MICROSOFT®

The High Performance Software™

Microsoft®
Pascal Compiler

for the MS-DOS™ Operating System

User's Guide

Microsoft Corporation

If you have comments about the software or this manual, complete the Software Problem Report at the back of this manual and return it to Microsoft Corporation.

Document Number 8511L-330-05
Part Number 020-014-026

About the Microsoft Pascal Compiler

Microsoft® Pascal, also called MS®-Pascal, is a highly extended, portable version of the Pascal language. Compatible with the International Standards Organization (ISO) proposed standard, its extensions facilitate systems programming as well as applications programming.

You can use Microsoft Pascal at the ISO standard level for transporting programs to and from other machines. Or, to make full use of the capabilities of a specific computer, you can make your programs more efficient by using the language at its extend or system levels.

The Microsoft Pascal Compiler generates native machine code; many other Pascal compilers for microcomputers produce intermediate p-code. Programs compiled to native code execute much faster than those compiled to p-code. Thus, with MS-Pascal, you get the programming advantages of a high-level language without sacrificing execution speed. Because of many low-level escapes to the machine level, programs written in MS-Pascal are often comparable in speed to programs written in assembly language.

The MS-Pascal Compiler also generates code for fast numeric processing in the 8087 processing environment and provides 8087 emulation in the system software package.

System Requirements

The Microsoft Pascal Compiler can be used with any computer that has one disk drive and a minimum of 140K random access memory available after the operating system is loaded. (The Microsoft Disk Operating System 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™.

This implementation of the Microsoft Pascal Compiler can take advantage of, but does not require, an INTEL® 8087 numeric coprocessor.

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 pathnames 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.

How to Use These Manuals

Documentation for the Microsoft Pascal Compiler is provided in two binders containing the two manuals described below:

Microsoft Pascal Compiler User's Guide

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

Microsoft Pascal Reference Manual

This manual describes the syntax and use of the Microsoft Pascal language. With the exceptions noted in Appendix B, "Version Specifics," of the *User's Guide* and any recent changes described in the README file (if present), this is the language supported by the Microsoft Pascal Compiler.

Microsoft LIB Library Manager Reference Manual

This manual explains how to use Microsoft LIB to create, organize and maintain runtime libraries.

Microsoft Pascal Quick Reference Guide

This guide briefly lists reference information, such as the definitions of intrinsic functions and the syntax of Microsoft Pascal statements.

The following descriptive devices are used throughout these manuals to emphasize elements of the text. Descriptions of Microsoft Pascal syntax requirements can be found in Chapter 3, "Notation" of the *Microsoft Pascal 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 Pascal is not case sensitive. Small capital letters indicate that you must press a key named by the text; for example, "press the RETURN key."
Italics	Italics indicate user-supplied data, for example, filenames, variable names, and array names.
. . .	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.

Contents

Figures

Tables

Update: Microsoft Pascal 3.3

This update notice lists changes that have been made for version 3.3 of Microsoft Pascal, and tells where you can find more information about these changes.

Appendix A of the *Microsoft Pascal Compiler User's Guide* describes features of version 3.3 that are different from version 3.2. The *Microsoft Pascal User's Guide* and the *Microsoft Pascal Reference Manual* for version 3.2 still apply to version 3.3, except for corrections, changes, and additions described in Appendix A.

Most of the changes made for version 3.3 make it easier to communicate with programs written in other languages, especially C. Mixed-language programming is discussed in Appendix E, "Mixed-Language Programming."

The following items have been added:

new predefined types	ADSFUNC and ADSPROC (Section A.2) INTEGERC (Section A.5)
new attributes	C (Section A.6) and VARYING (Section A.7)
new enumerated types	sharemodes (Section A.3.1) accessmodes (Section A.3.2), and lockmodes (Section A.4)
new procedure	locking (Section A.4)

Also

- There is a new version of the linker, Microsoft LINK 3.01. In LINK 3.01, two new switches allow you to set the maximum number of segments and to use DOS conventions for the order of segments (Sections A.10.1 and A.10.2). Also, the default for the overlay interrupt number has been changed from CD (hexadecimal) to 3F.

- There is one correction to the language, involving precedence when using the ADS and ADR operators (Section A.1).
- You need no longer assign a name to a file before using reset or rewrite. A temporary file name is assumed, as if you had specified

  assign (*file*, char(0)).

- Path names are allowed in compiler directives.
- If you use $decmath+ in your source code, you must link with the library DECMATH.LIB.

The package contents have been changed:

- All disks are now double-sided and double-density.
- The library manager LIB.EXE has been added. Refer to the *Microsoft LIB Reference Manual* for information on LIB.
- The link map files for the libraries (PASCAL.MAP, MATH.MAP, 8087.MAP, DECMATH.MAP, and ALTMATH.MAP) are no longer included. Now that Microsoft LIB is included with Microsoft Pascal, you can create these link map files yourself, as described in Section A.9, "Creating Link Map Files with Microsoft LIB."
- The files DOS2PAS.LIB and DOS2PAS.MAP are no longer included.
- DOS 2.0 support is now the default, so these files are not necessary. Note that DOS 1.0 is no longer supported. Version 3.3 of the Microsoft Pascal Compiler, and programs compiled by it, will not run under DOS 1.0.
- ENTX6L.ASM is not included.

  The initialization and termination systems have been changed, so this file is no longer necessary. All initialization is handled automatically.

Chapter 1

Introduction to the Microsoft Pascal Compiler

The Microsoft Pascal Compiler, version 3.20, implemented for the Microsoft Disk Operating System, MS-DOS, version 1.25, accepts programs written according to the ISO standard (Level 0) and the ANSI-IEEE standard. It also accepts those written in the full Microsoft Pascal language as described in the May 1983 release of the *Microsoft Pascal Reference Manual.*

If you compile your programs with the default compiler options and link them with the standard libraries, PASCAL.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 Pascal Compiler are:

A double precision option for real number calculations in IEEE floating-point format.

Support for linking of 8086 assembly language, Microsoft Pascal, and Microsoft FORTRAN programs.

Extensive program development through support of SUPERARRAYS, flow control, separately compiled UNITS, variable length strings, the address types, constants and functions of ARRAY and RECORD types, and other development features.

1.1 How to Use This Guide

The *Microsoft Pascal Compiler User's Guide* describes the operation of the Microsoft Pascal Compiler, from the most rudimentary procedures to more advanced topics that may be of interest only to experienced programmers. This document assumes that you have a working knowledge of both the Microsoft Pascal language and MS-DOS.

The *Microsoft Pascal Compiler User's Guide* also describes the compiling and linking options 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. The Microsoft Pascal Compiler offers a wealth of options for developing your programs.

For a list of the differences between Microsoft Pascal Versions 3.3 and 3.2, see Appendix A, "Differences Between Versions 3.2 and 3.3."

The initial chapters (Chapters 1 through 6) should be read in their entirety by the first-time user of the Microsoft Pascal Compiler. Included in this material are the procedures you should perform before compiling and linking your first program, a description of the process of program development, and a step-by-step walk-through of each of the procedures that follow the writing of a program: compiling, linking, and running.

The remaining chapters (Chapters 7 through 10) provide information about using a batch command file, compiling and linking large programs, and using assembly language routines. They also provide additional technical information on compiler structure, the Microsoft Pascal file system, floating-point issues, and runtime architecture.

Included in the appendix material are the version specifics of the Microsoft Pascal Compiler for MS-DOS, some of the attributes of the 8087 numeric coprocessor, information on Mixed Language programming, and the list of error messages.

1.2 System Software

The Microsoft Pascal Compiler software package includes two or more disks which contain the following files:

File	Contents
PAS1.EXE	Pass one of the Microsoft Pascal Compiler
PAS2.EXE	Pass two of the Microsoft Pascal Compiler

PAS3.EXE	Pass three of the Microsoft Pascal Compiler
PASCAL.LIB	The runtime library
LIB.EXE	Microsoft Library Manager file
MATH.LIB	The default floating-point package library contained in PASCAL.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
DECMATH.LIB	An auxiliary library containing decimal floating-point support routines
ALTMATH.LIB	An auxiliary library containing high-speed floating-point support routines
LINK.EXE	Default version of Microsoft LINK
NULE6.OBJ	The dummy error system
FINU	Declarations of low-level file system routines (the Unit U interface)
FINK	Declaration of the generic file control block type, FCBFQQ
SORT.PAS	Bubble sort demonstration program
PRIMES.PAS	Prime number generator program
README	If present, contains information that is more up to date than the documentation contained in these manuals

1.3 Learning More About Pascal

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

Findlay, W., and D. F. Watt. *Pascal: An Introduction to Methodical Programming*. London: Pittman, 1978.

Holt, Richard C., and J. N. P. Hume. *Programming Standard Pascal*. Reston, Va.: Reston Publishing Company, 1980.

Jensen, Kathleen, and Niklaus Wirth. *Pascal User Manual and Report*, New York: Springer-Verlag, 1978.

Koffman, E. B. *Problem Solving and Structured Programming in Pascal*. Reading, Mass.: Addison-Wesley Publishing Company, 1981.

Schneider, G. M., S. W. Weinhart, and D. M. Perlman. *An Introduction to Programming and Problem Solving With Pascal*. 2d ed. New York: John Wiley & Sons, 1982.

Chapter 2
Getting Started

This chapter provides a brief review of the procedures, terms, and concepts involved in developing Microsoft Pascal programs on your microcomputer.

2.1 Preliminary Procedures

This section describes several preliminary procedures, some of which are required and some of which are highly recommended before you begin the sample session or compile any programs of your own. If you are unfamiliar with any of the MS-DOS procedures mentioned, consult your *MS-DOS User's Guide* 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 future 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.

And, 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. This setup assumes you have two 160K disk drives available.

Table 2.1

A Suggested Disk Setup

Disk	Contents
1	COMMAND.COM text editor† miscellaneous utilities†† PAS1.EXE
2	COMMAND.COM PAS2.EXE PAS3.EXE
3	COMMAND.COM PASCAL.LIB LINK.EXE

† Any text editor that fits.
†† MS-DOS utilities to set up printer, clear screen, sort directory, etc.

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

2.1.3 If You Have an 8087 Coprocessor

This step may be required if you have an 8087 coprocessor. The auxiliary library, 8087.LIB, supports a particular arrangement of 8086/8087/8088 hardware. Specifically, it expects that i8087 interrupts will be directed through to the 8086/8088 via the 8086/8088 interrupt vector 2 (NMI), without the intervention of an 8259 interrupt controller or its equivalent.

Check to see 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 C, "Customizing i8087 Interrupts," and customize the runtime library as described there.

2.2 Program Development

This section provides a short introduction to program development, a multistep process which includes first writing the program, and then compiling, linking, and executing 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 of 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-Pascal 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-Pascal 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-Pascal Compiler has a three-part structure. The first two parts, pass one and pass two, together carry out the optimization 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 Pascal 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 libraries. The linker saves your program on disk as an executable file, ready to run.

You may, at linktime, 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 Pascal Program," and in Chapter 6, "More About Linking."

Figure 2.1 illustrates the entire program development process. The paragraphs following the figure describe the process in more detail.

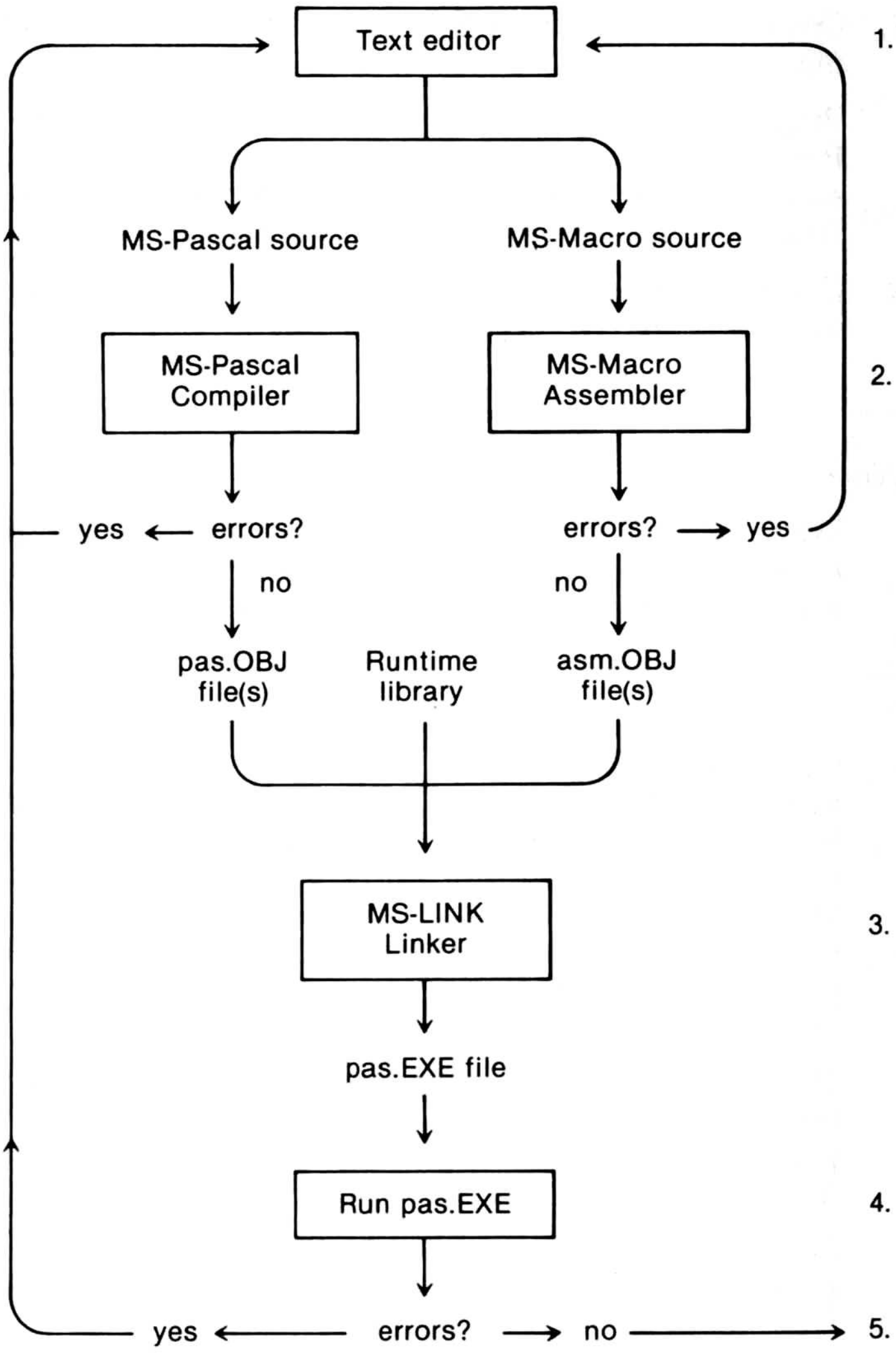


Figure 2.1. Program Development

1. Create and edit the MS-Pascal (and MS-Macro) source file.

   Program development begins when you write an MS-Pascal program; any general purpose text editor will serve. 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.

Note

When your program has been successfully compiled, remove the $debug+ metacommand and recompile to enhance your program's execution time.

Once you have written a program, compile it with the MS-Pascal Compiler. The compiler flags all syntax and logic errors as it reads your source file. Use the error-checking switches or their corresponding metacommands (described in Section 5.4, "Pass One Compiler Switches") to generate diagnostic calls for all 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, MS-Macro.

(You may have received MS-Macro as part of the utility package that came with your computer system. If not, it is available separately from your software dealer.)

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

   A compiled (or assembled) object file is not executable and must be linked with the runtime library, PASCAL.LIB, using either LINK.EXE or LINK.V2. Separately compiled Microsoft FORTRAN subroutines can also be linked to your program at this time.

Note

Auxiliary math runtime libraries and the MS-DOS file system library, DOS2PAS.LIB, may also be linked. For more details, see Section 6.1.1.2, "Auxiliary Libraries."

---

4. Run EXE file.

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

5. Recompile, relink, and rerun with $debug−.

   Repeat this process until your program has been successfully compiled, linked, and run without errors. Then recompile, relink, and rerun it without the runtime error-checking switches, to reduce the amount of time and space required. Chapter 8, "Compiling and Linking Large Programs," discusses how to work within 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-Pascal 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 .PAS and .OBJ, respectively. After the source code has been compiled, the object file(s) must be linked with the runtime libraries 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-Pascal source file and creates a relocatable object file.

2. Linktime

   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 libraries:

1. Module

   A general term for a discrete unit of code. There are several types of modules, including relocatable and executable modules. (Furthermore, in the MS-Pascal language, "module" has a specific meaning as one type of MS-Pascal compiland. See the *Microsoft Pascal Reference Manual* for details. In this *User's Guide*, we use the term "module" in its general sense, unless otherwise specified.)

   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 procedure or function. More than one routine may reside in a module.

3. External reference

   A variable or routine in one module that is referred to by a routine in another module.

   The variable or routine is often said to be "defined" or "public" 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 library.

   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

   The module's 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 Microsoft Pascal language. A library module usually corresponds to a feature or subfeature of the MS-Pascal language.

Chapter 3
A Sample Session

This chapter provides step-by-step instructions for compiling and linking an MS-Pascal program. We strongly recommend that you compile the sample program before compiling any of your own MS-Pascal programs.

If you enter commands exactly as described, you should have a successful session. If a problem does arise, check to see that you have correctly carried out all of the required procedures described in Section 2.1, "Preliminary Procedures," and carefully redo each step in the sample session up to the point where you had trouble.

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

1. Write and save an MS-Pascal source file.
2. Compile your program with the MS-Pascal 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.)
3. Link your object file to the MS-Pascal runtime libraries.
4. Execute (i.e., run) your program.

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

The sample session makes the following assumptions:

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 compiler or 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 PAS1.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 the proper system disk before running MS-LINK.

3.1 Creating a Microsoft Pascal 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-Pascal programs with any available text editor. The source file should, in most cases, have the .PAS extension. For this sample session, we will use the program named SORT.PAS, which came with the system software.

Copy SORT.PAS to drive B:, which is where it would be if it were your own program.

3.2 Compiling Your Microsoft Pascal 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 the sample session, we will run all three passes.

3.2.1 Pass One

Insert the disk containing PAS1.EXE in drive A:. In response to the operating system prompt, type:

```
A:PAS1
```

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

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."

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. Source file

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

   ```
   Source filename [.PAS]:
   ```

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

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

2. Object file

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

   ```
   Object filename [SORT.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 *SORT* or press the RETURN key.

3. Source listing file

   The third prompt is for the name of the source listing file, 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, the source listing will be sent to the null file, NUL.

   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 to send the source listing file to the terminal screen. Therefore, type *USER* in response to the source listing prompt. (Typing *CON* has essentially the same effect; see Section 5.2, "Filename Conventions," for further information.)

4. Object listing file

   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 *USER* (or *CON*) to request that the object listing be displayed on your terminal screen when you run pass three.

Compilation begins as soon as you have responded to all four prompts. The source listing is displayed on your screen, 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, messages like the following would appear instead:

```
Pass One      2 Warnings Detected.
Pass One      3 Errors Detected.
```

The error and warning messages would appear in the source listing as it comes on your screen.

1. Errors are mistakes that prevent a program from running correctly.
2. Warnings indicate a variety of conditions, none of which will prevent the program from running, but which may reflect poor programming practice or produce invalid results.

See Appendix F, "Error Messages," for a complete listing of messages and information about how to correct the errors in your program.

Pass one creates two intermediate files, PASIBF.SYM and PASIBF.BIN. The compiler saves these two files on the default drive for use during pass two.

If there are errors, the two intermediate files are deleted and the remaining passes cannot be run. If pass one generates only warning messages, you can still run passes two and three, but you should go back and correct the source file at some point.

3.2.2 Pass Two

Remove the disk containing PAS1.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 PAS1.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 like 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 intermediate files created during pass two, and, because of your earlier response to the object listing prompt, writes the object code listing to your screen.

When pass three is complete, the two intermediate 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 Microsoft Pascal Compiler

Pass	Reads	Writes	Deletes
1	SORT.PAS	USER.LST PASIBF.SYM PASIBF.BIN	
2	PASIBF.SYM PASIBF.BIN	SORT.OBJ PASIBF.OID PASIBF.TMP	PASIBF.SYM PASIBF.BIN
3	PASIBF.OID PASIBF.TMP	USER.COD	PASIBF.OID PASIBF.TMP

See Chapter 5, "More About Compiling," for details about compiler switches and other ways of responding to the compiler prompts.

3.3 Linking Your Microsoft Pascal Program

Now you are ready to link your program with one of the two versions of MS-LINK provided with the Microsoft Pascal Compiler version 3.20. Linking converts the relocatable object

file into an executable program by assigning absolute addresses and setting up calls to the runtime libraries.

Remove the disk containing PAS3.EXE from drive A: and insert the disk containing LINK.EXE. You won't need to do this 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 the front end of the compiler, gives a series of four prompts to which you must respond before linking begins. The linker prompts 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 location of the runtime library

Each of these prompts is discussed briefly in the following paragraphs and in somewhat more 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 PASCAL.LIB. You won't need to do this if the linker and the runtime library are on the same disk.

Note

Prompting specific to LINK.V2, the optional linker, is discussed in detail in Section 6.1.2, "Linking Libraries." Included in that discussion are details concerning the overlay manager of the optional linker.

---

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. Here, the prompt indicates that .OBJ is the default extension for any file(s) you name. Type *SORT*, and the file SORT.OBJ, created during compilation, will be linked with PASCAL.LIB during the linking process. If, for any reason, the 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 the name of the run file, the file created by the linker that will contain your executable program:

   ```
   Run File [SORT.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 at your screen (console), without having it written to a disk file, type *CON* in response to the list file prompt. (The linker does not recognize *USER* as a name for your screen.)

   If you want the linker map written to a disk file, respond to this prompt with a name for the file.

4. Runtime libraries prompt

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

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

   Here, to indicate that PASCAL.LIB is found on drive A:, you should simply type:

   ```
   A:
   ```

   If PASCAL.LIB is 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, MS-LINK links your program, SORT.OBJ, with the necessary modules in the MS-Pascal runtime library, A:PASCAL.LIB. This linking process creates an executable file, named SORT.EXE, on the default drive.

3.4 Executing Your Microsoft Pascal Program

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

```
SORT
```

This command directs MS-DOS to load the executable file SORT.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 see displayed on the screen first an unsorted list of numbers and then the same list in sorted order.

This concludes the sample session. Additional information on compiling and on 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 small capital letters), and files written to the terminal screen.

```
A> B:
B> A:PAS1
Source filename [.PAS]:SORT
Object filename [SORT.PAS]:RETURN
Source listing [NUL.LST]:USER
Object listing [NUL.COD]:USER

    [Source listing display]

    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 display]

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

B> SORT

    [Program display]
```

Chapter 4
Options for Compiling and Linking

This chapter contains descriptions of optional libraries and compiler features that are available to the users of Microsoft Pascal 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 Pascal.

4.1 MS-DOS 2.0 File System Library

When you specify DOS2PAS.LIB at linktime, it will automatically replace the standard file system in the runtime library, PASCAL.LIB, with the MS-DOS 2.0 file system. This will not be true if you are specifying PASCAL.LIB explicitly and have not specified DOS2PAS.LIB *before* PASCAL.LIB in the list of libraries to be searched by the linker.

Note

Programs linked with DOS2PAS.LIB will not run under MS-DOS 1.25. An error message, "Incorrect DOS version" will be returned by the operating system.

4.2 Alternative Linkers

Two versions of the Microsoft LINK utility are provided with this version of Microsoft Pascal. 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 and will run only on MS-DOS 2.0.

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 Pascal.

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 accurate 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 Precision of Basic Numeric Types

In Microsoft Pascal, the basic INTEGER type is, by default, equivalent to the Microsoft Pascal type INTEGER2 which is a 16-bit, two's complement integer. The basic REAL type is similarly equivalent to a REAL4 which is a four-byte real number.

You can use the $integer:n metacommand with (n=4) to change the default and make INTEGER equivalent to INTEGER4 (but note the restrictions on INTEGER4). You can use the $real:n metacommand with (n=8) to make the REAL type equivalent to REAL8. Note that the default, which is equivalent to {$real:4 $integer:2}, will result in the fastest and smallest code.

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 $floatcalls— 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 Pascal 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.

Note

Decimal floating-point and IEEE floating-point *are not* compatible.

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 End Cases for Compilation and Execution

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

1. Fastest: (with 8087)

   To get the best possible performance if you have an 8087, use the $floatcalls– metacommand and {$integer:2 $real:4} (the default) and link with 8087.LIB. This will also be the smallest version of your program.

2. Fastest: (without 8087)

   To get the best possible performance without an 8087, use {$integer:2 $real:4} metacommands and link with ALTMATH.LIB.

3. Most portable, most consistent:

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

Chapter 5
More About Compiling

This chapter provides additional procedural information on the compiler, supplementing the discussion in Section 3.2, "Compiling Your Microsoft Pascal Program." For a more 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-Pascal runtime libraries 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 H, "Messages," in the *Microsoft Pascal Reference Manual* for a list and explanation of all error messages.

If you include files in the compilation with the $include metacommand, these files are also shown in the source listing.

Both the $include metacommand and the source listing are discussed in the *Microsoft Pascal Reference Manual*. See Section 18.3, "Source File Control," for information on $include; see Section 18.4, "Listing File Control," for a description of metacommands that control listing file format; see Section 18.5, "Listing File Format," for a discussion of features of the listing file.

The various flags, level numbers, error message indicators, and symbol tables make the source listing 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 libraries.

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 can 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. 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 the 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.

If you have a disk "drive" that is implemented with random access memory (or other faster file device), instead of an actual disk, this is a good location for intermediate files. Since they are processed twice each, first written and then read, compilation is faster if these files are written to such a fast access device.

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.

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-Pascal Compiler and MS-LINK recognize and assign certain conventional filename extensions by default, as shown in the following list:

Filename Extension	Function of File
.PAS	MS-Pascal source file
.FOR	MS-FORTRAN source file
.OBJ	Relocatable object file
.LST	Source listing file
.COD	Object listing file
.ASM	Assembler source file
.MAP	Linker map file
.LIB	Library files
.EXE	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-Pascal Compiler supplies one of those shown in Table 5.1.

Table 5.1

Filenames Assigned by the Compiler

File	Device	Extension	Full Filename
Source file	dev:	.PAS	dev:filename.PAS
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 filenames 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 in 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.

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

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.PAS; typing *ABC.* instructs the compiler to accept ABC, with no extension, as the name.

3. Leading and trailing spaces are permitted. Therefore, the following is an acceptable response to the prompt for the source filename:

   ABC ;

   The filename itself must not contain spaces.

4. 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.

5. 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, 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.

6. 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 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.PAS
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.

5.3 Starting the Compiler

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

1. You can let the compiler prompt you for each of the three 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:PAS1
```

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

```
Source filename [.PAS]: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 filename 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:PAS1 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:PAS1 DATABASE,DATABASE,DATABASE,DATABASE;
A:PAS1 DATABASE,,,;
```

Both result in the following four filenames being assigned:

Source file	B:DATABASE.PAS
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:PAS1 YOYO,YOYO,NUL,NUL;
A:PAS1 YOYO;
```

You may include spaces before or after filenames, but not within them. The command line may also include switches, described in Section 5.4, "Pass One Compiler Switches," anywhere that spaces can go.

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. This feature of the compiler makes it relatively failsafe to use.

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 and small capital letters):

```
B> A:PAS1 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.PAS
Object file	B:TEST.OBJ
Source listing	B:TEST.LST
Object listing	B:NUL.COD

5.4 Pass One Compiler Switches

By adding switches to the command line when you start pass one of the compiler, or to your response to any of the pass one prompts, you can direct the MS-Pascal Compiler to perform additional or alternate functions. The switch tells the compiler to "switch on" a special function or to alter a normal compiler function. More than one switch may be used, but each must begin with a slash (/). Do not confuse these switches with the linker switches, which are discussed in Section 6.4, "Linker Switches."

Switches affect the entire compilation and may be placed anywhere that spaces may go. You may enter them either on the command line or in response to compiler prompts. Table 5.2 shows the compiler switches that are currently available to you, the default position of the switch, and the corresponding metacommand.

Table 5.2

Command Line Switches

Switch	Default	Metacommand	Action
/A	off	$indexck	Checks for array index values in range, including super array indices
/C	off	$decmath	Directs the compiler to use the decimal math routines in the auxiliary math runtime library, DECMATH.LIB
/D	off	$debug	Switches on all debugging switches
/E	off	$entry	Generates procedure entry and exit calls for debugger
/I	off	$initck	Checks for use of uninitialized values
/L	off	$line	Generates line number calls for debugger
/M	off	$mathck	Checks for mathematical errors such as overflow and division by zero
/N	off	$nilck	Checks for dereferencing of any pointers that are NIL
/Q	off	$debug	Switches off all debugging switches
/R	off	$rangeck	Checks for subrange validity including assignments
/S	off	$stackck	Checks for stack overflow at procedure or function entry
/T	off	$tagck	Checks tag fields in variant records

Since all of the pass one switches correspond to MS-Pascal metacommands, you can achieve the same effect either by using the metacommand in the source file or by giving the command as a switch to the compiler. However, any instruction given as a metacommand in the source file overrides the corresponding switch given at compile time.

The MS-Pascal metalanguage is discussed in detail in Chapter 18, "Microsoft Pascal Metacommands," of the *Microsoft Pascal Reference Manual*. The two switches, /D and /Q, are equivalent to the $debug+ and $debug− metacommands, respectively, except that they also turn on and off $entry and $line.

Several of the switches correspond to runtime error checking metacommands that end with the letters "CK". Because of the way these switches are implemented, turning one off does not guarantee that the check will never be performed; it only means that no extra effort will be spent to perform the check.

Important

One strong caution should be observed. Error-checking switches and their corresponding metacommands cause a significant increase in the volume of code generated. You may wish to use them in the early stages of program development, then recompile your program without them to reduce your program's code size and decrease its execution time.

The following sample command lines and responses illustrate the use of compiler switches (your responses are shown in italics):

This turns on $indexck:

```
B> A:PAS1 /A DEMO,,,NUL
```

This turns on all of the switches:

```
B> A:PAS1 DEMO,,,/D
```

This sequence first turns all switches off, and then turns on $mathck and $indexck, and later $initck:

```
B> A:PAS1 DEMO/Q
Object filename [DEMO.OBJ]:/M/A
Source listing  [NUL.LST]:DEMO
Object listing  [NUL.COD]:/ /
```

Chapter 6
More About Linking

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

Interfacing to Microsoft FORTRAN routines is quite straightforward. The Microsoft FORTRAN procedure or function must be external in the Microsoft Pascal source code, and all parameters must be VARS or CONSTS. For other languages, see the appropriate reference and user manuals.

You may need to write assembly language interface routines to translate from the Microsoft Pascal or Microsoft FORTRAN

calling convention or function return to the one used by that language. Whatever the language, it must be able to produce linkable object modules. For information on linking assembly languages routines, see Chapter 9, "Using Assembly Language Routines." For further information on Microsoft LINK, see the appropriate chapter in your MS-DOS manual.

The ability to link together programs, units, and modules of Microsoft Pascal 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 larger than 64K bytes of code. See Chapter 8, "Compiling and Linking Large Programs," for more information.

For now, assume that you have created a program that uses one Microsoft Pascal unit and one Microsoft Pascal module 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
UNIT.OBJ
MODU.OBJ
ASM1.OBJ
ASM2.OBJ
```

To link these all together, first start 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 PASCAL.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+UNIT+MODU+ASM1+ASM2
```

The first object file listed must be a Microsoft Pascal program, module, or unit, 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 Microsoft Pascal object file, you may list the other modules, units, 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 that follows).

Table 6.1

Linker Defaults

Prompt	Default Response
Object modules	None
Run file	prog.EXE
List map	NUL.MAP (i.e., no map)
Libraries	PASCAL.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.) It

also may contain fixups used in floating-point instructions. (See Section 10.4.1, "The $floatcalls– Option," for details.)

Microsoft Pascal causes the linker to search for the runtime libraries PASCAL.LIB and MATH.LIB which are supplied with the compiler and contain the standard runtime modules for Pascal. 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 runtime. The auxiliary libraries supplied with Microsoft Pascal are:

1. ALTMATH.LIB which contains a high speed 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. DOS2PAS.LIB which contains the MS-DOS 2.0 input/output system.

Note

The auxiliary math libraries completely replace MATH.LIB and can be specified before or after an explicit reference to PASCAL.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.

DOS2PAS.LIB replaces the equivalent part of PASCAL.LIB and must be searched *before* PASCAL.LIB. This will occur automatically unless you are specifying PASCAL.LIB explicitly. In this case, DOS2PAS.LIB must come before PASCAL.LIB in the list of libraries supplied to the linker.

6.1.2 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 PASCAL.LIB to be searched, you would enter *pascal.lib* in the sequence of prompts;

```
Object Modules [.OBJ]:your modules
Run File [SORT.EXE]:RETURN
List File [NUL.MAP]:RETURN
Libraries [.LIB]:pascal.lib
```

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

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

If PASCAL.LIB or any library that you have specified cannot be found, the following message will appear on your screen:

```
Cannot find library filename.lib
Enter new library 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 PASCAL.LIB. However, this will produce unresolved reference error messages unless you replace every required runtime routine with a routine of your own.

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

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

See your *MS-DOS User's Guide* for complete information on using different libraries with Microsoft LINK. Because a library is read twice by the linker, this file is also a good candidate for a file to put on a RAM-based disk drive or other fast file device.

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, PASCAL.LIB, you respond,

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

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

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

the linker will look for FOO.LIB.

Note

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

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.

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 or drive name and pathname for LINK.V2 in response to the run file prompt.

6.2.2 The Linker Listing File

The link 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 PUBLIC variables. See Section 6.4, "Linker Switches," for information on the /MAP switch.

The link 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. (The special filename *USER* is not recognized by the linker.) 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 terminate.

If the linker is terminated with a CTRL-C, 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 + j)*

elements (b+c), (e+f) and (i+j) 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 drive 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 drive 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 Pascal. 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 2.0 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	Tells the linker not to automatically search PASCAL.LIB.
/NOGROUPASSOCIATION	If you are using LINK.V2 and find that you must link in an old Microsoft Pascal/ FORTRAN library, you must use this switch. This option causes LINK.V2 to initiate the addressing scheme of linkers delivered with MS-Pascal versions 3.14 and earlier.
/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.

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

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.

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

MS-DOS allows you to create a batch file for executing a series of commands. Creating and using batch command files 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-Pascal 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 prompts, $inconst values, or the like in a batch file. All compiler parameters must be given on the command line, as described in Section 5.3.2, "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:PAS1 %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 SORT

SORT 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, they should have different filenames.

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

Chapter 8

Compiling and Linking Large Programs

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 code that can be generated at once by the MS-Pascal Compiler is 64K bytes. However, since you can compile any number of compilands separately and link them together later, the real program size limit is not 64K but the amount of 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 MS-Pascal modules and units to better structure the development and maintenance process. As always, there is a tradeoff between size and speed. Procedure and function calls within a module to routines without the PUBLIC attribute are somewhat faster, since intrasegment calls, which run faster, are generated rather than intersegment calls.

Finally, if your program is still too big, you should consider devising an overlaying scheme. See Section 6.3, "The Overlay Linker," for details.

8.2 Avoiding Limits on Data Size

Data includes your main variables, the stack, and the heap. MS-Pascal operates with data in two regions of memory:

1. the default data segment
2. the segmented data space

The upper limit on the amount of data that can reside in the default data segment is also 64K bytes. You can go beyond this limit by taking advantage of the ability to place certain kinds of data outside the default data segment, using ADS variables, the long heap allocator (HESM), VARS and CONSTS parameters, and segmented ORIGIN variables.

The default data segment normally holds the following:

1. all statically allocated variables
2. constants that reside in memory
3. heap variables
4. the stack, which holds parameters, return addresses, stack variables, etc.

The segmented data space includes the entire 8086 address space, including the default data segment. See the appropriate chapters in the *Microsoft Pascal Reference Manual* for a discussion of these MS-Pascal features.

Although operations with data in the default data segment are more efficient (i.e., generate less code and run faster) than those with data that may be in any other segment, almost all MS-Pascal operations work equally well on data outside the default data segment. Only in the following cases must data reside in the default data segment:

1. file variables
2. the LSTRING parameters to ENCODE and DECODE
3. all parameters to READSET

To allocate data outside the default data segment, you can use the long heap allocator which works in a similar way to NEW and DISPOSE. If you wish to make static allocation of data outside of the default data segment, you must go outside the MS-Pascal system itself. If you already know the address of free blocks of memory on your computer, you can use these addresses in a segmented ORIGIN attribute or assign them to an ADS variable. Otherwise, you can get the addresses of free memory from MS-DOS, using a process (described in item 4 in Section B.1, "Implementation Additions") to get a pair of ADS variables to the lower and upper bounds of available memory.

Many applications use a large block of memory for primary data, as well as various other variables to control access and processing of this data. For example, a text editor will have a work area; a data base system will have a data area (or index area); and so on. This large block can be managed outside the default data segment with ADS variables.

In the default data segment, the heap and the stack grow toward each other. Heap allocation will attempt to use existing disposed blocks in the heap itself, before growing into memory shared with the stack. As a part of this process, adjacent disposed blocks are merged, and free blocks at the end of the heap become available to the stack. However, only heap allocation (i.e., NEW or ALLHQQ) releases free heap blocks to the stack. Therefore, it you are running out of stack after a number of DISPOSE operations, make the following call:

```
EVAL (ALLHQQ (65534));
```

ALLHQQ should be declared as an external function, like this:

```
FUNCTION ALLHQQ (SIZE:WORD); WORD; EXTERN
```

8.2.1 Long Heap Allocation

A "long" heap module, HESM (segmented addresses), has been provided with this release of Pascal. The module assumes that all memory from the top of DGROUP to the bottom of MS-DOS is available to the long heap allocator.

Four library procedures (system extensions) have been provided that allow the user to access segmented memory beyond the default DGROUP data segment ASSUMED by the MS-Pascal runtime. The routines assume that all memory beyond DGROUP has been allocated to the user by the MS-DOS loader. Requests to the ALLMQQ and GETMQQ routines may be up to 64K.

You can use one of the following four routines to allocate and free up to 64K bytes of memory at a time. The amount of memory actually reserved is always a unit number of 16-byte paragraphs. These routines are at the system level of extension.

```
{Allocate block of "wants" bytes.}
FUNCTION  ALLMQQ (wants : word) : adsmem;
```

```
{Free a block. No error handling.}
FUNCTION  FREMQQ (block : adsmem) : word;

{Call ALLMQQ, check for error returns.}
FUNCTION GETMQQ (wants : word) : adsmem;

{Call FREMQQ, check for error returns.}
PROCEDURE DISMQQ (block : adsmem);
```

Procedure ALLMQQ allocates segmented memory blocks. (No single block may be greater than 64K bytes.) Function FREMQQ frees a segmented block of memory; returns 0 if no errors encountered, nonzero otherwise. Function GETMQQ performs an ALLMQQ with error checking. Procedure DISMQQ performs a FREMQQ with error checking.

If the space above DGROUP is exhausted using ALLMQQ, blocks of memory are allocated by ALLMQQ from the "small" unsegmented heap between the end of program space and the user stack. Thus, ALLMQQ can utilize all of the available user memory on 8086 based machines.

In the above routines the parameter "wants" is the size in bytes of the desired memory request and the parameter "block" is a segmented address. ALLMQQ(w).r will be (0) if a block of the correct size cannot be found and (1) if the segmented address space has been corrupted.

Unlike allocating with NEW, the compiler cannot check that sufficient bytes are allocated to contain the variable you are going to reference. This is the responsibility of your program.

For variables of fixed size including variant records, you can use the SIZEOF function. You must use ADS variables to reference the allocated variable, so be careful when you assign one ADS variable to another, since there is no type checking on such assignments.

This means that you could accidentally retype the variable to one that is longer than the space allocated. If you reference parts of the variable for which no memory is assigned, you will get unpredictable results. If you *assign* to those same parts, you may get serious and, occasionally, catastrophic errors.

8.2.2 Allocating Dynamic Arrays on the Long Heap

While you can use SIZEOF and appropriately typed variables (with care) to reference fixed size variables correctly, SIZEOF is less effective if you want to allocate arrays whose bounds are to be fixed at runtime. (Unlike NEW, you cannot use a pointer to supertype, since pointers cannot reference the long heap.)

However, you can achieve the effect of "dynamic" bounds, by declaring your address type as if it referenced an array with fixed bounds greater than the maximum you would have allocated if you were using NEW.

In any particular instance, you allocate memory only for the array elements you require for that instance. To be safe (at least when you are developing your program), you should explicitly check index values against the value you used to allocate the array, before using them to reference the array.

The following example illustrates how to use the long heap allocator and includes an appropriate test on the index.

```
PROGRAM LONGALLOCATION;

CONST
    MAXINDEX = 4000;

TYPE
    MAXARRAY = ARRAY [1 . . MAXINDEX] OF RECORD
                                          i : integer;
                                          j : real
                                        END;
    ADSARRAY = ADS OF MAXARRAY;
    REKORD = RECORD
            CASE INTEGER OF
            0 : ( );
            1 : (b : byte);
            2 : (i4 : integer4)
            END;
    ADSREKORD = ADS OF REKORD;

VAR
    w : word;
    r : rekord;
    rp : adsrekord;
    ap : adsarray;
```

```
{Allocate a block of "wants" bytes}
FUNCTION ALLMQQ (wants : word) : adsmem; extern;

{Free a previously allocated block (no error handling)}
FUNCTION FREMQQ (block : adsmem) : word; extern;

{Call ALLMQQ, and check for error returns}
FUNCTION GETMQQ (wants : word) : adsmem; extern;

{Call FREMQQ, and check for error returns}
PROCEDURE DISMQQ (block : adsmem); extern;

PROCEDURE P (U : INTEGER);
BEGIN
    W := 20;
    AP := ALLMQQ (SIZEOF (AP^[1]) * W);
    {You must not use an index greater than w}
    RP := ALLMQQ (SIZEOF (R, 2));
    IF U > ORD(W) THEN ABORT
            ('ARRAY BOUND EXCEEDED', WRD(U), W);
    AP^[U].I := 5;
    { . . . }
    { . . . }
    DISMQQ (AP);
    DISMQQ (RP)
END;
BEGIN
p (9);
END.
```

8.3 Working With Limits on 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 the 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 about a thousand identifiers visible at any one time. This assumes a 64K default data segment (i.e., about 160K of memory total); it also assumes that most of your identifiers are seven characters or shorter and are not PUBLIC or EXTERN.

Once a procedure or function is compiled, its local identifiers can be released to provide room for new ones. Several methods of reducing the number of identifiers in a program are described in the following paragraphs.

1. Break your program into modules or units.

   The best way to reduce the number of identifiers is to break up your program into modules or units. When dividing your application into pieces, one guiding principle is to minimize the number of shared (i.e., PUBLIC and EXTERN) identifiers. This not only is good programming practice, it makes compilation easier.

   Breaking up a program may force you to choose between a shared variable and a shared procedure or function. Usually a shared procedure or function is "cleaner"; it is easier to trace the use of a procedure than the use of a variable, for example. However, a shared variable is usually more efficient in terms of memory required and number of identifiers used.

2. Simplify your identifiers.

   Although it reduces the readability of a program (since naming something is a more readable way of referring to it than giving an arbitrary number), you may simplify your identifiers by replacing names with numbers. If necessary, any of the following may help:

   a. Change enumerated types into WORD types and use numbers instead of identifiers.

   b. Use constant literals instead of constant identifiers.

   c. Combine related procedures and functions into single ones, with a parameter indicating the type of call.

   d. Combine variables into an array and refer to the variables using constant array indices.

A special caution is required regarding interfaces. When an interface USES another interface, it must import all identifiers in the other interface. To do this, the other interface must have been declared, so now its identifiers occur twice. If a third interface USES both of the first two, the first interface's identifiers occur three times and the second interface's identifiers occur twice, and so on. This is an easy way to run out of identifiers!

The only reason an interface needs to USE another interface is to import identifiers for types; an interface has no use for variables, procedures, and functions. You can declare a single interface with global types; this is the only interface used by other interfaces. Once compilation gets past the USES clause in the PROGRAM, MODULE, or IMPLEMENTATION, many of these "extra" identifiers are removed.

8.3.2 Complex Expressions

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

1. a very complex statement or expression (i.e., one that is very deeply nested)
2. a large number of error messages
3. a large number of structured constants, including string constants

You may be able to change literal strings and other structured constants into EXTERN READONLY variables which get initialized (as PUBLIC variables) in another module.

Usually, if a program gets through pass one without running out of memory, it will get through pass two. The major exception occurs with complex 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 a WRITE or WRITELN procedure call with many expressions)

If pass two runs out of memory, it displays the following 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 calls). 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.

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 PAS1.EXE, just type *PAS1* (or *dev:PAS1*, if necessary) to load pass one. When the "Source File" prompt appears, you can remove the disk containing PAS1.EXE. If you have a single-drive system, replace the system disk with the disk containing your source file. PAS1 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. One way to do this is to create a master file with lines such as:

```
{$message:'insert B:P1.PAS'
    $inconst:P1 $include:'B:P1.PAS'}
{$message:'insert B:P2.PAS'
    $inconst:P2 $include:'B:P2.PAS'}
{$message:'insert B:P3.PAS'
    $inconst:P3 $include:'B:P3.PAS'}
```

The $inconst metacommand makes the compiler pause while you switch disks. These $include metacommands can also be simply typed at your screen. 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. If you think you could get a listing file on the disk, except that the source and intermediate (PASIBF) files take up too much room, include a line like the following near the start of your source file:

```
{$inconst:ZERROR} CONST ERROR = 1 DIV ZERROR;
```

If you respond with *0* to the "Inconst ZERROR" prompt, you will get a compiler error. The compiler error stops the writing of the intermediate files, which will leave room on the disk for your listing. However, then you will have to run the front end twice, once to generate intermediate files for pass two and once for the listing.

Another way to control a large listing file is use of the $list metacommand. Turn off generation of listing code with the $list− metacommand, and then use the pair of metacommands, $list+ and $list−, to bracket only those portions of the program for which you want a source listing.

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 input 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 or RETURN to begin pass two.
```

Take out the PAS2.EXE disk and insert the disk with the intermediate file from pass one. Now press ENTER or RETURN 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 input intermediate files PASIBF.SYM and PASIBF.BIN on one drive and PASIBF.TMP on the other drive (also PASIBF.OID if you are making an object listing file).

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 will also need the pause command; for example, *PAS2 B/PAUSE*. Pass two will respond with a message like 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 linker 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:.
2. Insert the disk containing PAS1.EXE in drive A:, type *A:PAS1*, and wait for the "Source File" prompt.
3. Remove the disk containing PAS1.EXE from drive A: and insert the disk containing the source file, LARGE.PAS.
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 the 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-Pascal 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 handling
3. error checking

8.5.1 I/O

Because most MS-Pascal programs perform I/O, they require linking to the MS-Pascal file system in the runtime library. However, some programs do not perform I/O and others perform I/O by directly calling the MS-Pascal 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 INIFQQ 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 INIFQQ and ENDYQQ subroutines in your program, as follows:

```
PROCEDURE INIFQQ [PUBLIC];
BEGIN
END;

PROCEDURE ENDYQQ [PUBLIC];
BEGIN
END;
```

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

However, if you do include the dummy procedures shown and the linker produces any error messages for global names that end with the "FQQ" or "UQQ" suffix, your program requires the file system and the process described above will not work. The most common ones would be NEWFQQ, the file variable initializer, and BUFFQQ, the lazy evaluation evaluator.

On the other hand, if your program doesn't require the I/O-handling procedures called by Unit U, you can use the dummy file NULF.OBJ instead. NULF.OBJ contains the dummy subroutines for INIFQQ and ENDYQQ, as well as dummies for INIUQQ and ENDUQQ.

8.5.2 Runtime Error Handling

If runtime error handling is not required, the load module can be further reduced in size by eliminating the error message module and replacing it with the null object module, NULE6.OBJ. NULE6.OBJ provides for simple termination of a program if an error occurs.

INUXQQ, the unit initialization helper, also resides in the error unit. If you want to replace error handling with NULE6, you must do any unit initialization yourself and remove the keyword BEGIN from all the interfaces in your source program.

8.5.3 Error Checking

Compiling and linking a program with the error-checking switches or metacommands on may generate up to 40 percent more code (or even more with $line+) than with these switches or metacommands off. Therefore, after a program has been successfully compiled, linked, and run, turn the error-checking switches off and do the entire process again to create a program that will run considerably faster.

Chapter 9

Using Assembly Language Routines

After describing the MS-Pascal calling conventions and internal representations of data types, this chapter shows how to interface 8086 assembly language routines to MS-Pascal compilands. The information in this chapter is not required for most MS-Pascal programs and is intended primarily for the advanced programmer who is familiar with the following material:

1. The EXTERN directive. (See Chapter 14, "Introduction to Procedures and Functions," of the *Microsoft Pascal Reference Manual.*)
2. Procedure and function parameters. (See Chapter 14, "Introduction to Procedures and Functions," of the *Microsoft Pascal Reference Manual.*)
3. MS-Macro Assembler. See your MS-DOS manual.

9.1 Calling Conventions

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

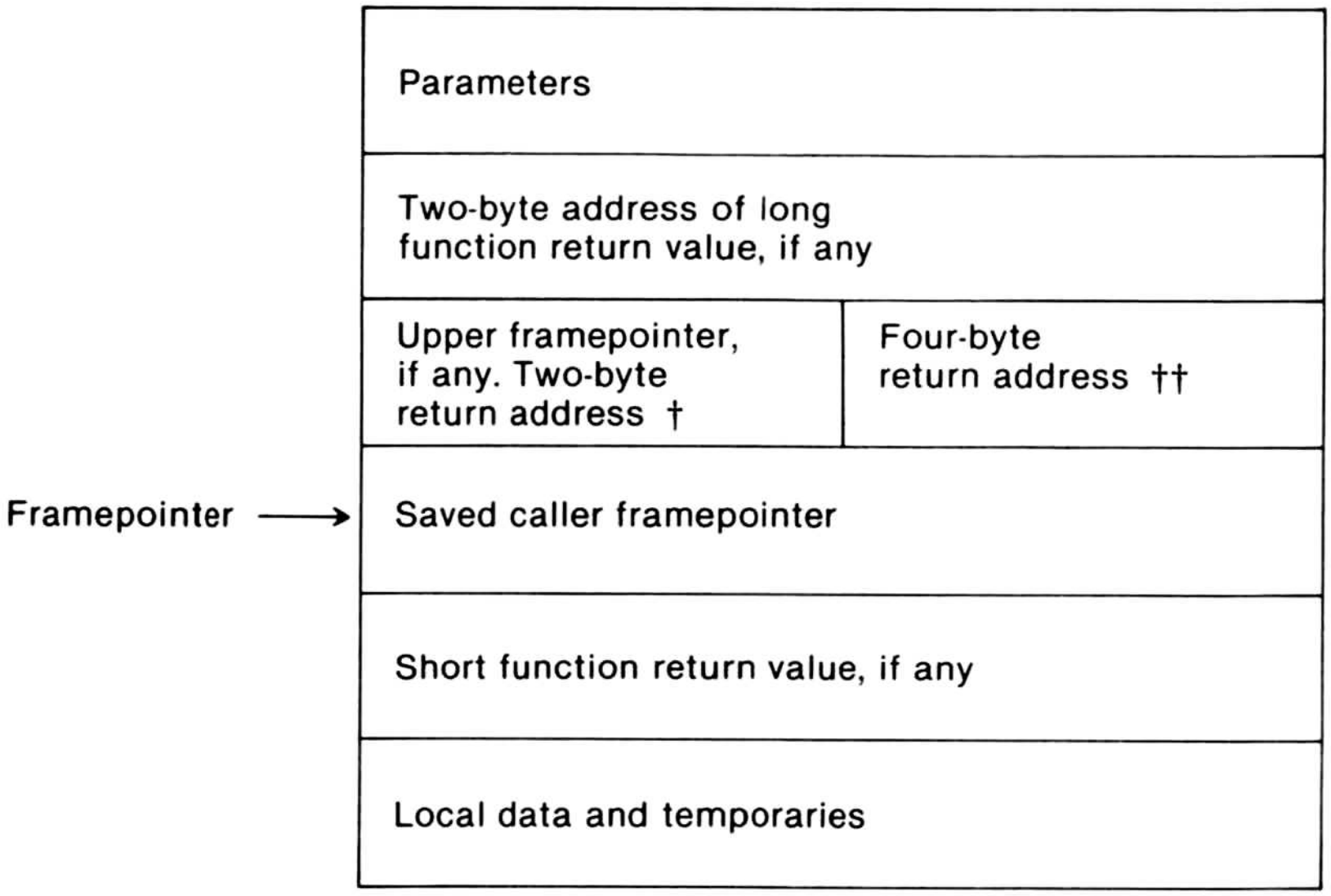


Figure 9.1. Contents of the Frame

The framepointer points at the saved caller framepointer, below the return address, and is used to access frame data. A procedure or function nested within another procedure or function has an upper framepointer, so it can access variables in the statically 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. It also checks for adequate stack space if $stackck was on.

To return to the calling routine, the called routine restores the caller's framepointer, releases the entire frame (including

parameters), 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 information on the assembly language interface, see Section 9.3, "Interfacing to Assembly Language Routines."

The standard entry and exit sequences (with $stackck−) are as follows:

```
PUSH        BP
MOV         BP,SP
{body of routine}
MOV         SP,BP
POP         BP
RET         PARAMETERSIZE
```

A function always returns its value in registers. For real types, structured types, and pointers to super arrays, regardless of length, the caller allocates a temporary frame for the result and passes the offset address to the function like a parameter. When the called routine returns, it places the address back in the normal return register (AX).

8086 and 8088 microprocessors perform a long call if the called routine is PUBLIC or EXTERN. In all other cases, they perform a short call.

The called routine must save the BP register, which contains the MS-Pascal framepointer, as well as save 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 (FLAGS, 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-Pascal data types. Programmers who use both MS-Pascal and MS-FORTRAN should pay particular attention to the data type and parameter passing differences when passing data between the two languages. For internal representations of MS-FORTRAN data types, see the *Microsoft FORTRAN Compiler User's Guide.*

1. INTEGER and WORD

   INTEGER values are 16-bit two's complement numbers, but a subrange requiring 8 bits or less (i.e., in the range −127 through 127) is allocated an 8-bit byte. WORD values are 16-bit unsigned numbers, but a WORD subrange in the range 0 through 255 is allocated an 8-bit byte. For 16-bit INTEGER and WORD values, the least significant byte has the lower, even address.

2. INTEGER4 and REAL

   INTEGER4 values are 32-bit two's complement numbers, with the least significant byte at the lowest, even address and more significant bytes at increasing addresses. There are no subranges for INTEGER4 (as there are for INTEGER2).

   IEEE 4-byte real numbers have a sign bit, 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**38 and 7 digits of precision. The maximum real number is normally 1.701411E38.

   IEEE 8-byte real numbers have a similar format, except that the exponent is 11-bits excess 1023, and the mantissa has 52 bits (plus the implied high-order 1 bit). (This gives an exponent range of 10**306 and 15 digits of precision.)

   In either case, a number with an exponent of all zeros is considered zero. An exponent of all ones is a flag for an invalid real number, or NAN ("Not A Number").

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) and 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	+.1E−63 to +.9999999999999E63
	−.1E−63 to −.9999999999999E63

4. CHAR, BOOLEAN, and enumerated types

   CHAR values and BOOLEAN values take 8 bits. CHAR values correspond to the ASCII collating sequence. For BOOLEAN values, FALSE is 0 and TRUE is 1. The low-order bit (bit 0) is generally used to check this value. Bits 1 through 7 are presumed to be 0.

   Enumerated values take 8 bits if 256 or fewer values are declared; otherwise 16 bits are declared. Values are assigned starting at 0. Subrange values take either 8 or 16 bits.

5. Reference types

   Pointer values currently take 16 bits. A pointer is an offset within the current default data segment. Other representations, such as an offset from an address kept in a global variable or an address divided by a power of two, may be used in the future. A pointer to a super array type is followed by the bounds (see item number 6 of this list), increasing the length of the pointer value (DS/SS).

   ADR and ADS are offset addresses and segmented addresses, respectively. For segmented addresses, the offset is the lower address, and the segment follows.

   The heap contains heap blocks, which may be allocated or free. A heap block contains a header WORD, with a 15-bit length (in WORDs) and the lower-order bit, which is 1 for free blocks and 0 for allocated blocks. The starting and ending heap addresses are WORD variables in BEGHQQ and ENDHQQ.

6. Procedural and functional parameters

   Procedural parameters contain a reference to the procedure or function's location along with a reference to the "upper framepointer" (a list of stack frames of statically enclosing routines). The parameter always contains two words, in one of two formats. In the first format, the first word contains the actual routine's address (a local code segment offset), and the second word contains the upper framepointer. The upper framepointer is zero if the actual routine is not nested in a procedure or function and, therefore, the routine has no upper framepointer.

   In the second format, used for segmented address targets, 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. There is never an upper framepointer in this case.

7. Super arrays

   A super array type's representation is similar whether it is a reference parameter or the referent of a pointer. First comes the address (reference parameter) or pointer value, which is either 2 or 4 bytes long. Following the address are the upper bounds, which are signed or unsigned 16-bit quantities. The bounds occur in the same order as they are declared. A pointer value to a super array type is normally longer than other pointers, since the upper bounds are included.

8. Sets

   The number of bytes allocated for a SET is:

   ```
   (ORD (upperbound) DIV 16) * 2 + 2
   ```

   This is always an even number from 2 to 32 bytes. For example,

   ```
   SET OF 'A' . . 'Z'
   ```

   requires 12 bytes. Internally, a set consists of an array of bits, with one bit for every possible ORD value from 0 to the upper bound. Bits in a byte are accessed starting with the most significant bit. The occurrence of a given ORD value as an element of a set implies the bit is 1, and the

byte and bit position of a given ORD value of any set is the same. For example, the ORD value of 'A' is 65, and the second bit (i.e., 2#01000000) of the ninth byte in a set is 1 if 'A' is in the set.

9. Files

   A FILE type in a program is a record called a file control block (of type FCBFQQ) in the file unit. The initial portion of the FCBFQQ record is standard for all files, but the remainder is available for use by the particular target file system. The end of the FCB contains the current buffer variable. The internal form of a file varies depending on the target file system. Under MS-DOS, ASCII files consist of lines followed by a carriage return and linefeed pair, which together are a "line marker." MS-DOS binary files are simply a stream of bytes.

10. Structures

    For arrays and records, the internal form is comprised of the internal forms of the components, in the same order as in the declaration. Arrays, records, variants, sets, and files always start on a word boundary. In any case, variables cannot be allocated more than MAXWORD (64K) bytes.

    A PACKED type has the same representation as an unpacked one.

    A variable or component 16 bits or larger is always aligned on a word boundary; therefore, it always has an even byte address. The only exception is when explicit field offsets are given by the user in a program.

    An 8-bit variable is also aligned on a word boundary, but an 8-bit component of a structure (array or record) is aligned on a byte boundary, which can be at an even or odd address. Currently, an array of 8-bit values starts on a word boundary. (This may change in future versions of MS-Pascal.)

Some variables are initialized automatically, whether they reside in fixed memory, on the stack, or on the heap.

1. Files (FCBFQQ records) are initialized by calling NEWFQQ, by passing the size of a text file line buffer or

binary file component, and by passing a Boolean flag value to indicate whether the file is a textfile.

2. If $initck is on, INTEGER values and their 2-byte subranges are initialized to 16#8000, 1-byte INTEGER subranges to 16#80, IEEE REAL values to 16#FFFF, and pointers to 16#0001. The following variables are never initialized by $initck, however:

   a. variables found in a VALUE section

   b. variant fields in a record

   c. super arrays allocated on the heap

The compiler generates the extra code necessary to initialize stack and heap variables.

9.3 Interfacing to Assembly Language Routines

In general, interfaced procedures and functions are declared EXTERN in the MS-Pascal source. When an EXTERN procedure or function is called, actual parameters are pushed on the stack in the order in which they are declared. If a parameter is a value parameter, an actual value is pushed on the stack.

If a parameter is a VAR or CONST reference parameter, the address of the variable is pushed on the stack. Only the two-byte offset is pushed, and not the segment. The offset is within the default data segment, DS (where SS = DS).

In contrast, a VARS or CONSTS parameter includes both a two-byte segment and a two-byte offset, with the segment pushed first.

Super array reference parameters include their upper bounds, pushed as value parameters before the address is pushed. For multidimensional super arrays, bounds are pushed in reverse order (i.e., the last flexible bound is pushed first).

For some functions, a final, hidden offset address for the return value temporary variable is pushed last.

After all parameters have been pushed, the return address for PUBLIC and EXTERN procedures is pushed by a far call instruction. The return address is segmented, so the segment is pushed first, followed by the offset. This is the general starting state of the stack for any assembly language routine that wishes to access parameters. For example, assume that you have created and compiled the following program, which contains the EXTERN function ADD:

```
PROGRAM ASM_INTERFACE (INPUT, OUTPUT);
VAR I, TOTAL : INTEGER;
FUNCTION ADD (VAR A : INTEGER; B : INTEGER):
        INTEGER; EXTERN;
BEGIN
        I := 10;
       TOTAL := ADD (I, 15);
        WRITELN (OUTPUT, TOTAL)
END.
```

When the program executes the ADD function at runtime, it sets up the stack as shown in Figure 9.2.

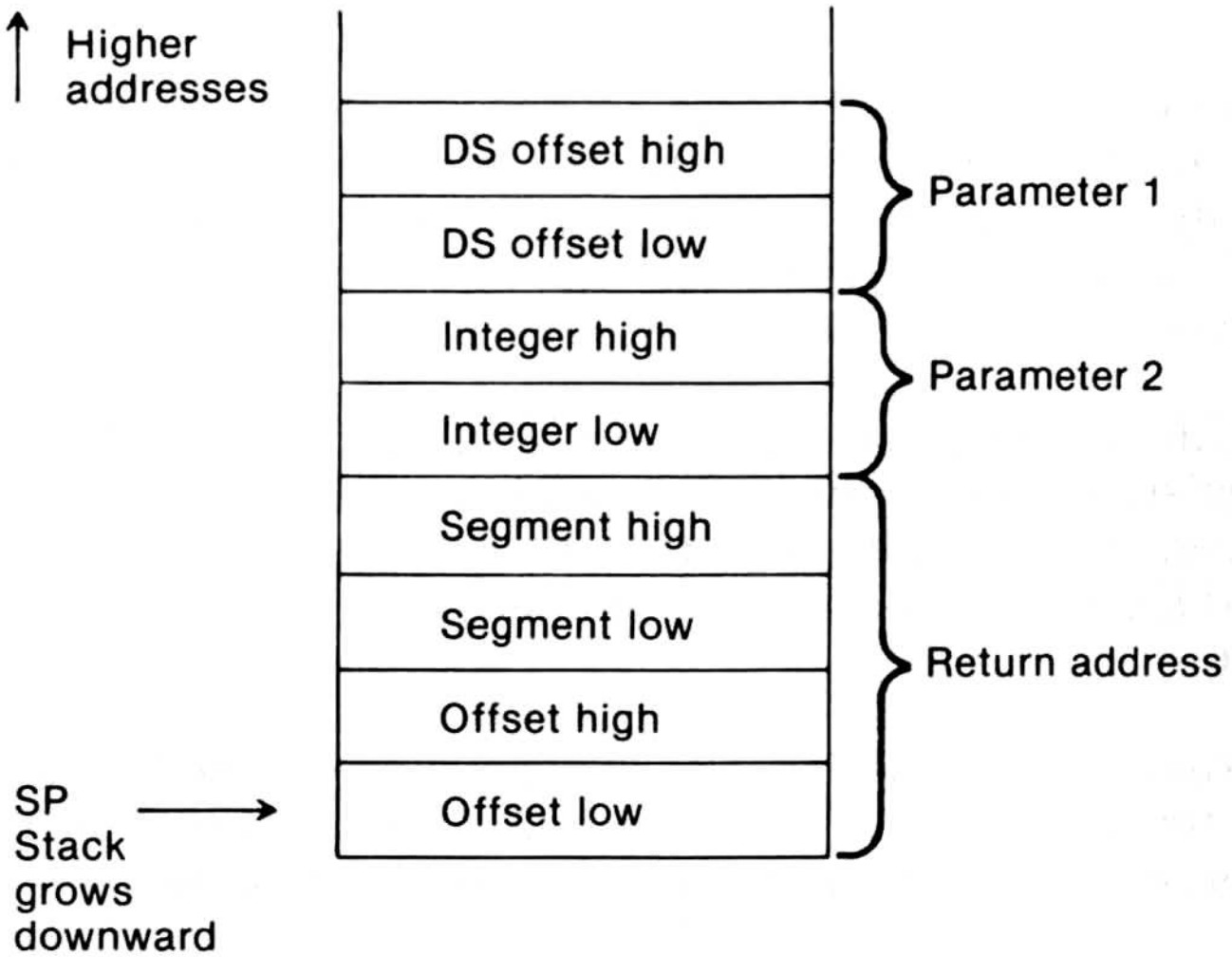


Figure 9.2. Stack Before Transfer to ADD

Before you could run such a program, however, you would have to link it to a routine that implements the ADD function. Implementation of ADD in assembly language might look like this:

```
DATA    SEGMENT PUBLIC      'DATA'
                            ;PUBLIC and EXTERN data
                            ;declarations go here.
DATA    ENDS
DGROUP  GROUP DATA
        ASSUME CS : ADDS,DS : DGROUP,SS : DGROUP

ADDS    SEGMENT 'CODE'
PUBLIC  ADD
ADD     PROC FAR
        PUSH  BP            ;Save framepointer on stack
        MOV  BP,SP          ;Address parameters
        MOV  AX,6[BP]       ;AX := value of B
        MOV  BX,8[BP]       ;BX := address of A
        ADD  AX,[BX]        ;AX := integer A + integer B
        POP  BP             ;Restore framepointer
        RET  4              ;Return, pop 4 bytes
ADD     ENDP
ADDS    ENDS
        END
```

Remember that when an EXTERN procedure or function is called at runtime, parameters are pushed on the stack. An assembly language routine must rely on these pushed parameters being in a certain sequence and format. It must also remove all parameters from the stack before returning.

Assembly language routines must save and restore the BP and DS registers. They must not even modify the SS register. However, the remaining registers (FLAGS, AX, BX, CX, DX, SI, DI, and ES) can be changed by the assembly language routines as needed.

If the routine is a function, the return value is placed in registers. If the return value is a one-byte value, it is placed in the AL register, as shown in Figure 9.3. AH need not be set.

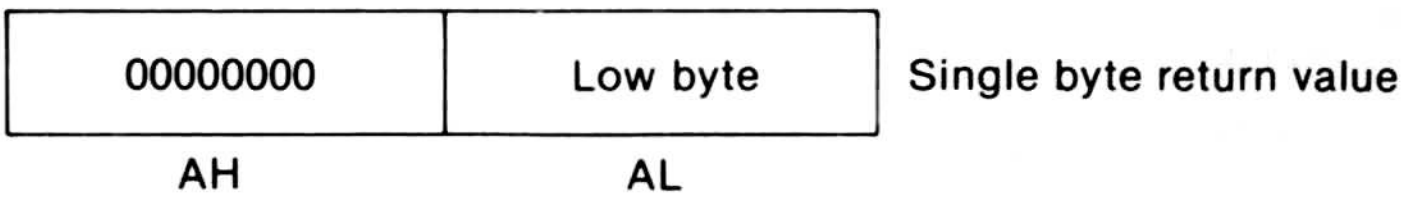


Figure 9.3. One-Byte Return Value

If the return value is a two-byte value, the returned value is placed in the AX register pair, high byte in AH and low byte in AL, as shown in Figure 9.4.

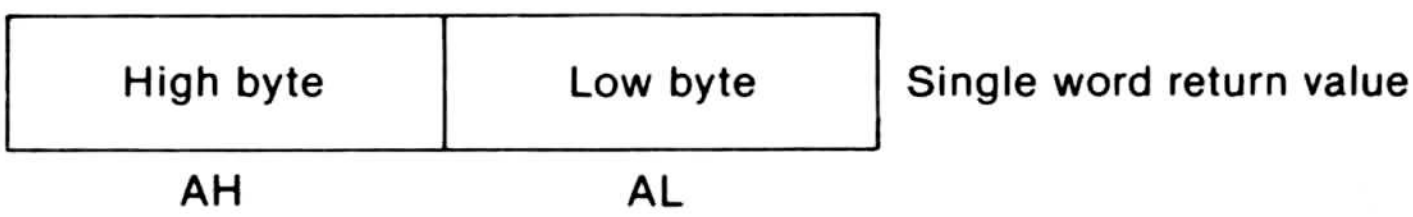


Figure 9.4. Two-Byte Return Value

If the return value is a four-byte value, the high part (or segment) of the return value is placed in the DX register and the low part (or offset) in the AX register. (This is sometimes shown as DX:AX.) Note that this only applies to INTEGER4 and ADS types.

Since MS-Pascal permits structured values to be retrieved by a function, it is possible for the return value's size in bytes to be extremely large. Therefore, for all function returns of any real or structured type (REAL4, REAL8, array, record, or set) or of a pointer to a super array type, the compiler allocates its own temporary variable. This occurs even if the size of the return value is 1, 2, or 4 bytes. The address of this temporary variable is pushed on the stack after all parameters, just before the return address is pushed, as shown in Figure 9.5. (This address is an offset, and therefore only one word is pushed.)

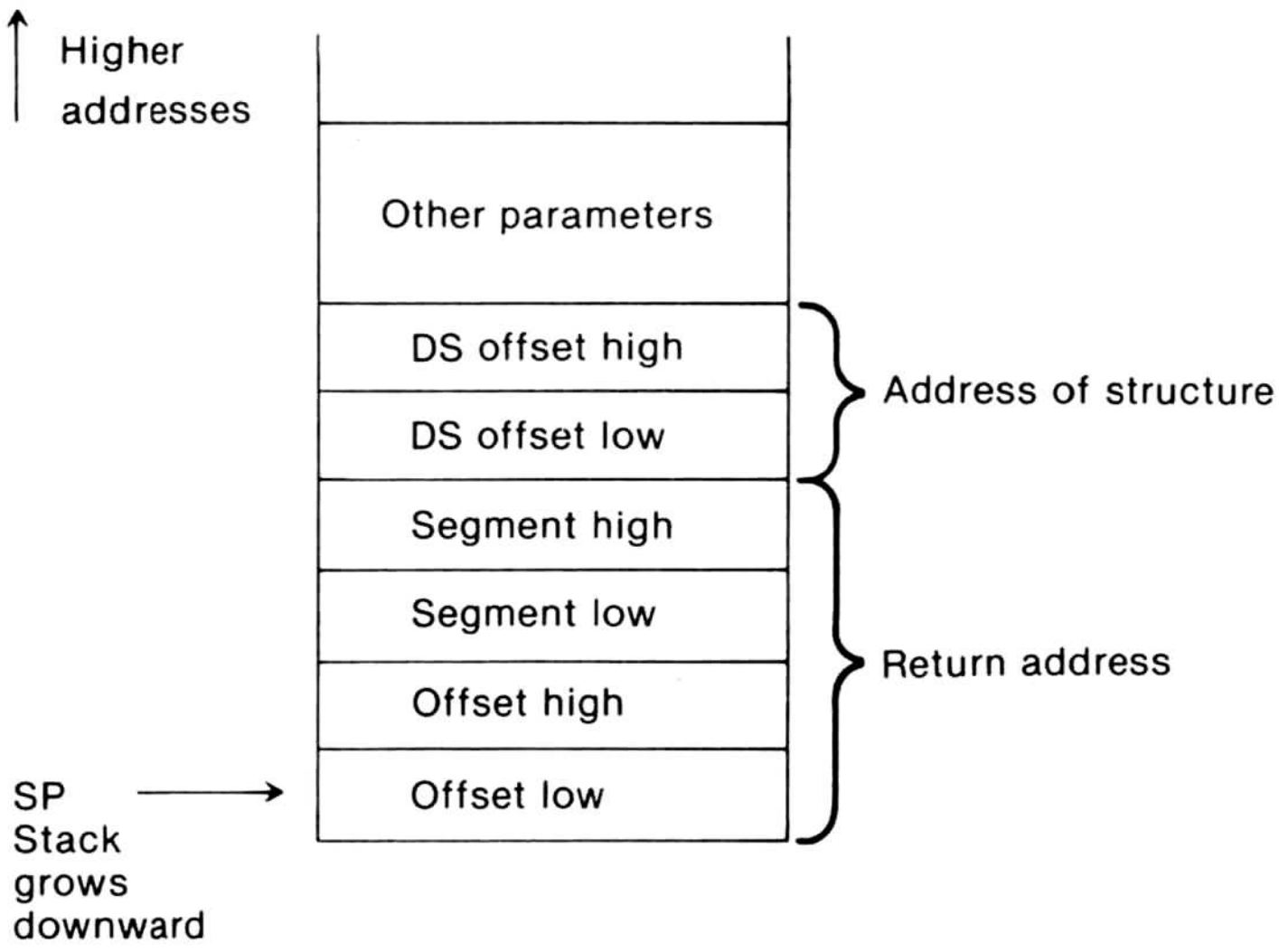


Figure 9.5. Four-Byte Return Value

On exit from the function, the address of this temporary variable should be placed in the AX register in lieu of the full structure. This address is simply an offset returned in the AX register.

You may wish to pass data using PUBLIC and EXTERN variables instead of parameters. If so, these variable declarations go into a segment named DATA with class name 'DATA', in group DGROUP. It is important that you give the correct segment, class, and group names, as shown in the last example.

Chapter 10
Advanced Topics

This chapter contains advanced technical information that will be of interest primarily to experienced programmers. Since Microsoft Pascal and Microsoft 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 illustrates the basic structure of the compiler and its relationship to the files that it reads and writes.

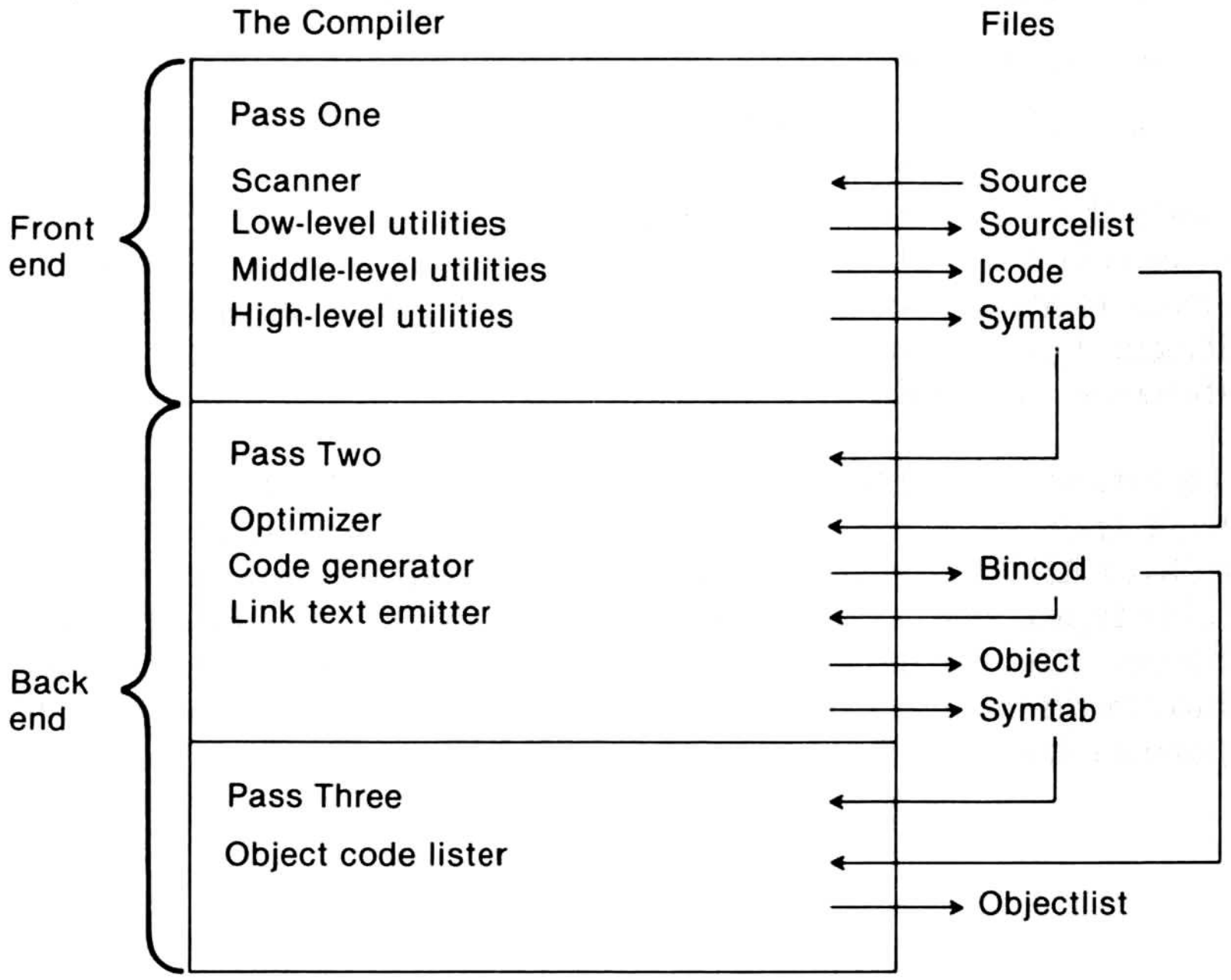


Figure 10.1. The Structure of the Microsoft Pascal Compiler

Pass one, which normally corresponds to a file named PAS1.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 (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 Microsoft Pascal, in a source format that can be transformed into either relatively standard Pascal or into system level Microsoft Pascal. For information on these levels, see the *Microsoft Pascal Reference Manual.*

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 (assorted data types, variables, procedures and functions). 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 Microsoft Pascal front end can be divided into several parts:

1. the scanner
2. low-level utilities
3. intermediate-level utilities for identifiers, symbols, Icodes, memory allocation, and type compatibility
4. high-level routines for processing procedure and function calls, expressions, statements, and declarations

The front end is driven by recursive descent syntax analysis, using a set of procedures such as EXPR (for expressions), STATEMT (for statements), and TYPEDEC (for type declarations).

The front end maintains a "current" symbol and a "lookahead" symbol. While not necessary for parsing correct programs, these symbols are useful for error recovery. Syntax errors are processed by a procedure that forces the current symbol to one of a set of symbols legal at a given point. If the current symbol is wrong, but the following one is correct, the current symbol is deleted. In all cases the correct symbol is inserted if possible. However, common substitution mistakes, such as confusing (=) and (:=), cause only a warning message to be given during compilation.

The scanner is relatively large, since it must process meta-language and produce a listing with error messages, data about variables, and other information for the user.

Intermediate code is written to the Icode file on disk as soon as it is generated: there is no reason to keep it in memory. The symbol table is built as a binary tree of identifiers with pointers to semantic records. At the end of each block, all new semantic records are written to the symbol table file. When an error is detected, all writing to intermediate files stops, since the code may not be acceptable to the back end. Detecting a warning, rather than an error, does not invalidate the intermediate files.

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. Pass two produces the object file, while pass three produces the object listing.

10.1.2.1 Pass Two

The optimizer reads the interpass files in the following order: first the symbol table for a block is read; then the intermediate code for the block. Optimization is performed on each "basic block," that is, 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 this block, the optimizer can reorder and condense expressions so long as the intent of the programmer 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;
{J := J - 1;}
IF A [J, K] = MAX THEN PUNT;
```

However, if the preceding program 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, then it gives up, and 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 must handle 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 must also keep track of register contents, and several memory segment addresses (code, static variables, constant data, etc.). The code generator also allocates any needed temporary variables.

The code generator writes a file of binary intermediate code (BINCOD), which contains actual byte values for machine instructions, symbolic references to external routines and variables, and other kinds of data. A final internal pass reads the BINCOD file and writes the object code file.

10.1.2.2 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, it writes the generated code in an assembler-like format.†

10.2 An Overview of the File System

Since Microsoft Pascal and Microsoft FORTRAN share the same file system, this section includes references to differences between the two, wherever they exist. Microsoft Pascal and Microsoft FORTRAN 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 Microsoft Pascal or Microsoft FORTRAN at runtime to perform input and output

---

† 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 C. Lewis, *Electronic Design*, May 14, 1981).

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 that are independent of the operating system. Following these standard fields are fields such as channel numbers, buffers, and other data, that are dependent on the operating system.

The advanced Microsoft 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 Microsoft FORTRAN.

Both Microsoft Pascal and Microsoft FORTRAN 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 (which you can reassign and generally treat like any other files); in MS-FORTRAN, they are unit number 0 (or *) and accessed through a variable TRMVQQ, 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.

For Microsoft Pascal files, each FCB ends with the buffer variable that contains the current file component. This means that the length of an FCB in Microsoft Pascal is the length of its fixed portion plus the length of the buffer variable. Microsoft FORTRAN files do not require buffer variables, so all are of a fixed length.

File control blocks always reside in the default data segment, so they can be referenced with the offset (ADR) addresses instead of the segmented (ADS) addresses.

Microsoft Pascal file variables can occur:

1. in static memory
2. on the stack as local variables
3. in the heap as heap variables

In Microsoft Pascal, generated code initializes file control blocks when they are allocated and CLOSEs them when they are deallocated. FORTRAN files are allocated during OPEN and deallocated during CLOSE or at program termination.

The manner of allocation and deallocation depends on the operating system. For example, a fixed number of file "slots" may be available, or the routines for Microsoft Pascal heap allocation may be used. In both Microsoft Pascal and Microsoft FORTRAN, an FCB can be created or destroyed, but never moved or copied.

The Microsoft Pascal Compiler must know enough about an FCB to allocate one. Thus, it needs to know the length of an FCB less the length of its buffer variable.

The Microsoft FORTRAN Compiler itself does not allocate files, so it doesn't need to know the length of an FCB.

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 Microsoft FORTRAN are called Unit V. Code generated by the compiler of either language contains calls to Unit F (Microsoft Pascal) or Unit V (Microsoft FORTRAN), which in turn call Unit U routines. This relationship is shown schematically in Figure 10.2.

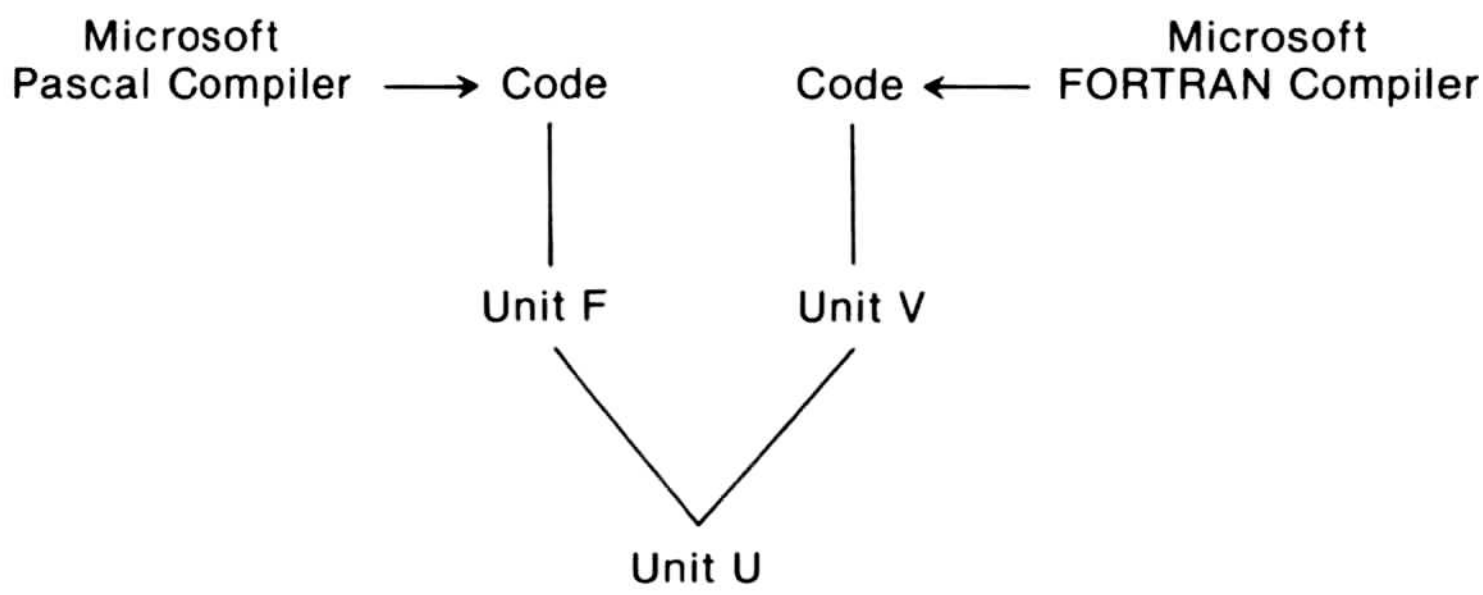


Figure 10.2. The Unit U Interface

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

File system error conditions may be detected at the lower Unit U level, detected at the higher Unit F or V level, or 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 Microsoft Pascal with the TRAP flag, in Microsoft FORTRAN with the ERR=nnn or IOSTAT=var clauses).

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 to terminate use the standard error-handling system, which gives the context of the error and provides entry to the target debugging system.

The distributed implementation of the Microsoft Pascal Compiler includes the following three source files:

1. FINU contains procedure and function headers for all Unit U routines.

2. FINK contains the common FCB declarations for all MS-Pascal systems, along with the declaration of the FILEMODES type.

3. FINKxx contains the FCB declarations as extended for use in a particular environment. For the MS-DOS version 1.0 environment, the name is FINKXM. For the MS-DOS 2.0 environment, the name is FINKXU. (These extensions are currently the same for MS-DOS, CP/M-80™, and CP/M-86® environments.)

The Microsoft Pascal Compiler runtime package supports MS-DOS 1.0 (2.0 compatible) I/O. A library named DOS2PAS.LIB is supplied with the Pascal Compiler. When this library is linked with a Pascal program, the program will use MS-DOS 2.0 I/O. In particular, the 2.0 MS-DOS Pascal I/O system takes advantage of pathnames and file handles.

10.3 Runtime Architecture

This section describes several topics related to the runtime structure of Microsoft Pascal and Microsoft FORTRAN, with mention of the languages' differences where they exist.

10.3.1 Runtime Routines

Microsoft Pascal and Microsoft FORTRAN 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 10.1 shows the current unit identifier suffixes.

Table 10.1

Unit Identifier Suffixes

Suffix	Unit Function
AQQ	Complex real
BQQ	Compile time utilities
CQQ	Encode, decode
DQQ	Double precision real
EQQ	Error handling
FQQ	Microsoft 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	Microsoft 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 Microsoft 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.

Microsoft Pascal and Microsoft FORTRAN 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; its 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 all static variables, constants which reside in memory, the stack, the heap, and the segmented addresses of Microsoft FORTRAN blank common and named common blocks. The blank and named common blocks themselves reside in their own segments, not in DGROUP.

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 blank common and named common, and other data segments

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

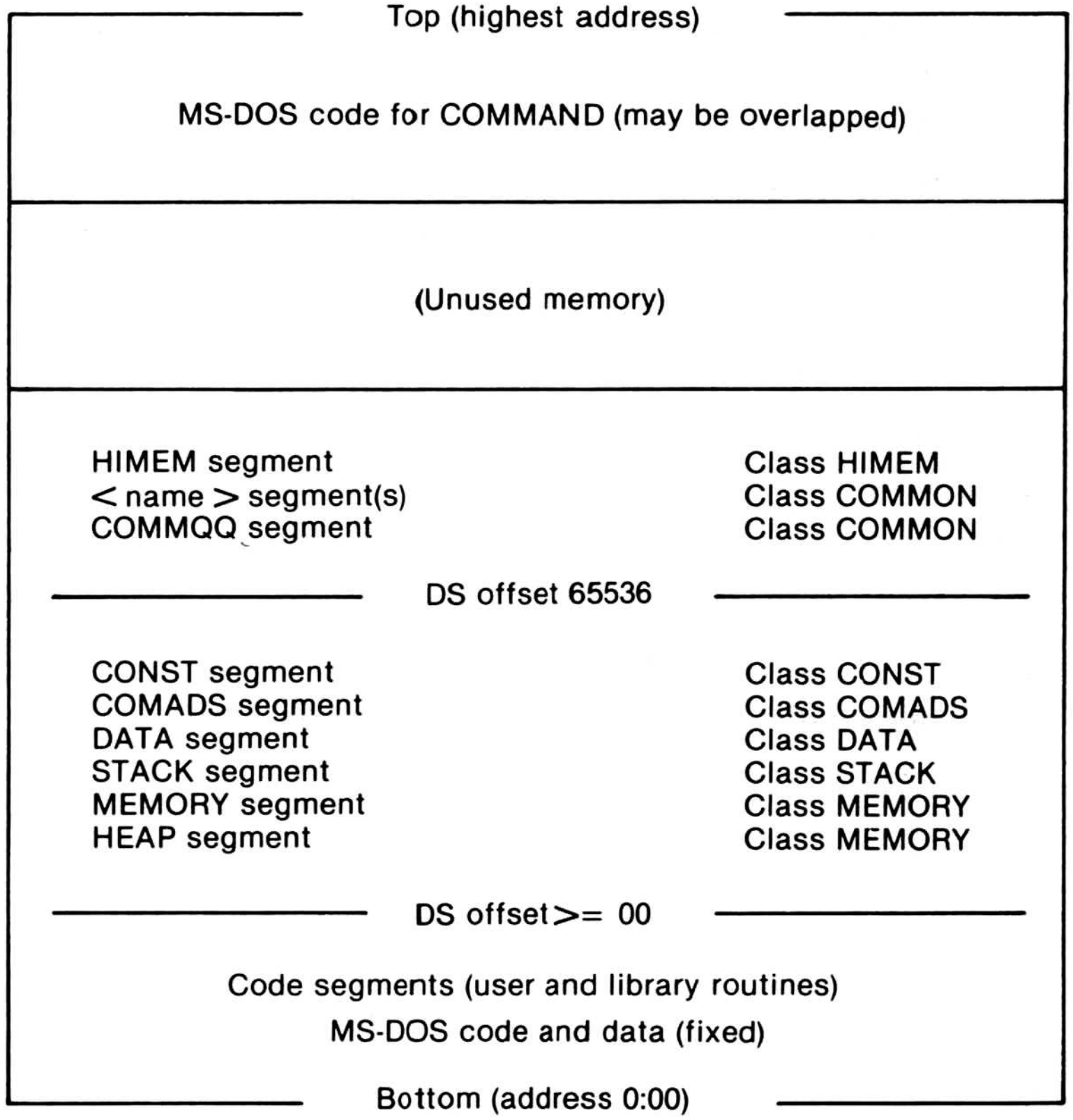


Figure 10.3. Memory Organization

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 addi-

tional 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 a Microsoft Pascal or Microsoft FORTRAN 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

   Next comes the DGROUP data area, containing 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 named commons
CONST	CONST	Constant data

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

4. DGROUP:TOP

   Here TOP means 64K bytes (4K paragraphs) above DGROUP:0000 (i.e., just past the end of DGROUP). Microsoft FORTRAN blank and named common blocks and other data segments generated for FORTRAN start here. Named common has a segment name as declared in the Microsoft FORTRAN program as the common block name, and the class name COMMON. Each blank and named common block has one segmented (ADS) address in the COMADS segment in DGROUP. All references to common block 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 Microsoft 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 Microsoft Pascal or Microsoft FORTRAN 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-Pascal or MS-FORTRAN 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-Pascal or MS-FORTRAN 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-Pascal or MS-FORTRAN 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 Microsoft Pascal or Microsoft FORTRAN. 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 INIFQQ 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 Pascal Program Structure

10.3.3.1 Machine Level Initialization

The entry point of an MS-Pascal 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 at the end of the introduction to this chapter), creating a gap for the stack and the heap. It sets the stackpointer to the top of this area. The initial stackpointer is put into PUBLIC variable STKBQQ and is used to restore the stackpointer after an interprocedure GOTO to the main program.

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, indirectly through segment EINQQQ. 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.

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 Microsoft Pascal program.

10.3.3.2 Program Level Initialization

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

If the main program is in Microsoft Pascal, and MS-FORTRAN file routines will be used, you must call INIVQQ to initialize the MS-FORTRAN file system. If the main program is in Microsoft FORTRAN, and MS-Pascal file routines will be used, you must call INIFQQ to initialize the MS-Pascal file system.

MS-Pascal main programs automatically call INIFQQ; MS-FORTRAN main programs automatically call INIVQQ. To avoid loading the file system, you must provide an empty procedure to satisfy one or both of these calls.

After file initialization, ENTEQQ is called to set the source error context (but only if $entry is on during compilation). Next, each file at the program level gets an initialization call to NEWFQQ.

After static data initialization comes unit initialization. Every USES clause in the source, including those in INTERFACEs, generates a call to the initialization code for the unit.

Units may or may not contain initialization code. If the interface contains a trailing pair of BEGIN and END statements, then initialization code in the implementation is presumed. Units are initialized in the order that the USES clauses are encountered.

Finally, any program parameters are read or otherwise initialized, and your program begins. Program parameters are set in one of a number of ways, depending on the target operating system. In general, except for INPUT and OUTPUT, PPMFQQ

is called for each parameter to set the parameter's string value as the next line in the file INPUT. Then one of the READFN routines "reads" and decodes the value, assigning it to the parameter. The parameter's identifier is passed to PPMFQQ for use as a prompt. PPMFQQ first calls PPMUQQ to get the text of any command line parameter or other parameters specific to the operating system. If PPMUQQ returns an error, then PPMFQQ does the prompting and reads the response directly.

10.3.3.3 Unit Level Initialization

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 because of an error condition, either with a user call to ABORT 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 can be declared in an external procedure and 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 file unit terminator that is operating system specific. Finally, ENDXQQ calls ENDX87 to terminate the real number processor (8087 or emulator) indirectly through segment EINQQQ. 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. The main initialization

and termination routines are in module ENTX. Procedures for BEGOQQ and ENDOQQ are in module MISO. ENDYQQ is in module ENDY.

10.3.4 Error Handling

Runtime errors are detected in one of four ways:

1. The user program calls EMSEQQ (i.e., ABORT).
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. The error type has three components:

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

In Microsoft Pascal, the message describes the error, and the number can be used to look up more information (see Appendix H, "Messages," in the *Microsoft Pascal Reference Manual*). In Microsoft FORTRAN, 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 Microsoft FORTRAN, the error status value is used for special purposes and has no significance for the user. In Microsoft Pascal, the error status value is undefined, although for file system errors it may be an operating system return code. However, the error status value may also be used for other special purposes. Table 10.2 shows the general scheme for error code numbering.

Table 10.2

Error Code Classification

Range	Classification
1- 999	Reserved for user ABORT calls
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 and double real 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 metacommands. If the $entry metacommand is on, 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 (program, unit, module, procedure, or function)
3. the line number of the routine in the listing file
4. the page number of the routine in the listing file

If the $line metacommand is also on, the line number of the statement containing the error is also given. Setting $line also sets $entry.

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 by 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 10.3.

Table 10.3

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 (a WORD)
2. the page number of the procedure (a WORD)
3. the procedure or function identifier (an LSTRING)

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. If one name is given in an INTERFACE and another in a USES clause, the USES identifier is used. The line and page are those designated by the procedure header.

Entry and exit calls are also generated for the main program, unit initialization, and module initialization, in which case the identifier is the program, unit, or module.

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 (because $line implies $entry), 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. The $line+ metacommand should be placed at least a couple of symbols before the start of the first statement intended for a line number call ($line− also takes effect "early").

Most of the error handling routines are in modules ERRE and PASE. 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 Pascal 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 is provided in the standard floating-point runtime library, MATH.LIB. It 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.

The real math package is also compatible with the 8087 numeric coprocessor. 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.)

Microsoft Pascal 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 $floatcalls– Option

The $floatcalls– metacommand directs the compiler to generate in-line instruction "skeletons" for floating-point operations. With $floatcalls– in your source code and the standard version of MATH.LIB linked into your program, fixups in MATH.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 coprocessor 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 per cent.

$floatcalls– 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 $floatcalls– 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 fast math pack ALTMATH.LIB. You will get linker errors if you try.

10.4.2 The Alternate Math Package

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 Package" (AltMath Pack). This is a 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.

The AltMath Pack assumes a much simpler model for floating-point arithmetic than the IEEE standard, although the external binary 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") error message in the IEEE model, will cause an error exit in the AltMath Pack model. Also, unlike the 8087 and emulator models that assume (and support) an infinite stack, the AltMath Pack assumes a fixed stack with a limited number of entries. This means that highly recursive functions may overflow the stack and generate an error.

You select the AltMath Pack by linking with the library ALTMATH.LIB, which is provided with the compiler. This library contains only the AltMath Pack, and the remainder of the runtime is obtained from PASCAL.LIB. See Section 6.1.2, "Linking Libraries," for a review of the procedure for linking auxiliary libraries. The following examples are equivalent:

```
A> LINK your module,,,altmath
A> LINK your module,,,altmath + pascal
```

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 eliminate the emulator altogether if you know that your program will only run on machines that have an 8087.

You do this by linking 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 linktime. $floatcalls interfaces are provided which use the 8087 to carry out the operation, so that you can use 8087.LIB whether or not you have used $floatcalls.

Note

You cannot use 8087.LIB and ALTMATH.LIB in the same program.

10.4.4 Decimal Math Option

Microsoft Pascal 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 meta-command in all of your program units that use floating-point. You must link with DECMATH.LIB to support this format.

Note

Decimal floating-point and IEEE floating-point *are not* compatible.

10.4.5 Loading the Emulator Kernel Transcendentals

The emulator is capable of emulating 8087 transcendental functions. However, if you write functions in 8087 assembly language that use 8087 transcendental instructions, the emulator is not guaranteed to be present. To ensure that the emulator will be loaded with your program, you must add the following segment and variable to your assembly code.

```
DATA SEGMENT PUBLIC 'DATA'
EXTRN TUGRQQ : WORD
DATA ENDS
```

The above lines need not be added to assembly code if calls are made to the library transcendental intrinsics.

In addition to the above precaution, assembly language programs containing 8087 instructions and intending to use the emulator library must be assembled with the most recent version of the Microsoft Macro Assembler using the assembler's "/E" switch.

10.5 MS-DOS 2.0 Issues

This version of the Microsoft Pascal Compiler is essentially an MS-DOS 1.25 compiler. This means that Microsoft Pascal and programs compiled by it, will run on both versions of MS-DOS, but cannot take advantage of MS-DOS 2.0 features 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 Pascal file system, DOS2PAS.LIB, which contains the interface to MS-DOS 2.0 file system. The modules contained in DOS2PAS.LIB provide the interface described in Section 10.2, "An Overview of the File System," of this *User's Guide.*

10.5.1 Interface to MS-DOS 2.0 File System

However, an additional interface is provided to allow the MS-DOS 2.0 "handle" of a Pascal file to be obtained. (A handle is an integer value recognized by the operating system as representing a file.) The function:

```
FUNCTION HDLUQQ (VAR F : FCBFQQ) : INTEGER;
```

returns the MS-DOS 2.0 file handle for a Pascal file. Consult the *Microsoft Pascal Reference Manual*, Sections 8.6, "File I/O: Extend Level," and 8.7, "File I/O: System Level," and the version specific interface unit, FINKXM, on your disk for the definition of FCBFQQ.

Programs linked with DOS2PAS.LIB, will give the runtime message

```
Incorrect DOS version
```

when run on earlier versions of MS-DOS.

10.5.2 Exit Status Available To MS-DOS 2.0

The compiler supplies an exit status to MS-DOS 2.0 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 user can set the global word DOSEQQ (defined in module ENTX) to any error code desired. For example,

```
PROGRAM FOO;
        VAR DOSEQQ [ EXTERN ] : WORD;
        BEGIN
            {Set DOSEQQ to 0, i.e. No errors encountered.}
        DOSEQQ := 0
        END.
```

The value of DOSEQQ is passed to MS-DOS and this becomes the argument to ERRORLEVEL.

Appendices

Appendix A
Differences Between Versions 3.2 and 3.3

A.1 Using ADS and ADR With Expressions

The ADS operator gives the full address (segment selector and offset) of the object to which it is applied. ADR gives the offset only. Version 3.2 used an incorrect interpretation of precedence when these operators were used to address expressions. Version 3.3 uses the correct interpretation, which is described in the *Microsoft Pascal Reference Manual*. The ADS and ADR operators are of the same precedence as the NOT operator, and are of higher precedence than the other Pascal operators.

If you have existing programs that use the incorrect precedence used in version 3.2, you must correct those programs if you want to compile them with version 3. 3. For example, in version 3.3,

```
ADS a + b
```

is interpreted as

```
(ADS a) + b
```

(because the ADS operator is of higher precedence than the + operator) and will not compile. You cannot add b to the address of a.

Version 3.2 interpreted

```
ADS a + b
```

as

```
ADS (a + b)
```

(incorrectly giving the addition operator higher precedence) which does compile. You can take the address of (a + b).

If you have existing programs that take the address of an expression with either ADS or ADR, your program may not compile with version 3.3 of the Microsoft Pascal Compiler. If so, put parentheses around those expressions.

The expressions

```
ADS a
```

and

```
ADS (a)
```

where a is a function, represent different values. The first expression evaluates to the address of the function (see the new predefined type ADSFUNC in Section A.2), and the second evaluates to the address of the result of the expression (the address of the value returned when a is called).

A.2 Addressing Procedures and Functions (ADSPROC and ADSFUNC)

The ADS operator can now be applied to procedures and functions, as well as to variables. When ADS is applied to a procedure, it produces a value of the predefined type ADSPROC. When it is applied to a function, it produces a value of the predefined type ADSFUNC. ADSPROC and ADSFUNC are similar in concept to ADSMEM.

ADSPROC and ADSFUNC can be used to declare variables or formal parameters. To call a procedure or function with these variables or parameters, you must pass their value to an external, non-Pascal routine and then call that routine. Note that ADSPROC and ADSFUNC are compatible with C function pointers; Pascal procedure parameters are not.

As with other address types, there is no type checking on assignments to ADSPROC or ADSFUNC. The compiler does not make sure that a function or procedure is being assigned or that the formal parameters are appropriate. You can also freely assign to and from other address types.

Example

This example could be used to communicate with a C routine that calls the routine af, then calls the routine ap with the result.

```
program p(output);

procedure cproc (ap: adsproc; af: adsfunc) [c] extern;

procedure pproc (i: integer);
begin
    writeln('C integer =', i);
end;

function pfunc :integer;
var
    i: integer;
begin
    readln (i);
    pfunc := i;
end;

begin
    cproc (pproc, pfunc);
end.
```

A.3 File Sharing

In systems that use networking, more than one program can attempt to access the same file at the same time. Two new fields in the file control block, share and access, allow you to control access to files. These fields have the new predefined enumerated types sharemodes and accessmodes, respectively. The value of access determines how the first process to open a file can use that file. You can choose to be able to read the file, write to the file, or do both. The value of share determines how subsequent processes are allowed to access the file (while that file is still open). You can choose to allow subsequent processes to read the file, write to the file, do both, or do neither. You can also choose not to allow any processes, including the process which originally opened the file, to open the file (while the file is still open).

A.3.1 Sharemodes

To control how other processes may access a file that you are opening, set the *share* field in the file control block. This field has the new predefined enumerated type sharemodes. For example,

```
.
.
.
share:sharemodes
.
.
.
```

The sharemodes type is defined as follows:

```
TYPE
    sharemodes = (sm_COMPAT, {compatibility mode}
                  sm_DENYRW, {deny read/write mode}
                  sm_DENYWR, {deny write mode}
                  sm_DENYRD, {deny read mode}
                  sm_DENYNONE{deny none mode}
                 );
```

The sharemode values are

sm_COMPAT	Compatibility mode This is the default. When a file is open in compatibility mode, the original USER (the process that opened the file) may open the file in compatibility mode any number of times. No other USER may open the file. A file that is already open in a mode other than compatibility mode cannot be opened in compatibility mode.
sm_DENYRW	Deny read/write mode While a file is open in deny read/write mode, no process may open the file.

sm_DENYWR	Deny write mode While a file is open in deny write mode, no process may open the file with write access. Other processes can open the file with read access.
sm_DENYRD	Deny read mode While a file is open in deny read mode, no process may open the file with read access. Other processes can open the file with write access.
sm_DENYNONE	Deny none mode While a file is open in deny none mode, any process may open the file in any mode (except compatibility mode).

As with filename assignments, the share value must be set before the file is opened with reset or rewrite. For example,

```
var
f:text;
 .
 .
 .
 .
f.share;=sm_COMPAT;
assign(f.....);
reset(f);
```

If you change share, you must reset or rewrite the file before accessing the file again.

A.3.2 Accessmodes

The predefined type accessmode specifies the type of access the original process (the process that initially opened the file) will be making to a file. You specify accessmode by setting the value of the access field in the file variable.

Accessmode is defined as

```
TYPE
accessmode=(am_read,am_write,am_readwrite)
```

The accessmode values are

am_read	The process can read the file.
am_write	The process can write to the file.
am_readwrite	The process can read and write to the file.

The following example opens a file with share equal to sm_DENYRW and access equal to readwrite:

```
  .
  .
  .
var f: text;
f.share := sm_DENYRW;
f.access := am_readwrite;
assign(f,.....);
rewrite(f);
  .
  .
  .
```

If you open a file without assigning to access, the Pascal runtime system always attempts to open with an access value of am_readwrite. If the open fails, the runtime system will try to open the file again, first using am_write, then using am_read. Note that this is not the same as specifying access=am_readwrite. If you specify access=am_readwrite, and the file cannot be opened with both read and write access, the open will fail. The default behavior is most flexible.

While am_readwrite is an appropriate default for single program environments, it is not always the best choice if a file will be shared. For example, suppose several processes want to read a file, and ensure that no process updates the file while they are reading it. The first process can open the file with share=sm_denywr, and default to access=am_readwrite. The share value will prevent other processes from writing to the file, and the access value will allow the first process to read the file. But no other process will be able to open that file with share=sm_denywr because the original process has access to writing to the file. However, if the first process opens the file with share=sm_denywr, and access=am_read, any number of processes may also open the file with share=sm_denywr and access=am_read.

Table A.1 indicates the restrictions placed on opening a file that has already been opened with a particular value of share and access.

Table A.1

Share and Access Values

A File Opened With These Values of *share* and *access*:		**Can Subsequently be Opened (by any process, unless noted) With These Values of *share* and *access*:**	
share =	**access =**	**share =**	**access =**
sm_COMPAT	am_readwrite am_read am_write	sm_COMPAT *by original process only*	am_readwrite am_read am_write
sm_DENYRW	am_readwrite am_read am_write	*cannot be subsequently opened*	
sm_DENYWR	am_readwrite	sm_DENYNONE	am_read
	am_read	sm_DENYNONE sm_DENYWR	am_read
	am_write	sm_DENYNONE sm_DENYRD	am_read
sm_DENYRD	am_readwrite	sm_DENYNONE	am_write
	am_read	sm_DENYNONE sm_DENYWR	am_write
	am_write	sm_DENYNONE sm_DENYRD	am_write
sm_DENYNONE	am_readwrite	sm_DENYNONE	am_readwrite am_read am_write
	am_read	sm_DENYNONE sm_DENYWR	am_readwrite am_read am_write
	am_write	sm_DENYNONE sm_DENYRD	am_readwrite am_read am_write

If, for example, a file is opened with share = sm_DENYWR and access = am_READ, that file can also be opened with share equal to *either* sm_DENYNONE *or* sm_DENYWR, and access equal to am_WRITE.

A.4 File Locking

A new procedure and ordinal type have been defined that allow you to lock a specific range of records in a DIRECT mode file, preventing access by other processes in a networked system. The procedure locking is defined as

```
PROCEDURE locking (VAR f: FCBFQQ;
      MODE: lockmodes; M,N: INTEGER4);
```

The parameters are

M	An integer expression that is the number of the first record to be locked. If M is zero (0), the next record (the one which a sequential read, such as GET, would read) will be locked.
N	An integer expression that is the number of records to be locked. If N is zero (0), one record is locked.

The type lockmodes is defined as

```
TYPE
  lockmodes = (lm_unlck, lm_lock, lm_nblck, lm_rlck, lm_nbrlck);
```

The acceptable values of lockmodes are

lm_nblck	Non-blocking lock Lock the specified region (N records, starting at record M). If any is already locked by a different process, give an error. This is the default.

lm_unlck	Unlock
	Unlock the specified region (N records, starting at record M)
lm_lock	Lock
	Lock the specified region (N records, starting at record M). Wait for any part of the region locked by a different process to become available.
lm_rlck	Read lock
	Same as lm_lock, except locks for write access only.
lm_nbrlck	Non-blocking read lock
	Same as lm_nblck, except locks for write access only.

The following example opens a DIRECT access file with share equal to sm_DENYNONE, locks two records starting at record 1, then unlocks them:

```
type info=record...end;
var f: file of info;
f.mode:=DIRECT;
f.share:=sm_DENYNONE;
assign(f,...);
reset(f);
locking(f,lm_lock,1,2);
get(f);
locking(f,lm_unlck,1 ,2);
 .
 .
 .
```

A.5 Using the C "int" Type (INTEGERC)

Microsoft Pascal Version 3.3 supports a predefined type called INTEGERC. For a given processor and operating system, variables with the type INTEGERC are equivalent to variables with the C "int" type as defined by the Microsoft C Compiler for the same system. For the 8086 family of microprocessors, INTEGERC is equivalent to INTEGER2.

A.6 Using C Calling Conventions (C attribute)

You can use Microsoft C calling and external naming conventions for a particular procedure by specifying the C attribute in the procedure declaration. It has the same syntax as the PUBLIC attribute. For example, in the statement

```
PROCEDURE myproc [public,c];
```

myproc is a public procedure that uses C calling conventions. Calling conventions are discussed in Appendix E, "Mixed-Language Programming."

A.7 Using a Variable Number of Arguments (VARYING attribute)

VARYING can be specified with the C attribute. It means that the number of actual arguments may be different from the number of formal arguments. Actual arguments for which a formal argument is defined must follow the type rules. Actual arguments for which there are no formal arguments defined are assumed to be passed by value, with no type coercions. Note that a subprogram written in Pascal can only access arguments that are formally defined, so the latter case does not apply.

When writing a Pascal procedure with the VARYING attribute, make sure your code does not execute references to arguments that are not passed in the call, or you will get undefined results. This usually means that you must tell the procedure which arguments were passed (usually by making one of the arguments describe the others).

Note that the FORTRAN/Pascal calling sequence cannot support varying numbers of arguments; this attribute will not work unless you have also specified the C attribute on the subprogram.

A.8 Compatibility with Version 3.2

Apart from the changes identified in the previous sections, programs that compiled correctly with version 3.2 should compile correctly with version 3.3.

Object modules compiled with versions 3.2 and 3.3 can be linked together. However, your main program should be compiled with version 3.3. If you have existing assembly-language source files that rearrange memory from the default, they may have to be modified because the memory model has been changed. Refer to Section A.12, "The Microsoft Pascal Memory Model," for information on assembly-language programs.

The only change in the way the compiler is invoked is that file names are no longer changed to uppercase before being passed to the operating system. This has no effect on the behavior of the compiler because MS-DOS is not case sensitive.

If your source files are very large, this version may no longer compile them. The compiler now uses a fixed stack internally, and highly nested recursions (such as unusually long expressions) may cause stack overflow. The fixed stack also reduces the amount of memory available to contain symbol table entries. However, the Pascal compiler makes more efficient use of memory for symbol tables than did version 3.2.

The memory allocation is preset to 6144 (6k) bytes. You can use the utility EXEMOD, as described in Section A.11.2, "The EXEMOD Utility," to change the allocation.

The intermediate files produced by the first two passes of the compiler (PASIBF.BIN, PASIBF.SYM, PASIBF.TMP, and PASIBF.OID) are about one third the size of those produced by version 3.2.

A.9 Creating Link Map Files With Microsoft LIB

Now that Microsoft LIB is included with Microsoft Pascal, you can create link map files. Therefore, the .MAP files for each library are no longer included. To create a .MAP file, follow these steps:

1. Enter

   LIB

2. You receive the "Library name:" prompt. Enter the name of the library (PASCAL, MATH, 8087, DECMATH, or ALTMATH).

3. You then receive the "Operations:" prompt. Press return.

4. You receive the "List file:" prompt. Enter the name of the library, followed by the ".MAP" extension.

For example, to create a .MAP file for the MATH library, you would use a procedure of the following type:

```
LIB
Library name: MATH
Operations:
Listing file: MATH.MAP
```

You can create the same file (MATH.MAP) by entering the command line

```
LIB MATH,MATH.MAP
```

A.10 Changes to the Linker

Version 3.3 of Microsoft Pascal comes with a new version of LINK: Version 3.01. Version 3.01 has two new switches which allow you to set the maximum number of segments and use the DOS segment ordering convention. Sections A.10.1 and A.10.2 explain these new switches.

Also, the default for the overlay interrupt number has been changed from CD to 3F.

A.10.1 Setting the Maximum Number of Segments

Syntax

/SEGMENTS: *number*

The **/SEGMENTS** option directs LINK to process no more than *number* segments per program. LINK displays an error message and stops if it encounters more than the given limit. The option is used to override the default limit of 128 segments.

The *number* can be any integer value in the range 1 to 1024. It must be a decimal, octal, or hexadecimal number. Octal numbers must have a leading zero. Hexadecimal numbers must start with "0x."

If **/SEGMENTS** is not given, LINK allocates enough memory space to process up to 128 segments. If a program has more than 128 segments, set the segment limit higher to increase the number of segments LINK can process.

Minimum abbreviation: **/SE**

Example

```
LINK file.obj/SE:10,file.exe,,;
```

This example sets the segment limit to 10.

```
LINK moda+modb,run/SEGMENTS:0xff,ab.map,;
```

This example sets the segment limit to 255 (FFH).

```
LINK startup+file,file/SE:030,,;
```

This example sets the segment limit to 24 (30 octal).

A.10.2 Using DOS Segment Order

Syntax

/DOSSEG

The **/DOSSEG** option causes LINK to arrange all segments in the executable file according to the MS-DOS segment ordering convention. This convention has the following rules:

1. All segments having the class name, "CODE," are placed at the beginning of the executable file.
2. Any segments that do not belong to the group named "DGROUP" are placed immediately after the "CODE" segments.
3. All segments belonging to "DGROUP" are placed at the end of the file.

Minimum abbreviation: **/DO**

Example

```
LINK start+test/DOSSEG,test,,math+common
```

This command causes LINK to create an executable file, named "file.exe," whose segments are arranged according to the MS-DOS segment ordering convention. The segments in the object files "start.obj" and "test.obj," and any segments copied from the libraries "math.lib" and "common.lib" are arranged in the order specified above.

A.11 Modifying Executable Files With EXEPACK and EXEMOD

The EXEPACK and EXEMOD utilities (supplied with your compiler software) allow you to modify an executable program file. The EXEPACK utility "compresses" the executable file by removing sequences of null characters from the file and by optimizing the relocation table. Using EXEPACK, you can significantly reduce the size of the executable file and the time required for loading.

The EXEMOD utility allows you to examine file header information, such as the size of the file and the number of relocation entries in the relocation table, and allows you to modify the initial stack pointer and maximum and minimum allocation values. The program assumes that you are familiar with the fields and format of the file header. See your *Microsoft MS-DOS Programmer's Reference Manual* for details.

The following sections explain how to invoke and pass arguments to the EXEPACK and EXEMOD programs.

A.11.1 The EXEPACK Utility

Command

EXEPACK *executable-file output-file*

The *output-file* must have a different name than the *executable-file*.

You can also use EXEPACK by specifying a new linker switch, **/EXEPACK.**

Syntax

/EXEPACK

The **/EXEPACK** option has the same effect as the EXEPACK utility. The **/EXEPACK** option directs the linker to remove sequences of null bytes (00 hexadecimal) before the linker outputs an .EXE file.

Minimum abbreviation: **/E**

Example

```
LINK file.obj/EXEPACK,file.exe,,;

LINK file.obj,file.exe/E;

LINK file/E;
```

These examples all produce .EXE files (file.exe) with sequences of null bytes removed.

The EXEPACK utility reduces the size and loading time of an executable file by removing sequences of null characters from the given *executable-file* and optimizing the relocation table. The compressed file is written to the output file, and the original file is not modified.

The EXEPACK utility produces self-explanatory error messages if it is unable to compress the given file. For a listing of these messages, see Appendix F, "Error Messages."

A.11.2 The EXEMOD Utility

Command

EXEMOD *executable-file* [/stack *n*] [/min *n*] [/max *n*]

The EXEMOD utility modifies fields in the header according to instructions given on the command line. To display the header fields without modifying them, give the *executable-file* without any options.

The options are shown with the forward slash (/) option character, but a hyphen (-) may be used instead if you prefer. They can be given in either uppercase or lowercase. The options have the effects listed below.

Option	Effect
/stack *n*	Sets the initial SP (stack pointer) value to *n*, where *n* is a hexadecimal value in bytes. The minimum allocation value is adjusted upward if necessary. There is no default for *n*.
/min *n*	Sets the minimum allocation value to *n*, where *n* is a hexadecimal value in paragraphs. The actual value set may be different from the requested value if adjustments are necessary to accommodate the stack. There is no default for *n*.
/max *n*	Sets the maximum allocation to *n*, where *n* is a hexadecimal value in paragraphs. The maximum allocation value must be greater than or equal to the minimum allocation value. There is no default for *n*.

Notice that the modifications you can make with the /stack and /max options can also be made by relinking the program with the corresponding linker options (/STACK and /CPARMAXALLOC). The advantage of the EXEMOD utility is that it modifies the executable file directly, without requiring the program to be relinked.

The EXEMOD program produces self-explanatory error messages when it is unable to carry out the given instructions. For a listing of these messages, see Appendix F, "Error Messages."

Warning

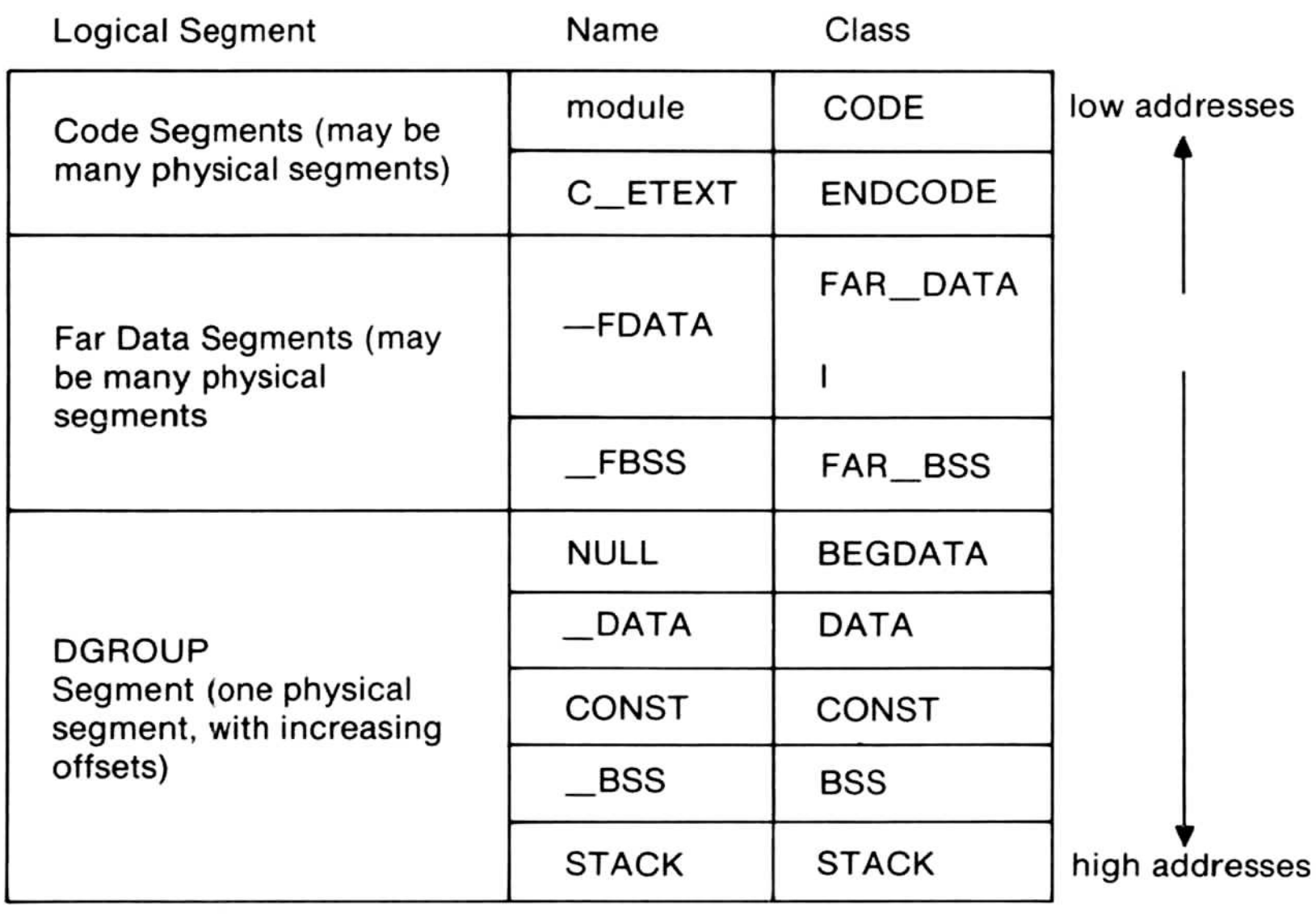
The /stack option can only be used on programs compiled with the Microsoft C Compiler Version 3.0 or later, the Microsoft Pascal Compiler Version 3.3 or later, or the Microsoft FORTRAN Compiler Version 3.3 or later. Use of the /stack option with other programs may cause the program to fail.

A.12 The Microsoft Pascal Memory Model

The memory model has been changed for version 3.3. If you use assembly language with Microsoft Pascal, you may have to modify your code.

The following illustration shows a summary of the memory model.

User programs can declare any additional segments they need, as long as they use the class names defined here to ensure they are loaded in the proper place in memory. (Only advanced users will ever need to do this.) The compiler often generates additional segments of the classes defined here for its own purposes.

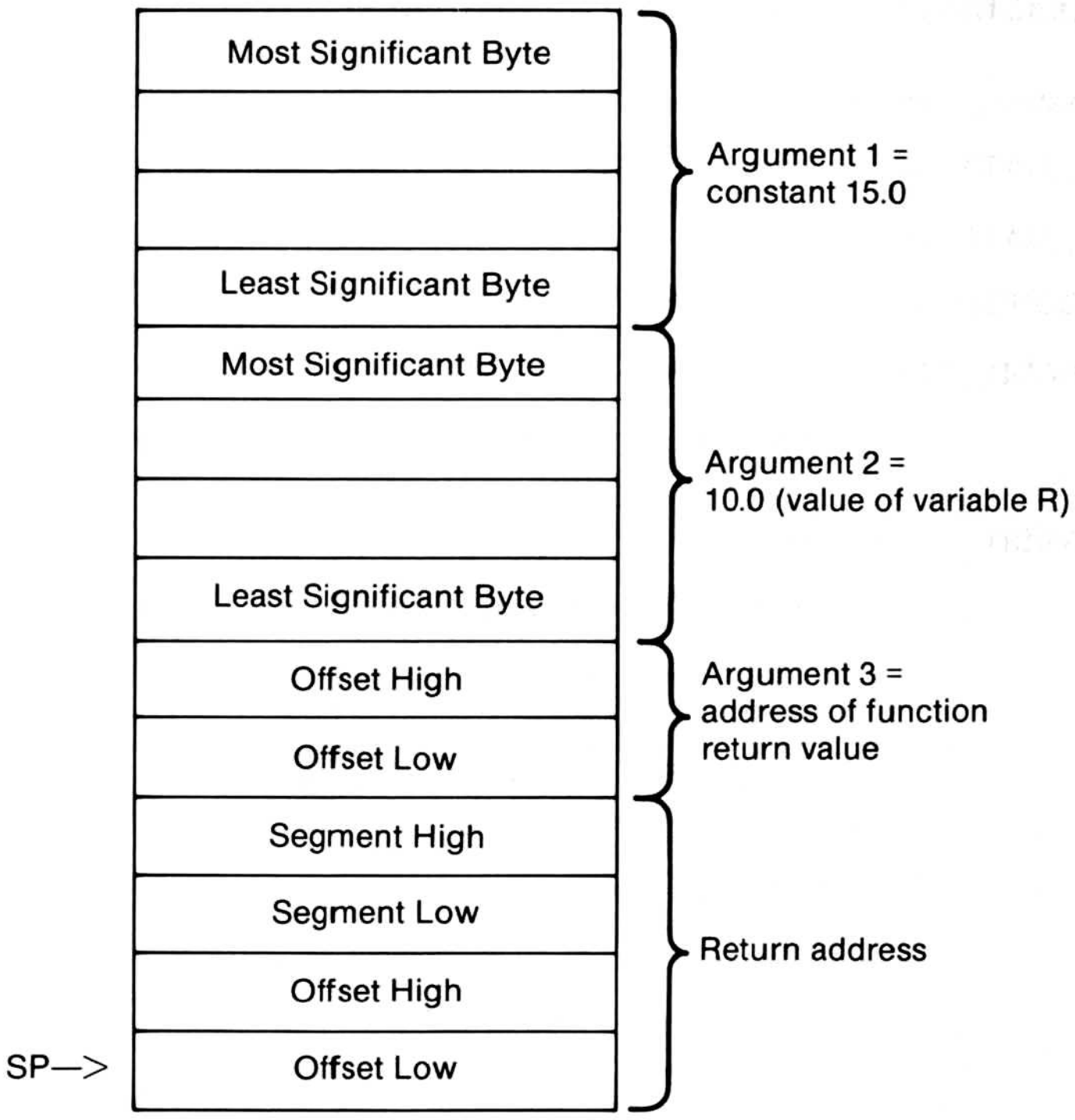


The following example is a Pascal routine that calls a routine written in assembly language for the 8086. The assembly-language routine adds real numbers. Note that to assemble this example, you need the Microsoft Macro Assembler version 1.25 or higher. This example should be assembled with the /E switch, which directs the assembler to generate 8087/80287 instructions that are compatible with the emulator. If you have version 3.0 or later of the Microsoft Macro Assembler, you should also specify the /MX switch, which tells the assembler to preserve lower case letters in public and external symbols when the object file is written.

Assume the following Pascal program has been compiled:

```
PROGRAM example2(input, output);
VAR
            r: REAL;
            total: REAL;
FUNCTION RADD (a,b: REAL): REAL;
            EXTERN;
BEGIN
            r := 10.0;
            total := RADD(15.0,r);
            WRITELN(total)
END.
```

Note that R and the constant 15.0 will be passed by value. At runtime, just before the transfer to the function RADD, the stack would be in the following form (each box contains one byte):



Stack grows downward

The following assembly-language subroutine implements the real add function, RADD. Notice that this example uses an in-line version of the entry code and exit code. The entry and exit sequences make sure that any virtual memory accessed in the frame of a routine is actually mapped to physical memory.

The entry code also verifies that the runtime stack is large enough for the local variables needed by the routine.

Note that the function return value is in the location specified by BP+6.

Note also that whenever the return value for a function is returned in a temporary variable created by the caller (rather than in registers), AX must be set to the two-byte offset address of this temporary variable by the function before it returns to the caller. In the following example, this is done by the mov ax, di instruction just before the standard exit sequence:

```
extrn ___chkstk:far        ;

_DATA   segment public 'DATA'
        ; if your routine needs static data, declare it here
_DATA   ends

DGROUP group _DATA

RADD_TEXT segment 'CODE'

                assume cs:RADD_TEXT,ds:DGROUP,ss:DGROUP

                public RADD
RADD            proc far

                ; Begin standard entry sequence
                inc bp              ; mark stack frame as a "far call" frame
                push bp             ; save old frame pointer
                mov bp,sp           ; set up new frame pointer
                mov a x,0           ; set AX to tot. # bytes for local vars
                call ___chkstk      ; reserve space on stack for local vars
                push di             ; save C register variables (si and di)
                push si
                ; End standard entry sequence

                fld dword ptr [bp+12]; push 1st param on 80287 stack

                fld dword ptr [bp+8]; push 2nd param on 80287 stack

                faddp st(1),st      ; add top two items on 80827 stack

                mov di,[bp+6]       ; di = addr of function return var
                fstp dword ptr [di]; store result in func. ret. var
                fwait
                mov ax, di          ; return address of result variable

                ; Begin standard exit sequence
                pop si              ; restore C register variables
                pop di
                mov sp,bp           ; recover space used by local variables
                pop bp              ; restore old frame pointer
                dec bp              ; restore low bit (used as frame marker)
                ; End standard exit sequence

                ret 10              ; return and recover space in parameters

RADD            endp

RADD_TEXT ends

end
```

Appendix B
Version Specifics

Microsoft Pascal has been implemented for a number of different microcomputer operating systems. This appendix describes the current implementation of the MS-Pascal language for MS-DOS. It discusses additions and restrictions to the language described in the current *Microsoft Pascal Reference Manual,* and identifies features of MS-Pascal that are not yet implemented.

For changes and additions to the MS-Pascal Compiler or language that may have been made after publication of this *User's Guide* and companion reference manual, see the README.DOC file, if present, provided on disk with the system files.

B.1 Implementation Additions

The following additions have been made to the language described in the May 1983 release of the *Microsoft Pascal Reference Manual.*

1. The following function can be declared EXTERN:

```
FUNCTION DOSXQQ
    (COMMAND, PARAMETER : WORD) : BYTE;
```

This function invokes the operating system, passing a command in the AH register and an additional parameter in the DX register. The BYTE function return value is identical to the value returned by the operating system in AL, the accumulator.

The PUBLIC variables CRCXQQ and CRDXQQ contain the values of the CX and DX registers after the call. The value of CRCXQQ is also loaded into CX before the call.

Several operating system functions are particularly useful:

a. DOSXQQ (1, 0);

   Returns the next character typed. If no character has been typed, DOSXQQ waits for input. The ASCII value of the typed character is returned, and the typed character is echoed on the terminal screen.

b. DOSXQQ (2, WRD ('x'));

   Outputs the character 'x' to your terminal. The function return value should be ignored. The CONTROL-S and CONTROL-Q commands to stop and start scrolling, and the CONTROL-P command to toggle the printer, are executed if entered. Tabs are expanded.

c. DOSXQQ (6, 255);

   Returns the next character typed on the keyboard, or zero, if no character has been typed. CONTROL-S, CONTROL-Q, and CONTROL-P are not treated specially. The character typed is not echoed on the terminal screen.

d. DOSXQQ (6, WRD ('x'));

   Outputs the character 'x' to your terminal. This is the same as DOSXQQ (2, WRD ('x')), above, except that CONTROL-S, CONTROL-Q, and CONTROL-P are not treated specially. The function return value should be ignored in this case.

e. DOSXQQ (11, 0);

   Returns screen status. The value 255 is returned if a character has been typed, a 0 is returned if a character has not been typed. This function is used to check for a keypress condition without actually reading the character.

f. DOSXQQ (13, 0);

   This function is not necessary in MS-DOS, but is provided for compatibility with other operating systems (CP/M® and CP/M-86®), where this function resets disk tables.

2. The following MS-Pascal filenames are available to indicate devices:

Name	Description	MS-DOS Code
USER	Console	1, 2, and 6
LINE	Auxiliary input	3, 4

Special MS-DOS filenames, like CON and NUL, are also available (see your MS-DOS manual for details). However, using CON for the terminal causes buffering of

input and output data and precludes interactive input and output. The filename USER should be used instead.

3. Program parameters are available. When a program starts, there is a prompt for every program parameter. You may also give program parameters on the command line with which you invoke the program. If a program requires more parameters than appear on the command line, the remaining parameters are prompted for.

   For example, assume that you want to execute the following program:

```
PROGRAM DEMO (INFILE, OUTFILE, P1, P2, P3);
VAR INFILE, OUTFILE : TEXT;
        P1, P2, P3 : INTEGER;
BEGIN
    .
    .
END.
```

   From the command line, you could run this program as follows:

```
A:DEMO DATA1.FIL DATA2.FIL 7 8 123
```

   If you give only the first parameter on the command line, the compiler will proceed to prompt you as follows (your responses are shown in italics):

```
A:DEMO DATA1.FIL
OUTFILE:DATA2.FIL 7
P2:8
P3:123
```

   An LSTRING parameter value of NULL cannot be read from the command line and is assumed to be missing. You can enter it by pressing the RETURN key in response to the prompt.

4. The PUBLIC variable CESXQQ, containing the segment register value for the start of the MS-DOS data area, is available. This allows you to reference the command line, as shown:

```
VAR MSDATA : ADS OF LSTRING (80);
        CESXQQ [EXTERN] : WORD;
BEGIN
    MSDATA.S := CESXQQ; MSDATA.R := 128
    {MSDATA^ now contains the command line.}
END;
```

The MS-DOS data area also contains, at offset 2, the upper memory limit, expressed as the segment (i.e., paragraph) address of the first byte after available memory. The lower memory segment address is simply 4K paragraphs (i.e., 64K bytes) above the default data segment. For example:

```
VAR LOMADS, HIMADS, MSDATA : ADS OF WORD;
    CESXQQ [EXTERN] : WORD;
BEGIN
    LOMADS := ADS LOMADS;
    LOMADS.S := LOMADS.S + 4096;
    LOMADS.R := 0;
    {LOMADS is first available address.}

    MSDATA.S := CESXQQ; MSDATA.R := 2;
    HIMADS.S := MSDATA^; HIMADS.R := 0
    {HIMADS is first unavailable address.}
END;
```

5. TIME, TICS, and DATE are supported for MS-DOS systems with clocks. TICS returns hundredths of seconds.

6. Real number conversion utilities.

Releases of MS-Pascal starting with 3.0 use the IEEE real number format. Releases of MS-Pascal 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 may do so with the following library routines:

a. Single Precision Reals

Microsoft to IEEE format

```
PROCEDURE M2ISQQ (vars rms, rieee : real4)
```

IEEE to Microsoft format

```
PROCEDURE I2MSQQ (vars rieee, rms : real4)
```

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

b. Double Precision Reals

Microsoft to IEEE format

```
PROCEDURE M2IDQQ (vars dms, dieee : real8)
```

IEEE to Microsoft format

```
PROCEDURE I2MDQQ (vars dieee, dms : real8)
```

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

7. Bankers' rounding is used when truncating real numbers that end with .5; that is, odd numbers are rounded up to an even integer, even numbers are rounded down to an even integer. For example:

```
TRUNC (4.5) = 4
TRUNC (207.5) = 208
```

B.2 Implementation Restrictions

The following restrictions apply to this implementation of MS-Pascal:

1. Identifiers can have up to 31 characters. Longer identifiers are truncated.
2. Numeric constants can have up to 31 characters. Like identifiers, numeric constants longer than 31 characters are truncated.
3. The PORT attribute for variables is identical to the ORIGIN attribute. It does not use I/O port addresses.
4. The maximum level to which procedures can be statically nested is 15. Dynamic nesting of procedures is limited by the size of the stack.
5. The FORTRAN attribute does nothing. MS-Pascal and MS-FORTRAN share the same code generator and calling sequence. MS-FORTRAN parameters are always passed as MS-Pascal VARS parameters.
6. $simple currently turns off common subexpression optimization. $size and $speed turn it back on (and have no other effect).

B.3 Unimplemented Features

The following MS-Pascal features are not presently implemented, or are implemented only as discussed below:

1. OTHERWISE is not accepted in RECORD declarations.
2. Code is generated for PURE functions, but no checking is done.
3. The extend level operators SHL, SHR, and ISR are not available.
4. No checking is done for invalid GOTOs and uninitialized REAL values.
5. READ, READLN, and DECODE cannot have M and N parameters.
6. Enumerated I/O, permitting the reading and writing of enumerated constants as strings, is not available.
7. The metacommands $tagck, $standard, $extend, and $system can be given, but have no effect.
8. The $inconst metacommand does not accept string constants.

Appendix C
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 8087.LIB and MATH.LIB for a particular hardware configuration.

8087.LIB, 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 runtime library to handle 8087 interrupts. Each of the elements of the structure is described briefly below:

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 is 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:8087.lib
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 8087.LIB.
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 D
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.

The following list contains the five exceptions and their default and alternate actions:

1. Invalid Operation—Any operation with a NAN, square 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

D.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 sections.

D.1.1 The 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.

D.1.2 The 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

D.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 and functions:

The procedure LCWRQQ loads the 8087 CONTROL word.

```
PROCEDURE LCWRQQ (consts w : word);
```

The function SCWRQQ stores the 8087 CONTROL word.

```
FUNCTION SCWRQQ : word;
```

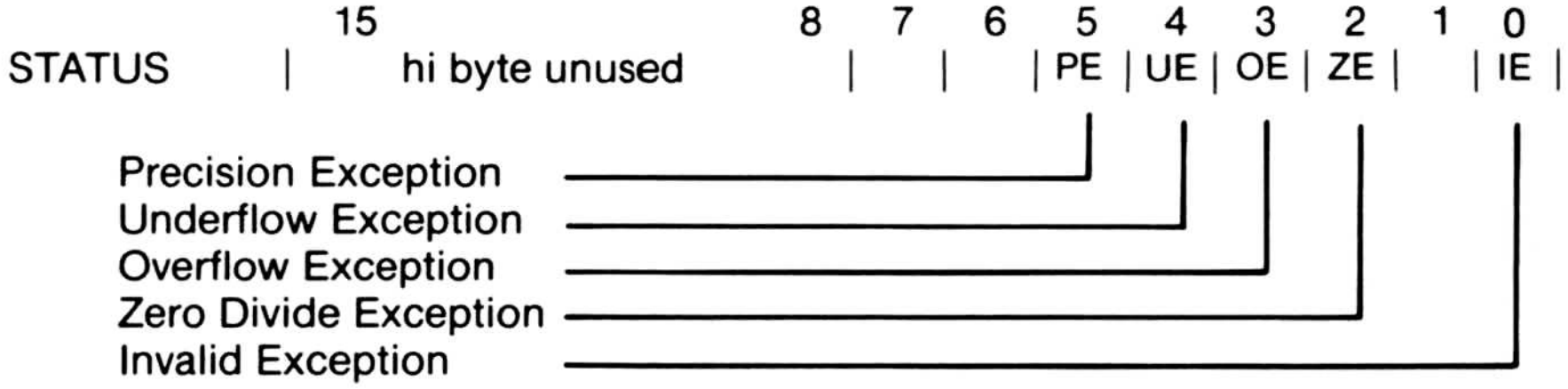
The function SSWRQQ stores the 8087 STATUS word.

```
FUNCTION SSWRQQ : word;
```

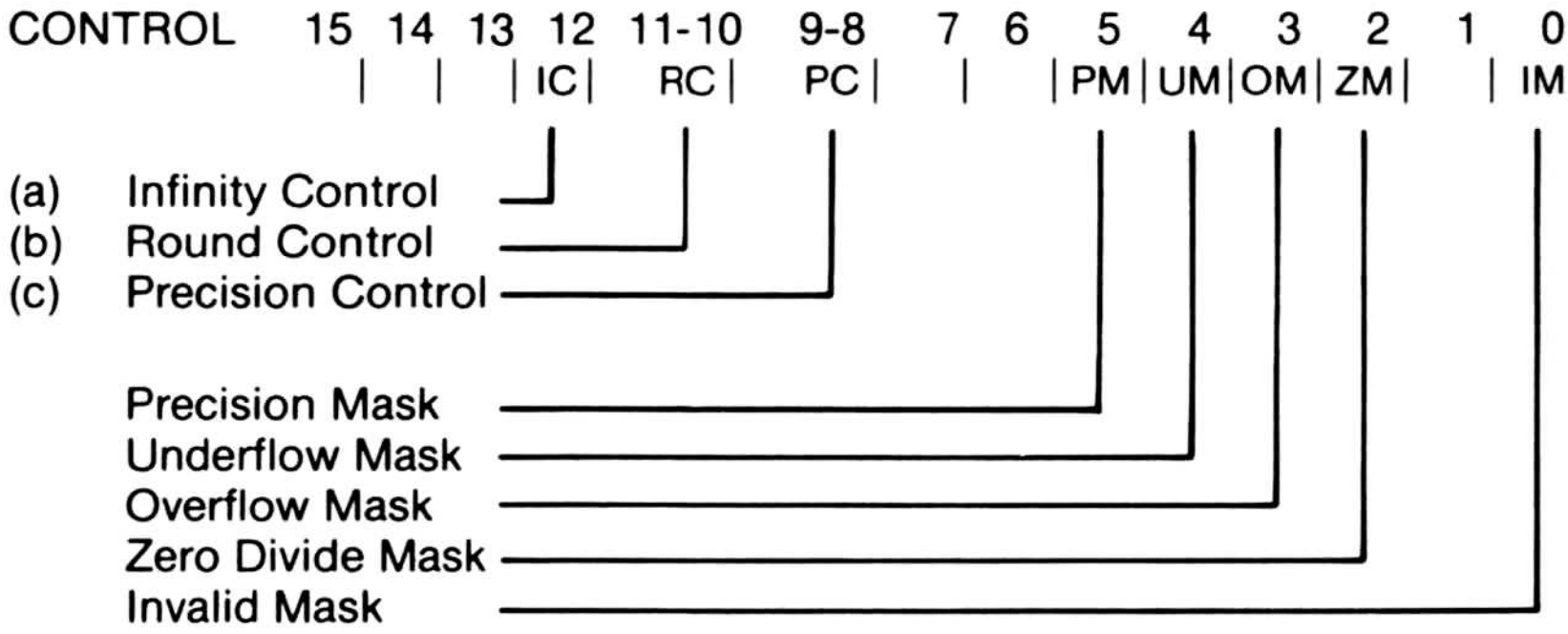
If you disable the exceptions listed at the beginning of this appendix, you will either get NAN, Infinite, or Indefinite values in your variables. If you print such a value, the output field will contain NAN, INF, or IND padded with periods to the field width. If the output field has less than three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

D.3 Formats for 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 E
Mixed-Language Programming

Microsoft FORTRAN and Pascal (versions 3.3 or later) and Microsoft C (version 3.0 or later) provide support for programmers who use more than one of these languages. The information in this appendix is not required for most programs.

Note

Microsoft C for XENIX does not include the fortran and pascal keywords, which are described later in this document. You cannot call FORTRAN and Pascal from the XENIX version of C unless you use the C calling conventions.

(*FORTRAN and C programmers, please note:* Throughout this section, the terms *procedure* and *parameter* are used instead of *subroutine* or *function* and *argument*. This is the terminology used in Pascal.)

Mixed-language programming offers several advantages:

1. You can use libraries of procedures written in different languages.

   For example, you can access the Microsoft C library from programs written in FORTRAN or Pascal. There are also many proprietary libraries available for use with Microsoft FORTRAN, which you can access from Microsoft Pascal and C.

   To use a library written for a particular language, you must have the library supplied with that language's compiler. To use a proprietary FORTRAN library from C, for example, you need the library supplied with the FORTRAN compiler, as well as the proprietary library itself. This is because programs written in Microsoft Pascal, C, or FORTRAN contain calls to their respective runtime libraries.

2. You can use features not available in your language.

   It is hard to write bit manipulation procedures in FORTRAN, for example, but it is easy in C or Pascal. Also, some interfaces, such as those that use C or Pascal structures, are not compatible with FORTRAN.

3. If you write your own libraries, you can now produce one library that is compatible with all three languages.

   Of course, to ensure compatibility, you must pay close attention to the guidelines given in this section. Remember that users also need the runtime library provided with the language in which your library is written, as mentioned in point one.

E.1 Memory Models

If you use C procedures in mixed-language programming

- You must compile your C code with the large model switch.

  The current versions of Pascal and FORTRAN do not offer a choice of memory models; they are only compatible with large model C.

- You must use the large model C library.

  Some components of the C library are referenced from the other languages' libraries. If you use the library for the wrong memory model these interfaces will be incorrect.

E.2 Choosing a Calling Convention

FORTRAN, Pascal, and C each have conventions for passing parameters.

The languages differ first in the order in which parameters are pushed on the stack. Microsoft Pascal and FORTRAN push parameters on the stack in the order in which they appear in the procedure declaration. C pushes its parameters in the reverse order.

The languages also differ in whether code telling how to restore the stack when a procedure returns is in the calling procedure or in the called procedure. In the FORTRAN/Pascal convention, this code is in the called procedure; in the C language, this code follows the procedure call.

The FORTRAN/Pascal convention is slightly faster and produces less code. The C convention allows you to use a varying number of parameters (because the first parameter is always the last one pushed, it is always on top of the stack, and always has the same address relative to the start of the frame). These conventions are incompatible.

Finally, the languages differ in what parameters they pass by reference and by value. Section E.2.1, "Passing Parameters by Reference or Value," discusses these differences.

If you control both the calling and the called code, you can choose which calling convention to use. If you intend to pass varying numbers of parameters, you must use the C calling convention. For more information, see Section E.2.2, "Using Varying Numbers of Parameters." Otherwise, you may want to use the convention of the language that you use most often, so that you can usually use the default calling convention.

To make calls from one language to another, you must tell the compiler which convention to use. Microsoft C, Pascal, and FORTRAN all provide ways of specifying which convention you are using, both when you call an external procedure and when you define a public procedure. Table E.1 indicates how to specify calling conventions from each language.

Table E.1
Specifying Calling Conventions

When Calling From This Language:	Use These Attributes/Keywords:
To Use C Calling Conventions:	
Pascal	C attribute on procedure declaration
FORTRAN	C attribute on INTERFACE statement
C	*default*
To Use FORTRAN Calling Conventions:	
Pascal	FORTRAN attribute on procedure declaration
FORTRAN	*default*
C	fortran keyword on procedure declaration
To Use Pascal Calling Conventions:	
Pascal	*default*
FORTRAN	PASCAL attribute on INTERFACE statement
C	pascal keyword on procedure declaration

E.2.1 Passing Parameters by Reference or Value

When a parameter is passed by reference, the address of the parameter is passed. Procedures access the parameter's value through the address; any changes to the parameter affect the stored value. When a parameter is passed by value, a copy of the parameter is placed on the stack when the procedure is called. The procedure can change the value of the parameter without affecting the original value from which the copy was taken.

For each parameter, you must decide whether to pass by value or by reference. If you pass by reference, you also have to choose whether to pass a long address (segment and offset) or a short address (offset only).

If the called procedure needs to change the actual value in the variable as a way of returning a result, you have to pass by reference. Passing by value protects against accidental updating and, for variables smaller than about four bytes, can be more efficient.

The following list describes the defaults for each language:

- FORTRAN passes all parameters by reference (including constants and expressions), but passing by value can be specified. If a procedure is given the C or Pascal attribute, the default is changed: all parameters for that procedure are passed by value unless otherwise specified.
- C always passes arrays by reference, and passes all other parameters by value. In C, you can pass pointers as parameters; the procedure can use the pointers to modify stored values, producing the same effect as passing by reference.
- Pascal passes by value, but passing by reference can be specified.

If you do not choose the default case, you have to specify certain keywords, attributes, or pointer types. These will vary, according to the calling conventions you are using. See Tables E.2 through E.4.

If you are passing parameters when using C calling conventions, use the constructs described in Table E.2 when declaring parameters.

Table E.2

Passing Parameters With C Calling Conventions

Parameter C		Pascal	FORTRAN
long address	pointer to *type*	VARS keyword	REFERENCE attribute
short address	near pointer to *type*	VAR keyword	REFERENCE,NEAR attributes
value	default	default	default

For example, assume that you are using the C calling conventions. Table E.1 shows what attributes and keywords are necessary to use the C calling conventions. When calling from Pascal, specify the C attribute on the procedure declaration. When calling from FORTRAN, specify the C attribute on the INTERFACE statement. When calling from C, the C calling conventions are the default.

Now, assume that you want to pass an integer parameter, x, using a long address. Compatibility of data types is discussed in Section E.10; for now, assume that the C int type, the Pascal integer type, and the FORTRAN INTEGER type are equivalent. Table E.2 shows that when declaring the parameter x in your C procedure, you should use a pointer (a far pointer, the default) of the appropriate type (in this case, int). The C declaration is

```
int *x;
```

When declaring the parameter x in your Pascal procedure, use the VARS keyword

```
VARS x:INTEGER;
```

For the FORTRAN procedure, specify the reference attribute

```
INTEGER X[REFERENCE]
```

If you want to pass using a short address instead, the appropriate declarations are

```
int near *x;
```

```
VAR x:INTEGER;
```

```
INTEGER X[REFERENCE,NEAR]
```

You follow the same steps when declaring parameters even if you are using other calling conventions. If you are passing parameters using Pascal or FORTRAN calling conventions, use the constructs described in Tables E.3 and E.4 when declaring parameters.

Table E.3

Passing Parameters With Pascal Calling Conventions

Parameter	C	Pascal	FORTRAN
long address	pointer to *type*	VARS keyword	REFERENCE attribute
short address	near pointer to *type*	VAR keyword	REFERENCE, NEAR attributes
value	default	default	default

Table E.4

Passing Parameters With FORTRAN Calling Conventions

Parameter	C	Pascal	FORTRAN
long address	pointer to *type*	VARS keyword	default
short address	near pointer to *type*	VAR keyword	NEAR attribute
value	default	default	VALUE attribute

If you are not writing both the called procedure *and* the calling procedure, you must pass the parameter as declared in the existing procedure's definition. If you are not experienced with the language you are accessing, it is not always easy to determine if a parameter is being passed by value or by reference. The following lists indicate how to tell the difference.

The following kinds of parameters are passed by value:

- In Pascal, any parameter declared, except VAR, CONST, VARS, and CONSTS parameters
- In C, any parameter declared except arrays
- In FORTRAN, a parameter that is declared with the VALUE attribute
- In FORTRAN, a parameter in a procedure when that procedure is declared with the C or Pascal attribute (unless the REFERENCE attribute is specified)

The following kinds of parameters are passed by reference with a short (two-byte, offset only) address:

- In Pascal, a formal parameter declared as VAR or CONST.
- In Pascal, a variable that is passed by passing a pointer to that variable. The pointer itself is passed by value. (It is not recommended that you use pointers in this way; the correspondence between pointers and machine addresses is implementation dependent.)
- In Pascal, a variable that is passed by passing ADR *variable*. The address itself (as with pointers) is passed by value.
- In C, a parameter that is passed by passing a near pointer to the parameter. (The pointer is passed by value.)
- In C, an array declared with the keyword near.
- In FORTRAN, in procedures without the C or PASCAL attributes, a parameter with the NEAR attribute.
- In FORTRAN, in procedures with the C or PASCAL attributes, a parameter with the NEAR and REFERENCE attributes.
- In FORTRAN, a variable that is passed by short address by taking LOCNEAR(*variable*), then passing the result as an INTEGER*2, by value.

The following kinds of parameters are passed by reference with a long (4-byte, segmented) address:

- In Pascal, ADS *variable*. (The address is passed by value.)
- In Pascal, parameters declared with the VARS or CONSTS keywords.
- In C, a parameter passed by passing a far pointer to the parameter. (The pointer is passed by value.) Note that in large-model C, far pointers are the default pointer type.
- In C, arrays not declared with the keyword near.
- In FORTRAN, any parameter of a FORTRAN-protocol routine except those declared with the NEAR or VALUE attributes.
- In FORTRAN, a variable passed by long address by taking LOC(*variable*) or LOCFAR(*variable*), then passing the result as an INTEGER*4, by value.

E.2.2 Using Varying Numbers of Parameters

If you are going to use varying numbers of parameters:

- The number of actual parameters must be less than or equal to the number of formal parameters (if the called procedure is written in FORTRAN or Pascal).

  There is no easy way in Pascal and FORTRAN to access parameters that have not been formally defined. However, you may use the VARYING attribute to pass fewer arguments than are defined.

- You must use the C and VARYING attributes on your FORTRAN INTERFACE statement or Pascal procedure declaration.

  The VARYING attribute tells the FORTRAN or Pascal compiler not to check if there are more or fewer actual parameters than formal parameters. However, actual parameters for which a formal parameter is specified *will* be checked for type compatibility according to the usual rules of the calling procedure's language.

E.3 Naming Conventions

If you follow these two rules, the Microsoft Pascal, FORTRAN, and C compilers handle all the necessary adjustments in names:

1. If you are using any FORTRAN routines, all identifiers (names) should be six characters or less in length.
2. Avoid using uppercase characters in C identifiers. If you must use uppercase characters, specify IGNORECASE when you link, and don't use any other identifiers that have the same spelling as the uppercase or mixed-case C identifier. (For example, if one C identifier is AnExample, don't use anexample, ANEXAMPLE, or AnExAmPlE as identifiers.)

If you cannot follow those two rules, you must make certain adjustments yourself. The remainder of this section explains the default naming conventions of each language, and how certain

attributes and keywords affect those naming conventions. This information should allow you to solve any special problems in naming.

In all three languages, names appear differently in the object and source files. There are three differences in the naming conventions used by the three languages:

Case	In FORTRAN and Pascal, any lowercase letters in a public identifier are changed to uppercase before the name is inserted in the object file. By default, no such transformation is done on C names, but at link time you can specify that case distinctions are to be ignored.
Length	In FORTRAN, by default, names are truncated to six significant characters.
Underscores	In C, public names are always prefixed with an underscore character (_) before they are inserted in the object file.

These differences in naming conventions mean that default FORTRAN and Pascal public names will not correspond to default C public names. Certain attributes and keywords can help you make names correspond.

If you specify the C attribute on

- the name of a public or external procedure or data object in Pascal
- the name of a procedure, interface or named common block in FORTRAN

the name is changed to lowercase and a leading underscore is added. FORTRAN identifiers will still be truncated to six characters. To specify a longer name, or to specify external C routines that have uppercase letters in their identifiers, you can use the ALIAS in FORTRAN. There is no ALIAS feature in Pascal; to refer to a C object with uppercase letters in its identifier, you *must* link with the IGNORECASE option, and all your C identifiers must have unique spellings.

If you use the pascal or fortran keywords in C, the name is changed to uppercase and the leading underscore is deleted from the name. All such names must have unique spellings.

Note that in FORTRAN, if an INTERFACE and the subprogram referred to in that INTERFACE are in the same unit of compilation, the same names must be used for parameters in each. Error 87 is generated if you violate this rule.

E.4 Writing Interfaces to Pascal or C from FORTRAN

To declare external procedures in C or Pascal from FORTRAN, FORTRAN provides the INTERFACE statement.

Suppose, for example, that you want to access the procedure *time* in the C library. There are three basic steps to follow:

1. Find the declaration of the C procedure
2. Build an INTERFACE program unit
   - Determine the attributes and type for the procedure
   - Determine the attributes and types for the parameters
3. Add the INTERFACE to the program

The final step, calling the C procedure, is described in Section E.5.

For this example, the declaration of the C procedure *time* looks like this:

```
long time (tloc)
long *tloc;
```

The first step in building the INTERFACE is to determine what attributes and type to use for the procedure. First, determine what FORTRAN type is equivalent to the type of the procedure *time*. The first word in the C procedure declaration, *long* time (tloc);

shows that *time* has type long. Referring to the "Signed Four-Byte Integers" section of Table E.5, you can see that the FORTRAN INTEGER*4 type is equivalent to the C long type. This gives enough information to write

```
INTERFACE TO INTEGER*4 FUNCTION TIME
```

Second, decide which calling convention to use. Since you have no control over the C procedure, you must use the calling conventions that it uses. To specify the C calling conventions, use the C attribute as follows:

```
INTERFACE TO INTEGER*4 FUNCTION TIME[C]
```

Now, determine what attributes and data types to use for the parameters. In this case, there is just one parameter, tloc. You can write

```
 INTERFACE TO INTEGER*4 FUNCTION TIME[C]
+(TLOC)
```

However, note that in the second line of the C procedure declaration, tloc is preceded by an asterisk, indicating that a pointer is being passed. You can pass a pointer from FORTRAN using the LOCFAR or LOC procedures, or you can pass the argument itself by reference. For now, assume that you want to pass by reference. FORTRAN normally defaults to passing by reference, but the procedure TIME is qualified by the C attribute, so TLOC will default to being passed by value. To specify passing by reference, add the REFERENCE attribute

```
 INTERFACE TO INTEGER*4 FUNCTION TIME[C]
+(TLOC[REFERENCE])
```

The type of the parameter tloc is indicated by the first word in the second line of the C procedure declaration, *long* * tloc. Since the FORTRAN INTEGER*4 type is equivalent to the C long type, you can finish the INTERFACE unit as follows:

```
 INTERFACE TO INTEGER*4 FUNCTION TIME[C]
+(TLOC[REFERENCE])
 INTEGER*4 TLOC
 END
```

If you decide to pass a pointer to TLOC, instead of passing it by reference, you proceed, in the same manner, to this point

```
      INTERFACE TO INTEGER*4 FUNCTION TIME[C]
     +(TLOC)
```

Pointers are passed by value so do not specify the REFERENCE attribute. Since pointers are normally 4-byte segmented addresses, the result of LOC is a 4-byte integer, and therefore you must declare the parameter TLOC to be a 4-byte integer.

```
      INTERFACE TO INTEGER*4 FUNCTION TIME[C]
     +(TLOC)
      INTEGER*4 TLOC
      END
```

Step number three, adding the INTERFACE unit to your program, is identical for both cases. The only rule to follow is that the INTERFACE must occur before any references to the procedure are made. It is usually easiest to put all INTERFACES at the beginning of the compiland.

The final step, calling the procedure, is different for the REFERENCE and pointer cases, as described in the next section.

E.5 Calling Procedures in Pascal or C from FORTRAN

Once you have declared a procedure, you can call it in your program just as if it were in the same language as your calling procedure. Note that when calling from FORTRAN, you must always declare the procedure in the program units which use it.

For the example discussed in Section E.4, start writing the calling routine like this

```
SUBROUTINE CLOCK
INTEGER*4 TIME
INTEGER*4 TLOC
```

Don't forget to declare the procedure, as in the line INTEGER*4 TIME above.

Now, if you passed TLOC by reference, you can complete the call as follows:

```
SUBROUTINE CLOCK
INTEGER*4 TIME
INTEGER*4 TLOC
WRITE (*,*) TIME (TLOC)
END
```

If you passed a pointer, your procedure call looks like this:

```
SUBROUTINE CLOCK
INTEGER*4 TIME
INTEGER*4 TLOC
WRITE(*,*) TIME(LOC(TLOC))
END
```

You could substitute the LOCFAR procedure for the LOC procedure. In this implementation they are identical.

Note that if *time* were a subroutine instead of a function, you could call that subroutine with the FORTRAN CALL statement.

E.6 Writing Interfaces to FORTRAN or C from Pascal

From Pascal, attach the fortran or c attribute to an EXTERN procedure declaration to interface with procedures written in FORTRAN or C.

E.7 Calling Procedures in FORTRAN or C from Pascal

Once you have declared a procedure, you can call it in your program just as if it were in the same language as your main program.

For example, the following Pascal program fragment calls time, passing tloc by reference:

```
FUNCTION time (VARS tloc:INTEGER4):INTEGER4[C];EXTERN;
PROCEDURE clock;
    VAR tloc: INTEGER4;
    BEGIN
        WRITELN (time(tloc))
    END;
```

If you pass a pointer by value, the program fragment looks like this:

```
FUNCTION time (tloc:ADSMEM):INTEGER4 [C]; EXTERN;
PROCEDURE clock;
      VAR tloc:INTEGER4;
      BEGIN
          WRITELN(time(ADS tloc))
      END;
```

E.8 Writing Interfaces to FORTRAN or Pascal from C

From C, use the fortran and pascal keywords to declare procedures written in or compatible with FORTRAN and Pascal. These keywords imply changes in external naming, calling conventions, and return variable conventions. You may need to specify a particular option when compiling to enable these keywords; refer to your *Microsoft C User's Guide* for more information.

You declare FORTRAN and Pascal procedures in the same manner as C procedures: you specify the procedure identifier, the return type, and the type and number of parameters to the procedure. (See the *Microsoft C Language Reference* for a complete discussion of the syntax of procedure declarations.)

The following additional rules apply when you use the fortran and pascal keywords:

1. Whenever a fortran or pascal keyword is used in a declaration, the types of parameters must be declared with a parameter type list.
2. The fortran and pascal keywords modify the item immediately to the right in a declaration.
3. The special near and far keywords can be used with the fortran and pascal keywords in declarations. The sequences far fortran and fortran far are equivalent.

Complex declarators are allowed in pascal and fortran declarations, just as in C procedure declarations. The following examples illustrate the syntax of pascal and fortran declarations. For more on declarators, see the *Microsoft C Language Reference*.

Examples

```
1. short pascal thing(short, short);

2. long (pascal *thing)(void);

3. short near pascal thing(short);

4. short pascal near thing(short);
```

Example 1 declares *thing* to be a Pascal procedure taking two short parameters and returning a short value.

In Example 2, *thing* is declared as a pointer to a Pascal procedure that takes no parameters and returns a long value. Note that void is used to indicate that there is no return value.

Examples 3 and 4 are equivalent. Both declare *thing* to be a near Pascal procedure. The procedure takes one short parameter and returns a short value.

E.9 Calling Procedures in FORTRAN or Pascal from C

To call a Pascal or FORTRAN procedure from C, you must declare that procedure external. For example

```
extern void fortran m(long);
```

Note that void is used to indicate that there are no parameters.

Once you have declared a procedure, you can call it in your program just as if the procedure were in C.

E.10 Data Types

FORTRAN, Pascal, and C each have a variety of data types. Some are completely compatible, others require manipulation to work between languages.

E.10.1 Integers

In C, any integral parameters shorter than an int (such as char) are converted to int type before being passed by value. Unsigned integral types shorter than an unsigned int (such as unsigned char) are converted to unsigned int type.

To ensure that your FORTRAN or Pascal routine handles C parameters correctly, you have two options:

1. You can allow for the C conversions when you declare parameters to the FORTRAN or Pascal procedure. This means, for example, that all integer parameters must be declared to have the size corresponding to a C int, or long int, for integer parameters larger than an int.

2. You can pass pointers to the parameters instead of the values themselves (passing by reference). In the FORTRAN or Pascal routine, declare the passed parameters as a pointer to or reference parameter of the appropriate type, then use the pointer to access the value indirectly.

Also, note that the C int type is machine specific. For the 8086 family of microprocessors, the C int type is equivalent to the following types:

- INTEGER2 in Pascal
- INTEGER*2 in FORTRAN
- INTEGERC in Pascal
- INTEGER[C] in FORTRAN

For any given processor and operating system, variables defined with the last two types are equivalent to variables of the C int type as defined by the Microsoft C Compiler for the same system. The last two types are therefore more portable than the first two.

Table E.5 shows integer data types and their equivalents in Pascal, C, and FORTRAN.

Using the *Equivalent Data Types* Tables

To use Tables E.5 through E.14 to pass parameters, you also have to refer back to Tables E.2 through E.4.

For example, suppose that you want to pass an INTEGER*2 variable from FORTRAN to C. First, you have to choose a calling convention, as explained in Section E.2, "Choosing a Calling Convention." Assume that you want to use the C calling conventions. Refer to Table E.2, "Passing Parameters With C Calling Conventions."

Second, decide whether to pass the parameter by reference or by value. Assume that you want to pass the parameter by reference, using a short address. Table E.2 shows that you use the REFERENCE and NEAR attributes in FORTRAN, and a near pointer of the appropriate type in C.

Third, determine what data type in C is equivalent to the INTEGER*2 type in FORTRAN. Find the "Equivalent Data Types" table that lists integers: Table E.5. Look for the part of the table that lists signed, two-byte integers. Note that INTEGER*2 is listed as an appropriate FORTRAN data type. Check the "Notes" column to see if there is anything to watch out for when using INTEGER*2.

Now, look at the "C" row. You can choose between short and int, but the Notes column shows that int is machine-dependent. For maximum portability, choose the C short type. Finally, applying the appropriate attributes and keywords to the data types

```
INTEGER*2 X [REFERENCE,NEAR]
```

in a FORTRAN INTERFACE declared with the C attribute is equivalent to a C parameter declared with

```
short near * x
```

Note that using a REFERENCE parameter in FORTRAN corresponds to using a pointer type in C.

Table E.5
Equivalent Data Types: Integers

Signed One-Byte Integers

Language	Data Type	Notes
Pascal	x:sint	
	x:*a*..*b*	for *a* >=-127 and *b* <=127
C	char x	when passed by reference
	struct { char x;} x	when passed by value
FORTRAN	*none*	

Unsigned One-Byte Integers

Language	Data Type	Notes
Pascal	x:byte	
	x:wrd(*a*)..wrd(*b*)	for 0 <= *a* <= b for *b* <= 255
	x:(*a*,*b*,..*n*)	for ord (*n*) <256
C	unsigned char x	when passed by reference
	struct { unsigned char x;} x	when passed by value
FORTRAN	CHARACTER*1 X	FORTRAN has no unsigned types so you must use CHARACTER*1, and use the ICHAR and CHAR functions to transfer values. Do not pass negative values.

Table E.5 *(continued)*

Signed Two-Byte Integers

Language	Data Type	Notes
Pascal	x:integer2	
	x:integerc	
	x:integer	if $INTEGER:2 (the default) is in effect
C	short x	
	int x	machine-dependent
FORTRAN	INTEGER*2 X	
	INTEGER[C] X	
	INTEGER X	if $STORAGE:2 is in effect

Unsigned Two-Byte Integers

Language	Data Type	Notes
Pascal	x:word	
	x:wrd(*a*)..wrd(*b*)	for *b* >255
	x:(*a*,*b*,..*n*)	for ord (*n*) >255
C	unsigned short x	
	unsigned int x	machine-dependent
FORTRAN	INTEGER*2 X	FORTRAN has no unsigned types so you must use INTEGER*2. Do not pass negative values or values greater than 32767. Note that many unsigned operations can be safely performed on INTEGER*2 values.

Table E.5 *(continued)*

Signed Four-Byte Integers

Language	Data Type	Notes
Pascal	x:integer4	
	x:integer	if $INTEGER:4 is in effect
C	long x	
FORTRAN	INTEGER*4 X	
	INTEGER X	if $STORAGE:4 (the default) is in effect

C also has unsigned 4-byte integers. FORTRAN and Pascal do not. However, many unsigned arithmetic operations can be performed on signed variables, and will yield correct results. This level of type equivalence may be sufficient for some applications.

E.10.2 Boolean and Character

For Pascal Boolean values, the integer one (1) means true. Zero (0) means false.

Table E.6 shows how Boolean and character types are represented in Pascal, C, and FORTRAN.

Table E.6

Equivalent Data Types: Boolean and Character

Boolean

Language	Data Type	Notes
Pascal	x:boolean	
C	unsigned char x	
FORTRAN	CHARACTER*1 X	Use as for unsigned one-byte integers; 1 = false and 0 = true. FORTRAN LOGICAL types are *not* equivalent. See Table E.14 for FORTRAN LOGICAL types.

Character

Language	Data Type	Notes
Pascal	x:char	
C	unsigned char x	
FORTRAN	CHARACTER X	

E.10.3 Real Numbers

C passes all real parameters by value and as double precision values. To ensure that your FORTRAN or Pascal routine handles C parameters correctly, you have three options:

1. You can allow for the C conversions when you declare parameters to the FORTRAN or Pascal procedure. This means that you must declare all floating-point parameters as double-precision parameters (REAL*8 in FORTRAN, real8 in Pascal), and specify the VALUE attribute in FORTRAN.

2. You can pass pointers to the parameters instead of the values themselves. In the FORTRAN or Pascal routine, declare the passed parameters as a pointer to the appropriate type, then dereference the pointer to access the value.

3. To avoid expansion of a float value to a double, you can pass the value as a structure. The members of structures do not undergo type conversion when the structure is passed as a parameter. For example, the declaration

```
struct fptype {float a;} arg;
```

   defines a structure variable, *arg*, with a single float member. The structure variable *arg* can then be passed as a parameter. Passing such a struct as a parameter in C is equivalent to pushing a REAL*4 in FORTRAN (except that FORTRAN normally passes by reference) or a real4 value in Pascal.

Floating-point values returned to C from Pascal or FORTRAN are handled as structured values.

Table E.7 shows equivalent real types in Pascal, C, and FORTRAN.

Table E.7
Equivalent Data Types: Reals

Single Precision Real Numbers

Language	Data Type	Notes
Pascal	x:real4	
	x:real	if $real:4 (the default) is in effect
C	float x	
	struct { float x;} x	when passed by value
FORTRAN	REAL X	
	REAL*4 X	

Double Precision Real Numbers

Language	Data Type	Notes
Pascal	x:real8	
	x:real	if $real:8 is in effect
C	double x	
FORTRAN	REAL*8 X or DOUBLE PRECI- SION X	

E.10.4 Passing Strings

Pascal, FORTRAN and C each store character strings in memory in a different way. In order to pass strings from one language to another, you must give the computer the appropriate information about how the string is set up.

C strings are considered arrays of characters. The null (zero-value) character marks the end of the string and is the last character of the array. For example, the string

String of text.

is indicated in C as

```
unsigned char str[ ]="String of text."
```

This is stored in memory as a 16-byte array: 15 bytes of significant text (i.e., the string itself) and 1 null character that marks the end of the string:

S	t	r	i	n	g		o	f		t	e	x	t	.	\0

FORTRAN strings do not have delimiters in memory. The length of the string is determined in advance. The above string is written in FORTRAN as

```
STR='String of text.'
```

It is stored in memory as 15 bytes of text:

S	t	r	i	n	g		o	f		t	e	x	t	.

Pascal has two forms of string: a fixed-length string type, STRING, which is the same as the FORTRAN string type, and a variable-length string type, LSTRING. Using LSTRING, the above string is designated as follows:

```
VAR STR: LSTRING(15);
STR := 'String of text.';
```

It is stored in memory as 16 bytes. The first byte indicates the number of bytes allocated in memory for the string, the remaining 15 are the string itself:

15	S	t	r	i	n	g		o	f		t	e	x	t	.

Table E.8 summarizes how each language handles string and array types. Where *a* is a constant, each type occupies *a* bytes.

Table E.8
String and Array Types

Language	Type
Pascal	c:STRING(*a*)
	c:ARRAY[1..*a*] OF CHAR;
	c:LSTRING(*a*–1);
FORTRAN	CHARACTER*a C
	CHARACTER*1 C(*a*)
C	unsigned char c[*a*]
	struct cstr {unsigned char c[*a*];}c

Table E.9 shows equivalent string types in each language.

Table E.9
Equivalent Data Types: Strings

Language	Data Type	Notes
Pascal	x:array[1..*n*] of char	
C	char x[*n*];	
FORTRAN	CHARACTER*n x	Not equivalent in future releases of FORTRAN. Not recommended.
	INTEGER x ((*n* + 1)/2)	Can be equivalenced to a CHARACTER variable to allow access to individual bytes. This option will be equivalent in future releases, as well as in the present release.

The following sections explain how to pass strings from one language to another.

E.10.4.1 Passing FORTRAN Strings to C or Pascal

FORTRAN strings have the same format in memory as Pascal STRINGs, so you may pass them directly.

To pass FORTRAN strings to C, use the new C string feature. When a standard FORTRAN string constant is followed by the character C, that string is then interpreted as a C string constant. A null character is automatically appended to the end of the string, and backslashes (\ ) are treated as escapes. See the *Microsoft FORTRAN Compiler User's Guide* for information on the new C string feature.

Note

In subsequent releases of Microsoft FORTRAN, strings will be passed differently. In Version 3.3 and all earlier versions, the length of a string is not passed with the string. In later versions, the length of a string will be passed with the string as a super array type. These two methods are incompatible.

If you are calling FORTRAN from C or Pascal and you are using strings, your calling code may have to be changed for a later version of Microsoft FORTRAN.

E.10.4.2 Passing Pascal Strings to C and FORTRAN

Since Pascal STRINGs and FORTRAN strings have the same format in memory, you may pass them directly.

To pass Pascal STRING types to C, use concatenation to add an extra null byte to the end of the string. For example, if "strg" is a variable of type STRING, the null byte can be added as follows:

```
strg:"String of text."+CHR(0) ;
```

Then "strg" can be passed to any C function that expects a string argument.

To pass LSTRINGs to C and FORTRAN, you must convert them to STRINGs and handle the length byte yourself.

E.10.4.3 Passing C Strings to Pascal and FORTRAN

To FORTRAN and Pascal, C strings are just arrays. When passing C strings to Pascal and FORTRAN, allow room for the null byte at the end of the string.

E.10.5 Pointers

Table E.10 shows equivalent pointer types for each language.

When using procedure pointers and calling a FORTRAN or Pascal routine from C with the C calling convention, use this syntax to declare the procedure pointers in the argument declaration section of your C procedure:

returntype (* x)*(types-list)*

The *returntype* is the C type of the return value. The *types-list* is given with the same syntax used to declare the argument list of a pascal or fortran routine from C. When using the Pascal calling convention, use the syntax

returntype (pascal * x)*(types-list)*

And when using the FORTRAN calling convention, use the syntax

returntype (fortran * x)*(types-list)*

For example, you could pass a Pascal ADSPROC to this C routine:

```
f(x)
short (pascal * x)(short);
```

In this example, x is a pointer to a pascal routine that takes a short and returns a short.

Table E.10
Equivalent Data Types: Pointers

Near Pointers

Language	Data Type	Notes
Pascal	x:^t	machine-dependent
	ADR t	
C	t near * x	
FORTRAN	T OBJECT INTEGER*2 X X=LOCNEAR(OBJECT)	

Far Pointers

Language	Data Type	Notes
Pascal	ADS t	
C	t * x	
	t far * x	
FORTRAN	T OBJECT INTEGER*4 X X=LOC(OBJECT)	
	T OBJECT INTEGER*4 X X=LOCFAR(OBJECT)	

Table E.10 *(continued)*

Procedure Pointers

Language	Data Type	Notes
Pascal	x:adsproc x:adsfunc	You must declare the procedure public so that the ADS operator can get a far address. The compiler gives near addresses for local routines.
C	t (*x) ()	
FORTRAN	T PROC EXTERNAL PROC INTEGER*4 X X=LOC(PROC) T PROC EXTERNAL PROC INTEGER*4 X X=LOCFAR(PROC)	EXTERNAL must be used if the procedure name is used other than to invoke the function (in this example, the address of the procedure is taken). Otherwise, FORTRAN will create a new variable (with the same name) and take the address of that variable, rather than of the procedure.

E.10.6 Arrays, Superarrays and Huge Arrays

FORTRAN arrays are allocated in column order. A(2,1), for example, is followed by A(3,1). C and Pascal arrays are allocated in row order. A(2,1), for example, is followed by A(2,2).

The lower bound of indices to a C array is always zero. For FORTRAN, it is always one, and for Pascal it can be any value.

For example, if you define a C array x[6][3], an equivalent array in FORTRAN would be X(3,6). An equivalent Pascal array would be x:array[0..5,0..2]. If you specify element x[5,0] in Pascal, or element x[5][0] in C, the equivalent FORTRAN element is X(1,6).

Or, if you define a Pascal array like this:

```
x:array[2..6,2..3] of integer2
```

the equivalent FORTRAN array is

```
INTEGER*2 X(2,5)
```

and the equivalent C array is

```
short x[5][2]
```

FORTRAN *large* arrays (arrays specified with the HUGE attribute or the $LARGE metacommand) cannot be used from Pascal or C.

In C, arrays are always passed by reference. If, from FORTRAN, you use the C attribute, arrays are passed by value, like C structs. That is, the entire array is laid out on the stack. To pass an array as an array (from FORTRAN to C), you must use the REFERENCE attribute, or pass the result of LOC, LOCNEAR, or LOCFAR.

Table E.11 shows equivalent array types for Pascal, C, and FORTRAN.

Table E.11
Equivalent Data Types: Arrays

Arrays: When Lower Bound of Pascal Array = 0

Language	Data Type	Notes
Pascal	x:array [0..j,0..m] of *type*	
C	*type* x[j+1][m+1]	when passed by reference
	struct { *type* x[j+1][m+1];} x	when passed by value
FORTRAN	*type* x(m+1,j+1)	

Arrays: When Lower Bound of Pascal Array is Non-Zero

Language	Data Type	Notes
Pascal	x:array [i..j,k..m] of *type*	
C	*type* x[j-i+1][m-k+1]	when passed by reference
	struct { *type* x[j-i+1][m-k+1];} x	when passed by value
FORTRAN	*type* x(m-k+1,j-i+1)	

A super array pointer is a near pointer to the start of an array, followed by the upper bounds (in the same order as they are declared). Table E.12 indicates how to specify a super array pointer from each language.

Table E.12

Equivalent Data Types: Super Array Pointers

Super Array Pointers		
Language	**Data Type**	**Notes**
Pascal	type v=super array [0..*,0..*] of *type* x:^v	
C	struct {*type* near *ptr; short *a*; short *b*;} x:	Set *a* equal to (1st-dimension-of-target -1). Set *b* equal to (2nd-dimension-of-target - 1).
FORTRAN	*none*	

E.10.7 Records and Structs

Pascal record types and C structs correspond fairly well. Variant records are more difficult, but can be used if you declare the tag field as a structure member, then build a union of all the variant parts.

In FORTRAN you can simulate a single instance of a record by using EQUIVALENCE, but there is no way to replicate the instance or apply such a structure to a parameter. If the record or struct contains only fields of the same size, you can use an array. Otherwise, you need to define an equivalence "group" with variables equivalenced so that they map on to the appropriate elements of the struct. If the whole structure is less than 127 bytes long, you can use a character variable to represent the whole structure. This means that you can assign a parameter with a single statement. This approach results in inefficient code and programs that are difficult to follow. It is recommended that you use Pascal and C to write interface procedures where possible. These could, for example, translate the structure into separate variables and scalars which are easier to use with FORTRAN.

Note that you cannot pass a Pascal set type to FORTRAN.

Table E.13
Equivalent Data Types: Complex Numbers

Single Precision Complex Numbers

Language	Data Type	Notes
Pascal	x: record re, im:real; end;	
C	struct complex8 { float re,im;} x	
FORTRAN	COMPLEX X	

Double Precision Complex Numbers

Language	Data Type	Notes
Pascal	x: record re, im:real8; end;	
C	struct complex 16 { double re,im;} x	
FORTRAN	COMPLEX*16 X	

Pascal records and C structs can also be used to pass FORTRAN logical values. For FORTRAN logical values, the integer one (1) means true. Zero (0) means false. Table E.14 gives examples of passing FORTRAN logical values.

Table E.14
Equivalent Data Types: LOGICAL Values

Two-Byte LOGICAL Values

Language	Data Type	Notes
Pascal	x: record logical: boolean; pad: array [0..0] of byte; end;	
C	struct { char logical; char pad[1]; } x;	
FORTRAN	LOGICAL*2 X	
	LOGICAL	If $STORAGE:2 is in effect

Four-Byte LOGICAL Values

Language	Data Type	Notes
Pascal	x: record logical: boolean; pad: array [0..2] of byte; end;	
C	struct { char logical; char pad[3]; } x;	
FORTRAN	LOGICAL*4 X	

E.10.8 Procedural Parameters

Formal procedural arguments in Pascal and FORTRAN are compatible. They are not compatible with procedure pointers in C.

However, Pascal and FORTRAN procedure arguments can be represented by a C struct that mimics the Pascal/FORTRAN sequence.

If you are calling C from Pascal or FORTRAN, it is recommended that you use C procedure pointers. If you want to pass a procedure to a Pascal or FORTRAN procedure, you must use Pascal arguments, since neither Pascal nor FORTRAN can call through procedure pointers. See Table E.10 for equivalent procedure pointer types.

E.10.8.1 Return Values

FORTRAN and Pascal routines can return values to a C program. For the C program to handle the return values correctly, the programmer must understand the correspondence between data types in the different languages.

The C compiler performs conversions on return values before they are actually returned to the calling procedure. These conversions are the same as those given for parameters. Integral values shorter than an int are expanded to int size, and float values are converted to double. These types are discussed in Sections E.10.1, "Integers," and E.10.3, "Real Numbers."

The C compiler detects structured return values that are 4 bytes or less in length and returns them as integers of the appropriate size.

E.11 Sharing Data

Pascal and C can refer to each other's public data items, as long as you specify appropriate attributes to use the correct naming conventions and keywords to ensure correct storage allocation. (All Pascal static variables should be declared with the near keyword in C.) FORTRAN COMMON blocks are public data areas and can be referenced as external data objects in C and

Pascal. You can use the COMMON block names as the names of struct variables in C or record variables in Pascal, for example. To access a common block from Pascal, however, the common block must have the NEAR attribute. Blank common has the public name COMMQQ. FORTRAN *cannot* access C data objects.

Alternatively, you can use the LOC procedures in FORTRAN to give the address of a common block, pass the address to a C or Pascal procedure, then use that address from C or Pascal. For example:

```
INTERFACE TO SUBROUTINE CFUNC[C] (EXTP)
INTEGER*4 EXTP
END
COMMON/EXT/I,J
CALL CFUNC (LOC(I))
    .
    .
    .
END
void cfunc (ext)
struct {long i,j;}*ext
{
        ext->i = ext ->j;
}
```

or

C When you have several common blocks to set up

```
SUBROUTINE SETADS (ADSEXT, ADSPAR, ADSBL)
INTEGER*4 ADSEXT,ADSPAR,ADSBL
COMMON/EXT/I1
COMMON/PAR/I2
COMMON I3
ADSEXT=LOCFAR(I1)
ADSPAR=LOCFAR(I2)
ADSBL=LOCFAR(I3)
END
```

E.12 Input and Output

A given file can be opened by only one language at a time, except the standard output channel when that channel refers to the terminal. In this case, each FORTRAN WRITE statement that refers to the terminal should be followed by

```
WRITE(*,*)
```

if there is a possibility that a C or Pascal routine might write to the terminal immediately thereafter. This will clear the carriage control character.

E.13 Compiling and Linking

The order in which modules are linked is important. You must make sure that you link them as follows:

1. If you are linking with the C floating point library, it must be specified first.
2. If you are using Pascal or FORTRAN, their math libraries must be specified second. The math libraries for Pascal and FORTRAN are identical, and need be specified only once when you are mixing Pascal and FORTRAN.
3. If you are using Pascal or FORTRAN, their language libraries must be specified third.
4. If you are using the C language library it must be specified last.

E.14 Error Messages

Errors that occur during compile time are generated by the compiler for the language in which the error occurs. Most runtime errors come from the language in which the part of the program causing the error was written. However, floating point errors may come from any of the languages which were used in the program. For Pascal and FORTRAN, these errors are identical. However, for C, the messages are slightly different, and there is no message number.

Appendix F
Error Messages

This appendix explains what happens when errors occur, and lists the error messages generated by the compiler, the linker, EXEMOD, and EXEPACK.

Error messages are organized by where they occur. The following table shows how error messages are numbered:

Table F.1

How Errors are Numbered

Error Code	Type of Error
1-899	Front end errors
900-999	Back end errors
1000-1099	Unit U 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 and double real arithmetic
2150-2199	Structures, sets, and strings
2200-2399	Reserved
2400-2449	Pcode interpreter
2450-2499	Other internal errors
2500-2999	Reserved

Some errors are unnumbered. These errors occur during linking, or during use of EXEMOD and EXEPACK, and are listed in the sections "Unnumbered Errors," "Linker Errors," "EXEPACK Errors," and "EXEMOD Errors."

F.1 How the Compiler Handles Error Locations

This section describes two ways the compiler displays information about error locations.

F.1.1 Source Program Context

If an error occurs during compile time, the error number, source line number, and sometimes the text of the source line containing the error are printed on your screen (and in your source listing file, if you have requested one). If the $debug+ metacommand is present, additional information, called the *source program context* appears, including

1. the source filename of the compiland containing the error
2. the name of the procedure or function containing the error
3. the listing line number of the first line of the statement containing the error

You can also get this information for errors that occur during runtime, but only if you compiled the program with the $debug+ metacommand. If you compiled without the $debug+ metacommand and errors occur during runtime, the filename and line number will not be be printed on your screen.

Giving the source program context during runtime involves extra overhead, since source location data must be included in the object code in some form. This is done with calls that set the current source context as it occurs. The overhead of source location data, especially line number calls, can be significant. Section F.1.3, "How Source Program Context is Handled," explains how the compiler provides this information.

F.1.2 Machine Error Context

There is another way of determining where an error occurred, called the *machine error context*. This information is provided in most runtime error messages. The machine error context is always available, but to use this information you must access certain global variables.

The machine error context is made up of

1. the program counter (PC) at the point of the error
2. the stack segment (SS) at the point of the error
3. the framepointer (FP) at the point of the error
4. the stackpointer (SP) at the point of the error

For runtime errors, this information will be printed on your screen in this format:

```
? Error: Data format error in file -
  Error Code 1233, Status 000E
PC = 000C: 0006; SS = 0047, FP = 003F, SP = 1C1C
```

The values of PC, SS, FP, and SP are in hexadecimal.

The machine error context does not indicate the location of errors within runtime routines. If an error is in a runtime routine, the machine error context indicates where the user program called the runtime routine or routines. The program counter is always the address following a call to a runtime routine (a return address).

You can usually use the program counter (PC) value along with a link map file to find the routine in which the error occurred. Read the addresses in the section of the link map file that is sorted by address until you find a pair that brackets the PC value. The first address of the pair should be the name of the routine that was executing when the error occurred. This does not work for routines that don't have public names. Sometimes the stack segment (SS) and stackpointer (SP) values are useful in determining if there has been a stack overflow. Check the link map file (or use a debugger); if SP is close to or below the location of the label "_end", there was probably a stack overflow.

The variables that contain the machine error context are used by the runtime entry helper, BRTEQQ, as follows:

This variable:	Contains:
RESEQQ	Stackpointer
REFEQQ	Framepointer
REPEQQ	Program counter offset
RECEQQ	Program counter segment

F.1.3 How Source Program Context is Handled

This information is not necessary for most Microsoft Pascal programmers. It is provided for advanced programmers because it can be useful for machine-language debugging or for writing specialized debugging aids.

The routine EMSEQQ is called when an error is detected during runtime. The source program context entry points are ENTEQQ, EXTEQQ, and LNTEQQ: they are in the debug module, DEBE.

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

1. the line number of the procedure (a WORD)
2. the page number of the procedure (a WORD)
3. the subroutine or function name (an LSTRING)

The filename is that of the compiland (the main source filename, not the names of any $include files). The procedure name is the full name 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 name is the program name.

The procedure exit call to EXTEQQ passes no 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.

F.2 Compile Time Error Messages

This section lists, in numerical order, all of the numbered messages issued by the Microsoft Pascal Compiler.

Error and warning messages consist of a number and a message. Most messages appear with a row of dashes and an arrow that points to the location of the error; messages 128, 129, and 130 appear after the body of the routine in which they occur.

If the word "Warning" appears before a message, the intermediate code files produced by the compiler are correct. A warning is an error that the compiler attempts to correct so that the compiled source may run correctly. Common errors that generate warnings include substitution mistakes, such as using a colon (:) instead of an equal sign (=), and syntax errors like using a semicolon (;) before an ELSE. The condition that produced the message is not severe, but is considered unsafe. If you get a warning message, you should correct the source file; otherwise you will get the same warning message every time you compile.

Errors are conditions in the program that the compiler cannot correct. The compiler recovers from most errors and continues the compilation, but the object file will not be correct. There are a few errors (called "panic" errors) from which the compiler cannot recover. When a panic error is detected the compiler displays the following message:

```
Compiler Cannot Continue!
```

and lists the rest of the program without compiling it. Writing to intermediate files stops and the intermediate files are discarded. Errors 900 through 904 also cause immediate termination.

The compiler also makes a number of internal consistency checks. These checks should always be correct and never give an internal error. When internal errors occur, the error messages have the following format:

```
Error: Compiler Internal Error
```

or

```
*** Internal Error NNN
```

NNN is the internal error number, which ranges from 1 to 999. There is little you can do when an internal error occurs, except report it to Microsoft and perhaps modify your program near the line where the error occurred.

The following list includes the error number and message, and a brief explanation of the condition that generates the message:

Code Message

101 Invalid Line Number

There are more than 32767 lines in the source file.

102 Line Too Long Truncated

There are more than 142 characters in the line.

103 Identifier Too Long Truncated

An identifier is longer than 32 characters, and has been truncated.

104 Number Too Long Truncated

A numeric constant is longer than 32 characters and has been truncated.

105 End Of String Not Found

The line ended before the closing quotation mark was found.

106 Assumed String

The compiler encountered double quotation marks (") or back quotation marks (') and assumed that they enclosed a string. Use single quotation marks (') instead.

107 Unexpected End Of File

While scanning, the compiler found an unexpected end of file in a number, metacommand, or other illegal location.

108 Meta Command Expected Command Ignored

The compiler found a dollar sign ($) at the start of a comment, but no metacommand identifier.

109 Unknown Meta Command Ignored

The compiler found a metacommand identifier that it didn't recognize or that is invalid in this version of Microsoft Pascal.

110 Constant Identifier Unknown Or Invalid Assumed Zero

The constant identifier following a metacommand is unknown (as in $debug:A) or of the wrong type. The compiler has replaced the unknown or incorrect value with zero.

112 Invalid Numeric Constant Assumed Zero

The constant following a metacommand was a numeric constant (e.g., $debug:123456) that has the wrong format or is out of range. The compiler has replaced the incorrect value with zero.

113 Invalid Meta Value Assumed Zero

The value following a metacommand is neither a constant nor an identifier. The compiler has replaced the incorrect value with zero.

114 Invalid Meta Command

The compiler expected but did not find one of the following after a metacommand: +, -, or :. The compiler has ignored the metacommand.

115 Wrong Type Value For Meta Command Skipped

The value following the metacommand was an integer, but should have been a string (or vice versa). The compiler ignored the metacommand.

116 Meta Value Out Of Range Skipped

The integer value given for the $linesize metacommand was below 16 or above 16 0. Or, "n" is neither 4 nor 8 for $real:n, nor 2 for $integer. In any of these cases, the compiler ignores the metacommand.

117 File Identifier Too Long Skipped

The string value given for the filename in an $include metacommand was longer than 96 characters. The compiler ignored the metacommand.

118 Too Many File Levels

There are too many nested levels of files brought in by the $include metacommand. The $include metacommand is ignored.

119 Invalid Initialize Metacommand

A $pop metacommand has no corresponding $push metacommand.

120 CONST Identifier Expected

The compiler didn't find an identifier following an $inconst metacommand. The $inconst metacommand is ignored.

121 Invalid INPUT Number Assumed Zero

The user input invoked by $inconst was invalid and is assumed to be zero.

122 Invalid Meta Command Skipped

The compiler found an $if metacommand but no subsequent $then or $else. The compiler ignores the $if command.

123 Unexpected Meta Command Skipped

The compiler found a $then metacommand unrelated to any $if metacommand. The $then command is ignored.

124 Unexpected Meta Command

The compiler found a metacommand not enclosed in comment delimiters, but processed it anyway.

126 Invalid Real Constant

The compiler found a type real constant with a leading or a trailing decimal point. The constant's value is accepted anyway.

127 Invalid Character Skipped

The compiler found a character in the source file that is not acceptable in program text.

128 Forward Proc Missing: *procedure*

The compiler found a procedure or function declared FORWARD but couldn't find the procedure or function itself.

129 Label Not Encountered : *label*

The compiler couldn't find any use of a label declared in a LABEL section.

130 Program Parameter Bad : *parameter*

The compiler encountered a program parameter that was never declared or has an unacceptable type.

133 Type Size Overflow

The data type declared implies a structure bigger than the maximum of 65534 bytes.

134 Constant Memory Overflow

Constant memory allocation has exceeded the maximum of 65534 bytes.

135 Static Memory Overflow

Static memory allocation has exceeded the maximum of 65534 bytes.

136 Stack Memory Overflow

Stack frame memory allocation has exceeded the maximum of 65534 bytes.

137 Integer Constant Overflow

The value of a type INTEGER, signed constant expression is out of range.

138 Word Constant Overflow

The value of a type WORD constant or other unsigned constant expression is out of range.

139 Value Not In Range For Record

In a structured constant, or in the long form of the NEW procedure, DISPOSE procedure, the SIZEOF function, or other application, the record tag value is not in the range of the variant.

140 Too Many Compiler Labels

Your program is too big. You must break your program into smaller pieces.

141 Compiler

This message should never appear. If it does, please report the condition to Microsoft Corporation.

142 Too Many Identifier Levels

The identifier scope level exceeds 15. This is a "panic" error from which the compiler cannot recover. The rest of the file is not compiled.

143 Compiler

This message should never appear. If it does, please report the condition to Microsoft Corporation.

144 Compiler

This message should never appear. If it does, please report the condition to Microsoft Corporation.

145 Identifier Already Declared

The compiler found an identifier declared more than once in a given scope level.

146 Unexpected End Of File

While parsing, the compiler found an end-of-file in the wrong place, for example in a statement or declaration.

147 : Assumed =

The compiler found a colon where there should have been an equal sign and proceeded as if the correct symbol were present.

148 = Assumed :

The compiler found an equal sign where it expected a colon and proceeded as if the correct symbol were present.

149 := Assumed =

The compiler found a colon followed by an equal sign where it expected an equal sign only and proceeded as if the correct symbol were present.

150 = Assumed :=

The compiler found an equal sign where it expected a colon followed by an equal sign and proceeded as if the correct symbol were present.

151 [Assumed (

The compiler found a left bracket where it expected a left parenthesis and proceeded as if the correct symbol were present.

152 (Assumed [

The compiler found a left parenthesis where it expected a left bracket and proceeded as if the correct symbol were present.

153) Assumed]

The compiler found a right parenthesis where it expected a right bracket and proceeded as if the correct symbol were present.

154] Assumed)

The compiler found a right bracket where it expected a right parenthesis and proceeded as if the correct symbol were present.

155 ; Assumed ,

The compiler found a semicolon where it expected a comma and proceeded as if the correct symbol were present.

156 , Assumed ;

The compiler found a comma where it expected a semicolon and proceeded as if the correct symbol were present.

162 Insert Symbol

The compiler didn't find a symbol it expected, but proceeded as if it were present. This message should not occur; it is a minor compiler error. If it does, please report it to Microsoft Corporation.

163 Insert ,

The compiler didn't find a comma where it expected one, but proceeded as if it were present.

164 Insert ;

The compiler didn't find a semicolon where it expected one, but proceeded as if it were present.

165 Insert =

The compiler didn't find an equal sign where it expected one, but proceeded as if it were present.

166 Insert :=

The compiler didn't find a colon followed by an equal sign where it expected one, but proceeded as if it were present.

167 Insert OF

The compiler didn't find an OF where it expected one, but proceeded as if it were present.

168 Insert]

The compiler didn't find a right bracket where it expected one, but proceeded as if it were present.

169 Insert)

The compiler didn't find a right parenthesis where it expected one, but proceeded as if it were present.

170 Insert [

The compiler didn't find a left bracket where it expected one, but proceeded as if it were present.

171 Insert (

The compiler didn't find a left parenthesis where it expected one, but proceeded as if it were present.

172 Insert DO

The compiler didn't find a DO where it expected one, but proceeded as if it were present.

173 Insert :

The compiler didn't find a colon where it expected one, but proceeded as if it were present.

174 Insert .

The compiler didn't find a period where it expected one, but proceeded as if it were present.

175 Insert ..

The compiler didn't find a double period where it expected one, but proceeded as if it were present.

176 Insert END

The compiler didn't find an END where it expected one, but proceeded as if it were present.

177 Insert TO

The compiler didn't find a TO where it expected one, but proceeded as if it were present.

178 Insert THEN

The compiler didn't find a THEN where it expected one, but proceeded as if it were present.

179 Insert *

The compiler didn't find an asterisk where it expected one, but proceeded as if it were present.

185 Invalid Symbol Begin Skip

The compiler found a symbol it expected, but only after some other invalid symbols. The invalid symbols were skipped, beginning at the point where message 185 appears and ending where message 186 appears.

186 End Skip

The compiler found a symbol it expected, but only after some other invalid symbols. The invalid symbols were skipped, beginning at the point where message 185 appears and ending where message 186 appears.

187 End Skip

This message marks the end of skipped source text for any message that ended with the phrase "Begin Skip," except message 185.

188 Section Or Expression Too Long

Try rearranging the program or breaking up expressions with assignments to intermediate values.

189 Invalid Set Operator Or Function

Your source file includes an incorrect use of a set operator or function (for example, attempting to use the MOD operator or the ODD function with sets).

190 Invalid Real Operator Or Function

Your source file includes an incorrect use of an operator or function on a REAL value (for example, MOD operator or ODD function with reals).

191 Invalid Value Type For Operator Or Function

For example, MOD operator or ODD function with enumerated type.

195 Compiler

This message should never appear. If it does, please report the condition to Microsoft Corporation.

196 Zero Size Value

Your source file includes the empty record "RECORD END" as if it had a size.

197 Compiler

This message should never appear. If it does, please report the condition to Microsoft Corporation.

198 Constant Expression Value Out Of Range

The value of a constant expression is out of range in an array index, subrange assignment, or other subrange.

199 Integer Type Not Compatible With Word Type

An expression tries to mix INTEGER and WORD type values. This error indicates confusing signed and unsigned arithmetic; either change the positive signed value to unsigned with WRD () or change the unsigned value (MAXINT) to signed with ORD ().

201 Types Not Assignment Compatible

You have attempted to use incompatible types in an assignment statement or value parameter. See the *Microsoft Pascal Reference Manual* for type compatibility rules.

202 Types Not Compatible In Expression

You have attempted to mix incompatible types in an expression. See the *Microsoft Pascal Reference Manual* for type compatibility rules.

203 Not Array Begin Skip

A variable followed by a left bracket (or parenthesis) is not an array. The compiler skipped from here to where message 187 appears.

204 Invalid Ordinal Expression Assumed Integer Zero

The expression has the wrong type or a type that is not ordinal. The compiler assumed the value of the expression to be zero.

205 Invalid Use Of PACKED Components

A component of a PACKED structure has no address (it may not be on a byte boundary) and cannot be passed by reference.

206 Not Record Field Ignored

A variable followed by a period is not a record, address, or file, and was ignored by the compiler.

207 Invalid Field

A valid field name does not follow a record variable and a period, and was ignored by the compiler.

208 File Dereference Considered Harmful

This message appears when you are using a file buffer as a reference parameter to a procedure, a function, or as a record in a WITH statement. When the compiler calculates the address of a file buffer variable, it cannot do the special actions normally done with buffer variables (i.e., lazy evaluation for textfiles, or concurrency for binary

files). If the file position changes, the buffer variable at this address may not be valid, thus such a practice is considered harmful.

209 Cannot Dereference Value

The variable followed by an arrow is not a pointer, address, or file; therefore the compiler cannot dereference the value pointed to.

210 Invalid Segment Address

A variable resides at a segmented address, but a default segment address is needed. You may need to make a local copy of the variable.

211 Ordinal Expression Invalid Or Not Constant

The compiler found an invalid or nonconstant expression where it expected a constant ordinal expression.

214 Out Of Range For Set 255 Assumed

The compiler found an element of a set constant whose ordinal value exceeded 255 and assumed a value of 255.

215 Type Too Long Or Contains File Begin Skip

The compiler found a structured constant that either exceeds 255 bytes or contains a FILE or LSTRING type.

216 Extra Array Components Ignored

The compiler found an array constant that had too many components for the array type. The excess components were ignored.

217 Extra Record Components Ignored

The compiler found a record constant that had too many components for the record type. The excess components were ignored.

218 Constant Value Expected Zero Assumed

The compiler found a nonconstant value in a structured constant and assumed its value was zero.

220 Compiler

This message should never appear. If it does, please report the condition to Microsoft Corporation.

221 Components Expected For Type

The compiler found too few components for the type of a structured constant.

222 Overflow 255 Components In String Constant

The compiler found a string constant that exceeded 255 bytes.

223 Use NULL

Use the predeclared constant NULL instead of two quotation marks.

224 Cannot Assign With Supertype Lstring

A super array LSTRING cannot be the source or the target of an assignment.

225 String Expression Not Constant

String concatenation with the asterisk applies only to constants.

226 String Expected Character 255 Assumed

The compiler found a string constant with no characters, perhaps the result of using NULL, and assumed the value CHR(255).

227 Invalid Address Of Function

An assignment or other address reference to the function value is not within the scope of the function. Or, RESULT is used outside the scope of the function .

228 Cannot Assign To Variable

Assignment to READONLY, CONST, or FOR control variables is not permitted.

230 Unknown Identifier Assumed Integer Begin Skip

The compiler found an unknown identifier, for which it requires an address, and has skipped to a comma, semicolon, or right parenthesis.

231 VAR Parameter Or WITH Record Assumed Integer Begin Skip

The compiler found an invalid symbol where it requires an address, and has skipped to a comma, semicolon, or right parenthesis.

232 Cannot Assign To Type

The target of an assignment is a file or cannot be assigned for some other reason.

233 Invalid Procedure Or Function Parameter Begin Skip

The compiler found an incorrect use of an intrinsic procedure or function. The error could be one of the following:

1. The first parameter of NEW or DISPOSE is not a pointer variable.
2. The record tag value of a NEW, DISPOSE, or SIZEOF couldn't be found.
3. The super array in a NEW, DISPOSE, or SIZEOF had too many bounds.
4. The super array in a NEW, DISPOSE, or SIZEOF had too few bounds.
5. The super array for a NEW or SIZEOF has been given no bounds.
6. You attempted to use a WRD or ORD function on a value not of an ordinal type.
7. You attempted to use the LOWER or UPPER functions on an invalid value or type.
8. You attempted to use PACK or UNPACK on a super array or file, or an array that is or is not packed as expected.
9. The first parameter for a RETYPE is not a type identifier.

10. The parameter for a RESULT function is not a function identifier.
11. You attempted to use an intrinsic procedure or function not available in this version of Microsoft Pascal.
12. The ORD or WRD of an INTEGER4 value is out of range.
13. The parameter given for HIWORD or LOWORD is not an ordinal or INTEGER4.

234 Type Invalid Assumed Integer

The parameter given to READ, WRITE, ENCODE, or DECODE is not of type INTEGER, WORD, INTEGER4, REAL, BOOLEAN, enumerated, a pointer; or, the parameter given for a READ or WRITE is not of type CHAR, STRING, LSTRING; or, the parameter for a READFN is not of one of these types or type FILE. The compiler assumed it to be of type INTEGER. This error also occurs if a program parameter does not have a readable type, in which case the error occurs at the keyword BEGIN for the main program.

235 Assumed File INPUT

Because the first parameter for a READFN is not a file, INPUT is assumed.

236 Invalid Segment For File

File parameters must always reside in the default segment.

237 Assumed INPUT

INPUT was not given as a program parameter and has been assumed.

238 Assumed OUTPUT

OUTPUT was not given as a program parameter and has been assumed.

239 Not Lstring Or Invalid Segment

The target of a READSET, ENCODE, or DECODE must be an LSTRING in the default segment. One or both of these conditions is missing.

242 File Parameter Expected Begin Skip

The READSET procedure expects, but cannot find, a textfile parameter. The compiler ignored the procedure and resumed where message 187 appears.

243 Character Set Expected

The READSET procedure expects, but cannot find, a SET OF CHAR parameter.

244 Unexpected Parameter Begin Skip

The compiler found more than one parameter given for an EOF, EOLN, or PAGE, and ignored the extra one.

245 Not Text File

You attempted to use an EOLN, PAGE, READLN, or WRITELN on a file that is not a textfile.

248 Size Not Identical

The RETYPE function may not work as intended, since the parameters given are of unequal length.

249 Procedural Type Parameter List Not Compatible

The parameter lists for formal and actual procedure parameters are not compatible. That is, the number of parameters, the function result type, a parameter type, or attributes are different.

250 Cannot Use Procedure With Attribute

You attempted to call a procedure with the attribute INTERRUPT, directly or in directly. INTERRUPT does not allow this.

251 Unexpected Parameter Begin Skip

The compiler found a left parenthesis, indicating a procedure or function, but no parameters, and has skipped to where message 187 appears.

252 Cannot Use Procedure Or Function As Parameter

You attempted to pass an intrinsic procedure or function as a parameter, which is not permitted.

253 Parameter Not Procedure Or Function Begin Skip

The compiler expected, but cannot find, a procedural parameter here, and has skipped to where message 187 appears.

254 Supertype Array Parameter Not Compatible

The actual parameter given is not of the same type or is not derived from the same super type as the formal parameter.

255 Compiler

This message should never appear. If it does, please report the condition to Microsoft Corporation.

256 VAR Or CONST Parameter Types Not Identical

The actual and formal reference parameter types are not identical, as they must be.

257 Parameter List Size Wrong Begin Skip

The compiler found too many or too few parameters in a list. If too many, the excess parameters were skipped.

258 Invalid Procedural Parameter To EXTERN

A procedure or function that is neither PUBLIC nor EXTERN is being passed as a parameter to a procedure or function declared EXTERN. (The compiler invokes the actual procedure or function with intrasegment calls, and so cannot pass them to an external code segment.)

259 Invalid Set Constant For Type

The set is not constant, base types are not identical, or the constant is too big.

260 Unknown Identifier In Expression Assumed Zero

The identifier in an expression is undefined or possibly misspelled.

261 Identifier Wrong In Expression Assumed Zero

The identifier in an expression is incorrect (e.g., file type identifier) and was assumed to be zero.

262 Assumed Parameter Index Or Field Begin Skip

After error 260 or 261, anything in parentheses or square brackets, or a dot followed by an identifier, is skipped.

265 Invalid Numeric Constant Assumed Zero

There is a decode error in an assumed INTEGER or INTEGER4 literal constant; the number may be too big, or contain invalid characters. The incorrect constant was assumed to be zero.

267 Invalid Real Numeric Constant

There is a decode error in an assumed type REAL literal constant; the number may be too big, or contain invalid characters.

268 Cannot Begin Expression Skipped

The compiler found an illegal symbol at the beginning of an expression and deleted the symbol.

269 Cannot Begin Expression Assumed Zero

At the beginning of an expression the compiler found a symbol that is permitted within an expression, but not at the beginning (a floating-point number beginning with a . instead of a 0, for example). The compiler placed a 0 before the symbol.

270 Constant Overflow

The divisor in a DIV or MOD function is the constant zero (INTEGER or WORD), which is not permitted.

272 Word Constant Overflow

A WORD constant minus a WORD constant gave a negative result. A WORD constant is always a positive value.

275 Invalid Range

The lower bound of a subrange is greater than the upper bound (e.g., 2..1).

276 CASE Constant Expected

The compiler expects, but cannot find, a constant value for a CASE statement or record variant.

277 Value Already In Use

In a CASE statement or record variant, the value was already assigned (as in CASE 1..3: XXX; 2: YYY; END).

279 Label Expected

The compiler expects, but cannot find, a label.

280 Invalid Integer Label

A label uses nondecimal notation (e.g., 8_ 77), which is not allowed.

281 Label Assumed Declared

The compiler found a label that did not appear in the LABEL section.

283 Expression Not Boolean Type

The expression following an IF, WHILE, or UNTIL statement must be BOOLEAN.

284 Skip To End Of Statement

The compiler found, and skipped, an unexpected ELSE or UNTIL clause.

285 Compiler

This message should never appear. If it does, please report the condition to Microsoft Corporation.

286 ; Ignored

The compiler found, and ignored, a semicolon before an ELSE statement. (The semicolon is not required in this case.)

288 : Skipped

The compiler found, and ignored, a colon after an OTHERWISE statement. (The colon is not required in this case.)

289 Variable Expected For FOR Statement Begin Skip

The compiler expected, but couldn't find, a variable identifier after a FOR statement and skipped to where message 187 appears.

291 FOR Variable Not Ordinal Or Static Or Declared In Procedure

The compiler found an incorrect control variable in a FOR statement. Control variables may not be any of the following:

1. type REAL, INTEGER4, or another non-ordinal type
2. the component of an array, record, or file type
3. the referent of a pointer type or address type
4. in the stack or heap, unless locally declared
5. nonlocally declared, unless in static memory
6. a reference parameter (VAR or VARS parameter)
7. a variable with a segmented ORIGIN attribute

292 Skip To :=

The compiler expected, but could not find, an assignment in a FOR statement, and skipped to the next :=

293 GOTO Invalid

The GOTO or label involves an invalid GOTO statement or a nonexistent label.

294 GOTO Considered Harmful

If the $goto metacommand is on, the compiler found a GOTO statement.

296 Label Not Loop Label

The label after a BREAK or CYCLE statement is not a loop label (i.e., does not label a FOR, WHILE, or REPEAT statement).

297 Not In Loop

The compiler found a BREAK or CYCLE statement outside a FOR, WHILE, or REPEAT statement.

298 Record Expected Begin Skip

The compiler expected, but could not find, a record variable in a WITH statement and skipped to where message 187 appears.

300 Label Already In Use Previous Use Ignored

The compiler found a label that had already been used and ignored the previous use.

301 Invalid Use Of Procedure Or Function Parameter

The compiler found a procedure parameter used as a function or a function parameter used as a procedure.

303 Unknown Identifier Skip Statement

The compiler found an undefined (or possibly misspelled) identifier at the beginning of a statement and ignored the entire statement.

304 Invalid Identifier Skip Statement

The compiler found an incorrect identifier at the beginning of a statement (e. g., file type identifier) and ignored the entire statement.

305 Statement Not Expected

The compiler found a MODULE or uninitialized IMPLEMENTATION with a body enclosed with the reserved words BEGIN and END.

306 Function Assignment Not Found

The compiler expects, but cannot find, an assignment of the value of a function somewhere in its body.

307 Unexpected END Skipped

The compiler found, and ignored, an END without a matching BEGIN, CASE, or RECORD.

308 Compiler

This message should never appear. If it does, please report the condition to Microsoft Corporation.

309 Attribute Invalid

The compiler found an attribute valid only for procedures and functions given to a variable, an attribute valid only for a variable given to a procedure or function, or an invalid mix of attributes (e.g., PUBLIC and EXTERN).

310 Attribute Expected

The compiler expected, but could not find, a valid attribute following the left bracket.

311 Skip To Identifier

The compiler skipped an invalid (i.e., unexpected) symbol to get to the identifier that follows.

312 Identifier Expected

The compiler expected, but could not find, a list of identifiers.

314 Identifier Expected Skip To ;

The compiler expected, but could not find, the declaration of a new identifier, and skipped to the next semicolon.

315 Type Unknown Or Invalid Assumed Integer Begin Skip

The return type for a parameter or function is incorrect; that is, it is not an identifier or is undeclared, or the value parameter or function value is a file or super array. The compiler assumed the type is INTEGER and skipped to where message 187 appears.

316 Identifier Expected

The compiler expects, but cannot find, an identifier after the word PROCEDURE or FUNCTION in parameter list.

318 Compiler

This message should never appear. If it does, please report the condition to Microsoft Corporation.

319 Compiler

This message should never appear. If it does, please report the condition to Microsoft Corporation.

320 Previous Forward Skip Parameter List

The compiler found a definition of a FORWARD (or INTERFACE) procedure or function that unnecessarily repeats the parameter list and function return type.

321 Not EXTERN

The compiler found a procedure or function with the ORIGIN attribute but missing the required EXTERN attribute.

322 Invalid Attribute With Function Or Parameter

The compiler found an incorrectly used INTERRUPT procedure, that is, one that has parameters or is a function.

323 Invalid Attribute In Procedure Or Function

The compiler found a nested procedure or function that has attributes or is declared EXTERN. Neither of these conditions is permitted.

324 Compiler

This message should never appear. If it does, please report the condition to Microsoft Corporation.

325 Already Forward

You attempted to use FORWARD twice for the same procedure or function.

326 Identifier Expected For Procedure Or Function

The compiler expects, but cannot find, an identifier following the keywords PROCEDURE or FUNCTION.

327 Invalid Symbol Skipped

The compiler found, and ignored, a FORWARD or EXTERN directive in an interface.

328 EXTERN Invalid With Attribute

The compiler found an EXTERN procedure also declared PUBLIC. This is not permitted.

329 Ordinal Type Identifier Expected Integer Assumed Begin Skip

The compiler expected, but could not find, an ordinal type identifier for a record tag type. It skipped what was given in the source file and assumed type INTEGER.

330 Contains File Cannot Initialize

You have used a file in a record variant. This is allowed, but considered unsafe, and is not initialized automatically with the usual NEWFQQ call.

331 Type Identifier Expected Assumed Character

The compiler expects, but cannot find, an ordinal type identifier. It assumes that what it does find is of type CHAR.

333 Not Supertype Assumed String

The compiler found what looks like a super array type designator. However, the type identifier is not for a super array type, so the compiler assumed it to be of the super array type STRING.

334 Type Expected Integer Assumed

The compiler expected, but could not find, a type clause or type identifier and assumed the expected type to be type INTEGER.

335 Out Of Range 255 For Lstring

The compiler found an LSTRING designator whose upper bound exceeds 255.

336 Cannot Use Supertype Use Designator

A super array type can be used only as a reference parameter or a pointer referent. Other variables cannot be given a super array type. Use a super array designator.

337 Supertype Designator Not Found

The compiler expected, but could not find, a super array designator that gives the upper bounds of the super array.

338 Contains File Cannot Initialize

The compiler found a super array of a file type. While allowed, this is considered unsafe and is not initialized automatically with the usual NEWFQQ call.

339 Supertype Not Array Skip To ; Assumed Integer

The compiler expected, but could not find, the keyword ARRAY following SUPER in a type clause. It assumed that the type is INTEGER and skipped to the next semicolon.

340 Invalid Set Range Integer Zero To 255 Assumed

The compiler found an invalid range for the base type of a set and assumed it to be of type INTEGER with a range from zero to 255.

341 File Contains File

The compiler found, but does not permit, a file type that contains a file type, either directly or indirectly.

342 PACKED Identifier Invalid Ignored

The compiler expected, but could not find, one of words ARRAY, RECORD, SET, or FILE following the reserved word PACKED. A type identifier following PACKED is not permitted.

343 Unexpected PACKED

The compiler found the keyword PACKED applied to a nonstructured type.

345 Skip To ;

The compiler expected, but could not find, a semicolon at the end of a declaration which is not at the end of the line. It assumed the next semicolon is the end of the declaration.

346 Insert ;

The compiler expected, but could not find, a semicolon at the end of the declaration which coincides with the end of a line. It inserted a semicolon where it expected to find one.

347 Cannot Use Value Section With ROM Memory

If the $rom metacommand is on, your program may not contain a VALUE section.

348 UNIT Procedure Or Function Invalid EXTERN

A required EXTERN declaration occurs later than it should in an IMPLEMENTATION. Any interface procedures and functions not implemented must be declared EXTERN at the beginning.

350 Not Array Begin Skip

In a VALUE section, the variable followed by a left bracket is not an array.

351 Not Record Begin Skip

The variable followed by a period, in a VALUE section, is not a record type.

352 Invalid Field

Within a VALUE section, the identifier assumed to be a field is not in the record.

353 Constant Value Expected

Within a VALUE section, a variable has been initialized to something other than a constant.

354 Not Assignment Operator Skip To ;

Within a VALUE section, the assignment operator is missing.

355 Cannot Initialize Identifier Skip To ;

Within a VALUE section, there is a symbol that is not a variable declared at this level in fixed (STATIC) memory. Or, it has an illegal ORIGIN or EXTERN attribute.

356 Cannot Use Value Section

A VALUE section has been incorrectly included in the INTERFACE, rather than in the IMPLEMENTATION.

357 Unknown Forward Pointer Type Assumed Integer

The identifier for the referent of a reference type declared earlier in this TYPE (or VAR) section was never declared itself.

358 Pointer Type Assumed Forward

The TYPE section includes a pointer or address type for which the referent type was already declared in an enclosing scope. Since the identifier for the referent type was declared again later in the same TYPE section, the compiler used the second definition. In the following example the forward type, REAL, is used:

```
PROGRAM outside;
TYPE a = WORD;
PROCEDURE b;
TYPE c= a;
a = REAL;
```

359 Cannot Use Label Section

The compiler found a LABEL section incorrectly included in an INTERFACE, rather than in an IMPLEMENTATION.

360 Forward Pointer To Supertype

The referent of a reference type declared in this TYPE section is a super array type. The declaration of the super array type doesn't occur until after the reference.

361 Constant Expression Expected Zero Assumed

An expression in a CONST section is not a constant.

362 Attribute Invalid

A VAR section mixes incorrectly the PUBLIC or ORIGIN attribute with EXTERN. Or, ORIGIN appears in attribute brackets after the keyword VAR.

364 Contains File Initialize Module

The compiler found an uninitialized file variable in a module. You must call the module as a parameterless procedure to initialize the files.

365 Origin Variable Contains File Cannot Initialize

The compiler found an uninitialized file variable with the ORIGIN attribute. Since ORIGIN variables are never initialized, you must initialize this file yourself.

366 UNIT Identifier Expected Skip To ;

The compiler expects, but cannot find, an identifier after the keyword USES.

367 Initialize Module To Initialize UNIT

You must call the module as a procedure in order to initialize it (a USES clause triggers a unit initialization call).

368 Identifier List Too Long Extra Assumed Integer

In a USES clause with a list of identifiers, the compiler found more identifiers in the list than are constituents of the interface. The extra ones are assumed to be type identifiers identical to INTEGER.

369 End Of UNIT Identifier List Ignored

In a USES clause with a list of identifiers, the compiler found fewer identifiers in the list than are constituents of the interface. The remaining interface constituents are not provided as part of the USES clause.

371 UNIT Identifier Expected

An identifier is missing after the phrase "INTERFACE; UNIT."

372 Compiler

Compiler expects, but cannot find, the keyword UNIT in an INTERFACE.

373 Identifier In UNIT List Not Declared

One of the identifiers in the interface UNIT list was not declared in the body of the interface.

374 Program Identifier Expected

An identifier is missing after the keyword PROGRAM or MODULE. This is a "panic" error from which the compiler cannot recover. The rest of the file is not compiled.

375 UNIT Identifier Expected

The unit identifier is missing after the phrase "IMPLEMENTATION OF." This is a "panic" error from which the compiler cannot recover. The rest of the file is not compiled.

376 Program Not Found

The compiler expects, but cannot find, one of the reserved words PROGRAM, MODULE, or IMPLEMENTATION OF. This is a "panic" error from which the compiler cannot recover. The rest of the file is not compiled. (This error can occur if the source file is not a Microsoft Pascal compiland.)

377 File End Expected Skip To End

The compiler found additional source text after what appeared to be the end and ignored everything after that point.

378 Program Not Found

The compiler expected, but could not find, the main body of a compiland, or the final END.

F.3 Compiler Back End Errors

The following errors cause immediate termination of the compilation. No object file is created. The main source of these errors is user error from either the optimizer or the code generator.

900 Attempt to divide by zero.

For example you attempted A DIV 0.

901 Overflow during integer constant folding.

For example, you attempted MAXINT + A + MAXINT.

902 Expression too complex/Too many internal labels.

Try breaking up expression with intermediate value assigns.

903 Too many procedures and/or functions.

Try breaking up compiland into modules or units.

904 Range error (number too large to fit into target).

F.4 Runtime File System Errors (1000-1199)

File system error codes range from 1000 to 1199. Error codes go into the ERRC field of the file control block. Codes from 1000 to 1099 designate errors (from Unit U) that are specific to your operating system, while those codes from 1100 to 1199 identify Pascal file system errors (from Unit F).

File system errors all have the format

error type error in file *filename*

followed by the error code, and in some versions an error status, which is an operating system error return word. The *error type* codes, based on the ERRS field of the file control block, are:

Code Message

1 Hard Data

Hard data error (parity, CRC, check sum, etc.).

2 Device Name

Invalid unit/device/volume name, format, or number.

3 Operation

Invalid operation: GET if EOF, RESET a printer, etc.

4 File System

File system internal error, ERRS > 15, etc.

5 Device Offline

Unit/device/volume no longer available.

6 Lost File

File itself no longer available.

7 File Name

Invalid syntax, name too long, no temporary names, etc.

8 Device Full

Disk full, directory full, all channels allocated.

9 Unknown Device

Unit/device/volume not found.

10 File Not Found

File itself not found.

11 Protected File

Duplicate filename; write-protected.

12 File In Use

File in use, concurrency lock, already open.

13 File Not Open

File closed, I/O to unopen FCB.

14 Data Format

Data format error, decode error, range error.

15 Line Too Long

Buffer overflow, line too long.

F.4.1 Operating System Runtime Errors (1000-1099)

Errors 1000 through 1048 are specific to the MS-DOS operating system:

Code	Message
1000	Write error when writing end of file
1002	Filename extension with more than 3 characters.
1003	Error during creation of new file (disk or directory full.)
1004	Error during open of existing file (file not found.)
1005	Filename with more than 8 characters, or zero characters
1007	Total filename length over 21 characters.
1008	Write error when advancing to next record.
1009	File too big (over 65535 logical sectors).
1010	Write error when seeking to direct record.
1011	Attempt to open a random file to a nondisk device
1027	Filename error
1028	Device full error
1030	File system
1031	Operation
1032	File not found
1033	File not found
1034	File system
1035	Protected file
1036	File system
1037	File system
1038	File system

1039 File system

1040 File system

1041 Data format

1042 File system

1043 Data format

1044 File system

1045 Unknown Device

1046 File system

1047 File system

1048 File system

F.4.2 Microsoft Pascal File System Errors (1100-1199)

Code	Message
1100	**ASSIGN or READFN of filename to open file** This error is only caught for textfiles.
1101	**Reference to buffer variable of closed textfile**
1102	**Textfile READ or WRITE call to closed file**
1103	**READ when EOF is true (SEQUENTIAL mode)**
1104	**READ to REWRITE file, or WRITE to RESET file (SEQUENTIAL mode)**
1105	**EOF call to closed file**
1106	**GET call to closed file**
1107	**GET call when EOF is true (SEQUENTIAL mode)**
1108	**GET call to REWRITE file (SEQUENTIAL mode)**
1109	**PUT call to closed file**
1110	**PUT call to RESET file (SEQUENTIAL mode)**
1111	**Line too long in DIRECT textfile**

1112 **Decode error in textfile READ BOOLEAN**

1113 **Value out of range in textfile READ CHAR**

1114 **Decode error in textfile READ INTEGER**

1115 **Decode error in textfile READ SINT (integer subrange)**

1116 **Decode error in textfile READ REAL**

1117 **LSTRING target not big enough in READSET**

1118 **Decode error in textfile READ WORD**

1119 **Decode error in textfile READ BYTE (word subrange)**

1120 **SEEK call to closed file**

1121 **SEEK call to file not in DIRECT mode**

1122 **Encode error (field width > 255) in textfile WRITE BOOLEAN**

1123 **Encode error (field width > 255) in textfile WRITE INTEGER**

1124 **Encode error (field width > 255) in textfile WRITE REAL**

1125 **Encode error (field width > 255) in textfile WRITE WORD**

1126 **Decode error (field width > 255) in textfile READ INTEGER4**

1127 **Encode error (field width > 255) in textfile WRITE INTEGER4**

F.4.3 Other Runtime Errors (2000-2999)

Nonfile system error codes range from 2000 to 2999. In some cases, metacommands control whether or not the compiler checks for the error. In other cases, the compiler always checks. The list below indicates which, if any, metacommand controls the error checking.

F.4.4 Memory Errors (2000-2049)

Code	Message

2000 Stack Overflow

The stack (frame) ran out of memory while calling a procedure or function. This condition is checked if the $stackck metacommand is on, and may be checked in some other cases.

2001 No Room In Heap

The heap ran out of room for a new variable during the NEW (GETHQQ) procedure.

2002 Heap Is Invalid

During the NEW (GETHQQ) procedure, the allocation algorithm discovered the heap structure is corrupt.

2003 Heap Allocator Interrupted

An interrupt procedure interrupted NEW (GETHQQ) and did a NEW call itself. The heap allocator modifies the heap, so it is a critical section. This error is not caught in this version.

2004 Allocation Internal Error

There was an unexpected error return when GETHQQ was requesting additional heap space from the operating system. Please report occurrences of this error to Microsoft Corporation.

2031 NIL Pointer Reference

DISPOSE or $nilck+ found a pointer with a NIL (i.e., 0) value.

2032 Uninitialized Pointer

DISPOSE or $nilck+ found an uninitialized (value 1) pointer. This occurs only if the metacommand $initck is on.

2033 Invalid Pointer Range

DISPOSE or $nilck+ found a pointer that does not point into the heap or is otherwise invalid. (It may have pointed to a disposed block that was removed from the heap and given back to the system.)

2034 Pointer To Disposed Var

DISPOSE or $nilck+ found a pointer to a heap block that has been disposed. Calling DISPOSE twice for the same variable is invalid.

2035 Long DISPOSE Sizes Unequal

In a long form of DISPOSE, the actual length of the variable did not equal the length based on the tag values given.

F.4.5 Ordinal Arithmetic Errors (2050-2099)

Code Message

2050 No CASE Value Matches Selector

In a CASE statement without an OTHERWISE clause, none of the branch statements had a CASE constant value equal to the selector expression value. This error is checked only if the $rangeck metacommand is on.

2051 Unsigned Divide By Zero

A WORD value was divided by zero. This error is checked only if the $mathck metacommand is on.

2052 Signed Divide By Zero

An INTEGER value was divided by zero. This error is checked only if the $mathck metacommand is on.

2053 Unsigned Math Overflow

A WORD result is outside the range zero to MAXWORD. This error is checked only if the $mathck metacommand is on.

2054 Signed Math Overflow

An INTEGER result is outside the range from -MAXINT to +MAXINT. This error is checked only if the $mathck metacommand is on.

2055 Unsigned Value Out Of Range

The source value for assignment or value parameter is out of range for the target value. The target may be a subrange of WORD (including BYTE), or CHAR, or an enumerated type. This error can also occur in SUCC and PRED functions and when the length of an LSTRING is assigned. All of these conditions are checked if the $rangeck metacommand is on.

The error also occurs when an array index is out of bounds and the array has an unsigned index type. This condition is checked when the $indexck metacommand is on.

2056 Signed Value Out Of Range

This error is similar to message 2055, but applies to the INTEGER type and its subranges.

2057 Uninitialized 16 Bit Integer Used

Either an INTEGER or 16-bit INTEGER subrange variable is used without being assigned first, or such a variable has the invalid value of -32768. This condition is checked if the $initck metacommand is on.

2058 Uninitialized 8 Bit Integer Used

Either a SINT or 8-bit INTEGER subrange variable is used without being assigned first, or such a variable has the invalid value of -128. This condition is checked if the $initck metacommand is on.

2084 Integer Zero To Negative Power

There was an attempt to raise zero to a negative power.

F.4.6 Type REAL Arithmetic Errors (2100-2149)

Code	Message

2100 REAL Divide By Zero

A REAL value is divided by zero.

2101 REAL Math Overflow

A REAL value is too large for representation.

2102 SIN or COS Argument Range

The parameter for a SIN or COS function is too large to yield a meaningful result.

2103 EXP Argument Range

The parameter for an EXP function is too large to yield a result that fits in representation.

2104 SQRT of Negative Argument

The parameter for a square root function is less than zero.

2105 LN of Non-Positive Argument

The parameter of a natural log function is less than or equal to zero.

2106 TRUNC/ROUND Argument Range

The REAL parameter of a TRUNC, TRUNC4, ROUND, or ROUND4 function is outside the range of INTEGERs.

2131 Arctan Argument 0

The parameter for a TANRQQ function is so small that the result is invalid.

2132 Arcsin or Arccos of REAL > 1.0

The parameter of an ASNRQQ or ACSRQQ function is greater than one.

2133 Negative Real To Real Power

The first argument of an PRDRQQ or PRSRQQ function is less than zero.

2134 Real Zero To Negative Power

There was an attempt to raise zero to a negative power in one of the functions PISRQQ, PIDRQQ, PRDRQQ, or PRSRQQ.

2135 REAL Math Underflow

The significance of a REAL expression has been reduced to zero.

2136 REAL Indefinite (Uninitialized Or Previous Error)

The REAL value called "infinity" was encountered. This may occur if the $initck metacommand is on and an uninitialized REAL value is used, or if a previous error set a variable to indefinite as part of its masked error response.

2137 Missing Arithmetic Processor

You linked your program with the runtime library intended for use with the 80287 numeric coprocessor, but there is no coprocessor on your system. Relink your program with the runtime library that emulates floating-point arithmetic.

2138 REAL IEEE Denormal Detected

A very small real number was generated and may no longer be valid due to loss of significance.

2139 REAL Precision Loss

An arithmetic operation on the 80287 numeric coprocessor has generated a loss of numeric precision in the result of an operation.

2140 REAL Arithmetic Processor Instruction Illegal Or Not Emulated

An attempt was made to execute an illegal arithmetic coprocessor instruction, or the floating-point emulator cannot emulate a legal coprocessor instruction.

2145 Real Stack Overflow

Using the alternate math package and expression, too many real operands were encountered.

F.4.7 Structured Type Errors (2150-2199)

Code Message

2150 String Too Long in COPYSTR

The source string for a COPYSTR intrinsic function is too large for the target string.

2151 Lstring Too Long In Intrinsic Procedure

The target LSTRING is too small in an INSERT, DELETE, CONCAT, or COPYLST intrinsic procedure.

2180 Set Element Greater Than 255

The value in a constructed set exceeds the maximum of 255.

2181 Set Element Out Of Range

The value in a set assignment or set value parameter is too large for the target set. This error is caught only if the $rangeck metacommand is on.

F.4.8 INTEGER4 Errors (2200-2249)

Code Message

2200 Long Integer Divide By Zero

An INTEGER4 value is divided by zero.

2201 Long Integer Math Overflow

An INTEGER4 value is too large for representation.

2234 Long Integer Zero To Negative Power

There was an attempt to raise zero to a negative power.

F.4.9 Other Errors (2400-2999)

Code **Message**

2450 **Unit Version Number Mismatch**

During unit initialization, the user (contains the USES clause) and implementation of an interface were discovered to have been compiled with unequal interface version numbers.

F.5 Unnumbered Error Messages

Compiler Cannot Continue

This error occurs under the following circumstances:

- There are more errors than the number set by the $errors metacommand.
- An end-of-file occurs when not expected.
- Identifier scopes are nested too deeply.
- The compiler cannot find the keyword PROGRAM, MODULE, or IMPLEMENTATION.
- The compiler cannot find the PROGRAM, MODULE, or IMPLEMENTATION identifier.
- A file system error occurs. File system error messages include the filename and one of the following phrases:

```
HARD DATA       (check sum error)
DISK FULL       (disk is full)
FILE ACCESS     (file not found)
FILE SYSTEM     (other or internal error)
```

Error: Compiler Internal Error

This error signifies a serious problem with the compiler. This message should never occur; if it does please report it to Microsoft Corporation. There isn't much you can do if this error occurs except perhaps modify your program near the line where the error occurred.

Error: Compiler Out of Memory

This error usually occurs when too many identifiers have been declared. Refer to Section 3.2, "Working With Large Programs" for suggestions on how to avoid this problem.

F.6 Linker Error Messages

This section lists the error messages that can occur when linking programs. The messages are in alphabetical order.

About to generate .EXE file. Change diskette in drive A: and press ENTER.

This message appears before the .EXE has been written if the /P switch is given. Insert diskette that the .EXE file is to be written to into the specified drive (Drive A, for example).

Ambiguous switch error: '*x*'

You did not enter a correct switch name after the switch indicator '/'. For example, the command

```
A>LINK /N main;
```

will generate this error. LINK will abort.

Attempt to put segment *name* in more than one group in file *filename*

A segment was declared to be a member of two different groups. Correct the source and recreate the object files.

Cannot find library: *filename*.lib. Enter new file spec:

The linker cannot find *filename*.lib and is requesting a new file name or a new path specification or both. Respond to the prompt with a new filename or a new path specification.

Cannot open list file

The directory or disk is full. Make space on the disk or in the directory.

Cannot open response file

You named a response file the linker cannot open. You have probably made a typing mistake.

Cannot nest response files

You named another response file in the response file. Fix the file and relink.

Cannot open run file

The directory or disk is full. Make space on the disk or in the directory.

Cannot open temporary file

The directory or disk is full. Make space on the disk or in the directory.

Cannot reopen list file

You did not replace the original disk when prompted to. Restart the linker.

Data record too large

The LEDATA record (in an object module) contains more than 1024 bytes of data. This is a translator error. Note the translator (compiler or assembler) that produced the incorrect object module and the circumstances under which it was produced, and report the information to Microsoft Corporation.

Dup record too large

The LIDATA record (in an object module) contains more than 512 bytes of data. Most likely, an assembly module contains a struc definition that is very complex, or a series of deeply nested DUP statements (e.g. ARRAY db 10 dup (11 dup (12 dup (13 dup (...))))). Simplify the module and reassemble it.

filename is not a valid library

The file specified as a library is invalid. LINK will abort.

Fixup overflow near *num* in segment *name* in *filename*(*name*) offset *num*

Some possible causes of this message are:

1. A group is larger than 64K bytes
2. Your program contains an intersegment short jump or intersegment short call
3. You have a data item whose name conflicts with that of a subroutine in a library included in the link, and
4. You have an EXTRN declaration inside the body of a segment, for example:

```
CODE    segment public 'code'
extrn   main:far
start   proc    far
        call    main
        ret
start   endp
CODE    ends
```

The following construction is preferred:

```
extrn   main:far
CODE    segment public 'code'
start   proc    far
        call    main
        ret
start   endp
CODE    ends
```

Revise the source and recreate the object file.

Incorrect DOS version, use DOS 2.0 or later

LINK runs only on MS-DOS Version 2.0 or higher. Restart your system using the correct MS-DOS version and try linking again.

Insufficient stack space

There is not enough memory to run the linker.

Interrupt number exceeds 255

You have specified a number greater than 255 after the /OVERLAYINTERRUPT switch. Try again with a number in the range 4 to 255.

Invalid numeric switch specification

You have probably made a typographical error when you entered a value for one of the linker switches, such as entering a character string for a switch that requires a numeric value. LINK will abort.

Invalid object module

One of the object modules is invalid. Try recompiling. If the error persists, contact Microsoft Corporation.

Nested left parentheses

You have probably made a typing mistake while specifying the contents of an overlay on the command line.

No object modules specified

You didn't give the linker an object file name.

Out of space on list file

The disk on which the list file is being written is full. Free more space on the disk and try again.

Out of space on run file

The disk on which the .EXE file is being written is full. Free more space on the disk and try again.

Out of space on scratch file

The disk in the default drive is full. Delete some files on that disk, or replace with another disk, and restart the linker.

Overlay manager symbol already defined: *name*

You have defined a symbol name that conflicts with one of the special overlay manager names. Change the symbol name and relink.

Please replace original disk in drive A: and press ENTER.

This message appears after the .EXE file has been written if the /P switch is used. Insert the disk with the list file so that it can be reopened.

Relocation table overflow

You have more than 16384 long calls, long jumps or other long pointers in your program. First try turning debugging off, then try rewriting the program, replacing long references with short references where possible. Then recreate the object module.

Segment limit set too high

Using the /SEGMENTS switch, you set the limit too high. LINK will abort.

Segment limit too high

There is not enough memory for the linker to allocate tables to describe the number of segments requested (either the value specified with /SEGMENTS or the default: 128). Either try the link again using /SEGMENTS to select a smaller number of segments (e.g. 64, if the default were used previously) or free some memory.

Segment size exceeds 64K

You have a small model program with more than 64K bytes of code.

Stack size exceeds 65536 bytes

The size specified for the stack with the /STACK switch is more than 65536 bytes.

Symbol table overflow

Your program has more than 256K of symbolic information (Publics, extrns, segments, groups, classes, files, etc). Combine modules and/or segments and recreate the object files. Eliminate as many public symbols as possible.

Terminated by user

You entered Ctrl-C while the linker was executing.

Too many external symbols in one module

Your object module specified more than the allowed number of external symbols. Break up the module.

Too many group-, segment-, and class-names in one module

Your program contains too many group, segment, and class names. Reduce the number of groups, segments, or classes and recreate the object files.

Too many groups

Your program defines more than nine groups. Reduce the number of groups.

Too many GRPDEFs in one module

LINK encountered more than nine GRPDEFs in a single module. Reduce the number of GRPDEFs or split up the module.

Too many libraries

You tried to link with more than 16 libraries. Combine libraries or link modules that require fewer libraries.

Too many overlays

Your program defines more than 63 overlays. Reduce the number of overlays.

Too many segments

Your program has too many segments. Relink using the /SEGMENTS switch with an appropriate number of segments specified.

Too many segments in one module

Your object module has more than 255 segments. Split the modules or combine segments.

Unexpected end-of-file on library

The disk containing the library has probably been removed. Try again after inserting the disk that contains the library.

Unexpected end-of-file on scratch file

The disk containing VM.TMP was removed. Replace it and restart the linker.

Unmatched left parenthesis

You have made a typing mistake while specifying the contents of an overlay on the command line.

Unmatched right parenthesis

You have made a typing mistake while specifying the contents of an overlay on the command line.

Unrecognized switch error: '*filename*'

You entered an unrecognized character after the switch indicator '/', such as:

```
A>LINK /ABCDEF main;
```

LINK will abort.

VM.TMP is an illegal file name and has been ignored

You have used VM.TMP as an object file name. Rename the file and link again.

Warning: no stack segment

Your program contains no segment of combine-type stack.

Warning: too many local symbols

You have asked for a sorted listing of local symbols in the list file, but there are too many symbols to sort. The linker will produce an unsorted listing of the local symbols.

Warning: too many public symbols

You have asked for a sorted listing of public symbols in the list file, but there are too many symbols to sort. The linker will produce an unsorted listing of the public symbols.

F.7 EXEPACK Error Messages

filename : No such file or directory

The given file can't be found.

filename: Permission denied

You tried to use the EXEPACK utility on a read-only file.

can't change load-high program

When the minimum allocation value and the maximum allocation value are both zero, the file cannot be compressed.

error reading relocation table

The file cannot be compressed because the relocation table cannot be found or is invalid.

invalid .EXE file (actual length < reported)

The second and third fields in the header indicated a file size greater than the actual size.

invalid .EXE format (bad header)

The given file is not an executable file or has an invalid file header.

out of memory

There is not enough memory for the EXEPACK utility to operate.

too many segments in relocation table

The given file is too large to be compressed in the available system memory.

usage: exepack infile [outfile]

You mistyped the command.

You may also encounter MS-DOS error messages if the EXEPACK program cannot read, write to, or create a file.

F.8 EXEMOD Error Messages

***filename*: No such file or directory**

The given file cannot be found.

***filename*: Permission denied**

The given file is read-only.

can't change load-high program

When the minimum allocation value and the maximum allocation value are both zero, the file cannot be modified.

file not .EXE:*filename*

EXEMOD automatically appends the .EXE extension to any filename without an extension. In this case, no file with the given name and an .EXE extension could be found.

invalid .EXE file (actual length < reported)

The second and third fields in the file header indicate a file size greater than the actual size.

invalid .EXE file (bad header)

Your executable file is not in the correct format.

min > max (correcting max)

If the minimum allocation value is greater than the maximum allocation value, the maximum allocation value is adjusted. (Note: this is a warning message only; the modification is still performed.)

min > stack (correcting min)

If the minimum allocation value is not enough to accommodate the stack (either the original stack request or the modified request), the minimum allocation value is adjusted. (Note: this is a warning message only; the modification is still performed.)

usage: exemod file [-/h] [-/stack *n*] [-/max *n*] [-/min *n*]

You mistyped the command.

The EXEMOD utility also produces error messages when the file header is not in recognizable ".EXE" format, or if errors occur in reading or writing to a file.

Index

Microsoft® LIB

Library Manager

Reference Manual

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 this software on magnetic tape, disk, or any other medium for any purpose other than the purchaser's personal use.

If you have comments about the software or this manual, complete the Software Problem Report at the back of this manual and return it to Microsoft Corporation.

Document Number 8440-300-00

Contents

Chapter 1
Introduction

1.1 About This Manual

This manual describes the Microsoft® LIB Library Manager. This utility enables you to create and maintain your own libraries of useful functions. You can use these libraries to customize the runtime support available to your programs.

Notational Conventions

The following notational conventions are used throughout this manual:

italics

Italics mark the places in command line and option specifications and in the text where specific terms appear in an actual command. For example, in

/W *number*

number is italicized to indicate that this is a general form for the /W option. In an actual command, the user supplies a particular number for the placeholder *number*.

Italics are also used when referring to specific identifiers supplied for functions, variables, types, and labels.

Occasionally, italics are used to emphasize particular words in the text.

brackets [ ]

Brackets enclose optional fields in command line and option specifications. For example, in

/D *identifier*[=[*string*]]

the brackets around the phrase "=[*string*]" indicate that you are not

required to supply this phrase when you use the /D option. Furthermore, within this phrase, *string* is enclosed in brackets. Thus, when you give an equal sign (=), the *string* is optional. Notice, however, that you may not give a *string* without first giving the equal sign.

ellipses...

Ellipses following an item indicate that more items having the same form may appear. For example, in

```
LIB library [/pagesize] operations...
```

the ellipses indicate that one or more *operations* are allowed.

CAPITALS

Capital letters are used for the names of files, directories, environment variables, and manifest constants and macros. Commands typed at the MS™-DOS level are also capitalized.

SMALL CAPITALS

Small capital letters are used for the names of keys and key sequences, such as RETURN and CONTROL-C.

"Quotation marks"

Quotation marks set off terms defined in the text. For example, the term "module" appears in quotation marks the first time it is defined.

Quotation marks are also used to set off program fragments and to refer to command line prompts.

```
programming
examples
```

Programming examples are displayed without proportional spacing so that they look like the programs you create with a text editor.

1.2 Managing Libraries

The Microsoft LIB Library Manager is a utility designed to help you create, organize, and maintain runtime libraries. Runtime libraries are collections of compiled or assembled functions that provide a common set of useful routines. Any program can call a runtime routine, exactly as if the function were included in the program. The program is linked with the runtime library file and the call to the runtime routine is resolved by finding the routine in the library file.

Runtime libraries are created by combining separately compiled object files into one library file. Library files are usually identified by their ".LIB" extension, although other extensions are allowed.

In addition to accepting MS-DOS™ object files and library files, Microsoft LIB accepts 286 XENIX™ archives and INTEL®-style libraries. You can add the contents of a 286 XENIX archive or an INTEL-style library to an MS-DOS library by using the append operator (+).

Once an object file is incorporated into a library, it becomes an object "module." LIB makes a distinction between object files and object modules: an object "file" exists as an independent file while an object "module" is part of a larger library file. An object file can have a full pathname, including a drive designation, directory pathname, and filename extension (usually ".OBJ"). Object modules have just a name. For example, "B:\RUN\SORT.OBJ" is an object filename, while "SORT" is the corresponding object module name.

Using LIB, you can create a new library file, add object files to an existing library, delete library modules, replace library modules, and create object files from library modules. LIB also lets you combine the contents of two libraries into one library file.

The command syntax is straightforward, and LIB prompts you for responses. Once you have learned how LIB works and what its prompts mean, you can use one of the alternate methods of invoking LIB, described in Sections 2.5, "Using the Command Line," and 2.6, "Using a Response File." These alternatives let you give LIB commands without waiting for the LIB prompts.

1.3 Overview of LIB Operation

You can perform a number of library management functions with Microsoft LIB. LIB can

- Create a library file
- Delete modules
- Extract a module and place it in a separate object file
- Extract a module and delete it
- Append an object file as a module of a library, or append the contents of a library
- Replace a module in the library file with a new module
- Produce a listing of all public symbols in the library modules

For each library session, LIB first reads and interprets the user's commands. It determines whether a new library is being created or an existing library is being examined or modified.

Deletion and extraction commands (if any) are the first commands processed. LIB does not actually delete modules from the existing file. Instead, it marks the selected modules for deletion, creates a new library file, and copies only the modules *not* marked for deletion into the new library file.

Next, LIB processes any addition commands. Like deletions, additions are not performed on the original library file. Instead, the additional modules are appended to the new library file. (If there were no deletion or extraction commands, a new library file is created in the addition stage by copying the original library file.)

As LIB carries out these commands, it reads the object modules in the library, checks them for validity, and gathers the information necessary to build a library index and a listing file. The library index is used by the linker to search the library.

The listing file contains a list of all public symbols in the index and the names of the modules in which they are defined. LIB produces the listing file only if you ask for it during the library session.

LIB never makes changes to the original library; it copies the library and makes changes to the copy. Thus, when you terminate LIB for any reason, you do not lose your original file. It also means that when you run LIB, enough space must be available on your disk for both the original library file and the copy.

When you modify a library file, LIB gives you the option of specifying a different name for the file containing the modifications. If you use this option, the modified library is stored under the name you give, and the original, unmodified version is preserved under its own name. If you choose not to give a new name, LIB gives the modified file the original library name, but keeps a backup copy of the original library file. This copy has the extension ".BAK" instead of ".LIB".

Example

```
LIB LANG HEAP+HEAP;
```

This command first deletes the library module HEAP from the library file LANG.LIB, then adds the file HEAP.OBJ as the last module in the library. The same command could be given as

```
LIB LANG+HEAP HEAP;
```

The effect is the same because delete operations are always carried out before append operations, regardless of the order of the operations in the command line. This order of execution prevents confusion in LIB when a new version of a module replaces an old version in the library file.

After the library is modified, the modified file is written back to LANG.LIB. A copy of the original LANG.LIB is stored in LANG.BAK.

Chapter 2
Running LIB

LIB requires two types of input: a command to start LIB and responses to command prompts. Start LIB at the MS-DOS command level by typing

```
LIB
```

LIB prompts you for the input it needs by displaying the following four messages, one at a time. LIB waits for you to respond to each prompt, then prints the next prompt.

```
Library name:
Operations:
List file:
Output library:
```

The responses you can make to each prompt are described in Sections 2.1 through 2.4.

Once you understand the LIB prompts and operations, you may want to use one of the two alternate methods of running LIB. The command line method lets you type all commands, options, and filenames on the line used to start LIB. With the response file method, you create a file that contains all the necessar commands, then tell LIB where to find that file.

Both of the alternate methods require that you understand how LIB works and what your responses to its prompts mean. For this reason it is recommended that you allow LIB to prompt you for responses until you are comfortable with its commands and operations.

2.1 Library Name Prompt

At the "Library name" prompt, give the name of the library file you want. Usually library files are named with the ".LIB" extension. You can omit the ".LIB" extension when you give the library filename since LIB assumes that the filename extension is ".LIB".

If your library file does not have the ".LIB" extension, be sure to include the extension when you give the library filename. Otherwise, LIB cannot find the file.

Pathnames are allowed with the library filename, so you can give LIB the pathname of a library file in another directory or on another disk.

Because LIB manages only one library file at a time, only one filename is allowed in response to this prompt. There is no default response, so LIB produces an error message if you do not give a filename.

If you give the name of a library file that does not exist, LIB displays the prompt

```
Library file does not exist.  Create?
```

Type "y" (yes) to create the library file. If you answer "n" (no), LIB returns control to the MS-DOS level.

If you type a library filename and follow it immediately with a semicolon (;), LIB performs only a consistency check on the given library. A consistency check tells you whether all the modules in the library are in usable form. LIB prints a message only if it finds an invalid object module; no message appears if all modules are intact.

You can also set the library page size following this prompt. See Section 3.9, "Setting the Library Page Size," for details.

2.2 Operations Prompt

Following the "Operations" prompt, you can type one of the command symbols for manipulating modules (+, −, −+, *, −*) followed immediately (no space) by the module name or the object filename. You can specify more than one operation following this prompt, in any order. The default for the "Operations" prompt is no changes.

When you have a large number of modules or files to manipulate (more than can be typed on one line), type an ampersand (&) as the last symbol on the line, immediately before the carriage return. The ampersand must follow a filename; you cannot give an operator as the last character on a line to be continued. The ampersand causes LIB to repeat the "Operations" prompt, allowing you to specify more operations and names.

The following list describes the command symbols and their meanings and uses:

Command Symbol	Meaning and Use
+	The plus sign appends an object file as the last module in the library file. Give the name of the object file immediately after the plus sign. You can use pathnames for the object file. LIB automatically supplies the ".OBJ" extension, so you can omit the extension from the object filename. You can also use the plus sign to combine two libraries. When you give a library name after the plus sign, a copy of the contents of the given library is added to the library file being modified. You must include the ".LIB" extension when you give a library filename. Otherwise, LIB uses the default ".OBJ" extension when it looks for the file.
−	The minus sign deletes a module from the library file. Give the name of the module to be deleted immediately after the minus sign. A module name has no pathname and no extension.
− +	Type a minus sign followed by a plus sign to replace a module in the library. Give the name of the module to be replaced after the replacement symbol. Module names have no pathnames and no extensions. To replace a module, LIB first deletes the given module, then appends the object file

having the same name as the module. The object file is assumed to have an ".OBJ" extension and to reside in the current working directory.

* Type an asterisk followed by a module name to copy a module from the library file into an object file of the same name. The module remains in the library file. When LIB copies the module to an object file, it adds the ".OBJ" extension and the drive designation and pathname of the current working directory to the module name to form a complete object filename. You cannot override the ".OBJ" extension, drive designation, or pathname given to the object file, but you can later rename the file or copy it to whatever location you like.

\- * Use the minus sign followed by an asterisk to move an object module from the library file to an object file. This operation is equivalent to copying the module to an object file, as described above, then deleting the module from the library.

2.3 List File Prompt

After the "List file" prompt, you can give a filename for a cross-reference listing file. You can specify a full pathname for the listing file to cause it to be created outside your current working directory. You can give the listing file any name and any extension. LIB does not supply a default extension if you omit the extension.

A cross-reference listing file contains two lists. The first is an alphabetical listing of all external (public) symbols in the library. Each symbol name is followed by the name of the module in which it is referenced.

The second list is a list of the modules in the library. Under each module name is an alphabetical listing of the public

symbols referenced in that module. The default when you omit the response to this prompt is the special filename NUL., which tells LIB *not* to create a listing file.

2.4 Output Library Prompt

After the "Output library" prompt, you can give the name of a new library file to create with the specified modifications. The default is the current library filename; the original, unmodified library is saved in a library file with the same name but with a ".BAK" extension replacing the ".LIB" extension. This prompt appears only if you specify modifications to the library following the "Operations" prompt.

2.5 Using the Command Line

The command line method of starting LIB has the form

LIB *library* [*/pagesize*] [;] *operations*. . . [, *listing*, *newlibrary*]

The entries following LIB are responses to the LIB command prompts.

The *library* entry, with the optional */pagesize* specification, corresponds to the "Library name" prompt. If you want LIB to perform a consistency check on the library, follow the *library* entry with a semicolon (;).

The *operations* entries are any of the operations allowed following the "Operations" prompt. The *listing* entry, if you include it, tells LIB to create a listing file with the given name. The *newlibrary* entry, if it appears, is the name of the revised library.

If you want to create a cross-reference listing, the name of the listing file must be separated from the last *operations* entry by a comma. If you give a filename in the new library field, the library name must be separated from the listing filename or the last *operations* entry by a comma.

You can use a semicolon after any entry but the first to tell LIB to use the default responses for the remaining entries. The semicolon should be the last character on the command line.

Examples

```
1.   LIB LANG +HEAP;

2.   LIB C;

3.   LIB LANG,LCROSS.PUB
```

The first command instructs LIB to replace the HEAP module in the library LANG.LIB. LIB first deletes the HEAP module in the library, then appends the object file HEAP.OBJ as a new module in the library. The semicolon command symbol at the end of the command line tells LIB to use the default responses for the remaining prompts. This means that no listing file is created and that the changes are written back to the original library file instead of creating a new library file.

The second command causes LIB to perform a consistency check of the library file C.LIB. No other action is performed. If LIB displays any consistency errors it finds and returns to the operating system level. The third command tells LIB to perform a consistency check of the library file LANG.LIB, then output a cross-reference listing file named LCROSS.PUB.

2.6 Using a Response File

The command to start LIB with a response file has the form

LIB @*response-file*

The *response-file* field is the name of a response file. The *response-file* name may be qualified with a drive and directory specification to name a response file from a directory other than the current working directory.

Before you use this method you must set up a response file containing answers to the LIB prompts. This method lets you conduct the library session without typing responses at the keyboard.

A response file has one text line for each prompt. Responses must appear in the same order as the command prompts appear. Use command symbols in the response file the same way you would for responses typed on the keyboard.

When you run LIB with a response file, the prompts are displayed along with the responses from the response file. If the response file does not contain answers for all the prompts, LIB uses the default responses.

Example

```
SLIBC
+CURSOR+HEAP HEAP*FOIBLES
CROSSLST
```

This response file causes LIB to delete the module HEAP from the SLIBC.LIB library file; extract the module FOIBLES and place it in an object file named FOIBLES.OBJ; and then append the object files CURSOR.OBJ and HEAP.OBJ as the last two modules in the library. Finally, LIB creates a cross-reference file named CROSSLST.

2.7 Extending Lines

If you have many operations to perform during a library session, use the ampersand (&) command symbol to extend the operations line. Give the ampersand symbol after an object module or object filename; do not put the ampersand between an operations symbol and a name.

If you use the ampersand with the prompt method of invoking LIB, the ampersand will cause the "Operations" prompt to be repeated, allowing you to type in more operations. With the response file method, you can use the ampersand at the end of a line and then continue typing operations on the next line.

2.8 Terminating the Library Session

At any time, you can use CONTROL-C to terminate a library session. If you type an incorrect response, such as a wrong or incorrectly spelled filename or module name, you must enter CONTROL-C to exit LIB. You can then restart the program.

2.9 Selecting Default Responses to Prompts

After any entry but the first, use a single semicolon (;) followed immediately by a carriage return to select default responses to the remaining prompts. You can use the semicolon command symbol with the command line and response file methods of invoking LIB, but it is not really necessary, since LIB supplies the default responses wherever you omit responses.

The default response for the "Operations" prompt is no operation. The library file is unchanged.

The default response for the "List file" prompt is the special filename NUL., which tells LIB not to create a listing file.

The default response for the "Output library" file is the current library name. This prompt appears only if you specify at least one operation following the "Operations" prompt.

Chapter 3
Library Tasks

This section summarizes the library management tasks you can perform with LIB.

3.1 Creating a Library File

To create a new library file, simply give the name of the library file you want to create following the "Library name" prompt. LIB supplies the ".LIB" extension.

The name of the new library must not be the name of an existing file, or else LIB will assume you want to modify the existing file. When you give the name of a library file that does not currently exist, LIB displays the following prompt.

```
Library file does not exist.  Create?
```

Type "yes" to create the file, "no" to terminate the library session.

You can specify a page size for the library when you create it. The default page size is 16 bytes. See Section 3.9, "Setting the Library Page Size," for a discussion of this option.

Once you have given the name of the new library file, you can insert object modules into the library by using the add operation (+) following the "Operations" prompt. You can also add the contents of another library if you wish. These options are discussed in Sections 3.3, "Adding Library Modules," and 3.6, "Combining Libraries."

3.2 Modifying a Library File

You can modify an existing library file by giving the name of the library file following the "Library name" prompt. All operations you specify following the "Operations" prompt are performed on that library.

However, LIB lets you keep both the unmodified library file and the newly modified version, if you like. You can do this by giving the name of a new library file following the "Output library" prompt. The modified library file is stored under the new library filename, while the original library file is preserved unchanged.

If you don't give a filename following the "Output library" prompt, the modified version of the library file replaces the original library file. Even in this case, LIB saves the original, unmodified library file. The unmodified library file has the extension ".BAK" instead of ".LIB". Thus, at the end of the session you have two library files: the modified version with the ".LIB" extension and the original, unmodified version with the ".BAK" extension.

3.3 Adding Library Modules

Use the plus sign (+) following the "Operations" prompt to add an object module to a library. Give the name of the object file to be added, without the ".OBJ" extension, immediately after the plus sign.

LIB strips the drive designation and the extension from the object file specification, leaving only the basename. This becomes the name of the object module in the library. For example, if the object file B:\CURSOR.OBJ is added to a library file, the name of the corresponding object module is "CURSOR".

Object modules are always added to the end of a library file.

3.4 Deleting Library Modules

Use the minus sign (−) following the "Operations" prompt to delete an object module from a library. Give the name of the module to be deleted immediately after the minus sign. A module name has no pathname and no extension; it is simply a name, like "CURSOR".

Replacing Library Modules

Use a minus sign followed by a plus sign (− +) to replace a module in the library. Give the name of the module to be replaced after the replacement symbol (− +). Remember that module names have no pathnames and no extensions.

To replace a module, LIB first deletes the given module, then appends the object file having the same name as the module. The object file is assumed to have an ".OBJ" extension and to reside in the current working directory.

3.5 Extracting Library Modules

Use an asterisk (*) followed by a module name to copy a module from the library file into an object file of the same name. The module remains in the library file. When LIB copies the module to an object file, it adds the ".OBJ" extension and the drive designation and pathname of the current working directory to the module name to form a complete object filename. You cannot override the ".OBJ" extension, drive designation, or pathname given to the object file, but you can later rename the file or copy it to whatever location you like.

Use the minus sign followed by an asterisk (− *) to move an object module from the library file to an object file. This operation is equivalent to copying the module to an object file, as described above, then deleting the module from the library.

3.6 Combining Libraries

You can add the contents of one library to another by using the plus sign (+) with a library filename instead of an object filename. Following the "Operations" prompt, give the plus sign (+) followed by the name of the library whose contents you wish to add to the library being modified. When you use this option you must include the ".LIB" extension of the library filename. Otherwise, LIB assumes that the file is an object file and looks for the file with an ".OBJ" extension.

In addition to allowing MS-DOS libraries as input, LIB also accepts 286 XENIX archives and Intel-format libraries. Thus, you can use LIB to convert libraries from either of these formats to the Microsoft format.

LIB adds the modules of the library to the end of the library being modified. Notice that the added library still exists as an independent library. LIB copies the modules without deleting them.

Once you have added the contents of a library or libraries, you can save the new, combined library under a new name by giving a new name following the "Output library" prompt. If you omit the "Output library" response, LIB saves the combined library under the name of the original library being modified.

3.7 Creating a Cross-Reference Listing

Create a cross-reference listing by giving a name for the listing file following the "List file" prompt. If you omit the response to this prompt, LIB uses the special filename NUL., which means that no listing file is created.

You can give the listing file any name and any extension. You can specify a full pathname, including drive designation, for the listing file to cause it to be created outside your current working directory. LIB does not supply a default extension if you omit the extension.

A cross-reference listing file contains two lists. The first is an alphabetical listing of all public symbols in the library. Each symbol name is followed by the name of the module in which it is referenced.

The second list is an alphabetical list of the modules in the library. Under each module name is an alphabetical listing of the public symbols referenced in that module.

3.8 Performing Consistency Checks

When you give just a library name followed by a semicolon following the "Library name" prompt, LIB performs a consistency check, displaying messages about any errors it finds. No changes are made to the library. This option is not usually necessary, since LIB automatically checks object files for consistency before adding them to the library.

To produce a cross-reference listing along with a consistency check, use the command line method of invoking LIB. Give the library name followed by a semicolon, then give the name of the listing file. LIB performs the consistency check and then creates the cross-reference listing.

3.9 Setting the Library Page Size

The page size of a library affects the alignment of modules stored in the library. Modules in the library are aligned so that they always start at a position that is a multiple of *n* bytes from the beginning of the file. The value of *n* is the page size. The default page size is 16 for a new library or the current page size for an existing library.

Because of the indexing technique used by LIB, a library with a large page size can hold more modules than a library with a smaller page size. However, for each module in the library, an average of $n/2$ bytes of storage space is wasted (where *n* is the page size). In most cases, a small page size is advantageous, and you are advised to use the smallest page size possible.

To set the library page size, add a page size option after the library filename following the "Library name" prompt:

library /pagesize:*n*

The value of *n* is the new page size. It must be a power of 2 and must fall between 16 and 32,768.

3.10 LIB Error Messages

The following are Microsoft LIB error messages:

***symbol* is a multiply defined PUBLIC. Proceed?**

Cause: Two modules define the same public symbol. You are asked to confirm the removal of the definition of the old symbol.

Cure: Remove the PUBLIC declaration from one of the object modules and recompile or reassemble. If you respond No, the library will be left in an indeterminate state.

Allocate error on VM.TMP

Out of disk space

Cannot create extract file

No room in directory for extract file

Cannot create list file

No room in directory for library file

Cannot nest response file

@filespec in response (or indirect) file

Microsoft LIB cannot open VM.TMP

There is no room for VM.TMP in disk directory

Cannot write library file

Out of disk space

Close error on extract file

Out of disk space

Error: An internal error has occurred

Contact Microsoft Corporation

Fatal Error: Cannot open input file

You mistyped an object filename

Fatal Error: Module is not in the library

You tried to delete a module that is not in the library

Input file read error

Bad object module or faulty disk

Invalid object module/library

Bad object module and/or library

Library Disk is full

No more room on disk

Listing file write error

Out of disk space

No library file specified

No response to Library File: prompt

Read error on VM.TMP

Disk not ready for read

Symbol table capacity exceeded

Too many public symbols (about 30K chars in symbols)

Too many object modules

More than 500 object modules

Too many public symbols

1024 public symbols maximum

Write error on library/extract file

Out of disk space

Write error on VM.TMP

Out of disk space

Index

MICROSOFT®
10700 Northup Way, Bellevue, WA 98004

Software Problem Report

Name _____

Street _____

City _____ State _____ Zip _____

Phone _____ Date _____

Instructions

Use this form to report software bugs, documentation errors, or suggested enhancements. Mail the form to Microsoft.

Category

_____ Software Problem

_____ Software Enhancement

_____ Documentation Problem
(Document #_____)

_____ Other

Software Description

Microsoft Product _____

Rev. _____ Registration # _____

Operating System _____

Rev. _____ Supplier _____

Other Software Used _____

Rev. _____ Supplier _____

Hardware Description

Manufacturer _____ CPU _____ Memory _____ KB

Disk Size _____" Density: Sides:

Single _____ Single _____

Double _____ Double _____

Peripherals _____

Problem Description

Describe the problem. (Also describe how to reproduce it, and your diagnosis and suggested correction.) Attach a listing if available.

Microsoft Use Only

Tech Support _____ Date Received _____

Routing Code _____ Date Resolved _____

Report Number _____

Action Taken:

MICROSOFT®