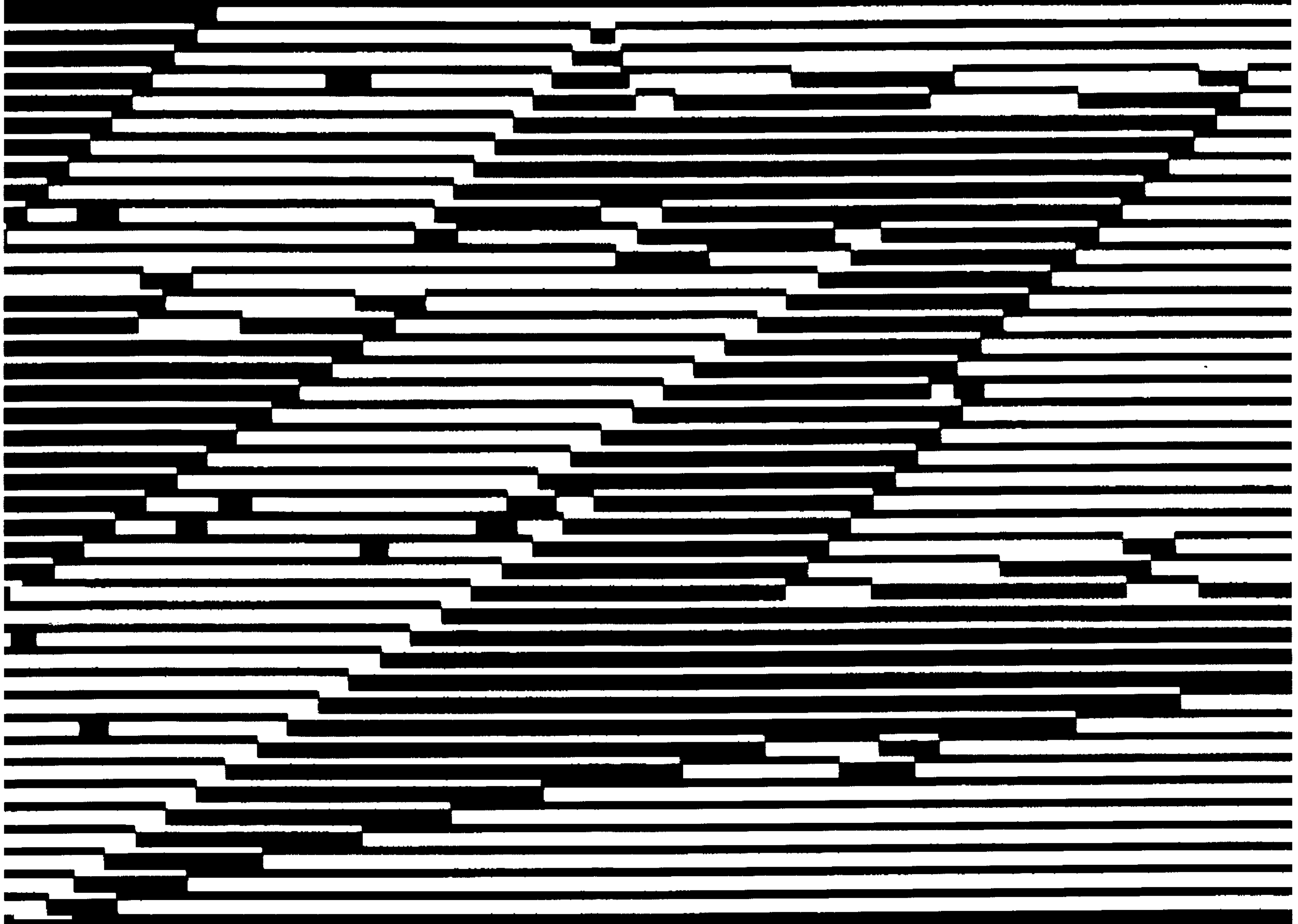
MICROSOFT

FORTRAN Compiler



HEATH | ZENITH data systems

Microsoft FORTRAN-80

Software User's Manual

for HEATH 8-bit digital computer systems

595-2435

HEATH COMPANY
BENTON HARBOR, MICHIGAN 49022

First Printing
Printed in the United
States of America

Table of Contents

Chapter Four – FORTRAN I/O

Chapter Five – User's Guide to the FORTRAN Library Functions

Chapter Six - MACRO-80 Assembler Operating Procedure

Chapter One

Introduction

OVERVIEW

This Microsoft FORTRAN-80 User's Manual will show you how to use the FORTRAN System with HDOS. If you have experience with FORTRAN on a big machine, this Manual will make your transition to the microcomputer world much easier. This Manual is divided into six chapters, which are summarized below.

1. Chapter 1 - "Introduction" - Basic introductory material. The minimum operating environment (hardware and system software) is defined, and the various components of both the documentation and the software are explained.

2. Chapter 2 - "Configuring the Working Diskettes" - A step by step explanation of how to set up your diskettes for use.

3. Chapter 3 - "Operating the FORTRAN System" - How to compile, link and execute a FORTRAN program. Several sample programs are provided so you get actual "hands on" experience.

4. Chapter 4 - "FORTRAN I/O" - The important concepts of FORTRAN I/O. Emphasis is placed upon the interface between FORTRAN I/O and HDOS. Most of the I/O material is concerned with disk files, although output to hard copy devices is also covered.

5. Chapter 5 - "User's Guide to the FORTRAN Library Functions" - Defines and illustrates the library functions available to the FORTRAN user.

6. Chapter 6 - "Operating the Macro-80 System" - This chapter follows much the same format as Chapter 3. A sample program is provided to illustrate the features of this Assembler.

OPERATING REQUIREMENTS

Hardware Configuration

In order to run the FORTRAN-80 System, you must have a minimum hardware configuration. This must include at least 40K of RAM. It also must include at least two (2) mini-floppy disk drives. The use of a hard copy device is optional.

System Software Configuration

FORTRAN-80 will run under HDOS version 1.6 or higher.

THE FORTRAN-80 SYSTEM

FORTRAN-80 Software Package

The FORTRAN-80 Software package consists of the following programs:

1. FORTRAN-80 Compiler - F80.ABS
2. FORTRAN-80 Library - FORLIB.REL
3. LINK-80 Linking Loader - L80.ABS
4. MACRO-80 Assembler - M80.ABS
5. Cross Reference Generator - CREF80.ABS
6. Sample FORTRAN-80 and Macro Assembler source programs

This software package is, in essence, two different systems: the FORTRAN-80 system and the MACRO-80 system. You will probably want to keep each system on a separate diskette. Chapter 2, "Configuring the Working Diskettes", tells you how to set up your working diskettes.

The FORTRAN-80 Compiler provides most of the important features of ANSI Standard FORTRAN (X3.9-1966). (The deviations from the standard are listed at the end of this Chapter.) This will allow you to take advantage of the large base of application programs already written in FORTRAN.

This Compiler is also unique in that it provides a microprocessor based FORTRAN development package that generates relocatable object modules. You can link these modules using the LINK-80 Loader and generate an absolute file. This absolute file can be executed under HDOS in the same manner as any other absolute file.

The FORTRAN-80 Compiler can compile several hundred statements per minute in a single pass. If it detects an error during compilation, a descriptive error message will be printed.

You can use the MACRO-80 Assembler included with this package to assemble either 8080 or Z80 (Zilog format) opcodes. This Assembler has complete facilities for Macro development. The Assembler generates relocatable object modules that you can link using the LINK-80 Loader.

FORTRAN-80 Documentation Package

The FORTRAN-80 documentation package consists of the following documents:

1. The Microsoft FORTRAN-80 Reference Manual

   This is the reference document for the entire package. It is divided into three different sections: the FORTRAN-80 Reference Manual, the MACRO-80 Reference Manual, and the LINK-80 Linking Loader Reference Manual.

   Refer to this document when you have a question about the proper syntax, or the use of a particular statement or group of statements.

   This Manual also contains the operating procedures for the Compiler, Assembler and Linker.

2. The Microsoft FORTRAN-80 User's Manual

   The User's Manual will help you to become comfortable using the FORTRAN-80 system. This Manual will not teach you how to write FORTRAN programs, it assumes that you already know how to write them.

3. The Microsoft FORTRAN-80 Reference Card

   The FORTRAN-80 Reference Card contains a summary of various concepts important to the FORTRAN-80 system. Keep this Card handy, as it contains information you will probably need during your programming effort.

Extensions

The FORTRAN-80 Compiler includes the following extensions to ANSI Standard FORTRAN (X3.9-1966):

1. When you use c in a "STOP c" or "PAUSE c" statement, c may be up to six ASCII characters in length.
2. You may use the ERR= and END = options to specify error and End of File branches in READ and WRITE statements.
3. The standard subprograms PEEK, POKE, INP and OUT have been added to the FORTRAN library.
4. Statement functions may use subscripted variables as arguments.
5. You may use hexadecimal constants wherever integer constants are normally allowed.
6. The literal form of Hollerith data (a character string between apostrophe characters) is permitted in place of the standard nH form.
7. There is no restriction to the number of continuation lines.
8. Mixed mode expressions and assignments are allowed, and the conversion is done automatically.
9. You may use logical variables as integer quantities in the range + 127 to -127.
10. Logical operations may be performed on integer data. (.AND.,.OR.,.NOT., .XOR., can be used for 16-bit or 8-bit Boolean operations.)
11. You may use ENCODE/DECODE for editing and converting data.
12. Complete language facilities are provided for random access files.

These features have been added for the convenience of the FORTRAN programmer. Note the above features and use them to the fullest advantage. This list is also included in the Microsoft FORTRAN-80 Reference Manual.

Restrictions

FORTRAN-80 places the following restrictions upon ANSI Standard FORTRAN:

1. The COMPLEX data type has not been implemented.

2. The statements in a program unit must appear in the following order:

    1. PROGRAM, SUBROUTINE, FUNCTION, BLOCK DATA
    2. Type, EXTERNAL, DIMENSION
    3. COMMON
    4. EQUIVALENCE
    5. DATA
    6. Statement Functions
    7. Executable statements.

    Make sure you understand fully the above restriction. These statements must be in this order.

3. A different amount of computer memory is allocated for each of the following data types: integer, real, double precision, logical.

4. The equal sign of an assignment statement and the first comma of a DO statement must appear on the initial statement line.

5. Unformatted sequential I/O statements must always provide a variable list.

Make sure you read and understand the above restrictions. Your FORTRAN programs must adhere to these restrictions.

Chapter Two

Configuring the Working Diskettes

OVERVIEW

The FORTRAN system is distributed on two floppy diskettes. The one labeled "HDOS Microsoft FORTRAN-80" contains the FORTRAN system files. The one labeled "Microsoft FORTRAN-80 Sample Programs" contains the sample programs used in this Manual. This Chapter contains the procedure you should follow to generate your "working" copy of the FORTRAN diskettes.

In order to perform the configuration procedure, you will need to know how to operate several HDOS utilities. These utilities are listed on the following page. It is very important that you know how to operate these utilities.

Carefully perform the procedures listed on the following pages. Make sure you read and understand each step before you proceed to the next one. A set of parenthesis are provided before each step so you can "check it off" after you complete it. When you are done with this procedure, you will be ready to use your FORTRAN system.

GENERATING WORKING DISKETTES

This Chapter will outline the steps you should follow in order to set up your FORTRAN and MACRO-80 diskettes for use. Before you get started, you should know how to operate the following HDOS (Heath Disk Operating System) utilities:

1. SYSGEN
2. INIT
3. TEST
4. PIP
5. FLAGS
6. SET

If you do not know how to operate any of the above utilities, go back to your HDOS Reference Manual and read the appropriate section. The configuration procedure assumes that you know how to operate each of them.

You will also need several diskettes other than those provided with the FORTRAN system. The following list tells you what you will need in order to get started:

1. HDOS distribution diskette (version -1.6 or higher).
2. "Microsoft FORTRAN-80" distribution diskette (provided with FORTRAN).
3. "Microsoft FORTRAN-80 Sample Programs" diskette (provided with FORTRAN).
4. Three (3) blank diskettes.

The distribution copy of a diskette refers to the copy provided to you when you purchase a software package. For example, when you purchased FORTRAN, you were provided with two diskettes. These are FORTRAN distribution diskettes. The HDOS distribution diskette was provided to you when you purchased HDOS.

Procedure A - Preparing the Diskettes for Data Storage

Perform the following steps to prepare the diskettes for data storage. You will need your HDOS distribution diskette and the three blank diskettes.

( ) 1. Boot your system with the HDOS distribution diskette. Using the HDOS utility INIT, initialize the three blank diskettes.

During this step, which will prepare the blank diskettes for use, you will be asked to provide a volume number and name for each diskette. (Be sure you give each one a unique name and volume number.)

( ) 2. Boot your system with the HDOS distribution diskette. Using the HDOS utility TEST, run the media test (menu function - M) on all three diskettes.

Remember to select menu function U before attempting to replace the previously tested diskette with another one. This will restart the TEST utility and allow you to test all three diskettes without having to reboot each time. For more information on the TEST utility, refer to the HDOS Reference Manual.

This step will check the diskettes for defects. Replace any that have defects.

( ) 3. Boot your system with the HDOS distribution diskette. Using the HDOS utility INIT, reinitialize the three blank diskettes.

The diskettes must be reinitialized; and always run INIT after the TEST program. This is necessary because the TEST program will destroy the directory on a diskette. (Again, be sure each one has a unique name and volume number.)

This completes the diskette preparation procedure. The following page will tell you how to perform the next procedure.

Procedure B - Copying the HDOS System Files

You will need two of the three blank, initialized diskettes in this procedure. You will also need your HDOS distribution diskette.

Perform the following sequence of steps two times. Use a different diskette each time. After you have finished this procedure, you will have created two bootable diskettes.

( ) ( ) 1. Boot your system with the HDOS distribution diskette. Using the HDOS utility SYSGEN, generate the HDOS operating system on one of the diskettes you just reintialized. When the SYSGEN process is over, the following message will be displayed on you console device: .

```
17 FILES COPIED
```

This step will copy the HDOS operating system files on to the diskette, allowing you to "BOOT" your system using this newly created diskette.

( ) ( ) 2. Boot your system using the diskette you previously SYSGENed. Now, using the HDOS utility FLAGS, delete the protection flags from the files listed below. To delete them, simply press RETURN when asked for the "NEW FLAGS:". This step is needed because these files will eventually be deleted to make room for the FORTRAN and MACRO-80 files. (Do not delete the files yet!)

1. ONECOPY.ABS
2. HELP.
3. SYSHELP.DOC
4. SET.ABS
5. FLAGS.ABS
6. ND.DVD, ATH84.DVD, ATH85.DVD,
   LPH14.DVD, LPH24.DVD, LPH44.DVD

( ) ( ) 3. Mount your HDOS distribution diskette on drive SY1:. Using PIP, copy the HDOS text editor from drive SY1: to SY0:.(HDOS text editor file name-EDIT.ABS)

```
*.*=SY1:EDIT.ABS
```

The purpose of this step is to copy the text editor to the diskette you previously SYSGENed. The editor will be used to create source programs when you are using the FORTRAN and MACRO-80 systems.

Note: If you prefer to use another text editor to create your source programs, copy it instead of the HDOS editor.

( ) ( ) 4. Perform this step only if you plan to use a hard copy device.

Using the HDOS utility SET, configure the proper device driver for your hard copy device. (Refer to the HDOS Reference Manual for detailed information if necessary.)

This step will allow you to use a hard copy device with the FORTRAN system. Make sure you are using the correct device driver for your specific hard copy device. Also make sure that you have your port addressed correctly. The HDOS Reference Manual contains all the information you will, need in order to configure your device driver.

( ) ( ) 5. Using the HDOS SET utility, configure your console device.

This step will enable you to set up your console device driver to meet your own specific requirements.

( ) ( ) 6. Exit HDOS. (HDOS command-BYE) Remove both diskettes from the drives.

You have now completed the second procedure.

With a felt-tip pen (do NOT use a pencil or hard-tipped pen), write the name "FORTRAN SYSTEM DISKETTE" on a label and then put the label on the diskette you removed from SY0:.

If this is your first time through the procedure, you will need to perform it again, with another diskette. After you have performed the procedure the second time, write the name "MACRO-80 ASSEMBLER SYSTEM DISKETTE" on a label and then put the label on the diskette you removed from SY0:.

Procedure C - Copying the FORTRAN System Files

This page contains the sequence of steps you need to copy the FORTRAN system files to the proper diskette. In this procedure, you will use the FORTRAN system diskette you SYSGENed in the previous procedure. You will also need the FORTRAN system distribution diskette labeled "HDOS Microsoft FORTRAN80" that you received with this software package. (NOT the Sample Program diskette!)

( ) 1. Boot your system using FORTRAN system diskettes you previously SYSGENed. Using the HDOS utility PIP, delete the files listed below. Before you delete the device drivers, you may wish to list a directory of them on SY0:. If you do not have a hard copy device, simply delete all the device drivers. You must perform this step so there will be enough room left on the diskette for the FORTRAN system files.

   1. SET.ABS
   2. FLAGS.ABS
   3. ONECOPY.ABS
   4. SYSHELP.DOC
   5. HELP.
   6. All device drivers that you are not using.

( ) 2. Mount the FORTRAN system distribution diskette on drive SY1: so you can copy the FORTRAN files to the correct diskette.

( ) 3. Using the HDOS utility PIP, copy the following files from SY1: to SY0:. After you copy them, exit PIP by typing CTRL D.

1.	F80.ABS	*.*=SY1:F80.ABS
2.	L80.ABS	*.*=SY1:L80.ABS
3.	FORLIB.REL	*.*=SY1:FORLIB.REL

Make sure that you have deleted all the files you were instructed to delete. If you should run out of room on the destination diskette, you probably did not delete all the files you should have. You may want to list the directory of SY0: on your console device to verify that the correct files were deleted.

( ) 4. Now perform a preliminary test of this FORTRAN diskette by typing F80 and pressing RETURN. This tells HDOS to load the FORTRAN compiler in memory.

This process will take a little time, as the FORTRAN compiler is a very large file. Eventually, an asterisk "*" should appear on your console device. This is the prompt from the FORTRAN compiler. We are not ready to use FORTRAN yet, so type CTRL D. This will return control to HDOS.

To continue this test, type L80 and press RETURN. This will load the linking loader into memory. After the asterisk is displayed, again type CTRL D.

If you are unable to perform the above test, list a directory of SY0: on your console device. These three files must be present: F80.ABS, L80.ABS, FORLIB.REL. If any of them are not listed in the SY0: directory, try to copy them from SY1: again. If these files are not on SY1:, you probably have the wrong diskette mounted on SY1:. (The FORTRAN distribution diskette should be mounted on SY1:, not the Sample Program diskette!)

( ) 5. Exit HDOS (HDOS command-BYE). Remove both diskettes from the disk drives.

You now have a configured FORTRAN system diskette. This is the diskette you will use to boot your system when you wish to use the FORTRAN Compiler.

Please refer to the next page for a complete list of the steps you must follow in order to configure the MACRO-80 Assembler system diskette.

Procedure D - Copying the MACRO-80 Assembler System Files

In this next sequence of steps, you will need your FORTRAN distribution diskette and the diskette you previously SYSGENed and labeled "MACRO-80 Assembler System Diskette." Now use the following procedure to create a MACRO-80 Assembler system diskette.

( ) 1. Boot your system using the MACRO-80 Assembler diskette you SYSGENed. (NOT the one you used for the FORTRAN system files.) Using the HDOS utility PIP, delete the files listed below. As in the last procedure, you may want to list a directory of the device drivers before you delete them. You must perform this step to allow room for the MACRO-80 Assembler system files.

1. SET.ABS
2. FLAGS.ABS
3. ONECOPY.ABS
4. SYSHELP.DOC
5. HELP.
6. All device drivers that you are not using.

( ) 2. Mount the FORTRAN system distribution diskette on drive SY1:. This step lets you gain access to the MACRO-80 Assembler system files.

( ) 3. Using the HDOS utility PIP, copy the following files from the diskette on SY1: to the diskette on drive SY0:. After copying the files, type CTRL D to exit PIP.

1.	M80.ABS	*.*=SY1:M80.ABS
2.	L80.ABS	*.*=SY1:L80.ABS
3.	FORLIB.REL	*.*=SY1:FORLIB.REL
4.	CREF80.ABS	*.*=SY1:CREF80.ABS

Again, make sure that you have deleted all the files you were instructed to delete. If you run out of room on the destination diskette, you probably did not delete all the files you should have. It is a good idea to list the directory on your console device to verify that all the correct files were deleted.

( ) 4. Now you should perform a preliminary test of your newly created MACRO-80 Assembler diskette. To do this , type M80 and press RETURN.

When the asterisk prompt is displayed on your console device, type CTRL D to return to HDOS. After returning to HDOS, type L80 and press RETURN. This will load the LINK-80 Loader into memory. When the asterisk prompt is displayed, press CTRL D.

Finally, type CREF80 and RETURN. This will load the cross reference program into memory. After the asterisk is displayed, exit by typing CTRL D.

If you are unable to perform the above test, list a directory of SY0: on your console. These files must be present: M80.ABS, L80.ABS, FORLIB.REL, CREF80.ABS. If any of them are not listed in the directory, try to copy them from SY1: again. If they are not on SY1:, you probably have the wrong diskette mounted in SY1:. The FORTRAN distribution diskette should be mounted on SY1:. (Not the Sample Program diskette!)

( ) 5. Exit HDOS (HDOS command-BYE). Remove both diskettes from the disk drives.

You now have a configured MACRO-80 Assembler system diskette. This diskette contains the HDOS operating system as well as the MACRO-80 Assembler system files.

Procedure E - Copying the Sample Program Diskette

You will need one of the two bootable system diskettes you created in the previous procedures. You will also need the "Microsoft FORTRAN Sample Programs" distribution diskette, as well as the remaining one blank, initialized diskette.

( ) 1. Boot your system with either of the bootable system diskettes.

( ) 2. Using the HDOS utility PIP, mount the sample program distribution diskette on drive SY1:.

( ) 3. Now, using the HDOS utility PIP, dismount the diskette from drive SY0:.

( ) 4. Mount the blank, initialized diskette on drive SY0:.

( ) 5. Using the HDOS utility PIP, copy all the files from drive SY1: to drive SY0 :.

```
*.* = SY1:*.*
```

When you are finished, the following message will appear on your console device:

```
11 FILES COPIED
```

You have now copied all the sample programs to your working diskette. Now, exit PIP by typing CTRL D.

( ) 6. Remove both diskettes from the disk drives.

( ) 7. With a felt-tip pen, write the name "SAMPLE PROGRAM DISKETTE" on a label. Put the label on the diskette you removed from SY0:.

( ) 8. Put the distribution diskettes in a safe place.

You now have a complete set of working diskettes.

The next Chapter will instruct you on how to compile, link and execute FORTRAN programs. Several small but enjoyable sample programs are included.

Chapter Three

FORTRAN Compiler Operating Procedure

OVERVIEW

This Chapter will explain the procedure you must follow in order to create, compile, link and execute a FORTRAN program.

First, the overall process is illustrated and briefly explained. Then a step-by-step procedure is provided for you to follow. This procedure makes extensive use of the sample programs included with this software package.

The first sample illustrates the basic principles behind the process. After running this first example, you should understand most of the basic functions of the Compiler and the LINK-80 Loader.

The second sample program contains several obvious errors. You will use the Compiler to flag these errors. Next, you will correct these errors with your editor. You will then compile, link and execute the corrected program. This example will help illustrate some of the error messages generated by the Compiler.

The third sample program is written in several small modules. You will compile each of these modules, and then use the Linker to link all of them together. Finally, you will execute this program. After running this example, you should have an excellent understanding of the capabilities of both the Compiler and the Loader.

The FORTRAN-80 System

The procedure you must follow in order to execute a FORTRAN-80 program is: create the source program, compile, link and execute. You must perform each step in this process. The following illustration depicts this multi-step process.

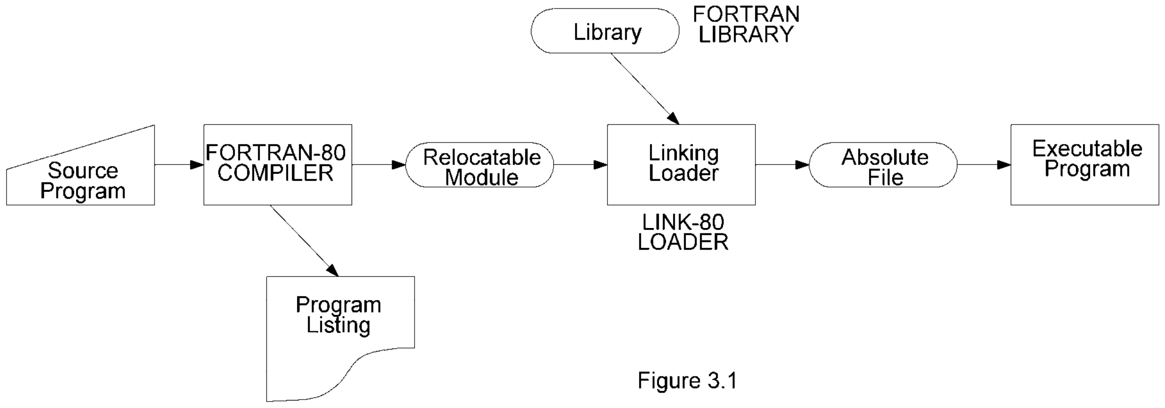


Figure 3.1
Steps in compiling and executing a FORTRAN-80 program.

Use a text editor to create the source program. The HDOS text editor EDIT is an example of an editor you can use to create a FORTRAN source program. Each line of the FORTRAN source program follows a rigorous, predefined format. This format is explained in the FORTRAN Reference Manual.

You can then compile the source program using the FORTRAN Compiler, which will check it for proper syntax. You can also use the Compiler to generate a hard copy listing, and an object file.

The object file generated by the Compiler is a relocatable module which you must then link using the LINK-80 Loader. The Loader will search the FORTRAN library to resolve any undefined references. After the references have been resolved, an absolute file can be generated.

This absolute file can be executed under HDOS.

TABLE SAMPLE PROGRAM

Compiling the First Sample Program

Now let's run through the first sample program. This program will generate a small table of trig functions that will be displayed on your console. Keep in mind that these sample programs are not intended to be examples of good FORTRAN programming. They will simply teach you how to use the FORTRAN system so you can write your own FORTRAN programs.

First, boot up your system using your FORTRAN system diskette. List a directory of the files on SY0: You will notice that you have very little, if any, room left on this diskette. Because of this, you will not want to store any of your FORTRAN program files on the system disk.

Then mount your working copy of the "Microsoft FORTRAN-80 Sample Programs" diskette on drive SY1:. List a directory of the files on SY1:. Make sure that the file TABLE.FOR is present: If you do not have this file on SY1:, you have mounted the wrong diskette.

The file TABLE.FOR is the FORTRAN source program. It was created with the HDOS text editor and it is ready to compile. Let's go ahead and compile this sample program.

You must first load the FORTRAN Compiler into memory to do this, type F80 and press RETURN. When the Compiler is ready for input, you will see an asterisk displayed on your screen. This is the prompt from the FORTRAN Compiler. At this point, you will supply a "command string" to the Compiler.

This command string tells the Compiler what to compile, where to put the output file, and what options to use during the compilation process. The options are input by using "switches". The various switches are explained in the FORTRAN Reference Manual. The general form of the command string is:

```
            output-file,listing-file=input-file/switches
```

The FORTRAN Compiler will provide the default file name extensions. The default extension for the output file is .REL, and the default extension for the input file is .FOR. The .REL means "relocatable file" and the .FOR means "FORTRAN source file". It is a good idea to adhere to these naming conventions.

The listing file can be a disk file, in which case the default extension will be .LST. The listing file can also be the console, in which case you type TT: as the listing device. You can even have the listing file be a hard copy device, in which case you supply the symbolic HDOS name of the hard copy device. For example, if you wish to list the file to an H14 line printer and your device driver name is LP.DVD, you type LP: as the listing file name.

If you do use a hard copy device for the listing file, you must remember to LOAD the device driver into memory. You can do this by typing the following HDOS command:

```
LOAD dev:
```

where dev: is the two letter symbolic name for the device driver. You must load the device driver before you invoke the FORTRAN Compiler.

To compile this sample program, type the following command string. (Do not type the asterisk, as this represents the prompt from the Compiler.):

```
*SY1:TABLE,TT:=SY1:TABLE
```

This command string tells the FORTRAN Compiler to compile the file TABLE.FOR. The listing will be written on the console and the output from the Compiler will be written in the file TABLE.REL. Both the input and the output files will reside on drive SY1:.

The Compiler will read the source file and the compilation process will begin. You will see the listing from the Compiler displayed on your console device. Pay special attention to the end of the listing. This will contain a symbol table of all symbols used in the program.

This symbol table can be an invaluable aid in your programming effort. It contains a list of all the subroutines, variables, and labels referenced by the FORTRAN source program. You will notice that some of the subroutines and labels are preceded with a dollar sign ($). These are internal symbolic references generated by the Compiler. The numbers following the variables and labels are the relative locations of these variables and labels.

When the compilation process is finished, an asterisk will again be displayed on your console device. To return to HDOS, type CTRL D. Then list a directory of the file SY1:TABLE.REL. This is the output file produced by the compilation process. This file is a relocatable module which must now be linked using the LINK-80 Loader. This relocatable module can not be executed in its present form.

Linking the First Sample Program

To invoke the LINK-80 Loader, type L80 and press RETURN. When the Loader is ready to accept an input file, an asterisk will be displayed on your console device.

The function of the Loader is to take the relocatable module created during the compilation process and resolve any external references that may exist. For example, the library function SIN is referenced in this sample program. The Loader will search the library FORLIB.REL for this function. After it locates the SIN function in the library, it will load the code needed in order to calculate the sine. The library FORLIB.REL must always reside on drive SY0:. The Loader will always search SY0: for the library. An additional function of the Loader is to generate an executable absolute file.

After the prompt from the Loader is displayed, "load" the relocatable module you created during the compilation process. To do this, type the following command string: (Do not type the asterisk, as this represents the prompt from the Loader.):

```
*SY1:TABLE
```

This command string tells the Loader to load the file SY1:TABLE.REL. Notice that the default extension .REL is supplied by the loader. After you type this command string, the Loader will load the relocatable file you previously created with the Compiler.

When the file has been loaded, the Loader will list the origin of the data area as well as the end of the program. This list will be in the following format:

```
DATA xxx yyy
```

The xxx represents the location of the beginning of the program. The yyy represents the location of the end of the program. In this example, both of these numbers will be in hexadecimal. The symbolic names of the unresolved references will be displayed followed by an asterisk "*".

You will want to search the library in order to resolve all of these references. The Loader will always search the library before exiting to HDOS. So you want to tell it to exit back to HDOS. You must also tell it what file to store the .ABS file in. Both of these steps can be performed with one command string. To save the .ABS file and exit to HDOS, type:

```
*SY1:TABLE/N/E
```

This will tell the Loader to save the absolute file in the' file SY1:TABLE.ABS. Notice that it will supply the extension .ABS. The /N switch is used to tell it where to store the output file. The /E switch is used to exit the Loader. But, before it exits to HDOS, it will search the FORTRAN library.

After you type the above command string, the Loader will search the library to solve all unresolved references. This will be a time consuming process. After all the references are solved, it will display the origin of the program area and the address of the next available byte on your console device. Then it will save the resulting absolute file. Finally, it will exit to HDOS.

At this point, you should list a directory of the file SY1:TABLE.ABS. Note that this is a much larger file than the source program. The .ABS file produced by the Loader will always take up at least 16 sectors on the diskette. These 16 sectors contain the initialization routines that allow the program to run under HDOS.

The command strings you supply to the Compiler or the Loader can be input at the same time they are invoked. When you use this method, the asterisk prompt is not displayed. For example, to compile the first sample program with this method, you would type the HDOS command:

```
>F80 SY1:TABLE,TT:=SY1:TABLE
```

Make sure you put a space between F80 and the command string. After the compilation process is complete, the Compiler will exit to HDOS. This method is useful when you wish to compile just one program.

To link the program, you would type the HDOS command:

```
>L80 SY1:TABLE,SY1:TABLE/N/E
```

The space between L80 and the command string is required.

When you use this method, make sure the command string you type is correct. If you do make a mistake, exit to HDOS by typing CTRL-D and then reinvoke the Compiler or Linker.

You have now compiled and linked the first sample program. If you need to review the process, go back and run through it again.

Executing the First Sample Program

Now you are ready to execute this first sample program. You can execute the absolute file by typing the following HDOS command. (Do not type the greater than ">" symbol, as this represents the prompt from HDOS.)

```
>RUN SY1:TABLE
```

The absolute file will be loaded into memory and executed. A small table of trig functions will be displayed on your console device as a result of executing this program.

Brief Review of the Process

The entire process can be summarized as follows:

1. Create the source program using a text editor.

   This is the first step in the process. Using a text editor, you will create the FORTRAN program. The format of the program must adhere to a rigorous, predefined syntax. The FORTRAN Reference Manual is the best place to refer when questions arise concerning the syntax of a particular statement.

2. Compile the source program using the FORTRAN Compiler.

   This step will take the source program you created in the previous step as input and the Compiler will check it for correct syntax. If the Compiler finds any errors, they will be displayed on your console. You should correct any errors before you proceed to the next step. The output from this step will be a relocatable module which you must now link using the LINK-80 Loader. An optional output is a listing of the program you are compiling.

3. Link the relocatable module using the LINK-80 Loader.

   This step will use the relocatable module you created in the previous step, and generate an absolute file that can be executed. During this step, all unresolved external references will be solved. The FORTRAN library will be searched to aid in this process. Finally, an absolute file will be saved on the disk.

4. Execute the absolute file.

   You can execute the absolute file in the same manner as you execute all absolute files. Use the HDOS command RUN.

ERROR SAMPLE PROGRAM

Compiling the Second Sample Program

The second sample program contains four errors that will be flagged by the FORTRAN Compiler. You will correct these errors and then recompile the program.

This sample does not contain as much tutorial information as the last sample. It is assumed that you retained the information from the last sample; thus, it does not have to be presented again. If you are still not sure how to compile, link and execute a FORTRAN program, you may wish to run through the previous sample again.

The following is a listing of the sample program:

```
              PROGRAM ERROR
      C THIS PROGRAM CONTAINS SEVERAL ERRORS
              DO 600, 1 = 1,10
              WRITE(1,100) I
         100  FORMAT(' ',THIS IS ITERATION NUMBER ",I2)
         600  CONTINU
              WRITE(1,150)
         150  FORMAT(' ',PLEASE INPUT AN INTEGER NUMBER ')
              READ(1,200) J
         200  FORMAT(I5)
              WRITE(1,175) J
         275  FORMAT(' ','THE INTEGER NUMBER IS `,I5)
              STOP
```

First, we will attempt to compile this program. Invoke the Compiler and type the following command string: (Do not type the asterisk, as this represents the prompt from the Compiler.)

```
*SY1:ERROR=SY1:ERROR
```

The Compiler will find the four errors in the program and display the appropriate error messages on your console device in the following manner:

```
?LINE: 3 STATEMENT UNRECOGNIZABLE OR MISSPELLED:DO 60

?LINE: 6 ILLEGAL HOLLERITH CONSTRUCTION:RATION NUMBER",I2)

?LINE: 7 STATEMENT UNRECOGNIZABLE OR MISSPELLED:CONTINU

?LINE: 15 PREMATURE END OF FILE ON INPUT DEVICE:STOP

?4 FATAL ERROR(S) DETECTED
```

Note that the line numbers do not represent the physical line number of the error. The line number represents the logical point at which the Compiler discovers the error.

The four errors in the above program are:

1. The DO statement contains an extra comma. The first comma is not allowed. You must delete this extra comma in order to correct this statement.

2. The FORMAT statement labeled 100 is incorrect. The literal contained in this statement must be enclosed in single quotes ('). The closing quote in this statement is a double quote ("). This double quote must be changed to a single quote in order to correct this statement.

3. The CONTINUE statement is not spelled correctly.

4. The last statement in a FORTRAN-80 source program must be an END statement. In this sample program, the END statement has been omitted. Insert an END statement to correct this problem.

Use your text editor to correct the above errors. Then write the corrected version of the source program to disk. Give this corrected version a different name than the original version.

Now you will want to recompile the corrected program. This time, the Compiler should not flag any errors. If you still have errors in the program, go back and make sure that you corrected all the errors listed above.

Now link and execute the program.

CALENDAR SAMPLE PROGRAM

Compiling the Third Sample Program

This sample program is somewhat different than the last two, it is written as a series of small modules. Each of these small modules will first be compiled. Then the Loader will be used to combine all of these small modules into one executable program.

One of these modules is written in assembly language. The MACRO-80 Assembler was used to assemble it and it will be combined with the other modules using the LINK-80 Loader.

This sample program will illustrate most of the features of the Loader. Because the first two sample programs were used to illustrate the Compiler, this sample will not contain much detailed information about it.

At this point, you should still have the FORTRAN system diskette in SY0: and the sample program diskette in SY1: You must now delete all of the absolute files on SY1: so that you will have enough room to compile and link this sample program. The HDOS command to delete all of the absolute files on SY1: is:

```
>DELETE SY1:*.ABS
```

Now invoke the FORTRAN Compiler by typing F80 and RETURN. When the asterisk prompt is displayed on your console, compile the source program CALENDAR.FOR. Use the following command string:

```
*SY1:CALENDAR=SY1:CALENDAR
```

Compile the next module:

```
*SY1:IDATE1=SY1:IDATE1
```

Compile the next module:

```
*SY1:IDATE3=SY1:IDATE3
```

Compile the last module:

```
*SY1:NPRINT=SY1:NPRINT
```

At this point you have compiled all of the modules you need for the main program. Type CTRL-D to exit to HDOS. When the HDOS prompt is displayed on your console, list a directory of the file SY1:NUMBER.REL. This file was created using the MACRO-80 Assembler. You can look at the source to this file by typing:

```
>TYPE SY1: NUMBER.MAC
```

Linking the Third Sample Program

You will be using the Loader to link all the programs you compiled above, plus this assembly language program. So invoke the Loader by typing L80 and RETURN.

After the asterisk is displayed on your console, load the first module: (Note: In this example, the order of loading is not significant.)

```
SY1:CALENDAR
```

After the module is loaded, you will notice a list of all the unresolved external references. You can solve these references by either loading the appropriate module or by searching the library. The Loader will always search the library before it exits to HDOS. Therefore, the most efficient way to solve these references is to load all the appropriate modules, and then exit to HDOS.

You should now load the next module:

```
*SY1:IDATE3
```

Load the next module:

```
*SY1:IDATE1
```

Load the next module:

```
*SY1:NPRINT
```

Load the final module:

```
*SY1:NUMBER
```

After this module has been loaded, a list of all the unresolved external references will be displayed. In order to resolve these references, you must search the library. Use the following command string, and then save the absolute file:

```
*SY1:CALENDAR/N/E
```

Executing the Third Sample Program

In order to execute the sample program, type the following HDOS command:

```
>RUN SY1:CALENDAR
```

This program will display a calendar on your console device.

OTHER SAMPLE PROGRAMS

The Sample Programs diskette contains several other FORTRAN programs that you can use to gain more "hands-on" experience. One of these is an Othello game, called OTHELLO.FOR. This is a rather large program, so make sure you have room on the diskette before you attempt to compile it.

If you want to delete all of the .ABS files on SY1: to allow room for this program, use the following HDOS command. (Do not type the greater than symbol ">"; this is the HDOS prompt.)

```
>DELETE SY1: *. ABS
```

Invoke the Compiler by typing F80 and RETURN. You can use the following command string to compile this sample program:

```
*SY1:OTHELLO=SY1:OTHELLO
```

This will take several minutes to compile, as this is a very large program. After compilation, link and execute this sample program. It plays a fairly decent game, so you may wish to take a handicap.

Another sample program, called MAZE.FOR, will draw a maze on your console. You may find it interesting to compile, link and execute this program. Some of the mazes it draws will trap you forever, so watch out!

The procedures listed in this chapter do not illustrate all of the capabilities of the FORTRAN system. There are more switches that you can use with both the Compiler and the Loader. So use the sample programs to experiment. Try some of the different switches of both the Compiler and Loader.

Note: Othello and Calendar, as well as their associated subroutines are public domain programs from the CP/M User's Group.

Chapter Four

FORTRAN I/O

OVERVIEW

This Chapter gives you a brief overview of FORTRAN I/O. Several of the important concepts are explained. When appropriate, sample programs are used for illustration.

The purpose of this Chapter is to acquaint you with the interface between HDOS and FORTRAN I/O. It is assumed that you already know how to write FORTRAN programs.

For a complete discussion of FORTRAN I/O, refer to Chapter 7 of the FORTRAN-80 Reference Manual.

FORTRAN I/O

FORTRAN uses the READ statement for input and the WRITE statement for output. The I/O statement may reference a FORMAT statement to control the editing and formatting of the data. This editing will be performed during the transmission of the data. If the I/O statement does not reference a FORMAT statement, the transmission of data is considered to be unformatted.

Logical Unit Numbers

The READ or WRITE statement will contain a reference to a Logical Unit Number that will represent a file or a device. Logical Unit Numbers 1-10 (inclusive) have certain HDOS files or devices assigned to them by default. Logical Unit Numbers 1 and 3-5 are assigned to the console device. Logical Unit Number 2 is assigned to the LP:, and Logical Unit Numbers 6-10 are assigned to disk files.

You can change the Logical Unit Number assignments by using the OPEN subroutine. Use the following general form to reference the OPEN subroutine:

```
CALL OPEN(lun, filename)
```

where:

```
lun is a positive integer or integer variable in the range 1-10.
filename is the HDOS file name
```

Chapter 7 of the FORTRAN-80 Reference Manual contains a more detailed discussion of Logical Unit Numbers.

Input/Output Lists

A FORTRAN I/O statement will usually contain a list of data items to be manipulated by the I/O statement. This I/O list contains the names of variables, arrays, and array elements whose values are to be transmitted. The I/O list can be of two types, the simple list and the implied DO list.

Random Access Files

FORTRAN-80 includes the facility for random access of a disk file. The length of a formatted random record must not be greater than 255 bytes. The length of an unformatted random record must not be greater than 256 bytes. For a complete discussion of random access files, refer to Chapter 7 of the FORTRAN-80 Reference Manual.

Formatted I/O

A formatted I/O statement will reference a FORMAT statement in order to specify data editing and formatting. This editing will take place during the transmission of the data. The transmission of data can be performed in either a sequential or random manner.

Formatted I/O has one very strict requirement that must be met. It requires that the amount of data transmitted be no greater than 255 bytes. This restriction applies to both random and sequential formatted files.

The results of a formatted READ or WRITE if the amount of data transmitted is greater than 255 cannot be predicted. This includes any carriage control character that may be transmitted with the record.

The carriage control character will always be transmitted as the first field of the formatted record. When you are writing to a disk file, you can omit the carriage control by using the plus (+) character. If you use the plus character, you will have access to all 255 bytes in the record.

If you use a one (1) or a blank ( )for the carriage control character, you will have access to 254 bytes. If you use a zero (0) for carriage control, you will have access to only 253 bytes. When you are writing to a disk file, it is usually best to suppress the carriage control by using a plus character (+).

Examples:

```
      WRITE(6,100) I
       .
       .
       .
  100 FORMAT('+',I6)
```

Note the use of the plus character. This will suppress the carriage control character.

```
          WRITE (6,150) J,K,L
           .
           .
           .
      150 FORMAT('0',3I6)
```

The first two bytes of this output record will be the carriage control characters.

The following program segments will illustrate the importance of the carriage control character in formatted I/O.

In the following segment we will suppress the carriage control with the plus character (+).

```
            BYTE BUFF(160) DATA
            BUFF/160*'A'/ WRITE(6,100)
            BUFF
       100  FORMAT('+',160A1)
            ENDFILE 6
            STOP
            END
```

After the above program segment is executed; a hexadecimal dump of the file FORT06.DAT (default file name for lun 6) would look similar to this:

SAMPLE DUMP OF OUTPUT FILE:

```
      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0000: 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
0010: 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
0020: 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
0030: 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
0040: 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
0050: 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
0060: 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
0070: 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
0080: 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
0090: 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
00A0: 0A 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00B0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00C0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00D0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00E0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00F0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
```

The dump reveals that no carriage control character was stored with the record. The 160 byte' array BUFF is written to the disk, followed by the end of record mark OA. Because each HDOS sector is 256 bytes in length, this record will be padded with ASCII NUL characters (y)).

Of these 256 bytes, your FORTRAN file can use only 255. One byte is reserved for the end of record mark (Hex--OA).

The FORTRAN I/O processor will always look for this end of record mark when it is reading a file. Therefore, a formatted READ can only read a file' created with a formatted WRITE.

This next program segment will use a zero (0) for the carriage control character. Now, the carriage control will be stored as the first two bytes of the, record.

```
            BYTE BUFF(160)
            DATA BUFF/160*'A'/
            WRITE(6,100) BUFF
      100   FORMAT('0',160A1)
            ENDFILE 6
            STOP
            END
```

After the above program segment is executed, a hexadecimal dump of the. file FORT06.DAT (default file name for lun 6) would look similar to' this:

```
 SAMPLE DUMP OF OUTPUT FILE:

       0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0000: 0D 0D 41 41 41 41 41 41 41 41 41 41 41 41 41 41 ..AAAAAAAAAAAAAA
0010: 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
0020: 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
0030: 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
0040: 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
0050: 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
0060: 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
0070: 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
0080: 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
0090: 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
00A0: 41 41 0A 00 00 00 00 00 00 00 00 00 00 00 00 00 AA..............
00B0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00C0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00D0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00E0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00F0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
```

The first two bytes of this record are used by the carriage control character. After the carriage control character is the 160-byte array BUFF. Following the data is the end of record mark OA. This record will also be padded with ASCII NUL characters.

When this file is read, the first two bytes must be skipped, or the carriage control character will be read as data. This will produce unpredictable results.

Unformatted I/O

Unformatted I/O will transmit data without any editing or formatting.

The unformatted sequential record may be any length you desire. However, it is very important to keep the length of the input record the same as the length of the output record. An attempt to input longer records than were output will result in a premature end of file. The result of inputting a shorter record than was output will be an unusable record.

The unformatted random record may not be greater than 256 bytes. An attempt to READ more than 256 bytes will result in the same record being read as many times as needed to fill the I/O list. An attempt to WRITE more than 2 56 bytes will result in an unusable record.

This next program segment will write a 160 byte unformatted sequential record.

```
BYTE BUFF(160)
DATA BUFF/160*'A'/
WRITE(6) BUFF
ENDFILE 6
STOP
END
```

After the above program segment is executed, a hexadecimal dump of the file FORT06.DAT (default file name for lun 6) would look similar to this:

SAMPLE DUMP OF OUTPUT FILE:

```
       0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0000: 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
0010: 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
0020: 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
0030: 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
0040: 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
0050: 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
0060: 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
0070: 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
0080: 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
0090: 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
00A0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00B0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00C0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00D0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00E0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00F0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
```

The 160-byte array BUFF is written to the disk file. Notice that no end of record character is stored with the disk file. Also, the record will be padded with ASCII NUL characters (0).

Using a Hard Copy Device with FORTRAN-80

FORTRAN will use the HDOS device driver to communicate with a hard copy device. Logical Unit Number 2 has been preassigned to the HDOS device driver LP.DVD. If you name your device driver LP.DVD, then a reference in your FORTRAN program to LUN 2 will produce hard copy output.

You must remember to load the device driver into memory before you execute the FORTRAN program. The following HDOS command will load the device driver into memory:

```
>LOAD dev:
```

where:

dev: is the two letter symbolic name for the device driver.

For example, if you wish to load the device, driver AT.DVD into memory, use the following HDOS command:

```
>LOAD AT:
```

You can also use the OPEN subroutine to assign a different LUN to a hard copy device. For example, if you wish to assign the LUN 10 to a device driver named DB.DVD, use the following FORTRAN statement:

```
CALL OPEN(10,'DB: ')
```

Of course you must still load the device driver into memory before you attempt to write to LUN 10.

To use a hard copy device with the FORTRAN-80 Compiler or the MACRO-80 Assembler, you must first load the device driver into memory. To generate a hard copy listing, specify the proper device name in the command string.

Chapter **Five**

User's Guide to the FORTRAN Library Functions

OVERVIEW

This Chapter will give you an overview of all the library functions available to the FORTRAN user. Each function is briefly explained and a sample program segment is provided to illustrate the proper format for referencing the function. Several of the functions are of interest for scientific and engineering applications.

FORTRAN-80 LIBRARY FUNCTIONS

Absolute Value Function

FORM	DEFINITION	ARGUMENT TYPE	RESULT TYPE
ABS(X)	Real absolute value	Real	Real
IABS(X)	Integer absolute value	Integer	Integer
DABS(X)	DPabsolute value '	Double	Double

The absolute value functions simply return the value of the argument without a + or - **sign.**

Examples:

```
C EXAMPLE OF REAL ABSOLUTE VALUE
C
C
      ARG = -1234.567
      RESULT = ABS(ARG)


C EXAMPLE OF INTEGER ABSOLUTE VALUE
C
C
      IARG = -3450
      K = IABS(IARG)


C EXAMPLE OF DOUBLE PRECISION ABSOLUTE VALUE
C
C
      DOUBLE PRECISION DPARG,DPRES
      DPARG = -234.123456789
      DPRES = DABS(DPARG)
```

Truncation Function

FORM	DEFINITION	ARGUMENT TYPE	RESULT TYPE
AINT(X)	Real to Real truncation	Real	Real
INT(I)	Real to Integer truncation	Real	Integer
IDINT(Z)	DP to Integer truncation	Double	Integer

These are the truncation functions. They will return:

sign of the argument (+ or -) * largest integer <= absolute value of (arg)

Examples:

```
C EXAMPLE OF REAL TO REAL TRUNCATION
C
C
      X = 3425.1234
      Y = AINT(X)


C EXAMPLE OF REAL TO INTEGER TRUNCATION
C
C
      X = -456.1234
      I = INT(X)


C EXAMPLE OF DOUBLE PRECISION TO INTEGER TRUNCATION
C
C
      DOUBLE PRECISION Z
      Z = 2345.123456
      I = IDINT(Y)
```

Remainder Function

FORM	DEFINITION	ARGUMENT TYPE	RESULT TYPE
AMOD(V,X)	Real remainder	Real	Real
MOD(I,j)	Integer remainder	Integer	Integer
DMOD(Y,Z)	DP remainder	Double ,,	Double

These are the remainder functions. They will return the remainder when the first argument is divided by the second.

```
C EXAMPLE OF REAL REMAINDER
C
C
      V=125.1234
      X=25.0
      Y = AMOD(V,X)


C EXAMPLE OF INTEGER REMAINDER
C
C
      I   25
      J = 2
      K = MOD(I,J)


C EXAMPLE OF DOUBLE PRECISION REMAINDER
C
C
      DOUBLE PRECISION X,Y,Z
      X = 100.1234567
      Y = 10.0
      Z = DMOD(X,Y)
```

Maximum Value Function

FORM	DEFINITION	ARGUMENT TYPE	RESULT TYPE
AMAX0(I,J,...)	Real max from Integer list	Integer	Real
AMAX1(V,X,...)	Real max from Real list	Real	Real
MAX0(I,J,...)	Integer max from Integer list	Integer	Integer
MAX1(V,X....)	Integer max from Real list	Real	Integer
DMAX1(Y,Z....)	Double max from Double list	Double	Double

The above functions return the maximum value from among the argument list. There must be at least two arguments. Also note that the result type is, in several cases, different than the data type of the argument list.

```
C REAL MAX FROM INTEGER LIST EXAMPLE
C
       I = 10
       J  = 23
       X = AMAX0(I,J)

C REAL MAX FROM REAL LIST EXAMPLE
C
       V = 234.12
       X = 7345.1
       Z = 1.67
       Y = AMAX1(V,X,Z)

C INTEGER MAX FROM INTEGER LIST EXAMPLE
C
       I=1
       K=90
       J = MAX0(I,K)

C INTEGER MAX FROM REAL LIST EXAMPLE
C
       A = 100.145
       B = .123
       C = 908.23
       D = 44.1
       I = MAX1(A,B,C,D)

C DOUBLE MAX FROM DOUBLE LIST EXAMPLE
C
       DOUBLE PRECISION X,Y,Z
       X = 7823.145
       Y = 567.456677
       Z = DMAX1(X,Y)
```

Minimum Value Function

FORM	DEFINITION	ARGUMENT TYPE	RESULT TYPE
AMIN0(I,J....)	Real min of Integer list	Integer	Real
AMIN1(V,X,...)	Real min of Real list	Real	Real
MIN0(I,J....)	Integer min of Integer list	Integer	Integer
MIN1(V,X,...)	Integer min of Real list	Real	Integer
DMIN1(Y,Z,...)	Double min of Double list	Double	Double

The above functions return the minimum value from among the elements of the argument list. There must be at least two arguments. Also note that the data type of the result is, in several cases, different from the data type of the argument list.

```
C REAL MIN FROM INTEGER LIST EXAMPLE
C
      I = -1
      K = 45
      X = AMIN0(I,K)

C REAL MIN FROM REAL LIST EXAMPLE
C
      V = 234.12
      X = 7345.1
      Z = 1.67
      Y = AMIN1(V,X,Z)

C INTEGER MIN FROM INTEGER LIST EXAMPLE
C
      I = 1
      K = 90
      J = MIN0(I,K)

C INTEGER MIN FROM INTEGER LIST EXAMPLE
C
      A = 100.145
      B = .123
      C = 908.23
      D = 44.1
      I = MIN1(A,B,C,D)

C DOUBLE MIN FROM DOUBLE LIST EXAMPLE
C
      DOUBLE PRECISION X,Y,Z
      X = 7823.145
      Y = 567.456677
      Z = DMIN1(X,Y)
```

Conversion Function

FORM	DEFINITION	ARGUMENT TYPE	RESULT TYPE
FLOAT(I)	Integer to Real conversion	Integer	Real
IFIX(X)	Real to Integer conversion	Real	Integer
SNGL(Z)	Double to Real conversion	Double	Real
DBLE(X)	Real to Double conversion	Real	Double

The functions listed above perform conversions between the various data types. The result of the conversion must produce a value that conforms to the range of
the data type of that result.

For example, if a value is to be converted to the integer data type using the IFIX function, the result of this conversion must be a value in the range of -32768 to +32767 inclusive.

```
C INTEGER TO REAL CONVERSION EXAMPLE
C
      I = 123
      X = FLOAT(I)


C REAL TO INTEGER CONVERSION EXAMPLE
C
      X = 456.1234 I = IFIX(X)


C DOUBLE TO REAL CONVERSION EXAMPLE
C
      DOUBLE PRECISION Z Z=234.1234567890 R = SNGL(Z)


C REAL TO DOUBLE CONVERSION EXAMPLE
C
      DOUBLE PRECISION Z
      R = 451.123456
      Z = DBLE(R)
```

Transfer of Sign Function

FORM	DEFINITION	ARGUMENT TYPE	RESULT TYPE
SIGN(X,Y)	Real transfer of sign	Real	Real
ISIGN(I,J)	Integer transfer of sign	Integer	Integer
DSIGN(Y,Z)	DP transfer of sign	Double	Double

The transfer of sign functions require that two arguments be supplied. These functions will return:

(sign of second argument [+ or -]) * (absolute value of the first argument).

```
C REAL TRANSFER OF SIGN EXAMPLE
C
C
      X = -100.0
      Y = 2.0
      R = SIGN(Y,X)


C INTEGER TRANSFER OF SIGN EXAMPLE
C
C
      I = -20
      J  =  -5
      K = ISIGN(I,J)


C DOUBLE PRECISION TRANSFER OF SIGN EXAMPLE
C
C
      DOUBLE PRECISION A,B,C
      A = -729.12345678
      B = 10.0
      C = DSIGN(A,B)
```

Positive Difference Function

FORM	DEFINITION	ARGUMENT TYPE	RESULT TYPE
DIM(V,X)	Real positive difference	Real	Real
IDIM(I,J)	Integer positive difference	Integer	Integer

The positive difference functions require that two arguments be supplied. These functions will return the first argument minus the minimum of the two arguments.

```
C REAL POSITIVE DIFFERENCE EXAMPLE
C
C
      X = 300.0
      Y = -200.0
      R = DIM(X,Y)


C INTEGER POSITIVE DIFFERENCE
C
C
      I = 10
      J = -10
      K = IDIM(I,J)
```

Exponential Function

FORM	DEFINITION	ARGUMENT TYPE	RESULT TYPE
EXP(X)	e raised to the power of X	Real	Real
DEXP(Y)	e raised to the power of Y	Double	Double

The above functions will compute the natural logarithm's base value e (2.71828) raised to the power of the supplied argument.

For example:

```
B = EXP(4)
```

is equivalent to:

```
B = 2.71828 * 2.71828 * 2.71828 * 2.71828
```

```
C REAL EXPONENTIAL FUNCTION EXAMPLE
C
C
      X=10.0
      Y = EXP(X)


C DOUBLE PRECISION EXPONENTIAL FUNCTION EXAMPLE
C
C
      DOUBLE PRECISION X,Y
      X = 5.0
      Y=EXP(X)
```

Logarithm Function

FORM	DEFINITION	ARGUMENT TYPE	RESULT TYPE
ALOG(X)	natural log of X ln(X)	Real	Real
DLOG(Y)	natural log of Y ln(Y)	Double	Double
ALOG10(X)	common log of X log10(X)	Real	Real
DLOG10(Y)	common log of Y log10(Y)	Double	Double

The ALOG and DLOG functions will compute the natural logarithm of the supplied argument.

The ALOG10 and DLOG10 functions will compute the common logarithm of the supplied argument. The argument must be greater than zero.

To convert a common log to a natural log, multiply the common log x 2.3026. To convert a natural log to a common log, multiply the natural log x .434295.

```
C REAL NATURAL LOG EXAMPLE
C
C
        X = 100.0
        Y = ALOG(X)


C DOUBLE PRECISION NATURAL LOG EXAMPLE
C
C
        DOUBLE PRECISION A,B
        A = 100.O
        B = DLOG(A)


C REAL COMMON LOG EXAMPLE
C
C
        X = 100.0
        Y = ALOG10(X)


C DOUBLE PRECISION COMMON LOG EXAMPLE
C
C
        DOUBLE PRECISION A,B
        A = 100.0
        B = DLOG10(A)
```

SINE and COSINE Function

FORM	DEFINITION	ARGUMENT TYPE	RESULT TYPE
SIN(X)	sine of X	Real	Real
DSIN(Y)	sine of Y	Double	Double
COS(X)	cosine of X	Real	Real
DCOS(Y)	cosine of Y	Double	Double

The SIN and DSIN functions will compute the sine of the angle supplied as the argument. The COS and DCOS functions will compute the cosine of the angle supplied as the argument.

The argument must be expressed in radians (not degrees). To convert degrees to radians, multiply the angle in degrees x .0174533. To convert radians to degrees, multiply the angle in radians x 57.29578.

```
C EXAMPLE OF REAL SINE COMPUTATION
C
C
       X = 1.0
       Y = SIN(X)


C EXAMPLE OF DOUBLE PRECISION SINE COMPUTATION
C
C
       DOUBLE PRECISION A,B
       A = 1.0
       B = DSIN(A)


C EXAMPLE OF REAL COSINE COMPUTATION
C
C
       X = 2.0
       Y = COS(X)


C EXAMPLE OF DOUBLE PRECISION COSINE COMPUTATION
C
C
       DOUBLE PRECISION A,B
       A = 2.0
       B = DCOS(A)
```

Hyberbolic Tangent Function

FORM	DEFINITION	ARGUMENT TYPE	RESULT TYPE
TANH(X)	hyberbolic tangent of X	Real	Real

The hyberbolic tangent is defined as:

```
TANH(X) = (EXP(X)-EXP(-X)/EXP(X)+EXP(-X)
```

You can calculate several other hyberbolic functions using the EXP function. These functions are:

```
SINH(X) = (EXP(X)-EXP(-X))/2
COSH(X) = (EXP(X)+EXP(-X))/2
SECH(X) = 2/(EXP(X)+EXP(-X))
CSCH(X) = 2/(EXP(X)-EXP(-X))
COTH(X) = (EXP(X)+EXP(-X))/(EXP(X)-EXP(-X))
```

Arctangent Function

FORM	DEFINITION	ARGUMENT TYPE	RESULT TYPE
ATAN(X)	Real arctangent	Real	Real
DATAN(Z)	Double arctangent	Double	Double
ATAN2(V,X)	Real arctangent of (V/X)	Real	Real
DATAN2(Y,Z)	Double arctangent of (Y/Z)	Double	Double

The functions listed in this table compute the arctangent (in radians) of the ratio of the sides supplied as arguments.

The ATAN and DATAN functions have only one argument, the ratio of the two sides.

The ATAN2 and DATAN2 functions require that the actual lengths of the two sides (the side opposite and the side adjacent) be supplied as arguments.

```
C EXAMPLE OF REAL ARCTANGENT COMPUTATION
C
C
       X = 2.0
       Y = ATAN(X)

C EXAMPLE OF DOUBLE PRECISION ARCTANGENT COMPUTATION
C
C
       DOUBLE PRECISION A,B
       A = 2.0
       B = DATAN(A)

C EXAMPLE OF REAL ARCTANGENT COMPUTATION
C
C
       X = 10.0
       Y = 2.0
       Z = ATAN2(X,Y)

C EXAMPLE OF DOUBLE PRECISION ARCTANGENT COMPUTATION
C
C
       DOUBLE PRECISION A,B,C
       A = 10.0
       B = 2.0
       C = DATAN2(A,B)
```

Square Root Function

FORM	DEFINITION	ARGUMENT TYPE	RESULT TYPE
SQRT(X)	Square root of X	Real	Real
DSQRT(Z)	Square root of Z	Double	Double

The functions listed in the preceding table will compute the square root of the supplied argument. The supplied argument must be a positive number.

```
C EXAMPLE OF REAL SQUARE ROOT FUNCTION
C
C
       X = 9
       Y = SQRT(X)


C EXAMPLE OF DOUBLE PRECISION SQUARE ROOT FUNCTION
C
C
       DOUBLE PRECISION A,B
       A = 9.0
       B = DSQRT(A)
```

Special Functions

FORM	**DEFINITION**
PEEK(a)	Returns contents of decimal location a.
CALL POKE(a,b)	Replace contents of location a with value b.
INP(a)	Returns 8-bit input value from port a (decimal).
CALL OUT(a,b)	Outputs value of b to port a (decimal).

PEEK and INP are logical functions. The value returned to these functions will be in the range -127 to +127. You must add 256 to the negative value to calculate the actual decimal value. For example, if the result of a PEEK operation is -10, this is actually the decimal value 246.

POKE and OUT are integer subroutines. The OUT subroutine is used to output an 8-bit value to a specific output port. HDOS has a vested interest in several of the output ports. Refer to the HDOS manual for a complete discussion of the ports reserved for HDOS.

POKE is used to output an 8-bit value to a specific memory location. The address must be in the range 0-32767 inclusive. To POKE or (PEEK) an address greater than 32767, use the following formula:

```
-1 * (65536 - desired address) = POKE or PEEK address
```

Other Functions

You may calculate some functions that are not included in the FORTRAN-80 Library as follows:

FUNCTION	FORTRAN EQUIVALENT
TANGENT	TAN(X)=SIN(X)/COS(X)
SECANT	SEC(X)=1/COS(X)
COSECANT	CSC(X)=l/SIN(X)
COTANGENT	COT(X)=l/TAN(X)
INVERSE SINE	ARCSIN(X) =ATN(X/SQRT(-X+ 1))
INVERSE COSINE	ARCCOS(X) = -ATN(X/SQRT(-X*X+1))+1.570796
INVERSE COTANGENT	ARCCOT(X)= -ATN(X)+1.5708

FORTRAN-80 LIBRARY SUBROUTINES

Arithmetic Routines

The FORTRAN-80 library contains a number of subroutines that you can use. In the following descriptions, $AC is the address of the low byte of the mantissa, and it refers to the floating accumulator. $AC+3 is the address of the exponent. $DAC is the address of the low byte of the mantissa and it refers to the double precision accumulator. $DAC+7 is the address of the double precision exponent.

All arithmetic routines adhere to the following calling conventions:

1. Argument 1 is passed as follows:
    1. Integer in HL.
    2. Real in $AC.
    3. Double in $DAC.
2. Argument 2 is either passed in registers or in memory, depending upon the data type:
    1. Integers are passed in HL, or DE if HL contains argument 1.
    2. Real and double-precision values are passed in memory pointed to by HL. (HL points to the low byte of the mantissa.)

The following arithmetic routines are contained in the library:

FUNCTION	NAME	ARGUMENT 1 TYPE	ARGUMENT 2 TYPE
Addition	$AA	Real	Integer
	$AB	Real	Real
	$AQ	Double	Integer
	$AR	Double	Real
	$AU	Double	Double
Division	$D9	Integer	Integer
	$DA	Real	Integer
	$DB	Real	Real
	$DQ	Double	Integer
	$DR	Double	Real
	$DU	Double	Double
Exponentation	$E9	Integer	Integer
	$EA	Real	Integer
	$EB	Real	Real
	$EQ	Double	Integer
	$ER	Double	Real
	$EU	Double	Double
Multiplication	$M9	Integer	Integer
	$MA	Real	Integer
	$MB	Real	Real
	$MQ	Double	Integer
	$MR	Double	Real
	$MU	Double	Double
Subtraction	$SA	Real	Integer
	$SB	Real	Real
	$SQ	Double	Integer
	$SR	Double	Real
	$SU	Double	Double

Conversion Subroutines

Additional library routines are provided for converting between data types. Arguments are passed to and returned in the following registers:

1. Logical in A.

2. Integer in HL.

3. Real in $AC.

4. Double in $DAC.

NAME	FUNCTION
$CA	Integer to Real
$CC	Integer to Double
$CH	Real to Integer
$CJ	Real to Logical
$CK	Real to Double
$CX	Double to Integer
$CY	Double to Real
$CZ	Double to Logical

SUBPROGRAM LINKAGES

The following information is included for those of you who wish to know the format of the subprogram calls generated by the FORTRAN-80 Compiler. Most users will want to skip this section, as it does not contain material necessary to the operation of the package.

This section defines a normal subprogram call as generated by the FORTRAN-80 Compiler. It is included to facilitate linkages between FORTRAN programs and programs written in other languages, such as 8080 assembly.

A subprogram reference with no parameters generates a simple "CALL" instruction. The corresponding subprogram should return via a simple "RET". (CALL and RET are 8080 opcodes - consult the HDOS Reference Manual for more information about the 8080 opcodes.)

A subprogram reference with parameters results in a somewhat more complex calling sequence. Parameters are always passed by reference. (The entity passed is actually the address of the low byte of the argument.) Therefore, parameters will always occupy two bytes each, regardless of the actual data type.

The method of passing the parameters depends upon the number of parameters to be passed:

1. If the number of parameters is less than or equal to three, they are passed in the registers. Parameter one will be in HL, two in DE (if present), and three in BC (if present).

2. If the number of parameters is greater than three, they are passed as follows:

    1. Parameter one in HL.

    2. Parameter two in DE.

    3. Parameters three through n in a contiguous data block. BC will point to the low byte of the data block.

Note that, with this scheme, the subprogram must know how many parameters to expect in order to find them. Conversely, the calling program is responsible for passing the correct number of parameters. Neither the Compiler nor the runtime system checks for the correct number of parameters.

If the subprogram expects more than three parameters and needs to transfer them to a local data area, there is a system subroutine which will perform this transfer. The argument transfer routine is named $AT, and is called with HL pointing to the local data area, BC pointing to the third parameter, and A containing the number of arguments to transfer (the total number of arguments minus two). The subprogram is responsible for saving the first two parameters before calling $AT. For example, if a subprogram expects five parameters, it should use the following procedure:

```
SUBR: SHLD  P1    ;SAVE PARAMETER 1 XCHG
      SHLD  P2    ;SAVE PARAMETER 2
      MVI   A,3   ;NO. OF PARAMETERS LEFT
      LXI   H,P3  ;POINTER TO LOCAL AREA
      CALL  $AT   ;TRANSFER THE OTHER 3 PARAMETERS
            .
            .
            .
         [BODY OF SUBPROGRAM]
            .
            .
            .
      RET         ;RETURN TO SENDER
P1:   DS    2     ;SPACE FOR PARAMETER 1
P2:   DS    2     ;SPACE FOR PARAMETER 2
P3:   DS    6     ;SPACE FOR PARAMETERS 3-5
```

When you are accessing parameters in a subprogram, remember that they are only pointers to the actual arguments passed.

It depends entirely upon the programmer to insure that the arguments in the calling program match in number, type, and length with the parameters expected by the subprogram. This applies to FORTRAN subprograms as well as those written in assembly language.

FORTRAN Functions return their values in registers or memory depending upon the data type of the result. Logical results are returned in A. Integer results are returned in HL, reals in memory at $AC, and double-precision in memory at $DAC. $AC and $DAC are the addresses of the low bytes of the mantissas.

Chapter Six

MACRO-80 Assembler Operating Procedure

OVERVIEW

This chapter will explain the procedure you must follow in order to create, assemble, link, and execute a MACRO-80 program. It will not teach you assembly language programming, but it will illustrate how to use the MACRO-80 Assembler with HDOS. A sample program has been included.

The interface to HDOS via the SCALL is also briefly illustrated. Consult your HDOS Reference Manual for detailed information about using the SCALL.

MACRO-80 ASSEMBLER

To execute a MACRO-80 program, you must: 1.) create the source program, 2.) assemble, 3.) link, and 4.) execute. You must perform each step in this process. Figure 6-1 shows this multi-step process.

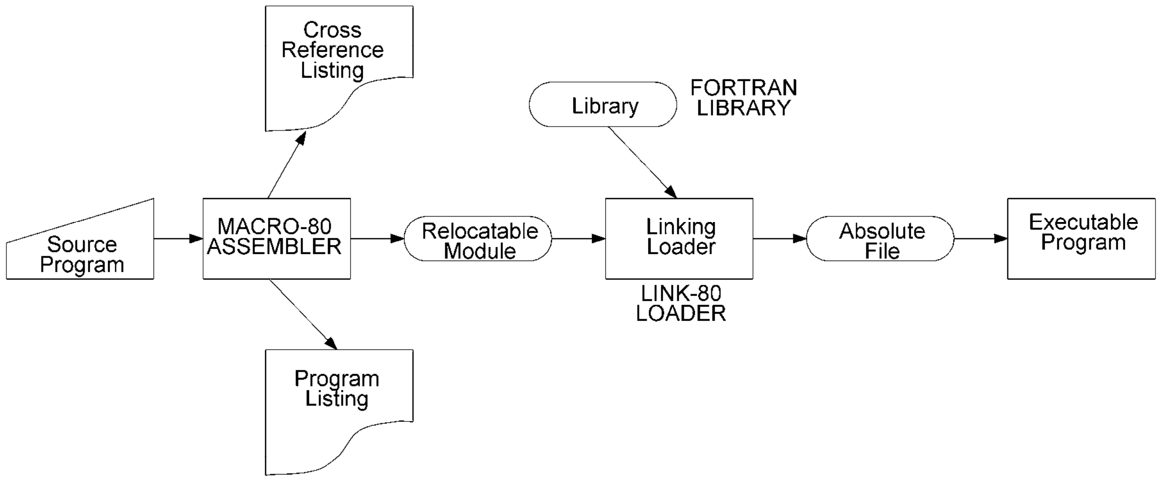


Figure 6.1
Steps in compiling and executing a FORTRAN-80 program.

Use a text editor to create the source file. The HDOS editor EDIT is an example of a text editor suitable for creating MACRO-80 programs.

You must then assemble the source program using the MACRO-80 Assembler. It will check the program for proper syntax and generate a hard copy listing. The Assembler will also generate an object file.

The MACRO-80 Assembler includes a cross reference facility. You can use the cross reference program CREF80 to generate a cross reference listing.

The object file generated by the assembler is a relocatable module which you must then link using the LINK-80 Loader. The Loader will generate an absolute file, and it can also be used to link the main program to subroutines contained in the FORTRAN library.

The absolute file created by the LINK-80 Loader can then be executed under HDOS.

DEMO SAMPLE PROGRAM

Assembling the Sample Program

The sample program, a very simple one that will print a message on your console, uses the HDOS PRINT SCALL. For more information about using the HDOS SCALL, refer to your HDOS Reference Manual.

The first thing you should do is to boot up your system using your MACRO-80 system diskette. List a directory of the files on SY0:. You will notice that you have enough room on this diskette to store several small programs. If you write any large programs, you will not want to store them on the system diskette.

Then mount your working copy of the "Microsoft FORTRAN-80 Sample Programs" diskette on drive SY1:. List a directory of the files on SY1:. Make sure the file DEMO.MAC is present. This is the MACRO-80 source program. It was created with the HDOS text editor and it is ready to assemble. If you do not have this file on SY1:, you have mounted the wrong diskette.

Now load the MACRO-80 Assembler into memory. To do this, type M80 and a press RETURN. When the Assembler is ready for input, you will see an asterisk displayed on your screen. This is the prompt from the Assembler. You must now supply a "command string."

The command string tells the Assembler what to assemble, where to put the output file, and what options to use during the assembly process. You can input the options by using the various "switches" explained in the MACRO-80 Reference Manual. The general form of the command string is:

```
output-file,listing-file=input-file/switches
```

If you omit the file name extensions, they will be provided by the Assembler. The default extension for the output file is REL and the default extension for the input file is MAC. The REL means "relocatable file" and the MAC means "MACRO-80 source file." You should adhere to these naming conventions.

The listing file can be a disk file, in which case the default extension will be .LST. On the listing file can be the console, in which case you type TT: as the listing device. You can also use a hard copy device for the listing file, in which case you supply the symbolic HDOS name of the hard copy device.

If you do use a hard copy device for the listing file, you must remember to LOAD the device driver into memory. This is done with the HDOS command:

```
>LOAD dev:
```

Where dev: is the two letter symbolic name for the device driver. If you wish to generate a hard copy listing, you must load the device driver before you invoke the Assembler.

To assemble the sample program, type the following command string: (Do not type the asterisk as this represents the prompt from the Assembler.):

```
*SY1:DEMO=SY1:DEMO
```

This tells the Assembler to assemble the file DEMO.MAC. The output from the' assembly process will be written to the file DEMO.REL. Both the input and the output files will reside on drive SY1:.

The Assembler will read the source file and the assembly process will begin. When the assembly process: is finished, the following message should appear on your console:

```
NO FATAL ERROR(S)
```

The asterisk prompt from the Assembler will then appear on your console. To return to HDOS, type CTRL-D. Then list a directory of the file SY1:DEMO.REL, the output file produced by the Assembler. This file is a relocatable module which you must now link using the LINK-80 Loader. The relocatable module can not be executed in its present form.

Linking the Sample Program

To invoke the Loader, type L80 and press RETURN. When the Loader is ready to accept an input file, an asterisk will be displayed on your console.

The Loader can be used to link a function from the FORTRAN library to an assembly language program. It can also be used as we will use it in this sample, to generate an absolute file.

When you see the prompt from the Loader, "load" the relocatable module you created during the assembly process. To do this, type the following command string: (Do not type the asterisk, as this represents the prompt from the Loader.):

```
*SY1:DEMO
```

This tells the Loader to load the file SY1:DEMO.REL. Notice that the default extension is supplied by the Loader. After you type this command string, it will load the relocatable module you previously created with the Assembler.

After the file has been loaded, the Loader will list the origin of the data area as well as the end of the program area. This list will be in the following format:

```
DATA xxx yyy
```

The xxx represents the location of the beginning of the data area. The yyy represents the location of the end of the program. Both of these numbers will be in hexadecimal.

Next, we will want to tell the Loader where to put the absolute file. (Use the N switch to supply a name for the absolute file). We also want to tell the Loader to exit back to HDOS. Use the /E switch and the command string:

```
*SY1:DEMO/N/E
```

The Loader will save the absolute file in the file SYI:DEMO.ABS. Notice that it will supply the extension ABS. Also note that we did not search the library to resolve any undefined references as we did with FORTRAN. This step was not necessary because we did not reference any of the functions in the FORTRAN library. Chapter 5, "User's Guide to the FORTRAN Library Functions, contains a list of the functions in the library, as well as the format of the subprogram linkages.

Executing the Sample Program

You can execute the sample program with the HDOS command:

```
            >RUN SY1:DEMO
```

The absolute file will then be loaded into memory and executed. A message will appear on your console as a result of executing this program.

Using the Cross Reference Facility

The cross reference facility will generate a special listing file that can be an important diagnostic tool. Assume, for example, that your program uses a field called FIELD1, and that program testing reveals an error in the manipulating of this field. You could then use the cross reference listing to check every instruction that references this field.

To use the facility, you must first use the /C switch to tell the Assembler to produce a special listing file. To produce the special listing file with the DEMO program, you should invoke the Assembler by typing M80 and RETURN. Then supply the Assembler with the command string: (Do not type the asterisk, as this represents the prompt from the Assembler.):

```
*SY1:DEMO=SY1:DEMO/C
```

When the Assembler encounters the /C switch in the command string, it will open a special listing file with the default extension .CRF. After it is finished, you must exit to HDOS by typing CTRL D. Then invoke the cross reference facility by typing CREF80. The command string for this facility is of the following general form: (Do not type the asterisk as this represents the prompt from the Assembler.):

```
*listing file=source file
```

To obtain a cross reference listing of the DEMO program, use the command string: (Do not type the asterisk as this represents the prompt from the Assembler.):

```
*SY1:DEMO=SY1:DEMO
```

This will create a cross reference listing of the DEMO program and put it in the file SY1:DEMO.LST. When the cross reference program has finished, the asterisk will again appear on your console. At this point, type CTRL-D to return to HDOS. Now, you can use PIP to copy this file to a hard copy device or your console.

Note that cross reference listings differ from ordinary listings in the following ways:

1. Each source statement is numbered with a cross reference number.

2. At the end of the listing, variable names appear in alphabetical order along with the numbers of the lines in which they are referenced or defined. Line numbers in which the symbol is defined are flagged with a pound sign (#).

USING THE HDOS SCALL

The MACRO-80 Assembler requires the HDOS SCALL to be referenced in a slightly different manner than with the HDOS Assembler. The most efficient method to use is to create a MACRO which will reference the SCALL. The following program segment illustrates this point:

```
.EXIT     EQU    0Q

.SCIN     EQU    1Q

.SCOUT    EQU    2Q

.PRINT    EQU    3Q

.CONSL    EQU    6Q

SCALL     MACRO  TYPE
          DB     377Q,TYPE
          ENDM
```

Now, you would reference the SCALL in the same manner as you would with the HDOS Assembler. For example, to exit to HDOS, use the following program segment:

```
      XRA     A
      SCALL   .EXIT
```

The PRINT SCALL is used in a similar manner:

```
            LXI     H,LINE
            SCALL   .PRINT
            .
            .
            .
      LINE: DB    12Q,'HELLO',2000
```

This would cause the message

```
HELLO
```

to be printed on the system console.

Appendix A – Three-Drive Fortran

The procedure for configuring the "working diskettes" is somewhat different when you are using a three-drive system. The third drive allows a more flexible method of using the FORTRAN system.

First, you will need three initialized diskettes. SYSGEN one of these diskettes.

Locate the distribution diskette labeled "HDOS Microsoft FORTRAN-80". This diskette contains the FORTRAN system files. You will want to copy three files from this distribution diskette to the diskette you previously SYSGENed. These three files are:

1. FORLIB.REL
2. CREF80.ABS
3. L80.ABS

You will also have to copy the HDOS editor, EDIT.ABS, to the diskette you previously SYSGENed.

Now, you should copy the two remaining files from the distribution diskette to another working diskette. These two files are:

1. F80.ABS
2. M80.ABS

Finally, locate the distribution diskette labeled "HDOS Microsoft FORTRAN-80 Sample Programs". You will want to copy all eleven files from this diskette to the last remaining working diskette.

When you want to use the FORTRAN system, you will boot your system with the SYSGENed diskette you created. You will also want to mount one of the other diskettes in SY1: and the third diskette in SY2:. This method will allow you to store program source files on either drive SY1: or SY2:.

You should also keep in mind that the sample program exercises in this Manual assume a two-drive system, so your files may be on different drives.

Appendix B – In Case Of Difficulty

FORTRAN-80 Compiler

Problem: The Compiler will not generate a hard copy listing.

Suggestions:

This problem is almost always caused by the device driver. Check to make sure the device driver is configured properly. Make sure the device driver is SET to use the correct port as well as the correct baud rate. Also make sure you LOAD the device driver into memory before you invoke the Compiler.

Problem: The Compiler will not load into memory.

Suggestions:

This problem can usually be solved by making a new copy of the Compiler. Use your FORTRAN-80 Distribution Diskette to make this new copy of the Compiler. If you still can not load the Compiler into memory, you may have a hardware problem. Consult your hardware service manual for troubleshooting information.

Problem: The logical IF statement produces unexpected results.

Suggestions:

This problem is caused by using integer variables with the logical IF. The best way to correct this problem is to only use real variables with the logical IF.

LINK-80 Linking Loader

Problem: Can not resolve ell the external references with the Linking Loader.

Suggestions:

Look over the source program end make sure that each external reference can be solved by either searching the library or loading e subroutine. If you have any references to non-existent functions or subroutines, you should correct them.

Problem: The Linking Loader will not exit to HDOS with the /E switch.

Suggestions:

This problem can be solved using several different methods. One way is to eliminate all STOP statements from your FORTRAN source program. The problem with the STOP statement appears to be random in nature, but in ell cases can be solved by eliminating the STOP statement.

Another method is to load the entire library. To do this, simply type FORLIB in response to the-prompt from the Linking Loader. Although this will solve the problem, this method is not very efficient with disk space.

Problem: The Linking Loader can not find the library.

Suggestions:

The library must reside on SY0: during the linking process. The Linking Loader will always assume that the library is on SY0:. The library must also have the file name: FORLIB.REL.

Problem: Can not save the object file.

Suggestions:

You have most likely run out of room on your diskette. Delete any files that you do not need. You should then be able to save the object file generated by the Linking Loader.

Runtime Problems

Problem: Can not generate hard copy output.

Suggestions:

This problem is generally caused by the device driver. Make sure the device driver is configured properly. The device driver should be SET to the proper port as well as the correct baud rate. Remember to LOAD the device driver into memory before executing the program. Also make sure that you ere referencing the correct logical unit number.

Problem: I/O errors during runtime.

Suggestions:

An I/O error during runtime could be caused by any number of problems. Several of the most common problems are discussed below.

One of the most common runtime I/O errors is the IT (I/O transmission) error. This error is generated when the FORTRAN I/O processor is unable to communicate with en HDOS I/O device. If you forget to mount SY1: or SY2: and then you attempt to access this drive, you will generate en I/O transmission error. You will also generate en I/O transmission error if you attempt output to a hard copy device without first LOADing the device driver. A reference to a non-existent I/O device will also result in an I/O transmission error.

Another common I/O error is the IN (input record too long) error. This error can usually be traced to en I/O list that is too long. It is very important to make sure that your I/O lists do not produce a record that is too long.

Problem: Errors in FORMAT statement during runtime.

Suggestions:

Two runtime warning errors can be the result of en incorrectly structured FORMAT statement. One of these errors is the TL (too many left parentheses) error. The other error is the RC (negative repeat count) error. This problem can be solved by locating and correcting any illegal FORMAT statements.

One of the most common runtime fatal errors is the FW (field width too smell) error. If the magnitude of your data is such that it will overflow the field width, then you will generate this error. To correct this error, you should make the field width larger.

Microsoft FORTRAN-80 LANGUAGE

SOFTWARE REFERENCE MANUAL

for HEATH 8-bit digital computer systems

Part A
595-2286
First Printing
Printed in the United States of America

HEATH COMPANY

BENTON HARBOR, MICHIGAN 49022

Table of Contents

Chapter Seven - Input/Output Statements

Chapter Eight - FORMAT Statements

Chapter One

Introduction

OVERVIEW

FORTRAN is a universal, problem-oriented programming language designed to simplify the preparation and check-out of computer programs. The name of the language - FORTRAN - is an acronym for FORmula TRANslator.

The syntactical rules for using the language are rigorous and require the programmer to define fully the characteristics of a problem in a series of precise statements. These statements, called the source program, are translated by a program called the FORTRAN compiler into a relocatable module. This module is then translated by another program, called the linker, into the machine language of the computer on which the program is to be executed.

This Reference Manual defines the Microsoft FORTRAN-80 source language for the Heath H8 and H89 computers.

This language includes most of the provisions of the American National Standard FORTRAN language as described in ANSI document X3.9-1966, approved on March 7, 1966, plus a number of language extensions and some restrictions.

Examples are included throughout this Manual to illustrate the construction and use of the language elements. The programmer should be familiar with all aspects of the language to take full advantage of its capabilities.

The following is a list of the extensions to ANSI Standard FORTRAN (X3.9-1966)

1. When c is used in a "STOP c" or "PAUSE c" statement, c may be up to six ASCII characters in length.
2. "Error" and "End of File" branches may be specified in READ and WRITE statements using the ERR= and END= options.
3. The standard subprograms PEEK, POKE, INP and OUT have been added to the FORTRAN library.
4. Statement functions may use subscripted variables as arguments.
5. Hexadecimal constants may be used wherever Integer constants are normally allowed.
6. The literal form of Hollerith data (a character string between apostrophe characters) is permitted in place of the standard nH form.
7. There is no restriction to the number of continuation lines.
8. Mixed mode expressions and assignments are allowed, and the conversion is done automatically.
9. Logical variables maybe used as integer quantities in the range +127 to -127.
10. Logical operations may be performed on integer data. (.AND.,.OR.,.NOT., XOR., can be used for 16-bit or 8-bit Boolean operations.)
11. ENCODE/DECODE may be used for both editing and converting data.
12. Complete language facilities are provided for random access files.

The FORTRAN programmer should note the above added features and utilize them to the fullest advantage.

FORTRAN-80 places the following restrictions upon ANSI Standard FORTRAN.

1. The COMPLEX data type has not been implemented.
2. The statements within a program unit must appear in the following order:
    1. PROGRAM, SUBROUTINE, FUNCTION, BLOCK DATA
    2. Type, EXTERNAL, DIMENSION
    3. COMMON
    4. EQUIVALENCE
    5. DATA
    6. Statement Functions
    7. Executable Statements
3. A different amount of computer memory is allocated for each of the data types: integer, real, double-precision, logical.
4. The equal sign of an assignment statement and the first comma of a DO statement must appear on the initial statement line.
5. Unformatted sequential I/O statements must always provide a variable list.

The FORTRAN programmer should note the above restrictions and adhere to them when writing a FORTRAN source program.

FORTRAN PROGRAM FORM

FORTRAN source programs consist of one program unit called the main program and any number of program units called subprograms. Main programs and program units are constructed of an ordered set of statements which precisely describe procedures for solving problems and which also define information to be used by the FORTRAN compiler during compilation of the object program. Each statement is written using the FORTRAN character set and following a prescribed line format.

FORTRAN Character Set

To simplify reference and explanation, the FORTRAN character set is divided into four subsets and a name is given to each.

LETTERS

A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z,$
(The $ is considered a letter.)

No distinction is made between upper and lower case letters. However, for clarity and legibility, exclusive use of upper case letters is recommended.

DIGITS

0,1,2,3,4,5,6,7,8,9

Strings of digits representing numeric quantities are normally interpreted as decimal numbers. However, in certain statements, the interpretation is in the hexadecimal number system in which case the letters A, B, C, D, E, F may also be used as hexadecimal digits.

ALPHANUMERICS

A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z,$
0,1,2,3,4,5,6,7,8,9

All letters and digits are considered part of this subset.

SPECIAL CHARACTERS

=	Equality Sign
+	Plus Sign
-	Minus Sign
(	Left Parenthesis
)	Right Parenthesis
.	Decimal Point

The following special characters are classified as Arithmetic Operators and are significant in the unambiguous statement of arithmetic expressions.

+	Addition or Positive Value
-	Subtraction or Negative Value
*	Multiplication
**	Exponentiation
/	Division

The other special characters have specific application in the syntactical expression of the FORTRAN language and in the construction of FORTRAN statements.

Any printable character may appear in a Hollerith or literal field.

FORTRAN Line Format

The lines of a FORTRAN source program consist of 80 character positions or columns, numbered 1 through 80, and are divided into four fields.

1. Statement Label (or Number) field- Columns 1 through 5 (See definition of statement labels).

2. Continuation character field - Column 6

3. Statement field - Columns 7 through 72

4. Identification field - Columns 73 through 79

The identification field is available for any purpose the FORTRAN programmer may desire and is ignored by the FORTRAN processor.

NOTE: The last column (80) must **never** contain a character. This column is reserved for the carriage return character.

LINE TYPES

The lines of a FORTRAN statement are placed in Columns 1 through 72 formatted according to line types. The four line types, their definitions, and column formats are:

Comment line – used for source program annotation at the convenience of the programmer.

1. Column 1 contains the letter C
2. Columns 2 - 72 are used in any desired format to express the comment or they may be left blank.
3. A comment line may be followed only by an initial line, an END line, or another comment line.
4. Comment lines have no effect on the object program and are ignored by the FORTRAN processor except for display purposes in the listing of the program.

Example:

```
1   5    .    .    .    .    .    .    .    (columns by 5)

C   COMMENT LINES ARE INDICATED BY THE
C   CHARACTER C IN COLUMN 1.
C   THESE ARE COMMENT LINES
```

END line – the last line of a program unit.

1. Columns 1-5 may contain a statement label.
2. Column 6 must contain a zero or blank.
3. Columns 7-72 must contain the characters "E", "N" and "D", in that order. They may be separated by blank characters.
4. Each FORTRAN program unit must have an END line as its last line to inform the processor that it is at the physical end of the program unit.
5. An END line may follow any other type line.

   Example:

   ```
   END
   ```

Initial Line - the first or only line of each statement.

1. Columns 1-5 may contain a statement label to identify the statement.
2. Column 6 must contain a zero or blank.
3. Columns 7-72 must contain all or part of the statement.
4. An initial line may begin anywhere within the statement field.

Example:

```
1   .    .    .    .    .    .    .    .      (columns by 5)

C   THE STATEMENT BELOW CONSISTS
C   OF AN INITIAL LINE
       A= .5*SQRT(3-2.*C)
```

Continuation Line – used when additional lines of coding are required to complete a statement originating with an initial line.

1. Columns 1-5 are ignored. Column 1 must not contain a C.
2. If Column 1 contains a C, it is a comment line.
3. Column 6 must contain a character other than zero or blank.
4. Columns 7-72 contain the continuation of the statement.
5. There may be as many continuation lines as needed to complete the statement.

Example:

```
1   .    .    .    .    .    .    .    .      (columns by 5)

C  THE STATEMENTS BELOW ARE AN INITIAL LINE
C  AND 2 CONTINUATION LINES
         BETA(1,2) =
     1 A6BAR*'7-(BETA(2,2)-A5BAR*50
     2 +SQRT (BETA(2,1)))
```

STATEMENT LABEL

A statement label may be placed in columns 1-5 of a FORTRAN statement initial line and is used for reference purposes in other statements. The rules for a statement label are:

1. The label is an integer from 1 to 99999.
2. The numeric value of the label, leading zeros and blanks are not significant.
3. A label must be unique within a program unit.
4. A label on a continuation line is ignored by the FORTRAN processor.

STATEMENTS

Individual statements deal with specific aspects of a procedure described in a program unit and are classified as either executable or non-executable.

EXECUTABLE

Executable statements specify actions and cause the FORTRAN compiler to generate object program instructions. There are three types of executable statements:

1. Assignment statements.

2. Control statements.

3. Input/Output statements.

NON-EXECUTABLE

Non-executable statements describe to the compiler the nature and arrangement of data and provide information about input/output formats and data initialization to the object program during program loading and execution. There are five types of non-executable statements:

1. Specification statements.

2. DATA Initialization statements.

3. FORMAT statements.

4. FUNCTION defining statements.

5. Subprogram statements.

Chapter Two

Compiling FORTRAN Programs

OVERVIEW

After a FORTRAN source program is created, it must then be compiled. To tell the FORTRAN Compiler what to compile and with which options, it is necessary to input a "command string," which is read by the FORTRAN-80 command scanner.

This command string contains the information needed by the Compiler in order to compile the source program.

After the source program has been compiled without errors, it is then necessary to link the program before it can be executed. This process is explained in Section C, "LINK-80", of this Reference Manual.

FORMAT OF COMMANDS

To run FORTRAN-80, type "F80" followed by a RETURN. FORTRAN-80 will return the prompt "* ", indicating it is ready to accept commands. At this point, the command string followed by a RETURN should be typed. The general format of a FORTRAN-80 command string is:

```
objprog-dev:filename.ext,list-dev:filename.ext= source-dev:filename.ext
```

where:

objprog-dev: The device on which the object program is to be written. If the device name is omitted, it defaults to SY0:.

list-dev: The device on which the program listing is written. It can be the terminal, a hardcopy device or a disk file. The default device for the disk file is SY0:.

source-dev: The device from which the source-program input to - FORTRAN-80 is obtained. If a device name is omitted, it defaults to SY0:.

filename.ext These are the filename and filename extensions for the object program file, the listing file, and the source file. Filename extensions may be omitted.

The default filename extensions are:

source file	.FOR
object file	REL
listing file	LST

Either the object file or the listing file or both may be omitted.

If neither a listing file nor an object file is desired, place only a comma to the left of the equal sign. This is useful for checking the syntax of a newly created FORTRAN source program.

If the names of the object file and the listing file are omitted, the default is the name of the source file with the above default extensions.

Examples:

* =TEST	Compile the program TEST.FOR and place the object in TEST.REL
*SY1:TEST=SY1:TEST	Compile SYI:TEST.FOR put object in SY1:TEST.REL and listing in SY1:TEST.LST
*,=TEST.FOR Compile	TEST.FOR but produce no object or listing file. This is useful when checking for errors.
*SY1:SAMPLE,LP:=SAMPLE	Compile the program SAMPLE.FOR, write the listing to LP: and put the object in SY1:SAMPLE.REL. (A valid device driver named LP.DVD must exist on drive SY0:.)

After the program has compiled, the prompt "*" will be displayed on the terminal device. In order to return control to HDOS, type CTRL Z twice or CTRL D.

The command string can also be typed on the same line as the command to invoke the Compiler. In order to do this, type "F80 <command string>". (The space is required.) When using this method, the Compiler will return to HDOS after the program has been compiled.

Examples:

F80 TEST=TEST	Invoke the Compiler, compile TEST.FOR and write the object in TEST.REL. After compilation, return to HDOS. The Compiler prompt "*" is not displayed with this method.

Note: With this method, if an invalid command string is typed, first exit to HDOS by typing CTRL-D and then reinvoke the Compiler.

FORTRAN-80 COMPILATION SWITCHES

A number of different switches may be given in the command string that will affect the compilation process.

Each switch should be preceded by a slash

Switch	Action
O	Print all listing addresses, in octal.
H	Print all listing addresses, in hexadecimal. (default)
N	Do not list the compiler generated opcodes. (default)
A	List compiler generated opcodes.
R	Force generation of an object file.
L	Force generation of a listing file.
P	Each /P allocates an extra 100 bytes of stack space for use during compilation. Use /P if stack overflow errors occur during compilation. Otherwise this switch should not be needed.
M	Specifies to the Compiler that the generated code should be in a form which can be loaded into ROMs. When a /M is specified, the generated code will differ from normal in the following ways: 1. FORMATs will be placed in the program area, with a JMP" around them. 2. Parameter blocks (for subprogram calls with more than 3 parameters) will be initialized at runtime, rather than being initialized by the loader.

Examples:

* =TEST/L	Compile file TEST.FOR, write listing in file TEST.LST and produce object file TEST.REL.
* =BIGGONE/P/P	Compile file BIGGONE.FOR and produce object file BIGGONE.REL. Compiler is allocated 200 extra bytes of stack space.
*PROG,PROG=PROG/O	Compile file PROG.FOR, write listing in file PROG.LST and produce object file PROG.REL. The addresses in the listing file will be in octal.
* =SAMPLE/L/A	Compile file SAMPLE.FOR, write listing in SAMPLE.LST and produce object file SAMPLE.REL. The compiler generated opcodes will be printed in the listing file.
*=FIRM,FIRM=FIRM/M	Compile file FIRM.FOR, write listing in file FIRM.LST and produce object file FIRM.REL. Generates code suitable for ROM's.

If a FORTRAN program is intended for ROM, the programmer should be aware of the following ramifications:

1. DATA statements should not be used to initialize RAM. Such initialization is done by the loader, and will therefore not be present at execution. Variables and arrays may be initialized during execution via assignment statements, or by READing into them.

2. FORMATs should not be read into during execution.

3. DISK files should not be OPENed on any LUNs other than 6, 7, 8, 9,10.

Sample Compilation

```
SY1:SAMPLE,TT:=SY1:SAMPLE

FORTRAN-80 Ver. 3.35 Copyright 1978 (C) By Microsoft - Bytes: 9320
04-JUN-80

1 C  SAMPLE PROGRAM AVERAGE
2 C  WILL COMPUTE AVERAGE OF THREE NUMBERS 3    PROGRAM SAMPLE
4     REAL NUMBER(3)
5     DATA NUMBER /100.,200.,300./
6     DATA SUM,AVER /0.,O./
7 C  LOOP TO CALCULATE SUMMATION
8     DO 600 I = 1,3
9     SUM = SUM + NUMBER(I)
10 600      CONTINUE
11    AVER = SUM/3
12    WRITE(1,700) AVER
13 700      FORMAT(' ',F6.2)
14    STOP SAMPLE
15    END

PROGRAM UNIT LENGTH=0065 (101) BYTES
DATA AREA LENGTH=0021 (33) BYTES

SUBROUTINES REFERENCED:

$I1         $INIT       $L1
$AB         $T1         $DA
$W2         $ND         $ST

VARIABLES:

NUMBER  0001"      SUM   000D"          AVER  0011"
I       0015"

LABELS:

$$L   0006' 600L  0024' 700L  0017"
```

Note: The single quote mark (') implies a program relative value and the double quote mark (") implies a data relative value.

FORTRAN COMPILER ERROR MESSAGES

The FORTRAN-80 Compiler detects two kinds of errors: Warnings and Fatal Errors.

When a Warning is issued, compilation continues with the next item on the source line. When a Fatal Error is found, the compilation ignores the rest of the logical line, including any continuation lines.

Warning messages are preceded by percent signs (%), and Fatal errors by question marks (?).

Next, a line number will be printed by the Compiler. This line number represents the point at which the Compiler discovers the error. It is important to note that this line number may not correspond to the actual physical line number of the error. The line number is followed by the error code or error message. The last twenty characters scanned before the error is detected are also printed.

Example:

```
?Line 25: Mismatched Parenthesis
%Line 16: Missing Integer Variable
```

When either type of error occurs, the program should be changed so that it compiles without errors. No guarantee is made that a program that compiles with errors will execute sensibly.

The next several pages contain an explanation of the various errors detected by the Compiler. The runtime error messages are also explained.

Appendix D, "Microsoft Errors," contains a detailed explanation of both the Compiler and the runtime errors.

Fatal Errors:

Number	Message
100	Illegal Statement Number
101	Statement Unrecognizable or Misspelled
102	Illegal Statement Completion
103	Illegal DO Nesting
104	Illegal Data Constant
105	Missing Name
106	Illegal Procedure Name
107	Invalid DATA Constant or Repeat Factor
108	Incorrect Number of DATA Constants
109	Incorrect Integer Constant
110	Invalid Statement Number
111	Not a Variable Name
112	Illegal Logical Form Operator
113	Data Pool Overflow
114	Literal String Too Large
115	Invalid Data List Element in I/O
116	Unbalanced DO Nest
117	Identifier Too Long
118	Illegal Operator
119	Mismatched Parenthesis
120	Consecutive Operators
121	Improper Subscript Syntax
122	Illegal Integer Quantity
123	Illegal Hollerith Construction
124	Backwards DO reference
125	Illegal Statement Function Name
126	Illegal Character for Syntax
127	Statement Out of Sequence
128	Missing Integer Quantity
129	Invalid Logical Operator
130	Illegal Item in Type Declaration
131	Premature End Of File on Input Device
132	Illegal Mixed Mode Operation
133	Function Call with No Parameters
134	Stack Overflow
135	Illegal Statement Following Logical IF

Warnings

Number	Message
0	Duplicate Statement Label
1	Illegal DO Termination
2	Block Name = Procedure Name
3	Array Name Misuse
4	COMMON Name Usage
5	Wrong Number of Subscripts
6	Array Multiply EQUIVALENCEd within a Group
7	Multiple EQUIVALENCE of COMMON
8	COMMON Base Lowered
9	Non-COMMON Variable in BLOCK DATA
10	Empty List for Unformatted WRITE
11	Non-Integer Expression
12	Operand Mode Not Compatible with Operator
13	Mixing of Operand Modes Not Allowed
14	Missing Integer Variable
15	Missing Statement Number on FORMAT
16	Zero Repeat Factor
18	Format Nest Too Deep
19	Statement Number Not FORMAT Associated
20	Invalid Statement Number Usage
21	No Path to this Statement
22	Missing Do Termination
23	Code Output in BLOCK DATA
24	Undefined Labels Have Occurred
25	RETURN in a Main Program
27	Invalid Operand Usage
28	Function with no Parameter
29	Hex Constant Overflow
30	Division by Zero
32	Array Name Expected
33	Illegal Argument to ENCODE/DECODE

FORTRAN RUNTIME ERROR MESSAGES

FATAL ERRORS:

Code	Meaning
ID	Illegal FORMAT Descriptor
F0	FORMAT Field Width is Zero
MP	Missing Period in FORMAT
FW	FORMAT Field Width is Too Small
IT	I/O Transmission Error
ML	Missing Left Parenthesis in FORMAT
DZ	Division by Zero
LG	Illegal Argument to LOG Function (Negative or Zero)
SQ	Illegal Argument to SQRT Function (Negative)
DT	Data Type Does Not Agree With FORMAT Specification
EF	EOF Encountered on READ

WARNING ERRORS:

Code	Meaning
TL	Too Many Left Parentheses in FORMAT
DE	Decimal Exponent Overflow (Number in input stream had an exponent larger than 99)
IS	Integer Size Too Large
IN	Input Record Too Long
OV	Arithmetic Overflow
CN	Conversion Overflow on REAL to INTEGER Conversion
SN	Argument to SIN Too Large
A2	Both Arguments of ATAN2 are 0
IO	Illegal I/O Operation
RC	Negative Repeat Count in FORMAT

Runtime errors are surrounded by asterisks as follows:

```
**FW**
```

Fatal errors cause execution to cease (control is returned to the operating system). Execution continues after a warning error. However, after 20 warnings, execution ceases.

Chapter Three

Data Representation/Storage Format

OVERVIEW

The FORTRAN language specifies that data types be categorized into several classifications. These classifications are; integer, real, double-precision, logical and Hollerith.

Within each individual data type there exists another classification referred to as the data name. The classifications of data names are: constants, variables, arrays and array elements.

A constant represents a fixed value, and therefore may not be changed during program execution. Variable data, on the other hand, may be subject to modification. An array is a group of storage locations associated with a single symbolic name. This symbolic name is called the array name. An array element refers to a single entity within an array.

DATA TYPES

FORTRAN recognizes several unique data types: integer, real, double-precision, logical and Hollerith. A data type can be selected in the source program by using the data type statement. (The data type statement is discussed in Chapter 9, "Specification Statements".)

The data type can also be established by following the predefined default convention. The default convention associates all symbolic names starting with the letters I,J,K,L,M,N with the data type integer. All other symbolic names are associated with the real data type.

A data type is usually determined by specific program requirements. For instance, because integer arithmetic always executes faster than double-precision arithmetic, integer variables are almost always assigned to repetitive "DO LOOPS" as index counters.

Integer

Integers are precise representations of integral numbers (positive, negative or zero) having precision to 5 digits in the range -32768 to *+32767* inclusive (-2**15 to 2**15-1).

[Integers are stored as a 16-bit value. The low order bits, 0 through 14, represent the binary value. The high-order bit, (bit 15) is used to indicate if a value is positive or negative. Negative integers are stored as the two's complement of a positive integer.

RULES:

- 1 to 5 decimal digits are interpreted as a decimal number.

```
Examples:    -763
                1
           +00672
```

- A preceding plus (+) or minus (-) sign is optional.

```
Examples:    -32768
             +32767
```

- No decimal point (.) or comma (,) is allowed.

Real

Approximations of real numbers (positive, negative or zero) in computer storage is represented in a four-byte, floating-point form. Storage of real data is precise to seven significant digits and their magnitude may lie between the approximate limits of 10**-38 and 10**38 (2**-127 and 2**127).

Both real and double-precision values are stored in a floating-point format. The storage unit for a real value is 32 bits in length. Bits zero through 23 are allocated to the mantissa. Bits 24 through 31 are reserved for the characteristic.

The mantissa is stored in two's complement notation. The mantissa is a binary fraction such that the radix point is always assumed to be to the left of the fraction. The mantissa is always normalized such that the high-order bit is one, eliminating the need to actually save that bit. This bit is assumed to be a one unless the exponent is zero. In this case only, the high-order bit is assumed to be a zero.

RULES

A decimal number with precision to seven digits is represented in one of the following forms:

```
a.    +/-.f      or    +/-i.f
b.    +/-i.E+    or    +/-i.-e
c.    +/-.fE+    or    +/-.f-e
d.    +/-i.fE+   or    +/-i.f-e
```

where i, f, and E (-e) are each strings representing integer, fraction, and exponent respectively.

Plus (+) and minus (-) characters are optional.

The decimal point is not optional. All real numbers must be expressed with a decimal point. The value of the exponent E (-e) is interpreted as a real number times 10**e, where the range of the exponential value is between plus or minus 38 (i.e., -38<=a<=+38).

If the constant preceding E+ or -e contains more significant digits than the precision for real data allows, truncation occurs, and only the most significant digits in the range will be represented.

Double-Precision

Approximations of real numbers (positive, negative or zero) are represented in computer storage in 8-byte, floating-point form. Double-precision data are precise to 16 significant digits in the same magnitude range as real data.

Both real and double-precision values are stored in a floating-point format. However, the storage unit for a double-precision value is 64 bits in length. See the discussion on real data for more information on the floating-point format.

RULES

A decimal number with precision to 16 digit is represented in one of the following forms:

```
a.    +/-.f      or    +/-i.f
b.    +/-i.D+    or    +/-i.-d
c.    +/-.fD+    or    +/-.f-d
d.    +/-i.fD+   or    +/-i.f-d
```

where *i, f,* and *D (-d)* are each strings representing integers, fraction, and exponent respectively. Note that a real constant is assumed single precision unless it contains a "D" exponent.

Plus (+) and minus (-) characters are optional. The decimal point is not optional. All real numbers must be expressed with a decimal point. In the form shown in *b* above, if r represents any of the forms preceding D+ or -d (i.e., rD+ or -d), the value of the constant is interpreted as *r* times 10**e, where -38<=d<=38.

If the constant preceding D+ or -d contains more significant digits than the precision for real data allows, truncation occurs, and only the most significant digits in the range will be represented.

Logical

A logical data type is an element in computer storage representing only the logical true and false values. The storage unit for a single logical value is seven bits. However, bit eight is frequently used to represent a signed integer. A logical true constant is assigned a negative one value. Any non-zero value is also treated as a true constant.

The logical expression may take on only two values, true or false. The internal representation of false is zero. A non-zero value is always represented and internally stored as true.

When a logical expression appears in a FORTRAN statement it is evaluated according to the rules given below. Logical types may also be used as one-byte signed integers in the range from -128 to +127.

RULES

Any non-zero value is assigned the logical value of ".TRUE.".

A logical ".FALSE." value is only assigned when all bits in the byte are set to zero.

Logical values maybe used as one-byte integers. The rules for using logical values as integers remain the same. Note that the range is only from -128 to +127 (i.e., -2**7 to 2**7-1) when using logical values as integers.

Hollerith

Any number of characters from the computer's character set are valid entries in the string. All characters including blanks are significant. Hollerith data require one byte for storage for each character in the string. Hexadecimal data may be associated (via a DATA statement) with any type data. Its storage allocation is the same as the associated datum.

Hollerith or literal data may be associated with any data type by use of DATA initialization statements (see Chapter 9, "Specification Statements").

Up to eight Hollerith characters may be associated with double-precision type storage. Real type storage allows up to four characters, and integer storage uses up to two characters. Logical type storage uses only one character.

DATA NAMES

Data names recognized by the FORTRAN compiler are separated into three distinct groups: constants, variables and arrays. In the source language these names are frequently used to identify and assign values to a particular item.

Constants

FORTRAN constants are always identified by stating their actual value. A constant may be either a positive or a negative value. The symbol for a negative value (-) must always precede a negative constant. However, a positive value may or may not use the (+) symbol before the constant.

Variables

FORTRAN variables are always identified by stating their symbolic name. A symbolic name in FORTRAN is always a unique string of from one to six alphabetical or numeric characters. The first character of a name is always alphabetical.

NOTE: System variable names and runtime subprogram names are distinguished from other variable names in that they begin with the dollar sign character ($). It is therefore strongly recommended that in order to avoid conflicts, symbolic names in FORTRAN source programs begin with some letter other than "$".

Arrays

An array in FORTRAN is an ordered set of data arranged in a meaningful pattern. This data is always characterized by the property of dimension. An array may have 1, 2 or 3 dimensions and is identified by a symbolic name. Each element in 3n array is uniquely addressable by means of a subscript. (For more information on arrays, see Chapter 9, "Specification Statements".)

ARRAY ELEMENT

An array element is one member of the data set that makes up an array. Reference to an array element in a FORTRAN statement is made by appending a subscript to the array name. The subscript is used to uniquely identify a single element within the array.

Rules that govern the use of subscripts are as follows:

1. A subscript contains 1, 2 or 3 subscript expressions enclosed in parentheses.

2. If there are two or three subscript expressions within the parentheses, they must be separated by commas.

3. The number of subscript expressions must be the same as the specified dimensionality of the array. (An exception to this rule is the EQUIVALENCE statement. See Chapter 9, "Specification Statements" for a complete discussion of this exception.)

4. A subscript expression is written in one of the following forms:

K	C*V	V-K
V	C*V+K	C*V-K
V+K		

   where C and K are integer constants and V is an integer variable name

5. Subscripts themselves may not be subscripted.

Examples of valid subscripts:

```
x(2*J-3,7)
A(I,J,K)
I(20)
C(L-2)
Y(I)
```

TYPE	ALLOCATION
INTEGER	2 bytes are required for storage. Negative numbers are the two's complement of positive representations.
LOGICAL	1 byte is required for storage. Zero (false) or non-zero (true) A non-zero valued byte indicates true (the logical constant .TRUE. is represented by the hexadecimal value FF). A zero valued byte indicates false. When used as an arithmetic value, a logical datum is treated as an integer in the range -128 to +127.
REAL	4 bytes are required for storage. The first byte is the characteristic expressed in excess 200 (octal) notation; i.e., a value of 200 (octal) corresponds to a binary exponent of 0. Values less than 200 (octal) correspond to negative exponents, and values greater than 200 correspond to positive exponents. By definition, if the characteristic is zero, the entire number is zero. The next three bytes constitute the mantissa. The mantissa is always normalized such that the high order bit is one, eliminating the need to actually save that bit. The high order bit is used instead to indicate the sign of the number. A one indicates a negative number, and zero indicates a positive number. The mantissa is assumed to be a binary fraction whose binary point is to the left of the mantissa.
DOUBLE PRECISION	8 bytes are required for storage. The internal form of double-precision data is identical with that of real data except double-precision uses 4 extra bytes for the mantissa.

TABLE 3-1
Storage Allocation by Data Types

Chapter Four

FORTRAN Expressions

OVERVIEW

A FORTRAN expression is composed of a single operand or a string of operands connected by operators.

Two expression types - Arithmetic and Logical - are provided by FORTRAN. The operands, operators, and rules of use for both types are described in the following chapter.

ARITHMETIC EXPRESSIONS

Arithmetic expressions are composed of arithmetic operands and arithmetic operators. The evaluation of an arithmetic expression yields a single numeric value.

An arithmetic operand may be:

- A constant
- A variable name
- An array element
- A FUNCTION reference

The arithmetic operators and their functions are:

Operator	Function
**	Exponentiation
*	Multiplication
/	Division
+	Addition and Unary Plus
-	Subtraction and Unary Minus

The following rules define all permissible arithmetic expression forms:

1. A constant, variable name, array element reference or FUNCTION reference standing alone is an expression.

   Examples:

   ```
   S(I)      JOBNO    217    17.26    SQRT(A+B)
   ```

2. If E is an expression whose first character is not an operator, then +E and -E are called signed expressions.

   Examples:

   ```
   -S        +JOBNO   -217   +17.26   -SQRT(A+B)
   ```

3. If E is an expression, then (E) means the quantity resulting when E is evaluated.

   Examples:

   ```
   (-A) (JOBNO) -(X+1) (A-SQRT(A+B))
   ```

4. If E is an unsigned expression and F is any expression, then: F+E, F-E, F*E, F/E and F**E are all expressions.

   Examples:

   ```
   -(B(I,J)+SQRT(A+B(K,L))) 1.7E-2**(X+5.0) -(B(I+3,3*J+5)+A)
   ```

5. An evaluated expression maybe integer, real, double-precision, or logical. The type is determined by the data types of the elements of the expression.

   If the elements of the expression are not all of the same type, the type of the expression is determined by the element having the highest type. The type hierarchy (highest to lowest) is as follows:

   double-precision real
   integer logical

6. Expressions may contain nested parenthesized elements as in the following:

   ```
   A*(Z-((Y+X)/T))**J
   ```

   where Y+X is the innermost element, (Y+X)/T is the next innermost, Z-((Y+X)/T) the next. In such expressions, care should be taken that the number of left parentheses and the number of right parentheses are equal.

Arithmetic Expression Evaluation

Arithmetic expressions are evaluated according to the following rules:

1. Parenthesized expression elements are evaluated first. If parenthesized elements are nested, the innermost elements are evaluated, then the next innermost until the entire expression has been evaluated.

2. Within parentheses and/or wherever parentheses do not govern the order or evaluation, the hierarchy of operations in order of precedence is as follows:

   a. FUNCTION evaluation
   b. Exponentiation
   c. Multiplication and Division
   d. Addition and Subtraction

   The expression:

   ```
   A*(Z-((Y+R)/T))**J+VAL
   ```

   is evaluated in the following sequence:

   ```
   Y+R = e1, (e1)/T = e2, Z-e2 = e3, e3**J = e4, A*e4 = e5. e5 +VAL = e6
   ```

3. The expression X**Y**Z is not allowed. It should be written as follows:

   ```
   (X**Y)**Z or X**(Y**Z)
   ```

4. Use of an array element reference requires the evaluation of its subscript. Subscript expressions are evaluated under the same rules as other expressions.

LOGICAL EXPRESSIONS

A Logical Expression may be any of the following:

1. A single logical constant (i.e., TRUE. or .FALSE.), a logical variable, logical array element or logical FUNCTION reference. (see Chapter 10 "Functions and Subprograms".)

2. Two arithmetic expressions separated by a relational operator (i.e., a relational expression).

3. Logical operators acting upon logical constants, logical variables, logical array elements, logical FUNCTIONS, relational expressions or other logical expressions. The value of a logical expression is always either.TRUE. or.FALSE.

Relational Expressions

The general form of a relational expression is as follows: e1 r e2

where e1 and e2 are arithmetic expressions and r is a relational operator. The six relational operators are as follows:

.LT.	Less Than
.LE.	Less than or equal to
.EQ.	Equal to
.NE.	Not equal to
.GT.	Greater than
.GE.	Greater than or equal to

The value of the relational expression is TRUE. if the condition defined by the operator is met. Otherwise, the value is .FALSE.

Examples:

```
A.EQ.B
(A**J).GT.(ZAP*(RHO*TAU-ALPH))
```

Logical Operators

Table 4-1 lists the logical operations. U and V denote logical expressions.

.NOT.U	The value of this expression is the logical complement of U (i.e., 1 bits become 0 and 0 bits become 1).
U.AND.V	The value of this expression is the logical product of U and V (i.e., there is a 1 bit in the result only where the corresponding bits in both U and V are 1).
U.OR.V	The value of this expression is the logical sum of U and V (i.e., there is a 1 in the result if the corresponding bit in U or V is 1 or if the corresponding bits in both U and V are 1).
U.XOR.V	The value of this expression is the exclusive OR of U and V (i.e., there is a one in the result if the. corresponding bits in U and V are 1 and 0 or 0 and 1 respectively).

Examples:

```
If U = 01101100 and V = 11001001, then:

.NOT.U = 10010011

U.AND.V = 01001000

U.OR.V = 11101101

U.XOR.V = 10100101
```

Table 4-1. Logical Operators

The following are additional considerations for construction of logical expressions:

1. Any logical expression may be enclosed in parentheses. However, a logical expression to which the .NOT. operator is applied **must** be enclosed in parentheses if it contains two or more elements.

2. In the hierarchy of operations, parentheses may be used to specify the ordering of the expression evaluation. Within parentheses, and where parentheses do not dictate evaluation order, the order is understood to be as follows:

   a. FUNCTION Reference
   b. Exponentition (**)
   c. Multiplication and Division (* and /)
   d Addition and Subtraction (+ and -)
   e. .LT., .LE., .EQ., .NE., .GT., .GE.
   f. .NOT.
   g. .AND.
   h. .OR, .XOR..

   Examples:

   The expression:

```
X .AND. Y .OR. B(3,2) .GT. Z
```

   is evaluated as:

```
e1 = B(3,2) .GT. Z
e2 = X .AND. Y
e3 = e2 .OR. e 1
```

   The expression:

```
X .AND. (Y .OR. B(3,2) .GT. Z)
```

   is evaluated as:

```
e1 = B(3,2) .GT. Z
e2 = Y .OR. e1
e3 = X .AND. e2
```

3. It is invalid to have two contiguous logical operators except when the second operator is .NOT.

   Examples:

```
A .AND. .NOT. B   is permitted
A .AND. .OR. B is not permitted
```

HOLLERITH, LITERAL, AND HEXADECIMAL CONSTANTS IN EXPRESSIONS

Hollerith, literal, and hexadecimal constants are allowed in expressions in place of integer constants. These special constants always evaluate to an integer value and are therefore limited to a length of two bytes. The only exceptions to this are:

1. Long Hollerith or literal constants may be used as subprogram parameters.

2. Hollerith, literal, or hexadecimal constants may be up to four bytes long in DATA statements when associated with real variables, or up to eight bytes long when associated with double-precision variables.

The Hollerith and literal constants are constructed by enclosing the entire string of characters in a set of single quotation marks. Two quotation marks in succession may be used to represent the quotation mark character within the string.

Example:

```
'THIS IS A LITERAL'
```

A hexadecimal constant is specified by the letter Z or X followed by up to four hexadecimal digits (0-9) and (A-F) enclosed in a set of single quotation marks.

```
X'FFFF'

Z'AB'
```

Chapter Five

ASSIGNMENT Statements

OVERVIEW

Assignment statements are used to associate the value of an expression with a symbolic name. The symbolic name is usually referred to as a variable.

The three types of assignment statements are:

1. Arithmetic assignment statement
2. Logical assignment statement
3. ASSIGN statement

The arithmetic assignment statement assigns the value of an arithmetic expression to an arithmetic variable or array element.

The logical assignment statement assigns the value of a logical expression to a logical variable or array element.

The ASSIGN statement assigns a statement label to an integer variable.

ARITHMETIC ASSIGNMENT STATEMENT

The general form of the arithmetic assignment statement is:

```
v = e
```

where v is any variable or array element and e is an expression.

FORTRAN semantics defines the equality sign (=) as meaning "to be replaced by" rather than the normal "is equivalent to".

An arithmetic assignment statement, when executed, will evaluate the expression on the right of the equality sign and place that result in the storage space allocated to the variable or array element on the left of the equality sign.

The following conditions apply to arithmetic assignment statements:

1. Both v and the equality sign must appear on the same line. This holds even when the statement is part of a logical IF statement. (A detailed discussion of the logical IF statement is presented in Chapter 6 "FORTRAN Control Statements".)

2. The line containing v = must be the initial line of the statement unless the statement is part of a logical IF statement. In that case the v = must occur no later than the end of the first line after the end of the IF.

3. If the data types of the variable, v, and the expression, e, are different, then the value determined by the expression will be converted, if possible, to conform to the data type of the variable.

When the data type of the variable (v) and the expression (e) are different, the value of the expression must conform to the range of the variable. For example, if the value of an expression is 32,800, an attempt to assign this value to an integer variable will produce undesirable results.

Table 5-1 shows which type expressions may be equated to which type of variable. Y indicates a valid replacement. Footnotes to Y indicate conversion considerations.

	EXPRESSION TYPES (e)			
Variable Types (V)	Integer	Real	Logical	Double
Integer	Y	Ya	Yb	Ya
Real	Yc	Y	Yc	Ye
Logical	Yd	Ya	Y	Ya
Double	Yc	Y	Yc	Y

Footnotes:

a. The real expression value is converted to integer.

b. The sign is extended through the second byte.

c. The variable is assigned the real representation of the integer value of the expression.

d. The variable is assigned the truncated value of the integer expression (the low-order byte is used, regardless of sign).

e. The variable is assigned the rounded value of the real expression.

Table 5-1. Assignment by Type

LOGICAL ASSIGNMENT STATEMENT

The general form of the logical assignment statement is:

```
v = e
```

where v is a logical variable or a logical array element and e is a logical expression. The expression e is evaluated at the time of the execution of the assignment statement.

The results of this evaluation will be either zero (false) or non-zero (true). The results of evaluating e will be placed in the storage location allocated to the logical variable v.

The variable or array element v must be explicitly defined as a logical variable type.

Examples:

```
FLAG=.TRUE

TEST=.FALSE

COMP=(L .LT. 10)
```

ASSIGN STATEMENT

The ASSIGN statement is used to assign a statement label to an integer variable. The integer variable can then be used as the transfer destination in a subsequent Assigned GO TO statement. (For a detailed discussion of the Assigned GO TO, refer to Chapter 6 "FORTRAN Control Statements".)

The general form of the statement is:

```
            ASSIGN j to i
```

where j is a statement label of an executable statement and i is an integer variable.

The integer variable to which a statement number has been assigned may not be used as an arithmetic variable, although the integer variable may be redefined as an arithmetic integer value and used accordingly.

Example:

The statement:

```
            ASSIGN 400 TO LABEL
```

will associate the variable LABEL with the statement label 400. Arithmetic operations with the variable LABEL are now invalid.

The statement:

```
            LABEL=100
```

will disassociate the variable LABEL from statement label 400. The variable LABEL can now be used as an arithmetic operand.

Chapter Six

FORTRAN Control Statements

OVERVIEW

FORTRAN control statements are executable statements which affect and guide the logical flow of a FORTRAN program. The statements in this category are as follows:

1. GO TO statements:
    1. Unconditional GO TO
    2. Computed GO TO
    3. Assigned GO TO
2. IF statements:
    1. Arithmetic IF
    2. Logical IF
3. DO
4. CONTINUE
5. STOP
6. PAUSE
7. CALL
8. RETURN
9. END

GO TO STATEMENTS

Unconditional GO TO Statement

The unconditional GO TO statement transfers control to some other statement within the program unit.

The statement is of the following form:

```
          GO TO k
```

where k is the statement label of an executable statement in the same program unit.

This statement will unconditionally transfer control to the statement identified by the specified label. It will transfer control to the same statement every time it is executed.

Example:

```
      GO TO 376
        .
        .
        .
  376 A=100
```

This statement will transfer control to the statement labeled 376.

Computed GO TO Statement

The computed GO TO statement transfers control to a statement based on the value of an integer variable given within the statement.

The statement is of the following form:

```
GO TO (k1,k2,...,kn),j
```

where the ki are statement labels, and j is an integer variable, 1 <= j <= n. This statement causes transfer of control to the statement labeled kj. If j < 1 or j > n, control will be passed to the next statement foIlowing the Computed GO TO.

Example:

```
GO TO(7, 70, 700, 7000, 70000), J
```

When j = 3, the computed GO TO transfers control to statement 700. Making j = 0 or j = 6 would cause control to be transferred to the next statement after the GO TO statement.

Assigned GO TO Statement

This statement will transfer control to the statement label represented by an integer variable.

Assigned GO TO statements are of the following forms:

```
GO TO j,(k1,k2,...,kn)
GO TO j
```

where j is an integer variable name, and the ki (if present) are statement labels of executable statements.

This statement causes transfer of control to the statement whose label is equal to the current value of the integer variable j.

The value of j must be established via an ASSIGN statement. (For detailed information on the ASSIGN statement, refer to Chapter 5 "Assignment Statements".)

(cont'd.)

QUALIFICATIONS

1. The ASSIGN statement must logically precede an assigned GO TO.

2. The ASSIGN statement must assign a value to j which is a statement label included in the list of k's, if the list is specified.

Examples:

```
ASSIGN 80 TO LABEL
.
.
.
GO TO LABEL, (80,90, 100)
```

Only the statement labels 80, 90 or 100 may be assigned to LABEL. Upon execution of the GO TO statement, control would be transferred to the label assigned the variable LABEL. In the above program segment, control would be transferred to the statement with the label 80.

```
GO TO TRANS
```

The value assigned to TRANS must be the label of an executable statement within the program unit. Upon execution of the GO TO statement, control would be transferred to the label assigned to the integer variable TRANS.

IF STATEMENTS

Logical IF Statement

A logical IF statement will cause a conditional statement execution. The logical IF statement is of the form:

```
IF (u )s
```

where u is a logical expression and s is any executable statement except a DO statement or another logical IF statement. The logical expression u is evaluated as .TRUE. or .FALSE. Chapter 4 "FORTRAN Expressions", contains a discussion of logical expressions.

CONTROL CONDITIONS:

If u is FALSE, the statements is ignored and control goes to the next statement following the logical IF statement.

If, however, the expression is TRUE, then control goes to the statements, and subsequent program control follows normal conditions.

If s is a replacement statement (v = e), the variable and equality sign (=) must be on the same line. This line can be either immediately following IF (u) or on a separate continuation line.

Examples:

```
IF(I .GT. 20) GO TO 115

IF(Q .AND. R) ASSIGN 10 TO J

IF(Z) CALL DECL(A,B,C)

IF(A .OR. B .LE. PI/2) I=J
```

Arithmetic IF Statement

The arithmetic IF statement transfers control to one of a series of statements depending upon the value of an arithmetic expression.

The arithmetic IF statement is of the form:

```
IF(e) m1,m2,m3
```

where e is an arithmetic expression and m1, m2 and m3 are statement labels. Evaluation of expression e determines one of three transfer possibilities:

If e is:	Transfer to:
< 0	m1
= 0	m2
> 0	m3

Examples:

Statement	Expression Value	Transfer to
IF(A)3,4,5	15	5
IF(N-1)50,73,9	0	73
IF(B-100)6,7,8	-50	6

DO STATEMENT

The DO statement provides a method for repetitively executing a series of statements.

The statement takes one of the two following forms:

```
        DO k i = m1,m2,m3

or      DO k i = m1,m2
```

k must be a statement label. i must be an integer or logical variable. ml, m2, m3 must be integer constants or integer or logical variables. i, ml, m2, and m3 must be positive.

The variables are defined as follows:

k	is the terminal statement
i	is the control variable
m1	is the initial variable
m2	is the terminal variable
m3	is the incremental variable If m3 is 1, it may be omitted

The statement labeled *k*, called the terminal statement, must be an executable statement. The terminal statement must physically follow its associated DO. The executable statements following the DO, up to and including the terminal statement, constitute the range of the DO statement. The terminal statement may not be an arithmetic IF, GO TO, RETURN, STOP, PAUSE or another DO.

If the terminal statement is a logical IF and its expression is .FALSE., then the statements in the DO range are reiterated. If the expression is .TRUE., the statement of the logical IF is executed and then the statements in the DO range are reiterated. The statement of the logical IF may not be a GO TO, arithmetic IF, RETURN, STOP or PAUSE. (The logical IF statement is discussed earlier in this chapter.)

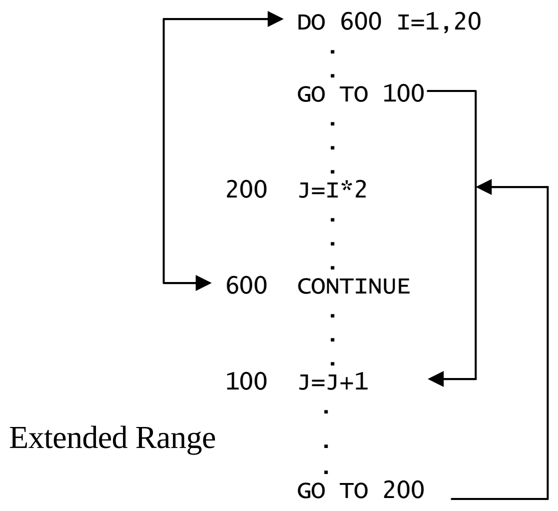
The controlling integer variable, i, is called the index of the DO range. The index must be positive and may not be modified by any statement in the range. If *m1*, *m2*, and *m3* are INTEGER*1 variables or constants, the DO loop will execute faster and be shorter, but the range is limited to 127 iterations.

During the first execution of the statements in the DO range, i is equal to m 1; the second execution, i = m 1 +m3; the third, i =m 1 +2 *m3, etc., until i is equal to the highest value in this sequence less than or equal to m2, and then the DO is said to be satisfied. The statements in the DO range will always be executed at least once, even if m 1 > m2.

When the DO has been satisfied, control passes to the statement following the terminal statement, otherwise control transfers back to the first executable statement following the DO statement.

The range of a DO statement may be extended to include all statements which may logically be executed between the DO and its terminal statement. Thus, parts of the DO range may be situated such that they are not physically between the DO statement and its terminal statement but are executed logically in the DO range. This is called the **extended range.**

Extended Range Example:

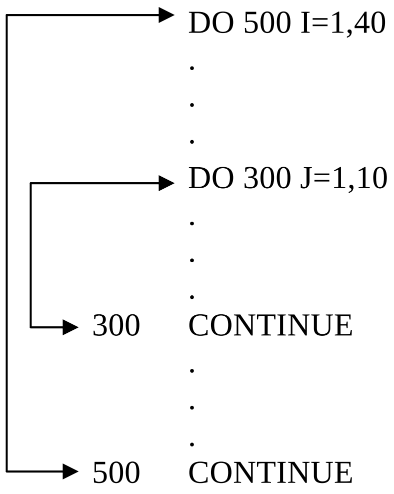


```
           DO 600 I=1,20
             .
             .
           GO TO 100
             .
             .
             .
      200  J=I*2
             .
             .
             .
      600  CONTINUE
             .
             .
             .
      100  J=J+1
             .
             .
             .
           GO TO 200
```

In the above program segment, the range of the DO loop is extended to include the statement labeled 100, as well as the statement which returns control to the DO loop. It is important to note that the control variable should not be changed either in the DO loop or in the extended range. Also note that the transfer back to the DO loop must be made to a statement before the terminal statement.

Within the range of a DO statement, there may be other DO statements, in which case the DO's are said to be **nested.** That is, if the range of one DO contains another DO, then the range of the inner DO must be entirely included in the range of the outer DO. The terminal statement of the inner DO may also be the terminal statement of the outer DO.

Nested DO loop Examples:



```
        DO 500 I=1,40
        .
        .
        .
        DO 300 J=1,10
        .
        .
        .
  300   CONTINUE
        .
        .
        .
  500   CONTINUE
```

In the above program segment note that the range of the inner loop is entirely contained in the range of the outer loop.

CONTINUE STATEMENT

The CONTINUE statement transfers control to the next executable statement.

The form of the CONTINUE statement is as follows:

```
CONTINUE
```

CONTINUE is frequently used as the terminal statement in a DO statement range when the statement which would normally be the terminal statement is one of those which are not allowed.

STOP STATEMENT

The STOP statement is used to terminate program execution.

A STOP statement has one of the following forms:

```
        STOP
   or
        STOP c
```

where c is any string of one to six characters. When STOP is-encountered during execution of the object program, the characters c (if present) are displayed on the operator control console and execution of the program terminates. The STOP statement, therefore, constitutes the logical end of the program.

PAUSE STATEMENT

The PAUSE statement temporarily suspends program execution.

A PAUSE statement has one of the following forms:

```
          PAUSE
or
          PAUSE c
```

where c is any string of up to six characters. When PAUSE is encountered during execution of the object program, the characters c (if present) are displayed on the operator control console and execution of the program ceases. Execution may be terminated by typing a "T" and a carriage return at the operator console. Typing any other character and a carriage return will cause execution to resume.

CALL STATEMENT

CALL statements control transfers into SUBROUTINE subprograms and provide parameters for use by the subprograms.

A call statement has one of the following forms:

```
          CALL s(a1,a2,...an)
or
          CALL s
```

Where s is the SUBROUTINE name and the ai are the actual arguments to be used by the subprogram. A detailed discussion of CALL statements appears in Chapter 10, "Functions and Subprograms".

RETURN STATEMENT

The logical termination of a subprogram is a RETURN statement.

The general form of the RETURN statement is: **RETURN**

A more detailed discussion of the form, use and interpretation of the RETURN statement is in Chapter 10, "Functions and Subprograms".

END STATEMENT

The END statement must physically be the last statement of any FORTRAN program.

It has the following form:

```
END
```

The END statement is an executable statement and may have a statement label. When it is encountered at the end of a main program unit, it causes a transfer of control to be made to the system exit routine, $EX, which returns control to the operating system.

Chapter Seven

INPUT/OUTPUT STATEMENTS

OVERVIEW

FORTRAN I/O is performed with the READ and WRITE statements. The READ statement is used for input, and the WRITE statement is used for output. Unformatted I/O will transmit binary information. Formatted I/O is used in conjunction with format specifications to control data editing and conversions.

The READ and WRITE statements reference the HDOS I/O devices via a logical unit number. Some logical unit numbers are pre-assigned to specific devices.

READ and WRITE statements are grouped as follows:

1. Unformatted Sequential I/O

   Specifies binary data transmission using a sequential I/O device.

2. Formatted Sequential I/O

   Specifies character data transmission using a sequential I/O device. A FORMAT statement must be referenced in order to control the editing and translation of data.

3. Unformatted Random I/O

   Specifies binary data transmission using a random access I/O device.

4. Formatted Random I/O

   Specifies character data transmission using a random access I/O device. A FORMAT statement must be referenced in order to control the editing and translation of data.

LOGICAL UNIT NUMBERS (LUN)

FORTRAN communicates with the HDOS I/O devices via a Logical Unit Number (LUN). The Logical Unit Number must be an unsigned integer number or integer variable in the range 1-10. If an integer variable is used, an integer value must have been assigned to the variable prior to execution of the I/O statement.

The Logical Unit Numbers 1-10 have been pre-assigned to specific HDOS I/O devices. Logical Unit Numbers 1, 3, 4, and 5 refer to the console terminal. Unit 2 is assigned to a hardcopy device. Units 6-10 are assigned to disk files. The following table summarizes these assignments:

LUN	**HDOS Device Name**
1	TT:
2	LP:
3,4,5	TT:
6	SY0:FORT06.DAT
7	SY0:FORT07.DAT
8	SY0:FORT08.DAT
9	SY0:FORT09.DAT
10	SY0:FORT10.DAT

Changing the Logical Unit default assignment is done with the OPEN subroutine. The following FORTRAN statement illustrates this point:

```
        CALL OPEN (3,'SY1:INPUT.DATΔ')       (Δ=blank or space)
```

This would associate LUN 3 with the file SY1:INPUT.DAT. Subsequent I/O to this file would be accomplished by referencing unit 3 in the READ or WRITE statement.

```
        CALL OPEN (10,`FILENAME.DATΔ')       (Δ=blank or space)
```

Note that in both of these examples the file name was terminated with a blank. **This blank must precede the right hand quote.** If this blank is not present, communication with the HDOS I/O device will not be established.

Output to Hard Copy Devices

FORTRAN uses the HDOS device driver to communicate to a hard copy device. LUN 2 is the pre-assigned hard copy device number. The device driver associated with LUN 2 must always have the file name "LP.DVD". In addition, the device driver must be loaded into memory before execution of the FORTRAN program begins. The HDOS command to load the device driver is:

```
>LOAD dev:
```

Where "dev:" is the two letter name used by HDOS to reference the device. For example to load the device driver named LP.DVD, the HDOS command would be:

```
>LOAD LP:
```

In order to use a device driver other than LP:, the device driver name must be changed from its present file name to LP.DVD. This can be accomplished with the HDOS utility PIP.

An alternate method would be to OPEN another file using the present device driver file name. For example, if output to a hard copy device using the driver called AT.DVD is desired, the following FORTRAN statement could be used:

```
CALL OPEN(7,'AT: ')       (Note the blank!)
```

The AT: driver must be loaded into memory before execution of the FORTRAN program. Subsequent output statements to the device AT: would refer to Logical Unit Number 7.

FORMAT Specifiers

All Formatted I/O statements reference FORMAT statements to define the conversion and editing performed during the transmission of data. The reference can either be a statement label of a FORMAT statement or an array name containing the format specifiers. FORMAT statements are discussed in Chapter 8.

INPUT/OUTPUT LISTS

Most forms of READ/WRITE statements may contain an ordered list of data names which identify the data to be transmitted. The order in which the list items appear is the same as that in which they will be transmitted.

Lists have the following form:

```
m1,m2,...,mn
```

where the mi are list items separated by commas, as shown.

Lengths of I/O Lists

The length of an I/O list refers to the number of bytes needed to store the elements of the list. With several forms of I/O it is very important to know the length of the I/O list. The actual number of bytes required by an I/O list is a function of the number and data type of each individual I/O element. The following table illustrates this relationship.

Data Type	**Bytes Required per Element**
LOGICAL	1
INTEGER	2
REAL	4
DOUBLE PRECISION	8

Simple Lists

A simple I/O list consists of the name of a variable, an array element or array name. One or more of these items may be enclosed in parentheses without changing their intended meaning.

Example:

```
        A
        C(1,2)
```

An array name appearing in a list without subscript(s) is considered equivalent to the listing of each successive element of the array.

Example:

If B is a two dimensional array, the list item B is equivalent to:

```
    B(1,1),B(2,1),B(3,1). . . . , B(1,2),B(2,2)...,B(j,k).
```

where j and k are the subscript limits of B.

Implied DO Lists

Implied DO lists provide for iteration within an I/O list. Implied DO lists are of the form:

```
        (I/O list, i = m1,m2,m3)

  or    (I/O list, i = m1,m2)
```

The elements i,m1,m2,m3 have the same meaning as defined for the DO statement. The implied DO list functions as though the I/O statement was within the range of a DO loop. (For more information on the DO statement, refer to Chapter 6.) The DO implication applies to all list items enclosed in parentheses with the implication.

Examples:

Implied Do Lists	**Equivalent Lists**
(X(I),I=1,4)	X(1),X(2),X(3),X(4)
(Q(J),R(J),J=1,2)	Q(1),R(1),Q(2),R(2)
(G(K),K=1,7,3)	G(1),G(4),G(7)

((A(I,J),I=3,5),J=1,9,4)	A(3,1),A(4,1),A(5,1)
	A(3,5),A(4,5),A(5,5)
	A(3,9),A(4,9),A(5,9)

(R(M),M = 1,2),I,ZAP(3)	R(1),R(2),I,ZAP(3)
(R(3),T(I),I=1,3)	R(3),T(1),R(3),T(2), R(3),T(3)

For example, the elements of an array may be transmitted in an order different from the order in which they appear in storage. The array A(3,3) occupies storage in the order:

```
A(1,1),A(2,1),A(3,1), A(1,2),A(2,2),A(3,2) A(1,3),A(2,3),A(3,3).
```

By specifying the transmission of the array with this DO-implied list item:

```
((A(I,J),J=1,3),I=1,3)
```

the order of transmission is:

```
A(1,1),A(1,2),A(1,3),
A(2,1),A(2,2),A(2,3),
A(3,1),A(3,2),A(3,3).
```

SEQUENTIAL I/O

Unformatted Sequential I/O

Unformatted sequential I/O is used to transfer binary data without any data editing or formatting. The amount of data transferred is a function of the number and data type of the elements in the I/O list.

An I/O list must be included in the I/O statement.. Failure to provide an I/O list will result in a compiler error.

The two forms of the unformatted sequential I/O statement are:

```
READ(u,ERR=L1,END=L2) k

WRITE(u,ERR=L1,END=L2) k
```

where:

u	specifies a Logical Unit Number.
L1	specifies an I/O error branch. (optional)
L2	specifies an EOF branch. (optional)
k	is an I/O list.

The length of an unformatted sequential record may be greater than 256 bytes if so desired. If the record is greater than 256 bytes, some special considerations must be noted.

NOTE: Special Considerations

The unformatted sequential I/O processor assumes a 256-byte record. Any I/O to an unformatted sequential file will position the record pointer at the end of a 256-byte physical record. This can cause unexpected results if a different number of bytes are read than were originally written.

For example, assume that several 280-byte records were written to an unformatted sequential file. Now assume that only 200 bytes are read from the first 280-byte record in this file. The 200-byte record will be input and assigned to the elements in the I/O list. The record pointer will then be positioned at the beginning of the next 256-byte physical record. Note that this will result in the excess data being read in subsequent READ statements. The excess data will NOT be skipped over.

The best way to avoid this is simply to keep the length of the input and output records the same. If this simple precaution is adhered to, then more than 256 bytes can be transmitted using unformatted sequential I/O.

READ (unformatted sequential input)

The unformatted sequential READ will read one logical record. The logical record may extend across more than one physical record. Some special considerations must be noted if the logical record size is greater than 256 bytes. See the preceding note on "Special Considerations").

The amount of data transmitted corresponds to the number and data type of the elements in the I/O list K.

If there are as many elements in the I/O list as fields in the input record, the entire record is read. If there are fewer elements in the I/O list than fields in the input record, then the unread items in the record are skipped.

If there are more elements in the I/O list than fields in the input record, then as many records as are necessary to fill the I/O list will be read.

Example:

```
READ(6) A,B,C
```

Reads one record from the external storage device associated with logical unit number 6. Assign the input data to the variables A,B and C.

WRITE (unformatted sequential output)

The output statement (WRITE) will output the data specified in the I/O list k to the file referenced by the logical unit number u. One logical record will be output every time the WRITE is executed. (If this logical record is more than 256 bytes, then some special considerations must be noted. See the preceding note on "Special Considerations").

Example:

```
WRITE(6) I,J,K,L
```

The data assigned to the integer variables I,J,K and L are output to the file associated with logical unit number 6. This statement will output 256 bytes, but only the first 8 will contain data. (4 integer variables x 2 bytes/integer element). The remaining 248 bytes will contain the ASCII NUL character (0).

Formatted Sequential I/O

Formatted sequential I/O is used to transfer character data. The transferring of data is done in a sequential manner. A FORMAT statement must be referenced in order to control the editing and formating of the data. (See FORMAT statements, Chapter 8.)

Two forms of the statement are available:

```
READ (u,f,ERR=L1,END=L2) k

WRITE (u,f,ERR=L1,END=L2) k
```

where:

u	Specifies a Logical Unit Number
f	Specifies the label of a FORMAT statement
L1	Specifies an I/O error branch. (optional)
L2	Specifies an EOF branch. (optional)
K	Is an I/O list (optional)

The size of each I/O record must not be longer than 255 bytes. An attempt to write more than 255 bytes per record will result in the record being truncated to exactly 255 bytes. If the record is long enough to overflow the internal I/O buffer, a runtime error will result.

An attempt to read more than 255 bytes per record will result in the internal I/O buffer overflowing, thus causing a runtime I/O error.

The FORTRAN programmer is responsible for verifying that the length of the record is less than or equal to 255 bytes. The length of the record is a function of the number and data type of the elements in the I/O list. (The subject of I/O lists is covered on pages 7-4 and 7-5).

Use of the END= and the ERR= statements is optional. The ERR= branch will be taken only when a hardware related I/O error occurs. The END= branch will be taken only when an end of file condition occurs. If these options are omitted, hardware related errors and end of file conditions will cause fatal runtime errors. Thus, execution of the program will be terminated.

READ (formatted sequential input)

The input statement (READ) when used with a variable list, will input a number of items, corresponding to the elements in the list k. The data will be edited and converted according to the specifications of the format statement labeled f.

Each time execution of the READ statement begins, a new record from the input file is read. The number of records to be input by a single READ statement is determined by the list, k, and format specifications. The length of each record must not exceed 255 bytes.

If there are as many elements in the I/O list as fields in the input record, then the entire record is read. If there are fewer elements in the I/O list then fields in the input record, the unread fields are skipped.

If there are more elements in the I/O list than fields in the input record, then as many records as are necessary to fill the I/O list will be read.

Example:

```
READ(6,100) A,B,C,D 100 FORMAT(4F6.2)
```

The above program segment will input data from the external storage device associated with logical unit number 6. The input data will be assigned to the variables A,B,C and D. The input data will be formatted according to the FORMAT statement referenced by the label 100.

The input statement (READ) when used without a variable list, will read literal data into an existing literal field. The FORMAT statement referenced in the READ statement will contain the new literal field.

Example:

```
     READ(6,100)
        .
        .
        .
100  FORMAT(10H1234567890)
```

This will result in the next 10 characters of the file associated with logical unit number 6 being read. The input data will replace the characters "1234567890" in the FORMAT statement.

WRITE (formatted sequential output)

The output statement (WRITE), when used with a variable list, will output the data specified in the list k to the file referenced by logical unit u. The FORMAT statement f will specify the external representation of the data.

As many records as are desired may be output with a single WRITE statement. The number of records output will be determined by the list and FORMAT specifications. Successive data are output until the data specified in the list are exhausted. The programmer is responsible for verifying that the length of the record does not exceed 255 bytes.

The first character of the FORMAT statement is assumed to be the carriage control character. When writing to a disk file, it is good programming practice to use a plus (+) for the carriage control character. If this is not used, then a carriage control character will be stored as the first field of each record. (See Chapter 8, FORMAT statements)

Example:

```
      WRITE(3,10)A,B,C,D
       .
       .
       .
10    FORMAT('+',4A4)
```

The data assigned to the variables A, B, C and D are output to Logical Unit Number 3, formatted according to the FORMAT statement labeled `10`. Note the use of the plus (+) carriage control character.

The output statement (WRITE), when used without a variable list, may be used to write alphanumeric information when the characters to be written are specified within the FORMAT statement. For example, to write the characters 'A CONVERSION' on unit `1`, the following program segment could be used:

```
   WRITE(1,26)
    .
    .
    .
26 FORMAT ('1',12HA CONVERSION)
```

RANDOM I/O

Unformatted Random I/O

For an unformatted random disk access, the record number desired is specified by using the REC=n option in the READ or WRITE statement. This type of I/O allows direct access to the nth record of a disk file.

Record number n may be an integer variable or integer constant. If an integer variable is used, an integer value must have been assigned to the variable prior to the execution of the I/O statement.

The transmission of data is done without any editing or formatting. Exactly one physical record (256 bytes) is transmitted.

The two types of unformatted random I/O are:

```
READ (u ,REC=n,ERR=L1,END=L2) k (input)
WRITE(u,REC=n,ERR=L1,END=L2) k (output)
```

where:

u	specifies a Logical Unit Number.
n	specifies the record no. to access.
L1	specifies an I/O error branch. (optional)
L2	specifies an EOF branch. (optional)
k	is an I/O list.

If the integer value specifying the record number is negative or zero, a runtime I/O error will result.

The Logical Unit Number u must reference a disk file. An attempt to randomly access a device that is not capable of random access will result in a runtime I/O error.

Use of the ERR= and END= statements is optional. The ERR= branch will be taken only when a hardware related I/O error occurs. The END= branch will be taken only when an end of file condition occurs.

If these options are omitted, hardware related errors and end of file conditions will cause fatal runtime errors. Thus, execution of the program will be terminated.

READ (unformatted random input)

The input statement (READ) will input the record corresponding to the integer value n and assign the data to the elements of the I/O list k.

If the I/O list contains as many elements as fields in the input record, then the entire record is read. If the I/O list contains fewer elements than fields in the input record, then the unread items are skipped.

If the I/O list contains more elements than fields in the input record, then the same record is read as many times as necessary to fill the I/O list.

Example:

```
READ(6,REC=3) I,J,K
```

Read record number 3 from the file associated with Logical Unit Number 6. Assign the input data to the integer variables I,J and K. (Note that 6 bytes will be read- 3 integer variables x 2 bytes per integer variable.)

WRITE (unformatted random output)

The output statement (WRITE) will output the values referenced by the I/O list k to the record corresponding to the integer variable n. If a previous record number n exists, it will be written over. If no record number n exists, the file will be extended to create one.

The amount of data output will correspond to the type and number of elements in the I/O list k. If the amount of data output is less than 2 56 bytes, the record will be padded with ASCII NUL characters (0) so that it is exactly 256 bytes.

If more than 256 bytes are output, the original 256 byte record will be written over until all the elements in the I/O list are output. This will result in some or all of the data being destroyed.

It is the responsibility of the programmer to insure that no more than 256 bytes are output. The amount of data output is a function of the number and type of elements in the I/O list.

Example:

```
        WRITE(6,REC=4) J,K,A
```

Write record number 4 to the file associated with Logical Unit Number 6. Note that 256 bytes will be written, but only the first 8 will contain data. (2 integer variables x 2 bytes/integer variable + (1 real variable x 4 bytes/real variable). The remaining 248 bytes will contain ASCII NUL characters (0).)

Formatted Random I/O

Formatted random I/O is used to directly access character data using a random access device. The REC=n option is used in the READ or WRITE statement to allow access to the nth record of the file. Exactly one. physical record (255 bytes for a formatted file) is transmitted.

A format statement is referenced to control the editing and formatting of the data during transmission.

It is recommended that the carriage control be suppressed by using the + character for carriage control. If the + character is used, then no carriage control character will be stored with the record. For a complete discussion of carriage control characters, refer to Chapter 8.

The two forms of formatted random I/O are:

```
READ(u,f,REC=n,ERR=L1,END=L2)k (input)
WRITE(u,f,REC=n,ERR=L1,END=L2)k (output)
```

where:

u	specifies a Logical Unit Number.
f	specifies the label of a FORMAT statement.
n	specifies the record number to access.
L1	specifies an I/O error branch (optional).
L2	specifies an EOF branch (optional).
k	is an I/O list

If the integer value specifying the record number is negative or zero, a runtime I/O error will result.

The Logical Unit Number must reference a disk file. An attempt to randomly access a device that is not capable of random access will result in a runtime I/O error.

Use of the ERR= and END= statements is optional. The ERR= branch will only be taken when a hardware related I/O error occurs. The END= branch will only be taken when an end of file condition occurs.

If these options are omitted, hardware related errors and end-of-file conditions will cause fatal runtime errors. Thus, execution of the program will be terminated.

READ (formatted random input)

The input statement (READ) will input the record corresponding to the integer value n and assign the data to the elements in the I/O list k. The data will be edited according to the specifications of the FORMAT statement referenced by the label f.

If there are as many elements in the I/O list as fields in the input record, then the entire record will be read. If there are fewer elements in the I/O list than fields in the record, the unread items will be skipped.

If there are more elements in the I/O list than fields in the input record, the input record will be read as many times as necessary to fill the I/O list.

It is important to note that no more than 255 bytes can be input with a formatted random read statement. Although each HDOS sector contains 256 bytes, only 255 are available with formatted random I/O. The byte following the data is assigned by the FORTRAN I/O processor to be a line feed. (ASCII 0A)

Upon execution of a formatted random read, the FORTRAN I/O processor must find this line feed character. If the processor can not find this character, a runtime I/O error will result.

Because of this, **the only files that can be read with a formatted random read are those which are created using a formatted random write.** An attempt to read a file not created with a formatted random write will usually result in a runtime I/O error.

Examples:

```
          READ(6,100,REC=2) A,B,C
            .
            .
            .
      100 FORMAT(3A4)
```

Read record number 2 from Logical Unit Number 6. Assign the input values to the variables A,B and C.

WRITE (formatted random output)

The output statement (WRITE) will output the values associated with the I/O list *k* to the record corresponding to the integer value n. If a previous record number n exists, it will be written over. If no record number n exists, the file will be extended to create one.

The amount of data transmitted will correspond to the number and type of elements in the I/O list *k*. If the amount is less than *255* bytes, the record will be padded with the ASCII NUL character (0) so that it is exactly *255* bytes. The last data byte will be a carriage return character. The FORTRAN I/O processor will generate this character for subsequent use when the file is READ.

NOTE: The programmer is responsible for verifying that only *255* bytes are output. This *255* byte record also includes the carriage control character. If a + is used for carriage control, then the user has access to all *255* bytes in the record.

If the carriage control character is omitted, the I/O processor will generate an ASCII line feed character (Hex -- 0A). The first byte of the output file will contain this character, thus leaving *254* bytes for use.

If more than *255* bytes are written, the FORTRAN I/O processor will be unable to generate the line feed character to mark the end of the record. Although no error will be displayed during execution of the WRITE statement, subsequent READ's to this file will generate runtime I/O errors.

Examples:

```
      DIMENSION J(120)
       .
       .
       .
      I=6
      WRITE(6,100,REC=I) J
100   FORMAT('+',120A2)
```

This statement will write the contents of the 120 element array j to the 6th record of the file associated with Logical Unit Number 6. No carriage control character will be stored with the record. 240 bytes will be written with this statement. The next byte stored will be the line feed character generated by the FORTRAN I/O processor. The remaining 15 bytes will be ASCII NUL (0) characters.

AUXILIARY I/O STATEMENTS

FORTRAN provides several auxiliary I/O statements to perform various file management functions.

OPEN Subroutine

A file may be OPENed using the OPEN subroutine. LUNs 1-5 may also be assigned to disk files with OPEN. The OPEN subroutine allows the program to specify a file name and device to be associated with a LUN.

An OPEN of a non-existent file creates a null file of the appropriate name. An OPEN of an existing file followed by sequential output deletes the existing file. An OPEN of an existing file followed by an input allows access to the current contents of the file. The open subroutine is of the form:

```
CALL OPEN(u,filename)
```

where:

u is the logical unit number to be associated with the file. u must be an unsigned integer constant or integer variable with a value in the range 1-10 inclusive. If an integer variable is used, a value must be assigned to the variable prior to CALLing the OPEN subroutine.

"filename" is an ASCII name which HDOS will associate with the file. The file name should be a Hollerith or literal constant, or a variable or array name which contains the ASCII file name. The filename must be a valid HDOS file name. The file name must be terminated by a blank (or another non-alphanumeric character).

Examples:

```
CALL OPEN (7,'SY1:INPUT.DAT ')
CALL OPEN (3, ANAME)
CALL OPEN(6,'SY2:TEST.DAT ')
CALL OPEN(3,'AT: ')
```

ENDFILE Statement

ENDFILE writes the end of file mark and then closes the file associated with LUN u. The ENDFILE statement is of the form:

```
ENDFILE u
```

REWIND Statement

REWIND closes the file associated with LUN u, then opens it again. The REWIND statement is of the form:

```
REWIND u
```

ENCODE/DECODE Statements

ENCODE and DECODE statements transfer data, according to format specifications, from one section of memory to another. DECODE changes data from internal format to the specified format. ENCODE changes data of the specified format into internal format. The two statements are of the form:

```
ENCODE(A,F) K
DECODE(A,F) K
```

where:

A is an array name
F is FORMAT statement number
K is an I/O List

DECODE is analogous to a READ statement, since it causes the character data in the array A to be converted according to the format specifications and then assigned to the elements in the I/O list K. The format specifications are referenced by the statement number F.

ENCODE is analogous to a WRITE statement, since it causes the elements in the I/O list K to be translated to character format and stored in array A. The FORMAT statement referenced by F will control the translation process.

The total number of characters that can be processed by an ENCODE or DECODE statement is determined by the data type of the array A. The following table illustrates this relationship.

Data Type	**Characters per Array Element**
LOGICAL	1
INTEGER	2
REAL	4
DOUBLE PRECISION	8

Care should be taken that the array A is always large enough to contain all of the data being processed. There is no check for overflow. An ENCODE operation which overflows the array will probably wipe out important data following the array. A DECODE operation which overflows will attempt to process the data following the array.

Chapter Eight

FORMAT Statements

OVERVIEW

FORMAT statements are non-executable statements used in conjunction with formatted I/O and with ENCODE and DECODE statements. They specify conversion methods and data editing information. FORMAT statements require statement labels for reference.

The general form of a FORMAT statement is as follows:

```
            m FORMAT (si,...,sn)
```

where m is the statement label and each si is a field descriptor.

The word FORMAT and the parentheses must be present as shown.

FIELD DESCRIPTORS

Field descriptors describe the sizes of data fields and specify the type of conversion to be exercised upon each transmitted datum. The FORMAT field descriptors may have any of the following forms:

Descriptor	**Classification**
rEw.d	Numeric Conversion
rFw.d	
rGw.d	
Dw.d	
rIw	
rAw	Hollerith Conversion
nHhlh2...hn	
l1,l2 ...ln'	
rLw	Logical Conversion
nX	Spacing Specification
mP	Scaling Factor

w is a positive integer constant defining the field width (including digits, decimal points, algebraic signs) in the external data representation.

d is an integer specifying the number of fractional digits appearing in the external data representation.

The characters F, G, E, D, I, A and L indicate the type of conversion to be applied to the items in an input/output list.

r is an optional, non-zero integer indicating that the descriptor will be repeated r times.

The hi and li are characters from the FORTRAN character set.

m is an integer constant (positive, negative, or zero) indicating scaling.

n is a positive integer constant defining the number of spaces to insert in the I/O record.

NUMERIC CONVERSIONS

Input operations with any of the numeric conversions will allow the data to be represented in a "Free Format"; i.e., commas may be used to separate the fields in the external representation.

F-type Conversion

Real or double-precision type data are processed using this conversion. w characters are processed of which *d* characters are considered fractional.

Form: *Fw.d*

F-INPUT

Data values which are to be processed under *F* conversion can follow a relatively loose format. The format is as follows:

1. Leading spaces (ignored)
2. A+or- sign (an unsigned input is assumed to be positive)
3. A string of digits
4. A decimal point
5. A second string of digits
6. The letter E (exponent indicator)
7. A + or - sign
8. An integer exponent

The following conditions must be observed:

If the integer exponent is present, then the exponent indicator and the sign (+ or -) must also be included. (If the sign is omitted, it is assumed to be positive.)

All non-leading spaces are considered zeros.

Input data can be any number of digits in length, and correct magnitudes will be developed, but precision will be maintained only to the extent specified in Chapter 3, "Data Representation/Storage Format", for real data.

F-input Examples:

FORMAT	**Input Value**	**Internal Value**
F8.5	234562341	234.56234
F6.2	12.123	12.123
F9.2	89.56E+3	89560.0
F5.2	1234567.89	123.45
F8.5	-12345678	-12.34567

Note in the above examples that if no decimal point is given among the input characters, the *d* in the FORMAT specification establishes the decimal point. If a decimal point is included in the input characters, the *d* specification is ignored.

F-OUTPUT

Values are converted and output as: a minus sign (if negative), followed by the integer portion of the number, a decimal point and *d* digits of the fractional portion of the number.

If a value does not fill the field, it is right-justified in the field and enough preceding blanks to fill the field are inserted.

If a value requires more field positions than allowed by w, a runtime error will result.

F-output examples:

FORMAT	**Internal Value**	**Output (b=blank)**
F10.4	368.42	bb368.4200
F7.1	-4786.361	-4786.4
F8.4	8.7E-2	bbb.0870
F6.4	4739.76	**FW**

E-Type Conversion

Form: Ew.d

Real or double-precision type data are processed using this conversion. w characters are processed of which d characters are considered fractional. The transmission of data is in exponential form.

E-INPUT

Data values which are to be processed under E conversion are edited in exactly the same way as with the F field descriptor.

E-OUTPUT

Values are converted, rounded to d digits, and output as:

1. a minus sign (if negative)
2. a decimal point
3. d decimal digits
4. the letter E (exponent indicator)
5. the sign of the exponent (minus or plus)
6. two exponential digits

The values as described are right-justified in the field w with preceding blanks to fill the field if necessary. The field width w should satisfy the relationship:

```
w >= d + 7
```

Otherwise a runtime error will result.

E-output examples:

FORMAT	Internal Value	Output (b=blank)
E12.5	76.573	bb.76573E+02
E14.7	-32672.354	bb-3267235E+05
E7.3	56.93	**FW**
E13.4	-0.0012321	bbb-.1232E-02
E9.2	76321.73	bb.76E+05

D-Type Conversions

Form: Dw.d

D-Input

D-Input functions in the same manner as E-input except the input data is converted and assigned to a double-precision data type.

D-input examples:

FORMAT	Input Value	Internal Value
D10.2	23456bbbbb	23456000.0D0
D10.2	bb234.56bb	234.56D0
D15.3	123.5678901D+04	1.235678901D+06

D-Output

D-Output functions in the same manner as E-output except that the D exponent indicator is used in place of the E exponent indicator.

D-output examples:

FORMAT	Internal Value	Output (b=blank)
D12.5	76.573	bb.76573D+02
D14.7	-32672.354	-.32672354D+05
D13.4	-0.0012321	bbb-.12321D-02
D8.2	76321.73	bb.76D+05

G-Type Conversions

Form: Gw.d

Real or double-precision type data are processed using this conversion. w characters are processed of which d characters are considered fractional.

G-INPUT

The G descriptor edits input data in exactly the same manner as the F descriptor.

G-OUTPUT

The method of output conversion is a function of the magnitude of the number being output. This method is useful when the magnitude of the number is not known beforehand.

The following table shows how the number will be output, where n is the magnitude of the number:

Magnitude	Equivalent Conversion (b=blank)
n < 0.1	Ew.d
.1 <= n < 1	F(w-4).dbbbb
1 <= n < 10	F(w-4).(d-1)bbbb
.	.
.	.
.	.
$10d^{-2}$ <= n < $10d^{-1}$	F(w-4).1bbbb
$10d^{-1}$ <= n < 10d	F(w-4).0bbbb
n > 10d	Ew.d

G-output examples:

FORMAT	Internal Value	Output (b=blank)
G13.6	0.01234567	bb.123457E-01
G13.6	123.45678901	bb123.457bbbb
G13.6	123456.7890	bb123457.bbbb
G13.6	-1234567.89012345	b-.123457E+07

For comparison, the following examples illustrate the same values output under the F field descriptor.

FORMAT	Internal Value	Output (b=blank)
F13.6	0.01234567	bbbbb.0123456
F13.6	123.45678901	bbb123.456789
F13.6	123456.7890	123456.789000
F13.6	-1234567.89012345	**FW**

Note in the last example that the F format descriptor field was too small for the magnitude of the data. In this case a runtime error will result.

I-Type Conversions

Form: Iw

This descriptor specifies the transmission of integer data.

I-INPUT

A field of w characters is input and converted to internal integer format. A minus sign may precede the integer digits. If a sign is not present, the value is considered positive. Integer values in the range -32768 to 32767 are accepted. Non-leading spaces are treated as zeros.

I-input examples:

FORMAT	Input (b=blank)	Internal Value
14	b124	124
14	-124	-124
17	bbb732b	7320

I-OUTPUT

Values are converted to integer constants. Negative values are preceded by a minus sign. If the value does not fill the field, it is right-justified in the field and enough preceding blanks to fill the field are inserted. If the value exceeds the field width, a runtime error will result.

I-output examples:

FORMAT	Internal Value	Output (b=blank)
16	+281	bbb281
16	-23261	-23261
13	126	126
14	-226	-226
13	1234	**FW**

HOLLERITH CONVERSIONS

A-Type Conversion

The form of the A conversion is as follows:

Form: Aw

This descriptor causes unmodified Hollerith characters to be read into or written from a specified list item.

The maximum number of actual characters which may be transmitted using Aw is equal to the number of bytes needed to store the corresponding list item. (i.e., 1 character for logical items, 2 characters for integer items, 4 characters for real items and 8 characters for double-precision items). Refer to Chapter 3, "Data Representation/Storage Format", for a discussion of the storage requirements of the various data types.

A-INPUT

If w is greater than n (where n is the number of bytes of storage required by the corresponding list item), the rightmost n characters are taken from the external input field.

If w is less than n, the w characters appear left justified with w-n trailing blanks in the internal representation.

A-input examples:

Format	Input	Type	Internal Value (b=blank)
A1	A	Integer	Ab
A4	ABCD	Integer	AB
A1	A	Real	Abbb
A7	ABCDEFG	Real	DEFG

A-OUTPUT

If w is greater than n the external output field will consist of w-n blanks followed by the n characters from the internal representation.

If w is less than n, the external output field will consist of the leftmost w characters from the internal representation.

A-output examples:

FORMAT	Internal Value	Type	Output
Al	AZ	Integer	A
A2	AB	Integer	AB
A3	ABCD	Real	ABC

H-Type Conversion

The forms of H conversion are as follows:

```
nHhlh2...hn
'h1h2...hn'
```

These descriptors process hollerith character strings between the descriptor and the external field, where each h represents any character from the ASCII character set.

Special consideration is required if an apostrophe (') is to be used within the literal string in the second form. An apostrophe character within the string is represented by two successive apostrophes. See the examples on the following page.

H-INPUT

The n characters of the string hi are replaced by the next n characters from the input record. This results in a new string of characters in the field descriptor. See the last example on Page 7-11 for more information.

H-input examples:

FORMAT	**Input(b =blank)**	**Resultant descriptor (b=blank)**
4H1234	ABCD	4HABCD
7HbbFALSE	TRUEbbb	7HTRUEbbb

H-OUTPUT

The n characters hi, are placed in the external field. In the nHh1h2...hn form the number of characters in the string must be exactly as specified by n. Otherwise, characters from other descriptors will be taken as part of the string.

In both forms, blanks are counted as characters.

H-output examples:

Format(b=blank)		**Output (b=blank)**
1HA	or 'A'	A
8HbSTRINGb	or 'bSTRINGb'	bSTRINGb
11HX(2,3)=12.0	or 'X(2,3)=12.0'	X(2,3)=12.0
12HIbSHOULDN'T	or 'IbSHOULDN''T'	IbSHOULDN'T

LOGICAL CONVERSIONS

The form of the logical conversion is as follows:

Form: Lw

L-Input

The external representation occupies w positions. It consists of optional blanks followed by a "T" or "F", followed by optional characters.

L-Output

If the value of an item in an output list corresponding to this descriptor is 0, an F will be output; otherwise, a T will be output. If w is greater than 1, w-1 leading blanks precede the letters.

L-Output examples:

FORMAT	Internal Value	Output(b=blank)
L1	<>0	T
L5	<>0	bbbbT
L7	=0	bbbbbbF

X DESCRIPTOR

The form of the X descriptor is as follows:

Form: nX

This descriptor causes no conversion to occur, nor does it correspond to an item in an input/output list. When used for output, it causes n blanks to be inserted in the output record. Under input circumstances, this descriptor causes the next n characters of the input record to be skipped.

X-input example:

FORMAT	**Input**	**External values**
10 FORMAT (F4.1,3X,F3.0)	12.5ABC120	12.5,120

X-output example:

FORMAT	**Output(b=blank)**
3 FORMAT (1HA,4X,2HBC)	AbbbbBC

SCALE FACTOR

The P descriptor is used to specify a scaling factor for real conversions.

Form: nP

where n is an integer constant (positive, negative, or zero).

The scale factor must precede the field descriptor with which it is used. Once a scale factor is specified, it applies to all real conversions encountered in the FORMAT statement. The scale factor remains unchanged until another P descriptor is encountered or the I/O terminates. The scale factor may be disabled by specifying a scale factor of 0.

Effects of Scale Factor on Input:

During input the scale factor produces the following result:

Value assigned to I/O list. element = external value / 10**n

If an exponent is present in the external data field, the scale factor will have no effect.

Examples:

FORMAT	Input	Internal Value
2PF9.5	bbb45.123	.45123
-2PF9.5	bbb45.123	4512.3

Effect of Scale Factor on Output:

E-OUTPUT, D-OUTPUT:

The coefficient is shifted left n places relative to the decimal point, and the exponent is reduced by n (the value remains the same).

F-OUTPUT

The external value will be 10**n times the internal value.

G-OUTPUT

The scale factor is ignored if the internal value is within the range to be output using F conversion. Otherwise, the effect is the same as for E output.

Examples:

FORMAT	Internal Value	Output(b=blank)
1PE12.5	34.567	b3.45670E+01
2PF9.3	12.345	b1234.500
3PG9.3	10.00	b10.0bb

OTHER CONTROL FEATURES OF FORMAT STATEMENTS

Repeat Specifications

The E, F, D, G, I, L, X and A field descriptors may be indicated as repetitive descriptors by using a repeat count r in the form rEw.d, rFw.d, rGw.d, rIw, rLw, rX and rAw.

Example:

The statement:	600 FORMAT(A2,A2,I2,I2,I2,F6.2,F6.2)
Is equivalent to:	600 FORMAT(2A2,3I2,2F6.2)

Repetition of a group of field descriptors is accomplished by enclosing the group in parentheses preceded by a repeat count. Absence of a repeat count indicates a count of one.

Up to two levels of parentheses, including the parentheses required by the FORMAT statement, are permitted.

Example:

The statement:	600 FORMAT(2(4X,I2,F5.2),A4)
Is equivalent to:	600 FORMAT(4X,I2,F5.2,4X,I2,F5.2,A4)

Repetition of FORMAT descriptors is also initiated when all descriptors in the FORMAT statement have been used but there are still items in the input/output list that have not been processed.

When this occurs the FORMAT descriptors are re-used starting at the first opening parenthesis in the FORMAT statement. A repeat count preceding the parenthesized descriptor(s) to be re-used is also active in the re-use.

This type of repetitive use of FORMAT descriptors terminates processing of the current record and initiates the processing of a new record each time the re-use begins.

Field Separators

Two adjacent descriptors must be separated in the FORMAT statement by either a comma or one or more slashes.

Example:

```
        2A4,F6.3 or 2A4/F6.3
```

The comma is used simply to separate the fields in the FORMAT statement. The slash not only separates field descriptors, but it also specifies the demarcation of formatted records.

Each slash terminates a record and sets up the next record for processing. When the slash is encountered during input, the remainder of the input record is ignored. When the slash is encountered during output, the remainder of the output record is filled with ASCII NUL characters, and the next output record is set up for processing.

Successive slashes (///.../) cause successive records to be ignored on input and successive blank records to be written on output.

Format Carriage Control

The first character of every formatted output record is used to convey carriage control information to the output device. This is a very useful feature when performing output to a CRT terminal or a hard copy device.

The carriage control character determines what action will be taken before the line is printed. The carriage control character should be the first literal field in a FORMAT statement used for formatted output.

If no carriage control character is present, then the first character of the FORMAT statement will be used for carriage control. The options are as follows:

Character	Effect	ASCII code (hex)
0	Skip 2 lines	0D,0D
1	Insert Form Feed	OC
+	No action	(none)
Other	Skip 1 line	0A

If formatted output is to be performed to a disk file, then special consideration should be given to the carriage control character. The first field of the disk file will be determined by the carriage control character in the FORMAT statement. If the plus (+) character is used for the carriage control character, then no carriage control character will be stored with the disk file.

Example:

```
FORMAT('+',12A2)
```

The carriage control character in this FORMAT statement is very useful when outputting data to a disk file. It will suppress any carriage control character from being stored with the record.

```
FORMAT('1',6I2,2X,I2)
```

The carriage control character in this FORMAT statement will issue a form feed before the line is printed.

Format Specifications in Arrays

The FORMAT reference, f , of a formatted READ or WRITE statement may be an array name instead of a statement label. This provides the facility for altering a FORMAT specification during execution of the object program.

[f such a reference is made, the information contained in the array, taken in sequential order, must constitute a valid FORMAT specification.

The FORMAT specification which is to be inserted in the array has the same form as defined for a FORMAT statement (i.e., it begins with a left parenthesis and ends with a right parenthesis).

The FORMAT specification may be inserted in the array by use of a DATA initialization statement. It can also be inserted by use of a formatted READ statement together with an Aw FORMAT.

For example, assume the FORMAT specification:

```
(3F10.3,6I6)
```

or a similar 12 character specification which is to be stored into an array. The array must allow a minimum of 12 bytes of storage. The FORMAT could be stored in an array as follows:

```
DATA A/'(3F','10.','3,6','I6)'/
```

The following statement will READ using the format in the array:

```
READ(6,A) B,C,D,I1,I2,I3,I4,I5,I6
```

Note that the array name A was referenced instead of a statement label.

Chapter Nine

Specification Statements

OVERVIEW

Specification statements are. non-executable statements which supply determinative information to the FORTRAN compiler. This information is used to define data types of variables and arrays, specify array dimensionality and size, and to allocate data storage.

The specification statements are:

1. PROGRAM statement
2. type statement
3. EXTERNAL statement
4. DIMENSION statement
5. COMMON statements
6. EQUIVALENCE statements
7. DATA Initialization Statements

All specification statements are grouped at the beginning of a program unit and must be ordered as they appear above. The PROGRAM statement must be the first statement of the main program unit, and it must not appear in any other program unit.

The other specification statements may be preceded only by a FUNCTION, SUBROUTINE or BLOCK DATA statement. All specification statements must precede statement functions and the first executable statement.

ARRAY DECLARATORS

Three kinds of specification statements may specify array declarators. These statements are the following:

```
DIMENSION statements
type statements
COMMON statements
```

Of these, DIMENSION statements have the declaration of arrays as their sole function. The other two serve dual purposes.

Array declarators are used to specify the name, dimensionality and sizes of arrays. An array may be declared only once in a program unit. An array declarator has one of the following forms:

ui (k)
ui (k1,k2)
ui (k1,k2,k3)

where ui is the name of the array, called the declarator name, and the k's are integer constants.

Array storage allocation is established upon appearance of the array declarator. Such storage is allocated linearly by the FORTRAN compiler where the order of ascendancy is determined by the first subscript varying most rapidly and the last subscript varying least rapidly.

For example, if the array declarator AMAT(3,2,2) appears, storage is allocated for the 12 elements in the following order:

AMAT(1,1,1)
AMAT(2,1,1)
AMAT(3,1,1)
AMAT(1,2,1)
AMAT(2,2,1)
AMAT(3,2,1)
AMAT(1,1,2)
AMAT(2,1,2)
AMAT(3,1,2)
AMAT(1,2,2)
AMAT(2,2,2)
AMAT(3,2,2)

STATEMENTS

PROGRAM Statement

The PROGRAM statement provides a means of specifying a name for the main program unit. It is of the form:.

PROGRAM n where n is the program name

If present, the program statement must appear before any other statement in the main program unit. The name consists of one through six alphanumeric characters, the first of which must be a letter. If no PROGRAM statement is present in a main program, the compiler will assign a name of $MAIN to that program. During the compilation process the program name will be displayed on the console device.

type **Statement**

Variable, array and FUNCTION names are automatically typed integer or real by the 'predefined' convention unless they are changed by a type statement.

For example, the type is integer if the first letter of an item is I, J, K, L, M or N. Otherwise, the type is real. Type statements provide for overriding or confirming the predefined convention by specifying the type of an item. In addition, these statements may be used to declare arrays.

Type statements have the following general form:

```
t v1,v2,...vn
```

where t represents one of the terms:

```
BYTE
INTEGER, INTEGER* 1, INTEGER* 2
REAL, REAL*4, REAL*8
LOGICAL, LOGICAL* 1, LOGICAL* 2
DOUBLE PRECISION
```

Each v is an array declarator or a variable, array, or FUNCTION name.

The following relationships should be noted:

1. BYTE, INTEGER* 1, LOGICAL* 1, and LOGICAL are all equivalent;
2. INTEGER* 2, LOGICAL* 2, and INTEGER are equivalent;
3. REAL and REAL*4 are equivalent
4. DOUBLE PRECISION and REAL*8 are equivalent.

Examples:

```
BYTE BUFF(256)

REAL IN,IOUT

DOUBLE PRECISION DPARG
```

EXTERNAL Statement

This statement allows for external procedure names to be used as arguments to other subprograms. The external procedure name can be a SUBROUTINE, BLOCK DATA or FUNCTION name. The statement is of the general form:

EXTERNAL ul,u2,...,un

where each ui is a SUBROUTINE, BLOCK DATA or FUNCTION name.

The EXTERNAL statement will allow any name in the list ui to be used as an argument when calling a subroutine.

When a BLOCK DATA subprogram is to be included as an argument to another subprogram, its name must have appeared in an EXTERNAL statement within the main program unit.

For example, if SUM and AFUNC are subprogram names to be used as arguments in the subroutine SUBR, the following statements would appear in the calling program unit:

```
EXTERNAL SUM, AFUNC
 .
 .
 .
CALL SUBR(SUM,AFUNC,X,Y)
```

DIMENSION Statement

The DIMENSION statement is a non-executable statement used to reserve storage for an array. It also defines the number of dimensions and elements in an array. The elements of the array are then referred to by using the array name followed by a subscript. The general form of the statement is:

DIMENSION ul,u2,u3

where each ui is an array declarator.

Example:

```
DIMENSION RAT(5,5),BAR(20)
```

This statement declares two arrays - the 25 element array RAT and the 20 element array BAR. (For information on arrays and array storage allocation, see the discussion of array declarators on Page 9-2.)

COMMON Statement

COMMON statements are non-executable, storage-allocating statements which assign variables and arrays to a storage area called COMMON storage. This provides the facility for various program units to share the same storage area. They are of the general form:

```
COMMON /cb/nlist/cb/nlist/.../cb/nlist
```

where each *cb* is a COMMON block storage name and each *nlist* is a sequence of variable names, array names or constant array declarators, separated by commas. The elements in *nlist* make up the COMMON block storage area specified by the name *cb*.

A COMMON block name is made up of from 1 to 6 alphanumeric characters, the first of which must be a letter. The name of a COMMON block may appear more than once in the same COMMON statement, or in more than one COMMON statement. The COMMON block name must be different from any subprogram names used throughout the program.

If any nlist is omitted, leaving two consecutive slash characters (/ /), the block of storage so indicated is called **blank COMMON.** If the first block name is omitted, the first two slashes may also be omitted.

The length of a COMMON area is the number of storage units required to contain the variables and arrays declared in the COMMON statement (or statements).,

The lengths of COMMON blocks of the same name need not be identical in all program units where the name appears. However, if the lengths differ, the program unit specifying the greatest length must be loaded first. (See Section C, "LINK-80", in this Manual.)

Example:

```
COMMON /AREA/A,B,C/BDATA/X,Y,Z,FL,ZAP(30)
```

In this example, two blocks of COMMON storage are allocated - AREA with space for three variables and BDATA, with space for four variables and the 30 element array, ZAP.

Example:

```
COMMON //A1,B1/CDATA/ZOT(3,3)//T2,Z3
```

In this example, Al, B1, T2 and Z3 are assigned to blank COMMONs. The pair of slashes preceding A1 could have been omitted. CDATA names COMMON block storage for the nine element array, ZOT and thus ZOT (3,3) is an array declarator. ZOT must not have been previously declared.

EQUIVALENCE Statement

The EQUIVALENCE statement permits the sharing of the same storage location by two or more entities.

Each element in the sequence is assigned to the same storage location by the compiler. Thus, the same storage location can be referenced with different variables.

The order in which the elements appear is not significant. The statement is of the general form:

EQUIVALENCE (nlist),(nlist),...,(nlist)

where each nlist represents a sequence of two or more variables or array elements, separated by commas.

Example:

```
EQUIVALENCE (A,B,C)
```

The variables A, B and C will share the same storage location during object program execution.

If an array element is used in an EQUIVALENCE statement, the number of subscripts must be the same as the number of dimensions established by the array declarator. It can also be one, where the one subscript specifies the array element's number relative to the first element of the array.

If the dimensionality of an array, Z, has been declared as Z(3,3) then in an EQUIVALENCE statement Z(6) and Z(3,2) have the same meaning. The subscripts of array elements must be integer constants.

Making Arrays **Equivalent**

If an element of one array is made equivalent to an element .of another array, the EQUIVALENCE statement will also set equivalences between the corresponding elements of the two arrays. If the first elements of two equally dimensioned arrays are made equivalent, both arrays will share the same storage area.

It is invalid to EQUIVALENCE two or more elements of the same array. It is also invalid to EQUIVALENCE two or more elements belonging to the same or different COMMON blocks.

Example:

```
DIMENSION A(7),B(3)
EQUIVALENCE (A (5)B(3)
```

This EQUIVALENCE statement will establish the following equivalences:

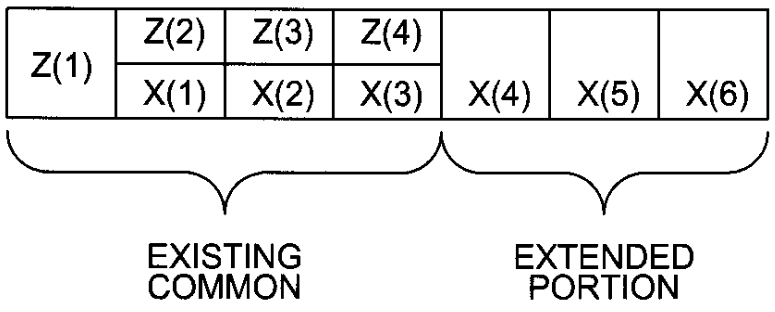
ARRAY A		ARRAY B
A(1)		
A(2)		
A(3)	=	B(1)
A(4)	=	B(2)
A(5)	=	B(3) A(6)
A(7)		

EQUIVALENCE and COMMON Interaction

Variables may be assigned to a COMMON block through an EQUIVALENCE statement. EQUIVALENCE statements can increase the size of a COMMON block by adding more elements to the end of the block. COMMON block size may be increased only from the last element established by the COMMON statement forward, not from its first element backward.

Example:

```
DIMENSION Z(4) X(6)
COMMON Z
EQUIVALENCE (Z(2),X(1))
```



DATA Initialization Statement

The DATA initialization statement is a non-executable statement which provides a means of initializing variables and array elements. The statement is of the following form:

DATA list/u1,u2,...,un/,list.../uk,uk+1,...uk+n/

where "list" represents a list of variable, array or array element names, and the ui are constants corresponding in number and type to the elements in the list.

There is an exception to the one-for-one correspondence of list items to constants. An array name (unsubscripted) may appear in the list, and as many constants as necessary to fill the array may appear in the corresponding position between slashes.

Instead of ui, it is permissible to write k*ui in order to declare the same constant, ui, k times in succession. k must be a positive integer.

Example:

```
DIMENSION C(7)
DATA A, B, C(1),C(3)/14.73,-8.1,2*7.5/
```

This implies that:

A=14.73, B=-8.1, C(1)=7.5, C(3)=7.5

The type of each constant ui must match the type of the corresponding item in the list, except that a Hollerith or literal constant may be paired with an item of any type.

When a Hollerith or literal constant is used, the number of characters in its string should be no greater than the number of bytes required by the corresponding item,i.e., 1 character for a logical variable, up to 2 characters for an integer variable up to 4 characters for a real variable and up to 8 characters for a double-precision variable.

If fewer Hollerith or literal characters are specified, trailing blanks are added to fill the remainder of storage. Hexadecimal data are stored in a similar fashion. If fewer hexadecimal characters are used, sufficient leading zeros are added to fill the remainder of the storage unit.

Chapter Ten

Functions and Subprograms

OVERVIEW

The FORTRAN language provides a means for defining and using often needed procedures such that the statement or statements of the procedures need appear in a program only once. These procedures may be referenced and brought into the logical execution sequence of the program whenever and as often as needed. These procedures are as follows:

1. Statement functions.
2. Library functions.
3. FUNCTION subprograms.
4. SUBROUTINE subprograms.

Each of these procedures has its own unique requirements. This Chapter will explain the various requirements for constructing and referencing these procedures.

In the following descriptions of these procedures, the term **"calling program"** means the program unit or procedure in which a reference to a procedure is made, and the term **"called program"** means the procedure to which a reference is made.

STATEMENT FUNCTIONS

Statement functions are defined by a single arithmetic or logical assignment statement and are relevant only to the program unit in which they appear.

The general form of a statement function is:

```
            f(al,a2,...an) = e
```

where *f is* the function name, the *ai* are dummy arguments and e is an arithmetic or logical expression.

Statement function definitions, if they exist in a program unit, must precede all executable statements in the unit and follow all specification statements.

The *ai* are distinct variable names or array elements, but, being dummy variables, they may have the same names as variables appearing elsewhere in the program unit.

The expression e is constructed according to the rules in Chapter 4, "FORTRAN Expressions". It may contain only references to the dummy arguments and non-literal constants, variable and array element references, utility and mathematical function references and references to previously defined statement functions.

The type of any statement function name or argument that differs from its predefined convention type must be defined by a type specification statement. (See Chapter 9, "Specification Statements" for a discussion of type specification statements.)

A statement function is called by its name followed by a parenthesized list of arguments. The expression is evaluated using the arguments specified in the call. The variable used for the reference is assigned the result.

The ith parameter in every argument list must agree in type with the ith dummy argument in the statement function.

The example below shows a statement function and a statement function call.

```
C STATEMENT FUNCTION DEFINITION
C
      FUNC1(A,B,C,D) = ((A+B)**C)/D



C STATEMENT FUNCTION CALL
C
      A12=A1-FUNC1(X,Y,Z7,C7)
```

LIBRARY FUNCTIONS

Library functions are a group of utility and mathematical functions which are "built-in" to the FORTRAN system.

The functions are listed in Tables 10-1 and 10-2. In the tables, arguments are denoted as a 1,a2,. . .,an, if more than one argument is required; or as a if only one is required.

A library function is called when its name is used in an arithmetic expression. Such a reference takes the following form:

```
f(a1,a2,...an)
```

where f is the name of the function and the ai are actual arguments. The arguments must agree in type, number and order with the specifications indicated in Tables 10-1 and 10-2.

In addition to the functions listed in 10-1 and 10-2, four additional library subprograms are provided to enable access to the 8080 (or Z80) hardware.

```
PEEK, POKE, INP, OUT
```

PEEK and INP are logical functions; POKE and OUT are subroutines. PEEK and POKE allow access to any memory location. PEEK(a) returns the contents of the memory location specified by a.

CALL POKE(a1,a2) causes the contents of the memory location specified by a1 to be replaced by the contents of a2. INP and OUT allow access to the I/O ports. INP(a) does an input from port a and returns the 8-bit value. CALL OUT(a1,a2) outputs the value of a2 to the port specified by al.

Examples:

```
A1=FLOAT(I7)
```

Convert the integer I7 to real and assign the result to the real variable Al.

```
POS=ABS(R1)
```

Assign the absolute value of the real variable R1 to the real variable POS.

```
S3=SIN(T12)
```

Calculate the sine of T12 and assign the result to the real variable S3.

Function Name	Definition	Types Argument	Types Function
ABS	Absolute Value	Real	Real
LABS		Integer	Integer
DABS		Double	Double
AINT	Sign of a times largest	Real	Real
INT	integer <= |a|	Real	Integer
IDINT		Double	Integer
AMOD	Returns remainder when first	Real	Real
MOD	argument is divided by second	Integer	Integer
AMAX0	Returns largest value from	Integer	Real
AMAX1	elements of argument list	Real	Real
MAX0		Integer	Integer
MAX1		Real	Integer
DMAX1		Double	Double
AMIN0	Returns smallest value from	Integer	Real
AMIN1	elements of argument list	Real	Real
MIN0		Integer	Integer
MIN1		Real	Integer
DMIN1		Double	Double
FLOAT	Conversion from Integer to Real	Integer	Real
IFIX	Conversion from Real to Integer	Real	Integer
SIGN	Returns the value:	Real	Real
ISIGN	sign of a2* |al|	Integer	Integer
DSIGN		Double	Double
DIM	a1 – Min(a1,a2)	Real	Real
IDIM		Integer	Integer
SNGL	Double to Real Conversion	Double	Real
DBLE	Real to Double Conversion	Real	Double

TABLE 10-1
Intrinsic Functions

Name	Number of Arguments	Definition	Type Argument	Type Function
EXP	1	e**a	Real	Real
DEXP	1		Double	Double
ALOG	1	In (a)	Real	Real
DLOG	1		Double	Double
ALOG10	1	log10(a)	Real	Real
DLOG10	1		Double	Double
SIN	1	sin (a)	Real	Real
DSIN	1		Double	Double
COS	1	cos (a)	Real	Real
DCOS	1		Double	Double
TANH	1	tanh (a)	Real	Real
SQRT	1	(a) ** 1/2	Real	Real
DSQRT	1		Double	Double
ATAN	1	arctan (a)	Real	Real
DATAN	1		Double	Double
ATAN2	2	arctan (a 1 /a2)	Real	Real
DATAN2	2		Double	Double
DMOD	2	a1 (mod a2)	Double	Double

TABLE 10-2

Basic External Functions

FUNCTION SUBPROGRAMS

A program unit which begins with a FUNCTION statement is called a FUNCTION subprogram.

A FUNCTION statement has one of the following forms:

```
            t FUNCTION f(a1,a2,...an)
            FUNCTION f(a1,a2,...an)
```

where:

t is either INTEGER, REAL, DOUBLE PRECISION or LOGICAL or is omitted as shown in the second form.

f is the name of the FUNCTION subprogram.

The *ai* are dummy arguments of which there must be at least one and which represent variable names, array names or dummy names of SUBROUTINE or other FUNCTION subprograms.

The data type of a FUNCTION name can be established by a type statement, by explicitly stating the data type in the FUNCTION statement, or by using the predefined data type.

In any case, the data type defined in the FUNCTION statement must agree with the data type used in the calling program.

The FUNCTION statement must be the first statement of the program unit. It must not contain a statement label.

The FUNCTION subprogram will return a single value to the calling program. The data type of this value will be determined by the data type of the FUNCTION name.

Constructing a FUNCTION Subprogram

Within the FUNCTION subprogram, the FUNCTION name must appear at least once on the left side of the equality sign in an assignment statement. The FUNCTION name can also be an element in the I/O list of an input statement. This defines the value of the FUNCTION so that it may be returned to the calling program.

Additional values may be returned to the calling program through assignment of values to dummy arguments.

The names in the dummy argument list may not appear in EQUIVALENCE, COMMON or DATA statements in the FUNCTION subprogram.

If a dummy argument is an array name, then an array declaration must appear in the subprogram with dimensioning information consistent with that in the calling program.

A FUNCTION subprogram may contain any defined FORTRAN statements other than BLOCK DATA statements, SUBROUTINE statements, or another FUNCTION statement. A FUNCTION subprogram must also not contain any statement which references either the FUNCTION being defined or another subprogram that references the FUNCTION being defined.

The logical termination of a FUNCTION subprogram is a RETURN statement and there must be at least one of them. A FUNCTION subprogram must physically terminate with an END statement.

Example:

```
    FUNCTION SUM(ARRAY,I)
    DIMENSION ARRAY(10)
    SUM = 0.0
    DO 100 K=1,I
100 SUM=SUM+ARRAY(K)
    RETURN
    END
```

The above program segment is used to construct a FUNCTION subprogram called SUM. The data type is not stated, so it follows the predefined convention. This subprogram will calculate the sum of the array named ARRAY.

Referencing a FUNCTION Subprogram

FUNCTION subprograms are called whenever the FUNCTION name, accompanied by an argument list, is used as an operand in an expression. Such references take the following form:

```
        f(a1,a2,...,an)
```

where f is a FUNCTION name and the ai are actual arguments. Parentheses must be present in the form shown.

The arguments ai must agree in type, order and number with the dummy arguments in the FUNCTION statement of the called FUNCTION subprogram. They may be any of the following:

1. A variable name.

2. An array element name.

3. An array name.

4. An expression.

5. A SUBROUTINE or FUNCTION subprogram name.

6. A Hollerith or literal constant.

If an argument is a subprogram name, that name must have previously been distinguished from ordinary variables by appearing in an EXTERNAL statement. In addition, the corresponding dummy arguments in the called FUNCTION subprograms must be used in subprogram references.

If an argument is a Hollerith or literal constant, the corresponding dummy variable should encompass enough storage units to correspond exactly to the amount of storage needed by the constant.

When a FUNCTION subprogram is called, program control goes to the first executable statement following the FUNCTION statement.

The following example shows a reference to a FUNCTION subprogram.

```
            DIMENSION ARRAY(10) RESULT = SUM(ARRAY,10)
```

This program will reference the subprogram constructed on the previous page. The single value returned to the calling program will be assigned to the variable RESULT.

SUBROUTINE SUBPROGRAMS

A program unit which begins with a SUBROUTINE statement is celled a SUBROUTINE subprogram. The SUBROUTINE statement has one of the following forms:

```
SUBROUTINE s (a1,a2,...,an)
SUBROUTINE s
```

where s is the name of the SUBROUTINE subprogram and each ai is a dummy argument which represents a variable or array name or another SUBROUTINE or FUNCTION name.

The SUBROUTINE statement must be the first statement of the subprogram. The SUBROUTINE subprogram name must not appear in any statement other then the initial SUBROUTINE statement. The dummy argument names must not appear in EQUIVALENCE, COMMON or DATA statements in the subprogram.

If a dummy argument is an array name, an array declarator must appear in the subprogram with dimensioning information consistent with that in the calling program.

If any of the dummy arguments represent values that are to be determined by the SUBROUTINE subprogram and returned' to the calling program, these dummy arguments must appear within the subprogram on the left side of the equality sign in an assignment statement, in the I/O list of an input statement or as a parameter within a subprogram reference.

A SUBROUTINE may contain any FORTRAN statements except the following:

- BLOCK DATA Statements
- FUNCTION Statements
- Another SUBROUTINE Statement
- A PROGRAM Statement

A SUBROUTINE subprogram may contain any number of RETURN statements. It must have at least one. The RETURN statement is the logical termination point of the subprogram. The physical termination of a SUBROUTINE subprogram is an END statement.

If an actual argument transmitted to a SUBROUTINE subprogram by the calling program is the name of a SUBROUTINE or FUNCTION subprogram, the corresponding dummy argument must be used in the called SUBROUTINE subprogram as a subprogram reference.

Subroutine Subprogram Example:

```
C SUBROUTINE TO COUNT POSITIVE ELEMENTS
C IN AN ARRAY.
      SUBROUTINE COUNTP(ARRAY,I,ICOUNT)
      DIMENSION ARRAY(10)
      ICOUNT=0
      DO 9 J=1,I
      IF(ARRAY(J))9,5,5
 9    CONTINUE
      RETURN
 5    ICOUNT=ICOUNT+1
      GO TO 9
      END
```

Referencing a SUBROUTINE Subprogram

A SUBROUTINE subprogram may be called by using a CALL statement. A CALL statement has one of the following forms:

```
CALL s(a1,a2,...,an)
CALL s
```

where s is a SUBROUTINE subprogram name and the ai are the actual arguments to be used by the subprogram.

The ai must agree in type, order and number with the corresponding dummy arguments in the subprogram-defining SUBROUTINE statement. The arguments in a CALL statement must comply with the following rules:

- FUNCTION and SUBROUTINE names appearing in the argument list must have previously appeared in an EXTERNAL statement.
- If the called SUBROUTINE subprogram contains a variable array declarator, then the CALL statement must contain the actual name of the array and the actual dimension specifications as arguments.

- If an item in the SUBROUTINE subprogram dummy argument list is an array, the corresponding item in the CALL statement argument list must be an array.

- When a SUBROUTINE subprogram is called, program control goes to the first executable statement following the SUBROUTINE statement.

Example:

```
      DIMENSION DATA(10)
       .
       .
       .
C THE STATEMENT BELOW CALLS THE
C SUBROUTINE CONSTRUCTED IN THE PREVIOUS SECTION
      CALL C0UNTP(DATA,10,CPOS)
```

RETURN FROM FUNCTION AND SUBROUTINE SUBPROGRAMS

The logical termination of a FUNCTION or SUBROUTINE subprogram is a RETURN statement which transfers control back to the calling program.

The general form of the RETURN statement is:

```
RETURN
```

These rules govern the use of the RETURN statement.

- There must be at least one RETURN statement in each SUBROUTINE or FUNCTION subprogram.
- RETURN from a FUNCTION subprogram is to the instruction sequence of the calling program following the FUNCTION reference.
- RETURN from a SUBROUTINE subprogram is to the next executable statement in the calling program which would logically follow the CALL statement.
- Upon return from a FUNCTION subprogram the single-valued result of the subprogram is available to the evaluation of the expression from which the FUNCTION call was made.
- Upon return from a SUBROUTINE subprogram the values assigned to the arguments in the SUBROUTINE are available for use by the calling program.

Example:

```
Calling Program Unit
 .
 .
 .
CALL SUBR(Z9,B7,R1)
 .
 .
 .
Called Program Unit
      SUBROUTINE SUBR(A,B,C)
      READ(3,7) B
      A = B**C
      RETURN
7     FORMAT(F9.2)
      END
```

(In this example, Z9 and B7 are made available to the calling program when the RETURN occurs.)

PROCESSING ARRAYS IN SUBPROGRAMS

If a calling program passes an array name to a subprogram, the subprogram must contain the dimension information pertinent to the array.

A subprogram must contain array declarators if any of its dummy arguments represent arrays or array elements.

For example:

Calling Program Unit

```
      DIMENSION Z1(50),Z2(25)
        .
        .
        .
      Al = AVG(Z1,50)
```

Called Program Unit

```
      FUNCTION AVG(ARG,I)
      DIMENSION ARG(50)
      SUM = 0.0
      DO 20 J=1,I
 20   SUM = SUM + ARG(J)
      AVG = SUM/FLOAT(I)
      RETURN
      END
```

Note that actual arrays to be processed by the FUNCTION subprogram are dimensioned in the calling program. The array names and their actual dimensions are transmitted to the FUNCTION subprogram by the FUNCTION subprogram reference. The FUNCTION subprogram itself contains a dummy array and specifies an array declarator.

Dimensioning information may also be passed to the subprogram in the parameter list. For example:

Calling Program Unit

```
DIMENSION A(3,4,5)
CALL SUBR(A,3,4,5)
END
```

Called Program Unit

```
SUBROUTINE SUBR(X,I,J,K)
DIMENSION X(I,J,K) RETURN
END
```

It is valid to use variable dimensions only when the array name and all of the variable dimensions are dummy arguments. The variable dimensions must be type integer. It is invalid to change the values of any of the variable dimensions within the called program.

BLOCK DATA SUBPROGRAMS

A BLOCK DATA subprogram has as its only purpose the initialization of data in a COMMON block during loading of a FORTRAN object program. BLOCK DATA subprograms begin with a BLOCK DATA statement and are of the .following form:

```
BLOCK DATA [subprogram-name ]
```

and end with an END statement.

The subprogram-name, which is optional, is a symbolic name associated with the BLOCK DATA subprogram.

Such subprograms may contain only type, EQUIVALENCE, DATA, COMMON and DIMENSION statements and are subject to the following considerations:

- If any element in a COMMON block is to be initialized, all elements of the block must be listed in the COMMON statement even though they might not all be initialized.
- Initialization of data in more than one COMMON block may be accomplished in one BLOCK DATA subprogram.
- There may be more than one BLOCK DATA subprogram loaded at any given time.
- Any particular COMMON block item should only be initialized by one program unit.

Example:

```
 BLOCK DATA TEST
 LOGICAL Al
 COMMON/BETA/B(3,3)/GAM/C (4)
 COMMON/ALPHA/A1,C,E,D
 DATA B/1.1,2.5,3.8,3*4.96,
+2*0.52,1.1/,C/1.2E0,3*4.0/
 DATA A1/.TRUE/,E/-5.6/
 END
```

Chapter Eleven

FORTRAN Statements Summary

OVERVIEW

This Chapter is a summary of the statements implemented in this version of FORTRAN. A brief description of each statement is also included. The structure of each statement adheres to the following notation conventions.

1. A notation variable is represented by an italicized sequence of lower case letters.
2. A notation constant or keyword is represented by a sequence of capital letters.
3. A set of brackets ([ ]) indicates an optional item.
4. The series of three periods, or ellipses, represents an item that can be repeated zero or more times.

SUMMARY OF STATEMENTS

ASSIGN *j* to *i*

j	is a statement label.
i	is an integer variable.

This statement is used with each assigned GO TO statement. When the assigned GO TO is executed, control will be transferred to the statement labeled j. If a list is specified within the assigned GO TO, j must be included in this list.

BLOCK DATA [*subprogram-name*]

["*subprogram name*"]	is any valid symbolic name.

This statement is used to specify the name of a BLOCK DATA subprogram. A BLOCK DATA subprogram is used to initialize variables in a COMMON block. BLOCK DATA subprograms begin with a BLOCK DATA statement and end with an END statement. A BLOCK DATA subprogram may contain only Type, EQUIVALENCE, DATA , COMMON and DIMENSION statements.

CALL*s*[([*a*] [,*a*]. . .)]

s	is the subroutine name.
a	are actual arguments to be used by the subprogram.

The CALL statement is used to transfer program control to a subroutine. When a SUBROUTINE program is called, program control goes to the first executable statement following the SUBROUTINE statement. The arguments passed to the subroutine must agree in type, order and number with the corresponding dummy arguments in the SUBROUTINE statement.

COMMON [/ [*cb*] /] nlist [[,] / [*cb*] /*list*] . . .

cb is the COMMON block name.
list is the list of variables.

COMMON statements are storage allocating statements which assign variables and arrays to a storage area called COMMON storage. This allows for various program units to use the same storage area. The list of variables must be a sequence of variable names, array names or constant array declarators. These names must be separated by commas. The COMMON block name may be omitted. This is referred to as a blank COMMON.

CONTINUE

CONTINUE is used as the terminal statement in a DO loop when the statement which would normally be the terminal statement is one of those which are not allowed.

DATA *list* /*clist*/[[,]*list*/*clist*/] . .

list is the list of variables separated by commas.
clist is the constant values to assign the variables.

The DATA statement is used to compile constant data values into the object program and assign these values to variables and array elements.

DECODE (*a*,*f*) *k*

a is an array name.
f is a FORMAT statement number.
k is an I/O list.

This statement causes the elements in the array a to be translated from character format to the internal format specified in the FORMAT statement *f*. The results of this conversion. are placed in the list of I/O elements k. This is analogous to a READ statement, except the data transfer is from one section of memory to another.

DIMENSION *s*(*d*) [,*s*(*d*)]. . .

s is the name of the array.
d is the array dimension declarator.

This statement reserves storage locations for each of the elements of an array. The elements of the array are then referred to by using the array name followed by a subscript.

DO*k i = m1,m2 [,m3]*

k is the statement label of the terminal statement
i is the index-variable.
m1 is the initial value.
m2 is the terminal value.
m3 is the incremental value. (If omitted defaults to *1)*

The DO statement provides a method for repetitively executing a series *of* statements. The statement labeled k must be an executable statement. The index variable i must be positive and cannot be modified by any statement in the range of the DO loop. The following steps take place when executing a DO loop:

1. Set i =*ml*
2. Execute statements through *k*
3. *i = m 1 +m3*
4. Has the terminal value been reached? (i =m2). YES- transfer to statement after *k*. NO - repeat steps *2-4*

ENCODE (*a*,*f*) *k*

a is an array name.
f is a FORMAT statement number.
k is an I/O list.

This statement causes the elements in the I/O list *k* to be translated to character format. The data is translated according to the specifications of the FORMAT statement *f* . The translated data is put into array a. This is analogous to a WRITE statement, except the data transfer is from one section of memory to another.

END

The END statement must physically be the last statement of any FORTRAN program. It causes a transfer of control to be made to the system exit routine, which exits to HDOS.

ENDFILE *u*

u is an integer variable or constant.

This closes the file associated with Logical Unit Number u. This statement is only used for disk files.

EQUIVALENCE (*list*) [, (*list*)] . . .

list is a sequence of two or more variables or array elements, separated by commas.

Use of EQUIVALENCE statements permits the sharing of the same storage area by two or more entities. Each element in the list is assigned to the same storage area. The order in which the elements appear is not significant.

EXTERNAL *v* [,*v*] . . .

v **is a subprogram name.**

This statement allows for external procedure names to be used as arguments to other subprograms. External procedure names can be a SUBROUTINE, BLOCK DATA or FUNCTION-name. The EXTERNAL statement will allow any name v to be used as an argument when CALLing a subroutine.

FORMAT (s [,s] . .) s is the field descriptor.

FORMAT statements are used in conjunction with formatted READ and WRITE statements. They specify conversion methods and data editing to be performed as the data is transmitted between computer memory and external storage devices. FORMAT statements require statement labels for reference in the READ and WRITE statements.

***t* FUNCTION *f* [([*a* [,*a*] . . .])]**

t	is the data type (optional).
f	*is* the subprogram name.
a	*is* the dummy argument names.

This denotes the beginning of a FUNCTION subprogram. The name *f is* used to reference this FUNCTION. t is either INTEGER, REAL, DOUBLE PRECISION or LOGICAL. The FUNCTION statement must be the first statement of the program unit. The program unit is terminated with a RETURN statement.

GO TO *k* (Unconditional GO TO)

k	is an executable statement label.

Control of the program is transferred to statement k.

GO TO (*k1*,*k2*,...,*kn*),*j* (Computed GO TO)

ki	are executable statement labels.
j	is an integer variable.

This statement causes transfer of control to the statement label kj. (If *j* = 1 then control is transferred to label k1. If *j* = 2 then control is transferred to label k2, etc.) If j<1 or j>n then control will be passed to the next statement following the computed GO TO.

GO TO *j*, [(*k1*,*k2*,. . .,*kn*)] (Assigned GO TO)

j	is an integer variable.
ki	are executable statement labels.

This statement causes transfer of control to the statement whose label is equal to the current value of j. j is assigned a value via the ASSIGN statement. If the statement labels ki are present, then j must have been ASSIGNed a value included in this list. The ASSIGN statement must logically precede the GO TO statement.

IF (*e*) *m1*,*m2*,*m3* (ARITHMETIC IF)

e	is an arithmetic expression.
mi	are statement labels.

Transfers control based on the results of the evaluation of (e). If the result of the evaluation of (e) is negative (<0) then control is transferred to the statement labeled ml. If the result of the evaluation of (e) is zero (=0), then control is transferred to the statement labeled m2. If the result of the evaluation of (e) is positive (>0) then control is transferred to the statement labeled m3.

IF (*u*) *s* (Logical IF)

u	is a logical expression.
s	is any executable expression except a DO statement.

The logical expression u is evaluated as TRUE. or .FALSE. If u is evaluated as FALSE, then the statement s is ignored and control goes to the next statement following the logical IF statement. If u is evaluated as TRUE, then control goes to statement s . Subsequent program control follows normal conditions. If s is a replacement statement (v = e), the variable v and the equality sign (=) must be on the same line.

PAUSE [*c*]

c is any string up to 6 characters.

When PAUSE is encountered during execution of the object program, the characters c (if present) are displayed on the terminal device and execution of the program ceases. Execution may be terminated by typing a "T" and a carriage return at the terminal device. Typing any other character and a carriage return will cause execution to continue.

PROGRAM *name*

name specifies the name of the main program.

The PROGRAM statement provides a means of specifying a name for a main program unit. The name consists of 1-6 alphanumeric characters, the first of which is a letter. If no PROGRAM statement is present in a main program, the Compiler assigns a name of $MAIN to the program.

READ (*u*,*f* [,ERR=*I1*] [,END=*I2*]) *k* (Formatted Sequential)

u is the logical unit number.
f is the label of the FORMAT statement.
I1 is the label to transfer to if an error is encountered.
I2 is the label to transfer to if EOF is reached.
k is the list of variable names.

This is used to input a number of items, corresponding to the names in the list k. The input is from the file assigned to logical unit number u. The input is converted according to the FORMAT statement f .

READ(*u* [,ERR=*I1*] [,END =*I2*]) *k* (Unformatted Sequential)

u is the logical unit number.
I1 is the label to transfer to if an error is encountered.
I2 is the label to transfer to if EOF is reached.
k is the list of variable names.

This statement is the same as the formatted READ, except this performs a memory image transmission of data with no data conversion or editing. The amount of data transmitted corresponds to the number and type of variables in the list k.

READ (*u*,*f*,REC=*i* [,ERR=*I1*]) *k* (Formatted Random Access)

u	is the logical unit number.
f	is the label of the FORMAT statement.
i	is the record number to read.
I1	is the label to transfer to if an error is encountered.
k	is an I/O list.

This statement is essentially the same as the formatted READ except any record in the file can be read by including the REC=i clause.

READ (*u*,REC=*i* [,ERR=*I1*]) *k* (Unformatted Random Access)

u	is the logical unit number.
i	is the record number to read.
I1	is the label to transfer to if an error is encountered.
k	is an I/O list.

This statement will cause the i th record of the file to be read. The input data will be assigned to the variables in the list k. The data will be transmitted without any editing or formatting.

READ (*u*,*f* [,ERR=*I1*] [,END=*I2*]) (H type conversions)

u	is the logical unit number.
f	is the label of the FORMAT statement.
I1	is the label to transfer to if an error is encountered.
I2	is the label to transfer to if EOF is reached.

(No variable list is needed.)

This statement may be used in conjunction with a FORMAT statement to read H-type alphanumeric data into an existing H-type field.

RETURN

Returns control to the calling program.

The logical termination of a FUNCTION or subprogram is a RETURN statement. This statement will return control to the calling program.

REWIND *u*

u	is an integer variable or constant.

This statement will close and then open the disk file associated with logical unit u. It has no effect on non-disk files.

STOP [*c*]

c	is any string up to 6 characters.

When STOP is encountered during execution of the object program, the characters c (if present) are displayed on the terminal device and the execution of the program terminates. The STOP statement is considered the logical end of the program.

SUBROUTINE *s* [([*a*] [,*a*] . . .])]

s	is the subroutine name
a	is the dummy arguments

A program unit which begins with a SUBROUTINE statement is called a SUBROUTINE subprogram. The subroutine name s must not appear in any other statement within the subroutine. The subroutine is referenced from the main program by the CALL statement.

***type v* [,*v*] .**

type	is a data type specifier
v	is a variable name, array name, or function name

Type statements provide for associating a variable name with a data type. This may be used to confirm or override the predefined convention. In addition, these statements may be used to declare arrays. The data type specifier may be any of the following:

INTEGER	REAL	LOGICAL	DOUBLE PRECISION
INTEGER* 1	REAL*4	LOGICAL* 1	BYTE
INTEGER* 2	REAL* 8	LOGICAL* 2	

WRITE *(u,f* [,ERR=*I1*] [,END=*I2*]) *k* (Formatted Sequential)

u is the logical unit number.
f *is* the label of the FORMAT statement.
I1 is the label to transfer to if an error is encountered.
I2 is the label to transfer to if EOF is reached.
k is the variable list.

This is used to output the data specified in the list k to the output device assigned to logical unit number u. The output is converted and edited according to the FORMAT statement *f*.

WRITE (*u*,ERR=*I1*,END=*I2*) *k* (Unformatted Sequential)

u is the logical unit number.
I1 is the label to transfer to if an error is encountered.
I2 is the label to transfer to if EOF is reached.
k is the variable list.

This statement is similar to the formatted WRITE, except this performs a memory image transmission of data to the output device with no data conversion or editing. The amount of data transmitted corresponds to the number and type of variables in the list k.

WRITE (*u*,*f*,REC=*i* [,ERR=*I1*]) *k* (Formatted Random Access)

u is the logical unit number.
f *is* the label of the FORMAT statement.
i is the record number to write.
I1 is the label to transfer to if an error is encountered.
k is the variable list.

This statement is essentially the same as the formatted WRITE except any record in the file can be written by including the REC=i clause.

WRITE (*u*,REC=*i* [,ERR=*I1*]) *k* (Unformatted Random Access)

u	is the logical unit number.
i	is the record number to write.
I1	is the label to transfer to if an error is encountered.
k	is the I/O list.

This statement is the same as the formatted random access, except no format statement is referenced, thus no formatting or editing takes place during the transmission of data.

WRITE *(u,f [* ,ERR=*I1*] [,END=*I2*]) (No variable list)

u	is the logical unit number.
f	*is* the label of the FORMAT statement.
I1	is the label to transfer to if an error is encountered.
I2	is the label to tranfer to if EOF is reached.

This is used to write alphanumeric information when the characters to be printed are specified within the FORMAT statement. In this case a variable list is not required.

Microsoft MACRO-80 ASSEMBLER

SOFTWARE REFERENCE MANUAL

for Heath 8-bit digital computer systems

Part B
595-2286

HEATH COMPANY

BENTON HARBOR, MICHIGAN 49022

First Printing
Printed in the United
States of America

Table of Contents

IV

Chapter One

Using the MACRO-80 Assembler

OVERVIEW

The MACRO-80 Assembler is an 8080/Z80 Assembler with complete facilities For macro development.

In order to use the Assembler, a source program must first be written using an editor, such as EDIT. A MACRO-80 source program is composed of a series of statements. The format of each statement must follow a predefined format.

After the source program has been written, it must be assembled using the Macro Assembler. The result of this process will be a relocatable module. This module must then be linked using the Linking Loader. (See Section 3, "LINK-80", for information on the Linking Loader.) After the relocatable module has been linked, it can be executed.

In order to provide the Assembler with the information it needs to successfully assemble a source program, a command string must be input. This command string tells the Assembler where to find the source program, where to put the relocatable module and where to write the listing.

There are also several switches which can be set in the command string. Some of these switches are used to control the format of the listing file. A switch can also be set to allow the Assembler to assemble Z80 mnemonics.

FORMAT OF MACRO-80 SOURCE FILES

In general, MACRO-80 accepts a source file that is almost identical to source files for INTEL-compatible assemblers. Input source lines up to 132 characters in length are allowed.

MACRO-80 preserves lower-case letters in quoted strings and comments. All symbols, opcodes and pseudo-opcodes typed in as lower-case will be converted to upper-case.

Statements

Source files input to MACRO-80 consist of statements of the form:

[**label:] [:] [operator] [arguments] [;comment]**

It is not necessary that statements begin in column one. Multiple blanks or tabs may be used to improve readability.

If a label is present, it is the first item in the statement and is immediately followed by a colon (:). If it is followed by two colons, it is declared as PUBLIC. Therefore:

```
F00:: RET
```

is equivalent to:

```
PUBLIC F00 F00: RET
```

The next item after the label (or the first item on the line if no label is present) is an operator. An operator may be an opcode (8080 or Z80 mnenomic), pseudo-op, macro call, or expression.

The evaluation order is as follows:

1. Macro call
2. Opcode/Pseudo-operation
3. Expression

Instead of flagging an expression as an error, the Assembler treats it as if it were a DB statement. The arguments following the operator will, of course, vary in form according to the operator.

A comment always begins with a semicolon and ends with a carriage return. A comment may be a line by itself or it may be appended to a line that contains a statement. Extended comments can be entered using the.COMMENT operation.

Symbols

MACRO-80 symbols may be of any length. However, only the first six characters are significant. The following characters are legal in a symbol:

A-Z 0-9 $?

The underline character is also legal in a symbol. A symbol may not start with a numeric digit. Lower case symbols are translated to upper case. If a symbol reference is followed by it is declared external.

Numeric Constants

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

Numbers are 16-bit unsigned quantities. A number is always evaluated in the current radix unless one of the following special notations is used:

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

Overflow of a number beyond two bytes is ignored and the result is the low order 16-bits.

Strings

A string is comprised of zero or more characters delimited by quotation marks. Either single or double quotes may be used as string delimiters. The delimiter quotes may be used as characters if they appear twice for every character occurance desired. If there are zero characters between the delimiters, the string is a null string.

FORMAT OF COMMANDS

To run MACRO-80, type M80 followed by a carriage return. MACRO-80 will return the prompt "*", indicating it is ready to accept commands. The format of a MACRO-80 command string is:

```
objprog-dev:filename.ext,list-dev:filename.ext=source-dev:filename.ext
```

Where:

objprog-dev: The device on which the object program is to be written.

list-dev: The device on which the program listing is written.

source-dev: The device from which the source-program input to MACRO-80 is obtained. If a device name is omitted, it defaults to the currently selected drive.

filename.ext The file name and file name extension of the object program file, the listing file, and the source file. If the file name extensions are omitted, the operating system will insert the default extensions.

The default file name extensions are:

source file	MAC
relocatable object file	REL
listing file	LST
cross reference file	CRF

Either the object file or the listing file or both may be omitted. If neither a listing file nor an object file is desired, place only a comma to the left of the equal sign. If the names of the object file and the listing file are omitted, the default is the name of the source file.

Examples:

(NOTE: The asterisk represents the prompt from the Assembler.)

*EXP.REL,EXP.LST=EXP.MAC	Assemble the program EXP.MAC and place the object file in EXP.REL and the list file in EXP.LST.
*=EXP	Assemble the program EXP.MAC and place the object file in EXP.REL.
*,LP:=EXP	Assemble the program EXP.MAC,, place the object file in EXP.REL and list on the device LP:
*SMALL,TT:=TEST	Assemble the program TEST.MAC, place the object file in SMALL.REL and list in TT:

MACRO-80 Switches

A number of different switches may be given in the MACRO-80 command string that will affect the format of the listing file. Each switch must be preceded by a slash (/):

<u>**Switch**</u>	<u>**Action**</u>
O	Print all listing addresses, etc. in octal.
H	Print all listing addresses, etc. in hexadecimal. (Default)
R	Force generation of an object file.
L	Force generation of a listing file.
C	Force generation of a cross reference file. (See Page 1-10, "CrossReference Facility".)
Z	Assemble Z80 (Zilog format) mnemonics.
I	Assemble 8080 mnemonics. (Default)

Examples:

(NOTE: The asterisk represents the prompt from the Assembler.)

`*=TEST/L`	Compile TEST.MAC with object file TEST.REL and listing file TEST.LST
`*LAST, LAST/C=M0D1`	Compile MODI.MAC with object file LAST.REL and cross reference file LAST.CRF for use with CREF-80

Symbol Table Listing

In the symbol table listing, all the macro names in the program are listed alphabetically, followed by all the symbols in the program, listed alphabetically. After each symbol, a tab is printed, followed by the value of the symbol. If the symbol is Public, an I is printed immediately after the value. The next character printed will be one of the following:

Character	**Definition**
U	Undefined symbol.
C	COMMON block name. (The "value" of the COMMON block is its length (number of bytes) in hexadecimal or octal.)
*	External symbol.
<space>	Absolute value.
'	Program Relative value.
“	Data Relative value.
!	COMMON Relative value.

MACRO-80 ERROR MESSAGES

MACRO-80 errors are indicated by a one-character flag in column one of the listing file. If a listing file is not being printed on the terminal, each erroneous line is also printed or displayed on the terminal.

Error Codes

A Argument error -
Argument to pseudo-op is not in correct format or is out of range (.PAGE 1;.RADIX 1; PUBLIC 1; STAX H; MOV M,N; INX C).

C Conditional nesting error -
ELSE without IF, ENDIF without IF, two ELSEs on one IF.

D Double Defined symbol -
Reference to a symbol which is multiply defined.

E External error -
Use of an external illegal in context (e.g., FOO SET NAME; MVI A,2-NAME).

M Multiply Defined symbol -
Definition of a symbol which is multiply defined.

N Number error -
Error in a number, usually a bad digit (e.g., 8Q).

O Bad opcode or objectionable syntax -
ENDM, LOCAL outside a block; SET, EQU or MACRO without a name; bad syntax in an opcode (MOV A:); or bad syntax in an expression (mismatched parenthesis, quotes, consecutive operators, etc.).

P Phase error -
Value of a label or EQU name is different on pass 2.

Q Questionable -
Usually means a line is not terminated properly. This is a warning error (e.g. MOV A,B,).

R Relocation -
Illegal use of relocation in expression, such as abs-rel. Data, code and COMMON areas are relocatable.

U Undefined symbol -
A symbol referenced in an expression is not defined. (For certain pseudo-ops, a V error is printed on pass 1 and a U on pass 2.)

V Value error -
On pass 1 a pseudo-op which must have its value known on pass 1 (e.g., RADIX, .PAGE, DS, IF, IFE, etc.), has a value which is undefined later in the program, a U error will not appear on the pass 2 listing.

Error Messages:

`No end statement encountered on input file'
No END statement: either it is missing or it is not parsed due to being in a false conditional, unterminated IRP/IRPC/REPT block or terminated macro.

'Unterminated conditional'
At least one conditional is unterminated at the end of the file.

'Unterminated REPT/IRP/IRPC/MACRO'
At least one block is unterminated.

[xx] [No] Fatal error(s) [,xx warnings]
The number of fatal errors and warnings. The message is listed on the console and in the list file.

CROSS REFERENCE FACILITY

The Cross Reference Facility is invoked by typing CREF80. To generate a cross reference listing, the Assembler must output a special listing file with embedded control characters. The MACRO-80 command string tells the Assembler to output this special listing file. /C is the cross reference switch. When the /C switch is encountered in a MACRO-80 command string, the Assembler opens a .CRF file instead of a LST file.

Example:

(NOTE: The asterisk represents the prompt from the Assembler.)

`*=TEST/C`	Assemble file TEST/MAC and create object file TES-T/REL and cross reference file TEST.CRF.
`*T,U=TEST/C`	Assemble file TEST/MAC and create object file T/REL and cross reference file U.CRF.

When the Assembler is finished, exit to HDOS with CTRL-D. Then call the Cross Reference Facility by typing CREF80.

The command string is:

```
*listing file=source file
```

The default extension for the source file is CRF. the /L switch is ignored, and any other switch will cause an error message to be sent to the terminal.

Possible command strings are:

`*=TEST`	Examine file TEST.CRF and generate a cross reference listing file TEST.LST.
`*T=TEST`	Examine file TEST.CRF and generate a cross reference listing file T.LST.

Cross Reference listing files differ from ordinary listing files in that:

1. Each source statement is numbered with a Cross Reference number.

2. At the end of the listing, variable names appear in alphabetic order along with the numbers of the lines on which they are referenced or defined. Line numbers on which the symbol is defined are flagged with '#'.

Chapter Two

Expression Evaluation

OVERVIEW

In most cases, the operand field of a given opcode may be coded as an operand expression. Such expression is a string of integers, symbols and characters. This character string is combined using certain operators.

The symbols used in the expression can be expressed in several modes. A symbol can also be classified as either external or not external.

Additionally, 8080 opcodes can be used as valid one-byte operands.

ARITHMETIC AND LOGICAL OPERATORS

The following operators are allowed in expressions and are listed in descending order of precedence.

NUL

LOW, HIGH

***, /, MOD, SHR, SHL**

Unary Minus +,

EQ, NE, LT, LE, GT, GE

NOT

AND

OR, XOR

Parentheses are used to change the order of precedence. During evaluation of an expression, as soon as a new operator is encountered that has precedence less than or equal to the last operator encountered, all operations up to the new operator are performed. That is, subexpressions involving operators of higher precedence are computed first.

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

The byte isolation operators (HIGH, LOW) isolate the high- or low-order eight bits of an Absolute 16-bit value. If a relocatable value is supplied as an operand, HIGH and LOW will treat it as if it were relative to location zero.

MODES

All symbols used as operands in expressions are in one of the following modes:

Absolute
Data Relative
Program (Code) Relative
COMMON

Symbols assembled under the ASEG, CSEG (default), or DSEG pseudo-ops are in Absolute, Code Relative or Data Relative mode respectively.

The number of COMMON modes in a program is determined by the number of COMMON blocks that have been named with the COMMON pseudo-op. Two COMMON symbols are not in the same mode unless they are in the same COMMON block.

In any operation other than addition or subtraction, the mode of both operands must be Absolute.

If the operation is addition, the following rules apply:

1. At least one of the operands must be Absolute.

2. Absolute + <mode> = <mode>

If the operation is subtraction, the following rules apply:

1. <mode> - Absolute = <mode>

2. <mode> - <mode> = Absolute

where the two <mode>s are the same.

Each intermediate step in the evaluation of an expression must conform to the above rules for modes, or an error will be generated. For example, if FOO, BAZ and ZAZ are three Program Relative symbols, the expression:

```
F00 = BAZ - ZAZ
```

will generate an R error because the first step (FOO + BAZ) adds two relocatable values. (One of the values must be Absolute.)

This problem can always be fixed by inserting parentheses.

```
FOO = (BAZ - ZAZ)
```

is legal because the first step (BAZ - ZAZ) generates an Absolute value that is then added to the Program Relative value, FOO.

Externals

Aside from its classification by mode, a symbol is either External or not External. An External value must be assembled into a two-byte field. (Single-byte Externals are not supported.)

The following rules apply to the use of Externals in expressions:

1. Externals are legal only in addition and subtraction.
2. If an External symbol is used in an expression, the result of the expression is always External.
3. When the operation is addition, either operand (but not both) may be External.
4. When the operation is subtraction, only the first operand may be External.

Opcodes as Operands

8080 opcodes are valid one-byte operands. Note that only the first byte is a valid operand.
For example:

```
MVI     A,(JMP)
MVI     B,(RNZ)
MVI     C,MOV A,B
```

Errors will be generated if more than one byte is included in the operand - such as (CPI 5), (LXI B,LABEL1) or (JMP LABEL2).

Opcodes used as one-byte operands need not be enclosed in parentheses.

Chapter Three

Pseudo-Op codes /Assembler Directives

Overview

Within the Macro-80 Assembler there exists a set of instructions known as psuedo-opcodes or assembler directives. These instructions represent commands to the Assembler. They are called pseudo because although they are coded into the source program, they are not translated as instructions.

The following chapter explains the form and usage of the available psuedoopcodes.

PSEUDO-OPCODES

ASEG

```
        ASEG
```

ASEG sets the location counter to an absolute segment of memory. The location of the absolute counter will be that. of the last ASEG (default is 0), unless an ORG is done after the ASEG to change the location. The effect of ASEG is also achieved by using the code segment (CSEG) pseudo operation and the /P switch in LINK-80.

COMMON

```
       COMMON          /<block name>/
```

COMMON sets the location counter to the selected common block inn memory. The location is always the initial address of the common area so that compatibility with the FORTRAN COMMON statement is maintained. If <block name> is omitted or consists of spaces, it is considered to be blank common.

CSEG

```
        CSEG
```

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

Define Byte

```
            DB      <exp> [ , <exp> . . . )

            DB      <string> [ <string> . . . )
```

The arguments to DB are either expressions or strings. DB stores the values of the expressions or the characters of the strings in successive memory locations beginning with the current location counter.

Expressions must evaluate to one byte. (If the high byte of the result is 0 or 2 55, no error is given; otherwise, an A error results.)

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

Example:

```
      0000'   4142       DB      'AB'
      0002'   42         DB      'AB' AND. OFFH
      0003'   41 42 43   DB      'ABC'
```

Define Character

```
            DC      <string>
```

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

An error will result if the argument to DC is a null string.

Define Space

```
            DS      <exp>
```

DS reserves an area of memory. The value of <exp> gives the number of bytes to be allocated. All names used in <exp> must be previously defined (i.e., all names known at that point on pass 1).

Otherwise, a V error is generated during pass 1 and a U error may be generated during pass 2. If a U error is not generated during pass 2, a phase error will probably be generated because the DS generated no code on pass 1.

DSEG

```
DSEG
```

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

Define Word

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

DW stores the values of the expressions in successive memory locations beginning with the current location counter. Expressions are evaluated as 2-byte (word) values.

END .

```
END      [ <exp> ]
```

The END statement specifies the end of the program. If <exp> is present, it is the start address of the program. If <exp> is not present, then no start address is passed to LINK-80 for that program.

ENTRY/PUBLIC

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

ENTRY or PUBLIC declares each name in the list as internal and therefore available for use by this program and other programs to be loaded concurrently. All of the names in the list must be defined in the current program or a U error results. An M error is generated if the name is an external name or commonblockname.

EQU

```
<name> EQU <exp>
```

EQU assigns the value of <exp> to <name>. If <exp> is external, an error is generated. If <name>> already has a value other than <exp>, an M error is generated.

EXT/EXTRN

```
EXT     <name> [ , <name> . . . ]

EXTRN <name> [ , <name> . . . ]
```

EXT or EXTRN declares that the name(s) in the list are external (i.e., defined in a different program). If any item in the list references a name that is defined in the current program, an M error results. A reference to a name where the name is followed immediately by two pound signs (e.g., NAME##) also declares the name as external.

NAME

```
NAME ('modname')
```

NAME defines a name for the module. Only the first six characters are significant in a module name. A module name may also be defined with the TITLE pseudoop. In the absence of both the NAME and TITLE pseudo-ops, the module name is created from the source file name.

Define Origin

```
ORG <exp>
```

The location counter is set to the value of <exp> and the Assembler assigns generated code starting with that value. All names used in <exp> must be known on pass 1, and the value must either be absolute or in the same area as the location counter.

PAGE

```
PAGE     [ <exp> ]
```

PAGE causes the Assembler to start a new output page. The value of <exp>, if included, becomes the new page size (measured in lines per page) and must be in the range 10 to 255. The default page size is 50 lines per page. The Assembler puts a form feed character in the listing file at the end of a page.

SET

```
<name> SET <exp>
```

SET is the same as EQU, except no error is generated if <name> is already defined.

SUBTTL

```
SUBTTL <text>
```

SUBTTL specifies a subtitle to be listed on the line after the title on each page heading. <text> is truncated after 60 characters. Any number of SUBTTLs may be given in a program.

TITLE

```
TITLE <text>
```

TITLE specifies a title to be listed on the first line of each page. If more than one TITLE is given, a Q error results. The first six characters of the title are used as the module name unless a NAME pseudo operation is used. If neither a NAME or TITLE pseudo-op is used, the module name is created from the source file name.

.COMMENT

```
.COMMENT <delim><text><delim>
```

The first non-blank character encountered after COMMENT is the delimiter. The following <text> comprises a comment block which continues until the next occurrence of <delimiter> is encountered. For example, using an asterisk as the delimiter, the format of the comment block would be:

```
.COMMENT       *
any amount of text entered
here as the comment block
 .
 .
 .       *
;return to normal mode
```

PRINTX

.PRINTX <delim><text><delim>

The first non-blank character encountered after PRINTX is the delimiter. The following text is listed on the terminal during assembly until another occurrence of the delimiter is encountered.

.PRINTX is useful for displaying progress through a long assembly or for displaying the value of conditional assembly switches.

For example:

```
IF HDOS
.PRINTX /HDOS version
ENDIF
```

.PRINTX will output on both passes. If only one printout is desired, use the IF1 or IF2 pseudo-op.

.RADIX

.RADIX <exp>

The default base (or radix) for all constants is decimal. The RADIX statement allows the default radix to be changed to any base in the range 2 to 16.

For example:

```
LXI H,0FFH
.RADIX 16
LXI H,0FF
```

The two LXIs in the example are identical. The <exp> in a RADIX statement is always in decimal radix, regardless of the current radix.

.REQUEST

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

.REQUEST sends a request to the LINK-80 Loader to search the file names in the list for undefined globals before searching the FORTRAN library. The file names in the list should be in the form of legal MACRO-80 symbols. They should-not include file name extensions or disk specifications. The LINK-80 loader will supply the default extension REL and will assume drive SY0:.

.Z80

.Z80 enables the Assembler to accept Z80 opcodes. Z80 mode may also be set by appending the /Z switch to the MACRO-80 command string.

.8080

.8080 enables the Assembler to accept 8080 opcodes. This is the default condition. 8080 mode may also be set by appending the /I switch to the MACRO-80 command string.

CONDITIONAL PSEUDO-OPERATIONS

The conditional pseudo-operations are:

IF/IFT <exp>	True if <exp> is not 0.
IFE/IFF <exp>	True if <exp> is 0.
IF1	True if pass 1.
IF2	True if pass 2.
IFDEF <symbol>	True if <symbol> is defined or has been declared External.
IFNDEF <symbol>	True if <symbol> is undefined or not declared External.
IFB <arg>	True if <arg> is blank. The angle brackets around <arg> are required.
IFNB <arg>	True if <arg> is not blank. Used for testing when dummy parameters are supplied. The angle brackets around <arg> are required.

All conditionals use the following format:

```
IFxx [argument]
 .
 .
 .
[ELSE
 .
 .
 .  ]
ENDIF
```

Conditionals may be nested to any level.

Any argument to a conditional must be known on pass 1 to avoid V errors and incorrect evaluation. For IF, IFT, IFF, and IFE the expression must involve values which were previously defined and the expression must be absolute. If the name is defined after an IFDEF or IFNDEF, pass 1 considers the name to be undefined, but it will be defined on pass 2.

ELSE

Each conditional pseudo-operation may optionally be used with the ELSE pseudo-opcode which allows alternate code to be generated when the opposite condition exists. Only one ELSE is permitted for a given IF, and an ELSE is always bound to the most recently opened IF. A conditional with more than one ELSE or an ELSE without a conditional will cause a C error.

ENDIF

Each IF must have a matching ENDIF to terminate the conditional. Otherwise, an 'Unterminated conditional' message is generated at the end of each pass. An ENDIF without a matching IF causes a C error.

Listing Control Pseudo-Operations

Output to the listing file can be controlled by two pseudo-ops:

.LIST

.XLIST

If a listing is not being made, these pseudo-ops have no effect. LIST is the default condition. When a MIST is encountered, source and object code will not be listed until a .LIST is encountered.

The output of cross reference information is controlled by,.CREF and.XCREF. If the cross reference facility has not been invoked, CREF and XCREF have no effect. The default condition is CREF. When a.XCREF is encountered, no cross reference information is output until CREF is encountered.

The output of MACRO/REPT/IRP/IRPC expansions is controlled by three pseudo-ops:

LALL, SALL, and XALL.

Where:

.LALL lists the complete macro text for all expansions.

.SALL lists only the object code produced by a macro and not its text.

.XALL is the default condition; it is similar to SALL, except a source line is listed only it generates object code.

RELOCATION PSEUDO-OPERATIONS

The ability to create relocatable modules is one of the major features of MACRO80. Relocatable modules offer the advantages of easier coding and faster testing, debugging and modifying. In addition, it is possible to specify segments of assembled code that will later be loaded into RAM (the Data Relative segment) and ROM/PROM (the Code Relative segment).

The pseudo-operations that select relocatable areas are CSEG and DSEG. The ASEG pseudo-op is used to generate non-relocatable (absolute) code. The COMMON pseudo-op creates a common data area for every COMMON block that is named in the program.

The default mode for the Assembler is Code Relative. That is, assembly begins with a CSEG automatically executed and the location counter in the Code Relative mode, pointing to location 0 in the Code Relative segment of memory. All subsequent instructions will be assembled into the Code Relative segment of memory until an ASEG or DSEG or COMMON pseudo-op is executed.

For example, the first DSEG encountered sets the location counter to location zero in the Data Relative segment of memory. The following code is assembled in the Data Relative mode, where, it is assigned to the Data Relative segment of memory. If a subsequent CSEG is encountered, the location counter will return to the next free location in the Code Relative segment and so on.

The ASEG, DSEG, CSEG pseudo-ops never have operands. If you wish to alter the current value of the location counter, use the ORG pseudo-op.

ORG Pseudo-Op

At any time, the value of the location counter may be changed by use of the ORG pseudo-op.

The form of the ORG statement is:

```
ORG <exp>
```

where the value of <exp> will be the new value of the location counter in the current mode. All names used in <exp> must be known on pass 1 and the value of <exp> must be either Absolute or in the current mode of the location counter.

For example, the statements

```
DSEG
ORG 50
```

set the Data Relative location counter to. 50, relative to the start of the Data Relative segment of memory.

LINK-80

The LINK-80 Linking Loader (see Section C) combines the segments and creates each relocatable module in memory when the program is loaded. The origins of the relocatable segments are not fixed until the program is loaded and the origins are assigned by LINK-80. The command to LINK-80 may contain user-specified origins through the use of the /P (for Code Relative) and /D (for Data and COMMON segments) switches.

For example, a program that begins with the statements:

```
ASEG
ORG   800H
```

and is assembled entirely in Absolute mode will always load beginning at 800 unless the ORG statement is changed in the source file. However, the same program, assembled in Code Relative mode with no ORG statement, may be loaded at any specified address by appending the /P:<address> switch to the LINK-80 command string.

Relocation Before Loading

Two pseudo-ops, PHASE and.DEPHASE, allow code to be located in one area, but executed only at a different, specified area.

For example:

```
0000'                   .PHASE100H
0100  CD   0106  FOO:   CALL  BAZ
0103  C3   0007'        JMP   ZOO
0106  C9         BAZ:   RET
                        .DEPHASE -
 0007'C3   0005   ZOO:  JMP   5
```

All labels within a PHASE block are defined as the absolute value from the origin of the phase area. The code, however, is loaded in the current area (i.e., from 0' in this example). The code within the block can later be moved to 100H and executed.

Chapter Four

Macros and Block Pseudo-Operations

OVERVIEW

The Macro-80 Assembler provides complete facilities for constructing macros within the source program. Three repeat psuedo-operations as well as the macro definition operation are included. The following chapter explains the construction and use of the macro facilities.

MACROS AND BLOCK PSEUDO OPERATIONS

The macro facilities provided by MACRO-80 include three repeat pseudooperations: repeat (REPT), indefinite repeat (IRP), and indefinite repeat character (IRPC). A macro definition operation (MACRO) is also provided. Each of these four macro operations is terminated by the ENDM pseudo-operation.

Terms

For the purposes of discussion of macros and block operations, the following terms will be used:

1. **<dummy>** is used to represent a dummy parameter. All dummy parameters are legal symbols that appear in the body of a macro expansion.

2. **<dummylist>** is a list of <dummy>s separated by commas.

3. **<arglist>** is a list of arguments separated by commas. <arglist> must be delimited by angle brackets. Two angle brackets with no intervening characters (< >) or two commas with no intervening characters (, ,) enter a null argument in the list. Otherwise an argument is a character or series of characters terminated by a comma or >.

   With angle brackets that are nested inside an <arglist>, one level of brackets is removed each time the bracketed argument is used in an <arglist>.

   A **"quoted string"** is an acceptable argument and is passed as such. Unless enclosed in <brackets> or a "quoted string", leading and trailing spaces are deleted from arguments.

4. **<paramlist>** is used to represent a list of actual parameters separated by commas. No delimiters are required (the list is terminated by the end of line or a comment), but the rules for entering null parameters and nesting brackets are the same as described for <arglist>.

Block Pseudo Op-Codes

REPT-ENDM

```
            REPT <exp>
             .
             .
             .
            ENDM
```

The block of statements between REPT and ENDM is repeated <exp> times. <exp> is evaluated as a 16-bit unsigned number. If <exp> contains any external or undefined terms, an error is generated.

Example:

```
          X SET 0
            REPT   10     ;generates DB1-DB10
            SET    X+1
            DB     X
            ENDM
```

IRP-ENDM

```
            IRP <dummy>,<arglist>
             .
             .
             .
            ENDM
```

The <arglist> must be enclosed in angle brackets. The number of arguments in the <arglist> determines the number of times the block of statements is repeated. Each repetition substitutes the next item in the <arglist> for every occurrence of <dummy> in the block. If the <arglist> is null (i.e., o), the block is processed once with each occurrence of <dummy> removed.

For example:

```
            IRP X,<1,2,3,4,5,6,7,8,9,10>
            DB X
            ENDM
```

generates the same bytes as the REPT example.

IRPC-ENDM

```
IRPC <dummy>,string (or <string>)
.
.
.
ENDM
```

IRPC is similar to IRP but the arglist is replaced by a string of text and the angle brackets around the string are optional. The statements in the block are repeated once for each character in the string. Each repetition substitutes the next character in the string for every occurrence of <dummy> in the block.

For example:

```
IRPC  X,0123456789
DB    X+1
.ENDM
```

generates the same code as the two previous examples. MACRO

Often it is convenient to be able to generate a given sequence of statements from various places in a program, even though different parameters may be required each time the sequence is used.

This capability is provided by the MACRO statement.

```
<name> MACRO <dummylist>
.
.
.
ENDM
```

where <name> conforms to the rules for forming symbols. <name> is the name that will be used to invoke the macro. The <dummy>s in <dummylist> are the parameters that will be changed (replaced) each time the MACRO is invoked. The statements before the ENDM comprise the body of the macro.

During assembly, the macro is expanded everytime it is invoked but, unlike REPT/IRP/IRPC, the macro is not expanded when it is encountered.

In the listing, the expansion of the macro will be marked with a plus (+).

The form of a macro call is:

<name> <paramlist>

where <name> is the name supplied in the MACRO definition, and the parameters in <paramlist> will replace the <dummy>s in the MACRO <dummylist> on a one-to-one basis. The number of items in <dummylist> and <paramlist> is limited only by the length of a line.

The number of parameters used when the macro is called need not be the same as the number of <dummy>s in <dummylist>. If there are more parameters than <dummy>s, the extras are ignored. If there are fewer, the extra <dummy>s will be made null. The assembled code will contain the macro expansion code after each macro call.

NOTE: A dummy parameter in a MACRO/REPT/IRP/IRPC is always recognized exclusively as a dummy parameter. Register names such as A and B will be changed in the expansion if they were used as dummy parameters.

Here is an example of a MACRO definition that defines a macro called FOO:

```
FOO     MACRO X
Y       SET     0
        REPT    X
Y       SET     Y+1
        DB      Y
        ENDM
        ENDM
```

This macro generates the same code as the previous three examples when the call:

```
FOO 10
```

is executed.

Another example, which generates the same code, illustrates the removal of one level of brackets when an argument is used as an arglist:

```
FOO     MACRO  X
IRP     Y,<X>
DB      Y
ENDM
ENDM
```

When the call

```
 FOO   <1,2,3,4,5,6,7,8,9,10>
```

is made, the macro expansion looks like this:

```
IRP     Y,<1,2,3,4,5,6,7,8,9,10>
DB      Y
ENDM
```

ENDM

Every REPT, IRP, IRPC and MACRO pseudo-op must be terminated with the ENDM pseudo-op. Otherwise, the 'Unterminated REPT/IRP/IRPC/MACRO' message is generated at the end of each pass. An unmatched ENDM causes an O error.

EXITM

The EXITM pseudo-op is used to terminate a REPT/IRP/IRPC or MACRO call. When an EXITM is executed, the expansion is exited immediately and any remaining expansion or repetition is not generated. If the block containing the EXITM is nested within another block, the outer level continues to be expanded.

LOCAL

LOCAL <dummylist>

The LOCAL pseudo-op is allowed only inside a MACRO definition.

When LOCAL is executed, the Assembler creates a unique symbol for each <dummy> in <dummylist> and substitutes that symbol for each occurrence of the <dummy> in the expansion. These unique symbols are usually used to define a label within a macro, thus eliminating multiply-defined labels on successive expansions of the macro. The symbols created by the Assembler range from ..0001 to..FFFF. Users will therefore want to avoid the form..nnnn for their own symbols. If LOCAL statements are used, they must be the first statements in the macro definition.

Special Macro Operators and Forms

& The ampersand "&" is used in a macro expansion to concatenate text or symbols. A dummy parameter that is in a quoted string will not be substituted in the expansion unless it is immediately preceded by an ampersand. To form a symbol from text and a dummy, put an "&" between them.

For example:

```
ERRGEN   MACRO   X
ERROR&X  PUSH    B
         MVI     B,'&X'
         JMP     ERROR
         ENDM
```

In this example, the call ERRGEN A will generate:

```
ERRORA  PUSH   B
        MVI    B,'A'
        JMP    ERROR
```

;; In a block operation, a comment preceded by two semicolons is not saved as part of the expansion (i.e., it will not appear on the listing even under.LALL). A comment preceded by one semicolon, however, will be preserved and appear in the expansion.

! When an exclamation point is used in an argument, the next character is entered literally (i.e., !; and <;> ¯are equivalent).

NUL NUL is an operator that returns true if its argument (a parameter) is null. The remainder of a line after NUL is considered to be the argument to NUL.

The conditional

IF NUL argument

is false if, during the expansion, the first character of the argument is anything other than a semicolon or carriage return. It is recommended that testing for null parameters be done using the IFB and IFNB conditionals.

Using Z80 Pseudo-Ops

When using the 8080/Z80 assembler, the following Z80 pseudo-ops are valid. The function of each pseudo-op is equivalent to that of its 8080 counterpart.

Z80 pseudo-op	**Equivalent 8080 pseudo-op**
COND	IFT
ENDC	ENDIP
*EJECT	PAGE
DEFB	DB
DEFS	DS
DEFW	DW
DEFM	DB
DEFL	SET
GLOBAL	PUBLIC
EXTERNAL	EXTRN

The formats, where different, conform to the 8080 format. That is, DEFB and DEFW are permitted a list of arguments (as are DB and DW), and DEFM is permitted a string or numeric argument (as is DB).

MACRO-80 Reference Manual Index

Microsoft LINK-80 LOADER

SOFTWARE REFERENCE MANUAL

for Heath 8-bit digital computer systems

Part C
595-2286

HEATH COMPANY

BENTON HARBOR, MICHIGAN 49022

First Printing
Printed in the United
States of America

Table of Contents

LINK-80, Linking Loader

LINK.-80 Linking Loader

OVERVIEW

The following Section contains reference information about the LINK-80 Linking Loader. The Linking Loader is used to load the relocatable modules produced by the FORTRAN-80 compiler and the Macro-80 Assembler. The Linking Loader also links these modules to any internal routines that may be needed for execution of the relocatable module.

For example, to perform formatted random I/O several routines are referenced in the FORTRAN-80 library. These routines contain the actual machine language code needed in order to access the disk drive. The linker is used to link the main program to these routines.

The linker can also be used to create an absolute file that can be executed under HDOS. This file has the default extension ABS and is completely compatible with HDOS.

NOTE: Be sure to use only 8080 op-codes if the absolute file is intended to run on an H8 {with 8080 processor}.

LINK-80 COMMAND STRINGS

To run LINK-80, type L80 followed by a carriage return. LINK-80 will return the prompt "* ". Each command to LINK-80 consists of a string of file names and switches separated by commas:

Objdev1:filename.ext/switch1,objdev2:filename.ext,...

If the input device for a file is omitted, the default is SY0:. If the extension of an input file is omitted, the default is REL. After each line is typed, LINK-80 will load or search (see /S below) the specified files. After LINK-80 finishes this process, it will list all symbols that remained undefined followed by an asterisk.

Before execution begins, LINK-80 will always search the system library (FORLIB.REL) to satisfy any unresolved external references. The system library must reside on disk SY0:.

If the user wishes to first search non-standard libraries, the file names that are followed by /S should be appended to the end of the loader command string.

The following examples illustrate a typical use of the Linking Loader.

```
*SY1:TEST
```

This will load the file SY1:TEST.REL.

```
*SY1:TEST/N/E
```

This command string tells the linker to output the results of the linking and loading process in a file called TEST.ABS. The file will reside on drive SY1:. The /E will cause the linker to first search the system library to clear up any unresolved references, then exit to HDOS..

LINK-80 Switches

A number of switches may be given in the LINK-80 command string to specify actions affecting the loading process. Each switch must be preceded by a slash (/).

These switches are:

/R

Reset. Put loader back in its initial state. Use /R if the wrong file is accessed and it is necessary to re-start. /R takes effect as soon as it is encountered in a command string.

/E or /E:Name

Exit LINK-80 and return to HDOS. The system library will be searched on SY0: to satisfy any existing undefined globals.

The optional form /E:Name (where Name is a global symbol previously defined in one of the modules) uses Name for the start address of the program.

/G or /G:Name

Start execution of the program as soon as the current command line has been interpreted. The system library will be searched on SY0: to satisfy any existing undefined globals if they exist.

Before execution actually begins, LINK-80 prints two numbers and a BEGIN EXECUTION message. The two numbers are the start address and the address of the next available byte.

The optional form /G:Name (where Name is a global symbol previously defined in one of the modules) uses Name for the start address of the program.

/N

If a <filename>/N is specified, the program will be saved on disk under the selected name (with a default extension of ABS when a /E or /G is done. A jump to the start of the program is inserted so the program will run properly.

/P and /D

/P and /D allow the origin(s) to be set for the next program loaded. /P and /D take effect when seen and they have no effect on programs already loaded. The form is /P:<address> or /D:<address>, where <address> is the desired origin in the current radix. (Default radix is hex. /O sets radix to octal; /H to hex.)

Do not use /P or /D to load programs or data into the locations of the loader's jump to the start address (2280 to 2282) unless it is to load the start of the program there. If programs or data are loaded into these locations, the jump will not be generated.

If no /D is given, data areas are loaded before program areas for each module. If a /D is given, all Data and Common areas are loaded starting at the data origin and the program area at the program origin.

/U

List the origin and end of the program and data area and all undefined globals as soon as the current command line has been interpreted. The program information is only printed if a /D has been done.

/M

List the origin and end of the program and data area, all defined globals and their values, and all undefined globals followed by an asterisk. The program information is only printed if a /D has been done.

/S

Search the filename immediately preceding the /S in the command string to satisfy any undefined globals.

LINK-80 Error Messages

LINK-80 has the following error messages:

?No Start Address

A /G switch was issued, but no main program had been loaded.

?Loading Error

The last file given for input was not a properly formatted LINK-80 object file.

?Out of Memory

Not enough memory to load program. (A minimum of 40K RAM is required.)

?Command Error

Unrecognizable LINK-80 command string.

?<file> Not Found

<file>, as given in the command string, did not exist.

%2nd COMMON Larger /XXXXXX/

The first definition of COMMON block /XXXXXX/ was not the largest definition. Re-order module loading sequence or change COMMON block definitions. (See Chapter 9 in the FORTRAN Reference Manual for more information on the COMMON statement.)

%Mult. Def. Global YYYYYY

More than one definition for the global (internal) symbol YYYYYY was encountered during the loading process.

%Overlaying Program Area

A /D or /P will cause already loaded data to be destroyed.

?Intersecting Program Area Data

The program and data area intersect and an address or external chain entry is in this intersection. The final value cannot be converted to a current value since it is in the area intersection.

?Start Symbol - <name> - Undefined

After a /E: or /G: is given, the symbol specified was not defined.

Origin Above Loader Memory, Move Anyway (Y or N)? Below

After a /E or /G was given, either the data or program area has an origin or top which lies outside loader memory. If a Y <cr> is given, LINK-80 will move the area and continue. If anything else is given, LINK-80 will exit.

In either case, if a /N was given, the image will already have been saved.

?Can't Save Object File

A disk error occurred when the file was being saved. Usually this occurs when there is no more room left on the disk.

FORMAT OF LINK-80 COMPATIBLE OBJECT FILES

The following information is reference material for users who wish to know the load format of LINK-80 relocatable object files.

LINK-compatible object files consist of a bit stream. Individual fields within the bit stream are not aligned on byte boundaries, except as noted below. Use of a bit stream for relocatable object files keeps the size of object files to a minimum, thereby decreasing the number of disk reads/writes.

There are two basic types of load items: Absolute and Relocatable. The first bit of an item indicates one of these two types. If the first bit is a 0, the following 8 bits are loaded as an absolute byte. If the first bit is a 1, the next 2 bits are used to indicate one of four types of relocatable items:

00	Special LINK item (see below).
01	Program Relative. Load the following 16 bits after adding the current Program base.
10	Data Relative. Load the following 16 bits after adding the current Data base.
11	Common Relative. Load the following 16 bits after adding the current Common base.

Special LINK items consist of the bit stream 100 followed by:

A four-bit control field. An optional A field consisting of a two-bit address type that is the same as the two-bit field above except 00 specifies absolute address.

An optional B field consisting of 3 bits that give a symbol length and up to 8 bits for each character of the symbol.

A general representation of a special LINK item is:

```
1 00 xxxx    yy nn         zzz + characters of symbol name
             A field                  B field
```

xxxx	Four-bit control field (0-15 below)
yy	Two-bit address type field
nn	Sixteen-bit value
zzz	Three-bit symbol length field

The following special types have a B-field only:

0	Entry symbol (name for search)
1	Select COMMON block
2	Program name
3	Request library search
4	Reserved for future expansion

The following special LINK items have both an A field and a B field:

5	Define COMMON size
6	Chain external (A is head of address chain, B is name of external symbol)
7	Define entry point (A is address, B is name)
8	Reserved for future expansion

The following special LINK items have an A field only:

9	External + offset. The A value will be added to the two bytes starting at the current location counter immediately before execution.
10	Define size of Data area (A is size)
11	Set loading location counter to A
12	Chain address. A is head of chain, replace all entries in chain with current location counter. The last entry in the chain has an address field of absolute zero.
13	Define program size (A is size)
14	End program (forces to byte boundary)

The following special Link item has neither an A nor a B field:

15	End File

LINK-80 Linking Loader Index

Appendix A

8080 Op-Codes

INSTRUCTION SET

Mnemonic	Description	Mnemonic	Description
ACI	Add immediate to A with carry	DAA	Decimal adjust A
ADC M	Add memory to A with carry	DAD B	Add B & C to H & L
ADC r	Add register to A with carry	DAD D	Add D & E to H & L
ADD M	Add memory to A	DAD H	Add H & to L to H & L
ADD r	Add register to A	DAD SP	Add stack pointer to H & L
ADI	Add immediate to A	DCR M	Decrement memory
ANA M	And memory with A	DCR r	Decrement register
ANA r	And register with A	DCX B	Decrement B & C
ANI	And immediate with A	DCX D	Decrement D & E
CALL	Call unconditional	DCX H	Decrement H & L
CC	Call on carry	DCX SP	Decrement stack pointer
CM	Call on minus	DI	Disable Interrupt
CMA	Complement A	El	Enable Interrupts
CMC	Complement carry	HLT	Halt
CMP M	Compare memory with A	IN	Input
CMP r	Compare register with A	INR M	Increment memory
CNC	Call on no carry	INR r	Increment register
CNZ	Call on no zero	INX B	Increment B & C registers
CP	Call on positive	INX D	Increment D & E registers
CPE	Call on parity. even	INX H	Increment H & L registers
CPI	Compare immediate with A	INX SP	Increment stack pointer
CPO	Call on parity odd	JC	Jump on carry
CZ	Call on zero	JM	Jump on minus

Mnemonic	Description	Mnemonic	Description
JMP	Jump unconditional	RAL	Rotate A left through carry
JNC	Jump on no carry	RAR	Rotate A right through carry
JNZ	Jump on no zero	RC	Return on carry
JP	Jump on positive	RET	Return
JPE	Jump on parity even	RLC	Rotate A left
JPO	Jump on parity odd	RM	Return on minus
JZ	Jump on zero	RNC	Return on no carry
LDA	Load A direct	RNZ	Return on no zero
LDAX B	Load A indirect	RP	Return on positive
LHLD	Load H & L direct	RPE	Return on parity even
LXI B	Load immediate register Pair B & C	RPO	Return on parity odd
LXI D	Load immediate register Pair D & E	RRC	Rotate A right
LXI H	Load immediate register Pair H & L	RST	Restart
LXI SP	Load immediate stack pointer	RZ	Return on zero
MVIM	Move immediate memory	SBB M	Subtract memory from A with borrow
MVI r	Move immediate register	SBB r	Subtract register from A with borrow
MOV M, r	Move register to memory	SBI	Subtract immediate from A with borrow
MOV r,M	Move memory to register	SHLD	Store H & L direct
MOV r1, r2	Move register to register	SPHL	H & L to stack pointer
NOP	No-operation	STA	Store A direct
ORA M	Or memory with A	STAX B	Store A indirect
ORA r	Or register with A	STAX D	Store A indirect
ORI	Or immediate with A	STC	Set carry
OUT	Output	SUB M	Subtract memory from A
PCHL	H & L to program counter	SUB r	Subtract register from A
POP B	Pop register pair B & C off stack	SUI	Subtract immediate from A
POP D	Pop register pair D & E off stack	XCHG	Exchange D & E, H & L Registers
POP H	Pop register pair H & L off stack	XRA M	Exclusive Or memory with A
POP PSW	Pop A and Flags off stack	XRA r	Exclusive Or register with A
PUSH B	Push register B & C on stack	XRI	Exclusive Or immediate with A
PUSH D	Push register pair D & E on stack	XTHL	Exchange top of stack, H & L
PUSH H	Push register pair H & L on stack		
PUSH PSW	Push A and Flags on stack		

Appendix B

Z80 Op-Codes

INSTRUCTION SET

Mnemonic	Description
ADC HL, ss	Add with Carry Reg. pair ss to HL
ADC A, s	Add with carry operand s to Acc.
ADD A, n	Add value n to Acc.
ADD A, r	Add Reg. r to Acc.
ADD A, (HL)	Add location (HL) to Acc.
ADD A, (IX+d)	Add location (IX+d) to Acc.
ADD A, (IY+d)	Add location (IY+d) to Acc.
ADD HL, ss	Add Reg. pair ss to HL
ADD IX, pp	Add Reg. pair pp to IX
ADD IY, rr	Add Reg. pair rr to IY
AND s	Logical `AND' of operand s and Acc.
BIT b, (HL)	Test BIT b of location (HL)
BIT b, (IX+d)	Test BIT b of location (IX+d)
BIT b, (IY+d)	Test BIT b of location (IY+d)
BIT b, r	Test BIT b of Reg. r
CALL cc, nn	Call subroutine at location nn if condition cc is true
CALL nn	Unconditional call subroutine at location nn
CCF	Complement carry flag
CP s	Compare operand s with Acc.
CPD	Compare location (HL) and Acc. decrement HL and BC
CPDR	Compare location (HL) and Acc. decrement HL and BC, repeat until BC=0
CPI	Compare location (HL) and Acc. increment HL and decrement BC
CPIR	Compare location (HL) and Acc. increment HL, decrement BC repeat until BC=O
CPL	Complement Acc. (1's comp)
DAA	Decimal adjust Acc. .
DEC m	Decrement operand m
DEC IX	Decrement IX
DEC IY	Decrement IY
DEC ss	Decrement Reg. pair ss
DI	Disable interrupts
DJNZ e	Decrement B and jump relative if B=0
El	Enable interrupts
EX (SP), HL HL	Exchange the location (SP) and
EX (SP), IX	Exchange the location (SP) and IX
EX (SP) IY	Exchange the location (SP) and IY
EX AF, AF	Exchange the contents of AF and AF
EX DE, HL	Exchange the contents of DE and HL
EXX	Exchange the contents of BC, DE, HL with contents of BC', DE', HL' respectively
HALT	HALT (wait for interrupt or reset)

Mnemonic	Description
IM 0	Set interrupt mode 0
IM 1	Set interrupt mode 1
IM 2	Set interrupt mode 2
IN A, (n)	Load the Acc. with input from device n
IN r, (C)	Load the Reg. r with input from device (C)
INC (HL)	Increment location (HL)
INC IX	Increment IX
INC (IX+d)	Increment location (IX+d)
INC IY	Increment IY
INC (IY+d)	Increment location (IY+d)
INC r	Increment Reg. r
INC ss	Increment Reg. pair ss
IND	Load location (HL) with input from port (C), decrement HL and B
INDR	Load location (HL) with input from port (C), decrement HL and decrement B, repeat until B=0
INI	Load location (HL) with input from port (C), and increment HL and decrement B.
INIR	Load location (HL) with input from port (C), increment HL and decrement B, repeat until
B=0	
JP (HL)	Unconditional jump to (HL)
JP (IX)	Unconditional jump to (IX)
JP (IY)	Unconditional, jump to (IY)
JP cc, nn	jump to location nn if condition cc is true
JP nn	Unconditional jump to location nn
JP C, e	jump relative to PC+e if carry=1
JR e	Unconditional jump relative to PC+e
JP NC, e	jump relative to PC+e if carry=0
JR NZ, e	jump relative to PC+e if non zero (Z=0)
JR Z, e	jump relative to PC+e if zero (Z=1)
LD A, (BC)	Load Acc. with location (BC)
LD A, (DE)	Load Acc. with location (DE)
LD A, I	Load Acc. with I
LD A, (nn)	Load Acc. with location nn
LD A, R	Load Acc. with Reg. R
LD (BC), A	Load location (BC) with Acc.
LD (DE), A	Load location (DE) with Acc.
LD (HL), n	Load location (HL) with value n
LD dd, nn	Load Reg. pair dd with value nn
LD HL, (nn)	Load HL with location (nn)
LD (HL), r	Load location (HL) with Reg. r
LD I, A	Load I with Acc.
LF IX, on	Load IX with value nn
LD IX, (nn)	Load IX with location (nn)
LD (IX+d), n	Load location (IX+d) with value n
LD (IX+d), r	Load location (LX+d) with Reg. r
LD IY, no	Load IY with value nn
LD IY, (nn)	Load IY with location (nn)
LD (IY+d), n	Load location (IY+d) with value n
LD (IY+d), r	Load location (IY+d) with Reg. r
LD (nn), A	Load location (nn) with Acc.
LD (nn), dd	Load location (nn) with Reg. pair dd
LD (nn), HL	Load location (nn) with HL
LD (nn), IX	Load location (nn) with IX
LD (nn), IY	Load location (nn) with IY
LD R, A	Load R with Acc.
LD r, (HL)	Load Reg. r with location (HL)
LD r, (IX+d)	Load Reg. r with location (IX+d)
LD r, (IY+d)	Load Reg. r with location (IY+d)
LD r, n	Load Reg. r with value n
LD r, r'	Load Reg, r with Reg. r'
LD SP, HL	Load SP with HL
LD SP, IX	Load SP with IX
LD SP, IY	Load SP with IY

Mnemonic	Description
LDD	Load location (DE) with location (HL), decrement DE, HL and BC
LDDR	Load location (DE) with location (HL), decrement DE, HL and BC, repeat until BC=0
LDI	Load location (DE) with location (HL), increment DE, HL, decrement BC
LDIR	Load location (DE) with location (HL), increment DE, HL, decrement BC and repeat until BC=O
NEG	Negate Acc. (2's complement)
NOP	No operation
OR s	Logical 'OR' or operand s and Acc.
OTDR	Load output port (C) with location (HL) decrement HL and B, repeat until B=0
OTIR	Load output port (C) with location (HL), increment HL, decrement B, repeat until B=0
OUT (C), r	Load output port (C) with Reg. r
OUT (n), A	Load output port (n) with Acc.
OUTD	Load output port (C) with location (HL), decrement HL and B
OUTI	Load output port (C) with location (HL), increment HL and decrement B
POP IX	Load IX with top of stack
POP IY	Load IY with top of stack
POP qq	Load Reg. pair qq with top of stack
PUSH IX	Load IX onto stack
PUSH IY	Load IY onto stack
PUSH qq	Load Reg. pair qq onto stack
RES b, m	Reset Bit b of operand m
RET	Return from subroutine
RET cc	Return from subroutine if condition cc is true
RETI	Return from interrupt
RETN	Return from non maskable interrupt
RL m	Rotate left through carry operand m
RLA	Rotate left Acc. through carry
RLC (HL)	Rotate location (HL) left circular
RLC (IX+d)	Rotate location (IX+d) left circular
RLC (IY+d)	Rotate location (IY+d) left circular
RLC r	Rotate Reg. r left circular
RLCA	Rotate left circular Acc.
RLD	Rotate digit left and right between Acc. and location (HL)
RR m	Rotate right through carry operand m
RRA	Rotate right Acc. through carry
RRC m	Rotate operand m right circular
RRCA	Rotate right circular Acc.
RRD	Rotate digit right and left between Acc. and location (HL)
RST p	Restart to location p
SBC A, s	Subtract operand s from Acc. with carry
SBC HL, ss	Subtract Reg. pair ss from HL with carry
SCF	Set carry flag (C=1)
SET b, (HL)	Set Bit b of location (HL)
SET b, (IX+d)	Set Bit b of location (IX+d)
SET b, (IY+d)	**Set** Bit b of location (IY+d)
SET b, r	Set Bit b of Reg. r
SLA m	Shift operand m left arithmetic
SRA m	Shift operand m right arithmetic
SRL m	Shift operand m right logical
SUB s	Subtract operand s from Acc.
XOR s	Exclusive 'OR' operands and Acc

Appendix C

ASCII Codes

**DECIMAL TO OCTAL TO HEX
TO ASCII CONVERSION**

DEC	OCT	HEX	ASCII	DEC	OCT	HEX	ASCII	DEC	OCT	HEX	ASCII	DEC	OCT	HEX	ASCII
0	000	00	NUL	32	040	20	SPACE	64	100	40	@	96	140	60	`
1	001	01	SOH	33	041	21	!	65	101	41	A	97	141	61	a
2	002	02	STX	34	042	22	"	66	102	42	B	98	142	62	b
3	003	03	ETX	35	043	23	#	67	103	43	C	99	143	63	c
4	004	04	EOT	36	044	24	$	68	104	44	D	100	144	64	d
5	005	05	ENQ	37	045	25	%	69	105	45	E	101	145	65	e
6	006	06	ACK	38	046	26	&	70	106	46	F	102	146	66	f
7	007	07	BEL	39	047	27	ʻ	71	107	47	G	103	147	67	g
8	010	08	BS	40	050	28	(	72	110	48	H	104	150	68	h
9	011	09	HT	41	051	29	)	73	111	49	1	105	151	69	i
10	012	OA	LF	42	052	2A	*	74	112	4A	J	106	152	6A	j
11	013	OB	VT	43	053	2B	+	75	113	4B	K	107	153	6B	k
12	014	OC	FF	44	054	2C	,	76	114	4C	L	108	154	6C	1
13	015	OD	CR	45	055	2D	-	77	115	4D	M	109	155	6D	m
14	016	OE	SO	46	056	2E	Period	78	116	4E	N	110	156	6E	n
15	017	OF	SI	47	057	2F	/	79	117	4F	O	111	157	6F	o
16	020	10	DLE	48	060	30	0	80	120	50	P	112	160	70	p
17	021	11	DC1	49	061	31	1	81	121	51	Q	113	161	71	q
18	722	12	DC2	50	062	32	2	82	122	52	R	114	162	72	r
19	023	13	DC3	51	063	33	3	83	123	53	S	115	163	73	s
20	024	14	DC4	52	064	34	4	84	124	54	T	116	164	74	t
21	025	15	NAK	53	065	35	5	85	125	55	U	117	165	75	u
22	026	16	SYN	54	066	36	6	86	126	56	V	118	166	76	v
23	027	17	ETB	55	067	37	7	87	127	57	W	119	167	77	w
24	030	18	CAN	56	070	38	8	88	130	58	X	120	170	78	x
25	031	19	EM	57	071	39	9	89	131	59	Y	121	171	79	y
26	032	1A	SUB	58	072	3A	:	90	132	5A	Z	122	172	7A	z
27	033	1B	ESC	59	073	3B	;	91	133	5B	[	123	173	7B	{
28	034	1C	FS	60	074	3C	<	92	134	5C	\	124	174	7C	1
29	035	1D	GS	61	075	3D	=	93	135	5D	]	125	175	7D	}
30	036	1E	RS	62	076	3E	>	94	136	5E	^	126	176	7E	~
31	037	1F	US	63	077	3F	?	95	137	5F	_	127	177	7F	DELETE

NUL	Null; Tape Feed,
SOH	Start of Heading; Start of Message
STX	Start of Text; End of Address
ETX	End of Text; End of Message
EOT	End of Transmission; Shuts off TWX machines
ENQ	Enquiry; WRU
ACK	Acknowledge; RU
BEL	Rings Bell
BS	Backspace; For at Effector
HT	Horizontal TAB
LF	Line Feed or Space (New Line)
VT	Vertical TAB
FF	Form Feed (PAGE)
CR	Carriage Return
SO	Shift Out
SI	Shift In
DLE	Data Link Escape
DC1	Device Control 1; Reader on
DC2	Device Control 2; Punch on
DC3	Device Control 3; Reader off
DC4	Device Control 4; Punch off
NAK	Negative Acknowledge; Error
SYN	Synchronous Idle (SYNC)
ETB	End of Transmission Block; Logical End of Medium
CAN	Cancel (CANCL)
EM	End of Medium
SUB	Substitute
ESC	Escape
FS	File Separator
GS	Group Separator
RS	Record Separator
US	Unit Separator

Note that these characters (Octal 000 through 037), can be generated from the combination CTRL and the character in the same row, but in the third or fourth column (Octal 100 through 137 or 140 through 177).

That is, BEL is Control/G or /g, and CAN is Control/X or /x.

Appendix D

Microsoft Errors

FORTRAN-80 COMPILER FATAL ERROR MESSAGES

100 Illegal Statement Number

There is an illegal statement number in the source program. A statement number must be integer constant in the range 1-99999. Locate the number that does not conform to this rule and correct it.

101 Statement Unrecognizable or Misspelled

There is a statement that does not conform to the proper format. Check the general format of the statement in error and correct the statement.

102 Illegal Statement Completion

A statement does not conform to the proper format. Check the general format of the statement in error and correct the statement.

103 Illegal DO Nesting

A DO loop has been nested improperly. The range of each DO loop must be completely within the range of the next outer loop. Correct the illegally nested DO.

104 Illegal Data Constant

An illegal data constant has been discovered. Check the statement in error and change the constant to conform to the rules for constant construction.

105 Missing Name

A valid symbolic name was expected in the source program. Check the statement in error and supply the proper name.

106 Illegal Procedure Name

A procedure has been assigned an illegal name. The name must begin with an alphabetic character and be no more than six characters in length. Change the statement in error so it conforms to this requirement.

107 Invalid DATA Constant or Repeat Factor

A DATA statement has an invalid constant or repeat factor. Literal data must be enclosed in single quotes. The repeat factor must not attempt to assign more data than is valid. Change the DATA statement to conform to these requirements.

108 Incorrect Number of DATA Constants

A DATA statement has too many or too few constants. A valid DATA statement must have the same number of variables as constants. Change the DATA statement in error so that is conforms to this requirement.

109 Incorrect Integer Constant

An incorrect integer constant has been discovered. An integer constant must be in the range -32768 to +32767 inclusive. The integer constant must also not contain any decimal points, commas, or alphabetic characters. Correct the statement in error so it conforms to these requirements.

110 Invalid Statement Number

A statement number is invalid. The statement label must be in the range 1-99999. Correct the statement label.

111 Not a Variable Name

A variable name was expected. The variable name must begin with an alphabetic character and be no more than six characters in length. Correct the variable name so that it conforms to these requirements.

112 Illegal Logical Form Operator

An illegal logical operator has been discovered. The logical operators must be of the form: .NOT., .AND., .OR. and .XOR.. It is also invalid to have two contiguous logical operators except when the second operator is .NOT.. Verify that the operator is constructed properly. Correct the statement in error.

113 Data Pool Overflow

Too much memory has been requested for data storage. Large arrays are usually responsible for exceeding the storage capabilities. The amount of memory needed to store an array is a function of the data type of the array and the number of elements in the array. Change either the number of elements in the array or the data type of the array.

114 Literal String Too Large

There is a literal string that is too large. The number of characters in a literal string should be no greater than the number of bytes required by the corresponding variable; i.e.: one character for a logical variable, up to two characters for an integer variable, up to four characters for a real variable, and up to eight characters for a double-precision variable. Correct the literal string so that it conforms to these requirements.

115 Invalid Data List Element in I/O

There is an invalid element in an I/O list. A valid element in an I/O list must be a variable, an array element or array name. Correct the I/O list so that the elements of the I/O list are valid elements.

116 Unbalanced DO Nest

An unbalanced DO nest has been discovered. Each DO loop must have a valid terminal statement. Correct the loop with the invalid terminal statement.

117 Identifier Too Long

An identifier is too long. The identifier must be no more than six characters in length. Correct the illegal identifier.

118 Illegal Operator

An illegal operator has been discovered. The valid arithmetic operators are: ** , * , / , + , -. The valid relational operators are: .LT., .LE., .EQ., .NE., .GT., .GE.. The valid logical operators are: .NOT., .AND., .OR., .XOR.. Correct the illegal operator so it will conform to the proper format.

119 Mismatched Parenthesis

There is a mismatched parenthesis. Correct the statement so each parenthesis is matched.

120 Consecutive Operators

Consecutive operators were encountered in the source program. Each operator must have a valid operand. Correct the statement so that each operator has an operand.

121 Improper Subscript Syntax

A improper subscript has been discovered. Subscripts must be written in one of the following forms:

```
K      C*V      V-K
V      C*V+K    C*V-K
V+K
```

where C and K are integer constants and V is an integer variable name. Verify that each subscript follows this format. Correct the improper subscript.

122 Illegal Integer Quantity

An integer constant or expression value is outside the range -32768 to +32767. Correct the value of the integer or expression so that it falls within the legal range (-32768 to +32767).

123 Illegal Hollerith Construction

An illegal Hollerith string has been encountered. The Hollerith string is constructed by enclosing the entire string of characters in a set of single quote marks. Two quotation marks in succession may be used to represent the quotation mark character within the string. Correct the Hollerith string so that is conforms to these requirements.

124 Backwards DO reference

A backward DO reference has been discovered. The terminal statement of a DO loop must physically follow its associated DO. Verify that the terminal statement physically follows the associated DO. Correct the backward DO reference.

125 Illegal Statement Function Name

A statement function has been assigned an illegal symbolic name. The name must begin with a alphabetic character and be no more than six characters in length. The statement function name must also be a unique name. Correct the statement so that the function name adheres to these requirements.

126 Illegal Character for Syntax

A character has been encountered that is illegal in the contex it is used. Correct the illegal character.

127 Statement Out of Sequence

One of the statements is out of sequence. The statements within a program unit must be in the following order:

1. PROGRAM, SUBROUTINE, FUNCTION, BLOCK DATA
2. Type, EXTERNAL, DIMENSION
3. COMMON
4. EQUIVALENCE
5. DATA
6. Statement Functions
7. Executable Statements

Verify that the statements adhere to the above requirement. Correct the out of sequence statement.

128 Missing Integer Quantity

An integer quantity was expected but not found. Several statements require that an integer quantity be present. For example, the DIMENSION statement requires an integer quantity be present. Correct the statement by providing the proper integer quantity.

129 **Invalid Logical Operator**

An invalid Logical operator was discovered. The valid Logical operators are .NOT., .AND., .OR., .XOR.. Verify that the operators conform to this format. Correct the invalid operator.

130 Illegal Item in Type Declaration

An illegal item was encountered after a type statement. Only an array declarator, an array, a variable or a FUNCTION name can follow a type specification statement. Verify that the type statement is correctly structured. Correct the illegal statement.

131 Premature End Of File on Input Device

The FORTRAN-80 Compiler has reached an end of file before it was anticipated. This happens when the END statement is omitted. Verify that the last physical statement in the program unit is an END statement.

132 Illegal Mixed Mode Operation

The logical, relational, and arithmetic operators have been used together in a manner that is not appropriate. Correct the illegal usage.

133 Function Call with No Parameters

A Function has been referenced and no parameters were provided in the reference. There must be at least one parameter passed to the Function. Verify that this condition has been met. Correct the reference to the Function.

134 Stack Overflow

The FORTRAN-80 Compiler has overflowed the stack. This occurs when a very large program is compiled. To correct it, use the /P switch during the compilation process. The /P switch will allocate 100 extra bytes of stack space.

135 **Illegal Statement Following Logical IF**

The statement contained in a logical IF was not valid. Verify that this statement is not a DO or another logic IF. Correct the illegal statement.

FORTRAN-80 COMPILER WARNING MESSAGES

0 Duplicate Statement Label

There is a duplicate statement label in the source program. Each statement label must be unique within the program unit. Change the duplicate label so it has a unique value.

1 Illegal DO Termination

A DO loop is terminated by an illegal statement. The terminal statement may not be an Arithmetic IF, GO TO, RETURN, STOP, PAUSE or another DO. Correct the illegal terminal statement.

2 Block Name = Procedure Name

A COMMON block has been assigned the same symbolic name as the main program. The COMMON block name must be different than any procedure names used throughout the program. Correct the illegal statement.

3 Array Name Misuse

An array name has been used where it is not appropriate. Correct the illegal reference to an array name.

4 COMMON Name Usage

A COMMON block name has been used as a variable. The name of a COMMON block may appear more than once in the same COMMON statement, or in more than one COMMON statement. The COMMON block name must be different than any variable names used throughout the program. Correct the illegal statement.

5 Wrong Number of Subscripts

An element of an array has been referenced with the wrong number of subscripts. The number of subscript expressions must be the same as the specified dimensionality of the array. (An exception to this rule is the EQUIVALENCE statement.) Correct the illegal subscript.

6 Array Multiply EQUIVALENCED within a Group

Two elements of the same array have been EQUIVALENCED. It is invalid to EQUIVALANCE two elements of the same array or two elements belonging to the same or different COMMON blocks. Correct the illegal statement.

7 Multiple EQUIVALENCE of COMMON

An attempt was made to EQUIVALENCE two elements belonging to the same or different COMMON blocks. It is invalid to EQUIVALENCE two elements of the same array or two elements belonging to the same or different COMMON blocks. Correct the illegal statement.

8 COMMON Base Lowered

While attempting to EQUIVALENCE elements in COMMON, an attempt was made to extend the COMMON past the recognized beginning of COMMON storage. COMMON block size may be increased only from the last element established by the COMMON statement forward. Correct the illegal statement.

9 Non-COMMON Variable in BLOCK DATA

There is a non-COMMON variable in a BLOCK DATA subprogram. If any element in a COMMON block is to be initialized by a BLOCK DATA subprogram, all elements of the block must be listed in the COMMON statement. Include the variable in a COMMON statement.

10 Empty List for Unformatted WRITE

There is an unformatted WRITE statement without an I/O list. The unformatted WRITE statement must have an I/O list. Correct the illegal statement.

11 Non-Integer Expression

An integer expression was expected but not found. Verify that the integer expression conforms to the rules for construction of expressions. Correct the illegal expression.

12 Operand Mode Not Compatible with Operator

An arithmetic, logical or relational operand was not compatible with the associated operator. Verify that the expression conforms to the rules for expression construction. Correct the invalid expression.

13 Mixing of Operand Modes Not Allowed

The arithmetic, logical, or relational operands have been used together in a manner that is not appropriate. Correct the illegal usage.

14 Missing Integer Variable

An integer variable was expected but not found. For example, ASSIGN 100 TO 4 is illegal, an integer variable name should follow the `TO' but does not. Correct the statement so that a valid integer name is included.

15 Missing Statement Number on FORMAT

There is a FORMAT statement without a statement number. The FORMAT statement must be labeled with a valid statement number. Correct the illegal statement.

16 Zero Repeat Factor

A FORMAT statement has a zero repeat factor preceding a field descriptor. The repeat factor must be a non-zero, positive integer. Correct the illegal repeat factor.

18 Format Nest Too Deep

A FORMAT statement has more than two levels of parentheses. Up to two levels of parentheses, including the parentheses required by the FORMAT statement, are permitted. Correct the illegal FORMAT statement.

19 Statement Number Not FORMAT Associated

A formatted I/O or ENCODE/DECODE statement referenced a statement number which was not FORMAT associated. A formatted I/O or ENCODE/DECODE statement must reference a FORMAT statement. Correct the illegal statement.

20 Invalid Statement Number Usage

A statement number has been used in a context that is invalid. Correct the invalid statement number reference.

21 No Path to this Statement

There is a statement with no path to it. A statement with no path to it will never be executed. Correct the program logic so that the statement will be included in the logical flow.

22 Missing Do Termination

There is a DO loop without a terminal statement. Each DO loop must have a valid terminal statement. Insert a valid terminal statement in the source program.

23 Code Output in BLOCK DATA

A BLOCK DATA subprogram contains an executable statement. A BLOCK DATA subprogram must contain only type, EQUIVALENCE, DATA, COMMON, and DIMENSION statements. Correct the illegal BLOCK DATA subprogram.

24 Undefined Labels Have Occurred

A reference was made to an undefined statement label. All labels referenced must be valid statement numbers. Correct the undefined label.

25 RETURN in a Main Program

There is a RETURN statement in a main program. The RETURN statement is used to mark the logical end of a subprogram. It must not appear in a main program. Remove the RETURN statement from the main program.

27 Invalid Operand Usage

An arithmetic, relational, or logical operand has been used in a manner that is not appropriate. Correct the illegal statement so that it conforms to the rules for expression construction.

28 Function with no Parameter

A function has been constructed or referenced and no parameters were listed. The function definition must include at least one dummy parameter. The reference to a function must provide a list of parameters for use by the function. Correct the illegal statement.

29 Hex Constant Overflow

A hex constant is too large. The number of hex characters that can be stored should be no greater than the number of bytes required by the corresponding variable; one character for a Logical variable, up to two characters for an Integer variable, up to four characters for a Real variable, and up to eight characters for a Double-Precision variable.

30 Division by Zero

An attempt was made to divide by zero. Correct the statement in error.

32 Array Name Expected

An array name was expected but not found. For example, the ENCODE/DECODE statements require that an array name be referenced. Correct the statement to include an array name.

33 Illegal Argument to ENCODE/DECODE

There is an illegal argument in either an ENCODE or a DECODE statement. The proper formats for the ENCODE/DECODE statements are:

```
ENCODE(A,F) K DECODE(A,F) K
```

where:

A is an array name
F is a FORMAT statement number K is an I/O list

Correct the illegal ENCODE/DECODE statement.

RUNTIME ERRORS
Fatal Errors

ID Illegal FORMAT Descriptor

A FORMAT statement has an illegal descriptor. The legal descriptors are F, E, D, G, I, A, H, L, and X. Correct the illegal descriptor.

FO FORMAT Field Width is Zero

The width of a FORMAT field is zero. The field width is a non-zero, positive constant used to define the number of digits in the external data representation. Correct the illegal field width.

MP Missing Period in FORMAT

A period was expected but not found. The field descriptors E, F, G, and D require the use of a period between the field width specifier and the fractional digit specifier. Correct the illegal FORMAT statement.

FW FORMAT Field Width is Too Small

An attempt was made to transfer data larger than the field width specifier. The field width specifier defines the total width of the field (including digits, decimal points, algebraic signs). Increase the size of the field width specifier.

I/O Transmission Error

An error occurred while communication was being established with an I/O device. This error usually occurs when output is attempted to a hard copy device without the proper device driver being LOADed into memory. This error can also occur when an attempt is made to perform I/O to a disk drive that has not been MOUNTed, or this error will also occur if output is attempted to a disk with no room left on it. Take the appropriate action to correct this problem. (LOAD the device driver, MOUNT the disk drive, etc.)

ML Missing Left Parentheses in FORMAT

A FORMAT statement has been discovered without a left parentheses. The FORMAT statement requires the use of a left parentheses. Correct the illegal FORMAT statement.

DZ Division by **Zero**

An attempt was made to divide by zero. Correct the illegal statement.

LG Illegal Argument to LOG Function

The library function LOG was passed an argument that was negative or zero. The LOG function is undefined when the argument is negative or zero. Correct the illegal statement.

SQ Illegal Argument to SQRT Function

The library function SQRT was passed an argument that was negative. The SQRT function is undefined when the argument is negative. Correct the illegal statement.

DT Data Type Does Not Agree with FORMAT

An attempt was made to use an integer field descriptor with a real variable or to use a real field descriptor with an integer variable. Correct the FORMAT statement associated with the READ or WRITE, ENCODE or DECODE.

EF EOF Encountered on READ

An attempt was made to READ beyond the last record of the file. Use the END= option to avoid this error.

RUNTIME ERROR MESSAGES
Warning Errors

TL To Many Left Parentheses in FORMAT

There were too many left parentheses in a FORMAT statement during execution of the program. This error is usually a result of incorrectly modifying a FORMAT statement during runtime. Correct any FORMAT statement that is invalid.

DE Decimal Exponent Overflow

A number in the input stream had an exponent larger than 99. Correct the invalid exponent.

IS Integer Size Too Large

An integer constant or expression value is outside the range -32768 to +32767. Correct the value of the integer constant so that it is within the legal range (-32768 to +32767).

IN Input Record Too Long

A formatted READ statement has attempted to input more than 255 bytes. Correct the program logic to avoid this condition.

OV Arithmetic Overflow

An arithmetic operation has resulted in a data value that is too large. Correct the statement so that the magnitude of the data is within the legal range for the data type.

CN Conversion Overflow on REAL to INTEGER Conversion

An attempt was made to convert a number outside the legal range for integer numbers to the integer data type. The legal range for integer numbers is -32768 to +32767. Correct the value of the integer so that it is within the legal range (-32768 to +32767).

SN Argument to SIN Too Large

The argument to the SIN function is too large. The SIN function is undefined for unusually large numbers. Correct the program.

A2 Both Arguments of ATAN2 are 0

The library function ATAN2 has been referenced and both arguments to the function are zero. The library function is undefined when both arguments are zero.

IO Illegal I/O Operation

An illegal I/O operation was attempted. For example, input from a hard copy device would be an illegal operation. An attempt to randomly access a device not capable of random access would also be an illegal operation. Assigning a nonvalid logical unit number would also be an illegal operation. Correct the illegal statement.

RC Negative Repeat Count in FORMAT

A FORMAT statement has been found to contain a negative repeat factor preceding a field descriptor. The repeat factor must be a positive integer. Correct the invalid FORMAT statement.

MACRO-80 ERROR MESSAGES

MACRO-80 errors are indicated by a one-character flag in column one of the listing file. If a listing file is not being printed on the terminal, each erroneous line is also printed or displayed on the terminal. Below is a list of the MACRO-80 Error Codes:

Error Codes

A	Argument error - Argument to pseudo-op is not in correct format or is out of range (.PAGE 1; RADIX 1; PUBLIC 1; STAX H; MOV M,N; INX C).
C	Conditional nesting error - ELSE without IF, ENDIF without IF, two ELSEs on one IF.
D	Double Defined symbol - Reference to a symbol which is multiply defined.
E	External error - Use of an external illegal in context (e.g., FOO SET NAME ; MVI A,2-NAME).
M	Multiply Defined symbol - Definition of a symbol which is multiply defined.
N	Number error - Error in a number, usually a bad digit (e.g., 8Q).
O	Bad opcode or objectionable syntax - ENDM, LOCAL outside a block; SET, EQU or MACRO without a name; bad syntax in an opcode (MOV A:); or bad syntax in an expression (mismatched parenthesis, quotes, consecutive operators, etc.).
P	Phase error - Value of a label or EQU name is different on pass 2.
Q	Questionable - Usually means a line is not terminated properly. This is a warning error (e.g., MOV A,B,).

R Relocation -
Illegal use of relocation in expression, such as abs-rel. Data, code and COMMON areas are relocatable.

U Undefined symbol -
A symbol referenced in an expression is not defined. (For certain pseudo-ops, a V error is printed on pass 1 and a U on pass 2.)

V Value error -
On pass 1 a pseudo-op which must have its value known on pass 1 (e.g., RADIX, .PAGE, DS, IF, IFE, etc.), has a value which is undefined later in the program, a U error will not appear on the pass 2 listing.

Error Messages:

`No end statement encountered on input file'
No END statement: either it is missing or it is not parsed due to being in a false conditional, unterminated IRP/IRPC/REPT block or terminated macro.

'Unterminated conditional'
At least one conditional is unterminated at the end of the file.

'Unterminated REPT/IRP/IRPC/MACRO'
At least one block is unterminated.

[xx] [No] Fatal error(s) [,xx warnings]
The number of fatal errors and warnings. The message is listed on the console and in the list file.

LINK-80 ERROR MESSAGES

?No Start Address

A /G switch was issued, but no main program had been loaded.

?Loading Error

The last file given for input was not a properly formatted LINK-80 object file.

?Out of Memory

Not enough memory to load program.

(A minimum of 40K RAM is required.)

?Command Error

Unrecognizable LINK-80 command string.

?<file> Not Found

<file>, as given in the command string, did not exist.

%2nd COMMON Larger /XXXXXX/

The first definition of COMMON block /XXXXXX/ was not the largest definition. Re-order module loading sequence or change COMMON block definitions. (See Chapter 9 in the FORTRAN Reference Manual for more information on the COMMON statement.)

%Mult. Def. Global YYYYYY

More than one definition for the global (internal) symbol YYYYYY was encountered during the loading process.

%Overlaying Program Area
Data

A /D or /P will cause already loaded data to be destroyed.

?Intersecting Program Area
Data

The program and data area intersect and an address or external chain entry is in this intersection. The final value cannot be converted to a current value since it is in the area intersection.

?Start Symbol - <name> - Undefined
After a /E: or /G: is given, the symbol specified was not defined.

Origin Above Loader Memory, Move Anyway (Y or N)?
Below

After a /E or /G was given, either the data or program area has an origin or top which lies outside loader memory. If a Y <cr> is given, LINK-80 will move the area and continue. If anything else is given, LINK-80 will exit.

In either case, if a /N was given, the image will already have been saved.

?Can't Save Object File
A disk error occurred when the file was being saved. Usually this occurs when there is no more room left on the disk.