Microsoft® COBOL for the 8086 Microprocessor and the MS-DOS Operating System

User's Guide

HEWLETT PACKARD

Manual Part No.
45448-90001

Printed in U.S.A.
2/84

Hewlett-Packard
Personal Software Division
3410 Central Expressway
Santa Clara, California 95051

Part Number 45448-90001

Addendum

Enhancements to Version 1.10 COBOL

** You may now use DOS 2.0 pathnames within COBOL programs. When the VALUE OF FILE-ID is a literal, the length is still limited to 16 characters. When the VALUE OF FILE-ID is an identifier, however, the length of the specified filename, including a directory path, may be up to 64 characters in length. Note that pathnames cannot be used with the compiler itself for source, object, or list filenames.

** In previous versions, the size of an item subordinate to an OCCURS clause was limited to a maximum size of 2048 bytes. This version of COBOL removes that limitation.

** The HP 150 offers no support for color. Therefore, the reserved words FOREGROUND-COLOR and BACKGROUND-COLOR, when used with screen displays on the HP 150, have no effect. This allows programs written in MS-COBOL for those Hewlett-Packard Series 100 Personal Computers that support color to also run on the HP 150.

** The compiler switches /T and /C now allow you to specify one of twenty-six drives (A through Z). The previous limit was A through D.

** The compiler now requires that the ACCESS MODE be SEQUENTIAL for a sequentially organized file.

** The compiler now limits numeric items to a maximum of 18 digits.

** The compiler now traps attempts to write either the object or the list file onto the source file, as well as onto each other.

** The compiler now traps attempts to use a VALUE clause in a statement that also contains an OCCURS clause.

** A program that has less than 120 CALLS is guaranteed to generate a valid object module. The exact number of CALLS required to generate an invalid code depends upon additional features, such as numeric edit, alphanumeric edit, ISAM, etc. that were required by the program.

** Multiple screen attributes (such as Half-bright, Reverse-video, Underline, and Blink) are now allowed in a single screen item.

** MS-DOS now checks for CTRL-BREAK at every procedure header. However, when you use CTRL-C to abort an MS-COBOL compiled program, you should press CTRL-C only once. Multiple occurences of CONTROL-C may cause the system to hang.

** Due to the HP 150 screen and keyboard configurations (see Chapter 1 and Appendix A), programs compiled with the MS-COBOL compiler, when run, will:

— home up and clear the display,
— put the console input device in "raw mode" (MS-DOS I/O control for devices) and keycode mode (the HP 150 alpha and graphics I/O system),
— show the application softkeys,
— turn off the touch screen fields, except for the eight function keys (f1-f8),
— turn off display functions,
— turn off memory lock,
— turn off insert character mode,
— unlock the keyboard,
— set the appropriate straps and modes,
— intercept the following keys: f1-f10, RIGHT ARROW, LEFT ARROW, Tab, Shift Tab, Return, Backspace, Delete char, Delete line, and ESC and translate them into COBOL escape codes (see Table A.1 in Appendix A).

When the program finishes or is aborted,

— the console input device is returned to normal mode,
— the touch screen fields are turned off except for the eight function keys (f1-f8),
— the "modes" softkey labels are displayed.

Enhancements to REBUILD Version 1.21

** The REBUILD utility now accepts valid MS-DOS 2.0 pathnames for its files. It also allows you to create a key file from an indexed data file without duplicating the data file. You may use the resulting key and data file as you would any other Indexed file.

** All ISAM files (that is, those files whose ORGANIZATION IS INDEXED), created or modified by versions of COBOL prior to 1.10 or REBUILD prior to 1.21, MUST be run through version 1.21 of the REBUILD program before they can be used by version 1.10 of COBOL. We strongly suggest REBUILDing the file and then using only version 1.10 of COBOL for ISAM file handling.

** REBUILD may now accept input from the command line. This simplifies its use in batch files.

Command Line Mode

You may use the following syntax for the REBUILD command line mode:

```
REBUILD <command line> <RETURN>
```

where <command line> assumes the form:

```
<source filename>,<target filename>,<key description>
```

Each parameter is separated from the next by a comma. You must avoid inserting space characters into the command line.

The format for <key description> is:

```
<key position>:<key length>
```

You may use the default values for any command line argument after <source filename> by placing a semicolon (;) after the last argument you desire to change. If you want the default setting for <target filename> but want to specify <key description>, you must type two commas between the <source filename> and <key description> parameters. (The commas show the place of the missing parameter.)

When you omit either of the last two parameters, COBOL uses the following default values:

<target filename>	The default is <source filename>. This creates a key file with the source filename and the file extension ".KEY". No new data file is created.
<key description>	The default for both <key position> and <key length> is 1 (that is, 1:1).

REBUILD prompts for any information that you fail to provide on the command line.

Example 1. Command Line mode — creating a key file only:

```
REBUILD IXFILE.DAT,,13:52
```

This command line creates the file IXFILE.KEY, overwriting any existing file with that name. You may use IXFILE.KEY with IXFILE.DAT in any MS-COBOL program.

Example 2. Command Line mode — creating both key and data files:

```
REBUILD IXFILE.DAT,NEWIX.DAT,13:52
```

This command line creates the files NEWIX.DAT and NEWIX.KEY. You may use these files in any MS-COBOL program. IXFILE.KEY, if a file by that name exists, remains unchanged.

Interactive Mode

To create only a new key file in interactive mode requires your typing REBUILD then pressing the <Return> key in response to the operating system prompt. The format for the Command Line requires the four parameters (<source filename>, <target filename>, <key position> in the data record, and <key length>). REBUILD issues prompts for the necessary information on the file to be rebuilt and continues to prompt for information until all the parameters have values. You should respond to the prompts for <source filename>, <key position>, and <key length> as you did in previous versions of the REBUILD utility. (See Appendix E for examples.)

The target filename prompt appears as:

> Input the filename of the target data file (should not have the extension of .KEY) or <RETURN> to return to the Key Length prompt.
>
> If the target filename is the same as the source filename, a key file with the source filename and extension ".KEY" will be produced without producing a new data file.

Example 3. Interactive mode — creating a key file only:

```
Input Key Length:          52
Input Key Position:        13
Input Source Filename:    IXFILE.DAT
Input Target Filename:    IXFILE.DAT
```

This interactive sequence creates the file IXFILE.KEY, overwriting any existing IXFILE.KEY file. You may use this file with IXFILE.DAT in any MS-COBOL program.

Example 4. Interactive mode — creating both a key and data file:

```
Input Key Length:          52
Input Key Position:        13
Input Source Filename:    IXFILE.DAT
Input Target Filename:    NEWIX.DAT
```

This interactive sequence creates the files NEWIX.DAT and NEWIX.KEY. You may use these files in any MS-COBOL program. IXFILE.KEY, if a file by that name exists, remains unchanged.

Remember if any previously existing Indexed key files are to be used with programs running under MS-COBOL version 1.10 (or later versions), you must first rebuild the file using version 1.21 or a later version of the REBUILD utility. Version 1.21 of REBUILD allows the key file to be recreated while still using the original data file. You may use Indexed files processed with version 1.21 of REBUILD with all MS-COBOL programs from version 1.0 and later.

See Appendix E - REBUILD: Indexed File Recovery Utility in the *COBOL Compilers User's Guide* for more details on REBUILD. Note that this appendix does not currently contain information on these enhancements.

Table of Contents

Introduction

Chapter 3 - Linking Microsoft COBOL Programs

Chapter 4 - Loading and Executing Microsoft COBOL Programs

Chapter 5 - Batch Command Files

Chapter 6 - Data Input and Output

Chapter 7 - The Interactive Debug Facility

Appendix F - Demonstration Programs

Appendix G - Microsoft COBOL Error Messages

Index

Introduction

Microsoft® COBOL (MS®-COBOL) Compiler is an extensive implementation of the COBOL language for microcomputers. This compiler has been certified with the Federal Compiler Testing Center at the Low Intermediate level of compliance with the ANSI X3.23-1974 Standard. MS-COBOL has many features that are standard for higher levels of validation. It also includes extensions to the Standard that are designed to optimize use of the COBOL language in the microcomputer environment.

Package Contents

Your Microsoft COBOL Compiler package includes:

Two floppy disks (see Chapter 1 of this manual for a list of the files on each disk).

One binder containing the following documentation:

Microsoft COBOL User's Guide

Contains the information that is specific to a particular implementation or operating system. This includes a list of system requirements and a description of the contents of your disks. The User's Guide also provides general instructions on how to compile, link, load, and execute programs on your operating system.

Microsoft COBOL Reference Manual

Contains detailed descriptions of the Microsoft COBOL language. With the exceptions (if any) noted in the *User's Guide*, the information in the *Reference Manual* applies to all implementations of Microsoft COBOL Compiler.

Microsoft COBOL Quick Reference Guide

Outlines the COBOL program structure and gives the formats of individual statements.

System Requirements

Your implementation of MS-COBOL requires:

128K bytes of memory, minimum:

MS-COBOL Compiler requires approximately 40K of memory, and the MS-LINK Linker requires approximately 41K. The exact amount of additional memory needed for the user's programs depends on the programs themselves — the amount of data storage used, the length of the PROCEDURE DIVISION, and the number of optional runtime support modules used.

Note that a single MS-COBOL program module is limited to 64K. The data for an MS-COBOL program is also limited to 64K. A linked MS-COBOL program and program modules, however, may be as large as the available memory.

Two disk drives are recommended, although you can use MS-COBOL with just one disk drive.

If your system does not meet these minimum requirements, ask your computer dealer how to expand it.

NOTE

The MS-COBOL compiler has been configured to the HP 150's terminal characteristics. The runtime executor COBRUN.EXE is configured to store all information that pertains to screen and keyboard features (such as reverse video and blinking, or Tab and Shift Tab). For more information refer to the paragraph on "Using the HP 150 Screen Features" in Chapter 1 and also Appendix A.

Royalty Information

The policy for distribution of parts of the Microsoft COBOL Compiler is as follows:

The COBRUN.EXE runtime module cannot be distributed without first entering into a license agreement with Microsoft Corporation for such distribution. A copy of the license agreement can be readily obtained by writing to Microsoft. In addition, a copyright notice reading "PORTIONS COPYRIGHTED BY MICROSOFT CORPORATION, 1982, 83" must be displayed on the media.

All other software in your Microsoft COBOL Compiler package cannot be duplicated, except for purposes of backing up your software. Other duplication of any of the software in the Microsoft COBOL Compiler package is illegal.

How to Use This Manual

This manual provides information about compiling and running MS-COBOL programs with the MS-COBOL Compiler.

Chapters 1 through 4 provide the information you need to compile, link, load, and execute an MS-COBOL program. Any information that is specific to your MS-COBOL implementation is also contained in these chapters. Chapter 5 tells you how to set up a batch command file to "compile, link, and go."

Chapter 6 explains the four disk file organizations: sequential, line sequential, relative, and indexed. It also describes how to use disk input/output files and other types of files.

Chapter 7 tells you how to use the Interactive Debug Facility to correct program errors at runtime.

Appendix A tells you how to interface your terminal with MS-COBOL (this must be done before compilation); Appendix B explains interprogram communication with the CALL and CHAIN statements. Appendix C shows you how to customize some of the MS-COBOL features.

Appendix D gives an overview of the five phases of the MS-COBOL Compiler. This appendix may be useful if your program generates a "Compiler phase error."

Appendix E describes the REBUILD program, which allows you to recover or restore information in indexed files.

Appendix F describes the demonstration programs that are included with MS-COBOL Compiler. These include a test program for the INSTALL terminal interface, a simple MS-COBOL program, and three programs that demonstrate the MS-COBOL SCREEN capabilities.

Error messages are listed in Appendix G. They are arranged alphabetically within five sections:

1. command input and operating system I-O errors
2. program syntax errors
3. runtime errors
4. program load errors
5. MS-LINK errors

Syntax Notation

The following notation is used throughout this manual for descriptions of the MS-COBOL general format:

[ ]	Square brackets indicate that the enclosed text is optional.
< >	Angle brackets indicate user entered data. When the angle brackets enclose lowercase text, the user must type in an entry defined by the text; for example, <filename> indicates that the user must enter the name of a file. When the angle brackets enclose uppercase text, the user must press the key named by the text; for example, <RETURN> means press the <RETURN> key.
{ }	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.
	Braces also delimit the portion of a statement that is referred to by an ellipsis.
|	Vertical bars separate the choices within braces. At least one of the entries 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. They are also used for words that the computer displays.

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

Learning More About COBOL

If you are new to COBOL programming, you will probably want to learn more about writing programs before using the MS-COBOL Compiler. The following texts are COBOL tutorials, written for the novice programmer:

Abel, Peter. COBOL Programming: *A Structured Approach.* Reston, Virginia: Reston Publishing Company, 1980.

McCracken, Daniel D. *A Simplified Guide to Structured COBOL Programming.* New York: John Wiley & Sons, Inc., 1976.

Parkin, Andrew. *COBOL for Students.* Beaverton, Oregon: Edward Arnold, Ltd., 1978.

Chapter 1

GETTING STARTED

The purpose of this *User's Guide* is to help you get a Microsoft COBOL program up and running on your computer. To do this, you need to perform some one-time tasks, and you need an understanding of the major steps involved in using MS-COBOL. This chapter begins by listing the contents of your disks and by telling you how to perform disk backup. Then it presents an overview of the compilation process and a sample program development session.

Your Distribution Disks

You receive two MS-COBOL distribution disks that are organized in the following manner:

Disk 1 files

The MS-COBOL Compiler, the Runtime System, and the Linker

`COBOL.COM`	— the main compiler program
`COBOL1.OVR`	— overlay 1
`COBOL2.OVR`	— overlay 2
`COBOL3.OVR`	— overlay 3
`COBOL4.OVR`	— overlay 4
`COBOL1.LIB`	— the runtime library of optional routines
`COBOL2.LIB`	— the runtime library containing the routines necessary for loading COBRUN.EXE
`COBRUN.EXE`	— the common runtime executor
`COBDBG.OBJ`	— the interactive debug facility
`LINK.EXE`	— the MS-DOS linker

Disk 2 files

Demonstration Programs

`CRTEST.COB`	— a test program for the terminal interface, as customized for the HP 150
`CENTER.COB`	— a test program for the MS-COBOL Compiler and runtime system

The MS-COBOL Demonstration System

`DEMO.COB`	— a program to demonstrate the MS-COBOL screen section, to call the subprogram BUILD, and to chain to the program UPDATE
`DEMO.CPY`	— a file used by the COPY verb in DEMO.COB
`BUILD.COB`	— a program to create an indexed (ISAM) file of names, addresses, and telephone numbers
`UPDATE.COB`	— a program to list or update the ISAM file created by BUILD
`DEMO.EXE`	— an executable version of DEMO already linked with BUILD
`UPDATE.EXE`	— an executable version of UPDATE, already linked
`DEMO_01.OVL`	— an overlay file generated by linking DEMO
`CLDEMO.BAT`	— a batch command file for compiling and linking the Demonstration System

The Rebuild Utility

`REBUILD.EXE`	— the utility for recovering damaged indexed files

Disk Backup

The first thing you should do when you receive your disk(s) is make copies to work with, saving the original disk(s) as backups. Do this by using the COPYDISK utility supplied on your operating system disk.

Having made backup copies, check your copy of the compiler and runtime system by compiling, linking, and executing the test program CENTER.COB. To do this, refer to the sample program development session in Chapter 1, "Sample Session."

Using the HP 150 Screen Features

As the runtime executor COBRUN.EXE contains information on the HP 150's screen functions, you may use any of the screen-specific features without doing any installation procedure. To access the screen functions, simply use the ACCEPT and DISPLAY statements with the appropriate format for an elementary or a group screen item. (See the DATA DIVISION's SCREEN SECTION in the *COBOL Reference Manual* and also the addendum in the front page of the *User's Guide*.)

The Compilation Process

The three major steps in compiling and executing an MS-COBOL program are:

1. compiling
2. linking
3. loading and executing

Consult Figure 1.1 as you read the following descriptions of these steps.

1. Compiling

   The MS-COBOL Compiler consists of the main compiler program (COBOL.COM) and four phases or overlays (COBOL1.OVR through COBOL4.OVR). The routines contained in the compiler analyze your COBOL program and produce an object code file. This file will have a filename extension of .OBJ.

   Compilation is performed in two passes. The first pass creates an intermediate version of the program, which is stored in a binary work file called COBIBF.TMP. The second pass creates the final version of the object code.

2. Linking

   The object code produced by the compiler is not executable machine code. The Microsoft Linker (MS-LINK) is responsible for producing the machine executable code, which will be placed in a file with an .EXE extension.

   MS-LINK performs the following tasks:

   a. combines separately produced object files
   b. searches library files for definitions of unresolved external references
   c. resolves external cross-references
   d. produces a printable listing of symbols
   e. produces the executable program

3. Loading and executing

   The runtime system (COBRUN.EXE) "runs" the executable program.

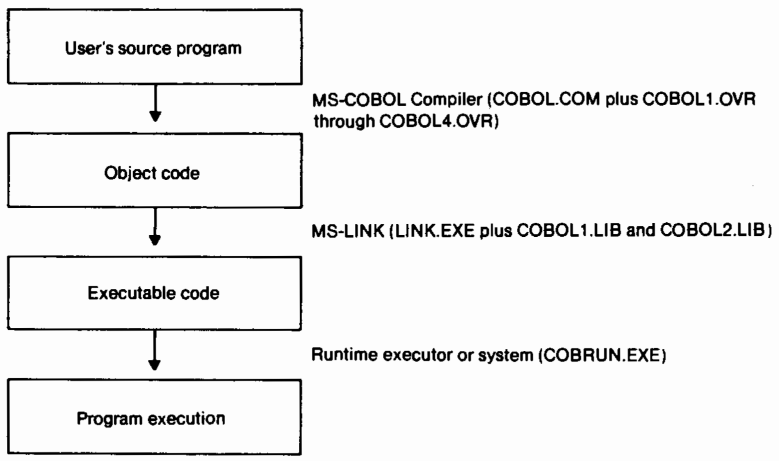

User's source program
MS-COBOL Compiler (COBOL.COM plus COBOL1.OVR through COBOL4.OVR)
Object code
MS-LINK (LINK.EXE plus COBOL1.LIB and COBOL2.LIB)
Executable code
Runtime executor or system (COBRUN.EXE)
Program execution

Figure 1.1. Major Steps in Compiling and Executing an MS-COBOL Program

Sample Session

The compilation, linking, and loading/execution of an MS-COBOL program are described in detail in Chapters 2, 3, and 4 of this manual. To give you an overview of the MS-COBOL system, however, the following sample session is provided. We recommend that you work through the sample session and then read Chapters 2 through 4 of this *User's Guide* before beginning to compile your own programs.

The examples in this sample session are designed for systems with two disk drives of 160K to 240K capacity. The program development steps themselves, however, are appropriate for all implementations of MS-COBOL.

1. **Organize Your Disks**

   Organize the files on your disks to minimize disk-swapping and "Disk full" errors during program development. Usually MS-COBOL program development will require three working disks: one for your source and object programs; one for the Compiler and the text editor; and the other for MS-LINK, the runtime executor, the runtime libraries, and any other necessary utilities. For example, your three working disks might contain the following files:

   a. Program disk

      ```
      COMMAND.COM
      <your source files>
      <intermediate files>
      <object files>
      <executable files>
      ```

      Intermediate files, object files, and executable files are generated during the compilation and linking process.

   b. Compiler disk

      ```
      COMMAND.COM
      COBOL.COM
      COBOL1.OVR
      COBOL2.OVR
      COBOL3.OVR
      COBOL4.OVR
      <text editor>
      <miscellaneous utilities>
      ```

      The <text editor> may be any editor that will fit in the remaining disk space. The <miscellaneous utilities> are MS-DOS utilities to set up the printer, sort the directory, clear the screen, etc.

   c. Utility disk

      ```
      COBDBG.OBJ
      LINK.EXE
      COBOL1.LIB
      COBOL2.LIB
      COBRUN.EXE
      ```

Your "program" disk contains your MS-COBOL source program, object, and executable files.

You should use your copies of the distribution disk to make a copy of the compiler disk. You do this by using the COPYDISK utility that comes with the HP 150 operating system disk. (See the discussion on "Disk Backup" in the first part of this chapter.)

During development, the program disk will be kept in drive B, and either the compiler disk or the utility disk will be in drive A, depending on which disk is required at the time.

Drive B should be selected as the default drive (the one where new files are placed unless specified otherwise in the command). This arrangement simplifies access to the program files by placing them all on the same disk. Use of the programs on the utility or compiler disk will then require an explicit drive specification (e.g., A:<text editor executable file> or A:COBOL).

2. Create the Source Program

In this sample session, we'll use the sample program CENTER.COB for the source program. CENTER asks you to enter a line of text and lets you choose whether to center the text or align it at the left or right margin. (CENTER can be easily converted into a subroutine for your own use later.)

Transfer the sample MS-COBOL program CENTER.COB from your copies of the MS-COBOL to your program disk. Then perform part "a" in the following list. You will not have to do parts "b" and "c."

However, you can use your text editor to create your own source program, also called CENTER.COB, instead of using the CENTER.COB program provided on your copies of the distribution disk. If you choose to create your own file, do parts "a," "b," and "c" in the following list.

a. After booting the system as usual, place your compiler disk in drive A. Then place the program disk in drive B and select B as the default drive by typing:

```
B:
```

b. Type the command

```
A:<text editor executable file> CENTER.COB
```

to run the editor program so you can write your MS-COBOL program.

c. When you have finished writing the program, use your editor command to keep the file CENTER.COB on your program disk and exit to the operating system.

3. Check Program Syntax With Trial Compilation

Before you go on, you can check your program for syntax errors with a "quick" compilation. This is done by compiling the program and displaying the error listing on the screen. No object or listing files are created, so compilation is faster than usual.

To compile CENTER.COB and display a list of errors on the terminal, use the following command:

```
A:COBOL CENTER,NUL;
```

(See Chapter 2 for a discussion of MS-COBOL Compiler commands.)

If you get errors during the trial compilation, go back to Step 2 and correct the source file (using the text editor in the compiler disk on drive A). See Appendix G for a list of error messages and explanations. When the trial compilation is completed without errors, you are ready to proceed to Step 4.

4. Compile the Source Program

Now the program is ready to be compiled. Compilation produces the object file. The compiler looks for the overlay phases (COBOL1.OVR - COBOL4.OVR) first on the default drive (drive B in this example) and then on drive A. With the disks arranged as in our example, the overlay phases will be found on drive A.

To compile the program so that an object file (named CENTER.OBJ) is produced, enter one of the following commands:

`A:COBOL CENTER;`	produces just the object file
`A:COBOL CENTER,,PRN`	produces an object file and printed listing
`A:COBOL CENTER,,CENTER`	produces an object file and a list file (named CENTER.LST)

When compilation is successfully completed, the message "No Errors or Warnings" is displayed, and the compiler exits to the operating system.

5. **Link and Save the Executable Program**

Put the utility disk in drive A. Note that the linker expects to find the MS-COBOL common runtime libraries (COBOL1.LIB and COBOL2.LIB from the utility disk) on the default drive (drive B in this case). If the libraries are not there, you will be prompted to specify the drive on which they are located, unless you instruct MS-LINK to look elsewhere. In this example, we will do just that by specifying drive A in the following link command.

Now type the command:

```
   A:LINK CENTER,,,A:;
```

This command links the object file with the runtime system, producing the executable file. The "A:" at the end of the command line tells MS-LINK to look on drive A for the MS-COBOL libraries. (See Chapter 3 for a discussion of MS-LINK commands.) The executable file (called CENTER.EXE) is saved on the disk in drive B.

Your program disk now contains the following files: CENTER.COB, CENTER.OBJ, CENTER.EXE, CENTER.DBG, CENTER.MAP, and, if you requested a list file, CENTER.LST. The DBG and MAP files will not be used in this session, and may be deleted.

6. **Load and Execute the Program**

To run a program, you need the executable file (CENTER.EXE) and the common runtime executor (COBRUN.EXE). COBRUN.EXE may be in either drive. The compiler will search for it first on the default drive, then on drive A.

In this example, CENTER.EXE is on the program disk and COBRUN is on the utility disk on drive A. Since we are keeping the program disk in drive B, and drive B is selected as the default drive, type just the name of the executable file (the .EXE is not required):

```
CENTER
```

Even though you've been very careful to remove all compile time errors, you may still get runtime errors when the program is run. Error messages are described in Appendix G of this manual. If you get runtime errors, return to Step 2 and edit the program to correct the errors.

COMPILING MICROSOFT COBOL PROGRAMS

As in Chapter 1, the sample commands in this chapter assume that: the Microsoft COBOL Compiler disk is in drive A, your program disk is in drive B, and drive B has been selected as the default drive.

In the following examples, the file CENTER.DBG will be produced in addition to the files specified. Files with a .DBG extension are used by the Interactive Debug Facility. (See Chapter 7 for more information on the Interactive Debug Facility.) Use the /D switch to suppress DBG files. However, the DBG file assists in debugging, and we therefore recommend that you produce it during program development.

Invoking the Compiler

The MS-COBOL Compiler may be invoked in one of the two ways listed below. Note that the following discussions on "Compiler Responses" and "Partial Command Strings" apply to both these methods. Therefore, read these paragraphs before you begin to compile your own programs.

1. You may invoke the compiler by entering the command

   ```
   A:COBOL
   ```

   The drive specification is necessary because the compiler is not in the default drive.

   Then reply to the following prompts. (Filenames are discussed under "Compiler Responses.")

   ```
   Source filename [.COB]:
   ```

Name of your source program. A filename must be specified. If no extension is specified, .COB will be appended by default.

```
Object filename [<source filename>.OBJ]:
```

Name of the object file to be created. The source filename is the default filename. The extension .OBJ is the default extension.

```
Source listing [NUL.LST]:
```

Name of the file to which the program listing is to be written.

If a filename is entered, its default extension is .LST. If a filename is not entered, the default is NUL (no list). See "The Source Listing File" paragraph for further discussion of the list file.

Example: The following series of responses compiles the source file CENTER.COB, producing the object file CENTER.OBJ and a listing file CENTER.LST on the default drive:

```
A:COBOL
Source filename                [.COB]: CENTER
Object filename [CENTER.OBJ]:  <RETURN>
Source listing [NUL.LST]:      CENTER
```

2. The compiler can also be invoked by entering

```
A:COBOL <command string>
```

where the command string contains

```
<source filename>,<object filename>
,<source listing>
```

as explained above and in "Compiler Responses."

The separator character is the comma (,). No spaces are allowed.

When compilation has finished, you will be notified of any errors. If errors exist, you must locate and correct them in the source program and recompile the program before linking it. If the compiler detected no errors, you will be told

```
No Errors or Warnings
```

and you may proceed with linking (as described in Chapter 3).

Compiler Responses

When you use either of the above methods to invoke the compiler, each of your responses can be the name of a disk file and/or system device. The format is:

```
<device><filename><extension>
```

<device> is the name of a system device. This can be a disk drive, terminal, line printer, or other device supported by the operating system. If the device is a disk drive, the filename must also be given, unless a default filename is available (see final example under "Partial Command Strings"). If the device is not a disk drive, only the device name is required. The device may be followed by a colon (:) for readability (it is required only for disk drives). MS-COBOL recognizes the following device names:

`NUL`	Do not create
`CON or USER`	Display on terminal
`A: or B: ...`	Disk drive (colon required)
`PRN or LPT1`	Printer
`LPT2 ...`	Additional printer(s)
`AUX or COM1`	RS232

<filename> is the name of the file on disk. If the filename is specified without a device, the default disk drive is assumed as the device. Maximum length of the filename is eight characters.

<extension> is a period (.) followed by a three-character suffix to the filename. If an extension is not specified, the following defaults are assumed:

`.COB`	for the source program file
`.OBJ`	for the object file
`.LST`	for the list file

Partial Command Strings

You may also enter a partial command string when invoking a compiler. Note that the default *object* filename may be specified by entering only the comma which normally follows the filename. Also note that if the comma which follows the object filename is entered, the source listing filename defaults to the source filename. You will be prompted for any files not specified in the command string. For example, the command

```
A:COBOL CENTER,,
```

would (1) prompt you for the source listing filename (with the default name CENTER.LST), (2) compile the source from CENTER.COB, and (3) produce the object file CENTER.OBJ.

Each prompt displays its default, which you may accept by pressing <RETURN> or override by entering another filename or device name.

If you enter an incomplete command string followed by a semicolon (;), default entries will be assumed for the unspecified files.

The following examples assume the compiler is in drive A and that drive B has been selected as the default drive:

1. `A:COBOL CENTER;`

   Compiles the source from CENTER.COB and produces the object file CENTER.OBJ. No listing file is produced.

2. `A:COBOL CENTER,;`

   Does exactly the same thing as the previous example.

3. `A:COBOL CENTER,,;`

   Compiles the source from CENTER.COB and produces the files CENTER.OBJ and CENTER.LST. (The second comma (,) tells the compiler to use the source filename as the default list filename.)

4. `A:COBOL CENTER,,CON`

   Compiles the source from CENTER.COB and places the source listing file on the terminal. The object program is CENTER.OBJ.

5. `A:COBOL CENTER,CENTOBJ,PRN`

   Compiles the source from CENTER.COB, sends the list file to the printer, and places the object in CENTOBJ.OBJ.

6. `A:COBOL A:CENTER,CENTOBJ,A:;`

   Compiles CENTER.COB from disk A, places the object into CENTOBJ.OBJ on drive B, and places the listing into CENTER.LST on drive A.

Using Compiler Switches

You can add one or more switches to the compiler command string or at the end of any interactive response. A switch is indicated by a slash (/). The switches and their effects are described below.

The format for a command string with switch(es) is:

```
<drive>:COBOL <command string>/<switch(es)>
```

Switches

`/C` Ordinarily, the compiler looks for the four overlay files (COBOL1.OVR through COBOL4.OVR) first on the default drive, then on drive A. To override the default drive, use the /C switch with the letter of the drive you want. (The colon is not required in the switch.)

Example: `A:COBOL CENTER,,/CB`

In this example, the compiler looks for the overlay files on drive B.

`/T` The compiler puts its intermediate file COBIBF.TMP on the default drive unless you use the /T switch followed by the desired drive designation. The disk in the drive you specify must not be write protected.

This option is particularly helpful for compiling very large programs on systems with more than two drives (see the paragraph on "Compiling Large Programs").

Example: `A:COBOL CENTER,,A:CENTERLIST/TA`

In this example, the intermediate file is placed on drive A. (The colon is not required in the switch.)

`/P` Each /P allocates an extra 100 bytes of stack space for the compiler's use. Use /P if stack overflow errors occur during compilation.

Example: `A:COBOL CENTER/P/P/P;`

In this example, 300 extra bytes of stack space are allocated.

Switches (continued)

`/D` This switch suppresses both generation of the debug information file (with a .DBG extension) and source line numbers, which are normally placed in the object file. The result is PROCEDURE DIVISION code that is about 16 percent shorter. However, when this switch is used, the runtime system will not be able to note the line number at which an error occurs. (See Chapter 7 for a discussion of the debug information file.)

Example: `A:COBOL CENTER/D;`

In this example, the object file will not contain source line numbers and CENTER.DBG will not be produced.

`/Fn` Fn (FIPS) flagging lets you tell the compiler to output a warning for each COBOL statement above the Federal Information Processing Standard level(n). The n must be a digit from 0 through 4 (4 is the default):

0	Flag everything above low level.
1	Flag everything above low intermediate level.
2	Flag everything above high intermediate level.
3	Flag everything above high level.
4	No flagging.

Example: `A:COBOL CENTER/F1;`

In this example, the compiler will display a warning for each COBOL statement above low intermediate level. If you create a source listing file, the warning will be included with the error messages.

The Source Listing File

The source listing file is a line-by-line account of the source file(s) with page headings and error messages. Each source line is preceded by a four-digit decimal number. This number will be referenced by any error messages pertaining to that source line.

Files which are included in the compilation via COPY statements in the source file are also included in the listing.

Compiler error messages are shown at the end of the listing file (as well as being displayed on the terminal). See Appendix G for a listing and explanation of error messages.

Compiling Large Programs

Occasionally, an MS-COBOL program may be too large to compile in the available memory space or may exhaust the available disk space. There are several ways to take care of this problem:

1. Use the /D switch in your command string (see "Using Compiler Switches") to prevent generation of a debug information file and to suppress generation of line numbers in the object file.
2. Use the /T switch in your command string (see "Using Compiler Switches") to place the intermediate file (COBIBF.TMP) on a separate disk.
3. Place the MS-COBOL Compiler (COBOL.COM) and its overlays (COBOL1.OVR - COBOL4.OVR) on two separate disks. Then load each portion into memory only as it is needed:
   a. With the program disk in drive B, place the COBOL.COM disk in drive A and invoke the compiler by typing "A:COBOL".
   b. When you receive the first prompt "Source filename [.COB]:", take out the COBOL.COM disk and place the overlay disk in drive A. Then respond to the compiler prompts as usual.

   This method allows the space normally used by COBOL.COM to be available for the intermediate file COBIBF.TMP.
4. Break the program into several program modules. These modules can be separately compiled and then combined into one program by the linker. See Appendix B, "Interprogram Communication," for information on using program modules.

5. Break the large program into several smaller programs which are chained. These programs are separately compiled and linked. See Appendix B, "Interprogram Communication," for information on chaining programs.

NOTE

If you want to check the contents of your disk to make sure that COBIBF.TMP has been deleted after compilation is completed, use the DIR operating system command. Then, to make sure the space has been released, use the CHKDSK program supplied with your operating system. CHKDSK reclaims available space from unclosed files and tells you the total amount of available space on the disk.

Chapter 3

LINKING MICROSOFT COBOL PROGRAMS

As in previous chapters, this discussion assumes that: the utility disk is in drive A, the program disk is in drive B, and drive B has been selected as the default drive.

The Microsoft linker (MS-LINK) converts the compiled object version of your program (the object file) into a version that is executable (the run file). To do so, it searches the disk in the default drive for the MS-COBOL runtime libraries COBOL1.LIB and COBOL2.LIB, which make up part of the common runtime system (described in Chapter 4). COBOL1.LIB is a library of optional routines that may be required for running the program, and COBOL2.LIB contains the routines that are always necessary for running the program. The routines you need are then linked to the object version of your MS-COBOL program. The routines you need depend on which MS-COBOL language features you used in the program and program modules.

MS-LINK can also be used to combine separately compiled program modules into one program. The modules may be specified individually or extracted from a library. They may be written in MS-COBOL or in Microsoft Macro Assembler language (MS-Macro Assembler). See the paragraph on "Linking Program Modules" for the necessary details.

Using MS-LINK

Files that are to be linked or which will contain linker output can be specified in one of three ways:

1. interactively
2. as part of the command line
3. as a command file

To invoke the linker, use one of these procedures which are described in more detail in the following pages.

To specify files interactively, enter

```
A:LINK
```

(The device specification is necessary because MS-LINK is not in the default drive.) Then reply to the following prompts:

```
Object Modules[.OBJ]:
```

Name(s) of object file(s). If no extension is specified, .OBJ will be used. If multiple object files are linked, they must be separated by a plus (+).

Files that are to be linked must be in object format. (If they were compiled with MS-COBOL or generated by MS-Macro Assembler, they will already be in object format.)

```
Run File[<object filename>.EXE]:
```

Name of file to contain executable code. The object filename is the default filename. The extension .EXE cannot be overridden.

```
List File[NUL.MAP]:
```

Name of list file. Defaults work much the same way as in the compiler. The default is no list file, unless the run file is followed by a comma (see discussion of partial command strings, below). If the run file is followed by a comma, the default list filename is the object filename, with the default extension .MAP.

```
Libraries[.LIB]:
```

"Libraries" refers to the runtime routines that MS-COBOL may need to run your program. All these routines are included in COBOL1.LIB and COBOL2.LIB.

Normally, you only have to press <RETURN> in response to this prompt. The names of the libraries are supplied to the linker by the MS-COBOL object file. If you wish however, you may specify your own libraries (see your MS-DOS manual), which will be searched before the MS-COBOL libraries.

MS-LINK assumes that the MS-COBOL libraries are in the default drive. If they are not in the default drive, you must enter a drive specification, regardless of which drive has been selected as the default drive.

In all of our examples, the libraries are on drive A and not the default drive. Therefore, you need to indicate the drive specification for the libraries. If you do not, MS-LINK will prompt you for the drive on which the libraries are located.

Filenames are specified in the same way as they are for the compiler (see Chapter 2), except that the default extension is always .EXE for the run file produced by the linker.

Example: The following series of responses links the files CENTER.OBJ and MYOBJ.OBJ and searches your library MYLIB1.LIB before searching COBOL1.LIB and COBOL2.LIB. The linker produces the executable file MYRUN.EXE and the source listing file MYLIST.MAP.

```
A:LINK
Object Modules[.OBJ]:     CENTER+MYOBJ
Run File[CENTER.EXE]:     MYRUN
List File[NUL.MAP]:       MYLIST
Libraries[.LIB]:          MYLIB1+
                          A:COBOL1+A:COBOL2
```

To use a command string, enter

```
A:LINK <command string>
```

where the command string contains

```
<objfile(s)>,<runfile>,<listfile>,<libfile(s)>
```

as defined in the preceding example.

An object filename must be specified. For the other files, a default filename may be selected in the command string by entering only the comma which would normally follow the filename (see the following examples).

As with the MS-COBOL Compiler, you may enter a partial command string or the entire string. If you specify an entry for all four files, or if an incomplete command string ends with a semicolon (;), linking will proceed without further prompting. Otherwise, the linker prompts for the remaining unspecified files. Each prompt displays its default, which you may either accept (by pressing <RETURN>) or override (by entering another filename or device name).

Examples: (In these examples, the utility disk is in drive A, default drive is B.) Since the MS-COBOL libraries are in drive A, and the default drive is drive B, MS-LINK will not find the libraries unless you specify the drive for the libraries or respond with "A" to the MS-LINK prompts. In these examples, we have specified the library on drive A, unless indicated otherwise.

1. `A:LINK CENTER;`

   Links CENTER.OBJ and puts the runfile into CENTER.EXE. No list file is produced. If CENTER.OBJ was produced by the MS-COBOL Compiler, MS-LINK prompts for the drive on which COBOL1.LIB and COBOL2.LIB are found. Type "A" in response to the prompt.

2. `A:LINK CENTER,,,A:;`

   Same as first example, except that a listing is produced in CENTER.MAP. (The second comma (,) indicates that the object filename is to be used as the default list filename.) The "A:" at the end of the command line tells MS-LINK to find the MS-COBOL libraries on drive A instead of the default drive.

3. `A:LINK CENTER+SUBFILE1+SUBFILE2,,,A:;`

   Same as previous example, except that SUBFILE1.OBJ and SUBFILE2.OBJ will be linked with CENTER.

You can also set up one or more command files which contain responses to the linker prompts. Command files are created by the user. They are especially useful when you are linking a number of object modules more than once (during debugging, for example), or when you are developing variations of a program. See Chapter 5 of this manual or the MS-DOS manual.

To specify this option on the command line, use the command:

```
A:LINK @<filename>
```

<filename> is the name of your command file. You must include the drive if the file is not on the default drive. You may also specify a file extension.

Example: `A:LINK @RESFIL.CMD`

After the command line is entered, the linker starts. If the linker needs more memory space to link your program than is in the computer, it will create a file called VM.TMP on the disk in the default drive and will display a message to that effect. ***Do not remove this disk during linking.*** If the additional space in VM.TMP is used up, or if the disk containing VM.TMP is removed before linking is completed, the linker will abort.

When the linker has finished, VM.TMP will be erased from the disk, and any errors that occur during linking will be displayed. The run file will be stored (with the extension .EXE) on the disk in the default drive or on the specified drive.

NOTE

If you want to check the contents of your disk to make sure that VM.TMP has been deleted after the linker aborts, use the DIR operating system command. Then, to make sure the space has been released, use the CHKDSK program supplied with your operating system. CHKDSK will reclaim available space from unclosed files and tell you the total amount of available space on the disk.

Linking Independent Segments (Overlays)

The MS-COBOL segmentation facility lets you run programs that are larger than the computer's central memory. Segmented programs have overlays that are referenced by MS-COBOL section numbers greater than 49 (see the chapter on "Segmentation," in the *Microsoft COBOL Reference Manual*). Each section is an independent segment.

No special commands are required for linking a segmented program. The linker creates a file for each independent segment of the program, with the filenames in the format:

```
PROGIDnn.OVL
```

PROGID is the PROGRAM-ID which you defined in the IDENTIFICATION DIVISION. If the PROGRAM-ID is less than six characters, MS-COBOL extends it to six characters by adding underlines (_) to the end.

nn is a two-digit hexadecimal number that is computed by subtracting 49 (decimal) from the program segment number (decimal).

Example: If the PROGRAM-ID is "SAMPLE" and the program contains segment number 99 (decimal), an overlay segment will be produced with the name SAMPLE32.OVL.

Linking Program Modules

If you have developed your program as separately compiled program modules, the linker can combine the modules into one program.

Before linking, compile or assemble all modules so that you have an object version of each. Then start the linker, specifying in the command string each module you want to link.

Example: `A:LINK CENTER+SUBFILE1+SUBFILE2,,,A:;`

See Appendix B, "Interprogram Communication," for more information about linking program modules.

Linking Large Programs

This discussion assumes that your files are arranged on three disks as in the "Sample Session" in Chapter 1.

If your system's disk space will not hold all the object files, required libraries, the run file, the linker, and the list (.MAP) file, you will need to separate the files. One or a combination of the following space-saving procedures should take care of this problem.

1. Do not request a list file (.MAP) — i.e., accept the no list default (.NUL).
2. Send the list file (.MAP) to the terminal (CON) or printer (PRN).
3. Transfer the runtime executor (COBRUN.EXE) from the program disk to a separate disk. Perform the link. Then copy the run file to the disk containing COBRUN.EXE, insert the disk (in either drive), and begin execution as usual.
4. Transfer the runtime libraries COBOL1.LIB and COBOL2.LIB from the utility disk to a separate disk. Invoke the linker with the LINK command and no command string. After the linker has been loaded, place the disk containing the runtime libraries in drive A and the program disk in drive B. Then answer the linker prompts, specifying that the run file should go to drive A.
5. Break the program into several programs which are chained. Compile and link each program separately. Note that the common runtime system works very efficiently with CHAIN; it only needs to be loaded once, rather than once for each program in the chain. See Appendix B, "Interprogram Communication," for more information on chaining.
6. Break the program into program modules connected by CALL statements. Compile the modules separately and link them together using the linker. This procedure is similar to CHAIN, except that the called program contains a return statement. See Appendix B, "Interprogram Communication," for instructions on linking program modules.

NOTE

If you want to check the contents of your disk to make sure that VM.TMP has been properly deleted after the linker aborts, use the DIR operating system command. Then, to make sure the space has been released, use the CHKDSK program supplied with your operating system. CHKDSK will reclaim available space from unclosed files and tell you the total amount of available space on the disk.

Chapter 4

LOADING AND EXECUTING MICROSOFT COBOL PROGRAMS

After your Microsoft COBOL program has been compiled and linked successfully, the final step is loading and execution. These functions are performed by specifying the name of the executable file to the operating system, as explained below.

Your runtime executor (COBRUN.EXE) is loaded automatically at the beginning of execution. When you begin execution, COBRUN.EXE must be in either the default drive or drive A.

To run your program, just enter the name of your run file, without the .EXE filename extension. For example, type:

```
CENTER
```

Execution of CENTER.EXE should begin immediately.

Chapter 5

BATCH COMMAND FILES

The MS-DOS operating system allows you to create a batch file for executing a series of commands. This file must have the extension .BAT. It should be kept on either the program disk or the utility disk.

As shown in the example below, the batch file may contain symbols that refer to parameters in its invocation line. The symbol %1 refers to the first parameter on the line, %2 to the second parameter, etc. The limit is %9. In the example which follows, %1 refers to the parameter <sourcefile>.

The batch file may also pause, display a prompt (defined by the user), and wait for the user or operator to continue. The PAUSE command, followd by the user-defined text of the prompt, performs this function.

If your program is already debugged and you are making only minor changes to it, you can speed up the compilation process by creating a batch file that issues the compile, link, and run commands.

For example, use the text editor to create the batch file CLGO.BAT (named for "compile, link, and go"). The text of the file might be:

```
A:COBOL %1,,;
PAUSE . . .Insert runtime libraries disk in drive A:
A:LINK %1,,,A:;
%1
```

To execute this file, type

```
CLGO <sourcefile>
```

<sourcefile> is the name of the source program you want to compile, link, and run. The first line of the batch file compiles the program; the second causes a pause followed by a prompt telling you to insert the runtime libraries disk; the third line links the object file; and the fourth runs the executable file.

NOTE

A BAT file is only executed if there is neither a COM file or EXE file with the same name.

For more information about batch command files, see your MS-DOS manual.

Chapter 6

DATA INPUT AND OUTPUT

A Microsoft COBOL program can read or write data to files on disk or to other MS-DOS devices. The instructions for creating and using these files are entered as part of the MS-COBOL source program. This section explains disk files and other types of files, and tells you how to use them with your MS-COBOL programs. See the *Microsoft COBOL Reference Manual* for more information.

Using Disk Files

To specify that a disk file is to be used in a program, include the ASSIGN TO DISK clause in the FILE-CONTROL paragraph of the ENVIRONMENT DIVISION.

The filename of the disk file must be declared in the VALUE OF FILE-ID clause in an FD paragraph, in the FILE SECTION of the DATA DIVISION. The FD paragraph must also include the clause LABEL RECORDS ARE STANDARD. BLOCK clauses are checked for syntax, but they have no effect on any file type. The FILE-ID clause should not be specified with a name that is an MS-DOS device name. (See the paragraph on "Using MS-DOS and Nondisk Files" for a list of MS-DOS device names.) Giving the FILE-ID clause an MS-DOS device name would cause the file to appear on the specified MS-DOS device rather than on a disk drive.

There are four types of disk file organization:

```
SEQUENTIAL
LINE SEQUENTIAL
RELATIVE
INDEXED
```

When an MS-COBOL program reads from or writes to a disk file, the ORGANIZATION clause in the FILE-CONTROL paragraph of the program's ENVIRONMENT DIVISION must specify the file organization of the disk file, unless it is SEQUENTIAL. Disk files are assumed to be SEQUENTIAL unless they are declared otherwise.

Note also that only LINE SEQUENTIAL files can be created with an editor. All others must be created by an MS-COBOL program or assembly language program. See the *Microsoft COBOL Reference Manual* or one of the tutorials recommended in "Learning More About COBOL," in the introduction to this manual, for more information about creating disk files.

The four types of disk files are described below. (All formats are subject to change without notice.)

1. SEQUENTIAL files have a two-byte count of the record length followed by the actual record, for as many records as are in the file.

2. In LINE SEQUENTIAL files, the record is followed by a carriage return/line feed delimiter, for as many records as are in the file. No COMP-0 or COMP-3 fields 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 SEQUENTIAL and LINE SEQUENTIAL organizations pad any remaining space in the last physical block with one or two Control-Z characters (indicating end-of-file), followed by binary zeros. To make maximum use of disk space, records are packed together with no unnecessary bytes in between.

Warning

Files created by line editors and non-COBOL programs are often in LINE SEQUENTIAL format. If you wish to use such a file as input to an MS-COBOL program, you must include the ORGANIZATION IS LINE SEQUENTIAL clause in its FILE-CONTROL paragraph. If the clause is not included, MS-COBOL assumes the file is in SEQUENTIAL format, and stops with a runtime error when the LINE SEQUENTIAL file is input.

---

3. RELATIVE files always have fixed length records of the size of the largest record defined for the file. Since no delimiter is needed, none is provided. Deleted records are filled with hex value "00". Additionally, six bytes are reserved at the beginning of the file to contain system bookkeeping information.

4. Each INDEXED file declared in an MS-COBOL program will generate two disk files: a key file and a data file. 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.*

   A 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

   The key file will have 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.

   Each Data Set Control Block and Key Set Control Block contains, in the fourth byte, a "damaged" flag which notifies you when the last file use was not terminated properly. The runtime executor sets these flags to nonzero values when the file is opened for updating and restores them to zero when the file is closed.

   The "data file" consists of data records. Each data record is preceded by a two-byte field and a one-byte "reference count" that indicates whether a record has been deleted. The data file is terminated by a control record with a length field containing a 2, followed by two bytes of high values.

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

Using MS-DOS and Nondisk Files

Files that will only be output need not be placed on a disk, but should be considered as a stream of characters going to a printer or other device. No permanent file is created. Records should be defined as the fields to appear on the output device. No extra characters are needed in the record for carriage control. Carriage return, line feed, and form feed are sent to the output device between lines. Note, however, that blank characters (spaces) on the end of a print line are truncated to make printing faster.

To send an output file to the printer, use the SELECT <filename> ASSIGN TO PRINTER clause. Then, in an associated FD, specify the clause LABEL RECORD IS OMITTED. Do not specify the VALUE OF FILE-ID clause.

MS-DOS provides special device names for character devices. Data may be sent to or read from the following devices:

`CON or USER`	display on terminal
`AUX or COM1`	serial port (RS232)
`PRN or LPT1`	printer
`LPT2 ...`	additional printer(s)

If you assign these names to the VALUE OF FILE-ID clause, MS-COBOL treats the files as if they were disk files (see the preceding discussion on "Using Disk Files"). That is, you assign the files to disk with the SELECT clause, but the operating system uses the designated device instead of a disk drive.

Chapter 7

THE INTERACTIVE DEBUG FACILITY

The MS-COBOL Interactive Debug Facility allows you to control the execution of a program and to examine or change data items in an MS-COBOL program. When a program is compiled, a "debug information file" is created along with the object file. The information file contains line numbers and data-names from the DATA DIVISION and PROCEDURE DIVISION of your MS-COBOL program. The debug commands listed below can use these line numbers and data-names to affect data-items and program execution in a number of ways.

The compiler will create the debug information file with the filename of the MS-COBOL source file, but with the extension .DBG. For example, compilation of a source file named MYFILE would produce MYFILE.OBJ (object file) and MYFILE.DBG (debug information file).

To suppress creation of a debug information file, use the /D compiler switch (see Chapter 2).

Using the Debug Facility

To use the debug facility, include the file COBDBG.OBJ in the command line when you link your program. For example,

```
LINK MYFILE+COBDBG;
```

enables the debug facility. When you issue the command to execute your program (MYFILE, in this example), the following message will be displayed:

```
MS-COBOL Interactive Debug Facility v. xxx
Program: MYFILE
Type help for list of commands
*
```

The asterisk prompt (*) indicates 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.

Note that without a debug information file, limited debugging is possible. By simply including COBDGB.OBJ in the linker command line, you can enable the debug facility and execute any of the debug commands listed at the end of this section 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 may be abbreviated to the first letter of the command name (the abbreviations are shown by the underlined characters in the following list). Uppercase and lowercase characters are equivalent. Arguments to the commands (line numbers, data-names, ALL, OFF) must be given in full. Though spaces are shown below, arguments can be separated from commands by any nonalphabetic character. When a numeric argument is expected, the debug facility will scan until the first digit on the line is found. For example, the following list of commands are all equivalent (i.e., set a breakpoint at line 100):

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

Pressing your terminal's interrupt key suspends program execution at the next statement, as if a breakpoint had been set at the next line. The key used as the interrupt key for the HP 150 is CTRL-C.

The following functions are available with the debug facility:

Function	**Description**
`Address <data-name>`	Displays absolute address (hexadecimal) of a data-item in memory.

Function (cont'd)	Description (cont'd)
`Breakpoints`	Lists all breakpoints. (A breakpoint is a point at which execution is interrupted so that you can insert a debug command.)
`Breakpoint <line-num>`	Sets 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 paragraph name.
`Change <data-name>`	Displays the contents of <data-name> and allows a new value to be entered. Change cannot be used on index-names or on subscripted or qualified variables.
`Dump [<addr1>[<addr2>]]`	Displays memory addresses (hexadecimal equivalents) from <addrl> through <addr2>. Both addresses are optional. If <addr2> is omitted, 128 bytes are dumped, starting at <addrl>. If both addresses are omitted, 128 bytes are dumped, starting at the last address dumped. Dump uses addresses, not data-names, as arguments. Addresses must start with a digit, even if it is zero (e.g., 0A02 is valid, but A02 is not).

Function (cont'd)	Description (cont'd)
`Exhibit <data-name>`	Displays contents of <data-name>. Data-items of less than 77 characters are displayed within brackets. For data-items greater than 77 characters, the field length of the contents is displayed without brackets. Group names may be displayed, although some components may not be displayable (e.g., binary characters). Exhibit cannot be used on index-names or on subscripted or qualified variables.
`Go`	Resumes execution from the last breakpoint or current program position until a breakpoint or end of program is encountered.
`Goto <line-num>`	Begins execution at <line-num>; continues until breakpoint or end of program is encountered. This command may be used to branch anywhere within a program, even from one overlay segment to another. If a PERFORM is active when Goto is issued, the debug session may abort.
`Help`	Displays the list of debug commands.
`Kill <line-num>`	Removes the breakpoint at <line-num>.
`Kill ALL`	Removes all breakpoints from the breakpoint list.
`Line`	Displays the <line-num> of the current line.

Function (cont'd)	**Description (cont'd)**
`Quit`	Terminates the program (closing all open files).
`Step [<count>]`	Executes one or <count> statements.
`Trace`	Sets trace mode. When trace is set, the line number of each line will be displayed as the line is executed.
`Trace OFF`	Turns off trace mode. (See description of Trace.)

Debugging Subprograms

The interactive debug facility allows you to debug systems of programs consisting of a main program and any number of subprograms. However, there are some limitations on what can be debugged in such a system:

1. Assembly language subroutines may be called, but none of the debugging features will be in effect while the subprogram is executing. For example, no breakpoints can be set in an assembly language subroutine.
2. If subroutines are nested to more than five levels, without a return to an earlier subprogram, the debug facility will not open the debug files for any subprograms beyond the fifth. If this is attempted, the message "No debug information file found" will be generated, even though the information file may actually be present. You may still set breakpoints and use the trace mode at these deeper levels of nesting, but you may not examine or change variables. On return to subprograms nested less than five levels deep, the full debug facilities will again be available.

   This limitation does not hold for systems where a program calls a large number of subprograms but returns to the main program before calling each subprogram.

Appendix A

THE HP 150 TERMINAL INTERFACE

Terminal input/output is performed by the ACCEPT and DISPLAY statements. For the SCREEN SECTION feature and Microsoft extensions to the interactive ACCEPT and DISPLAY statements to run correctly on your terminal, the MS-COBOL runtime system needs to be configured to the characteristics of the terminal. This is already done for the HP 150 in the runtime executor COBRUN.EXE. Therefore, programs compiled and linked with the old versions of the compiler may not run with this new version of COBRUN.EXE. In that case, you must recompile and relink your programs.

Chapter 6 in the *Microsoft COBOL Reference Manual* describes the use of ACCEPT and DISPLAY statements. The following discussion pertains to Format 1, 3, and 4 ACCEPT statements and DISPLAY statements which have positioning or SCREEN specifications.

The following paragraphs describe the screen and keyboard features that have been configured for the HP 150.

Terminal Functions, ASCII Key Names, and Escape Codes

This section lists functions, ASCII key names, and escape codes for the HP 150 terminal.

Table A.1 lists the escape codes which map to the specified functions internally. For example, to monitor which function keys have been pressed, your COBOL program would have to determine which one of the ten escape codes (02 to 11) has been intercepted.

Table A.1. Escape Codes

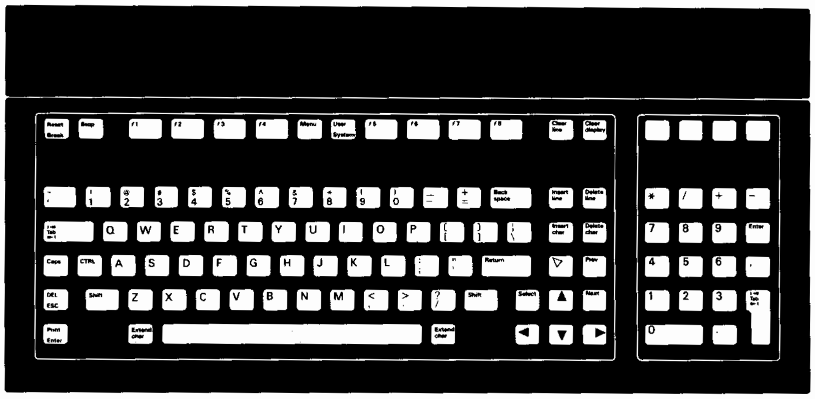
Function	Escape Code
TERMINATOR KEYS	
Backtab	99
Escape	01
Tab	00
Carriage Return	00
Line Feed	00
FUNCTION KEYS	
Function 1	02
Function 2	03
Function 3	04
Function 4	05
Function 5	06
Function 6	07
Function 7	08
Function 8	09
Function 9	10
Function 10	11

Table A.2 lists the HP 150 terminal characteristics.

Table A.2. HP 150 Terminal Interface

Functions	ASCII Key Name
EDITING KEYS	
Delete Line	<DELETE LINE>
Delete Character	<DELETE CHAR>
Nondestructive Forward Space	<RIGHT ARROW>
Destructive Forward Space	<SPACE BAR>
Nondestructive Back Space	<LEFT ARROW>
Destructive Back Space	<BACKSPACE>
Plus Sign	+
Minus Sign	−
TERMINATOR KEYS	
Escape	<ESC>
Back Tab	<SHIFT-TAB>
Tab	<TAB>
Carriage Return	<RETURN>
Line Feed	<CTRL-J>
FUNCTION KEYS	
Function 1	<f1>
Function 2	<f2>
Function 3	<f3>
Function 4	<f4>
Function 5	<f5>
Function 6	<f6>
Function 7	<f7>
Function 8	<f8>
Function 9	<f9>
Function 10	<f10>

All 10 COBOL application softkeys have been implemented (f1-f10). The function keys f9 and f10 are not labeled on the keyboard, but they are the left two of a bank of four keys located above the numeric pad. The application softkeys f1 through f8 are valid touch fields and can be activated by either pressing the function key on the keyboard or by touching the function label on the screen.



The following output functions are not provided with the HP 150 interface:

Cursor On
Cursor Off

Also, the high intensity enhancement is not available, since the HP 150 is normally in high intensity mode. The half-bright enhancement has been implemented in its place.

Due to the HP 150 screen and keyboard configurations (see Chapter 1 and Appendix A), programs compiled with the MS-COBOL compiler, when run, will:

- — home up and clear the display,
- — put the console input device in "raw mode" (MS-DOS I/O control for devices) and keycode mode (the HP 150 alpha and graphics I/O system),
- — show the application softkeys,
- — turn off the touch screen fields, except for the eight function keys (f1-f8),
- — turn off display functions,
- — turn off memory lock,
- — turn off insert character mode,
- — unlock the keyboard,
- — set the appropriate straps and modes,
- — intercept the following keys: f1-f10, RIGHT ARROW, LEFT ARROW, Tab, Shift Tab, <Return>, Backspace, Delete char, Delete line, and ESC and translate them into COBOL escape codes (see Table A.1 in Appendix A).

When the program finishes or is aborted,

- — the console input device is returned to normal mode,
- — the touch screen fields are turned off except for the eight function keys (f1-f8),
- — the "modes" softkeys are displayed.

Appendix B

INTERPROGRAM COMMUNICATION

Interprogram communication is accomplished by using the CALL or CHAIN statements. CALL temporarily transfers control to another program or assembly language subroutine, and CHAIN permanently transfers control to another program. In linking, the calling and called programs or subroutines are linked together, while chained programs are linked separately. The various communications possible with CALL and CHAIN are:

1. Temporary transfer of control from one MS-COBOL program to another (CALL).
2. Temporary transfer of control from an MS-COBOL program to an assembly language subroutine (CALL).
3. Permanent transfer of control from one MS-COBOL program to another (CHAIN).
4. Permanent transfer of control from an MS-COBOL program to an assembly language program (CHAIN).

In addition to transferring program control, these statements can transfer data between programs. This is done with the USING and CHAINING clauses. In a CALL statement, the USING clause lists parameters which give the addresses of data to be acted on within the called program. These data are specified in a corresponding USING clause in the PROCEDURE DIVISION statement of the called program. The called program makes any necessary changes and then returns control to the calling program.

When a program is chained, the USING clause of the CHAIN statement also contains parameters, but in this case the actual values of the parameters in the chaining program are substituted for those of the chained program. This happens because the runtime system copies the data values listed in the chaining program to high memory, loads the chained program into memory, and copies the data values into their corresponding parameters in the chained program. These parameters are specified by a CHAINING clause in the PROCEDURE DIVISION statement of the chained program.

Note that MS-COBOL programs are limited to passing 12 parameters, and the maximum number of files that may be open in one run unit (a program linked together with other programs or subroutines) may be limited. See the *Microsoft COBOL Reference Manual* for more information on space limitations.

Calling Microsoft COBOL Programs

The CALL statement is used to temporarily transfer control to another MS-COBOL program. The two programs are compiled separately and are then linked together (see Chapter 3). Control will be returned to the calling program by an EXIT PROGRAM statement in the called program.

The format of the CALL statement is:

<literal> is the PROGRAM-ID defined in the IDENTIFICATION DIVISION of a COBOL program. The literal must be non-numeric and enclosed in quotation marks.

<data-name(s)> are references whose addresses are passed to the called program. Data-names are discussed below.

The USING clause specifies data-items in the calling program (that can be used by the called program.) For example, a program that needed inventory totals could CALL another program to calculate the totals and place them into designated data-names in the calling program. When this clause is used, the following requirements must be met:

1. Within the calling program:

   The data-names listed in the USING clause must be declared in the WORKING-STORAGE SECTION of the DATA DIVISION.

2. Within the called program:

   The data-names corresponding to those in the USING clause of the calling program must be declared in the LINKAGE SECTION of the DATA DIVISION and in a USING clause after the PROCEDURE DIVISION header. The names in the LINKAGE SECTION and in the PROCEDURE DIVISION header must be in the same order.

Control is returned to the calling program by an EXIT PROGRAM statement in the PROCEDURE DIVISION.

The programmer must make sure that the data-items listed in the calling program and in the called program are equivalent. See the *Microsoft COBOL Reference Manual* for more detailed information on data-items.

Example:

```
 Calling Program
  .
  .
  .
DATA DIVISION.
WORKING-STORAGE SECTION.
01 DATA-NAME PIC 99.
  .
  .
  .
PROCEDURE DIVISION.
  .
  .
  .
CALL PROG2 USING DATA-NAME.
  .
  .
  .

 Called Program

IDENTIFICATION DIVISION.
PROGRAM-ID. PROG2.
  .
  .
  .
DATA DIVISION.
LINKAGE SECTION.
01 LOCAL-REFERENCE PIC 99.
  .
  .
  .
PROCEDURE DIVISION USING LOCAL-REFERENCE.
  .
  .
  .
EXIT PROGRAM.
```

Calling Assembly Language Subroutines

An MS-COBOL program may call assembler subroutines. (See your MS-DOS manual for instructions on writing assembly language programs.) The runtime system transfers execution to a subroutine by means of a machine language FAR CALL instruction. Execution should return via the MS-Macro Assembler RET instruction.

Parameters are passed by reference (i.e., by passing the address of the parameter). Parameter addresses are passed on the stack (see Figure B.1).

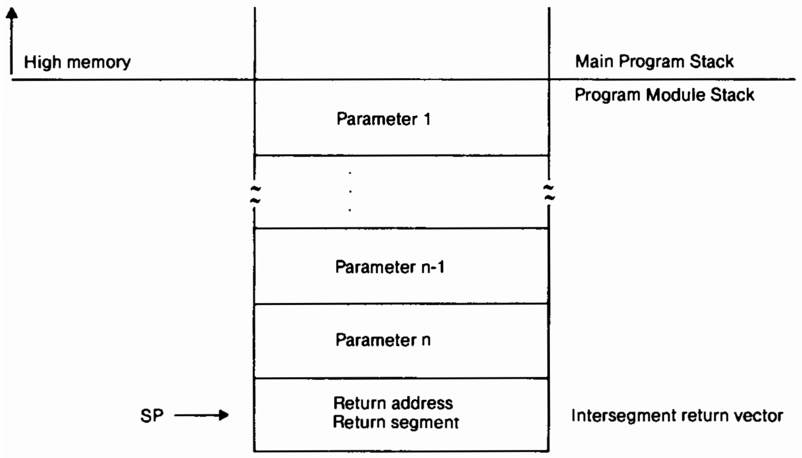


Figure B.1. Contents of Stack at Entry to a Routine

The called routine must preserve the BP register contents and remove the parameter addresses from the stack before returning.

The subroutine can expect only as many parameters as are passed, and the calling program is responsible for passing the correct number of parameters. It is up to the user to determine that the type and length of arguments passed by the calling program are acceptable to the called subroutine; neither the compiler nor the common runtime system checks for the correct number of parameters. Numeric values to be passed should be declared as binary (i.e., USAGE IS COMP-0 in the WORKING-STORAGE SECTION of the calling program).

Because the stack space used by an MS-COBOL program is contained within the program boundaries, assembler programs that use the stack must not overflow or underflow the stack. The best way to assure safety is to save the MS-COBOL stackpointer upon entering the routine and to set the stackpointer to another stack area. The assembler routine must then restore the saved MS-COBOL stackpointer before returning to the main program.

To call an assembler program module, use the name of the module in the CALL statement. The name of an assembler program module is defined by a PUBLIC directive and is declared as PROC FAR. Compile and/or assemble the program(s) and assembly language subroutine(s). Then link the called program module to the calling program using MS-LINK, as described in Chapter 3 in this manual and in your MS-DOS manual.

Example:

COBOL Program

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID.  EXAMPLE.
*DEMONSTRATE CALLING AN ASSEMBLY LANGUAGE PROGRAM
 ENVIRONMENT DIVISION.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 77  PARM1     PIC 99 COMP-0 VALUE 50.
 77  PARM2     PIC 99 COMP-0 VALUE 45.
 77  PARM3     PIC 99 COMP-0 VALUE 0.
 77  PAR1      PIC 99.
 77  PAR2      PIC 99.
 77  PAR3-DIF  PIC 99.
 PROCEDURE DIVISION.
 MAIN.
     CALL 'SUBIT' USING PARM1, PARM2, PARM3.
     MOVE PARM1 to PAR1.
     MOVE PARM2 to PAR2.
     MOVE PARM3 to PAR3-DIF.
     DISPLAY PAR1 ' - ' PAR2 ' = ' PAR3-DIF.
     STOP RUN.
```

Assembly Language Program

```
        assume   cs:codeseg
parm    struc                       ;stack definition
savebp  dw       ?                  ;saved caller's bp
        dw       ?                  ;caller's ip reg
        dw       ?                  ;caller's cs reg
parm3   dw       ?                  ;addr 3rd parameter
parm2   dw       ?                  ;addr 2nd parameter
parm1   dw       ?                  ;addr 1st parameter
        parm     ends

codeseg segment  para
        public   subit              ;entry point
subit   proc     far                ;long call
        push     bp                 ;save bp of caller
        mov      bp,sp              ;set up stack frame
        mov      bx,[bp].parm1      ;get addr of parm1
        mov      ax,[bx]            ;put value in ax
        mov      bx,[bp].parm2      ;get addr of parm2
        sub      ax,[bx]            ;sub values
        mov      di,[bp].parm3      ;get addr of parm3
        mov      [di],ax            ;put result into parm3
        pop      bp                 ;restore caller's bp
        ret      6                  ;restore stack
subit   endp
codeseg ends
        end
```

Chaining MS-COBOL Programs

The CHAIN statement is used to permanently transfer control to a separately compiled and separately linked program, which is loaded into memory and executed. The chained program can issue its own CHAIN statement or may even issue a CHAIN statement to its original chaining program, but it cannot issue an actual return to the original program.

The format of the CHAIN statement is:

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

<literal> or <identifier-1> is the file-name of an executable program. The only difference between them is that the literal must be enclosed in quotation marks, while the identifier does not use quotation marks. Both must be alphanumeric. <identifier-1> must contain a terminating space.

<identifier-2> is a data-item identified in the WORKING-STORAGE SECTION of the chaining program.

For more details about CHAIN format, see the *Microsoft COBOL Reference Manual.*

If the USING clause is included, the values of the data-items listed there will be copied to high memory, and when the chained program is loaded and run, they will be substituted for the equivalent values in the chained program. This allows the user to run a new program using values established in an earlier program. When this clause is used, the following requirements must be met:

1. Chaining Program

   The data-items listed in the USING clause must be declared in the WORKING-STORAGE SECTION of the DATA DIVISION.

2. Within the Chained Program

   The data-items corresponding to those in the USING clause of the chaining program must be declared in the WORKING-STORAGE SECTION of the DATA DIVISION and in a CHAINING clause after the PROCEDURE DIVISION.

Example:

```
 Chaining Program
  .
  .
  .
DATA DIVISION.
WORKING-STORAGE SECTION.
01 DATA-ITEM PIC 99.
  .
  .
  .
PROCEDURE DIVISION.
  .
  .
  .
    CHAIN PROG2 USING DATA-ITEM.


 Chained Program
  .
  .
  .
DATA DIVISION.
WORKING-STORAGE SECTION.
01 LOCAL-REFERENCE PIC 99.
  .
  .
  .
PROCEDURE DIVISION CHAINING LOCAL-REFERENCE.
  .
  .
  .
```

Chaining Assembly Language Programs

Assembly language programs are chained the same way as MS-COBOL programs (see the section on "Chaining MS-COBOL Programs"). The following additional information will be useful when you are writing assembly language programs that will be chained.

When the USING clause is included in the CHAIN statement, the parameters passed between programs are stored at the highest available memory address. This address is determined from byte 2 of the program header (see your MS-DOS manual for more information).

The memory layout is as follows, starting at the highest available address and proceeding toward location zero (see Figure B.2):

1. 256 bytes are reserved for stack space.
2. 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 the bytes were stored in the DATA DIVISION, beginning at the address of the length minus the length itself (see Figure B.2).
3. Each parameter in the USING list follows in order, each preceded by its length in bytes.

The chained program must expect the same number and format of parameters as were passed. No checking will be done by the compiler or the common runtime system.

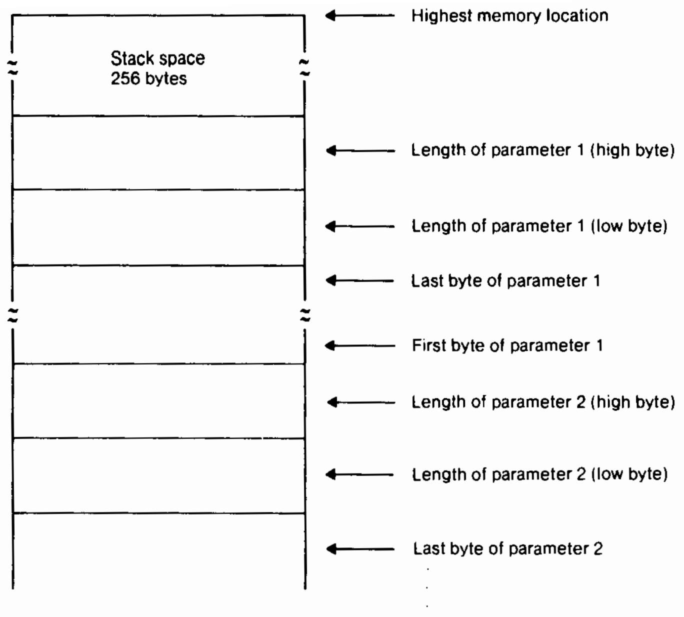

Highest memory location
Stack space
256 bytes
Length of parameter 1 (high byte)
Length of parameter 1 (low byte)
Last byte of parameter 1
First byte of parameter 1
Length of parameter 2 (high byte)
Length of parameter 2 (low byte)
Last byte of parameter 2

Figure B.2. Memory Layout for Chained Programs

Appendix C

CUSTOMIZATIONS

This appendix is intended for those who are proficient with a debugger and/or assembly language and would like to change some of the built-in parameters of Microsoft COBOL.

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. The table originally defines ten stops at the following columns (counting from column 1):

```
8, 12, 20, 28, 36, 44, 52, 60, 68, and 73
```

These may be changed by patching the table. The address is 15 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 that:

1. the numbers are in ascending order
2. no more than 10 stops are defined
3. the last tab stop is 73

Compiler Listing Page Length

One byte in the compiler defines the page length of the listing as 55 (hex 37) lines. Its location is 14 bytes from the start of COBOL.COM, and it may be patched to any value between 1 and 255.

Appendix D

COMPILER PHASES

Microsoft COBOL Compiler creates an object code program from your source program. This is done in five "phases," consisting of the root portion of the compiler, COBOL.COM, and four overlays, COBOL1.OVR through COBOL4.OVR. These are the phases referenced by an error message such as "?Compiler error in phase n."

Compilation is performed in two passes:

The first pass creates an intermediate version of the program, which is stored in a binary file called COBIBF.TMP. This is done in three steps:

Phase 0 (the root portion of the compiler) compiles the IDENTIFICATION and ENVIRONMENT DIVISIONS of the source program.

Phase 1 (COBOL1.OVR) compiles the DATA DIVISION of the source program.

Phase 2 (COBOL2.OVR) compiles the PROCEDURE DIVISION of the source program.

The compiler's second pass reads the intermediate file and creates the object code:

Phase 3 (COBOL3.OVR) reads the intermediate file and creates the object code.

Phase 4 (COBOL4.OVR) allocates file control blocks and finalizes the object code.

Appendix E

REBUILD: INDEXED FILE RECOVERY UTILITY

Please read the enhancements to REBUILD version 1.21 in the front of this manual first.

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 1.00 or later.

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 (in the preceding list) occurs when WRITE produces a boundary error (file status "24"), indicating that 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 is therefore not recoverable by REBUILD.

Recovery from situation 2 (in the preceding list) 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:

1. 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.

2. 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.

Running REBUILD

REBUILD is itself an MS-COBOL program. Therefore, when you are running REBUILD, COBRUN.EXE must be present on a disk in the default drive or drive A.

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 1.00 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 E.1. Following the diagram are detailed descriptions of the individual recovery steps and a sample REBUILD session.

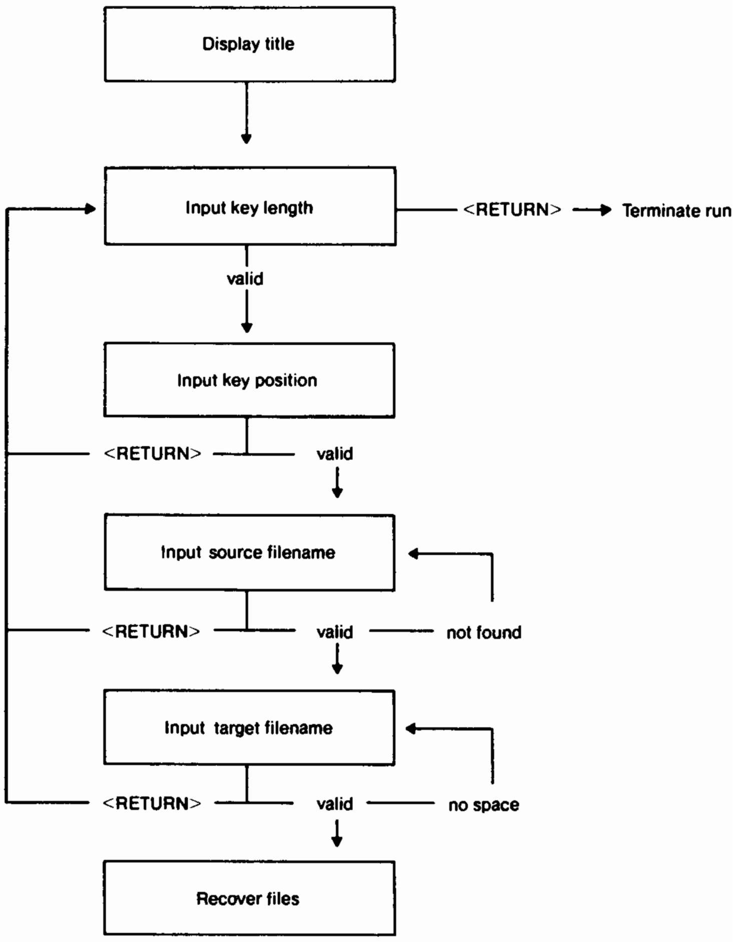


Figure E.1. Control Flow Within REBUILD

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 <RETURN> 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 <RETURN> 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 ---->
   ```

   Enter a filename; or press <RETURN> 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 <RETURN>. 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.

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 F

DEMONSTRATION PROGRAMS

The following demonstration programs are included with MS-COBOL Compiler.

CRTEST

CRTEST is a test program for the terminal interface, as modified for the HP 150. CRTEST must be compiled and linked before it can be run. (Follow directions for compiling and linking in Chapter 1, "Sample Session.") When you run the program, it will prompt you for input.

CENTER

CENTER is a program that centers a line of text or aligns it with the left or right margin. It is a simple MS-COBOL program that does not use sophisticated screen handling features. Like CRTEST, it must be compiled and linked before execution. It will also prompt you for input.

MS-COBOL Demonstration System

The MS-COBOL demonstration system consists of three MS-COBOL programs:

```
DEMO.COB
BUILD.COB
UPDATE.COB
```

Linked versions of these programs are also included on your disks (DEMO.EXE, DEMO_01.OVL, UPDATE.EXE), so you can run the demonstration system immediately.

DEMO is the executive program of the system. It asks if you would like a demonstration of the MS-COBOL SCREEN SECTION, or whether you would like to create or update an indexed (ISAM) file of names, addresses, and phone numbers.

Use the following procedure to run DEMO.

1. Either copy COBRUN.EXE onto the disk containing the files DEMO.EXE, UPDATE.EXE and DEMO_01.OVL; or insert a disk containing COBRUN.EXE into drive A.
2. Type

   ```
   B:
   ```

   to make drive B the default drive.
3. Now type:

   ```
   DEMO
   ```

   When DEMO has been loaded, it will ask you if INSTALL has been run. Since the HP 150 screen and keyboard characteristics have already been confgured into COBRUN.EXE, you can disregard this message and go on with the rest of the DEMO program.

The COBOL source files for DEMO, BUILD, and UPDATE are included to allow you to see the code that produces screens and system files. To recreate the system from the source files, perform the following steps:

1. Insert a disk containing the compiler (COBOL.COM) and COBOL overlays (COBOL1.OVR -COBOL4.OVR) into drive A. Insert the disk containing DEMO.COB, DEMO.CPY, BUILD.COB, and UPDATE.COB into drive B. We recommend that you copy these files onto a blank disk to allow room for object (OBJ) and executable (EXE) files on the disk.

Make drive B the default drive by typing:

```
B:
```

2. Now type:

```
A:COBOL DEMO,,CON;
```

This compiles DEMO.COB and produces DEMO.OBJ. The use of CON in the command line directs the compiler listing to the terminal screen (console); this allows you to watch the program compile. You should receive the message, "No errors or warnings" when the compilation process is finished.

3. Type

```
A:COBOL BUILD,,CON;
```

to compile BUILD.COB.

4. When the compilation process is finished, type

```
A:COBOL UPDATE,,CON;
```

to compile UPDATE.COB. When that compilation process is finished, type

```
<DIR *.OBJ>
```

You should find the files DEMO.OBJ, BUILD.OBJ, and UPDATE in the directory listing.

5. Replace the disk in drive A with your utility disk containing LINK.EXE, COBOL1.LIB, COBOL2.LIB, and COBRUN.EXE.

Link DEMO.OBJ and BUILD.OBJ together by typing:

```
A:LINK DEMO+BUILD,,,A:;
```

Note that both DEMO.EXE and DEMO_01.OVL are produced.

6. Link UPDATE.OBJ by typing:

```
A:LINK UPDATE,,,A:;
```

Command file CLDEMO.BAT will compile and link these programs as a batch process. This file uses the /D (debug) compiler switch, so the .DBG files, used by the debug facility, will not be produced.

This completes the demonstration programs.

Appendix G

MICROSOFT COBOL ERROR MESSAGES

This appendix lists all the error messages you may encounter while compiling and executing a Microsoft COBOL program. Errors fall into the categories described in the following paragraphs.

Compile Time Errors

Compile time errors can be:

1. Command input errors and operating system input/output errors. These errors will be displayed as the errors occur during compilation. When you receive one of these messages, correct the problem and recompile.
2. Program syntax errors in the MS-COBOL source program. These messages are placed at the end of the listing file and are also shown on the terminal. They consist of:

   The source program line number, which is four digits followed by a colon (:).

   An explanation of the error. If the explanation begins with an /F/ (inconsistent file usage) or a /W/ (warning), then the message is only a warning; if not, the error is severe enough to prevent you from linking and executing the object file.

   Whether or not a listing has been requested, the syntax error messages will always be listed on your terminal at the end of compilation. A message displaying the total number of errors or warnings is also displayed. This feature allows you to make a simple change to a program, recompile it without a listing, and still receive any error messages at your terminal.

Program syntax error messages in this manual are listed in alphabetical order, with /F/ and /W/ warnings placed at the end of the list. The number included with an /F/ warning represents the order in which files are entered in the FILE SECTION of the MS-COBOL program.

Runtime Errors

Runtime errors can be:

1. MS-COBOL execution errors

   Some programming errors cannot be detected by the compiler but cause the program to end prematurely during execution. These runtime errors are displayed in the format:

   ```
   **RUN-TIME ERR:
   reason
   line number*
   program-id
   ```

2. MS-COBOL program load errors

   Chained programs, independent segments (i.e., overlays), and the common runtime executor need to be loaded by the MS-COBOL runtime system. During the loading process, the normal mechanism for reporting runtime errors may have been overlayed by the new program. Therefore, the MS-COBOL loader generates its own error messages. The format is:

   ```
   **COBOL: problem
   ```

MS-LINK Errors

A list of MS-LINK error messages may be found in the manuals that are supplied with your MS-DOS software. For your convenience, we have also listed them in the last part of this appendix.

All linker errors cause the link session to abort. After the cause has been found and corrected, MS-LINK must be rerun.

*See the compiler switch /D, in Chapter 2.

Command Input and Operating System I-O Errors

`?Bad filename`

A filename is not constructed according to the rules of the operating system.

`?Bad switch:/X`

You have entered a switch parameter that the compiler does not recognize.

`?Can't create file`

An output file cannot be opened. For example, the output disk is write-protected.

`?Command error:'X'`

You have an invalid character (X) in the command line. For example, a filename contains an @.

`?Compiler error in Phase n at address`

Usually caused by a damaged source program or damaged compiler or overlay file. In the latter case, try your backup copy.

If this does not work, you can sometimes determine the cause of the error by compiling increasingly larger portions of the program, starting with only a few lines, until the error recurs.

See Appendix D for a discussion of compiler phases.

`?Disk X full`

The disk in the specified drive is full. If X is blank, it refers to the default drive.

`?File not found`

You have specified a filename for input that does not exist.

`?Memory full`

Occurs when there is insufficient memory for all the symbols and other information obtained from the source program. It indicates that the program is too large and must be decreased in size or split into modules and compiled separately.

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, with 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 uses about 14 bytes of memory during compilation, and each line in the PROCEDURE DIVISION uses about 3 1/4 bytes.

`?Overlay n not found`

One of the MS-COBOL Compiler overlay files (COBOLn.OVR) is not on the disk. It may have been written to another disk or destroyed. Recompiling and relinking may eliminate the problem.

Program Syntax Errors

`A FILE-ID NAME IS UNDEFINED.`

A data-name specified in a VALUE OF FILE-ID clause is not defined.

`A PARAGRAPH DECLARATION IS REQUIRED HERE.`

An EXIT statement is not followed by a section or paragraph header.

`AREA A NOT BLANK IN CONTINUATION LINE.`

A character was encountered in Area A.

`AREA-A VIOLATION; RESUMPTION AT NEXT PARAGRAPH/SECTION/ DIVISION/VERB.`

The entry starting in one of columns 8-12 cannot be interpreted as a division header, section name, paragraph name, file description indicator, or 01 or 77 level number.

`CLAUSES OTHER THAN VALUE DELETED.`

The data-description of a level 88 item includes a descriptive clause other than VALUE IS.

`ELEMENT LENGTH ERROR.`

The length of the quoted literal is over 120 characters; or the numeric literal is over 18 digits; or the identifier/name is over 30 characters.

`ERRONEOUS FILENAME IS IGNORED.`

An entry which has not been declared as a filename appears where a filename is required.

`ERRONEOUS QUALIFICATION; LAST DECLARATION USED.`

The qualifiers used with a data-name are incorrect or are not unique.

`ERRONEOUS SUBSCRIPTING; STATEMENT DELETED.`

Too few or too many subscripts are provided for a data-name.

`EXCESSIVE LITERAL POOL OR DISPLAY STRING LENGTH.`

The total length of the literals contained within a single paragraph is greater than 4096 bytes.

```
EXCESSIVE NUMBER OF FILES/4KB WORKING-STORAGE BLOCKS.
```

The sum of (number of files declared) + (size of WORKING-STORAGE divided by 4KB and rounded up) + (number of level 01 and level 77 entries in the LINKAGE SECTION) is greater than 14.

```
EXCESSIVE OCCURS NESTING IS IGNORED.
```

OCCURS clauses are nested more than three deep.

```
EXCESSIVE SEGMENT NUMBER.
```

A section header contains a section number greater than 99.

```
EXCESSIVE SEGMENT NUMBER IN DECLARATIVES.
```

A section header in the DECLARATIVES region contains a section number greater than 49.

```
FILE NOT SELECTED; ENTRY BYPASSED.
```

An FD is given for a filename which does not appear in any SELECT sentence.

```
FILL CHARACTER CONFLICT.
```

In a Format 3 ACCEPT statement, SPACE-FILL and ZERO-FILL are both specified.

```
FRACTIONAL EXPONENT OR NEGATIVE SCALED BASE (99P).
```

In a COMPUTE statement, an exponent is a numeric literal with a decimal point or a numeric data-item described with a digit to the right of an assumed decimal point; or the PICTURE of an exponentiation base (entry preceding **) contains the character P as the rightmost digit.

```
GROUP ITEM, THEREFORE PIC/JUST/BLANK/SYNC IS IGNORED.
```

A phrase which is only allowed for elementary data-items is used in the description of an item that is followed immediately by an item of a higher level number.

```
GROUP SIZE GREATER THAN 4095; LENGTH SET TO 1.
```

The size of an item at a level other than 01 is declared to be greater than 4095 bytes.

```
ILLEGAL CHARACTER.
```

An invalid character has been encountered.

`ILLEGAL COPY FILENAME.`

The filename for the copy file is invalid.

`ILLEGAL MOVE OR COMPARISON IS DELETED.`

The operands of a MOVE statement or relational condition are incompatible.

`IMPERATIVE STATEMENT REQUIRED. STATEMENT DELETED.`

A conditional statement is contained within a conditional statement other than IF.

`IMPROPER CHARACTER IN COLUMN 7.`

An invalid character in column 7 has been encountered.

`IMPROPER PICTURE. PIC X ASSUMED.`

An invalid PICTURE clause has been encountered.

`IMPROPER PUNCTUATION.`

Incorrect punctuation has been encountered. For instance, a comma or period must be followed by a space.

`IMPROPER REDEFINITION IGNORED.`

The data-name specified in a REDEFINES clause is not at the same level as the current data-name, or it is separated from it by an item with a lower level number.

`IMPROPERLY FORMED ELEMENT.`

Incorrect syntax for an item has been encountered. For instance, you may have ended a word with a hyphen or used multiple decimal points in a numeric literal.

`INCOMPLETE (OR TOO LONG) STATEMENT DELETED.`

A verb immediately follows a partial statement form, or an otherwise acceptable statement is too large for the compiler to read.

`INDEXED/RELATIVE REQUIRES DISK ASSIGNMENT.`

A file assigned to PRINTER is described as having indexed or relative organization.

`INVALID KEY SPECIFICATION.`

The key item for a relative or indexed file should not be subscripted, or it is inconsistent with the file organization in class or USAGE. This message is issued when the OPEN statement is processed.

INVALID QUOTED LITERAL.

A literal of zero length, improper construction, or missing end quotes has occurred.

INVALID SELECT-SENTENCE.

The syntax of a SELECT sentence in the FILE-CONTROL paragraph is incorrect.

INVALID VALUE IGNORED.

The value specified in a VALUE IS phrase is not a properly formed literal.

JUSTIFICATION CONFLICT.

In a Format 3 ACCEPT statement, LEFT-JUSTIFY and RIGHT-JUSTIFY are both specified.

KEY DECLARATION OF THIS FILE IS NOT CORRECT.

The RELATIVE KEY clause is missing for a relative file, or the RECORD KEY clause is missing for an indexed file.

KEYS MAY ONLY APPLY TO AN INDEXED/RELATIVE FILE.

A RECORD KEY or RELATIVE KEY clause was specified for a file with sequential or line sequential organization.

LITERAL TRUNCATED TO SIZE OF ITEM.

The literal specified in a VALUE IS phrase is larger than the data-item being declared.

MISORDERED/REDUNDANT SECTION PROCESSED AS IS.

A section in the IDENTIFICATION, ENVIRONMENT, or DATA DIVISION is out of order or repeated.

NAME OMITTED; ENTRY BYPASSED.

The data-name is missing in a data description entry.

NON-CONTIGUOUS SEGMENT DISALLOWED.

Two sections with the same number, larger than 49, are separated by one or more sections with a different number.

NO PICTURE; ELEMENTARY ITEM ASSUMED TO BE BINARY.

No PICTURE is given for an elementary data-item.

OCCURS DISALLOWED AT LEVEL 01/77, OR COUNT TOO HIGH.

An OCCURS clause appears in a data-description entry at level 01 or 77; or the number of occurrences specified is greater than 1023.

```
OMITTED WORD 'SECTION' IS ASSUMED HERE.
```

The required word SECTION is missing from the header of a section in the DATA DIVISION.

```
PROCEDURE-NAME IS UNRESOLVABLE.
```

A reference to a section name or procedure-name is not sufficiently qualified or is not unique.

```
PROCEDURE RANGE NOT IN CURRENT SEGMENT.
```

A PERFORM statement in a section with a number greater than 49 refers to a procedure in a section with a different number greater than 49.

```
PROCEDURE RANGE SPANS SEGMENTS.
```

A procedure range (procedure-name-1 THRU procedure-name-2) mentioned in a PERFORM statement contains paragraphs in sections with different section numbers greater than 49, or in sections numbered both less than or equal to 49 and greater than 49.

```
REDUNDANT FD PROCESSED AS IS.
```

The same filename appears in more than one file description.

```
REWRITE VALID ONLY FOR A DISK FILE.
```

The filename entry in a REWRITE statement is a file assigned to PRINTER.

```
SEMANTICAL ERROR IN SCREEN DESCRIPTION.
```

This message can be caused in five different ways:

The SCREEN SECTION does not begin with a level 01 screen item description.

A level 01 screen item description does not include a screen name.

A group screen item is described with a clause which is allowed only for elementary items.

An elementary screen item description is missing FROM, TO, USING, or VALUE clauses.

A screen item description contains inconsistent clauses (such as USING and VALUE).

```
SIGN CLAUSE IGNORED FOR UNSIGNED ITEM.
```

The PICTURE of a numeric item with USAGE IS DISPLAY describes it as unsigned, but a SIGN IS clause is present.

```
SINGLE-SPACING ASSUMED DUE TO IMPROPER ADVANCING COUNT.
```

The operand of the BEFORE or AFTER phrase of a WRITE statement is not numeric, or it is outside the range 0-120.

```
SOURCE BYPASSED UNTIL NEXT FD/SECTION.
```

An error in a file description prevents further analysis.

```
STATEMENT DELETED BECAUSE INTEGRAL ITEM IS REQUIRED.
```

A numeric data-item whose PICTURE specifies digits to the right of the decimal point is used where an integer is required.

```
STATEMENT DELETED BECAUSE OPERAND IS NOT A FILENAME.
```

A name appearing where a filename is required has not been declared as a filename.

```
STATEMENT DELETED DUE TO ERRONEOUS SYNTAX.
```

A syntax error, to which no more specific message applies, is present.

```
STATEMENT DELETED DUE TO NON-NUMERIC OPERAND.
```

An alphanumeric or alphanumeric-edited item is used as an operand of an arithmetic statement; a numeric-edited item is used as an operand other than the result; or a number is longer than 18 digits.

```
SUBSCRIPT 0 OR OVER MAX. NO. OCCURRENCES; 1 USED.
```

A literal used as a subscript is inconsistent with the range defined by the associated OCCURS clause.

```
SUBSCRIPT OR INDEX-NAME IS NOT UNIQUE.
```

A name which requires qualification is used as a subscript.

```
SYNTAX ERROR IN SCREEN DESCRIPTION.
```

A screen item description contains a clause which is unrecognizable, improperly constructed, or redundant.

UNRECOGNIZABLE ELEMENT IS IGNORED.

A required keyword is missing, or a data-name or procedure name is unidentified.

USING-LIST ITEM LEVEL MUST BE 01/77.

A name used in the PROCEDURE DIVISION header USING list is not declared at level 01 or level 77.

VALUE DISALLOWED--OCCURS/REDEFINES/TYPE/SIZE CONFLICT.

The VALUE IS clause is specified for a data-item described with (or included within an item described with) an OCCURS or REDEFINES clause; or the literal given in a VALUE IS clause is not compatible with the PICTURE of the declared item.

VALUE OF FILE-ID REQUIRED.

The VALUE OF FILE-ID clause is not specified in the file description of a file assigned to DISK.

VARYING ITEM MAY NOT BE SUBSCRIPTED.

The data-item controlled by the VARYING phrase of a PERFORM statement is subscripted.

File Usage Errors

/F/ FILE NEVER CLOSED.

No CLOSE statement is present for the file.

/F/ FILE NEVER OPENED.

No OPEN statement is present for the file.

/F/ INCONSISTENT READ USAGE.

An OPEN INPUT statement is present for a file, but no READ statement; or vice versa.

/F/ INCONSISTENT WRITE USAGE.

An OPEN OUTPUT statement is present for a file, but no WRITE statement; or vice versa.

Warning Errors

`/W/ BLANK WHEN ZERO IS DISALLOWED.`

The BLANK WHEN ZERO phrase appears in the description of an alphanumeric or alphanumeric-edited item.

`/W/ DATA DIVISION ASSUMED HERE.`

The DATA DIVISION header is missing.

`/W/ DATA RECORDS CLAUSE WAS INACCURATE.`

The record-name(s) given in a DATA RECORDS clause are not consistent with the record descriptions following the file description.

`/W/ ERRONEOUS RERUN-ENTRY IS IGNORED.`

A RERUN clause of the I-O-CONTROL paragraph contains a syntax error.

`/W/ FD-VALUE IGNORED SINCE LABELS ARE OMITTED.`

The VALUE OF FILE-ID clause is used in the description of a file which is assigned to PRINTER.

`/W/ FILE SECTION ASSUMED HERE.`

The FILE SECTION header is missing.

`/W/ INVALID BLOCKING IS IGNORED.`

The BLOCK clause of an FD contains an error.

`/W/ INVALID RECORD SIZE(S) IGNORED.`

The RECORD clause of an FD contains an error.

`/W/ 'LABEL RECORD STANDARD' REQUIRED.`

The LABEL RECORD(S) STANDARD phrase is not present in the FD of a file assigned to DISK.

`/W/ LABEL RECORDS OMITTED ASSUMED FOR PRINTER FILE.`

The LABEL RECORDS OMITTED clause is missing in the file description of a file assigned to PRINTER.

`/W/ LEVEL 01 ASSUMED.`

A record-description begins with a level number other than 01.

`/W/ PERIOD ASSUMED AFTER PROCEDURE-NAME DEFINITION.`

A section or paragraph header does not end with a period.

`/W/ PICTURE IGNORED FOR INDEX ITEM.`

A data-item described with USAGE IS INDEX phrase also has a PICTURE phrase.

`/W/ PROCEDURE DIVISION ASSUMED HERE.`

The PROCEDURE DIVISION header is missing.

`/W/ RECORD MAX DISAGREES WITH RECORD CONTAINS; LATTER SIZES PREVAIL.`

The record size specified in the RECORD CONTAINS clause of an FD is inconsistent with the sizes of the associated record-descriptions.

`/W/ REDUNDANT CLAUSE IGNORED.`

The same clause is specified more than once in a file description.

`/W/ RIGHT PARENTHESIS REQUIRED AFTER SUBSCRIPTS.`

The closing parenthesis for a subscript is missing.

`/W/ TERMINAL PERIOD ASSUMED ABOVE.`

A data-description entry or paragraph does not end with a period.

`/W/ WORKING-STORAGE ASSUMED HERE.`

The WORKING-STORAGE header is missing.

Runtime Errors

`CURSOR POSITION`

You tried to position the cursor beyond the line or column limits of the screen. A format 3 or 4 ACCEPT statement or a DISPLAY statement with a position-spec or screen-name is the statement responsible for the error. If a screen has been displayed or accepted, one or more fields within the screen have starting positions outside the maximum screen line or column.

`DATA UNAVAILABLE.`

You tried to reference data in a record of a file that is not open or has reached the AT END condition.

`DELETE; NO READ.`

You tried to DELETE a record of a sequential access mode file when the last operation was not a successful READ.

`FILE LOCKED.`

You tried to OPEN after an earlier CLOSE WITH LOCK.

`GO TO (NOT SET).`

You tried to execute a null GO TO statement which has never been altered to refer to a destination.

`ILLEGAL DELETE.`

Relative or indexed file not opened for I-O.

`ILLEGAL READ.`

You tried to READ a file that is not open in the INPUT or I-O mode.

`ILLEGAL REWRITE.`

You tried to REWRITE a record in a file not open in the I-O mode.

`ILLEGAL START.`

File not opened for INPUT or I-O.

`ILLEGAL WRITE.`

You tried 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.

`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, or DECLARATIVES SECTION.

`NEED MORE MEMORY.`

The indexed file manager has ended abnormally because of insufficient dynamically allocatable memory.

`NON-NUMERIC DATA.`

Whenever the content of a numeric item does not conform to the given PICTURE, this condition may arise. Always check input data, if it is subject to error (because input editing has not yet been done) by using the NUMERIC test.

`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.

`PERFORM OVERLAP.`

An illegal sequence of PERFORMs, as, for example,when paragraph A is performed and another PERFORM A is initiated prior to exiting from the first.

`REDUNDANT OPEN.`

You tried to open a file that is already open.

`REWRITE; NO READ.`

You tried to REWRITE a record of a sequential access mode file when the last operation was not a successful READ.

`SEG nn LOAD ERR.`

An error occurred while you were attempting to load an overlayed segment. nn is 31 hex (49 decimal) less than your overlay segment number.

`SUBSCRIPT FAULT.`

A subscript has an illegal value. This error may be caused by an index reference whose value is less than 1.

Program Load Errors

`**COBOL: Attempt to use non-updated runtime module (COBRUN.EXE)`

This message appears when the version number in the runtime libraries is not the same as that in the runtime interpreter (COBRUN.EXE).

`**COBOL: ERROR IN EXE FILE.`

Error in loading chained or common runtime EXE file.

`**COBOL: FILE 'filename' NOT FOUND. ENTER NEW DRIVE LETTER.`

The chained file, segment file, or common runtime file could not be found.

`**COBOL: PROGRAM TOO BIG TO FIT IN MEMORY.`

There is not enough memory available to load a chained program or common runtime file.

MS-LINK Errors

The following error messages are displayed by MS-LINK.

`Attempt to access data outside of segment bounds, possibly bad object module`

There is probably a bad object file.

`Bad numeric parameter`

Numeric value is not in digits.

`Cannot open temporary file`

MS-LINK is unable to create the file VM.TMP because the disk directory is full. Insert a new disk. Do not remove the disk that will receive the LIST.MAP file.

`Error: dup record too complex`

DUP record in assembly language module is too complex. Simplify DUP record in assembly language program.

`Error: fixup offset exceeds field width`

An assembly language instruction refers to an address with a short instruction instead of a long instruction. Edit assembly language source and reassemble.

`Input file read error`

There is probably a bad object file.

`Invalid object module`

An object module(s) is incorrectly formed or incomplete (as when assembly is stopped in the middle). Check for errors and recompile the module.

`Symbol defined more than once`

MS-LINK found two or more modules that define a single symbol name.

`Program size or number of segments exceeds capacity of linker`

The total size may not exceed 384K bytes and the number of segments may not exceed 255.

`Requested stack size exceeds 64K`

Specify a size greater than or equal to 64K bytes with the STACK switch.

`Segment size exceeds 64K`

64K bytes is the addressing system limit.

`Symbol table capacity exceeded`

Very many and/or very long names were entered, exceeding the limit of approximately 25K bytes.

`Too many external symbols in one module`

The limit is 256 external symbols per module.

`Too many groups`

The limit is 10 groups.

`Too many libraries specified`

The limit is 8 libraries.

`Too many PUBLIC symbols`

The limit is 1024 PUBLIC symbols

`Too many segments or classes`

The limit is 256 (segments and classes taken together).

`Unresolved externals: <list>`

The external symbols listed have no defining module among the modules of library files specified.

`VM read error`

This is a disk error; it is not caused by MS-LINK.

`Warning: No stack segment`

None of the object modules specified contains a statement allocating stack space, but the user typed the STACK switch.

`Warning: Segment of absolute or unknown type`

There is a bad object module or an attempt has been made to link modules that MS-LINK cannot handle (e.g., an absolute object module).

`Write error in TMP file`

No more disk space remains to expand VM.TMP file.

`Write error on run file`

Usually, there is not enough disk space for the run file.

INDEX