Microsoft®
FORTRAN Compiler

for the MS™-DOS Operating System

User's Guide

Microsoft Corporation

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

© Copyright Microsoft Corporation, 1981, 1982, 1984

If you have comments about the software or these manuals, please complete the Software Problem Report at the back of the *Microsoft FORTRAN Reference Manual* and return it to Microsoft Corporation.

Microsoft and the Microsoft logo are registered trademarks of Microsoft Corporation.

MS is a trademark of Microsoft Corporation.

Intel is a registered trademark of Intel Corporation.

Document Number 8206A-320-04
Part Number 005-014-001

Contents

System Requirements ix

Package Contents ix

Microsoft FORTRAN System Software x
Documentation xi

How to Use This Guide xi

1 Introduction to the Microsoft FORTRAN Compiler 1

1.1 Options 4
1.2 Previous Versions 4
1.3 Syntax Notation 4
1.4 Learning More About FORTRAN 5

2 Getting Started 7

2.1 Preliminary Procedures 9
2.2 Program Development 11
2.3 Vocabulary 15

3 A Sample Session 19

3.1 Creating a Microsoft FORTRAN Source File 22
3.2 Compiling Your Microsoft FORTRAN Program 23
3.3 Linking Your Microsoft FORTRAN Program 28
3.4 Executing Your Microsoft FORTRAN Program 30

4 Options for Compiling and Linking 33

4.1 MS-DOS 2.0 Interface Library 35
4.2 Alternative Linkers 35
4.3 16-Bit Integers as Variables 36
4.4 Floating-Point Options 37
4.5 Changing the Default Math Library 39
4.6 Best Cases for Compilation and Execution 39

5 More About Compiling 41

5.1 Files Written by the Compiler 43
5.2 Filename Conventions 45
5.3 Starting the Compiler 49

6 More About Linking 53

6.1 Files Read by the Linker 55
6.2 Files Written by the Linker 60
6.3 The Overlay Linker 62
6.4 Linker Switches 64

7 Using a Batch Command File 67

8 Compiling and Linking Large Programs 69

8.1 Avoiding Limits on Code Size 71
8.2 Avoiding Limits on Data Size 71
8.3 Working With Limits on Compile Time Memory 75
8.4 Working With Limits on Disk Memory 77
8.5 Minimizing Load Module Size 82

9 Using Assembly Language Routines 85

9.1 Calling Conventions 87
9.2 Internal Representations of Data Types 89
9.3 Interfacing to Assembly Language Routines 91

10 Advanced Topics 99

10.1 The Structure of the Compiler 101
10.2 An Overview of the File System 106
10.3 Runtime Architecture 108
10.4 Floating-Point Operations 123
10.5 MS-DOS 2.0 Issues 128

Appendices 131

A Differences From Earlier Versions of Microsoft FORTRAN 133

B Microsoft FORTRAN File Control Block 137

C Real Number Conversion Utilities 139

D Structure of External Microsoft FORTRAN Files 141

E Microsoft FORTRAN Scratch File Names 143

F Customizing i8087 Interrupts 145

G Exception Handling for 8087 Math 151

H Microsoft LINK Error Messages 157

Index 161

Figures

Figure 2.1 Program Development 13

Figure 9.1 Contents of the Frame 88

Figure 9.2 Two-Byte Return Value 92

Figure 9.3 Four-Byte Return Value 92

Figure 9.4 Stack Before Transfer to IADD 93

Figure 9.5 Stack Before Transfer to RADD 95

Figure 10.1 The Structure of the Compiler 101

Figure 10.2 The Unit U Interface 107

Figure 10.3 Memory Organization 111

Figure 10.4 Microsoft FORTRAN Program Structure 115

Tables

Table 2.1 A Suggested Disk Setup 10

Table 3.1 Files Used by the Microsoft FORTRAN Compiler 27

Table 5.1 Default File Specifications 47

Table 6.1 Linker Defaults 57

Table 6.2 Microsoft LINK Switches 65

Table 10.1 Front End Compilation Procedures 103

Table 10.2 Unit Identifier Suffixes 109

Table 10.3 Error Code Classification 120

Table 10.4 Runtime Values in BRTEQQ 121

System Requirements

The Microsoft FORTRAN Compiler can be used with any computer that has one disk drive and a minimum of 150K random access memory available after the operating system is loaded. (The MS-DOS utility CHKDSK will tell you how much RAM is available.)

We recommend at least two drives, however, for easier operation. The compiler can successfully take advantage of at least 196K RAM. Your machine should run MS-DOS.

The current implementation of the Microsoft FORTRAN Compiler can take advantage of, but does not require, an 8087 numeric coprocessor. If you have one, additional preliminary procedures may be required; see Section 2.1.3, "If You Have an 8087 Coprocessor," for details.

Two versions of Microsoft LINK are available for your use. They are LINK.EXE (the default linker) and LINK.V2 (the optional MS-DOS 2.0 linker which supports path names and overlays). You must use either one or the other to link your program modules (see Chapter 4, "Options for Compiling and Linking"). Microsoft LINK is the standard MS-DOS linking utility.

Package Contents

The Microsoft FORTRAN Compiler package includes four disks and one documentation binder, containing two manuals.

Microsoft FORTRAN System Software

The software for the Microsoft FORTRAN Compiler contains the following files on disk:

File	Contents
FOR1.EXE	Pass one of the Microsoft FORTRAN Compiler
PAS2.EXE	Pass two of the Microsoft FORTRAN Compiler
PAS3.EXE	Pass three of the Microsoft FORTRAN Compiler
FORTRAN.LIB	The default MS-FORTRAN runtime library
FORTRAN.MAP	The link map for FORTRAN.LIB
MATH.LIB	The default floating-point package library
MATH.MAP	The link map of MATH.LIB
8087.LIB	An auxiliary library for use with programs that are to run only on machines with the 8087 coprocessor installed and whose size you wish to reduce.
8087.MAP	The link map of 8087.LIB
DOS2FOR.LIB	An auxiliary library containing an MS-DOS version 2.0 file system
DOS2FOR.MAP	A map of DOS2FOR.LIB
ALTMATH.LIB	An auxiliary library containing high-speed floating-point support routines
ALTMATH.MAP	A map of ALTMATH.LIB
DECMATH.LIB	An auxiliary library containing decimal floating-point support routines
DECMATH.MAP	The map of DECMATH.LIB
LINK.EXE	The default Microsoft Linker
LINK.V2	Optional version of Microsoft LINK (MS-DOS 2.0)

NULF.OBJ	The dummy file system
NULE6.OBJ	The dummy error system
ENTX6L.ASM	The assembler source of the execution control module that initializes and terminates every program
DEMO.FOR	Bubble sort demonstration program
README.DOC	If present, this file contains documentation that is more up to date than the most current printed documentation.

Documentation

Documentation for the Microsoft FORTRAN Compiler is provided in the following two manuals:

Microsoft FORTRAN User's Guide

This manual provides an introduction to compiling and linking, a sample session, and a technical reference for the Microsoft FORTRAN Compiler.

Microsoft FORTRAN Reference Manual

This manual describes the grammar and use of the Microsoft FORTRAN language. With the exception of any recent changes noted in a README.DOC file, this is the language supported by the Microsoft FORTRAN Compiler.

How to Use This Guide

The *Microsoft FORTRAN User's Guide* describes the operation of the Microsoft FORTRAN Compiler, from the most rudimentary procedures to more advanced topics that may be of interest only to experienced programmers.

The document assumes that you have a working knowledge of the Microsoft FORTRAN language and MS-DOS. For information on programming in FORTRAN, see Section 1.4, "Learning More About FORTRAN."

Chapter 1, "Introduction to the Microsoft FORTRAN Compiler," introduces you to the Microsoft FORTRAN Compiler and its features. Microsoft FORTRAN version 3.20 offers a wealth of options for developing your programs. Be sure to consult Chapter 4, "Options for Compiling and Linking," for the discussion of how you can use these options to customize your own programs according to your requirements for portability and performance.

Chapter 2, "Getting Started," discusses several procedures you should perform before compiling and linking your first program, describes the process of program development, and provides a short vocabulary for those who may be unfamiliar with the terms used in this document. Those who are already familiar with compiling and linking may wish to skip most of this chapter, but should read Section 2.1, "Preliminary Procedures," before proceeding.

Chapter 3, "A Sample Session," provides a step-by-step walk-through of each of the procedures that follow the writing of a program: compiling, linking, and running.

Chapter 4, "Options for Compiling and Linking," provides you with a summary of the optional libraries and compiler features that will aid you in your program development.

Chapter 5, "More About Compiling," and Chapter 6, "More About Linking," supplement the material in Chapter 3 on compiling and linking, respectively.

Chapters 1 through 6 should be read in their entirety by the first-time user of the Microsoft FORTRAN Compiler.

Chapter 7, "Using a Batch Command File," and Chapter 8, "Compiling and Linking Large Programs," provide information on these topics for the programmer who has moved beyond the basics.

Chapter 9, "Using Assembly Language Routines," provides information for the experienced programmer who requires supplementary routines written in assembly language.

Chapter 10, "Advanced Topics," provides additional technical information on compiler structure, the Microsoft FORTRAN file system, floating-point issues, and runtime architecture.

Appendix A, "Differences From Earlier Versions of Microsoft FORTRAN," provides a list of the significant changes between Microsoft FORTRAN 3.2 and previous versions since 3.0.

Appendices B through E provide information specific to this implementation of the Microsoft FORTRAN Compiler for MS-DOS.

Appendix F, "Customizing i8087 Interrupts," describes preliminary procedures that may be necessary if you have an 8087 numeric coprocessor.

Appendix G, "Exception Handling for 8087 Math," describes the environment CONTROL words and exception handling conditions of the 8087 coprocessor.

Appendix H,"Microsoft LINK Error Messages," provides a list of Microsoft LINK error messages.

Chapter 1

Introduction to the Microsoft FORTRAN Compiler

1.1 Options 4

1.2 Previous Versions 4

1.3 Descriptive Notation 4

1.4 Learning More About FORTRAN 5

The Microsoft® version 3.2 FORTRAN Compiler (MS-FORTRAN Compiler) runs under version 1.25 of the Microsoft Disk Operating System (MS-DOS). MS-FORTRAN Compiler accepts programs written in the language defined by the *Microsoft FORTRAN Reference Manual.* The MS-FORTRAN language conforms to the ANSI X3.9-1978 subset FORTRAN requirements, but has many features of the full language plus extensions designed to optimize FORTRAN in the microcomputer environment. Microsoft FORTRAN is especially adept at quickly handling complex financial and scientific algorithms.

The Microsoft FORTRAN Compiler passes on the advantages of a high-level programming language without sacrificing speed by generating native code. Low-level escapes to the machine level allow Microsoft FORTRAN programs to achieve speeds comparable to assembly language. MS-FORTRAN also generates code for fast numeric processing in the 8087 processing environment and provides 8087 emulation in the system software package.

The Microsoft FORTRAN Compiler is designed so that by using the default options you will have the fewest possible problems in getting your programs to work. This should be true whether you are writing new programs from scratch or porting existing programs from some other machine.

If you compile your programs with the default compiler options and link them with the runtime standard libraries, FORTRAN.LIB and MATH.LIB, they will run under both MS-DOS version 1.0 and version 2.0. If you have an 8087 installed in your machine, your programs will use it to improve the speed of real arithmetic. If you don't have an 8087 installed, your programs will run perfectly well and give the same results.

Additional benefits of the Microsoft FORTRAN Compiler are:

1. Support for linking of 8086 assembly language, Microsoft FORTRAN, and Microsoft Pascal programs.
2. Extensive program development support from MS-FORTRAN's metalanguage, memory management, and array building tools and runtime support from auxiliary floating-point libraries and the runtime's intrinsic function routines.

1.1 Options

Compiling and linking options are provided that give you the flexibility of customizing your own programs according to your requirements for portability and performance. See Chapter 4, "Options for Compiling and Linking," for a summary of these options.

1.2 Previous Versions

For a list of the differences between Microsoft FORTRAN 3.20 and previous versions since 3.0, see Appendix A, "Differences From Earlier Versions of Microsoft FORTRAN."

1.3 Descriptive Notation

The following descriptive devices are used throughout this manual to emphasize elements of the text. Descriptions of Microsoft FORTRAN syntax requirements for statements can be found in Chapter 3 of the *Microsoft FORTRAN Reference Manual.*

CAPS	Capitalized text indicates statements, files, or commands. The text is capitalized only to emphasize procedures, files, compilands, or objects that the user may encounter. Microsoft FORTRAN is not case sensitive. Small capital letters indicate that you must press a key named by the text.
Italics	Italics indicate user-supplied data, for example, filenames, variable names, and array names.
[ ]	Square brackets indicate that the enclosed entry is optional.
...	Ellipses indicate that an entry may be repeated as many times as needed or desired.

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

Pressing the RETURN (or ENTER) key is assumed at the end of every line you enter in response to a prompt. Only if this is the only response required is RETURN shown.

1.4 Learning More About FORTRAN

The manuals in this package provide complete reference information for your implementation of the Microsoft FORTRAN Compiler. They do not, however, teach you how to write programs in FORTRAN. If you are new to FORTRAN or need help in learning to program, we suggest you read any of the following books:

Agelhoff, R., and Richard Mojena. *Applied FORTRAN 77, Featuring Structured Programming.* Wadsworth, 1981.

Ashcroft, J., R. H. Eldridge, R. W. Paulson, and G. A. Wilson. *Programming With FORTRAN 77.* Granada, 1981.

Friedman, F., and E. Koffman. *Problem Solving and Structured Programming in FORTRAN.* Addison-Wesley, 2nd edition, 1981.

Wagener, J. L. *FORTRAN 77: Principles of Programming.* Wiley, 1980.

Chapter 2
Getting Started

2.1 Preliminary Procedures 9
2.1.1 Backing Up Your System Files 9
2.1.2 Setting Up Your System Disks 9
2.1.3 If You Have an 8087 Coprocessor 10
2.2 Program Development 11
2.3 Vocabulary 15

2.1 Preliminary Procedures

Before you begin the sample session in Chapter 2 or compile any programs of your own, we recommend that you review the following preliminary procedures for backing up your system disks, preparing your runtime library and setting up your system disks. If you are unfamiliar with any of the MS-DOS procedures mentioned, please consult your MS-DOS manual for instructions.

2.1.1 Backing Up Your System Files

This step is optional but highly recommended.

The first thing you should do when you have unwrapped your system disks is to make copies to work with, saving the original disks for backup. Make the copies using the COPY or DISKCOPY utilities supplied with MS-DOS.

2.1.2 Setting Up Your System Disks

This step is recommended.

Before you begin compiling and linking a program, we recommend that you check the contents of each disk. You may wish to copy some files from one system disk to another to set up a working arrangement that is convenient for you. You will certainly need to have available the linker utility, MS-LINK, from your MS-DOS package.

In order to avoid continual reprompting from the system to reload certain MS-DOS files, you may also wish to set up your system disks as shown in Table 2.1. Table 2.1 assumes that you have a hardware setup with two 160K disk drives.

Table 2.1

A Suggested Disk Setup

Disk	Contents
1	COMMAND.COM text editor text editor† miscellaneous utilities†† FOR1.EXE
2	COMMAND.COM PAS2.EXE PAS3.EXE
3	COMMAND.COM LINK.EXE FORTRAN.LIB <auxiliary libraries>†††

Notes for Table 2.1.
† Any text editor that fits.
†† MS-DOS utilities to set up printer, clear screen, sort directory, etc.
††† Other libraries you may read are: 8087.LIB, ALTMATH.-LIB, DECMATH.LIB, and DOS2FOR.LIB.

For most implementations, you can copy the necessary MS-DOS files by first formatting the blank disks with the /S switch and then copying the other files to the appropriate disks. If you do not format disks with the /S switch, the compiler will prompt you to reinsert your MS-DOS disk after each step.

2.1.3 If You Have an 8087 Coprocessor

Unless you have linked with ALTMATH.LIB, the MS-FORTRAN runtime library expects to encounter a particular arrangement of 8088,86-8087 hardware. Specifically, it expects that i8087 interrupts will be directed through to the 8088,86 via the 8088,86 interrupt vector 2 (NMI), without the intervention of an 8259 interrupt controller or its equivalent.

If these interventions do occur, you will need to determine if your hardware configuration meets any of the following criteria:

1. It uses an 8087 interrupt vector number other than 2.
2. It uses an 8259 interrupt controller.

3. The 8087 shares interrupts with another device on the same vector.

If any of these criteria is true for your computer system, you must read Appendix F, "Customizing i8087 Interrupts," and customize the runtime library as described there.

2.2 Program Development

This section provides a brief introduction to program development, a multistep process which includes first writing the program, and then compiling, linking, and running it. For a brief explanation of terms that may be unfamiliar, see Section 2.3, "Vocabulary."

A microprocessor can execute only its own machine instructions; it cannot execute source program statements directly. Therefore, before you run a program, some type of translation, from the statements in your program, to the machine language of your microprocessor, must occur.

Compilers and interpreters are two types of programs that perform this translation. Depending on the language you are using, either or both types of translation may be available to you. MS-FORTRAN is a compiled language.

A compiler translates a source program and creates a new file called an object file. The object file contains relocatable machine code that can be placed and run at different absolute locations in memory.

Compilation also associates memory addresses with variables and with the targets of GOTO statements, so that lists of variables or of labels do not have to be searched during execution of your program.

Many compilers, including the MS-FORTRAN Compiler, are what are called "optimizing" compilers. During optimization, the compiler reorders expressions and eliminates common subexpressions, either to increase speed of execution or to decrease program size. These factors combine to measurably increase the execution speed of your program.

The MS-FORTRAN Compiler has a three-part structure. The first two parts, pass one and pass two, carry out the translation and create the object code. Pass three is an optional step that creates an object code listing. Compiling is described in greater detail in Section 3.2, "Compiling Your Microsoft FORTRAN Program," and in Chapter 5, "More about Compiling."

Before a successfully compiled program can be executed, it must be linked. Linking is the process in which MS-LINK computes absolute offset addresses for routines and variables in relocatable object modules and then resolves all external references by searching the runtime library. The linker saves your program on disk as an executable file, ready to run.

You may, at link time, link more than one object module, as well as routines written in assembly language or other high-level languages, and routines in other libraries. Linking is described in greater detail in Section 3.3, "Linking Your Microsoft FORTRAN Program," and in Chapter 6, "More about Linking."

Figure 2.1 illustrates the entire program development process.

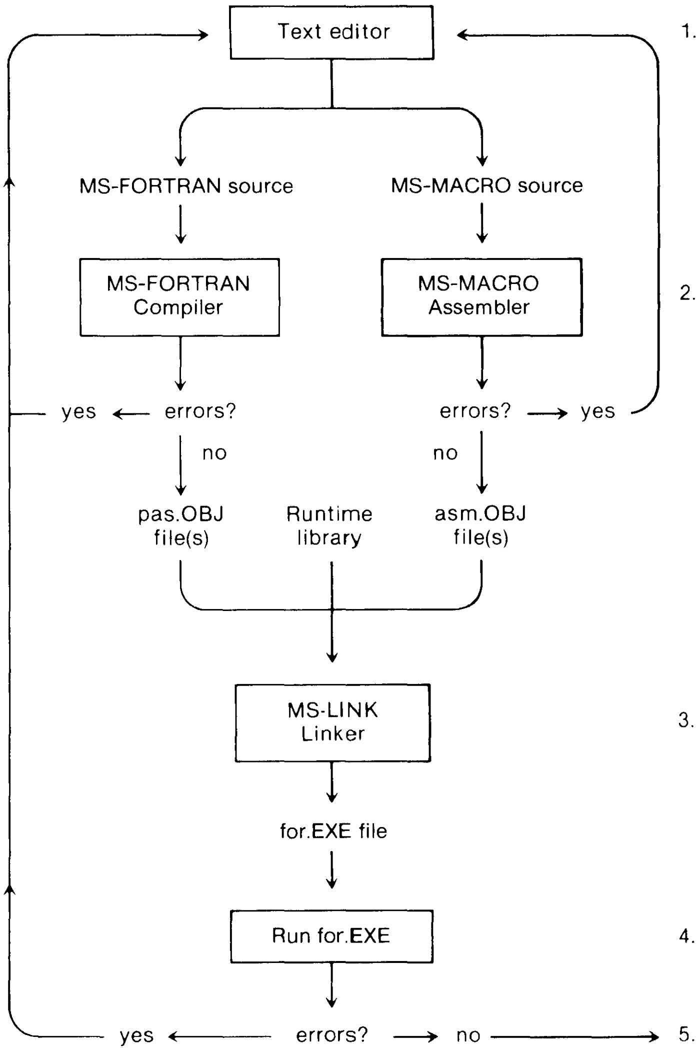


Figure 2.1. Program Development

1. Create and edit your MS-FORTRAN source file.

   Program development begins when you write an MS-FORTRAN program; any general purpose text editor will serve the purpose. Use a text editor also to write any assembly language routines you may plan to include.

2. Compile the program with $DEBUG. Assemble the assembler source, if any. When compilation is successful, remove the $DEBUG metacommand and recompile to enhance your program's execution time.

   Once you have written a program, compile it with the MS-FORTRAN Compiler. The compiler flags all grammatical errors as it reads your source file. Include the $DEBUG metacommand in your source file to generate diagnostic calls for runtime errors. If compilation is successful, the compiler creates a relocatable object file.

   If you have written your own assembly language routines (for example, to increase the speed of execution of a particular algorithm), assemble those routines with the Microsoft MACRO Assembler.

3. Link the compiled (and assembled) OBJ files with the runtime library.

   A compiled (or assembled) object file is not executable and must be linked with one of the runtime libraries, using MS-LINK. Separately compiled MS-Pascal subroutines and functions can also be linked to your program at this time.

4. Run the EXE file.

   The linker links all modules needed by your program and produces as output an executable run file with .EXE as the extension. This file can be executed by simply typing its filename.

5. Recompile, relink, and rerun with $NODEBUG.

   Repeat these processes until your program has successfully compiled, linked, and run without errors. Then recompile, relink, and rerun with $NODEBUG or remove $DEBUG from your source to reduce the amount of time and space required. Chapter 8, "Compiling and Linking Large Programs," discusses how to work within the various physical limits you may encounter in compiling, linking, and executing a program.

2.3 Vocabulary

This section reviews some of the vocabulary that is commonly used in discussing the steps in program development. The definitions given are intended primarily for use with this manual. Thus, neither the individual definition nor the list of terms is comprehensive.

An MS-FORTRAN program is more commonly called a "source program" or "source file." The source file is the input file to the compiler and must be in ASCII format. The compiler translates this source and creates, as output, a new file called a "relocatable object file." The source and object files generally have the default extensions .FOR and .OBJ, respectively. After compiling, the object file must be linked with the runtime library to produce an executable program or run file. The run file has the extension .EXE.

Some other terms you should know are related to stages in the development and execution of a compiled program. These stages are:

1. Compile time

   The time during which the compiler is executing and during which it compiles an MS-FORTRAN source file and creates a relocatable object file.

2. Link time

   The time during which the linker is executing and during which it links together relocatable object files and library files.

3. Runtime

   The time during which a compiled and linked program is executing. By convention, runtime refers to the execution time of your program and not to the execution time of the compiler or the linker.

The following terms pertain to the linking process and the runtime library:

1. Module

   A general term for a discrete unit of code. There are several types of modules, including relocatable and executable modules.

   The object files created by the compiler are said to be "relocatable," that is, they do not contain absolute addresses. Linking produces an "executable" module, that is, one that contains the necessary addresses to proceed with loading and running the program.

2. Routine

   Code, residing in a module, that represents a particular subroutine or function. More than one routine may reside in a module.

3. External reference

   A variable or routine in one given module that is referred to by a routine in another module. The variable or routine is often said to be "defined" in the module in which it resides.

   The linker tries to resolve external references by searching for the declaration of each such reference in other modules. If such a declaration is found, the module in which it resides is selected to be part of the executable module (if it is not already selected) and becomes part of your executable file. These other modules are usually library modules in the runtime libraries.

   If the variable or routine is found, the address associated with it is substituted for the reference in the first module, which is then said to be "bound." When a variable is not found, it is said to be "undefined" or "unresolved."

4. Relocatable module

   One whose code can be loaded and run at different locations in memory. Relocatable modules contain routines and variables represented as offsets relative to the start of the module. These routines and variables are said to be at "relative" offset addresses.

When the module is processed by the linker, an address is associated with the start of the module. The linker then computes an absolute offset address that is equal to the associated address plus the relative offset for each routine or variable. These new computed values become the absolute offset addresses that are used in the executable file. Compiled object files and library files are all relocatable modules.

These offset addresses are still relative to a "segment," which corresponds to an 8086 segment register. Segment addresses are not defined by the linker; rather, they are computed when your program is actually loaded prior to execution.

5. Runtime libraries

   Contain the runtime routines needed to implement the MS-FORTRAN language. A library module usually corresponds to a feature or subfeature of the MS-FORTRAN language.

Chapter 3
A Sample Session

3.1 Creating a Microsoft FORTRAN Source File 22

3.2 Compiling Your Microsoft FORTRAN Program 23

3.2.1 Pass One 23

3.2.2 Pass Two 26

3.2.3 Pass Three 27

3.3 Linking Your Microsoft FORTRAN Program 28

3.4 Executing Your Microsoft FORTRAN Program 30

This chapter provides step-by-step instructions for compiling and linking an MS-FORTRAN program.

Before proceeding with any of your own MS-FORTRAN programs, we strongly recommend that you work through the sample session that follows keeping in mind possible performance factors as you go. Section 2.1, "Preliminary Procedures," describes each step that should get your sample session off to a good start.

Creating an executable MS-FORTRAN program involves the following steps:

1. Write and save an MS-FORTRAN source file.
2. Compile your program with the MS-FORTRAN Compiler.
   a. Start pass one and enter your filenames in response to the prompts.
   b. Run pass two of the compiler.
   c. Run pass three of the compiler. (This step is optional.)

Note

For more information on starting the compiler and its command line options, see Chapter 5, "More About Compiling."

---

3. Link your object file to the MS-FORTRAN runtime library. For information about linking options (switches), see Chapter 6, "More About Linking."
4. Run your program.

Compiler passes one and two are required. You need to run pass three only if you require or want an object listing (as in this sample session).

The sample session assumes the following:

1. You have completed the necessary preliminary procedures.
2. You have two disk drives (A: and B:).

3. The sample program is already debugged, so that it will compile, link, and execute successfully.
4. An object listing is required, therefore all three passes of the compiler will be run.
5. No linker switches will be used.
6. There are no problems with data, code, or memory limits.

These complexities are discussed in Chapter 5, "More About Compiling," Chapter 6, "More About Linking," and Chapter 8, "Compiling and Linking Large Programs," and are referred to as appropriate in the following sample session.

If the files required for successive steps in the process are not on the same disk as one another, you will have to exchange disks between steps. For example, if FOR1.EXE and PAS2.EXE are not on the same disk, you will have to remove the first disk after completing pass one and replace it with the disk containing PAS2.EXE. Similarly, if the linker or the library file is on a different disk than pass three, you will have to insert a new system disk before running MS-LINK.

3.1 Creating a Microsoft FORTRAN Source File

Turn on your computer and load MS-DOS. Insert an empty work disk in drive B:. Log onto drive B:; this makes B: the default drive.

You can create MS-FORTRAN programs with any available text editor. The source file should, in most cases, have the .FOR extension.

Note

> We recommend that you use the $STORAGE:2 metacommand at the beginning of your new programs whenever possible. This will significantly improve the size and speed of your executable programs. Be advised, however, that this metacommand may cause problems in programs that are being ported from a mainframe processing environment and should be used with $DEBUG during the initial compilation. For more information on metacommands, see Chapter 6 in the *Microsoft FORTRAN Reference Manual.*

For this sample session, we will use the program named DEMO.-FOR, which came with the system software. Copy DEMO.FOR to drive B:, which is where it would be if it were your own program.

3.2 Compiling Your Microsoft FORTRAN Program

As mentioned previously, compiling a program is either a two- or a three-step process, depending on whether or not you choose to produce an object code listing. For this sample session, we will run all three passes.

3.2.1 Pass One

Insert the disk containing FOR1.EXE in drive A:. In response to the operating system prompt, type:

```
A:FOR1
```

This command starts pass one of the MS-FORTRAN Compiler.

(Actually, you may respond in either uppercase or lowercase characters. We use uppercase throughout this document simply for clarity.)

The compiler prints a header that includes the date and version number, then prompts you for four filenames:

1. your source filename
2. an object filename
3. a source listing filename
4. an object listing filename

Respond to the prompts as described in the following paragraphs. For additional information about the files themselves, see Chapter 5, "More about Compiling."

1. Source file prompt

   The first prompt is for the name of the file that contains your MS-FORTRAN program:

   ```
   Source filename [.FOR] :
   ```

   The prompt reminds you that .FOR is the default extension for the source filename. Unless the extension is something other than .FOR, you may omit it when you type in the filename.

   For now, type *DEMO* (to indicate that the source file is B:DEMO.FOR).

2. Object file prompt

   The second prompt is for the name of the relocatable object file, which will be created during pass two:

   ```
   Object filename [DEMO.OBJ] :
   ```

   The name in brackets is the name the compiler will give to the object file if you simply press the RETURN key at this point. The filename is taken from the source filename you gave in response to the first prompt; the .OBJ extension is the standard extension for object files.

   For now, either type *DEMO* or press the RETURN key.

3. Source listing file prompt

   The third prompt is for the name of the source listing file, which will be created during pass one:

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

As before, the prompt shows the default. Because the source listing is not required for linking and executing a program, it defaults to the null file; that is, if you press the RETURN key, no file is created at all. However, if you enter any part of a file specification, the default extension is .LST, the default device is the currently logged drive, and the filename defaults to the name given for the source file.

For this session, assume that you want the source listing written to a file called DEMO.LST; type *DEMO* in response to the source listing prompt.

4. Object listing file prompt

   The final prompt is for the object listing file, to be created during pass three:

   ```
   Object listing [NUL.COD] :
   ```

   The null file is the default for the object listing, as it is for the source listing. If you press the RETURN key, no intermediate files will be saved and you won't be able to run pass three. However, the same default naming rules apply here as elsewhere; if you enter any part of a file specification, the default extension is .COD, the default device is the currently logged drive, and the filename is the source filename.

   For now, type *DEMO* in response to the object listing prompt. When you run pass three, the object listing will be written to a disk file called DEMO.COD.

Compilation begins as soon as you have responded to all four prompts. The source listing is written to the file DEMO.LST, on the default drive B:, as requested. When pass one is complete, you should see the following message on your terminal screen:

```
Pass One     No Errors Detected.
```

If the compiler had detected errors during compilation, a message such as the following would appear instead:

```
Pass One     3 Errors Detected.
```

The error messages are also given in the source listing. Errors are mistakes that prevent a program from running correctly. See Appendix C, "Error Messages," in the *Microsoft FORTRAN Reference Manual* for a complete listing of the error messages you may encounter in MS-FORTRAN.

Pass one creates two intermediate files, PASIBF.SYM and PASIBF.BIN. The compiler saves both of these files on the default drive for use during pass two. If there are errors, these two files are deleted and pass two cannot be run.

3.2.2 Pass Two

Remove the disk containing FOR1.EXE from drive A: and insert the disk containing PAS2.EXE. You won't need to do this if PAS2.EXE is on the same disk as FOR1.EXE.

Start pass two by typing:

```
A:PAS2
```

Pass two does not ordinarily prompt you for any input. However, it does perform the following actions:

1. It reads the intermediate files PASIBF.SYM and PASIBF.BIN created in pass one.
2. It writes the object file.
3. It deletes the intermediate files created in pass one.
4. It writes two new intermediate files, PASIBF.TMP and PASIBF.OID, for use in pass three. These files are written to the currently logged drive.

When you are compiling your own programs, the last step described varies, depending on your response to the object listing prompt. If, as for this sample session, you plan to run pass three, pass two writes the two intermediate files. If in pass one you do not request an object listing, pass two writes and later deletes just one new intermediate file, PASIBF.TMP.

When pass two is complete, a message such as the following prints on your screen:

```
Code Area Size = #05EC   ( 1516)
Cons Area Size = #00E6   (  230)
Data Area Size = #0264   (  612)

Pass Two     No Errors Detected.
```

The first three lines indicate, first in hexadecimal and then in decimal notation, the amount of space taken up by executable code (Code), constants (Cons), and variables (Data). The message concerning the number of errors refers to pass two only, not to the entire compilation.

3.2.3 Pass Three

Remove the disk containing PAS2.EXE from drive A: and insert the disk containing PAS3.EXE. You won't need to do this if PAS3.EXE is on the same disk as PAS2.EXE.

Start pass three by typing:

```
A:PAS3
```

PAS3.EXE does not prompt you for any input. It reads PASIBF.TMP and PASIBF.OID, the temporary files created during pass two, and, because of your earlier response to the object listing prompt, writes the object code listing to the file DEMO.COD.

When pass three is complete, the two temporary files are deleted. If, after requesting an object listing, you choose not to run pass three, you should delete these files yourself (to save space).

Table 3.1 is a summary of the files read and written by each of the three passes of the compiler during this sample session.

Table 3.1

Files Used by the MS-FORTRAN Compiler

Pass	Reads	Writes	Deletes
1	DEMO.FOR	DEMO.LST PASIBF.SYM PASIBF.BIN	
2	PASIBF.SYM PASIBF.BIN	DEMO.OBJ PASIBF.OID PASIBF.TMP	PASIBF.SYM PASIBF.BIN
3	PASIBF.OID PASIBF.TMP	DEMO.COD	PASIBF.OID PASIBF.TMP

See Chapter 5, "More About Compiling," for more detailed information about filename conventions and responding to the compiler prompts.

3.3 Linking Your Microsoft FORTRAN Program

Now you are ready to link your program with one of the two versions of MS-LINK provided with Microsoft FORTRAN version 3.20 (see Section 4.2, "Alternative Linkers," for a description of these linkers). Linking converts the relocatable object file into an executable program by assigning absolute addresses and setting up calls to the runtime library.

Remove the disk containing PAS3.EXE from drive A: and insert the disk containing LINK.EXE. This step won't be necessary if the linker is on the same disk as PAS3.EXE.

Start the linker by typing:

```
A:LINK
```

The linker displays a header and then, like pass one of the compiler, gives a series of four prompts, to which you must respond before linking begins. The linker prompts you for the following information:

1. the name of your relocatable object file(s)
2. the name you wish to give to the executable program
3. the name you wish to give to the linker listing
4. the name of the runtime library or libraries

Each of these prompts is discussed briefly in the following paragraphs and in somewhat greater detail in Chapter 6, "More About Linking." For complete information on MS-LINK, see your MS-DOS manual.

After the first of the four linker prompts appears on the screen, remove the disk containing LINK.EXE and insert the disk containing FORTRAN.LIB and MATH.LIB. You won't need to do this if the linker and the runtime libraries are on the same disk.

1. Object modules prompt

   The first prompt is for the name of your relocatable object file (or files):

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

   Like the compiler prompts, the linker prompts always give certain defaults. This prompt indicates that .OBJ is the default extension for any file(s) you name here. Type *DEMO*, and the file DEMO.OBJ, created during compilation, will be linked with FORTRAN.LIB and MATH.LIB during the linking process. If, for any reason, an object file does not have the extension .OBJ, you must give the file specification in full.

2. Run file prompt

   The second prompt is for the name of the run file, the file created by the linker that will contain your executable program:

   ```
   Run File [DEMO.EXE] :
   ```

   The default filename is taken from your response to the first linker prompt; the .EXE extension identifies an executable file. To accept the default filename, simply press the RETURN key.

3. Linker listing file prompt

   The third prompt is for the linker listing file, sometimes called the linker map:

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

   As the prompt indicates, the default for the list file is the NUL file, that is, no file at all. For now, simply press the RETURN key to accept this default.

   If, when linking your own programs, you wish to see the list file on your console, without having it written to a disk file, type *CON* in response to the list file prompt. If you want the linker map written to a disk file, respond to this prompt with a name for the file.

4. Runtime library prompt

   The last linker prompt is for the location of the runtime library:

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

Here, to indicate that FORTRAN.LIB and MATH.LIB are on drive A:, you should type:

```
A:
```

If the libraries are not already on the disk in drive A:, you will have to exchange disks before linking can proceed.

After you have responded to the last of the four prompts, the linker links your compiled program, DEMO.OBJ, with the necessary modules in the MS-FORTRAN runtime library, A:FORTRAN.-LIB. This linking process creates an executable file, named DEMO.EXE, on the default drive.

3.4 Executing Your Microsoft FORTRAN Program

When linking is complete, the operating system prompt returns. To run the sample program, just type:

```
DEMO
```

This command directs MS-DOS to load the executable file DEMO.EXE, fix segment addresses to their absolute value (based on the address at which the file is loaded), and start execution.

Assuming the program runs correctly, which it should, you will be prompted to enter ten numbers, which will then be sorted and displayed on your screen in sorted order, from lowest to highest.

This concludes the sample session. Additional information on compiling and linking is provided in Chapter 5, "More About Compiling," and Chapter 6, "More About Linking," respectively. The following program shows a log of the entire sample session, including prompts, your responses (shown in italics), and comments on files written to disk:

```
A> B:
B> A:FOR1
Source file [.FOR]:DEMO
Object file [DEMO.OBJ]:RETURN
Source listing [NUL.LST]:DEMO
Object listing [NUL.COD]:DEMO

    [Source listing written as DEMO.LST]

    Pass One      No Errors Detected.

B> A:PAS2

    Code Area Size =  05EC   (  1516)
    Cons Area Size =  00E6   (   230)
    Data  Area Size = 0264   (   612)

    Pass Two      No Errors Detected.

B> A:PAS3

    [Object listing written as DEMO.COD]

B> A:LINK
Object modules [.OBJ]:DEMO
Run file [DEMO.EXE]:RETURN
List map [NUL.MAP]:RETURN
Libraries [.LIB]:A:

B> DEMO

    [Program prompting and display]
```

Chapter 4
Options for Compiling and Linking

4.1 MS-DOS 2.0 Interface Library 35

4.2 Alternative Linkers 35

4.3 16-Bit Integer Variables 36

4.4 Floating-Point Options 37

4.5 Changing the Default Math Library 39

4.6 Best Cases for Compilation and Execution 39

This chapter contains descriptions of optional libraries and compiler features that are available to the users of Microsoft FORTRAN for customizing the performance of executable programs. We recommend, however, that you start by using the compiler with its defaults, particularly if you are inexperienced with FORTRAN.

4.1 MS-DOS 2.0 Interface Library

When you specify DOS2FOR.LIB at linktime, it will automatically replace the standard file system in the runtime library, FORTRAN.LIB, with the MS-DOS 2.0 file system. If you are specifying FORTRAN.LIB explicitly you must specify DOS2-FOR.LIB *before* FORTRAN.LIB in the list of libraries to be searched by the linker.

The modules contained in DOS2FOR.LIB provide the interface described in Section 10.2, "An Overview of the File System," of this *User's Guide.*

Note

Programs linked with DOS2FOR.LIB will not run under MS-DOS 1.25. An error message, "Incorrect DOS version", will be returned by the program.

4.2 Alternative Linkers

Two versions of the Microsoft LINK utility are provided with this version of Microsoft FORTRAN. The first, named LINK.EXE, is the most current linker for MS-DOS versions 1.25 and earlier. It will run under MS-DOS 2.0 but cannot accept pathnames or subdirectories. The other version is named LINK.V2. It accepts pathnames, will only run on MS-DOS 2.0, and includes an Overlay option.

You must use either LINK.EXE or LINK.V2 to link your program because earlier versions of the MS-DOS linker lack some of the internal features necessary for support of this version of Microsoft FORTRAN.

LINK.EXE and LINK.V2 search libraries based on the contents of a library list. This list is derived from your command line specifications and the search directives produced by the compiler. If, after all the libraries have been searched, at least one reference has been resolved, the linkers will repeat the search and attempt to resolve the other references. Previous versions of the linker searched each library only once.

Rename LINK.V2 to be an .EXE file if you want to use it at linktime. See Section 6.1.2, "Linking Libraries," for command-line and prompting information.

Note

If you use the earlier versions of Microsoft LINK, an otherwise correct program may produce linker error messages and not execute properly.

For more information about LINK.V2, see Section 6.3, "The Overlay Linker."

4.3 16-Bit Integer Variables

If you know that none of the values assigned to INTEGER variables in your program will exceed the precision of a 16-bit integer, and that your program does not rely on REAL, LOGICAL and INTEGER variables being allocated the same amount of memory (as specified by the FORTRAN 77 Standard), you can use the $STORAGE:2 metacommand to specify that your INTEGER and LOGICAL variables will be mapped into two bytes, instead of the default four bytes. This will make your programs run more quickly since the Intel® 8086 processes 16-bit arithmetic much faster than 32-bit arithmetic.

You shouldn't have any problem in using $STORAGE:2 on programs you write for yourself or which you have migrated from other microprocessor FORTRANs that have 16-bit INTEGERs. Be careful, though, when using it with programs originally written for mainframe FORTRANs that have bigger integer variables. Values that were perfectly valid in the bigger integers may exceed the range of a 16-bit integer. Unless you compile with the $DEBUG option, the values may wrap around and your program will give wrong answers. To be safe, always use $DEBUG until you are sure your program is working properly.

Comments on the Integer Data Type

1. The range of values for both 16-bit and 32-bit integers does not include the most negative number that can be represented in 2's complement arithmetic in that number of bits. These numbers, 16#8000 and 16#80000000, are treated as "undefined" for error checking purposes.
2. Although the maximum 32-bit integer value is defined as 2**31-1, the compiler and runtime will read greater values which are nominally in the range up to 2**32. But these values will only be read without error if the radix is other than 10. They will be interpreted as the negative numbers with the corresponding internal representation. For example, 16#FFFFFFFF will result in all the bits in the 32-bit integer result being set, and will have an arithmetic value of -1.

4.4 Floating-Point Options

You can use metacommands and alternative libraries to change the way floating-point operations are carried out. (For more details, see Section 10.4, "Floating-Point Operations.") The options are:

8087.LIB / $FLOATCALLS

If you know that all machines on which you will be running your program will have an 8087 installed, you can use 8087.LIB to reduce its size. You can reduce its size still further and improve its performance by compiling with the $NOFLOATCALLS metacommand.

Alternate Math Option

If performance on machines without 8087s installed is an overriding concern, and you do not care if your program does not exploit an 8087 if it is installed, and if you do not require the full power of the proposed IEEE floating-point standard, you can use the fast math package by linking with ALTMATH.LIB.

Decimal Math Option

Microsoft FORTRAN supports an alternative floating-point format in which decimal floating-point numbers up to 14 digits and within a limited exponent range can be represented exactly. The results of the operations on the numbers in this format are also represented exactly if they are in the allowable range. This option is particularly useful in business and financial applications where exact results are important.

You select the decimal format by using the $DECMATH metacommand in all of your program units that use floating-point. You must link with DECMATH.LIB to support this format. Decimal floating-point and IEEE floating-point *are not* compatible.

Note

$FLOATCALLS and $NOFLOATCALLS will be ignored if you have specified $DECMATH.

4.5 Changing the Default Math Library

The default math library is contained in MATH.LIB. You can make either DECMATH.LIB, 8087.LIB, or ALTMATH.LIB the default by naming the one you want to be MATH.LIB.

4.6 Best Cases for Compilation and Execution

The Microsoft FORTRAN Compiler can create several versions of your executable program. Here are some "best case" combinations of Microsoft FORTRAN options for particular processor configurations.

Fastest (with 8087)

To get the best possible performance if you have an 8087, use the $NOFLOATCALLS and $STORAGE:2 metacommands, and link with 8087.LIB. This will also be the smallest version of your program.

Fastest (without 8087)

To get the best possible performance without an 8087, use $STORAGE:2 and link with ALTMATH.LIB.

Most portable, most consistent

If you want your program to run on any environment and give the most accurate results possible, use the default compiler and library options. You can also compile using the $NOFLOATCALLS metacommand, which will reduce the size of your program without affecting the arithmetic results.

Chapter 5
More About Compiling

5.1 Files Written by the Compiler 43

5.1.1 The Object File 43

5.1.2 The Source Listing File 43

5.1.3 The Object Listing File 44

5.1.4 The Intermediate Files 44

5.2 Filename Conventions 45

5.3 Starting the Compiler 49

5.3.1 Giving No Parameters on the Command Line 49

5.3.2 Giving All Parameters on the Command Line 50

5.3.3 Giving Some Parameters on the Command Line 51

This chapter provides procedural information on the compiler, supplementing the discussion in Section 3.2, "Compiling Your Microsoft FORTRAN Program." For a technical discussion of the compiler, see Section 10.1, "The Structure of the Compiler."

5.1 Files Written by the Compiler

In addition to creating several intermediate files, which it later reads and deletes, the compiler writes one required file and two optional files that represent your program in various ways. The object file is the one permanent file that must be created. The source listing and object listing files are optional; you may request that either or both of these be displayed or printed instead of being written to a disk file.

5.1.1 The Object File

The object file is written to disk after the completion of pass two of the compiler. It is a relocatable module, which contains relative rather than absolute addresses. Normally created with the .OBJ extension, the object module must be linked with the MS-FORTRAN runtime library to create an executable module containing absolute addresses.

5.1.2 The Source Listing File

The source listing file is a line-by-line account of the source file(s), with page headings and messages. Each line is preceded by a number that is referred to by any error messages that pertain to that source line.

Compiler error messages, shown in the source listing, are also displayed on your terminal screen. See Appendix C, "Error Messages," in the *Microsoft FORTRAN Reference Manual* for a complete list of MS-FORTRAN error messages.

If you include files in the compilation with the $INCLUDE metacommand, these files are also shown in the source listing. (For information on the $INCLUDE metacommand, see the entry for $INCLUDE in Section 6.2, "Metacommand Directory," in the *Microsoft FORTRAN Reference Manual*.)

The various flags, level numbers, error message indicators, and symbol tables in the source listing make it useful for error checking and debugging. Many programmers prefer a printout of the source listing file rather than of the source file itself as a working copy of the program.

5.1.3 The Object Listing File

The object listing file, a symbolic, assembler-like listing of the object code, lists addresses relative to the start of the program or module. Absolute addresses are not determined until the object file itself is linked with the runtime library.

The object listing file is used less often than the source listing file, but may be a useful tool during program development:

1. You can look at it simply to see what code the compiler generates and to familiarize yourself with it.
2. You can check to see whether a different construct or assembly language would improve program efficiency.
3. You use it as a guide when debugging your program with the MS-DOS DEBUG utility.

5.1.4 The Intermediate Files

Pass one creates two intermediate files, PASIBF.SYM and PASIBF.BIN, which incorporate information from your source file for use in creating the object file during pass two. These two intermediate files are always written to the default drive.

Pass two reads and then deletes PASIBF.SYM and PASIBF.BIN. Pass two itself creates one or two new intermediate files, depending on whether or not you've requested an object listing. If, as for the sample session, you plan to run pass three to produce the object listing, pass two writes the two intermediate files, PASIBF.TMP and PASIBF.OID.

If in pass one you do not request an object listing, pass two writes and later deletes just one new intermediate file, PASIBF.TMP.

PAS2.EXE assumes that the intermediate files created in pass one are on the default drive. If you have switched disks so that they are on another drive, you must indicate their location on the command that starts pass two. For example:

```
A:PAS2 A/PAUSE
```

The *A* immediately following the command tells the compiler that PASIBF.BIN and PASIBF.SYM are on drive A, instead of the default drive B:. The */PAUSE* tells the compiler to pause before continuing so that you can insert the disk that contains them into drive A:.

After pausing, pass two prompts as follows:

```
Press enter key to begin pass two.
```

When you have inserted the new disk in drive A:, press the RETURN key and the compiler proceeds with pass two.

PASIBF.TMP and PASIBF.OID are deleted from the default drive during pass three. If you change your mind after requesting an object listing file and decide not to run pass three, be sure to delete these files to recover the space on your disk.

5.2 Filename Conventions

When you start up the compiler, it prompts you for the names of four files: your source file, the object file, the source listing file, and the object listing file. The only one of these names you must supply is the source filename.

This section describes how the compiler constructs the remaining filenames from the source filename and how you can override these defaults.

A complete filename specification under MS-DOS has three parts:

1. Device name

   The name of the disk drive where the file is or will be. On a single-drive machine, all device names default to A:. On multidrive machines, if you do not specify a device, the compiler assumes the currently logged drive.

2. Filename

   The name you give to a file. Consult your operating system manual for any limitations on assigning filenames. Note that "line" and "user" are reserved by FORTRAN for the console and auxiliary port, respectively (the extension is ignored for these names).

3. Filename extension

   Added to the filename for further identification of the file. The extension consists of up to three alphanumeric characters and must be preceded by a period. Although you may give any extension to a filename, the MS-FORTRAN Compiler and MS-LINK recognize and assign certain extensions by default, as shown in the list below.

Extension	Function of File
.FOR	MS-FORTRAN source file
.PAS	MS-Pascal source file
.OBJ	Relocatable object file
.LST	Source listing file
.COD	Object listing file
.ASM	Assembler source file
.MAP	Linker map file
.LIB	Library file
.EXE	Executable run file

If you give unique extensions to your filenames, you must include the extension as part of the filename in response to a prompt. If you do not specify an extension, the MS-FORTRAN Compiler supplies one of those shown in Table 5.1.

Table 5.1

Default File Specifications

File	Device	Extension	Full File Spec
Source file	dev:	.FOR	dev:filename.FOR
Object file	dev:	.OBJ	dev:filename.OBJ
Source listing	dev:	.LST	dev:NUL.LST
Object listing	dev:	.COD	dev:NUL.COD

Table 5.1 also shows the default file specifications supplied by the compiler if you give a name for the source file and then press the RETURN key in response to each of the remaining compiler prompts.

The device "dev:" is the currently logged drive. Even if you specify a device with the source filename, the remaining file specifications will default to the currently logged drive. You must explicitly specify the name of another drive if that is where you want a particular file to go.

The NUL file is equivalent to creating no file at all; thus, by default, the compiler creates neither a source listing file nor an object listing file. If, in response to either of the last two prompts, you enter any part of a file specification, the remaining parts default as follows:

```
Source listing    dev:filename.LST
Object listing    dev:filename.COD
```

Neither listing file is created unless you explicitly request it. If you specify any non-null file for the object listing, pass two leaves PASIBF.TMP and PASIBF.OID, the input files for pass three, on your work disk until you delete them, either explicitly or by running pass three.

The general rules for filenames may be summarized as follows:

1. All lowercase letters in filenames are changed into uppercase letters. For example, the following three names are all considered equivalent to ABCDE.FGH:

   ```
   abcde.fgh    AbCdE.FgH    ABCDE.fgh
   ```

2. To enter a filename that has no extension in response to a prompt, type the name followed by a period.

   For example, typing *ABC* in response to the source filename prompt gives a filename of ABC.FOR; typing *ABC.* instructs the compiler to accept ABC with no extension as the name.

3. You may override any defaults by typing all or part of the name instead of pressing the RETURN key. For example, if the currently logged drive is B: and you want the object file to be written to the disk in drive A:, type *A:* in response to the following prompt:

   ```
   Object Filename [ABC.OBJ] :
   ```

   This results in a full filename of A:ABC.OBJ for the object file.

4. Listing files default to null. However, if you specify any part of a legal filename, the default changes so that the compiler creates a filename with the same default rules that apply to the source and object files. Specifically, if you give a drive or extension, then the base name is the base name of the source file. For example, typing *B:* in response to the object listing prompt gives a filename of B:ABC.COD.

5. Typing a semicolon after the source filename or in response to any of the later prompts tells the compiler to assign the default filenames to all the remaining files. This is the quickest way to start the compiler if you don't need either of the listing files.

   For example, typing *ABC;* in response to the source file prompt eliminates the remaining prompts and results in the following filenames:

Source file	B:ABC.FOR
Object file	B:ABC.OBJ
Source listing	B:NUL.LST
Object listing	B:NUL.COD

   You may not enter a semicolon to specify a source file, since the source file has no default filename.

6. Leading and trailing spaces are permitted, so the following is an acceptable response to the source file prompt:

   ```
   ABC  ;
   ```

   The filename itself must not contain spaces.

7. To send either listing file to your screen (console), use one of the special filenames USER or CON. USER is recognized only by MS-FORTRAN (and MS-Pascal) and writes to the screen immediately as the listing is created. CON is recognized by all MS-DOS programs, but saves the console output and writes it in blocks of 512 bytes.

5.3 Starting the Compiler

You can start the MS-FORTRAN Compiler in one of three ways:

1. You can let the compiler prompt you for each of the four filenames (as in the sample session).
2. You can give all four filenames on the command line.
3. You can give some of the filenames on the command line and let the compiler prompt you for the rest.

Each of these methods is discussed in the following sections. The second method, giving all four filenames on the command line, is particularly useful when you plan to use a batch command file. See Chapter 7, "Using a Batch Command File," for information.

5.3.1 Giving No Parameters on the Command Line

To start the compiler without giving any of the necessary parameters (filenames) on the command line, simply type the following:

```
A:FOR1
```

As in the sample session, the compiler prompts you for each of the four filenames it needs. A typical session might look like this (your responses are shown in italics):

Source filename [.FOR] : *MYFILE*
Object filename [MYFILE.OBJ] : RETURN
Source listing [NUL.LST] : *MYFILE*
Object listing [NUL.COD] : RETURN

This sequence of responses would give you an object file called B:MYFILE.OBJ, a source listing file called B:MYFILE.LST, and no object listing file.

Note

Pressing the RETURN key means that you accept the default shown in brackets; giving any part of a file specification creates a file with the same default rules that apply to other files.

5.3.2 Giving All Parameters on the Command Line

Instead of letting the compiler prompt you for each of the four filenames in turn, you may implicitly or explicitly give all four names on the same command line with which you start the compiler. This eliminates prompting for the filenames and is particularly useful when you are using the MS-DOS batch file facility. See Chapter 7, "Using a Batch Command File," for information on creating a batch command file for use with the compiler.

The general form of the command line that includes all of the compiler parameters is as follows:

A:FOR1 *source,object,sourcelist,objectlist;*

The same default naming conventions apply here as when you are prompted for the filenames.

You must separate each filename with a comma; spaces are optional. Put a semicolon at the end of the line to indicate that you do not want additional prompting.

If you omit a filename after a comma, the file by default is given the same filename as the source, the default device designation, and the default extension. Thus, these two command lines are equivalent:

```
A : FOR1 DATABASE,DATABASE,DATABASE,DATABASE;
A : FOR1 DATABASE, , , ;
```

Both result in the following four filenames being assigned:

Source file	B:DATABASE.FOR
Object file	B:DATABASE.OBJ
Source listing	B:DATABASE.LST
Object listing	B:DATABASE.COD

If you want the normal defaults, with NUL listing files, use the semicolon (;) following the source filename. Thus, these command lines are equivalent:

```
A : FOR1 YOYO,YOYO,NUL,NUL;
A : FOR1 YOYO;
```

You may include spaces before or after filenames, but not within them.

5.3.3 Giving Some Parameters on the Command Line

You may also start the compiler by giving one or more of the required filenames on the command line and letting the compiler prompt you for the rest.

For example, if you give only the names of the source file and the object file on the command line, the compiler will prompt you for the names of the source listing and the object listing (your responses are shown in italics):

B : *A : FOR1 TEST,TEST*
Source listing [NUL.COD] : *TEST*
Object listing [NUL.COD] : RETURN

This sequence of responses results in the following filenames:

Source file	B:TEST.FOR
Object file	B:TEST.OBJ
Source listing	B:TEST.LST
Object listing	B:NUL.COD

Chapter 6
More About Linking

6.1 Files Read by the Linker 55
6.1.1 Object Modules 55
6.1.1.1 Standard Runtime Libraries 57
6.1.1.2 Auxiliary Libraries 58
6.1.1.3 Linking Libraries 59
6.2 Files Written by the Linker 60
6.2.1 The Run File 61
6.2.2 The Linker Listing File 61
6.2.3 VM.TMP 62
6.3 The Overlay Linker 62
6.3.1 Restrictions 63
6.3.2 Overlay Manager Prompts 64
6.4 Linker Switches 64

This chapter provides an overview of what you will see on your screen when you start LINK.EXE, the default version of Microsoft LINK. Included in this overview is a description of the standard runtime libraries and the auxiliary libraries provided with this version of the Microsoft FORTRAN Compiler. Also included is a discussion of the optional version of Microsoft LINK, LINK.V2, which accepts pathnames and overlays.

6.1 Files Read by the Linker

A successful MS-FORTRAN compilation produces a relocatable object file. Linking, the next step in program development, is the process of converting one or more relocatable object files into an executable program.

6.1.1 Object Modules

Object files can come from any of the following sources:

1. MS-FORTRAN compilands (programs, subroutines, or functions)
2. MS-Pascal compilands (programs, modules, or units)
3. User code in other high-level languages
4. Assembly language routines
5. Routines in standard runtime modules that support facilities such as error handling, heap variable allocation, or input/output

Interfacing to MS-Pascal or other high-level language routines is quite straightforward. All procedures that are referenced in an MS-FORTRAN routine and that are not defined in the same program unit are automatically considered to be external. No additional EXTERNAL declarations are required. For information on how to specify in another language that a routine is public, see the appropriate reference and user manuals for that language.

Calling conventions and function returns between MS-FORTRAN and other languages may differ. (See Chapter 9, "Using Assembly Language Routines," for MS-FORTRAN calling conventions and

interface requirements.) You may need to write assembly language interface routines to interface between MS-FORTRAN and other languages. Whatever the language, it must be able to produce linkable object modules.

For further information on MS-LINK, see the appropriate chapter in your MS-DOS manual.

The ability to link together programs and subroutines of MS-FORTRAN source code, as well as assembly language and library routines, allows you to develop a program incrementally. Separate compilation and later linking of separate parts of a program not only reduces the need for continual recompilation, it also allows you to create programs that contain more than 64K bytes of code. (See Chapter 8, "Compiling and Linking Large Programs.") Separate compilation may increase the total size of your object module and run file, but will have no effect on the size of your executable program.

For now, assume that you have created a program that uses one MS-FORTRAN main program and one subroutine and also contains two assembly language external procedures. Assume further that these files have already been compiled or, in the case of the assembly language routines, already assembled and that the files thus created are the following:

PROG.OBJ
SUBR.OBJ
ASM1.OBJ
ASM2.OBJ

To link these all together, first invoke the linker by typing the following:

```
A:LINK
```

Like the compiler, the linker gives a sequence of four prompts. Before linking can proceed, you must explicitly or implicitly supply the following pieces of information:

1. the name(s) of the object modules to be linked
2. the name to be given to the executable run file
3. the linker listing file

4. the names of any libraries to be searched (other than FORTRAN.LIB)

As with the compiler, responses to all except the first prompt may be supplied by defaults.

In response to the first linker prompt, enter the names of the object files, separated by plus signs as shown:

```
PROG+SUBR+ASM1+ASM2
```

The first object file listed must be an MS-FORTRAN object file, although it need not be the main program. Do not put any assembly language module first; doing so may result in segments being ordered incorrectly. After the initial MS-FORTRAN object file, you may list the other subroutines or assembly language routines in any order.

Note

Typing a semicolon (;), at any point in the prompting session *after* you have specified the object files that you wish to link, tells the linker to omit the remaining prompts and to supply defaults for all remaining parameters (see Table 6.1).

Table 6.1

Linker Defaults

Prompt	Default Response
Object modules	None
Run file	prog.EXE
List map	NUL.MAP
Libraries	FORTRAN.LIB

6.1.1.1 Standard Runtime Libraries

A runtime library contains runtime routines that are required during linking to resolve references made during compilation (see Section 10.3.1, "Runtime Routines," for a complete list.)

MS-FORTRAN causes the linker to search for the runtime libraries FORTRAN.LIB and MATH.LIB, which are supplied with the compiler and contain the standard runtime modules for FORTRAN. MATH.LIB contains the default floating-point math package (see Chapter 10, "Advanced Topics," for more details about these elements of MATH.LIB.)

If you don't use any real numbers in your programs, MATH.LIB is not required at linktime. But if you have unresolved references when you link, the linker will request MATH.LIB to satisfy them. You can also change the default math package by renaming MATH.LIB and naming the library of your choice to be MATH.LIB.

6.1.1.2 Auxiliary Libraries

You may wish the linker to search additional libraries at link time. The auxiliary libraries supplied with MS-FORTRAN are:

1. ALTMATH.LIB, which contains a high speed software floating-point package
2. 8087.LIB, which contains "stubs" for the floating-point package
3. DECMATH.LIB, which contains decimal floating-point support routines
4. DOS2FOR.LIB, which contains the interface to the MS-DOS 2.0 file system.

Note

The auxiliary math libraries completely replace MATH.LIB and can be specified before or after an explicit reference to FORTRAN.LIB. If you specify them first, the automatic search for MATH.LIB will be suppressed. You must not specify more than one math library explicitly.

DOS2FOR.LIB replaces the equivalent part of FORTRAN.LIB and must be searched *before* FORTRAN.LIB. This will occur automatically unless you are specifying FORTRAN.LIB explicitly. In this case, DOS2FOR.LIB must come before FORTRAN.LIB in the list of libraries supplied to the linker.

6.1.1.3 Linking Libraries

To produce a library search using the LINK.EXE prompts, you specify the desired library at the "Libraries" prompt. For example, if you wanted FORTRAN.LIB to be searched, you would enter *fortran.lib* in the following sequence of prompts:

```
Object Modules [.OBJ] : your modules
Run File your program : RETURN
List File [NUL.MAP] : RETURN
Libraries [.LIB] : fortran.lib
```

On the command line, it would appear as shown here:

```
A>LINK your modules , , , fortran.lib
```

If you press the RETURN key in response to the final linker prompt, the linker will automatically search for FORTRAN.LIB on the default drive. If FORTRAN.LIB is not on the default drive, the following message will appear on your screen:

```
Cannot find library FORTRAN.LIB
Enter new drive spec :
```

Switch disks if necessary, and then type the name of the library that you wish to be searched. If instead you want linking to proceed without a library search, respond by pressing the RETURN key.

You can achieve the same effect by using the linker option switch, /NODEFAULTLIBRARYSEARCH, to override the automatic search for FORTRAN.LIB and MATH.LIB. This will produce unresolved reference error messages unless you replace every required runtime routine with a routine of your own. (Most MS-FORTRAN programmers never require this capability.)

If you are using LINK.EXE, you may specify just the drive, or just the library filename, or both the drive and the filename. If you are using LINK.V2, you may specify the library filename in a path

(drive:\pathname\filename and extension). For example, if you want the standard runtime library, \FORTRAN\FORTRAN.-LIB, you respond,

```
A>LINK your modules , , , \fortran\
```

The linker will look for FORTRAN.LIB in the FORTRAN directory on drive:A. If you respond,

```
A>LINK your modules , , , \fortran\foo
```

the linker will look for \FORTRAN\FOO.LIB.

Note

You cannot specify a pathname with the default linker, LINK.EXE.

To instruct the linker to search other libraries (for example, PASCAL.LIB, as well as FORTRAN.LIB) give the library names, separated by plus signs, in response to the final linker prompt,

```
Libraries [.LIB] : fortran.lib+pascal.lib+stat.lib
```

See your MS-DOS *User's Guide* for complete information on using different libraries with Microsoft LINK.

6.2 Files Written by the Linker

The primary output of the linking process is an executable run file. You may also request a linker map or listing file, which serves much the same purpose as the compiler listing files. The linker, if need be, also writes and later deletes one temporary file.

6.2.1 The Run File

The run file produced by the linker is your executable program.

The default filename, given in brackets as part of the prompt, is taken from the name of the first module listed in response to the first prompt. To accept this prompt, press the RETURN key. To specify another run filename, type in the name you want. All run files receive the extension .EXE, even if you specify something else.

The linker ordinarily saves the run file, with the extension .EXE, on the disk in the default drive. To specify another drive, which may be necessary if your program is large, type a drive name in response to the run file prompt.

6.2.2 The Linker Listing File

The linker map, also called the linker listing file, shows the addresses, relative to the start of the run module, for every code or data segment in your program. If you request it with the /MAP switch, the linker map can also include all EXTERN and PUBLIC variables. (See Section 6.4, "Linker Switches," for information on the /MAP switch).

The linker map defaults to the NUL file, unless you specifically request that it be printed, displayed on the screen, or saved on disk. In the early stages of program development, you may find it useful to inspect the linker map in these two instances:

1. When using the debugger to set breakpoints and locate routines and variables
2. To find out why a load module is so large (for example, what routines are loaded, how big they are, and what's in them)

As the prompt indicates, the default for the linker map is the NUL file, that is, no file at all. Press the RETURN key to accept this default. If you wish to see the linker map but not have it written to a disk file, type *CON* in response to the list file prompt. If you want the file written to disk, give a device or filename.

6.2.3 VM.TMP

Linking begins after you have responded to all of the linker prompts. If the linker needs more memory space to link your program than is available, it will create a file called VM.TMP on the disk in the default drive and will display a message like the following:

```
VM.TMP has been created.
Do not change disk in drive B:.
```

If the additional space is used up or if you remove the disk that contains VM.TMP before linking is complete, the linker will abort.

When the linker has finished, VM.TMP will be erased from the disk, and any errors that occurred during linking will be displayed. (For a list of MS-LINK error messages, see Appendix H, "Microsoft LINK Error Messages.")

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

6.3 The Overlay Linker

You can direct the MS-DOS 2.0 version of the linker (named LINK.V2) to create an overlayed version of your program. This means that parts of your program will only be loaded if and when they are needed, and will share the same space in memory. Your program will be smaller as a result, but will usually run more slowly because of the time needed to read and reread the code into memory.

Provided your modules obey the restrictions described below, all you have to do is specify the overlay structure to the linker. Loading of the overlays is automatic. You specify overlays in the

list of modules that you submit to the linker by enclosing them in parentheses. Each parenthetical list represents one overlay. For example, in the following response to the OBJECT MODULES prompt,

```
OBJECT MODULES [OBJ.] a + (b+c) + (e+f) + g + (i)
```

the elements (b+c), (e+f) and (i) are overlays. The remaining modules, and any drawn from the runtime libraries, make up the resident part of your program, or "root." Overlays are loaded into the same region of memory, so only one can be resident at a time. Duplicate names in different overlays are not supported, so each module can occur only once in a program.

The linker will replace calls from the "root" to an overlay and calls from an overlay to another overlay with an interrupt (followed by module identification and offset.) The interrupt is, by default, number #CD. If this conflicts with another use of this interrupt number in your program, you can specify another using the /OVERLAYINTERRUPT switch.

This switch takes a numeric parameter.

6.3.1 Restrictions

The name for the overlays is appended to the .EXE file, and the name of this file is encoded into the program so the Overlay Manager can access it. If, when the program is initiated, the overlay manager cannot find the .EXE file (perhaps you have renamed it or it is not in a directory specified by the path environment variable), then the linker will prompt you for a new name.

You can only overlay modules to which control is transferred and returned by a standard 8086 long (segmented) call/return instruction. This will always be true for Pascal and FORTRAN modules (although you should not attempt to overlay any of the modules in the standard runtime libraries). An exception is a function or a procedure parameter. In this case, the actual parameter (the function or procedure that you specify as the parameter) must either be in the same overlay in which the parameter is used to call it, or in the "root." You cannot use long jumps or indirect calls to pass control to an overlay.

6.3.2 Overlay Manager Prompts

Suppose that B: is the default drive; then in the following example,

```
Cannot find PAYROLL.EXE
Please enter new program spec:\employee\data\
```

the response *\employee\data* causes the overlay manager to look for \employee\data\payroll.exe on B:.

Now, suppose that it becomes necessary to change the disk in drive B:. If the overlay manager needs to swap overlays, it will find that PAYROLL.EXE is no longer on B:, and will give the following message,

```
Please put diskette containing B:\employee\data\payroll.exe
         in drive B: and strike any key when ready.
```

After the overlay has been read from the disk, the Overlay Manager will give the following message,

```
Please restore the original diskette.
Strike any key when ready.
```

6.4 Linker Switches

After any of the linker prompts, you may give one or more linker switches. Table 6.2 summarizes the linker switches you may use with Microsoft FORTRAN. See your MS-DOS manual for more information on linker switches and when and how to use them.

Table 6.2

Microsoft LINK Switches

Switch	Action
/CPARMAXALLOC:NNNN	By default, the cparMaxAlloc field (at offset #0C) in the EXE header (see Chapter 5 in the MS-DOS Programmer's Reference) is set to 65535. This switch allows you to set the value to any number between 1 and 65535; if the value you specify is less than the computed value of cparMinAlloc, the linker will use the value of cparMinAlloc (at offset #0A) instead. If you are running programs under MS-DOS 1.25, you should not use this switch.
/DSALLOCATE	Loads data at the high end of the data segment. For Microsoft Pascal and Microsoft FORTRAN programs, this switch is required and supplied automatically by the compiler.
/LINENUMBERS	Includes source listing line numbers and associated addresses in the linker listing, which allows you to correlate machine addresses with source lines when debugging. This correlation is also available on the object listing.
/MAP	Includes all EXTERN and PUBLIC variables in the linker list file.
/NODEFAULTLIBRARYSEARCH	This switch tells the linker not to automatically search FORTRAN.LIB.
/NOGROUPASSOCIATION	If you are linking old libraries (those supplied with versions of FORTRAN and Pascal prior to 3.20) you must use this switch.
/NOIGNORECASE	By default, "FOO", "foo", and "Foo" are treated by the linker as being equivalent. If /NOIGNORECASE is specified, then they are all different symbols.

Table 6.2 *(continued)*

Switch	Action
/OVERLAYINTERRUPT:NNNN	By default, the interrupt number used for passing control to overlays is CD hexadecimal. The overlay interrupt switch allows the user to select a different interrupt number. NNNN can be any of the following: • A decimal number from 0 to 255 (numbers that conflict with MS-DOS interrupts are not prevented, but their use is inadvisable.) • An octal number from 0 to 377. A number is interpreted as octal if it starts with a zero, e.g., 10 is 10 decimal but 010 is 8 decimal. • A hexadecimal number from 0 to FF. A number is interpreted as hexadecimal if it starts with "0x". Thus, 10 is ten decimal, 010 is 8 decimal, and 0x10 is 16 decimal.
/PAUSE	Tells Microsoft LINK to display the following message: About to generate .EXE file Change disks <press RETURN> You may then change disks before the linker continues. The /PAUSE switch is particularly useful for linking large programs, since it allows you to switch disks before writing the run file. However, if a VM.TMP file is created, you must not switch the disk in the default drive.

As with all linker switches, a unique abbreviation of the switch name is acceptable in place of the whole name.

Note

For Microsoft Pascal and Microsoft FORTRAN programs, do not use either of the additional linker switches /HIGH or /STACK.

Chapter 7
Using a Batch Command File

The MS-DOS batch file facility lets you create a batch file for executing a series of commands. This facility is described fully in your MS-DOS manual. This chapter provides a brief description of command files in the context of compiling, linking, and running an MS-FORTRAN program.

A batch command file is a text file of lines that are MS-DOS commands. If a batch file is open when MS-DOS is ready to process a command, the next line in the file becomes the command line. After processing all batch command lines (or if batch processing is otherwise terminated), MS-DOS goes back to reading command lines from the screen.

Batch file lines cannot be read by the compiler, the linker, or a user program. Thus, you cannot put responses to filename or other prompts in a batch file. All compiler parameters must be given on the command line, as described in Section 3.3.2, "Giving All Parameters on the Command Line."

The batch file may contain dummy parameters that you replace with actual parameters when you invoke it. The symbol %1 refers to the first parameter on the line, %2 to the second parameter, and so on. The limit is %9. A batch command file must have the extension .BAT and should be kept on either the program disk or the utility disk.

The PAUSE command, followed by the text of the prompt, tells the operating system to pause, display a prompt (which you have defined), and wait for some further input before continuing.

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

For example, use the line editor in MS-DOS to create the following batch file, COLIGO.BAT:

```
A : FOR1 %1, , ;
PAUSE ...If no errors, insert PAS2 disk in drive A : .
A : PAS2
PAUSE ...Insert runtime libraries disk in drive A : .
A : LINK %1;
%1
```

To execute this file, type:

```
COLIGO DEMO
```

DEMO is the name of the source program you want to compile, link, and run.

1. The first line of the batch file runs pass one of the compiler.
2. The second line generates a pause and prompts you to insert the pass two disk.
3. The third line runs pass two.
4. The fourth line generates a pause and prompts you to insert the runtime library.
5. The fifth line links the object file.
6. The sixth line runs the executable file.

A BAT file is only executed if there is neither a COM file or EXE file with the same name. Thus, if you keep your source file and BAT file on the same disk, give them different filenames.

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

Chapter 8
Compiling and Linking Large Programs

8.1 Avoiding Limits on Code Size 71
8.2 Avoiding Limits on Data Size 71
8.2.1 Limits on the Size of Arrays and COMMON Blocks 71
8.2.2 Passing Arrays as Arguments 72
8.2.3 Limits on Total Static Data 73
8.2.4 Restrictions on "Long" Data Allocation 73
8.2.5 $LARGE/$NOTLARGE Metacommand 73
8.3 Working With Limits on Compile Time Memory 75
8.3.1 Identifiers 75
8.3.2 Complex Expressions 76
8.4 Working With Limits on Disk Memory 77
8.4.1 Pass One 78
8.4.2 Pass Two 78
8.4.3 Linking 80
8.4.4 A Complex Example 81
8.5 Minimizing Load Module Size 82
8.5.1 I/O 82
8.5.2 Runtime Error Handling 83
8.5.3 Debugging 83

Occasionally, you may find that a large program exceeds one or more physical limits on the size of program the compiler, the linker, or your machine can handle. This chapter describes some ways to avoid or work within such limits.

8.1 Avoiding Limits on Code Size

The upper limit on the size of object code that can be generated at once by the MS-FORTRAN Compiler is 64K bytes. This limit applies only to generated code; data size limits are discussed in the following section.

Since you can compile any number of compilands separately and link them together later, the real limit on program size is not 64K but the amount of main memory available. For example, you can separately compile six different compilands of 50K bytes each. Linking them together produces a program with a total of 300K bytes of code.

In practice, a source file large enough to generate 64K bytes of code would be thousands of lines long and unwieldy both to edit and to maintain. A better practice is to break a large program into subroutines and functions and compile logical groups of them separately. Separate compilation will have no effect on final program size, but may increase the total size of object files.

8.2 Avoiding Limits on Data Size

8.2.1 Limits on the Size of Arrays and COMMON Blocks

The 8086 segmented memory addressing scheme makes it inefficient to address arrays which span more than one segment. We use the term "long" to describe such arrays as opposed to "short" arrays that can be contained within one segment. A segment consists of 65536 bytes aligned on a paragraph boundary. Arrays which are nearly or exactly 65536 bytes long may not fit in one segment if they do not also start on a paragraph boundary. To be safe, you should regard all arrays longer than 65521 bytes as "long."

8.2.2 Passing Arrays as Arguments

Microsoft FORTRAN allows you to define long arrays both statically and in COMMON blocks, but with the restriction that you see in the topic, "Restriction on 'long' data allocation," that follows. MS-FORTRAN will also generate the complex code sequences necessary to reference long arrays provided the compiler *knows* that the array is "long."

In the case of a subprogram argument that is an array with assumed or adjustable bounds, the compiler cannot "tell" if the actual argument was a long array, and for the sake of efficiency, assumes, by default, that it was not. Undefined results will occur if the subprogram tries to reference beyond the first segment.

You must use one of the following methods to get your arrays passed correctly:

1. You can fix the bounds of the formal argument and set them big enough so that the compiler knows the actual argument is "long."

   This method is inconvenient, however, because it implies that all actual arguments will have the same bounds, which is usually not the case. (The compiler doesn't enforce this correspondence and it is common practice to ignore it.)

2. You can continue to use assumed and adjustable bound arrays by using the $LARGE metacommand to specify that the actual arguments may (but not necessarily) be "long." See the topic "The $LARGE metacommand" that follows for a description of this metacommand. You may use $LARGE even when the array is known to be short (it has fixed bounds) and the long array reference code will be generated.

You can pass short arrays as actual arguments to long formal arguments with either method (1) or (2) and your program will work correctly.

8.2.3 Limits on Total Static Data

By default, static variables, including arrays, are allocated in the default data segment of the 8086 (i.e., in DGROUP). The default data segment also contains data defined by the runtime, the stack and space for dynamic allocation of file blocks (634 bytes per file) and, if $DEBUG has been used, some subprogram entry and exit information. There is a limit of 64k (65536) bytes on the total amount of data in the default data segment.

Each COMMON block or long array is allocated as many adjacent segments outside DGROUP as are required to accommodate it. So you can increase the amount of space available to you by moving data into COMMON blocks, either blank or named. However, should you make a COMMON block span more than one segment doing this, the restrictions described below may then apply. You can more conveniently move individual (short) arrays out of DGROUP by using the $LARGE metacommand, which causes them to be allocated in separate segments.

8.2.4 Restriction on "Long" Data Allocation

No scalar may be allocated so that it spans a segment boundary. Each static long array or COMMON block is allocated at the beginning of a segment (on a "paragraph" boundary in 8086 terms) so that the boundary between the 65536th and 65537th byte from the start of the array or COMMON block, for example, *must* fall between two scalar items. This applies whether the items are variables or array elements.

8.2.5 $LARGE/$NOTLARGE Metacommands

```
Syntax   $[NOT]LARGE [name [, name2]...]
```

If your program uses large amounts of data, you can avoid the limits on total data size by using the $LARGE metacommand when you compile. This moves arrays (but not scalars) from the default data segment to extra data segments. You are unlikely to fill up the default segment with scalars, so this option effectively allows you to use all the memory available. The indirect references that must be made to data in other segments are less efficient and result in more code being generated than references to the default data segment, so your programs will be bigger and slower if you use this option.

These metacommands are used to control the way arrays are allocated and referenced. $LARGE may be applied to individual arrays, or, by providing no arguments, to all the arrays in a subprogram. When $LARGE has been set to apply to all arrays, $NOTLARGE will disable its effect on those arrays that you want to remain in the default data segment. For example:

```
$LARGE
$NOTLARGE BAR
      SUBROUTINE F0002 (PARRAY)
      REAL*4 BAR, PARRAY (*)
      DIMENSION BAR (10)
      .
      .
      RETURN
      END
```

The effect of $LARGE on an array depends on whether the array is a formal argument. If it is a formal argument, it tells the compiler that the actual argument may be a long array. The compiler will then generate the more complex code sequence for references to it.

All arrays declared $LARGE will be allocated outside of DGROUP. Any array that is obviously larger than 64K bytes (i.e., an array with constant bounds that is larger than 64K) will automatically be allocated as a 'large' array regardless of its association with $LARGE.

Consider the following code fragments:

```
$NOTLARGE  BAR
      SUBROUTINE  F001 (PARRAY)
      INTEGER IARRAY (100)
$LARGE
      REAL*4 BAR, PARRAY(*)
      DIMENSION  BAR(10)
      BAR(1)=10.98
      .
      .
      .
      RETURN
      END
```

The integer array IARRAY and the parameter array PARRAY are declared 'large.' BAR is declared 'not large'.

```
      SUBROUTINE  F002 (PAR)
      REAL*8 FIGURE(10)
      LOGICAL LARRAY(70000)
$NOTLARGE
      INTEGER IARRAY (15)
      INTEGER PAR(*)
      .
      .
      .
      RETURN
      END
```

The arrays in this program unit are declared 'not large' and are allocated to the default data segment (except for LARRAY, which will unconditionally be allocated to additional segments).

8.3 Working Within the Limits of Compile Time Memory

During compilation, large programs are most often limited in the number of identifiers in any one source file. They are occasionally limited by the complexity of the program itself. If one of these limits is reached, you will see the following error message:

```
Compiler Out Of Memory
```

There is no particular limit on number of bytes in a source file. The number of lines is limited to 32767, but in practice, any source file this big will run into other limits first.

8.3.1 Identifiers

Pass one of the compiler can handle a maximum of around 1000 identifiers, assuming your memory is big enough to provide a full data segment of 64K. In MS-FORTRAN, identifer entries are created for the following objects:

1. the program
2. subroutines and functions declared in the program unit

3. subroutines and functions referenced in the program unit
4. COMMON blocks
5. common variables
6. statement functions
7. formal parameters
8. "local" variables

Identifiers of objects 5 through 8 are required only while the subroutine or function that contains them is being compiled. These identifiers are discarded at the end of the subroutine and the space they used is made available for other identifiers.

Hence, you can create much bigger programs by splitting up your code into more subroutines and functions. Such a practice allows the "local" identifier space to be shared.

You can go even further by placing the subroutines and functions in files of their own and compiling them separately, since this usually reduces the number of identifiers in groups being used per compiland.

Remember that you may have to create data items in common to communicate between the new procedures, or preferably, from the point of view of good program structure, write communication subroutines. However, either of these may tend to defeat the purpose of breaking up the program in the first place.

8.3.2 Complicated Expressions

It is also possible to run out of memory in pass one with any of the following cases:

1. a very complicated statement or expression (i.e., one that is very deeply nested)
2. a large number of error messages
3. a very large block of specification statements (EQUIVALENCE statements in particular)

Usually, if a program gets through pass one without running out of memory, it will get through pass two. The major exception occurs with complicated basic blocks, as in either of the following:

1. sequences of statements with no labels or other breaks
2. sequences of statements containing very long expressions or parameter lists (especially with I/O statements)

Also, pass two uses symbol table entries for objects 1 through 4 in Section 8.3.1. Unlike pass one, pass two also creates entries for many of the transcendental functions that are called by a program. However, these are limited in number. In any case, pass two makes a smaller number of symbol table entries than pass one.

If pass two runs out of memory, it displays the message:

```
Compiler Out Of Memory
```

The error message will give a line number reference. If there is a particularly long expression or parameter list near this line, break it up by assigning parts of the expression to local variables (or using multiple WRITE statements). If this does not work, add labels to statements to break the basic block.

8.4 Working With Limits on Disk Memory

Another type of limit you may encounter is in the number of disk drives on your computer or the maximum file size on one disk. As with other limits, there are several possible solutions discussed in the following sections.

The simplest method of avoiding these limits is to first load a compiler pass, then switch disks and run the pass.

8.4.1 Pass One

For FOR1.EXE, just type *FOR1* (or *dev:FOR1* if necessary) to load pass one. When the "Source File" prompt appears, you can remove the disk containing FOR1.EXE. If you have a single-drive system, replace the system disk with the disk containing your source file. FOR1.EXE will write its intermediate files on the same disk.

If you have a two-drive system, insert your source file in the nondefault drive. Since the intermediate files are always written to the default drive, you will need to give an explicit device (i.e., drive) letter for your source file. Typically a source listing file would go on the same drive as the source.

If your source file will not fit on one disk, you can break it into pieces and use the $INCLUDE metacommand to compile the pieces as a group.

These $INCLUDE files can be typed at your screen (or console). Just give *USER* as the name of your source file, and type your $INCLUDE metacommands directly, one per line. You will need to type <CONTROL-*Z*> (end-of-file) to end the compilation.

If your source file doesn't fit on one disk, your source listing file will not fit either, so you will need to send it directly to the printer.

Another way to control a large listing file is by including the $NOLIST metacommand at the beginning of your source file, and then using the $LIST and $NOLIST metacommands to bracket only those portions of the source for which a source listing is required. In particular, you may want to exclude $INCLUDEd files when compiling subprograms.

8.4.2 Pass Two

Two command line parameters available with pass two can help you with disk limitations.

1. You can indicate a drive letter on which your intermediate files, PASIBF.SYM and PASIBF.BIN, can be found.
2. The /PAUSE switch tells pass two to pause while you remove the disk containing PAS2.EXE and insert some other disk.

For example, if you have a single-drive system, insert your PAS2.EXE disk and type *PAS2 /PAUSE*. After PAS2.EXE is loaded, you will see the message:

```
Press ENTER key to begin pass two
```

Take out the PAS2.EXE disk and insert the disk with the intermediate file from pass one. Now press the ENTER (i.e., RETURN) key, and pass two will run.

If you have two drives, but you run out of disk memory when executing pass two, you need to have the files PASIBF.SYM and PASIBF.BIN on one drive and the intermediate file PASIBF.TMP (and PASIBF.OID if you are making an object listing file) on the other drive.

The PASIBF.TMP file (and the PASIBF.OID file used in pass three) are always written to the default drive.

Give pass two a drive letter to specify the drive containing the PASIBF.SYM and PASIBF.BIN files; for example, "PAS2 B". Normally, you would also need the pause command; for example, *PAS2 B/PAUSE*. Pass two will respond with a message such as the following:

```
PASIBF.SYM and PASIBF.BIN are on B:
```

This message is followed by the pause prompt:

```
Press enter key to begin pass two
```

When you run pass two with the PASIBF files on two disks, the object file should usually go on the same disk as PASIBF.TMP (and PASIBF.OID); that is, in the default drive. If it doesn't quite fit, and you are making an object listing file, you could compile your program twice, once without the object listing but with the object file itself, and once with an object listing but with NUL used for the object file.

8.4.3 Linking

If you are making a large program with small disks (or only one disk drive), you may run into similar problems when you link your program. Since you can split your program into pieces and compile them separately, but you must link the entire program at one time, you may run into disk limitations in the linker but not the compiler.

The linker will prompt you for any object files and/or libraries it cannot find, so you can swap in the correct disk and continue linking. Also, the /PAUSE switch makes the linker wait after linking but before writing the run (EXE) file, so you can create a run file that fills an entire disk. However, creation of the virtual file, VM.TMP, and the link map limit the amount of disk swapping you can do.

On a single-drive system:

1. Load the linker by typing *LINK*.
2. Remove the disk containing LINK.EXE and insert the disk containing your object file(s) and, if there is room, any libraries.
3. Respond normally to the linker prompts, except to include the /PAUSE switch with the run file if you want the run file on another disk.

Unless all object files, libraries, and the run file will fit on one disk, you must not write the linker listing to a disk file. Instead, send the linker map to NUL, CON, or directly to your printer. Since the map is written at various points in the linking process, you cannot swap the disk on which the map is written.

The linker will prompt you when it needs an object file, a library file, or is about to write the run file; exchange disks as necessary when this happens. If the linker gives a message that it is creating VM.TMP, its virtual memory file, you cannot switch disks anymore, so you may not be able to link without more memory or a second disk drive.

With two disk drives, you can devote one drive (the default) to the VM.TMP file (and to the link map, if you want one). Use the other drive for your object files, libraries, and run file (using the /PAUSE switch). With this method, you can link very large programs.

The linker makes two passes through the object files and libraries: one to build a symbol table and allocate memory, and one to actually build the run file. This means you will insert a disk containing object files or libraries twice, and finally insert the disk that will receive your run file.

8.4.4 A Complex Example

The following example illustrates compiling and linking a very large program. The example assumes that the machine has two drives and that the programmer doesn't want any of the listing files.

Pass one

1. Log onto drive B: and insert an empty disk in B:.
2. Insert the disk containing FOR1.EXE in drive A:, type *A:FOR1*, and wait for the "Source File" prompt.
3. Remove the disk containing FOR1.EXE from A: and insert the disk containing the source file LARGE.FOR.
4. Respond to the "Source File" prompt with *A:LARGE, A:LARGE*, and wait for pass one to run.

Pass two

1. Log onto drive A:. Remove source disk from A:.
2. Insert the disk containing PAS2.EXE in A:, type *PAS2 B/PAUSE* and wait for the pass two prompt.
3. Remove the disk containing PAS2.EXE from A:, insert an empty disk (to which the object file will be written).
4. Respond to the pass two prompt by pressing the RETURN key, and wait for pass two to run.
5. Remove the disk containing the object file from A:.

Linking

1. Log onto drive B: (which contains a now-empty disk).
2. Insert LINK.EXE in A:. Type *A:LINK* and wait for the "Object Modules" prompt.

3. Remove the disk containing LINK.EXE from A: and insert the disk containing the object file(s).
4. Respond to the "Object Modules" prompt by typing *A:LARGE* (plus any other object files).
5. Respond to the "Run File" prompt by typing *LARGE/PAUSE.*
6. Respond to the "List File" prompt by pressing the RETURN key, or type *B:LARGE* to get a linker map.
7. Respond to the "Libraries" prompt by pressing the RETURN key or with a library name (the library must be on A:).
8. Wait for the linker to run, swapping the A: disk after prompts as necessary.

8.5 Minimizing Load Module Size

Some MS-FORTRAN load modules can be reduced in size by eliminating runtime modules your program doesn't use. Reductions can be made in several areas:

1. I/O
2. runtime error messages
3. real number operations
4. debugging

8.5.1 I/O

Because most MS-FORTRAN programs perform I/O, they require linking to the MS-FORTRAN file system in the runtime library. However, some programs do not perform I/O and others perform I/O by directly calling MS-FORTRAN "Unit U" file routines or calling operating system I/O routines. (For more information on Unit U, see Section 10.2, "An Overview of the File System.")

Nonetheless, all programs include calls to INIVQQ and ENDYQQ, the procedures that initialize and terminate the file system. These calls increase the size of the load module by linking and loading routines that may never be used.

If a program doesn't need the file system routines, you can eliminate unnecessary file support by declaring dummy INIVQQ and ENDYQQ subroutines in your program, as follows:

```
SUBROUTINE INIVQQ
END

SUBROUTINE ENDYQQ
END
```

The linker will still load the Unit U procedures necessary to access the terminal (INIUQQ, ENDUQQ, PTYUQQ, PLYUQQ, and GTYUQQ), so that it can write any runtime error messages.

However, if you do include the dummy subroutines described, and the linker produces any error messages for global names that end with the "VQQ" or "UQQ" suffix, your program requires the file system and the process described above will not work.

8.5.2 Runtime Error Handling

If runtime error messages are not required, the load module can be further reduced in size by eliminating the error message module and the Unit U procedures. Two null object modules are provided as replacements: NULF.OBJ and NULE6.OBJ. NULF.OBJ contains the dummy subroutines for INIVQQ and ENDYQQ, as well as dummies for INIUQQ and ENDUQQ. NULE6.OBJ replaces EMSEQQ and provides simple termination of a program if an error occurs.

8.5.3 Debugging

Compiling and linking a program with the $DEBUG metacommand may generate up to 40 percent more code than with $NODEBUG. Therefore, after a program has been successfully compiled, linked, and run, remove the $DEBUG from your source file and repeat the entire process to create a program that will run considerably faster.

Chapter 9
Using Assembly Language Routines

9.1 Calling Conventions 87

9.2 Internal Representations of Data Types 89

9.3 Interfacing to Assembly Language Routines 91

This chapter first describes the MS-FORTRAN calling conventions and internal representations of data types, and then shows how to interface 8086 assembly language routines to MS-FORTRAN programs. The information in this chapter is not required for most MS-FORTRAN programs and is intended primarily for the experienced programmer who is familiar with the following material:

1. The EXTERNAL statement (see Section 3.2.17, "The EXTERNAL Statement," in the *Microsoft FORTRAN Reference Manual* for a description of the statement)
2. Subroutine and function arguments (see Section 3.2.5, "The CALL Statement," in the *Microsoft FORTRAN Reference Manual* for a description of the statement)
3. MS-MACRO Assembler (see your MS-DOS manual)

9.1 Calling Conventions

At runtime, each active subroutine or function has a "frame" allocated on the stack. The frame contains the data shown in Figure 9.1.

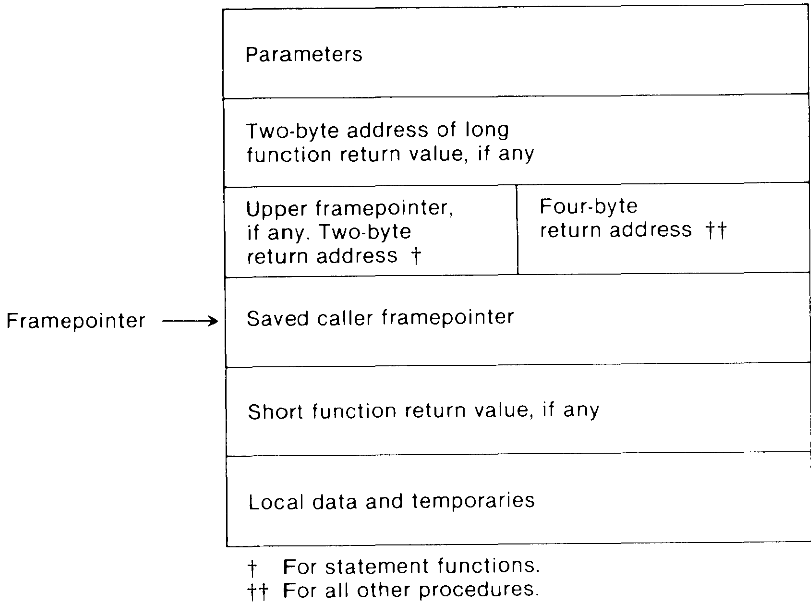


Figure 9.1. Contents of the Frame

The framepointer points at the saved caller framepointer, below the return address, and is used to access frame data. A statement function nested within another subroutine or function has an upper framepointer, so that it can access variables in the enclosing frame.

The following takes place during a procedure or function call:

1. The caller saves any registers it needs (except the framepointer).
2. The caller pushes parameters in the same order as they are declared in the source and then performs the call.
3. The called routine pushes the old framepointer, sets up its new framepointer, and allocates any other stack locations needed.

To return to the calling routine, the called routine restores the caller's framepointer, releases the entire frame, and returns. Not all of these steps need necessarily be taken in an assembly language routine. You must only ensure that the framepointer is not

modified and that the entire frame, including all parameters, is popped off the stack before returning. For further information on the assembly language interface, see Section 9.3, "Interfacing to Assembly Language Routines."

A function always returns its value in registers. For REAL*4 and REAL*8, the caller allocates a frame temporary for the result and passes the address to the function like a parameter. When the called routine returns, it places the address back in the normal return register.

In MS-FORTRAN, all subroutines and functions are PUBLIC or EXTERN. All calls to subroutines or functions are long calls (i.e., have four-byte addresses). All calls to statement functions are short calls (i.e., have two-byte addresses).

The called routine must save the BP register, which contains the MS-FORTRAN framepointer, as well as the DS segment register. The SS register is used by interrupt routines, both user-declared and 8087 support, to locate the default data segment, and so must not be changed (at least, if interrupts are enabled). Other registers (AX, BX, CX, DX, SI, DI, and ES) need not be saved.

Functions return a one-byte value in AL, a two-byte value in AX, and a four-byte value in DX:AX (high part:low part, or segment:offset).

9.2 Internal Representations of Data Types

This section describes the internal representation of MS-FORTRAN data types. Programmers who use both MS-FORTRAN and MS-Pascal should pay particular attention to the data type and parameter/argument passing differences when passing data between the two languages. For internal representations of MS-Pascal data types, see the *Microsoft Pascal Compiler User's Guide.*

1. Integer (INTEGER*2 and INTEGER*4)

   INTEGER*2 values are 16-bit two's complement numbers; INTEGER*4 values are 32-bit two's complement numbers.

2. Real (REAL*4) and double precision (REAL*8)

   Reals are IEEE 4-byte real numbers. They have a sign bit, an 8-bit excess 127 binary exponent, and a 24-bit mantissa. The mantissa represents a number between 1.0 and 2.0. Since the high-order bit of the mantissa is always 1, it is not stored in the number. This representation gives an exponent range of 10^(+ or -)38 and 7 digits of precision. The maximum real number is normally 1.701411E37. The most significant byte contains the sign bit and the most significant bits of the exponent. The least significant byte contains the least significant bits of the mantissa.

   Double precision real numbers are IEEE 8-byte real numbers. They have a format similar to 4-byte REAL numbers. The exponent is 11-bits (excess 1023), and the mantissa has 52 bits (plus the implied high-order 1 bit).

3. Single and double DECIMAL floating-point

   Decimal floating-point numbers consist of a byte containing a sign bit and a 7-bit exponent in excess 64 notation followed by a mantissa consisting of 6 (single) or 14 (double) binary coded decimal digits packed two to a byte (if the exponent byte is zero, the number is zero.)

   The allowable ranges of numbers are:

single	+ .1E-63 to +.999999E63 - .1E-63 to - .999999E63
double	+ .1D-63 to +.99999999999999E63 - .1D-63 to -.99999999999999E63

4. Complex (COMPLEX*8 and COMPLEX*16)

   Complex numbers consist of a real number representing the real part followed by a real number representing the imaginary part. Each component (real and imaginary) of a COMPLEX*8 is a REAL*4. Each component of a COMPLEX*16 is a REAL*8.

5. Logical (LOGICAL*2 and LOGICAL*4)

   LOGICAL*2 values occupy two bytes. The least significant (first) byte either is 0 (.FALSE.) or 1 (.TRUE.); the most significant byte is undefined. LOGICAL*4 variables occupy two words, the least significant (first) of which contains a LOGICAL*2 value. The most significant word is undefined.

6. Character

   Character values occupy 8 bits and correspond to the ASCII collating sequence.

7. Files

   MS-FORTRAN files use file control blocks (of type FCBFQQ), allocated dynamically on the heap. File control blocks for MS-FORTRAN are not identical to file control blocks for MS-Pascal. See Appendix B, "The Microsoft FORTRAN File Control Block," for a complete listing.

8. Procedural parameters (subroutine and function parameters)

   Procedural parameters contain a reference to the location of the subroutine or function. The parameter always contains two words: the first word is zero, and the second word contains a data segment offset address. This is an offset to two words in the constant area that contain the segmented address of the actual routine.

9.3 Interfacing to Assembly Language Routines

All subroutines and functions in MS-FORTRAN are external. They need not be declared as external with the EXTERNAL statement. When a subroutine or function is called, the addresses of the actual parameters are first pushed on the stack in the order that they are declared. MS-FORTRAN always uses calls by reference, even if the actual parameters are expressions or constants.

If the procedure called is a function and if the function return type is real or double precision, an additional implicit parameter for the function is pushed on the stack. This parameter is the two-byte address of a temporary variable created by the calling program.

After all parameters have been pushed, the return address is pushed. If the procedure called is a function, the return value is expected as follows:

1. If the return value is a two-byte integer or logical value, that value is expected in the AX register, as shown in Figure 9.2:

Figure 9.2. Two-Byte Return Value

2. If the return value is a four-byte integer or logical value, that value is expected in the DX, AX pair, as shown in Figure 9.3:

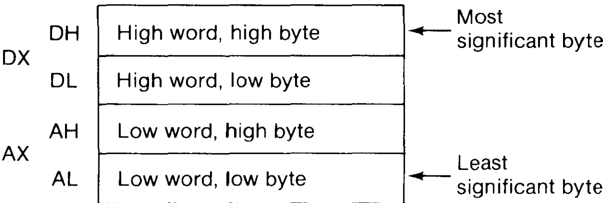


Figure 9.3. Four-Byte Return Value

3. If the return value is a four or eight byte REAL value, or an eight or sixteen byte COMPLEX value, that value is expected in the temporary variable created by the calling program. The two-byte address of this temporary variable is the last parameter pushed on the stack. It is always at BP+6 (see Example 2).

*Example 1. INTEGER*4 Add Routine*

Assume the following MS-FORTRAN program has been compiled:

```
PROGRAM   EXAMPL1
INTEGER I, TOTAL, IADD
I = 10
TOTAL = IADD (I,15)
WRITE (*,'(1X,I6)') TOTAL
END
```

At runtime, just prior to the transfer to IADD, the stack would be as shown in Figure 9.4:

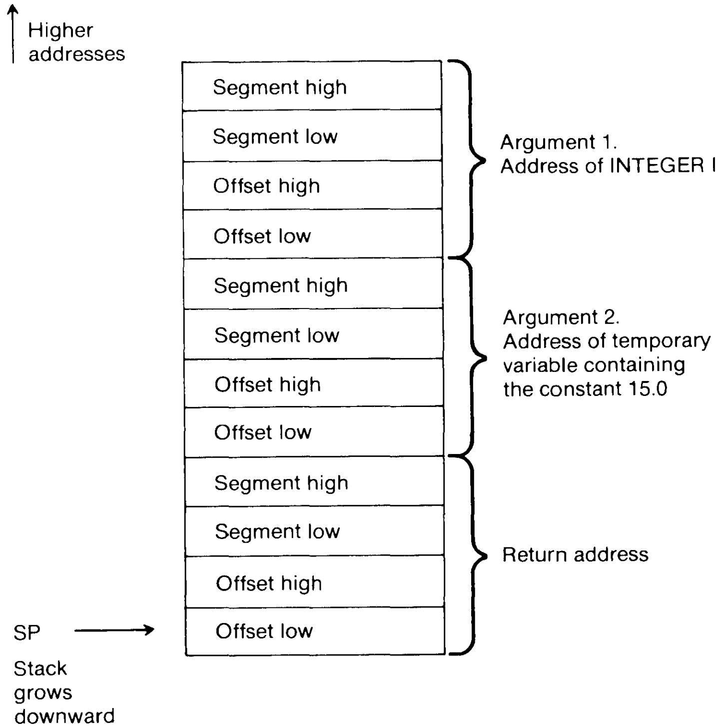


Figure 9.4. Stack Before Transfer to IADD

An example of an assembly language routine that implements the integer ADD function, IADD, is illustrated in the following routine. Note that the function return value is AX,DX.

```
DATA      SEGMENT PUBLIC 'DATA'
          ;See Note at end of this section.
DATA      ENDS

DGROUP    GROUP DATA   ;See Note.
CODE      SEGMENT 'CODE'
          ASSUME CS:CODE,DS:DGROUP,SS:DGROUP ;
          See Note.
PUBLIC    IADD
IADD      PROC FAR

          PUSH   BP                   ;Save framepointer
                                          on stack
          MOV    BP,SP
          LES    BX,DWORD PTR [BP+10] ;ES,BX := addr of
                                          1st param
          MOV    AX,ES:[BX]           ;AX,DX := value of
                                          1st param
          MOV    DX,ES:[BX]+2
          LES    BX,DWORD PTR [BP+6]  ;ES,BX := addr of
                                          2nd param
          ADD    AX,ES:[BX]           ;AX,DX := 1st
                                          parameter plus
          ADC    DX,ES:[BX]+2         ;2nd parameter
          MOV    SP,BP
          POP    BP                   ;Restore the
                                          framepointer
          RET    08H                  ;Return, pop 8 bytes

          IADD   ENDP
          CODE   ENDS

END
```

*Example 2. REAL*4 Add Routine*

Assume the following FORTRAN program has been compiled:

```
PROGRAM   EXAMPL2
REAL R, TOTAL, RADD
R = 10.0
TOTAL = RADD (15.0,R)
WRITE (*,'(1X,F10.3)') TOTAL
END
```

At runtime, just prior to the transfer to RADD, the stack would be as shown in Figure 9.5:

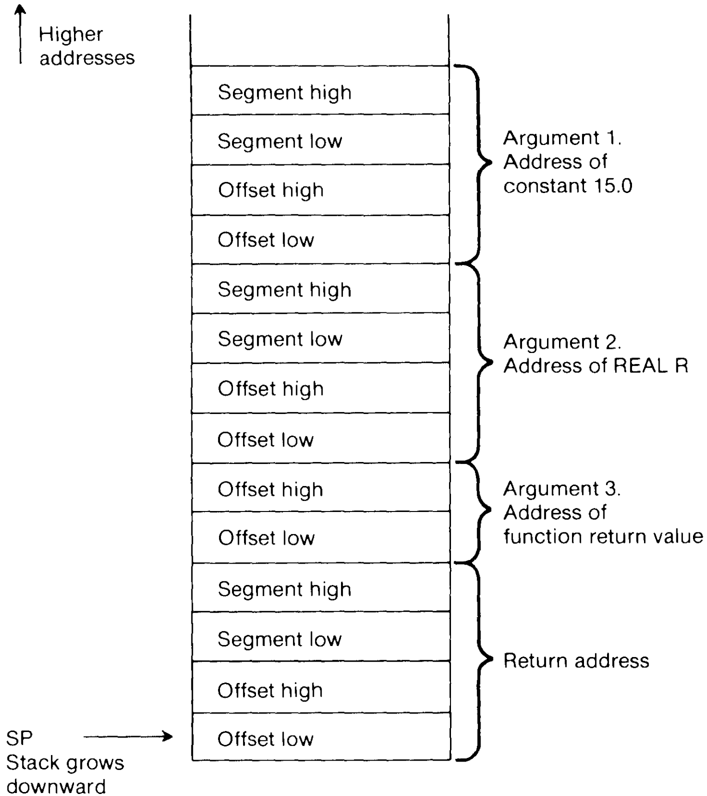


Figure 9.5. Stack Before Transfer to RADD

An example of an assembly language routine that implements the real add function, RADD, is illustrated in the following routine. Note that the function return value is in the location specified by BP+6.

```
DATA      SEGMENT PUBLIC 'DATA'
          ;See Note at end of section
DATA      ENDS

DGROUP    GROUP     DATA ;See Note
CODE      SEGMENT   'CODE'
          ASSUME    CS:CODE,DS:DGROUP,SS:DGROUP ;
          See Note
PUBLIC    RADD
RADD      PROC      FAR
          PUSH      BP                     ;Save framepointer
                                                on stack
          MOV       BP,SP
          LES       BX,DWORD PTR [BP+12]   ;ES,BX := addr of
                                                1st param
          FLD       ES:[BX]                ;Push value of
                                                1st param
                                           ;on 8087 stack
          LES       BX,DWORD PTR [BP+8]    ;ES,BX := addr of
                                                2nd param
          FLD       ES:[BX]                ;Push value of
                                                2nd param
                                           ;on 8087 stack
          FADDP     ST(1),ST               ;Add first two items
                                           ;on 8087 stack
          MOV       DI,[BP+6]              ;DI := addr of
                                                funct return
          FSTP      [DI]                   ;Store result on
                                                8087 stack
                                           ;at funct return
                                                location
          FWAIT
          MOV       SP,BP                  ;Restore the
                                                framepointer
          POP       BP
          RET       0AH                    ;Return, pop 10 bytes

          RADD      ENDP
          CODE      ENDS

END
```

Important

1) Data used by assembly language routines must be placed in a segment whose name is DATA, whose classname is 'DATA', and which is grouped in DGROUP. The ASSUME statement is required.

2) If you use 8087 instructions and you want them to be emulated, you must use the e switch in the assembler. Using this switch without an 8087 in your system may cause your program to wait indefinitely at the first FWAIT instruction.

Chapter 10
Advanced Topics

10.1 The Structure of the Compiler 101
10.1.1 The Front End 103
10.1.2 The Back End 104
10.1.2.1 Pass Two 104
10.1.2.2 Pass Three 106
10.2 An Overview of the File System 106
10.3 Runtime Architecture 108
10.3.1 Runtime Routines 109
10.3.2 Memory Organization 110
10.3.3 Initialization and Termination 114
10.3.3.1 Machine Level Initialization 116
10.3.3.2 Program Level Initialization 117
10.3.3.3 Unit Level Initialization 117
10.3.3.4 Program Termination 119
10.3.4 Error Handling 119
10.3.4.1 Machine Error Context 121
10.3.4.2 Source Error Context 122
10.4 Floating-Point Operations 123
10.4.1 The $NOFLOATCALLS Metacommand Option 124
10.4.2 The Alternate Math Package Option 125
10.4.3 No Emulation Option 126
10.4.4 Decimal Math Option 127
10.5 MS-DOS 2.0 Issues 128
10.5.1 Exit Status Available to 2.0 MS-DOS 128

This chapter contains advanced technical information that will be of interest primarily to experienced programmers. Since MS-Pascal and MS-FORTRAN (but not FORTRAN-80) have the same compiler back end, and share a common file and runtime system, much of the information that follows refers to both languages. Differences, where they exist, are noted.

10.1 The Structure of the Compiler

The compiler is divided into three phases, or passes, each of which performs a specific part of the compilation process. Figure 10.1, which follows, illustrates the basic structure of the compiler and its relationship to the files that it reads and writes.

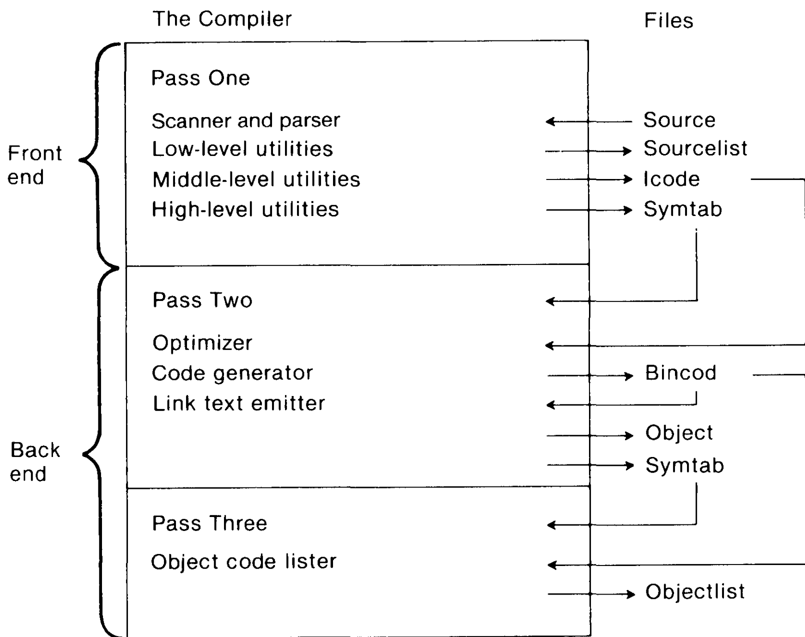


Figure 10.1 The Structure of the Compiler

Pass one, which normally corresponds to a file named FOR1.EXE, constitutes the front end of the compiler and performs the following actions:

1. reads the source program
2. compiles the source into an intermediate form
3. writes the source listing file
4. writes the symbol table file
5. writes the intermediate code file

Passes two and three (the files PAS2.EXE and PAS3.EXE) together make up the back end, which does the following:

1. optimizes the intermediate code
2. generates target code from intermediate code
3. writes and reads the intermediate binary file
4. writes the object (link text) file
5. writes the object listing file

Both the front and back end of the compiler are written in MS-Pascal, in a source format that can be transformed into either relatively standard Pascal or into system level MS-Pascal. (See the *Microsoft Pascal Reference Manual* for a discussion of implementation levels in MS-Pascal.)

All intermediate files contain MS-Pascal records. The front and back ends include a common constant and type definition file called PASCOM, which defines the intermediate code and symbol table types. The back ends use a similar file for the intermediate binary file definition. Formatted dump programs for all intermediate files and object files are available for special purpose debugging.

The symbol table record is relatively complex, with a variant for every kind of identifier (variables, procedures, intrinsic functions, and common blocks). The intermediate code (or Icode) record contains an Icode number, opcode, and up to four arguments; an argument can be the Icode number of another Icode to represent expressions in tree form, or something else (such as a symbol table reference, constant, or length). The intermediate binary code record contains several variants for absolute code or data bytes, public or external references, label references and definitions, etc.

10.1.1 The Front End

The MS-FORTRAN front end can be divided into several parts:

1. the scanner
2. various utilities
3. EXECSTMTS, which processes executable statements
4. DECLSTMTS, which processes declarative statements

The front end is driven by recursive descent syntax analysis, using a set of procedures such as EXPRESSION (for expressions) and VARIABLE (for variables). Parsing is performed on a strict statement basis. The scanner procedure GETSTMT gets the next MS-FORTRAN statement into the statement buffer.

Overall compilation control depends on a series of states, handled as shown in Table 10.1.

Table 10.1

Front End Compilation Procedures

Name	Function
INITSTATE	Initialize procedure
HEADSTATE	Process subroutine header
IMPSTATE	Process IMPLICIT statements
SPECSTATE	Process specification statements
DATASTATE	Process DATA statements
STMTFUNSTATE	Process statement functions
EXECSTATE	Process executable statements
ENDSTATE	End procedure

After initialization in INITSTATE, the current state cycles from HEADSTATE through EXECSTATE for the program and for all subroutines and functions. The final procedure, which carries out program termination, is ENDSTATE.

MS-FORTRAN intermediate files are written in the same manner as for the MS-Pascal front end. A few of the intermediate code operations are specific to MS-FORTRAN, particularly those concerned with assigned GOTO and DO statements. The symbol table contains special flags for COMMON and EQUIVALENCE variables, since these affect common subexpression optimization.

10.1.2 The Back End

Of the separate passes that make up the back end of the compiler, pass two is required while pass three is optional.

10.1.2.1 Pass Two

The optimizer reads the interpass files in the following order: first the symbol table for a block, then the intermediate code for the block. Optimization is performed on each "basic block," i.e., each block of intermediate code up to the first internal or user label or up to a fixed maximum number of Icodes, whichever comes first.

Within a block, the optimizer can reorder and condense expressions as long as the intent of the program(mer) is preserved. For instance, in the following program fragment, the array address A (J, K) need be calculated only once:

```
      A(J,K) = A(J,K)+1
C     J = J-1
      IF (A(J,K) .EQ. MAX) CALL PUNT
```

However, if the preceding fragment is rewritten to include the assignment to J, shown in the fragment as a comment, the array address in the IF statement must be partially recalculated.

This optimization is called common subexpression elimination. The optimizer also reorders expressions so that the most complicated parts are done first, when more registers for temporary values are available. It also does several other optimizations, such as:

1. constant folding not done by the front end
2. strength reduction (changing multiplications and divisions into shifts when possible)
3. peephole optimization (removing additions of zero, multiplications by one, and changing A := A + 1 to an internal increment memory Icode)

The optimizer works by building a tree out of the intermediate codes for each statement and then transforming the list of statement trees.

There are seven internal passes per basic block:

1. statement tree construction from the Icode stream
2. preliminary transformations to set address/value flags
3. length checks and type coercions
4. constant and address folding, and expression reordering
5. peephole optimization and strength reduction
6. machine-dependent transformations
7. common subexpression elimination

Finally, the optimizer calls the code generator to translate the basic block from tree form to target machine code.

The code generator must translate these trees into actual machine code. It uses a series of templates to generate more efficient code for special cases. For example, there is a series of templates for the addition operator. The first template checks for an addition of the constant one. If this addition is found, the template generates an increment instruction. If the template does not find an addition of one, the next template gets control and checks for an addition of any constant. If this is found, the second template generates an add immediate instruction.

The final template in the series handles the general case. It moves the operands into registers (by recursively calling the code generator itself), then generates an add register instruction. There is a series of templates for every operation. The code generator also keeps track of register contents and several memory segment addresses (code, static variables, constant data, etc.), and allocates any needed temporary variables.

The code generator writes a file of binary intermediate code (BINCOD), which contains machine instruction opcodes with symbolic references to external routines and variables. A final internal pass reads the BINCOD file and writes the object code file.

10.1.2.3 Pass Three

This short pass reads both the BINCOD file (described in the previous section) and a version of the symbol table file as updated by the optimizer and code generator. Using the data in these files, pass three writes a listing of the generated code in an assembler-like format.

For more information about the compiler (especially the back end), see the article "Native-code Compilers are Portable and Fast," (James G. Letwin and Andrea L. Lewis, *Electronic Design*, May 14, 1981).

10.2 An Overview of the File System

MS-FORTRAN and MS-Pascal are designed to be easily interfaced to existing operating systems. The standard interface has two parts:

1. a file control block (FCB) declaration
2. a set of procedures and functions, called Unit U, that are called from MS-FORTRAN or MS-Pascal at runtime to perform input and output

This interface supports three access methods: TERMINAL, SEQUENTIAL, and DIRECT.

Each file has an associated FCB (file control block). The FCB record type begins with a number of standard fields, whose details are independent of the operating system. These are followed by fields, such as channel numbers, buffers, and other operating system data, that are dependent on the operating system.

The advanced MS-Pascal user can access FCB fields directly, as explained in Chapter 8, "Files," of the *Microsoft Pascal Reference Manual*. There is no standard way to access FCB fields within MS-FORTRAN.

Both MS-FORTRAN and MS-Pascal have two special file control blocks that correspond to the keyboard and the screen of your terminal. These two file control blocks are always available. In MS-Pascal, they are the predeclared files INPUT and OUTPUT; in MS-FORTRAN, they are unit number 0 (or *) and are accessed through a variable TRMVQQ, which is declared as follows:

```
VAR TRMVQQ:ARRAY [BOOLEAN] OF ADR OF FCBFQQ;
```

The false element references the output file; the true element references the input file.

Unit U refers to the target operating system interface routines. The file routines specific to MS-Pascal are called Unit F; the file routines specific to MS-FORTRAN are called Unit V. Code generated by the compiler of either language contains calls to the appropriate unit (F or V), which in turn call Unit U routines.

Figure 10.2 shows this relationship schematically.

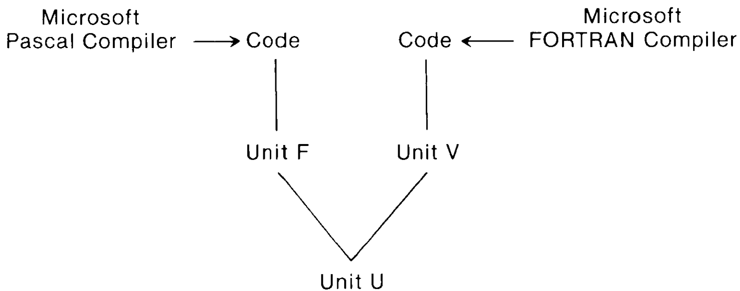


Figure 10.2. The Unit U Interface

The file system uses the following naming convention for public linker names:

1. All linker globals are six alphabetic characters, ending with QQ. (This helps to avoid conflicts with your program global names.)

2. The fourth letter indicates a general class, where:
   a. xxxFQQ is part of the generic MS-Pascal file unit
   b. xxxVQQ is part of the generic MS-FORTRAN file unit
   c. xxxUQQ is part of the operating system interface unit

File system error conditions may be:

1. detected at the lower Unit U level
2. detected at the higher Unit F or V level
3. undetected

When a Unit U routine detects an error, it sets an appropriate flag in the FCB and returns to the calling Unit F or V routine. When Unit F or V detects an error or discovers Unit U has detected one, it takes one of two possible actions:

1. An immediate runtime error message is generated, and the program terminates.
2. Unit F or V returns to the calling program if error trapping has been set (in MS-Pascal with the TRAP flag, in MS-FORTRAN with the ERR=nnn clause).

Units F and V will not pass a file with an error condition to a Unit U routine. For some access methods, certain file operations may lead to an undetected error, such as reading past the end of a record (this condition has undefined results). Runtime errors that cause a program termination use the standard error-handling system, which gives the context of the error and provides entry to the target debugging system.

10.3 Runtime Architecture

The remainder of this chapter describes several topics related to the runtime structure of MS-FORTRAN and MS-Pascal, with mention of differences where they exist.

10.3.1 Runtime Routines

MS-FORTRAN and MS-Pascal runtime entry points and variables conform to the same naming convention: all names are six characters, and the last three are a unit identification letter followed by the letters "QQ". Table 10.2 shows the current unit identifier suffixes.

Table 10.2

Unit Identifier Suffixes

Suffix	Unit Function
AQQ	Complex real
BQQ	Compile time utilities
CQQ	Encode, decode
DQQ	Double precision real
EQQ	Error handling
FQQ	MS-Pascal file system
GQQ	Generated code helpers
HQQ	Heap allocator
IQQ	Generated code helpers
JQQ	Generated code helpers
KQQ	FCB definition
LQQ	STRING, LSTRING
MQQ	Reserved
NQQ	Long integer
OQQ	Other miscellaneous routines
PQQ	Pcode interpreter
QQQ	Reserved
RQQ	Real (single precision)
SQQ	Set operations
TQQ	$FLOATCALLS interface
UQQ	Operating system file system
VQQ	MS-FORTRAN file system
WQQ	Reserved
XQQ	Initialize/Terminate
YQQ	Special utilities
ZQQ	Reserved

10.3.2 Memory Organization

Memory on the 8086 is divided into segments, each containing up to 64K bytes. The relocatable object format and MS-LINK also put segments into classes and groups. All segments with the same class name are loaded next to each other. All segments with the same group name must reside in one area up to 64K long; that is, all segments in a group can be accessed with one 8086 segment register.

MS-FORTRAN and MS-Pascal both define a single group, named DGROUP, which is addressed using the DS or SS segment register. Normally, DS and SS contain the same value, although DS may be changed temporarily to some other segment and changed back again. SS is never changed. The segment registers always contain abstract "segment values" and the contents are never examined or operated on. This provides compatibility with the Intel 80286 processor. Long addresses, such as MS-Pascal ADS variables or MS-FORTRAN named common blocks, use the ES segment register for addressing.

Memory is allocated within DGROUP for the stack, the heap, and all static variables and constants which reside in memory. The segment addresses for MS-FORTRAN common blocks and $LARGE arrays are also allocated within DGROUP, but the common blocks and $LARGE arrays themselves reside outside of the default data segment.

Memory in DGROUP is allocated from the top down; that is, the highest addressed byte has DGROUP offset 65535, and the lowest allocated byte has some positive offset. This allocation means offset zero in DGROUP may address a byte in the code portion of memory, in the operating system below the code, or even below absolute memory address zero (in the latter case the values in DS and SS are "negative").

DGROUP has two parts:

1. a variable length lower portion containing the heap and the stack.
2. a fixed length upper portion containing static variables, constants, and the addresses for common blocks and $LARGE arrays.

After your program is loaded, during initialization (in ENTX6L), the fixed upper portion is moved upward as much as possible to make room for the lower portion. If there is enough memory, DGROUP is expanded to the full 64K bytes; if there is not enough for this, it is expanded as much as possible.

Figure 10.3 illustrates this memory organization.

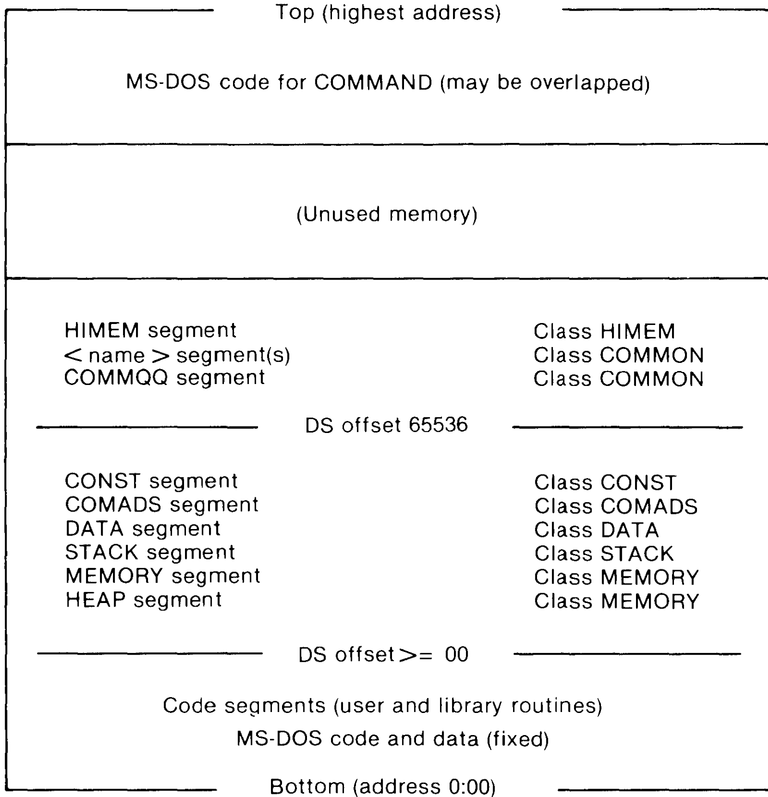


Figure 10.3 Memory Organization

The following paragraphs describe memory contents, starting at the bottom (address zero), when an MS-FORTRAN or MS-Pascal program is running. Addresses are shown in "segment:offset" form.

1. 0000:0000

   The beginning of memory on an 8086 system contains interrupt vectors, which are segmented addresses. Usually the first 32 to 64 are reserved for the operating system. Following these vectors is the resident portion of the operating system (MS-DOS in this case).

   MS-DOS provides for loading additional code above it, which remains resident and is considered part of the operating system as well. Examples of resident additional code are special device drivers for peripherals, a print spooler, or the debugger.

2. BASE:0000

   Here BASE means the starting location for loaded programs, sometimes called the transient program area. When you invoke an MS-FORTRAN or MS-Pascal program, loading begins here. The beginning of your program contains the code portion, with one or more code segments. These code segments are in the same order as the object modules given to the linker, followed by object modules loaded from libraries.

3. DGROUP:LO

   The DGROUP data area contains the following:

Segment	**Class**	**Description**
HEAP	MEMORY	Pointer variables, some files
MEMORY	MEMORY	(not used, Intel compatible)
STACK	STACK	Frame variables and data
DATA	DATA	Static variables
COMADS	COMADS	Address of other data segments; COMMONs, "long" and $LARGE arrays
CONST	CONST	Constant data

   The stack and the heap grow toward each other, the stack downward and the heap upward.

4. DGROUP:TOP

   TOP means 64K bytes (4K paragraphs) above DGROUP: 0000 of DGROUP:LO and just across the DGROUP segment boundary. Common blocks, "long" arrays and $LARGE arrays start here and are allocated to separate segments and where necessary, across segment boundaries. These additional segments for data items are referenced by three items: a segment name which was derived from the common block or array names provided with the MS-FORTRAN program, the class name COMMON, and a segment address pointer which resides in the COMADS class of DGROUP:LO. All references to common block, "long" array or $LARGE array component variables use offsets from this address.

5. HIMEM:0000

   The segment named HIMEM (class HIMEM) gives the highest used location in the program. The segment itself contains no data, but its address is used during initialization. Available memory starts here and can be accessed with MS-Pascal ADS variables.

6. COMMAND

   MS-DOS keeps its command processor (the part of itself which does COPY, DIR, and other resident commands) in the highest location in memory possible. Your MS-FORTRAN or MS-Pascal program may need this memory area in order to run. If so, the command processor is overwritten with program data. When your program finishes, the command processor is reloaded from the file COMAND.COM on the default drive.

   In some circumstances, the check may result in a message appearing on your screen telling you to insert a disk that contains the appropriate file, COMAND.COM. You can avoid this delay by making sure that COMAND.COM is on the disk in the default drive when the program ends.

10.3.3 Initialization and Termination

Every executable file contains one, and only one, starting address. As a rule, when MS-FORTRAN or MS-Pascal object modules are involved, this starting address is at the entry point BEGXQQ in the module ENTX. For some versions, the name ENTX may be appended with other letters. However, the name of the module always begins with the four letters "ENTX". An MS-FORTRAN or MS-Pascal program (as opposed to a module or implementation) has a starting address at the entry point ENTGQQ. BEGXQQ calls ENTGQQ.

The following discussion assumes that an MS-FORTRAN or MS-Pascal main program along with other object modules is loaded and executed. However, you can also link a main program in assembly or some other language with other object modules in either MS-FORTRAN or MS-Pascal. In this case, some of the initialization and termination done by the ENTX module may need to be done elsewhere.

When a program is linked with the runtime library and execution begins, several levels of initialization are required. The levels are the following:

1. machine level initialization
2. program level initialization
3. unit level initialization

The general scheme is shown in Figure 10.4.

ENTX Module

```
BEGXQQ:     Set stackpointer, framepointer
            Initialize PUBLIC variables
            Set machine-dependent flags,
                registers, and other values
            Call INIX87
            Call INIUQQ
            Call BEGOQQ
            Call ENTGQQ {Execute program}

ENDXQQ:     {Terminations come here}
            Call ENDOQQ
            Call ENDYQQ
            Call ENDUQQ
            Call ENDX87
            Exit to operating system
```

INTR Module

```
INIX87:     Real processor initialization
ENDX87:     Real processor termination
```

Unit U

```
INIUQQ:     Operating system specific
                file unit initialization
ENDUQQ:     Operating system specific
                file unit termination
```

MISO Module

```
BEGOQQ:     (Other user initialization)
ENDOQQ:     (Other user termination)
```

Program Module

```
ENTGQQ:     Call INIVQQ
            If $ENTRY on, CALL ENTEQQ
            Initialize static data
            Initialize units
            FOR program parameters DO
                Call PPMFQQ
            Execute program
            If $ENTRY on, CALL EXTEQQ
```

Figure 10.4 Microsoft FORTRAN Program Structure

10.3.3.1 Machine Level Initialization

The entry point of an MS-FORTRAN load module is the routine BEGXQQ, in the module ENTX (the module may also be called ENTX8, ENTX6M, etc.).

BEGXQQ does the following:

1. It moves constant and static variables upward (as described in Section 10.3.2, "Memory Organization"), creating a gap for the stack and the heap.
2. It sets the framepointer to zero.
3. It initializes a number of public variables to zero or NIL. These include:

RESEQQ,	machine error context
CSXEQQ,	source error context list header
PNUXQQ,	initialized unit list header
HDRFQQ,	MS-Pascal open file list header
HDRVQQ,	MS-FORTRAN open file list header

4. It sets machine-dependent registers, flags, and other values.
5. It sets the heap control variables. BEGHQQ and CURHQQ are set to the lowest address for the heap: the word at this address is set to a heap block header for a free block the length of the initial heap. ENDHQQ is set to the address of the first word after the heap. The stack and the heap grow together, and the public variable STKHQQ is set to the lowest legal stack address (ENDHQQ, plus a safety gap).
6. It calls INIX87, the real processor initializer. This routine initializes an 8087 or sets 8087 emulator interrupt vectors, as appropriate.
7. It calls INIUQQ, the file unit initializer specific to the operating system. If the file unit is not used and you don't want it loaded, a dummy INIUQQ routine that just returns must be loaded instead.

8. It calls BEGOQQ, the escape initializer. In a normal load module, an empty BEGOQQ that only returns is included. However, this call provides an escape mechanism for any other initialization. For example, it could initialize tables for an interrupt-driven profiler or a runtime debugger.
9. It calls ENTGQQ, the entry point of your MS-FORTRAN program.

10.3.3.2 Program Level Initialization

Your main program continues the initialization process. First, the language specific file system is called—INIVQQ for MS-FORTRAN, or INIFQQ for MS-Pascal. Both are parameterless procedures.

If the main program is in MS-FORTRAN, and MS-Pascal file routines will be used, INIFQQ must be called to initialize the MS-Pascal file system. If the main program is in MS-Pascal, and MS-FORTRAN file routines will be used, INIVQQ must be called to initialize the MS-FORTRAN file system. MS-FORTRAN main programs automatically call INIVQQ; MS-Pascal main programs automatically call INIFQQ. To avoid loading the file system, you must provide an empty procedure to satisfy one or both of these calls.

If $DEBUG has been set, ENTEQQ is then called to set the source error context.

10.3.3.3 Unit Level Initialization

The information in this section is generally useful only if you have replaced a portion of the FORTRAN runtime library with runtime code of you own, or if you have linked MS-FORTRAN program units with MS-Pascal compilands.

Unit initialization is much like user program initialization. The following actions occur:

1. error context initialization if $ENTRY metacommand was on during compilation
2. variable (file) initialization
3. unit initialization for USES clause
4. user's unit initialization

Calls to initialize a unit may come from more than one unit. The unit interface has a version number, and each initialization call must check that the version number in effect when the unit was used in another compilation is the same as the version number in effect when the unit implementation itself was compiled. Except for this, unit initialization calls after the first one should have no effect; i.e., a unit's initialization code should be executed only once. Both version-number checking and single, initial-code execution are handled with code automatically generated at the start of the body of the unit. This has the effect of:

```
IF INUXQQ (USEVERSION, OWNVERSION, INITREC, UNITID)
THEN RETURN
```

The interface version number used by the compiland using the interface is always passed as a value parameter to the implementation initialization code. This is passed as "useversion" to INUXQQ. The interface version number in the implementation itself is passed as "ownversion" to INUXQQ. INUXQQ generates an error if the two are unequal.

INUXQQ also maintains a list of initialized units. INUXQQ returns true if the unit is found in the list, or else puts the unit in the list and returns false. The list header is PNUXQQ. A list entry passed to INUXQQ as "initrec" is initialized to the address of the unit's identifier (unitid), plus a pointer to the next entry.

User modules (and uninitialized implementations of units) may have initialization code, much like a program and unit implementation's initialization code, but without user initialization code or INUXQQ calls.

The initialization call for a module or uninitialized unit cannot be issued automatically. When the module is compiled, a warning is given if an initialization call will be required (i.e., if there are any files declared or USES clauses). To initialize a module, declare the module name as an external procedure and call it at the beginning of the program.

10.3.3.4 Program Termination

Program termination occurs in one of three ways:

1. The program may terminate normally, in which case the main program returns to BEGXQQ, at the location named ENDXQQ.
2. The program may terminate due to an error condition, either with a user call or a runtime call to an error handling routine. In either case, an error message, error code, and error status are passed to EMSEQQ, which does whatever error handling it can and calls ENDXQQ.
3. ENDXQQ may be called directly.

ENDXQQ first calls ENDOQQ, the escape terminator, which normally just returns to ENDXQQ. Then ENDXQQ calls ENDYQQ, the generic file system terminator. ENDYQQ closes all open MS-Pascal and MS-FORTRAN files, using the file list headers HDRFQQ and HDRVQQ. ENDXQQ calls ENDUQQ, the operating system specific file unit terminator. Finally, ENDXQQ calls ENDX87 to terminate the real number processor (8087 or emulator). As with INIUQQ, INIFQQ, and INIVQQ, if your program requires no file handling, you will need to declare empty parameterless procedures for ENDYQQ and ENDUQQ.

As mentioned, the main initialization and termination routines are in module ENTX. Procedures for BEGOQQ and ENDOQQ are in module MISO. ENDYQQ is in module MISY.

10.3.4 Error Handling

Runtime errors are detected in one of four ways:

1. The user program calls EMSEQQ.
2. A runtime routine calls EMSEQQ.
3. An error checking routine in the error module calls EMSEQQ.
4. An internal helper routine calls an error message routine in the error unit that, in turn, calls EMSEQQ.

Handling an error detected at runtime usually involves identifying the type and location of the error and then terminating the program, or, with ERR= in an I/O statement, returning to the calling MS-FORTRAN procedure.

The error type has three components:

1. a message
2. an error number
3. an error status

The message describes the error, and the number can be used to look up more information (see Appendix C, "Error Messages," in the *Microsoft FORTRAN Reference Manual*). In MS-FORTRAN, the error status value is used for special purposes and has no significance for the user. In MS-Pascal, the error status value is undefined.

Table 10.3 shows the general scheme for error code numbering.

Table 10.3

Error Code Classification

Range	Classification
1- 999	Front end errors
1000-1099	Unit U file system errors
1100-1199	Unit F file system errors
1200-1299	Unit V file system errors
1300-1999	Reserved
2000-2049	Heap, stack, memory
2050-2099	Ordinal and long integer arithmetic
2100-2149	REAL*4 and REAL*8 arithmetic
2150-2199	Structures, sets, and strings
2200-2399	Reserved
2400-2449	Pcode interpreter
2450-2499	Other internal errors
2500-2999	Reserved

An error location has two parts:

1. the machine error context
2. the source program context

The machine error context is the program counter, stackpointer, and framepointer at the point of the error. The program counter is always the address following a call to a runtime routine (e.g., a return address). The source program context is optional; it is controlled by the $DEBUG metacommand. If $DEBUG is in effect, the program context consists of:

1. the source filename of the compiland containing the error
2. the name of the routine in which the error occurred
3. the listing line number of the first line of the statement

10.3.4.1 Machine Error Context

Runtime routines are compiled by default with the $RUNTIME metacommand set. This causes special calls to be generated at the entry and exit points of each runtime routine. The entry call saves the context at the point where a runtime routine is called in the user program. This context consists of the framepointer, stackpointer, and program counter. As a consequence of this saving of context, if an error occurs in a runtime routine, the error location is always in the user program. This is true even if runtime routines call other runtime routines. The exit call that is generated restores the context.

The runtime entry helper, BRTEQQ, uses the runtime values shown in Table 10.4.

Table 10.4

Runtime Values in BRTEQQ

Value	Description
RESEQQ	Stackpointer
REFEQQ	Framepointer
REPEQQ	Program counter offset
RECEQQ	Program counter segment

The first thing that BRTEQQ does is examine RESEQQ. If this value is not zero, the current runtime routine was called from another runtime routine and the error context has already been set, so it just returns. If RESEQQ is zero, however, the error

context must be saved. The caller's stackpointer is determined from the current framepointer and stored in RESEQQ. The address of the caller's saved framepointer and return address (program counter) in the frame is determined. Then the caller's framepointer is saved in REFEQQ. The caller's program counter (i.e., BRTEQQ's caller's return address) is saved: the offset in REPEQQ and the segment (if any) in RECEQQ.

The runtime exit helper, ERTEQQ, has no parameters. It determines the caller's stackpointer (again, from the framepointer) and compares it against RESEQQ. If these values are equal, the original runtime routine called by your program is returning, so RESEQQ is set back to zero.

EMSEQQ uses RESEQQ, REFEQQ, REPEQQ, and RECEQQ to display the machine error context.

10.3.4.2 Source Error Context

Giving the source error context involves extra overhead, since source location data must be included in the object code in some form. Currently, this is done with calls which set the current source context as it occurs. These calls can also be used to break program execution as part of the debug process. The overhead of source location data, especially line number calls, can be significant. Routine entry and exit calls, while requiring more overhead, are much less frequent, so the overhead is less.

The procedure entry call to ENTEQQ passes two VAR parameters: the first is an LSTRING containing the source filename; the second is a record that contains the following:

1. the line number of the procedure
2. the subroutine or function identifier

The filename is that of the compiland source (e.g., the main source filename, not the names of any $INCLUDE files). The procedure identifier is the full identifier used in the source, not the linker name. The line number is the first executable statement in the procedure.

Entry and exit calls are also generated for the main program, in which case the identifier is the program name.

The procedure exit call to EXTEQQ does not pass any parameters. It pops the current source routine context off a stack maintained in the heap.

The line number call to LNTEQQ passes a line number as a value parameter. The current line number is kept in the public variable CLNEQQ. Since the current routine is always available, the compiland source filename and routine containing the line are available along with the line number. Line number calls are generated just before the code in the first statement on a source line. The statement can, of course, be part of a larger statement.

Most of the error handling routines are in modules ERRE and FORE. The source error context entry points ENTEQQ, EXTEQQ, and LNTEQQ are in the debug module, DEBE.

10.4 Floating-Point Operations

By default, the Microsoft FORTRAN Compiler generates calls to a real number math package to carry out floating-point operations. This gives the best tradeoff between runtime performance, code size and flexibility. The real number math package performs floating-point arithmetic according to the proposed IEEE real math standard, using an 80-bit internal form, irrespective of the precision of the operands, and is provided in the standard math library, MATH.LIB. (MATH.LIB will be searched by default if you link with just FORTRAN.LIB.)

The real math package is also compatible with the Intel® 8087™ numeric processor. When you run your program on a machine with an 8087 installed, the real math package, which "emulates" the 8087, automatically uses the processor to carry out the arithmetic. This compatibility means that your programs will give the same, very accurate, results whether they run on a machine with an 8087 installed or in another processing environment. (In cases where the real math package is emulating the 8087, some transcendental functions may give different results, but the differences are very slight).

MS-FORTRAN also provides options that allow you to tailor your program for performance and size on specific system configurations. Specifically, you can choose to have in-line 8087 instructions generated to perform floating-point operations, you can select a math package optimized for performance but which gives less accurate results, or you can eliminate the math package altogether if you know an 8087 will always be present when your program will be run.

Don't forget that using these options will affect the portability of your program and the consistency of its results.

10.4.1 The $NOFLOATCALLS Metacommand Option

The $NOFLOATCALLS metacommand directs the compiler to generate in-line instruction "skeletons" for floating-point operations. With $NOFLOATCALLS in your source code and the standard versions of FORTRAN.LIB and MATH.LIB linked into your program, fixups in FORTRAN.LIB will cause the linker to transform the in-line instruction skeletons into software interrupts and control information. The interrupts and control information will be fielded at execution time by an emulator (software math package).

When you run your program, the first time each such interrupt is executed, the emulator gets control. If you have an 8087 installed, the emulator will overwrite the interrupt and control information with the equivalent 8087 instruction and re-execute it. This and all subsequent executions of the instruction will be carried out by the 8087. If you do not have an 8087, the emulator will use the control information to carry out a software equivalent of 8087 instruction processing. This will occur every time the instruction is executed.

The in-line instructions typically require half as much code as the equivalent call sequence and also permit additional optimizations to be performed. Otherwise, nonfloating-point operations are unaffected, and the total reduction in code size will usually be between 10 and 30 percent.

This option provides the most efficient execution if you have an 8087 installed. However, if you do not, the interrupt mechanism and processing of control information that occurs every time an instruction is emulated is time consuming and imposes a considerable overhead on the fundamental arithmetic operations. The

overhead may be up to 25 percent on the simpler instructions (for example, FADD). But for the same reasons that reduced the impact of this option on code size, you should expect the overall overhead to be somewhat less than this, depending on the mix of instructions.

The basic operations are, in fact, carried out by the same code that supports the calls to the emulator and using this option will have no effect on your program's results. Also, you can freely mix modules compiled with $NOFLOATCALLS with those compiled with the default option. You can even use a second metacommand, $FLOATCALLS, to switch between modes within the same module or even subroutine. However, this practice might not take effect exactly where you specified it, because optimizations may group statements or reorder code.

Important

> You cannot use this option in any modules that will be linked with the alternate math pack ALTMATH.LIB. You will get linker errors if you try. You will get a compilation error if you use $NOFLOATCALLS when $DECMATH is in effect.

10.4.2 The Alternate Math Pack Option

The IEEE math standard as supported by the emulator is complicated, and the emulator, as a result, will contribute about 6.5K bytes to your program. Also, arithmetic to 80-bit precision is much more time consuming than the minimum required to provide reasonable accuracy for 32-bit or even 64-bit floating-point numbers.

If you do not require consistency with the 8087, or if the speed of your program is more important than accuracy, you can use the "Alternate Math Pack." This is a more traditional floating-point support package. Its interface is compatible with the $FLOATCALLS interface to the emulator, but it is optimized for speed. The results of your calculations will be less accurate, particularly for single precision arithmetic, and will, in general, be slightly different than those produced using the 8087 or the emulator. However, basic operations will be typically at least twice as fast, and, if you

don't have an 8087, programs that do a lot of floating-point arithmetic will run much faster. If you do have an 8087 and use this option, your programs *will not* use the coprocesssor and will run slower than they would have had you linked in the standard math library.

The Alternate Math Pack assumes a much simpler model for floating-point arithmetic than the IEEE standard, although the external representation of real values is the same. For example, all overflow, divide-by-zero and other exceptions that would result in a NAN (Not A Number) in the IEEE model, will cause an error exit in the Alternate Math Pack model. Also, unlike the 8087 and emulator models that assume (and support) an infinite stack, the Alternate Math Pack assumes a fixed stack with a limited number of entries. This means that highly recursive functions may overflow the stack and give an error.

You select the Alternate Math Pack by linking with the library ALTMATH.LIB, which is provided with the compiler. This library contains only the Alternate Math Pack, and the remainder of the runtime is obtained from FORTRAN.LIB. (See Section 6.1.1.1, "Standard Runtime Libraries," for a review of the procedure for linking auxiliary libraries.)

Note

You can change the default math package by renaming ALTMATH.LIB to be MATH.LIB.

10.4.3 No Emulation Option

As mentioned above, the emulator contributes about 6.5K bytes to the size of your program. If you have an 8087 installed, its only purpose is to translate your emulated instructions into actual 8087 instructions. You can eliminate the emulator altogether if you know that your program will only run on machines that have an 8087.

To eliminate the emulator, you link in the object module 8087.LIB, provided with the compiler. This replaces the emulator and fixes up the in-line instruction skeletons to actual 8087 instructions at link time. $FLOATCALLS interfaces are also provided. These use the 8087 to carry out the operation, so that you can use 8087.LIB whether or not you have used $FLOATCALLS.

Note

You change the default math package by renaming 8087.LIB to MATH.LIB.

10.4.4 Decimal Math Option

Microsoft FORTRAN supports an alternative floating-point format in which decimal floating-point numbers up to 14 digits and within a limited exponent range can be represented exactly. The results of the operations on the numbers in this format are also represented exactly if they are in the allowable range. This option is particularly useful in business and financial applications where exact results are important.

You select the decimal format by using the $DECMATH metacommand in all of your program units that use floating-point. You must link with DECMATH.LIB to support this format. Decimal floating-point and IEEE floating-point are *not* compatible.

Note

You cannot use $NOFLOATCALLS if you have specified $DECMATH.

10.5 MS-DOS 2.0 Issues

This version of the Microsoft FORTRAN Compiler is essentially an MS-DOS 1.25 compiler. This means that Microsoft FORTRAN and programs compiled by it will run on both versions of MS-DOS, but cannot take advantage of special features of MS-DOS 2.0, such as pathnames.

However, if you know that your program (not compiler) will only be required to run under MS-DOS 2.0, you can link with the special version of the FORTRAN file system, DOS2FOR.LIB, which contains the interface to the MS-DOS 2.0 file system. The modules contained in DOS2FOR.LIB provide the interface described in Section 10.2, "An Overview of the File System," of this *User's Guide.*

Programs linked with DOS2FOR.LIB, will give the runtime message

```
Incorrect DOS version
```

when run on earlier versions of MS-DOS.

10.5.1 Exit Status Available to 2.0 MS-DOS

The compiler supplies an exit status to 2.0 MS-DOS that can be accessed via the "IF ERRORLEVEL n" batch command. The values returned by the compiler are:

n Value	Meaning
0	No warnings for errors issued
2	Warnings were issued
4	Fatal errors encountered

The value of the global word DOSEQQ (defined in module ENTX) is passed to MS-DOS and this becomes the argument to ERROR-LEVEL. DOSEQQ may be set to any error code desired by the user.

Note

You will have to use MS-Pascal or MS-Macro Assembler to set the value of DOSEQQ.

Appendices

A Differences From Earlier Versions of Microsoft FORTRAN 133

B Microsoft FORTRAN File Control Block 137

C Real Number Conversion Utilities 139

D Structure of External Microsoft FORTRAN Files 141

E Microsoft FORTRAN Scratch Files 143

F Customizing i8087 Interrupts 145

G Exception Handling for 8087 Math 151

H Microsoft LINK Error Messages 157

Appendix A
Differences From Earlier Versions of Microsoft FORTRAN

The significant differences between the 3.20 and 3.0 releases of Microsoft FORTRAN are:

- The PARAMETER statement has been added, including constant expressions. (See Chapter 3, "Statements," in the *Reference Manual.*) Constants may be of LOGICAL type, CHARACTER type, or any numeric type. Expressions may only be INTEGER or LOGICAL (REAL and COMPLEX constant arithmetic is not supported, nor are character operators.) After an identifier has been declared by the PARAMETER statement, it may be used exactly as a constant of the same type (for example, it may appear in constant expressions).
- Integer constant expressions are also now allowed in data declarations, for example:

  ```
  'CHARACTER foo*(10*2)', or 'REAL*(6-2)r1'
  ```

Note

Only integer and logical expressions are allowed; real, complex and character expressions are not supported.

- The SAVE statement now accepts full language syntax. (See Chapter 3, "Statements," in the *Reference Manual.*)
- The .EQV. and .NEQV. logical operators have been added; they have assumed the position of lowest precedence. (See Section 2.5.6, "Logical Expressions," in the *Reference Manual.*)

- The DATA statement will check for type compatibilty between the variable being initialized and the constant. It will carry out coercions as specified in the full language. The DATA statement may be interspersed with statement function statements and executable statements.
- The BLOCK DATA statement is supported although not all restrictions on it are checked.
- Alternate returns have been added. (See Section 3.2.5, "The CALL statement," in the *Reference Manual.*)
- The COMPLEX*8 data type has been added, as well as the specific and generic type conversion intrinsics that support it as defined by the full FORTRAN 77 language. COMPLEX*16 has also been added as a Microsoft FORTRAN extension. New specific intrinsics (see Table 5.1, "Intrinsic Functions," in the *Reference Manual*) have been added to support the COMPLEX*16 type.
- Arrays and common blocks longer than 64K are now supported; short (less than 64K) arrays may be declared to be outside DGROUP; and blank common is no longer within DGROUP
- The metacommands $LARGE and $NOTLARGE have been added (see Chapter 6, "Metacommands," in the *Reference Manual.*)
- Generic intrinsic functions as defined by the full language have been added. (See Table 5.1, "Intrinsic Functions," in the *Reference Manual.*)
- The edit descriptors T, TR, TL, S, SP, SS and " : " have been added. (See Section 3.1.4, "The FORMAT Statement," and Section 4.4.2, "Edit Descriptors for the Format Statement," in the *Reference Manual.*) You may not use the T descriptors to reposition to the left once you have positioned beyond character position 128, since the output data are held in a buffer of this size.
- The full language INQUIRE statement has been added. (See Chapter 3, "Statements," in the *Reference Manual.*)
- The IOSTAT=<iocheck> specifier has been added to the CLOSE, OPEN, READ, and WRITE statements.

<iocheck> is an integer variable or integer array element that becomes defined as: (1) a zero if no error or end-of-file conditions are encountered or (2) a processor-dependent positive integer value if an error condition is encountered or (3) a processor-dependent negative integer value if an end-of-file is encountered and no error condition exists.

- New linkers are provided; their use is required. (See Section 4.2, "Alternative Linkers," in this guide for more details.)
- A simple overlay scheme is supported. (See Section 6.3, "The Overlay Linker," in this guide for a discussion of how to overlay your program.)
- The library structure has been revised. (See Section 6.1.1, "Object Modules," in this guide for more information about the standard runtime library and the auxiliary libraries that it supports.)
- An alternative math package, optimized for speed, is provided. (See Chapter 10, "Advanced Topics," in this guide for the details about ALTMATH.LIB and the Alternate Math Package.)
- A "Decimal" math package is provided. (See Section 10.4.4, "Decimal Math Option," in this guide for the details about DECMATH.LIB and the Decimal Math Package.)
- $FLOATCALLS is now on by default. It is a metacommand which directs the compiler to generate calls to real math support routines. (See Section 10.4, "Floating-Point Operations," in this guide for more information on this metacomand.)
- An interface to the MS-DOS 2.0 file system is provided. (See Section 4.1, "MS-DOS 2.0 Interface Library," in this guide for the description of DOS2FOR.LIB.)
- NULR7.OBJ is no longer provided. Math routines are only included if math operations are performed.
- COMMQQ is no longer allocated to the default data segment. (See Section 10.3.2, "Memory Organization," in this guide for a review of the Microsoft Pascal memory model.)
- Unit numbers in the range -32767 to 32767 are accepted.
- The list-directed output line size is 80 columns.
- CHARACTER-typed functions are supported.

Appendix B
Microsoft FORTRAN File Control Block

This appendix lists the complete file control block specification for this version of the MS-FORTRAN runtime system. The underlying data type is an MS-Pascal record. Numbers in square brackets give the byte offset for each field of the file control block.

```
{MS-Pascal / MS-FORTRAN FCB Declaration Include File}

INTERFACE; UNIT
  FILKQQ (FCBFQQ, FILEMODES, SEQUENTIAL,
                              TERMINAL, DIRECT, BUFFER_SIZE);

const
    BUFFER_SIZE = 512;

TYPE
FILEMODES = (SEQUENTIAL, TERMINAL, DIRECT);

FCBFQQ = RECORD        {byte offsets start every field comment}
{fields accessible by Pascal user as <file variable>.<field>}
TRAP:  BOOLEAN;        {00 Pascal user trapping errors if true}
ERRS:  WRD(0) . . 15;  {01 error status, set only by all units}
MODE:  FILEMODES;      {02 user file mode; not used in unit U}
MISC:  BYTE;           {03 pad to word bound, special user use}
{fields shared by units F, V, U; ERRC / ESTS are write-only}
ERRC:  WORD;           {04 error code, error exists if nonzero}
                           {1000..1099: set for unit U errors}
                           {1100..1199: set for unit F errors}
                           {1200..1299: set for unit V errors}
ESTS:  WORD;           {06 error specific data usually from OS}
CMOD:  FILEMODES;      {08 system file mode; copied from MODE}
{fields set / used by units F and V, and read-only in unit U}
TXTF:  BOOLEAN;        {09 true: formatted / ASCII / TEXT file}
                           {false: not formatted / binary file}
SIZE:  WORD;           {10 record size set when file is opened}
                           {DIRECT: always fixed record length}
                           {others: max buffer variable length}
```

```
IERF:   BOOLEAN;         {12 Unit U Incomplete End Of Record}
                             {Set false by opnuqq and}
                             {pccuqq, and true by peruqq. Thus}
                             {if true in wefuqq, it means that}
                             {there is an incomplete line, and}
                             {pccuqq should be called to flush}
                             {it. Only applies to terminal files}
MISA:   BYTE;            {13 Used to keep alignment with old misa}
OLDF:   BOOLEAN;         {14 true: must exist before open; RESET}
                             {false: can create on open; REWRITE}
INPT:   BOOLEAN;         {15 true: user is now reading from file}
                             {false: user is now writing to file}
RECL:   WORD;            {16 DIRECT record number, lo order word}
RECH:   WORD;            {18 DIRECT record number, hi order word}
USED:   WORD;            {20 number bytes used in current record}
{fields used internally by units F or V not needed by unit U}
LINK:   ADR OF FCBFQQ;   {22 DS offset address of next open file}
BADR:   ADRMEM;          {24 F: DS offset address for buffer var}
TMPF:   BOOLEAN;         {26 F: is a temp file; delete on CLOSE}
FULL:   BOOLEAN;         {27 F: buffer variable lazy eval status}
UNFM:   BOOLEAN;         {28 V: for unformatted binary file mode}
OPEN:   BOOLEAN;         {29 F: file opened (by RESET / REWRITE)}
FUNT:   INTEGER;         {30 V: FORTRAN unit number (1 to 32767)}
ENDF:   BOOLEAN;         {32 V: last I/O statement was a ENDFILE}
{fields set / used by unit U, and read-only in units F and V}
REDY:   BOOLEAN;         {33 buffer ready if true; set by F / U}
BCNT:   WORD;            {34 number of data bytes actually moved}
EORF:   BOOLEAN;         {36 true if end of record read, written}
EOFF:   BOOLEAN;         {37 end of file flag set after EOF read}
        {unit U (operating system) information starts here}
{****************************************************************************}

FILE_NAME:      : ^STRING;       { points to file name }
FDSCP           : INTEGER;       { actual ZEUS file number }
PREDEFINED      : BOOLEAN;       {* True if file is a device. *}
FNER            : BOOLEAN;       {* True if File name error. *}
BEGIN_BUFFER: INTEGER;           {* Start loc of buffer. *}
END_BUFFER      : INTEGER;       {* Stop loc of buffer. *}
BUFFER          : STRING(512);   {* Internal buffering. *}
PADBUF          : STRING(68);    {* Make same size as MS-Dos. *}

{****************************************************************************}

{end of section for unit U specific OS information}
END;
END;
```

Appendix C
Real Number Conversion Utilities

Releases of MS-FORTRAN starting with version 3.0 use the IEEE real number format. Releases of MS-FORTRAN earlier than 3.0 used the Microsoft real number format.

The two formats are not compatible. However, if you need to convert real numbers from one format to the other, you can use the following library routines:

a. Single Precision Reals:

Microsoft to IEEE format

```
SUBROUTINE M2ISQQ (RMS , RIEEE)
```

IEEE to Microsoft format

```
SUBROUTINE 12MSQQ (RIEEE , RMS)
```

RMS and RIEEE are real numbers in Microsoft format and in IEEE format, respectively.

b. Double Precision Reals:

Microsoft to IEEE format

```
SUBROUTINE M2IDQQ (DMS , DIEEE)
```

IEEE to Microsoft format

```
SUBROUTINE 12MDQQ (DIEEE , DMS)
```

DMS and DIEEE are real numbers in Microsoft format and in IEEE format, respectively.

Appendix D

Structure of External Microsoft FORTRAN Files

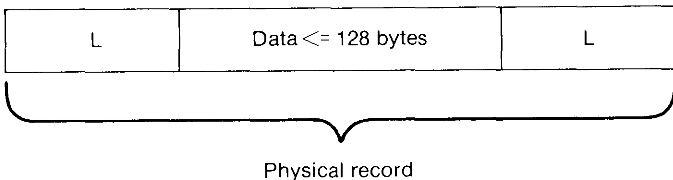
The structure of an external MS-FORTRAN file is determined by its properties. The structures used in MS-FORTRAN are as follows:

1. Formatted sequential files

   Records are separated by carriage return and linefeed (ASCII hex codes 0D and 0A, respectively).

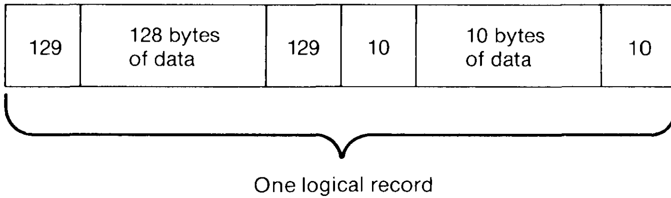
2. Unformatted sequential files

   A logical record is represented as a series of physical records, each of which has the following structure:



Each *L* shown above is a length byte that indicates the length of the data portion of the physical record. The data portion of the last physical record contains MOD (length of logical record, 128) bytes, and the length bytes will contain the exact size of the data portion.

Each of the preceding physical records will contain 128 bytes in the data portion, while the length byte will contain 129. For example, if the size of the logical record is 138:



The first byte of the file is reserved and contains the value 75, which has no other significance.

3. Formatted direct files, unformatted direct files, and binary files

   No record boundaries or any other special characters are used.

Appendix E
Microsoft FORTRAN Scratch File Names

Scratch files are created by the MS-FORTRAN system when no filename is specified in an OPEN statement. Scratch file names look like this:

```
T<u>.TMP
```

<u> is the unit number specified in the OPEN statement.

Appendix F
Customizing i8087 Interrupts

This appendix describes how to customize the i8087 interrupts on your computer system. Before proceeding, you should be familiar with the following:

1. the Intel publication, *iAPX 86/20, 88/20 Numeric Supplement*
2. MS-MACRO, the Microsoft MACRO Assembler
3. DEBUG, the MS-DOS debugger utility

In addition, we recommend that you make backup copies of any of the disks you plan to modify.

To change the way the runtime library processes interrupts, you must use the MS-DOS debugger DEBUG (or a similar utility). Although this utility is intended primarily for debugging assembly language programs, you can also use it to alter the binary contents of any file. You will use this second capability of DEBUG to customize FORTRAN.L87 for a particular hardware configuration.

FORTRAN.L87, the 8087 version of the runtime library, contains the following assembly language structure:

```
i8087control  STRUC
LABX87        DB     '<8087>'  ;48-bit tag
EOIX87        DB     0         ;EOI instruction
PRTX87        DB     0         ;i8259 port number
SHRX87        DB     0         ;Shared interrupt device
INTX87        DB     2         ;i8087 interrupt vector #
INTOFFSET     DW     0         ;
i8087control  ENDS
```

This structure defines the default control values used by the run-time library to handle 8087 interrupts. Each of the elements of the structure is described briefly in the following list:

1. LABX87

   A string label. LABX87 exists solely to locate the other structure fields in the executable binaries and libraries.

2. EOIX87

   The hexadecimal value of the i8259 "end of interrupt" instruction for a particular implementation. To the 8087 interrupt handler supplied by Microsoft, any nonzero value of this byte indicates the presence of an i8259 interrupt controller.

3. PRTX87

   The control port number associated with an i8259, if present.

4. SHRX87

   If nonzero, an indication that the i8087 shares its interrupt vector with another device. In such a case, when the 8087 interrupt handler supplied by Microsoft determines that an interrupt it receives is not an 8087 interrupt, it passes control to the other interrupt device.

5. INTX87

   The interrupt vector number to which the 8087 is connected.

Depending on the setup of your computer system, any or all (or none) of the last four items may require changing. Specifically, you must alter this structure if your hardware configuration meets any of the following criteria:

1. It uses an 8087 interrupt vector number other than 2.
2. It uses an 8259 interrupt controller.
3. The 8087 shares interrupts with another device on the same vector.

The example on the following pages demonstrates how to change all of the interrupt parameters on the 8087. In the example, the following specific changes are made:

1. The 8087 interrupt control block is altered to set EOIX87 to 255 decimal, thus informing the software that an i8259 exists and that its EOI instruction is 255.
2. The i8259 should issue its EOI request through port number 254 (PRTX87).
3. The nonzero value of SHRX87 indicates that the 8087 shares its interrupts with another device.
4. The interrupt vector number of the i8087 was changed to 4.

These values are used merely for the purpose of this sample session. Consult your hardware manual for the values required for your computer system.

For the sake of brevity and clarity, not all of the screen display issued by the debugger is shown in the example, only the parts that apply specifically to this procedure. Also, on most screens, the information shown in lines 4, 7, 8, and 10 will run to 80 columns on an 80-column screen.

Numbers 1 through 13 at the left-hand margin of the sample session do not appear on the screen; rather, they refer to the corresponding numbered comments on the page following the sample session.

See your MS-DOS manual for complete details on using DEBUG.

Sample DEBUG Session to Customize i8087 Interrupts:

```
1.  >

2.  >debug b:fortran.l87

3.  DEBUG-86        version 2.10

4.  >r
    AX=0000       BX=0001       CX=B800       DX=0000
    SP=FFEE       BP=0000       SI=0000       DI=0000
    DS=0AF9       ES=0AF9       SS=0AF9       CS=0AF9
    IP=0100         NV UP DI PL NZ NA PO NC
    0AF9:0100       F0                  LOCK
    0AF9:0101       FD                  STD

5.  >s ds:100       lefff '8087>'

6.  0AF9:2370

7.  >d af9:2370
    0AF9:2370   38   30   38   37   3E   00   00   00   8087>...
                02   00   00   F8   A0   DF   00   02   ...x ..

8.  >d af9:2375
    0AF9:2375   00   00   00-02      00   00   F8   A0   .....x
                DF   00   02                            ..

9.  >e af9:2375
    0AF9:2375     00.ff       00.fe       00.1
    0AF9:2378 02.4

10. >d af9:2375
    0AF9:2375   FF   FE  01-04      00   00   F8   A0   ......x
                  DF   00   02                          ..

11. >w

12. >q

13. >
```

Comments for Sample DEBUG Session

1. MS-DOS prompt.
2. Call DEBUG with FORTRAN.L87.
3. DEBUG utility prompt.
4. Instruct debugger to show 8086 registers.
5. Instruct debugger to search efff bytes beginning at DS:100 for the string '8087>'.

6. String found at 0AF9:2370.
7. Instruct debugger to display the string.
8. Advance to the beginning of the 'i8087control' structure.
9. Instruct the debugger to make the following alterations:

   EOIX87 to FF hex, 255 decimal
   PRTX87 to FE hex, 254 decimal
   SHRX87 to 1 hex
   INTX87 to 4

10. Instruct the debugger to display any changes.
11. Write any changes to the source file.
12. Stop the debugger.
13. MS-DOS prompt returns.

Appendix G
Exception Handling for 8087 Math

The five exceptions to floating-point arithmetic that are required by the IEEE standard are supported by the 8087 coprocessor and the real math support routines. Those which would result in a NAN (Not A Number) error message when enabled, are enabled by default. The others are disabled. They are not affected by the $DEBUG metacommand but are controlled by a STATUS word and a CONTROL word.

1. Invalid Operation—Any operation with a NAN; root(-1), 0*INF, etc.

   Default action. Enabled, gives runtime error 2136.
   Alternate action. Disabled, returns a NAN.

2. Divide by zero—r/0.0.

   Default action. Enabled, gives runtime error 2100.
   Alternate action. Disabled, returns a properly signed INF (infinity).

3. Overflow—Operation results in a number greater than maximum representable number.

   Default action. Enabled, gives runtime error 2101.
   Alternate action. Disabled, returns INF.

4. Underflow—Operation results in a number smaller than smallest valid representable number.

   Default action. Disabled, returns zero.
   Alternate action. Enabled, gives runtime error 2135.

5. Precision—Occurs whenever a result is subject to rounding error.

   Default action. Disabled, returns properly rounded result.
   Alternate action. Enabled, gives runtime error 2139.

G.1 Processing Environment Control

Two memory locations control the 8086 and the 8087 processors. These are called the STATUS word and CONTROL word. The effect of these memory locations is discussed in the following paragraphs.

STATUS word

When one of the exceptional conditions occurs, the appropriate bit in the STATUS word is set. This flag will remain set to indicate that the exception occurred until cleared by the user. If you set the bit in the CONTROL word relating to a given exception, that exception is masked and the operation proceeds with a supplied default. If the bit is unset, any exception of that type generates an error message, halts the operation, and your program will stop. In either case the exception is ORed into the STATUS word.

CONTROL word

In addition to masking exceptional conditions, the CONTROL word is also used to set modes for the internal arithmetic required by the IEEE standard. These are:

Rounding Control

Round to nearest (or even), Up, Down, or Chop

Precision Control

Determines at which bit of the mantissa rounding should take place (24, 53, or 64). Note all results are done to 64 bits regardless of the precision control. It only affects the rounding in the internal form. On storage any result is again rounded to the storage precision.

Infinity Control

Affine mode is the familar + and - INF style of arithmetic. Projective mode is a mode where + and - INF are considered to be the same number. The principal effect is to change the nature of comparisons. (Projective INF does not compare with anything but itself.)

The CONTROL word defaults are currently:

Infinity control = affine

Rounding control = near

Precision control = 64 bits

Interrupt-enable mask = unmasked

Precision mask = masked

Underflow mask = masked

Overflow mask = unmasked

Zerodivide mask = unmasked

Denormalized operand mask = unmasked

Invalid operation mask = unmasked

Special exception handling routines handle stack exceptions and denormal propagation. For these routines to work correctly, the 8087 CONTROL word and auxiliary variables need to be set up as done by LCWRQQ. Use LCWRQQ to revise the 8087 CONTROL word.

Important

Do not alter the 8087 CONTROL word with an FLDCW instruction when using the 8087 with a Microsoft language.

Since the denormal exception is not a part of the IEEE standard, LCWRQQ always alters the user's parameter word to unmask denormals and thus handle them with the Microsoft exception handler. The user cannot affect the handling of denormals with LCWRQQ.

Since stack overflow and underflow are indicated by the 8087 with an invalid exception, the invalid exception bit is also always unmasked by LCWRQQ. However, when an invalid exception occurs and it is not stack overflow or underflow, then the invalid exception bit of the user's control word input to LCWRQQ controls the handling of the exception.

The following list of control words defines the masking settings for the overflow, zerodivide, and invalid operation exceptions that are associated with several optional control words. Control word 4914 specifies the default masking settings that are customary during 8087 operations.

Control word	Overflow	Zerodivide	Invalid
4914 = 1332h	unmasked	unmasked	unmasked
4915 = 1333h	unmasked	unmasked	masked
4918 = 1336h	unmasked	masked	unmasked
4919 = 1337h	unmasked	masked	masked
4922 = 133Ah	masked	unmasked	unmasked
4923 = 133Bh	masked	unmasked	masked
4926 = 133Eh	masked	masked	unmasked
4927 = 133Fh	masked	masked	masked

G.2 Reading and Setting STATUS and CONTROL Values

The values of the STATUS word and CONTROL word can be read and set using the following procedures:

```
C Load Control Word

      SUBROUTINE LCWRQQ (CW)
      INTEGER*2 CW

C sets the control word to the value in CW

C Store Control Word

      INTEGER*2 FUNCTION SCWRQQ
```

C returns the value of the control word

C Store Status Word

```
INTEGER*2 FUNCTION SSWRQQ
```

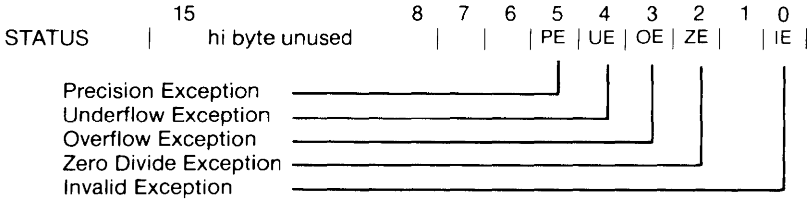
C returns the value of the status word

Printing NANs (Not A Number)

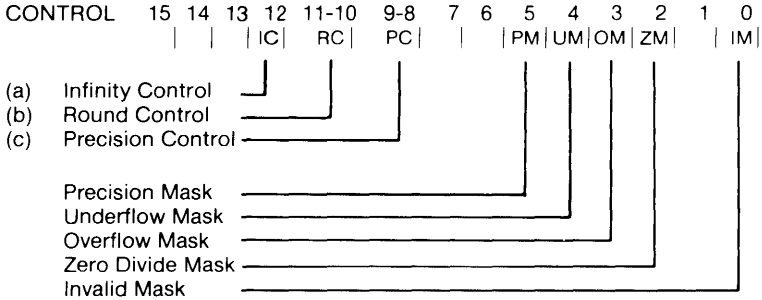
If you disable the above exceptions, you will either get NAN, Infinite, or Indefinite values in your variables. If you print such a value, the output field will contain NAN, INF, or IND padded with periods to the field width. If the output field has less than three spaces, only periods will be printed.

G.3 Formats for the STATUS And CONTROL Words

The bit locations for storing the cumulative record of exceptions are defined in the diagrams that follow.



(All other bits unused, may be either 1 or 0)



(All other bits unused, may be either 1 or 0)

(a) Infinity Control
- 0 = Projective
- 1 = Affine

(b) Round Control
- 00 = Round nearest or even
- 01 = Round down (toward -INF)
- 10 = Round up (toward +INF)
- 11 = Chop (Truncate toward 0)

(c) Precision Control
- 00 = 24 bits of mantissa
- 01 = (reserved)
- 10 = 53 bits of mantissa
- 11 = 64 bits of mantissa

Appendix H
Microsoft LINK Error Messages

Any link error will cause the link session to terminate. After you have found and corrected the problem, you must rerun MS-LINK. Link errors have no code number. See your MS-DOS manual for further information on MS-LINK.

Attempt to access data outside of segment bounds

There is probably a bad object file.

Bad numeric parameter

Numeric value is not in digits.

Cannot open temporary file

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

Dup record too complex

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

Fixup overflow in (name).OBJ, (module name) near 0000 in segment (name)

This message usually indicates that the total amount of static data exceeds the limit of the default data segment. Usually, the user had defined a segment in an assembly language module whose group or class name conflicts with those of the segments generated by the compiler.

Input file read error

There is probably a bad object file.

Invalid object module

An object module(s) is incorrectly formed or incomplete (as when assembly is stopped in the middle).

Symbol defined more than once

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

Program size exceeds capacity of linker

The total size may not exceed approximately 900K bytes.

Reloc table overflow

The program contains more than 12000 long calls (approximately). Typically, this happens in very large programs with debugging turned on ($DEBUG).

Requested stack size exceeds 64K

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

Segment size exceeds 64K

64K bytes is the addressing system limit.

Symbol table overflow

Very many and/or very long names were typed, exceeding the limit of approximately 64K bytes.

Too many external symbols in one module

The limit is 511 external symbols per module.

Too many groups

The limit is 10 groups.

Too many initialized segments

The number of segments may not exceed 255 but the number of initialized segments may not exceed 245.

Too many libraries specified

The limit is 8 libraries.

Too many public symbols

The limit is 1536 public symbols if you specify /MAP

Unresolved externals: <list>

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

VM read error

This is a disk error; it is not caused by the Microsoft Linker.

Warning: No stack segment

None of the object modules specified contains a statement allocating stack space.

Write error in TMP file

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

Write error on run file

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

Index

8086 assembly language, 87
8087 coprocessor, 10
8087 emulation, 116

Add routine, 92 to 96
Addresses, offset, 17
Alternative linkers, 35
ALTMATH.LIB, alternative math package, 38, 39, 125
Arrays
  long (exceed 64K), 71, 72, 73, 134
  passed as arguments, 72
  $LARGE, 72, 73, 110
  memory requirements, 75, 110
Assembler, 14
Assembly language
  data placement, 97
  interface, 56, 91
Auxiliary libraries
  8087.LIB, 38, 58
  ALTMATH.LIB, 38, 58
  DECMATH.LIB, 38, 58
  DOS2FOR.LIB, 35, 58, 128

Back end, 81, 83
Base name, 48
BAT file. *See* Batch file.
Batch file, 67
BEGXQQ, 114, 116

Calling conventions, 55, 87
Calls
  by reference, 91
  long, 89
  short, 89
Character values, 91
CHKDSK program, 62
Code
  relative, 16
  size, limits, 71
COMMON block, 71, 76, 113
Compile time
  definition, 15
  memory limits, 76
Compiler
  invocation, 49 to 51
  optimization, 11
  options, 39, 124
  passes, 21
    See also
      Pass one
      Pass two
      Pass three
  starting, 49
  structure, 12, 101
Compiling large programs, 71, 81
COPY, 9

Data types
  internal representations
    character, 91
    COMPLEX, 90
    decimal, 90
    INTEGER, 89
    LOGICAL, 90
    REAL, 90
    size limits, 71
DECMATH.LIB, decimal math package, 38
Decimal numbers, 90
Definitions, 15
Device drivers, 112
Device name, 45
DGROUP:LO, 110

DGROUP:TOP, 110
DIR, 62
DISKCOPY, 9
Disks
  backup copies, 9
  exchanging, 22, 80
  formatting, 9
  limits, 77
  memory, 77
  provided, 3
  set up, 9
DOS2FOR.LIB, interface library, 35, 58, 128
Dummy subroutines, 83

Emulator library, MATH.LIB, 28, 39, 58, 123, 126
ENDYQQ, 82
ENTER key, 5
ENTGQQ, 114
Error
  code classification, 120
  machine context, 121
  source context, 122
  trapping, 108
Examples
  large program, 81
  program session, 21
Exception handling
  CONTROL word, 152, 154, 155
  divide by zero, 153
  invalid operation , 153
  overflow and underflow, 153
  precision control, 153
  STATUS word, 152, 154, 155
EXE file, 14, 61
  files, 141
  reference, 16
EXTERNAL declarations, 55

File control block, 91, 106, 107, 137
Filenames
  conventions, 45
  default, 47
  defaults, 45
  extensions, 46
  general rules, 47
  spaces within, 37
Files
  backup copies, 9
  batch facility, 67
  compiler-written, 43
  control blocks. *See* File control block
  external, 141
  intermediate, 44
  linker-read, 55
  linker-written, 60
  naming. *See* Filenames.
  NUL or null, 25, 47
  object, 43, 55
  object listing, 47
  run, 61
  scratch, names, 143
  source listing, 43
Floating-point arithmetic compiler options, 37, 123
  emulation, 125
FORTRAN
  learning resources, 5
  standard, subset, 3
FORTRAN.LIB, default runtime library, 21, 28, 35, 58, 59, 124, 126
Frame contents, 87
Framepointer, 88, 116
Front end, 103
Function return values, 91

Hardware configuration, 10
Heap, 112, 116

I/O, 82
Identifier limits, 75
INIVQQ, 82
INPUT file, 107
INTEGER*2, 84

INTEGER*4, 89
Interface, standard, 106
Intermediate files, 44
Internal representations, 89
Interrupts
  i8087, 145
  parameter changes, 147
  vectors, 112

Libraries
  auxiliary, 38, 55, 58
  runtime, 10, 17, 35, 44, 57, 114, 117
Limits
  compile time memory, 75
  complex expressions, 76
  identifiers, 75
  physical, 71
Link time, 15
Linker
  alternative linkers, 35
  defaults, 57
  globals, 107
  linking with pathnames, 35, 59
  listing file, 61
  map, 61
  options, 33
  overlays, 62, 64
  prompts, 57, 80
  public names, 107
Linker switches, 64
  /CPARMAXALLOC:NNNN, 65
  /DISALLOCATE, 65
  /HIGH, 66
  /LINENUMBERS, 65
  /MAP, 65
  /NODEFAULTLIBRARY SEARCH, 65
  /NOGROUPASSOCIATION, 65
  /NOIGNORECASE, 65
  /OVERLAYINTERRUPT: NNNN, 66
  /PAUSE, 66
  /STACK, 66
Linking
  general discussion, 12, 53
  large programs, 66, 71, 80
  no library search, 59, 65
  object files, 14
  sample session, 28
LINK.V2, optional linker, 35
Load module, 82
LOGICAL*2, 90
LOGICAL*4, 90
Long calls, 89

Machine level initialization, 114
Memory
  8086, organization, 110
  contents, 112
  linking, 62
  organization, 110
  requirements
    COMMON, 71, 72, 73
    $LARGE arrays, 72, 73, 112
    long arrays, 71, 72, 73, 134
Metacommands
  $DEBUG, 14, 37, 83, 117
  $DECMATH, 38, 125, 127
  $FLOATCALLS, 38, 125, 127, 135
  $INCLUDE, 78
  $LARGE, 72, 73, 112, 134
  $LIST, 78
  $NODEBUG, 14, 83
  $NOFLOATCALLS, 38, 39, 124, 127
  $NOLIST, 78
  $NOTLARGE, 73, 134
  $STORAGE:2, 23, 36, 39
Metalanguage commands. See Metacommands.
Microsoft LINK
  Linker, 9
Microsoft Macro
  Assembler, 14

Microsoft Pascal, 14, 55, 101, 117
Module
  definition, 16
  load, 82
  object, 55
  relocatable, 16
  size, 82
MS-DOS procedures, 9

Naming conventions, 107
Notation conventions, 4
NUL or null file, 25, 47

Object
  code, 71
  file, 14, 43, 55
  listing file, 47
  modules, 55
Offset addresses, 17
Optimization, 11, 104
Options
  16-bit integers , 36
  alternative linkers, 35
  auxiliary libraries, 58
  default math library , 39
  floating-point, 37
  interface library, 35
  overlay linker (LINK.V2), 35, 64
OUTPUT file, 107
Overlay linker, LINK.V2, 64

Parameters
  batch file, 67
  procedural, 91
PASCOM, 102
Pass one, 23, 78, 102
Pass two, 26, 78, 102, 104
Pass three, 27, 102, 106
Preliminary procedures, 9
Program
  development steps, 11 to 14
  example, 21, 81
  execution, example, 30
  large, 81
  log, sample, 31
  source, 14
  termination, 114, 119
Program level initialization, 117
Public routines, 55

QQ naming convention, 107

Real number
  conversion utilities, 139
  format, 139
REAL*4, 90
REAL*8, 90
Reference books, 5
Register
  saved, 89
  segment, 110
Relocatable
  module, 16
  object file, 14
RETURN key, 5
Routine, 16
Run file, 14, 61
Runtime
  architecture, 108
  definition, 15
  error handling, 119
  initialization, 114
  libraries, 10, 17, 59
  routines, 109
  structure, 55
  termination, 14

Sample session, 19
Scratch file names, 143
Short calls, 89
Software provided, viii
Source
  file, 14, 15, 21, 53
  listing file, 43
  program, 14
Stack, 112, 116

Stackpointer, 121, 122
Statement trees, 105
Subexpression elimination, 104
Subroutines, 71
Switches, linker, 64
Symbol table, 77, 101

Technical information, 101
Templates, 105
TRAP flag, 108
TRMVQQ, 107

Undefined variable, 16
Unit F, 107
Unit identifiers, 109
Unit U, 107
Unit V, 107
Unresolved variable, 16

VM.TMP, 62
Vocabulary, 15