminal screen. Error messages are printed at the end of the listing file, or, if the listing file is not being displayed on the terminal screen, any error messages will be displayed at the end of the error code lines.

ERROR CODE	MEANING
A	Argument error. The argument to a pseudo-op is not in correct format or is out of range.
C	Conditional nesting error. ELSE without IF, ENDIF without IF, two ELSEs for one IF, ENDC without COND.
D	Double defined symbol. Reference to a symbol which has more than one definition.
E	External error. Use of an External is illegal in the flagged context. For example, FOO SET NAME or LXI B,2-NAME.
M	Multiply defined symbol. The definition is for a symbol that already has a definition.
N	Number error. An error in a number, usually a bad digit. For example, 8Q.

O Bad opcode or objectionable syntax.
ENDM, LOCAL outside a block; SET, EQU, or MACRO without a name; bad syntax in an opcode; or bad syntax in an expression (for example, mismatched parentheses, quotes, consecutive operators).

P Phase error.
The value of a label or EQU name is different during pass 2 from its value during pass 1.

Q Questionable.
Usually, a line is not terminated properly. For example, MOV AX,BX,. This is a warning error.

R Relocation.
Illegal use of relocation in an expression, such as abs-rel. Data, code, and COMMON areas are relocatable.

U Undefined symbol.
A symbol referenced in an expression is not defined. For some pseudo-ops, a V error is printed for pass 1 then a U error for pass 2. Compare with V error code definition below.

V Value error.
On pass 1 a pseudo-op which must have its value known on pass 1 (for example, .RADIX, .PAGE, DS, IF, IFE) has a value which is undefined. If the symbol is defined later in the program, a U error will not appear on the pass 2 listing.

ERROR MESSAGES

%No END statement

No END statement: either it is missing or it is not parsed because it is in a false conditional, unterminated IRP/IRPC/REPT block, or terminated macro.

Unterminated conditional

At least one conditional is unterminated at the end of the file.

Unterminated REPT/IRP/IRPC/MACRO

At least one block is unterminated.

Symbol table full

As MACRO-80 was building the symbol table, the memory available was exhausted. The most usual cause is a large number of macro blocks which also contain statements for many of the statement lines. Macro blocks are stored in the symbol table verbatim, including the comments appended to the lines inside the macro block. You should check all macro blocks in the source program. To exclude comments inside macro blocks from the symbol table, precede these comments by double semicolons (;;). This method should free enough space to assemble your program.

[xx][No] Fatal errors [,xx warnings]

The number of fatal errors and warning errors encountered in the program. The message is listed at the end of every assembly on the terminal screen and in the listing file. When the message appears, the assembler has finished. When the message No Fatal Errors appears, the assembly is complete and successful.

Contents

CHAPTER 6 LINK-80 Linking Loader

6.1 Invoking LINK-80 6-1
6.2 LINK-80 Commands 6-2
6.2.1 Filenames 6-3
6.2.2 Switches 6-4
Execute 6-6
Exit 6-8
Save 6-9
Address Setting 6-11
Library Search 6-15
Global Listing 6-16
Radix Setting 6-17
Special Code 6-18
6.3 Error Messages 6-19

CHAPTER 6

LINK-80 LINKING LOADER

The .REL files which MACRO-80 creates are not executable. To make a REL file executable, you need to load and link the REL file with the LINK-80 linking loader. The result is an executable object file.

Loading means physically placing the file in memory and assigning absolute addresses to the code and data in place of the relative addresses assigned by the assembler. This is one of the required steps for converting a relocatable (REL) file into an executable (COM) file.

Linking means that each loaded file (or module) that directs program flow outside itself (by a CALL, an EXTERNAL symbol, or an Include) will be "linked" to the module that contains the corresponding code.

LINK-80 can also save the assembled-and-linked program as an executable object program on disk in a file with the extension .COM. Consequently, any time you wish to run your program, you need only insert the disk which contains your COM file into an appropriate disk drive and "call" your program -- a simple process of typing in the filename you used to save the program, followed by a carriage return.

6.1 INVOKING LINK-80

To invoke LINK-80, enter:

```
L80
```

The program file L80.COM will be loaded. LINK-80 will display an asterisk (*) to indicate that the linking loader is ready to accept a command. The REL file(s) you want link-loaded must be available in a disk drive. If you have only one drive, you will need to swap diskettes in the drive at each step of the link-loading process.

6.2 LINK-80 COMMANDS

LINK-80 commands are filenames and switches.

You can enter your commands to LINK-80 one at a time; or, you can enter all of your commands (including L80) on one line.

A command line has a flexible format, allowing you a number of options for loading and linking files and for performing other operations. The basic rule for LINK-80 commands is that files are loaded in the order they are named, beginning at the (default) address 103H under CP/M. Even though the files will be loaded in the order entered, you do not have to enter the files in the order of execution. LINK-80 places a jump instruction at address 100H-102H which jumps to the start address of the first instruction to be executed, regardless of its location in memory.

LINK-80 can perform about eleven different tasks. Even though you could use them all, you will rarely use more than three or four at a time.

When you enter a command to LINK-80, LINK-80 returns an asterisk (*) prompt that tells you to enter another command. For example:

```
A>L80<RETURN>
*/switch<RETURN>
*filename<RETURN>
*/switch<RETURN>
*filename/switch<RETURN>
*/E<RETURN>       (to exit LINK-80)
```

Note that all of the above lines may be entered as one line. For example:

```
L80 /switch,filename/switch,filename/switch/E<RETURN>
```

This shows further the flexibility of the LINK-80 command line.

Although entering each command on a separate line is slow and tedious, the advantage is, especially if you are new to a linking loader, that you know at all times what function LINK-80 is performing.

6.2.1 Filenames

Files processed by LINK-80 are REL files. A filename commands LINK-80 to load the named file (also called a module). If any file has been loaded already, a filename also commands LINK-80 to link the loaded files as required.

Normally each linking session requires at least two filenames. One filename directs LINK-80 which REL file to load and link; the other commands LINK-80 to save the executable code in a file with the specified name.

If you enter only one filename during the link session, either the COM file will not be saved (in which case you may have wasted your time), or LINK-80 will return the error message

    ?NOTHING LOADED

Note, however, that if you enter only one filename followed by the /G switch, the COM file will not be saved, but the program will execute as soon as LINK-80 is finished loading and linking. (Refer to the description of the switches in the next section.)

You may enter as many filenames as will fit on one line. The files named may be REL files in different languages (BASIC, COBOL, FORTRAN, or assembly) or runtime library REL files for any of the high-level programming languages. (For exact procedures for high-level language REL files, see the product manual included with the high-level language compiler.)

When LINK-80 is finished, the results are saved in the file named by the programmer in the command line (the filename followed by a /N -- see below, Section 6.2.2, Switches). LINK-80 gives this filename the extension .COM.

A filename command in LINK-80 actually means a file specification. A file specification includes a device designation, a filename, and a filename extension. The format of a file specification is:

    dev:filename.ext

LINK-80 defaults the dev: to the default or currently logged disk drive. LINK-80 defaults the input filename extension to .REL and the output filename extension to .COM. You can alter the device designation to any applicable output device supported by MACRO-80 and/or the filename extension to any three characters by specifying a device or a filename extension when you enter a filename command.

6.2.2 Switches

Switches command LINK-80 to perform functions besides loading and linking. Switches are letters preceded by slash marks (/). You can place as many switches as you need in a single command line, but each switch letter must be preceded by a slash mark (/). For example, if you want to link and load a program named NEIL, save an image of it on diskette, then execute the program, you need two filenames and two switches, so you would enter the commands:

```
NEIL,NEIL/N/G<RETURN>
```

LINK-80 saves a memory image on diskette (the /N switch), then runs the NEIL program (the /G switch).

Some switches can be entered by themselves (/E, /G, /R, /P, /D, /U, /M, /O, /H). Some switches must be appended to the filename they affect (/N, /S). Some switches work only if other switches are also entered during the LINK-80 session (/X, /Y). Some switches must precede any filenames you want affected (/P, /D). Some switches command actions that are deferred until the end of the LINK-80 session (/N, /X, /Y). Some switches command actions that take place when entered (/S, /R -- a filename entered without a switch appended acts this way, too). These "rules of behavior" should be kept in mind when entering LINK-80 commands. See the descriptions for each switch for full details of its action.

The chart below summarizes the switches by function. Full descriptions of the switches by function follow the chart.

<u>BE CAREFUL:</u> Do not confuse the LINK-80 switches with the MACRO-80 switches.

FUNCTION	SWITCH	ACTION
Execute	/G /G:Name	Execute .COM file then exit to operating system. Set .COM file start address equal to value of Name, execute .COM file, then exit to operating system.
Exit	/E /E:Name	Exit to operating system. Set .COM file start address equal to value of Name, then exit to operating system.
Save	/N /N:P	Save all previously loaded programs and subroutines using filename immediately preceding /N. Alternate form of /N; save only program area.
Address Setting	/P /D /R	Set start address for programs and data. If used with /D, /P sets only the program start. Set start address for data area only. Reset LINK-80.
Library Search	/S	Search the library named immediately preceding /S.
Global Listing	/U /M	List undefined globals. List complete global reference map.
Radix Setting	/O /H	Octal radix. Hexadecimal radix (default).
Special Code	/X /Y	Save "COM" file in Intel ASCII Hex format. Requires /N switch. Gives "COM" file the extension .HEX. Creates a special file for use with SID/ZSID debugger. Requires /N and /E switches. Gives special file the extension .SYM.

Figure 6.1: Table of LINK-80 Switches

At least two switches will probably be used in every linking session. These switches belong to the first three functions -- Execute, Exit, and Save.

EXECUTE

Switch	Action
/G	The /G switch causes LINK-80 to load the filename(s) entered in the command line, to link the program(s) together, then to execute the link-loaded program. After the program run, your computer returns to operating system command level. For example,

```
L80 NEIL,NEIL/N/G
```

links NEIL.REL, saves the result in a disk file named NEIL.COM, then exits to the operating system.

Execution takes place as soon as the command line has been interpreted. Just before execution begins, LINK-80 prints three numbers and a BEGIN EXECUTION message. These three numbers can be very useful to you in developing future assembly language programs. The first number is the start address of the program. The second number is the address of the next available byte; that is, the end address plus one byte. The third number is the number of 256-byte pages taken up by the program (the difference between the start address and the end address converted to 256-byte pages).

If you do not want to save the .COM file, use the /G switch and enter only one filename on the command line. For example:

```
L80 NEIL/G
```

But Remember: No COM file is created (since you did not include /N). To run the program again, you will have to run LINK-80 again.

/G:<name> The /G:<name> switch performs exactly like the plain /G switch but with one additional feature. <name> is a global symbol which was defined previously in one of the modules which is being linked and loaded. When LINK-80 sees <name>, it uses <name> as the start of the program and loads the address of the line with <name> as its LABEL into the jump instruction at 100H-102H.

The value of this switch (and of /E:<name> below) is the ability to tell LINK-80 where to start execution when the assembled modules do not make this clear. Usually this is no problem because you link in a high-level language program (which LINK-80 takes as the main program by default), or you link only assembly language modules and only one has an END <name> statement to signal LINK-80 which assembly language program to execute first. But if two or more assembly language modules contain an END <name> statement, or if none of the assembly language modules contain an END <name> statement, then /G:<name> tells LINK-80 to use this module as the starting point for execution.

Programmers who want to execute an assembly language module before a high-level language program should use a CALL or INCLUDE statement at the beginning of the high-level language program to cause execution of the assembly language module before execution of the high-level language program.

EXIT

Switch	Action
/E	Use /E to link and load a program and perform some other functions on the files (for example, save it on a diskette) when you do not want to run the program at this time. When LINK-80 has finished the tasks, it will exit to the operating system. (The /G switch is the only other switch which exits LINK-80.) When linking is finished, LINK-80 outputs three numbers: start address, next available byte, number of 256-byte pages.
/E:<name>	The /E:<name> switch performs exactly like the plain /E switch but with one additional feature. <name> is a global symbol which was defined previously in one of the modules which is being linked and loaded. When LINK-80 sees <name>, it uses <name> as the start of the program and loads the address of the line with <name> as the LABEL into the jump instruction at 100H-102H. The value of this switch (and of /G:<name> above) is the ability to tell LINK-80 where to start execution when the assembled modules do not make this clear. Usually this is no problem because you link in a high-level language program (which LINK-80 takes as the main program by default), or you link only assembly language modules and only one has an END <name> statement to signal LINK-80 which assembly language program to execute first. But if two or more assembly language modules contain an END <name> statement, or if none of the assembly language modules contain an END <name> statement, then /E:<name> tells LINK-80 to use this module as the starting point for execution. Programmers who want to execute an assembly language module before a high-level language program should use a CALL or INCLUDE statement at the beginning of the high-level language program to cause this order of execution.

SAVE

Switch	Action
/N	The /N switch causes the assembled-linked program to be saved in a disk file. It is important that a filename always be specified for the /N switch. If you do not specify an extension, the default extension for the saved file is .COM. The COM filename will be the name the programmer appends the /N switch to. The /N switch must immediately follow the filename under which you wish to save the results of the link-load session.

The /N switch does not take effect unless a /E or /G switch follows it.

The most common error programmers make with the /N switch is to forget that they must specify at least two filenames; one as the file to be linked and another one as the name for the file to be saved. Therefore, at a minimum the command line should include:

```
L80 NEIL,NEIL/N/G
```

The first filename NEIL is the file to be loaded and linked; the second filename NEIL is the name for the COM file that will save the result of the link-loading session.

It is, of course, possible to specify filenames in any order. For example:

```
L80 NEIL/N,ASMSUB1,ASMSUB2,BASPROG/G
```

Here LINK-80 will load and link the files BASPROG, ASMSUB1, and ASMSUB2; then save the result in the file named NEIL.

From these two examples, it is possible to see that the filename followed by the /N save switch is not loaded; it is only a specification for an output file; you must also always name at least one input file, too.

You will use this switch almost every time you link a REL file because there is no other way to save the result of a link-load session and because not saving the result means you would have to link load again to run your program.

Once saved on disk, you need only type the COM filename at operating system command level to run the program.

/N:P By default, LINK-80 saves both the program and data areas in the COM file. If you wish to save only the program area to make your disk files smaller, use the /N switch in the form /N:P. With this switch set, only the program code will be saved.

Two of these switches (/N plus either a /G or a /E type) are all the switches required for most LINK-80 operations. Some additional functions are available through the use of other switches which allow programmers to manipulate the LINK-80 processes in more detail. The switches which turn on these additional functions are arranged in categories according to type of function. The function of each category is defined by the category name.

ADDRESS SETTING

Switch Action

/P The /P switch is used to set both the program and data origin. If you do not enter the /P switch, LINK-80 performs this task automatically, using a default address for both program and data. (103H for CP/M)

The format of the /P switch is:

/P:<address>,

The address value must be expressed in the current radix. The default radix is hexadecimal.

The /P switch is designed to allow you to place program (or code) segments at addresses other than the default. The default value for the /P switch is 103H.

REMEMBER: The /P switch takes effect as soon as it is seen, but it does not affect files already loaded. So be sure to place the /P switch before any files you want to load starting at the specified address. The /P switch and /D switch, when used, must be separated from the REL filename by a comma. For example,

```
L80 /P:103,NEIL,NEIL/N/E
```

The /P switch affects primarily the CSEG code in your assembly language program. If /P is given but not /D, both data and program (CSEG and DSEG) areas will be loaded starting at the /P:<address>. DSEG (and any COMMON areas) will be loaded first. If both /P and /D switches are given, /P sets the start of the CSEG area only. Normally, unless your programs are all CSEG, you will use /P and /D together.

Note especially that ASEG areas are not affected by the /P switch. So be careful to set the /P address outside any ASEG areas unless you want the program or data areas to write over the ASEG areas.

You may enter more than one /P switch during a single link session to place different program (code) segments at addresses which are not end to end. LINK-80 will automatically place one program segment (CSEG) after the next. You can cause space to be left between modules. However, some

restrictions one the placement of modules apply:

1. Be sure that program areas do not overlay one another. LINK-80 returns a warning error message if they do.

2. Be sure that the program areas are not split by data or COMMON areas; that is, a CSEG at 200H, a DSEG at 300H, and another CSEG at 400H is illegal. LINK-80 returns a fatal error in this case.

When the loading session is all done, LINK-80 wants to see a segment of memory loaded with data and COMMON and another segment loaded with program code. The code segments may have gaps between the modules as long as a data segment is not loaded between the start of the first code segment module and the end of the last code segment module, and vice versa. So, placing DSEG modules at 103H-115H, 150H-165H, 170H-175H, and CSEG modules at 200H-250H, 300H-350H, 400H-450H is acceptable. LINK and 80 will show Data between 103H and 175H and Program between 200H and 450H.

Note that any gaps you leave may contain data or program code from a previous program. LINK-80 does not initialize gaps to zero or null. This could cause unpredictable results.

/D The /D switch sets the origin for DSEG and COMMON areas. If you do not enter the /D switch, LINK-80 performs this task automatically, using a default address for both data and program. (103H for CP/M)

The format for the /D switch is:

```
/D:<address>,
```

The address for the /D switch must be in the current radix. The default radix is hexadecimal.

The /D switch is designed to allow you to place data and COMMON segments at addresses other than the default. The default value for the /D switch is 103H. The /D switch must be separated from the REL filenames by a comma. For example,

```
L80 /D:103,NEIL,NEIL/N/E
```

When the /P switch is used with the /D switch, data and common areas load starting at the address given with the /D switch. (The program will be

loaded beginning at the program origin given with the /P switch.) This is the only occasion when the address given in /P: is the start address for the actual program code.

REMEMBER: The /D switch takes effect as soon as LINK-80 "sees" the switch, so the /D switch has no effect on programs or data already loaded. Therefore, it is important to place the /D switch (as well as the /P switch) before the files you want to load starting at the address specified.

You may enter more than one /D switch during a single link session to place different program (code) segments at addresses which are not end to end. LINK-80 will automatically place one data segment (DSEG) after the next. You can cause space to be left between modules. However, some restrictions on the placement of modules apply:

1. Be sure that data areas do not overlay one another. LINK-80 returns a warning error message if they do.

2. Be sure that the data areas are not split by program areas; that is, a DSEG at 200H, a CSEG at 300H, and another DSEG at 400H is illegal. LINK-80 returns a fatal error in this case.

When the loading session is all done, LINK-80 wants to see a segment of memory loaded with data and COMMON and another segment loaded with program code. The data segments may have gaps between the modules as long as a program segment is not loaded between the start of the first data segment module and the end of the last data segment module, and vice versa. So, placing DSEG modules at 103H-115H, 150H-165H, 170H-175H, and CSEG modules at 200H-250H, 300H-350H, 400H-450H is acceptable. LINK and 80 will show Data between 103H and 175H and Program between 200H and 450H.

Note that any gaps you leave may contain data or program code from a previous program. LINK-80 does not initialize gaps to zero or null. This could cause unpredictable results.

ADDITIONAL NOTE FOR /P AND /D SWITCHES

If your program is too large for the loader, you will sometimes be able to load it anyway if you use /D and /P together. This way you will be able to load programs and data of a larger combined total. While LINK-80 is loading and linking, it builds a table consisting of five bytes for each program relative reference. By setting both /D and /P, you eliminate the need for LINK-80 to build this table, thus giving you some extra memory to work with.

To set the two switches, look to the end of the List file. Take the address you decided for the /D switch (where you want the DSEG to start loading), add the number for the total of data, add that number to 103H, add another 100H+1, and the result should be the /P: address for the start of the program area. The /D switch should be set at 103H or higher (D:103).

/R The /R switch "resets" LINK-80 to its initialized condition. LINK-80 scans the command line before it begins the functions commanded. As soon as LINK-80 sees the /R switch, all files loaded are ignored, LINK-80 resets itself, and the asterisk (*) prompt is returned showing that LINK-80 is running and waiting for you to enter a command line.

LIBRARY SEARCH

Switch	Action
/S	The /S switch causes LINK-80 to search the file named immediately prior to the switch for routines, subroutines, definitions for globals, and so on. In a command line, the filename with the /S switch appended must be separated from the rest of the command line by commas. For example:

```
L80 NEIL/N,MYLIB/S,NEIL/G
```

The /S switch is used to search library files only, including a library you constructed, using the LIB-80 Library Manager (see Chapter 8).

GLOBAL LISTING

Switch	Action
/U	The /U switch tells LINK-80 to list all undefined globals. The /U works only in command lines that do not include either a /G or a /E switch. Note that if your program contains any undefined globals, LINK-80 lists them automatically, unless the command line also contains a /S (library search) switch. In these cases, enter only the /U switch, and the list of undefined globals will be listed. Use CTRL-S to suspend the listing if you want to study a portion of the list that would scroll off the screen. Use CTRL-Q to restart the listing. The various runtime libraries provide definitions for the globals you need to run your high-level language programs. In addition to listing undefined globals, the /U switch directs LINK-80 to list the origin, end, and size of the program and data areas. These areas are listed as one lump area unless both the /P and /D switches are set. If both /P and /D are set, the start, end, and size of both areas are listed separately.
/M	The /M switch directs LINK-80 to list all globals, both defined and undefined, on the screen. The listing cannot be sent to a printer. In the listing, defined globals are followed by their values, and undefined globals are followed by an asterisk (*). In addition to listing all globals, the /M switch directs LINK-80 to list the origin, end, and size of the program and data areas. These areas are listed as one lump area unless both the /P and /D switches are set. If both /P and /D are set, the start, end, and size of both areas are listed separately.

RADIX SETTING

Switch	Action
/O	The /O switch sets the current radix to Octal. If you have a reason to use octal values in your program, give the /O switch in the command line. If you can think of no reason to switch to octal radix, then there is no reason to use this switch.
/H	The /H switch resets the current radix to Hexadecimal. Hexadecimal is the default radix. You do not need to give this switch in the command line unless you previously gave the /O switch and now want to return to hexadecimal.

SPECIAL CODE

Switch	Action
/X	The /X switch saves the "COM" file in Intel ASCII HEX format. The /X switch requires the /N switch appended to the same filename as the /X. For example: `L80 NEIL,NEIL/X/N/E` The file that is saved with the /X switch set is given the filename extension .HEX. The primary use of the /X switch is to prepare programs to be burned into PROMs. The hex format was originally developed to facilitate the movement of programs from one machine to another. The hex format provides more code checking than object code does. Also, a HEX file can be edited with some sophisticated line editors.
/Y	The /Y switch saves a file in a special format for use with Digital Research's Symbolic Debuggers, SID and ZSID. The /Y switch requires the /N and the /E (not /G) switches be given in the command line. For example: `L80 NEIL,NEIL/Y/N/E` The file that is saved with the /Y switch set is given the filename extension .SYM. A COM file will also be saved. So the sample command line above creates both NEIL.COM and NEIL.SYM. The SYM file contains the names and addresses of all globals, which allows you to use Digital Research's Symbolic Debuggers SID and ZSID with the SYM file.

6.3 ERROR MESSAGES

Errors encountered during the running of LINK-80 will return messages, most preceded by either the symbol ? or the symbol %. No error codes are returned, so once you understand the meaning of the message, error recognition should be easy.

?No Start Address

The /G switch was issued, but no main program has been loaded.

?Loading Error

The last file given for input was not a properly formatted LINK-80 object file.

?Out of Memory

Not enough memory to load the module.

?Command Error

Unrecognizable LINK-80 command.

?<filename> Not Found

<filename>, as given in the command string, did not exist.

?Start Symbol - <name> - Undefined

The /E:Name or /G:Name switch was given, but the Name specified was not defined.

?Nothing Loaded

A <filename>/S or /E or /G was given, but no object file was loaded. That is, an attempt was made to search a library, to exit LINK-80, or to execute a program, when in fact nothing had been loaded. For example:

```
TEST/N/E
```

Results in "?Nothing Loaded" because TEST/N names TEST.COM, but does not load TEST.REL.

To load a file, enter the filename. To save a file, enter a filename followed by the /N switch and either a /E or a /G switch. For example, any of the following sets of commands should work:

```
L80 NEIL,NEIL/N/E

or

L80
*NEIL
*NEIL/N/E

or

L80 NEIL/N,NEIL/E
```

?Can't Save Object File

A disk error occurred when the file was being saved. Usually, this means that the disk is full or that it is write-protected.

%2nd COMMON larger /XXXXXX/

When loading modules which include COMMON blocks, LINK-80 takes the size of the first COMMON block loaded to set the amount of memory needed before program code is loaded. If a subsequent module contains a COMMON block larger than the first one loaded, LINK-80 returns this error message. It means that the first definition of the COMMON block /XXXXXX/ encountered in the modules loaded was not the largest block defined with that name. Reorder module loading sequence or change COMMON block definitions so that all blocks are the same size.

%Mult. Def. Global YYYYYY

You have one global (PUBLIC) symbol name YYYYYY with more than one definition. Usually, two or more of the modules being loaded have declared the same symbol name as PUBLIC.

```
%Overlaying Program Area  ,Start    = xxxx
                          ,Public   = <symbol name> (xxxx)
                          ,External = <symbol name> (xxxx)
```

Usually this occurs when either /D or /P is set to an address inside the area taken by LINK-80. You should reset the switch address above 102H. It may also occur if you set addresses for programs loaded after some initial programs were loaded and the addresses were not set high enough. For example, if MYPROG is larger than 147 bytes and you enter the commands:

```
MYPROG,/P:150,SUBR1,FUNNY/N/E
```

you will receive the %Overlaying Program Area error message.

```
%Overlaying Data Area  ,Start    = xxxx
                       ,Public   = <symbol name> (xxxx)
                       ,External = <symbol name> (xxxx)
```

The /D and /P switches were set too close together. For example, if /D was given a higher address than /P but not high enough to be beyond the end of the program area, when the program is loaded, the top end will be laid over the data area. Or, if /D is lower than /P, /P was not high enough to prevent the beginning of the program from starting in the area already loaded with data.

?Intersecting Program Area
<u>or</u>
?Intersecting Data Area

The program and data areas intersect and an address or external chain entry is in this intersection. The final value cannot be converted to a current value since it is in the area intersection.

Origin Above Loader Memory, Move Anyway (Y or N)?
or
Origin Below Loader Memory, Move Anyway (Y or N)?

This message will appear only after either the /E or the /G switch command was given to LINK-80. If LINK-80 has not enough memory to load a module but a /E or /G has not been entered, you will receive the ?Out of Memory message.

LINK-80 can load modules only between its first address in memory and the top of available memory. If the program is too large for this space or if you set a /D and/or /P switch too high for the size of your program, LINK-80 runs out of memory and returns the Origin Above Loader Memory message.

If you set a /D and/or /P switch below the first address of LINK-80 (100H for CP/M), LINK-80 returns the Origin Below Loader Memory message. This prevents you from loading your program into memory designated for the operating system.

If a Y<CR> is given, LINK-80 will move the area and continue. If anything else is given, LINK-80 will exit. In either case, if the /N switch was given, the image will already have been saved.

Contents

Chapter 7 CREF-80 Cross Reference Facility

7.1 Creating a CREF Listing 7-1
    Creating a Cross Reference File 7-2
    Generating a Cross Reference Listing 7-2
7.2 CREF Listing Control Pseudo-ops 7-3

CHAPTER 7

CREF-80 CROSS REFERENCE FACILITY

A cross reference facility processes a specially assembled listing file to list the locations of all intermodule references and the locations of their definitions. The result is a cross reference listing. This cross reference listing can be used to aid debugging your program.

The CREF-80 Cross Reference Facility allows a programmer to process the cross reference file generated by MACRO-80. This cross reference file contains embedded control characters, set up during MACRO-80 assembly. CREF-80 interprets the control characters and generates a file that lists cross references among variables.

CREF-80 produces a listing, resembling the PRN listing of MACRO-80, with two additional features:

1. Each source statement is numbered with a cross reference number.

2. At the end of the listing, variable names appear in alphabetic order. Each name is followed by the line number where the variable is defined (flagged with #) followed by the numbers of other lines where the variable is referenced.

The CREF listing file replaces the MACRO-80 PRN List file and receives the filename extension .LST instead of .PRN.

7.1 CREATING A CREF LISTING

Creating a CREF listing involves two steps: (1) creating a cross reference file (.CRF), and (2) generating a cross reference listing (.LST). The first step occurs in the MACRO-80 macro assembler; the second in the CREF-80 Cross Reference Facility.

Creating a Cross Reference File

To create a cross reference file, set the /C switch in the MACRO-80 command line. For example:

```
M80 =NEIL/C
```

This command line assembles the file NEIL.MAC, generating the output files NEIL.REL (object file) and NEIL.CRF (cross reference file).

Generating a Cross Reference Listing

The cross reference listing is generated by running the .CRF file through CREF-80.

To invoke the cross reference facility, enter:

```
CREF80
```

CREF-80 will return an asterisk (*) prompt.

To create the cross reference listing file, enter:

```
=filename
```

where filename is the name of your .CRF file. For example:

```
CREF80 =NEIL
```

will generate a .LST file (NEIL.LST) containing the cross reference information.

This .LST file can be printed or sent to the terminal screen using operating system commands. Additionally, CREF-80 supports the same output device designations as MACRO-80. Simply enter the device designation in front of the filename. For example:

```
CREF80 LST:=NEIL
```

sends the .LST listing to the printer only (no disk file is generated).

```
CREF80 TTY:=NEIL
```

sends the .LST listing to the CRT only (no disk file is generated).

You will need to give a drive designation if you want the .LST file saved elsewhere than the currently logged drive (where the .CRF file resides). For example:

```
CREF80 B:=A:NEIL
```

saves NEIL.LST on drive B.

When finished, CREF-80 prompts with an asterisk. You may enter another =filename, or exit from CREF-80 to the operating system.

To exit CREF-80, enter:

```
CTRL-C
```

If you want the .LST file named differently from the default (.CRF filename and extension .LST), enter the name in front of the equal sign. For example:

```
   CREF80 NEIL.CRL=NEIL
or CREF80 NEILCREF=NEIL
```

The former command line generates a cross reference list file named NEIL.CRL; the latter generates a file named NEILCREF.LST.

Look at the filename extensions to distinguish a cross reference listing file from the listing file MACRO-80 normally generates. The listing file MACRO-80 normally generates (without the /C switch set in the command line) receives the default filename extension .PRN. The cross reference listing file generated by CREF-80 receives the default filename extension .LST.

7.2 CREF LISTING CONTROL PSEUDO-OPS

You may want the option of generating a cross reference listing for part of a program but not all of it. To control the listing or suppressing of cross references, use the cross reference listing control pseudo-ops, .CREF and .XCREF, in the source file for MACRO-80. These two pseudo-ops may be entered at any point in the program in the OPERATOR field. Like the other listing control pseudo-ops, .CREF and .XCREF support no ARGUMENTs.

Pseudo-op	Definition
.CREF	Create cross references. .CREF is the default condition. Use .CREF to restart the creation of a cross reference file after using the .XCREF pseudo-op. .CREF remains in effect until MACRO-80 encounters .XCREF. Note, however, that .CREF has no effect until the /C switch is set in the MACRO-80 command line.
.XCREF	Suppress cross references. .XCREF turns off the .CREF (default) pseudo-op. .XCREF remains in effect until MACRO-80 encounters .CREF. Use .XCREF to suppress the creation of cross references in selected portions of the file. Because neither .CREF nor .XCREF takes effect until the /C switch is set in the MACRO-80 command line, there is no need to use .XCREF if you want the usual List file (one without cross references); simply omit /C from the MACRO-80 command line.

Contents

CHAPTER 8 LIB-80 Library Manager

8.1 Sample LIB-80 Session 8-2
Building a Library 8-2
Listing a Library 8-2
8.2 LIB-80 Commands 8-3
Invoking LIB-80 8-3
Destination field 8-4
Source field 8-5
Additional Details About Source Modules 8-6
Switch field 8-8

CHAPTER 8

LIB-80 LIBRARY MANAGER

WARNING

> Read this chapter carefully and make a back-up copy of your libraries before using LIB-80. LIB-80 is very powerful and thus can be very destructive. It is easy to destroy a library with LIB-80.

LIB-80 is designed as a runtime library manager for CP/M versions of Microsoft FORTRAN-80 and COBOL-80. LIB-80 may also be used to create your own library of assembly language subroutines.

LIB-80 creates runtime libraries from assembly language programs that are subroutines to COBOL, FORTRAN, and other assembly language programs. The programs collected by LIB-80 may be special modules created by the programmer or modules from an existing library (FORLIB, for example). With LIB-80, you can build specialized runtime libraries for whatever execution requirements you design.

The value of building a library is that all the routines needed to execute a program can be linked with it into an executable object (COM) file by entering the library name followed by /S in a LINK-80 command line. For example:

```
L80 MAIN,NEWLIB/S,NEIL/N/G
```

This is much more convenient than entering the necessary subroutines individually, especially if there are many modules. With a library file you can be sure all the necessary modules will be linked into the COM file, plus there is no danger of running out of space on the LINK-80

command line. Additionally, the library makes this special collection of subroutines available for easy linking into any program.

8.1 SAMPLE LIB-80 SESSION

The two most common uses you will have for LIB-80 are building a library and listing a library. The following sample sessions illustrate the basic commands for these two uses.

BUILDING A LIBRARY:

```
A>LIB
*TRANLIB=SIN,COS,TAN,ATAN,ACOG
*EXP
*/E
A>
```

In this sample session, LIB invokes LIB-80, which returns an asterisk (*) prompt. TRANLIB is the name of the library being created. SIN,COS,TAN,ATAN,ACOG are filenames to be concatenated into TRANLIB. EXP is another filename to be concatenated into TRANLIB. (EXP could be listed on the previous command line; this example shows files entered singly and multiply.) /E causes LIB-80 to rename TRANLIB.LIB to TRANLIB.REL then to exit to CP/M.

LISTING A LIBRARY:

```
A>LIB
*TRANLIB.LIB/U
*TRANLIB.LIB/L
     .
     .
     .
(List of symbols in TRANLIB.LIB)
     .
     .
     .
*CTRL-C
A>
```

In this sample session, LIB invokes LIB-80. TRANLIB.LIB/U tells LIB-80 to search TRANLIB.LIB for any intermodule references that would not be defined during a single pass through the library

(that is, any "backward" referencing symbols). TRANLIB.LIB/L directs LIB-80 to list the modules in TRANLIB.LIB and the symbol definitions the modules contain. CTRL-C exits to CP/M without destroying any files.

WARNING

/E will destroy your current library if there is no new library under construction. This is a special danger to your FORTRAN runtime library FORLIB.REL. IF YOU ARE ONLY LISTING THE LIBRARY AND NOT REVISING IT, EXIT LIB-80 USING CTRL-C.

8.2 LIB-80 COMMANDS

Invoking LIB-80

To invoke LIB-80, enter:

```
LIB
```

LIB-80 will return an asterisk (*) prompt, indicating ready to accept commands. Each command in LIB-80 adds modules to the library under construction.

Commands to LIB-80 consist of an optional Destination field, a Source field, and an optional Switch field.

The format of a LIB-80 command is:

```
Destination=Source/Switch
```

Each field is described below. The general format for each field is shown in parentheses after the field name.

Destination field (filename=)

This field is optional. The equal sign is required if any entry is made in this field.

Enter in this field the filename (and extension, if you choose) for the library file you want to create.

If this field is omitted, LIB-80 defaults to the filename FORLIB. The default filename extension is .REL.

WARNING

> Do not confuse this default filename FORLIB.LIB with FORLIB.REL, the runtime library supplied with FORTRAN-80. These two libraries will not be the same unless you command LIB-80 to copy all the files from the FORTRAN runtime library to the new library. Furthermore, when you exit LIB-80, the default library name will be given the filename extension .REL, which means that it replaces the FORLIB.REL supplied with FORTRAN-80. For this reason, unless you want your FORTRAN-80 runtime library destroyed, we recommend emphatically that you always specify a Destination filename when creating a new library.

Source field (filename<module>)

Some entry is required in this field. All Source files must be REL files.

Source field entries tell LIB-80 which files or parts of files (modules) you want added to the destination library file. You have two choices for entries:

1. Filename(s) only

2. Any combination of filename(s) and module name(s)

The following syntax rules apply:

1. If a command consists of filenames only, the entries are separated by commas only. For example:

   FILE1,FILE2,FILE3

2. If a command consists of filenames and module names, the module names must be enclosed in angle brackets (<>). Modules follow the filename where they are found. Each filename<module name> combination is separated from other command line entries by commas. For example:

   FILE1,FILE2<MODZ>,FILE3<MODR>,FILE4

3. If more than one module is named from the same file, the module names, enclosed in angle brackets (<>), must be separated from each other by commas. For example:

   FILE1,FILE2<MODZ,MODR>,FILE3

   See Additional Details about Source Modules, option 2, below.

Files and modules are typically FORTRAN or COBOL subprograms or main programs, or ALDS assembly language programs that contain ENTRY, GLOBAL, or PUBLIC statements. (These statements are called entry points.) LIB-80 recognizes a module by its program name, which may be a filename, or a name given by either the .TITLE or the NAME pseudo-op in MACRO-80. All Source files must be REL files.

LIB-80 concatenates REL files and modules of REL files; that is, LIB-80 strings one file or module after the other.

So there is no difference between the command under syntax rule 2 above and

```
FILE1
FILE2<MODZ>
FILE3<MODR>
FILE4
```

Also, because the library file is built by concatenation, it is important to order the modules so that all intermodule references are "forward." That is, the module containing the external reference should physically appear ahead of the module containing the ENTRY point (the definition). Otherwise, when you direct LINK-80 to search the library, LINK-80 may not satisfy all references on a single pass through the library.

Additional Details about Source Modules

To extract modules from previous libraries and other REL files, LIB-80 uses a powerful syntax to specify ranges of modules within a REL file.

These ranges may be from one module to the entire file (in which case no module specification is given).

The basic principle of specifying a range of modules is, generally, that any module named in a command will be included. (There is an exception, when specifying a relative offset range--item 6, below.)

The options for specifying modules are:

1. One module only
   Enter the module name. For example:

   ```
   FILE1<MODZ>
   ```

   includes only module MODZ of FILE1.

2. Several discontiguous modules from one file
   Enter the module names separated by commas. For example:

   ```
   FILE1<MODZ,MODR,MODK>
   ```

   includes modules MODZ, MODR, and MODK. Note that these modules may be given in any order you need them concantenated for a proper one-pass search, regardless of their order in the original file.

3. From the first module through the named module
   Enter two periods (..) and the name of the last module to be included. For example:

   ```
   FILE1<..MODK>
   ```

   includes all modules from the first module in FILE1 through module MODK.

4. From a named module through the last module
   Enter the name of the module that starts the range followed by two periods (..). For example:

   ```
   FILE1<MODR..>
   ```

   includes all the modules, beginning with module MODR, through the last module in FILE1.

5. From one named module through another named module
   Enter the name of the module that starts the range followed by two periods (..) followed by the name of the module that ends the range. For example:

   ```
   FILE1<MODZ..MODK>
   ```

   includes all modules, beginning with module MODZ, through module MODK.

6. Relative offset range
   Enter the module name followed by a + or - and the number of modules to be included. + means following the named module. - means preceding the named module. The named module is not included in the library. The offset number must be an integer in the range 1 to 255. For example:

   ```
   FILE1<MODZ+2>
   ```

   includes the two modules immediately following module MODZ. While

   ```
   FILE1<MODK-3>
   ```

   includes the three modules immediately preceding module MODK.

Additionally, ranges and offsets may be used together. For example:

```
FILE1<MODR+1..MODK-1>
```

includes all the modules between module MODR and module MODK (but neither MODR nor MODK is included).

7. All modules in a file
   Enter the filename only. For example:

   ```
   FILE1
   ```

   includes the entire file (all modules in FILE1).

Switch field (/switch)

An entry in the Switch field commands LIB-80 to perform additional functions. A Switch field entry is a letter preceded by a slash mark (/).

WARNING

> /E will destroy your current library if there is no new library under construction. This is a special danger to your FORTRAN runtime library FORLIB.REL because FORLIB is the default filename used if you do not specify a destination filename. Therefore, unless you want to delete your complete FORTRAN runtime library, give LIB-80 a destination filename for the new library. If you are only listing the library and not revising it, exit LIB-80 using CTRL-C.

Switch	Action
/E	Exit to CP/M. <u>If you are not creating a new library or revising an existing library, use CTRL-C instead of /E.</u> The library under construction (.LIB) is renamed to .REL and any previous copy of the library file is deleted. This is why /E is so dangerous and not to be used unless you are constructing a new library. Again, we recommend emphatically that you <u>always</u> enter a filename in the Destination field of the LIB-80 command line.
/R	Rename the library currently being built (.LIB) to .REL. <u>The same warnings and cautions apply to /R as apply to /E.</u> The previous copy of the library is deleted. Use /R only if you are building a new library. /R performs the same functions as /E, but does not exit to CP/M on completion. Use /R instead of /E when you want to exit the current library but want to continue using LIB-80 for other library managing.
/L	List the modules in the file specified and the symbol definitions the modules contain. The contents of a file are listed in cross reference format. Listings are currently always sent to the terminal; use CTRL-P before running LIB-80 to send the listing to the printer.
/U	Use /U to list the symbols which could be undefined in a single pass through a library. If a symbol in a library module refers "backward" (to a preceding module), /U will list that symbol.
/C	Use /C to clear commands from LIB-80 without exiting the LIB-80 program. The library under construction is deleted and the LIB-80 session starts over. The asterisk (*) prompt will appear. Use /C if you specified the wrong module(s) or the wrong order and want to start over with new LIB-80 commands.

/O Use /O to set typeout mode to Octal radix. /O will be given together with the /L switch, which commands LIB-80 to list. REMEMBER: When switches are given together, a slash must precede each switch. For example:

NEWLIB/L/O

/H Use /H to set typeout mode to Hexadecimal radix. Hexadecimal is the default radix.

Contents

Appendix A Compatibility with Other Assemblers

Appendix B The Utility Software Package with TEKDOS

- B.1 TEKDOS Command Files B-1
- B.2 MACRO-80 B-1
- B.3 CREF-80 B-2
- B.4 LINK-80 B-2

Appendix C ASCII Character Codes

Appendix D Format of LINK Compatible Object Files

Appendix E Table of MACRO-80 Pseudo-ops

Appendix F Table of Opcodes

- F.1 Z80 Opcodes F-1
- F.2 8080 Opcodes F-3

APPENDIX A

Compatibility with Other Assemblers

The $EJECT and $TITLE controls are provided for compatability with Intel's ISIS assembler. The dollar sign must appear in column 1 only if spaces or tabs separate the dollar sign from the control word. The control word

    $EJECT

is the same as the MACRO-80 PAGE pseudo-op.

The control word

    $TITLE('text')

is the same as the MACRO-80 SUBTTL <text> pseudo-op.

The Intel operands PAGE and INPAGE generate Q errors when used with the MACRO-80 CSEG or DSEG pseudo-ops. These errors are warnings; the assembler ignores the operands.

When MACRO-80 is invoked, the default for the origin is Code Relative 0. With the Intel ISIS assembler, the default is Absolute 0.

With MACRO-80, the dollar sign ($) is a defined constant that indicates the value of the location counter at the start of the statement. Other assemblers may use a decimal point or an asterisk. Other constants are defined by MACRO-80 to have the following values:

```
A=7     B=0     C=1     D=2      E=3
H=4     L=5     M=6     SP=6     PSW=6
```

APPENDIX B

The Utility Software Package with TEKDOS

The command formats for MACRO-80, LINK-80, and CREF-80 differ slightly under the TEKDOS operating system.

B.1 TEKDOS COMMAND FILES

The files M80, L80, and C80 are actually TEKDOS command files for the assembler, loader, and cross reference programs, respectively. These command files set the emulation mode to 0 and select the Z-80 assembler processor (see TEKDOS documentation), then execute the appropriate program file. You will note that all of these command files are set up to execute the Microsoft programs from drive #1. LINK-80 will also look for the library on drive #1. If you wish to execute any of this software from drive #0, the command file must be edited. Then, LINK-80 should be given an explicit library search directive, such as MYLIB-S. See the Switches section in Chapter 6, LINK-80 Linking Loader.

Filenames under TEKDOS do not use the Utility Software Package default filename extensions.

B.2 MACRO-80

The MACRO-80 assembler accepts command lines only (the invoke command, M80, and all filenames and switches must be on one line). No prompt is displayed, and the interactive commands (,TTY:=TTY: and ,LPT:=TTY:) are not accepted. Commands have the same format as TEKDOS assembler commands; that is, up to three filenames or device names plus optional switches.

M80 [object] [list] source [switch [switch [...]]]

The object and list file entries are optional. These files will not be created if the parameters are omitted. Any

error messages will still be displayed on the console. The available switches are described in Chapter 5 of this manual. All command line entries may be delimited by commas or spaces. If you do not want to request an object file, you must enter a <space comma space> between the M80 entry and the name of the list file. For example:

```
M80 , LIST SOURCE
```

B.3 CREF-80

The form of commands to CREF-80 is:

```
C80 list source
```

Both filenames are required. The source file is always the name of a CREF-80 file created during assembly by the C switch.

Example:

To create a CREF-80 file from the source TSTMAC using MACRO-80, enter:

```
M80 , TSTCRF TSTMAC C
```

To create a cross reference listing from the CREF-80 file TSTCRF, enter:

```
C80 TSTLST TSTCRF
```

B.4 LINK-80

With TEKDOS, the LINK-80 loader accepts interactive commands only. Command lines are not supported.

When LINK-80 is invoked, and whenever it is waiting for input, it will prompt with an asterisk. Commands are lists of filenames and/or devices separated by commas or spaces and optionally interspersed with switches. The input to LINK-80 must be Microsoft relocatable object code (not the same as TEKDOS loader format).

Switches to LINK-80 are delimited by hyphens under TEKDOS, instead of slashes. All LINK-80 switches (as documented in Chapter 6) are supported, except -G and -N, which are not implemented at this time.

EXAMPLE:

1. Assemble a MACRO-80 program named XTEST, creating an object file called XREL and a listing file called XLST:

```
>M80 XREL XLST XTEST
```

2. Load XTEST and save the loaded module:

```
>L80
*XREL-E
[04AD  22B8]
*DOS*ERROR 46
L80 TERMINATED
>M XMOD 400 22B8 04AD
```

Note that -E exits via an error message due to execution of a Halt instruction. The memory image is intact, however, and the TEKDOS Module command may be used to save it. Once a program is saved in module format, it may then be executed directly without going through LINK-80 again.

The bracketed numbers printed by LINK-80 before exiting are the entry point address and the highest address loaded, respectively. The loader default is to begin loading at 400H. However, the loader also places a jump to the start address in location 0, which allows execution to begin at 0. The memory locations between 0003 and 0400H are reserved for SRB's and I/O buffers at runtime.

APPENDIX C

ASCII CHARACTER CODES

Dec	Hex	CHR	Dec	Hex	CHR	Dec	Hex	CHR
000	00H	NUL	043	2BH	+	086	56H	V
001	01H	SOH	044	2CH	,	087	57H	W
002	02H	STX	045	2DH	-	088	58H	X
003	03H	ETX	046	2EH	.	089	59H	Y
004	04H	EOT	047	2FH	/	090	5AH	Z
005	05H	ENQ	048	30H	0	091	5BH	[
006	06H	ACK	049	31H	1	092	5CH	\
007	07H	BEL	050	32H	2	093	5DH	]
008	08H	BS	051	33H	3	094	5EH	^
009	09H	HT	052	34H	4	095	5FH	
010	0AH	LF	053	35H	5	096	60H	`
011	0BH	VT	054	36H	6	097	61H	a
012	0CH	FF	055	37H	7	098	62H	b
013	0DH	CR	056	38H	8	099	63H	c
014	0EH	SO	057	39H	9	100	64H	d
015	0FH	SI	058	3AH	:	101	65H	e
016	10H	DLE	059	3BH	;	102	66H	f
017	11H	DC1	060	3CH	<	103	67H	g
018	12H	DC2	061	3DH	=	104	68H	h
019	13H	DC3	062	3EH	>	105	69H	i
020	14H	DC4	063	3FH	?	106	6AH	j
021	15H	NAK	064	40H	@	107	6BH	k
022	16H	SYN	065	41H	A	108	6CH	l
023	17H	ETB	066	42H	B	109	6DH	m
024	18H	CAN	067	43H	C	110	6EH	n
025	19H	EM	068	44H	D	111	6FH	o
026	1AH	SUB	069	45H	E	112	70H	p
027	1BH	ESCAPE	070	46H	F	113	71H	q
028	1CH	FS	071	47H	G	114	72H	r
029	1DH	GS	072	48H	H	115	73H	s
030	1EH	RS	073	49H	I	116	74H	t
031	1FH	US	074	4AH	J	117	75H	u
032	20H	SPACE	075	4BH	K	118	76H	v
033	21H	!	076	4CH	L	119	77H	w
034	22H	"	077	4DH	M	120	78H	x
035	23H	#	078	4EH	N	121	79H	y
036	24H	$	079	4FH	O	122	7AH	z
037	25H	%	080	50H	P	123	7BH	{
038	26H	&	081	51H	Q	124	7CH	|
039	27H	'	082	52H	R	125	7DH	}
040	28H	(	083	53H	S	126	7EH	~
041	29H	)	084	54H	T	127	7FH	DEL
042	2AH	*	085	55H	U			

Dec=decimal, Hex=hexadecimal (H), CHR=character.
LF=Line Feed, FF=Form Feed, CR=Carriage Return, DEL=Rubout

APPENDIX D

FORMAT OF LINK COMPATIBLE OBJECT FILES

This appendix contains reference material for users who wish to know the load format of LINK-80 relocatable object files. None of this material is necessary to the operation of ALDS. There is nothing in the format material presented here which can be manipulated by the user. The material is highly technical, and it is not presented in any tutorial manner.

LINK-compatible object files consist of a bit stream. Individual fields within the bit stream are not aligned on byte boundaries, except as noted below. Use of a bit stream for relocatable object files keeps the size of object files to a minimum, thereby decreasing the number of disk reads/writes.

There are two basic types of load items: Absolute and Relocatable. The first bit of an item indicates one of these two types. If the first bit is a 0, the following 8 bits are loaded as an absolute byte. If the first bit is a 1, the next 2 bits are used to indicate one of four types of relocatable items:

00	Special LINK item (see below).
01	Program Relative. Load the following 16 bits after adding the current Program base.
10	Data Relative. Load the following 16 bits after adding the current Data base.
11	Common Relative. Load the following 16 bits after adding the current Common base.

Special LINK items consist of the bit stream 100 (read one-zero-zero) followed by:

a four-bit control field

an optional A field consisting of a two-bit address type that is the same as the two-bit field described above, except 00 specifies absolute address

an optional B field consisting of 3 bits that give a symbol length and up to 8 bits for each character of the symbol

A general representation of a special LINK item is:

```
1 00 xxxx    yy nn     zzz + characters of symbol name
             \___/     \_______________________________/
            A field                B field
```

where:

xxxx	is	four-bit control field (0-15 below)
yy	is	two-bit address type field
nn	is	sixteen-bit value
zzz	is	three-bit symbol length field

The following special types have a B-field only:

0 Entry symbol (name for search)
1 Select COMMON block
2 Program name
3 Request library search
4 Extension LINK items (see below)

The following special LINK items have both an A field and a B field:

5 Define COMMON size
6 Chain external (A is head of address chain, B is name of external symbol)
7 Define entry point (A is address, B is name)

The following special LINK items have an A field only:

8	External - offset. Used for JMP and CALL to externals
9	External + offset. The A value will be added to the two bytes starting at the current location counter immediately before execution.
10	Define size of Data area (A is size)
11	Set loading location counter to A
12	Chain address. A is head of chain. Replace all entries in chain with current location counter. The last entry in the chain has an address field of absolute zero.
13	Define program size (A is size)
14	End program (forces to byte boundary)

The following special LINK item has neither an A nor a B field:

15 End file

An Extension LINK item follows the general format of a B-field-only special LINK item, but the contents of the B-field are not a symbol name. Instead, the symbol area contains one character to identify the type of extension LINK item, followed by from 1 to 7 characters of additional information.

Thus, every extension LINK item has the format:

1 00 0100 111 s bbbbbb

where:		
	111	is 3 bits containing the length of the field bbbbbb (0 implys 1 since F80 emits entry length of 0 for Blank Common),
	s	is an eight bit extension LINK item sub-type identifier, and
	bbbbbb	are 1 to 6 bytes for additional information. If used as B field for name, bbbbbb may be only 6 characters.

The present extension LINK item sub-types are:

5	X'35'	COBOL overlay segment sentinel
A	X'41'	Arithmetic Fixup (Arithmetic Operator)
B	X'42'	Arithmetic Fixup (External Reference)
C	X'43'	Arithmetic Fixup (Area Base + Offset)

Descriptions of Sub-types

Sub-type 5

When the overlay segment sentinel is encountered by LINK-80, lll receives the value 010 (binary), and the current overlay segment number is set to the value b+49. If the previously existing segment number was non-zero and the /N switch is in effect, the data area is written to disk in a file whose name is the current program name and whose extension is Vnn, where nn are the two hexadecimal digits representing the number b+49 (decimal).

Sub-types A,B,C

Sub-types A, B, and C allow the processing of Polish Arithmetic text. Items must be read as Reverse Polish Expression. One or more Value items (sub-type B or C) are followed by one or more Arithmetic Operators (sub-type A) and end with a Store-Result Arithmetic Operator (B.STBT or B.STWD).

All Items are put in the Fixup Table afer any offset entries have been converted to final addresses. The Polish expression is executed out of the Fixup Table at the end of link. The result is stored at the PC given when the Items were read.

APPENDIX E

Table of MACRO-80 Pseudo-ops

Notation: * means Z80 pseudo-op
no stars means 8080 pseudo-op

SINGLE-FUNCTION PSEUDO-OPS

Instruction Set Selection

.Z80
.8080

Data Definition and Symbol Definition

<name> ASET <exp>
BYTE EXT <symbol>
BYTE EXTRN <symbol>
BYTE EXTERNAL <symbol>
DB <exp>[,<exp>...]
DB <string>[<string>...]
DC <string>
DDB <exp>[,<exp>...]
* DEFB <exp>[,<exp>...]
* <name> DEFL <exp>
* DEFM <string>[,<string>...]
* DEFS <exp>[,<val>]
* DEFW <exp>[,<exp>...]
DS <exp>[,<val>]
DW <exp>[,<exp>...]
ENTRY <name>[,<name>...]
<name> EQU <exp>
EXT <name>[,<name>...]
EXTRN <name>[,<name>...]
* EXTERNAL <name>[,<name>...]
GLOBAL <name>[,<name>...]
PUBLIC <name>[,<name>...]
<name> SET <exp> (not in .Z80 mode)

PC Mode Pseudo-ops

```
ASEG
CSEG
DSEG
COMMON /<block name>/
ORG <exp>
.PHASE <exp>/.DEPHASE
```

File Related Pseudo-ops

```
.COMMENT <delim><text><delim>
END [<exp>]
INCLUDE <filename>
$INCLUDE <filename>
MACLIB <filename>
.RADIX <exp>
.REQUEST <filename>[,<filename>...]
```

Listing Pseudo-ops

Format Control Pseudo-ops

```
*   *EJECT [<exp>] (one star is part of *EJECT)
    PAGE <exp>
    SUBTTL <text>
    TITLE <text>
    $TITLE
```

General Listing Control Pseudo-ops

```
.LIST
.XLIST
.PRINTX <delim><text><delim>
```

Conditional Listing Control Pseudo-ops

```
.SFCOND
.LFCOND
.TFCOND
```

Expansion Listing Control Pseudo-ops

```
.LALL
.SALL
.XALL
```

Cross-Reference Listing Control Pseudo-ops

```
.XCREF
.CREF
```

MACRO FACILITY PSEUDO-OPS

Macro Pseudo-ops

```
<name> MACRO <parameter>[,<parameter>...]
ENDM
EXITM
LOCAL <parameter>[,<parameter>...]
```

Repeat Pseudo-ops

```
REPT <exp>
IRP <dummy>,<parameters in angle brackets>
IRPC <dummy>,string
```

Conditional Assembly Facility

```
*  COND <exp>
   ELSE
*  ENDC
   ENDIF
   IF <exp>
   IFB <arg>
   IFDEF <symbol>
   IFDIF <arg1>,<arg2>
   IFE <exp>
   IFF <exp>
   IFIDN <arg1>,<arg2>
   IFNB <arg>
   IFNDEF <symbol>
   IFT <exp>
   IF1
   IF2
```

APPENDIX F

Table of Opcodes

The opcodes are listed alphabetically by instruction set. For details, refer to the reference books listed in Chapter 1.

F.1 Z80 OPCODES

Opcode	Function
ADC A	Add with Carry to Accumulator
ADC HL,rp	Add Register Pair with Carry to HL
ADD	Add
AND	Logical AND
BIT	Test Bit
CALL addr	Call Subroutine
CALL cond,addr	Call Conditional
CCF	Complement Carry Flag
CP	Compare
CPD	Compare, Decrement
CPDR	Compare, Decrement, Repeat
CPI	Compare, Increment
CPIR	Compare, Increment, Repeat
CPL	Complement Accumulator
DAA	Decimal Adjust Accumulator
DEC	Decrement
DI	Disable Interrupts
DJNZ	Decrement and Jump if Not Zero
EI	Enable Interrupts
EX	Exchange
EXX	Exchange Register Pairs and Alternatives
HALT	Halt
IM x	Set Interrupt Mode
IN	Input
INC	Increment
IND	Input, Decrement
INDR	Input, Decrement, Repeat
INI	Input, Increment
INIR	Input, Increment, Repeat
JP addr	Jump
JP cond,addr	Jump Conditional
JR	Jump Relative

```
JR    cond,addr      Jump Relative Conditional
LD    A,(addr)       Load Accumulator Direct
LD  A,(BC) or (DE)   Load Accumulator Secondary
LD    A,I            Load Accumulator from Interrupt Vector Register
LD    A,R            Load Accumulator from Refresh Register
LD  HL,(addr)        Load HL Direct
LD    data           Load Immediate
LD  xy,(addr)        Load Index Register Direct
LD    reg,(HL)       Load Register
LD  reg,(xy+disp)    Load Register Indexed
LD  rp,(addr)        Load Register Pair Direct
LD    SP,HL          Move HL to Stack Pointer
LD    SP,xy          Move Index Register to Stack Pointer
LD    dst,scr        Move Register-to-Register
LD    (addr),A       Store Accumulator Direct
LD  (BC) or (DE),A   Store Accumulator Secondary
LD    I,A            Store Accumulator to Interrupt Vector Register
LD    R,A            Store Accumulator to Refresh Register
LD  (addr),HL        Store HL Direct
LD  (HL),data        Store Immediate to Memory
LD  (xy+disp),data   Store Immediate to Memory Indexed
LD  (addr),xy        Store Index Register Direct
LD    (HL),reg       Store Register
LD  (xy+disp),reg    Store Register Indexed
LD  (addr),rp        Store Register Pair Direct
LDD                  Load, Decrement
LDDR                 Load, Decrement, Repeat
LDI                  Load, Increment
LDIR                 Load, Increment, Repeat
NEG                  Negate (Two's Complement) Accumulator
NOP                  No Operation
OR                   Logical OR
OUT                  Output
OUTD                 Output, Decrement
OTDR                 Output, Decrement, Repeat
OUTI                 Output, Increment
OTIR                 Output, Increment, Repeat
POP                  Pop from Stack
PUSH                 Push to Stack
RES                  Reset Bit
RET                  Return from Subroutine
RET   cond           Return Conditional
RETI                 Return from Interrupt
RETN                 Return from Non-Maskable Interrupt
RL                   Rotate Left Through Carry
RLA                  Rotate Accumulator Left Through Carry
RLC                  Rotate Left Circular
RLCA                 Rotate Accumulator Left Circular
RLD                  Rotate Accumulator and Memory Left Decimal
RR                   Rotate Right Through Carry
RRA                  Rotate Accumulator Right Through Carry
RRC                  Rotate Right Circular
RRCA                 Rotate Accumulator Right Circular
RRD                  Rotate Accumulator and Memory Right Decimal
RST                  Restart
```

SET	Set Bit
SBC	Subtract with Carry (Borrow)
SCF	Set Carry Flag
SLA	Shift Left Arithmetic
SRA	Shift Right Arithmetic
SRL	Shift Right Logical
SUB	Subtract
XOR	Logical Exclusive OR

F.2 8080 OPCODES

Opcode	Function
ADC,ACI	Add with Carry
ADD,ADI	Add
ANA,ANI	Logical AND
CALL	Call Subroutine
CC	Call on Carry
CM	Call on Minus
CMA	Complement Accumulator
CMC	Complement Carry
CMP,CPI	Compare
CNC	Call on No Carry
CNZ	Call on Not Zero
CP	Call on Positive
CPE	Call on Parity Even
CPO	Call on Parity Odd
CZ	Call on Zero
DAA	Decimal Adjust
DAD	16-bit Add
DCR	Decrement
DCX	16-bit Decrement
DI	Disable Interrupts
EI	Enable Interrupts
HLT	Halt
IN	Input
INR	Increment
INX	Increment 16 bits
JC	Jump on Carry
JM	Jump on Minus
JMP	Jump
JNC	Jump on Not Carry
JNZ	Jump on Not Zero
JP	Jump on Positive
JPE	Jump on Parity Even
JPO	Jump on Parity Odd
JZ	Jump on Zero
LDA	Load Accumulator
LDAX	Load Accumulator Indirect
LHLD	Load HL Direct
LXI	Load 16 bits

MOV	Move
MVI	Move Immediate
NOP	No Operation
ORA,ORI	Logical OR
OUT	Output
PCHL	HL to Program Counter
POP	Pop from Stack
PUSH	Push to Stack
RAL	Rotate with Carry Left
RAR	Rotate with Carry Right
RC	Return on Carry
RET	Return from Subroutine
RLC	Rotate Left
RM	Return on Minus
RNC	Return on No Carry
RNZ	Return on Not Zero
RP	Return on Positive
RPE	Return on Parity Even
RPO	Return on Parity Odd
RRC	Rotate Right
RST	Restart
RZ	Return on Zero
SBB,SBI	Subtract with Borrow
SHLD	Store HL Direct
SPHL	HL to Stack Pointer
STA	Store Accumulator
STAX	Store Accumulator Indirect
STC	Set Carry
SUB,SUI	Subtract
XCHG	Exchange D and E, H and L
XRA,XRI	Logical Exclusive OR
XTHL	Exchange Top of Stack, HL

INDEX

$EJECT 4-28
$INCLUDE 4-23
$TITLE 4-30

8080 Opcodes 4-3
8080 Opcodes as Operands . . . 3-13

ASEG 4-14
ASET 4-12

BYTE EXT 4-10
BYTE EXTERNAL 4-10
BYTE EXTRN 4-10

Calling a Macro 4-38
Character Constants 3-11
Comments 3-2
COMMON 4-17
COND 4-49
CREF-80 Cross Reference Facility 7-1
CREF-80 Cross-Reference Facility 2-4
CSEG 4-15, A-1
Current Program Counter . . . 3-13, A-1

DB 4-5
DC 4-6
DEFB 4-5
DEFL 4-12
DEFM 4-5
DEFS 4-7
DEFW 4-8
Device names as files 5-12
DS 4-7
DSEG 4-16, A-1
DW 4-8

ELSE 4-50
END 4-22
ENDC 4-50
ENDIF 4-50
ENDM 4-44
ENTRY 4-11
EQU 4-9
Error Messages
  LINK-80 6-19
  MACRO-80 5-15
EXITM 4-44
EXT 4-10
EXTERNAL 4-10
EXTERNAL Symbols 3-6
EXTRN 4-10

Figure

Developing assembly programs 1-5
Device Designations without filenames 5-12
Loading changes Relative address to fixed 1-7
ORG in relative modes is an offset 1-8
PUBLIC symbol linked with EXTERNAL 1-6
Relationships among programs 1-10
Table of Link-80 Switches . 6-5
File Format 3-1, 5-13

GLOBAL 4-11

IF 4-49
IF1 4-49
IF2 4-49
IFB 4-49
IFDEF 4-49
IFDIF 4-50
IFE 4-49
IFF 4-49
IFIDN 4-50
IFNB 4-50
IFNDEF 4-49
IFT 4-49
INCLUDE 4-23
IRP 4-42
IRPC 4-43

LABEL: 3-4
LIB-80 Command Format 8-3
LIB-80 Library Manager 2-4
LIB-80 Modules 8-5
LINK-80 Error Messages 6-19
LINK-80 Linking Loader 2-3, 6-1
Listing Formats 5-13
LOCAL 4-45

MACLIB 4-23
MACRO 4-37
MACRO-80 Error Codes and Messages 5-15
MACRO-80 Listing Files 5-13
MACRO-80 Macro Assembler . . . 5-1
Modes 3-7
Modes Rules for symbols in expressions 3-12

NAME 4-24
Numbers as operands 3-10

Operands 3-10
Operator Order of Precedence . 3-17
Operators 3-14
ORG 4-18

PAGE 4-28, A-1
Pseudo-ops
$EJECT 4-28
$INCLUDE 4-23
$TITLE 4-30
ASEG 4-14

ASET 4-12
Block Listing 4-34
BYTE EXT 4-10
BYTE EXTERNAL 4-10
BYTE EXTRN 4-10
COMMON 4-17
COND 4-49
Conditional 4-48
Conditional Listing 4-33
CSEG 4-15, A-1
Data Definition 4-4
DB 4-5
DC 4-6
DEFB 4-5
DEFL 4-12
DEFM 4-5
DEFS 4-7
DEFW 4-8
DS 4-7
DSEG 4-16, A-1
DW 4-8
ELSE 4-50
END 4-22
ENDC 4-50
ENDIF 4-50
ENDM 4-44
ENTRY 4-11
EQU 4-9
EXITM 4-44
Expansion Listing 4-34
EXT 4-10
EXTERNAL 4-10
EXTRN 4-10
Format Control 4-28
General Listing 4-31
GLOBAL 4-11
IF 4-49
IF1 4-49
IF2 4-49
IFB 4-49
IFDEF 4-49
IFDIF 4-50
IFE 4-49
IFF 4-49
IFIDN 4-50
IFNB 4-50
IFNDEF 4-49
IFT 4-49
INCLUDE 4-23
IRP 4-42
IRPC 4-43
Listing 4-27
LOCAL 4-45
MACLIB 4-23
MACRO 4-37
Macro Listing 4-34
NAME 4-24
ORG 4-18

PAGE 4-28, A-1
PC Mode 4-13
PUBLIC 4-11
REPT 4-41
SET 4-12
SUBTTL 4-30, A-1
Symbol Definition 4-4
TITLE 4-29
.PHASE 4-19
.DEPHASE 4-19
.COMMENT 4-21
.RADIX 4-25
.REQUEST 4-26
*EJECT 4-28
.LIST 4-31
.XLIST 4-31
.PRINTX 4-32
.SFCOND 4-33
.LFCOND 4-33
.TFCOND 4-33
.XALL 4-34
.LALL 4-34
.SALL 4-34
.CREF 4-35
.XCREF 4-35
.CREF 7-3
.XCREF 7-3
PUBLIC 4-11
PUBLIC Symbols 3-5

REPT 4-41
Restrictions on module placement with LINK-80 6-12 to 6-13
Rules for EXTERNALS in expressions 3-12

SET 4-12
Special Macro Operators 4-46
% 4-46
! 4-46
;; 4-46
& 4-46
Special Radix Notation 3-10
Statement Line Format 3-1
Strings 3-11
SUBTTL 4-30, A-1
Switches
LIB-80 8-9
/C 8-9
/E 8-9
/H 8-10
/L 8-9
/O 8-10
/R 8-9
/U 8-9
LINK-80
/D 6-12
/E 6-8
/G 6-6
/H 6-17

/M 6-16
/N 6-9
/N:P 6-10
/O 6-17
/P 6-11
/R 6-14
/S 6-15
/U 6-16
/X 6-18
/Y 6-18
MACRO-80 5-6
/H 5-6
/I 5-7
/L 5-7
/M 5-8
/O 5-6
/P 5-7
/R 5-6
/X 5-8
/Z 5-7
Symbol Table format 5-14
Symbols 3-3
Symbols in expressions 3-12
Symbols Rules 3-3
Syntax Notation 1-3
System Requirements 1-2

TEKDOS B-1
TITLE 4-29

Z80 Opcodes 4-3

.PHASE 4-19
.DEPHASE 4-19
.COMMENT 4-21
.RADIX 4-25
.REQUEST 4-26
*EJECT 4-28
.LIST 4-31
.XLIST 4-31
.PRINTX 4-32
.PRINTX 4-32
.SFCOND 4-33
.LFCOND 4-33
.TFCOND 4-33
.XALL 4-34
.LALL 4-34
.SALL 4-34
.CREF 4-35
.XCREF 4-35
/O - MACRO-80 5-6
/H - MACRO-80 5-6
/R - MACRO-80 5-6
/L - MACRO-80 5-7
/Z - MACRO-80 5-7
/I - MACRO-80 5-7
/P - MACRO-80 5-7
/M - MACRO-80 5-8

/X - MACRO-80 5-8
/G - LINK-80 6-6
/E - LINK-80 6-8
/N - LINK-80 6-9
/N:P - LINK-80 6-10
/P - LINK-80 6-11
/D - LINK-80 6-12
/R - LINK-80 6-14
/S - LINK-80 6-15
/U - LINK-80 6-16
/M - LINK-80 6-16
/O - LINK-80 6-17
/H - LINK-80 6-17
/X - LINK-80 6-18
/Y - LINK-80 6-18
.CREF 7-3
.XCREF 7-3
/E - LIB-80 8-9
/R - LIB-80 8-9
/L - LIB-80 8-9
/U - LIB-80 8-9
/C - LIB-80 8-9
/O - LIB-80 8-10
/H - LIB-80 8-10
$ - Current Program Counter . A-1
% 4-46
! 4-46
;; 4-46
& 4-46

MICROSOFT®
10700 Northup Way, Bellevue, WA 98004

Software Problem Report

Name ______________________________

Street ______________________________

City ______________ State __________ Zip __________

Phone ______________ Date ______________

Instructions

Use this form to report software bugs, documentation errors, or suggested enhancements.
Mail the form to Microsoft.

Category

____ Software Problem

____ Software Enhancement

____ Documentation Problem
(Document # ________)

____ Other

Software Description

Microsoft Product ______________________________

Rev. ______ Registration # ______________________________

Operating System ______________________________

Rev. ______ Supplier ______________________________

Other Software Used ______________________________

Rev. ______ Supplier ______________________________

Hardware Description

Manufacturer ______________ CPU ______________ Memory ______________ KB

Disk Size ______" Density: Sides:

Single ______ Single ______

Double ______ Double ______

Peripherals ______________________________

____

Problem Description

Describe the problem. (Also describe how to reproduce it, and your diagnosis and suggested correction.) Attach a listing if available.

Microsoft Use Only

Tech Support ______________

Routing Code ______________

Report Number ______________

Date Received ______________

Date Resolved ______________

Action Taken: